**Алгоритмы**

**ТЕОРИЯ И ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ**

Численные алгоритмы Структуры данных

связными списками и сетями

Методы работы с массивами,



ROD

STEPHENS

Essential Algorithms

$ 35$&7,&$/ $3352$&+ 72 &20387(5 $/\*25,7+06

WILEY

ǜǚǐ ǝǞǔǎǑǙǝ

Алгоритмы

ǞǑǚǜǔǫ ǔ ǛǜǌǖǞǔǣǑǝǖǚǑ ǛǜǔǘǑǙǑǙǔǑ

ǘǚǝǖǎǌ

2016

УДК 004.4

ББК 32.973.26-018.2 С80

Rod Stephens

Essencial Algorithms: F Practical Approach to Computer Algorithms

© 2013 by John Wiley & Sons, Inc., Indianapolis, Indiana. All rights reserved. Authorised translation from the English language edition published by John Wiley & Sons Limited

Перевод:

*Кириленко Вадим* 1—12 главы

*Волошко Роман Владимирович* 13—19 главы

В оформлении переплета использована иллюстрация: VLADGRIN / Shutterstock.com

Используется по лицензии от Shutterstock.com

С80

**Стивенс, Род.**

Алгоритмы. Теория и практическое применение / Род Стивенс. — Москва : Издательство «Э», 2016. — 544 с. — (Мировой компьютер- ный бестселлер).

ISBN 978-5-699-81729-0

Алгоритмы – это рецепты, которые делают возможным эффективное программирование. Их изучение позволяет усвоить общие подходы к реше- нию задач и накапливать полезные методики для их решения. В этой книге представлено множество классических алгоритмов; вы узнаете, где они при- меняются и как их анализировать, чтобы понять их поведение.

Эта книга может быть полезной не только в вашей текущей профес- сиональной деятельности, но и может помочь вам получить новую работу.

**УДК 004.4**

**ББК 32.973.26-018.2**

**ISBN 978-5-699-81729-0**

**© Кириленко В., Волошко Р.В., перевод на русский язык, 2016**

**© ООО «АйдионоМикс», 2016**

**© ОфорМление. ООО «Издательство «Э», 2016**

# КРАТКОЕ ОГЛАВЛЕНИЕ

**Оглавление 6**

**Об авторе 16**

**Благодарности 16**

**Введение 17**

**Глава 1. Основы алгоритмизации 24**

**Глава 2. Численные алгоритмы 45**

**Глава 3. Связные списки 72**

**Глава 4. Массивы 96**

**Глава 5. Стеки и очереди 119**

**Глава 6. Сортировка 136**

**Глава 7. Поиск 163**

**Глава 8. Хеш-таблицы 168**

**Глава 9. Рекурсия 181**

**Глава 10. Деревья 215**

**Глава 11. Сбалансированные деревья 257**

**Глава 12. Деревья принятия решений 274**

**Глава 13. Основные сетевые алгоритмы 296**

**Глава 14. Дополнительные сетевые алгоритмы 324**

**Глава 15. Строковые алгоритмы 345**

**Глава 16. Криптография 365**

**Глава 17. Теория вычислительной сложности 387**

**Глава 18. Распределенные алгоритмы 402**

**Глава 19. Головоломки, встречающиеся на собеседованиях 432**

**Приложение А 442**

**Приложение Б 453**

**Глоссарий 522**

**Указатель 536**

**ОГЛАВЛЕНИЕ**

**Об авторе 16**

**Благодарности 16**

**Введение 17**

Выбор алгоритма 18

Для кого предназначена книга 19

Как извлечь наибольшую пользу из книги 19

Сайты с материалами книги 20

Структура книги 20

Что нужно для работы с книгой 22

Условные обозначения 22

Обратная связь 23

**Глава 1. Основы алгоритмизации 24**

Метод 24

Алгоритм и структура данных. 25

Псевдокод 25

Свойства алгоритма 28

Асимптотическая сложность алгоритма 29

Обычные функции рабочего цикла 33

Визуализация функций 38

Практические рекомендации 39

Резюме 41

Упражнения 41

[Глава 2. Численные алгоритмы 45](#_TOC_250291)

[Рандомизация данных 45](#_TOC_250290)

[Генерирование случайных величин 45](#_TOC_250289)

[Рандомизация массивов 50](#_TOC_250288)

[Генерирование неравномерных распределений 52](#_TOC_250287)

Нахождение наибольшего

общего делителя 52

[Возведение в степень 54](#_TOC_250286)

[Работа с простыми числами 56](#_TOC_250285)

[Нахождение простых множителей 56](#_TOC_250284)

[Нахождение простых элементов 58](#_TOC_250283)

[Проверка на простоту 59](#_TOC_250282)

[Численное интегрирование 60](#_TOC_250281)

[Формула прямоугольников 61](#_TOC_250280)

[Формула трапеций 62](#_TOC_250279)

[Адаптивная квадратура 63](#_TOC_250278)

[Интеграция Монте-Карло 66](#_TOC_250277)

[Нахождение нулей 67](#_TOC_250276)

[Резюме 69](#_TOC_250275)

[Упражнения 70](#_TOC_250274)

[Глава 3. Связные списки 72](#_TOC_250273)

[Основные положения 72](#_TOC_250272)

[Однонаправленные связные списки 73](#_TOC_250271)

[Передвижение по спискам 73](#_TOC_250270)

[Нахождение ячеек 74](#_TOC_250269)

[Использование ограничителей 75](#_TOC_250268)

[Добавление ячеек в начало списка 76](#_TOC_250267)

[Добавление ячеек в конец списка 77](#_TOC_250266)

[Вставка ячеек 77](#_TOC_250265)

[Удаление ячеек 78](#_TOC_250264)

[Двунаправленные связные списки 79](#_TOC_250263)

[Сортированные списки 80](#_TOC_250262)

Алгоритмы для работы

со связными списками 81

[Копирование 82](#_TOC_250261)

[Сортировка вставкой 82](#_TOC_250260)

[Сортировка методом выбора 84](#_TOC_250259)

[Многопотоковые связные списки 85](#_TOC_250258)

[Связные списки с циклами 86](#_TOC_250257)

[Маркировка ячеек 87](#_TOC_250256)

[Использование хеш-таблиц 88](#_TOC_250255)

[Повторная трассировка списка 89](#_TOC_250254)

[Реверсирование списка 90](#_TOC_250253)

[Черепаха и кролик 92](#_TOC_250252)

[Циклы в двунаправленных связных списках 94](#_TOC_250251)

[Резюме 94](#_TOC_250250)

[Упражнения 94](#_TOC_250249)

[Глава 4. Массивы 96](#_TOC_250248)

[Основные положения 96](#_TOC_250247)

[Одномерные массивы 98](#_TOC_250246)

[Нахождение элементов 98](#_TOC_250245)

Нахождение минимальной,

максимальной и средней величин 98

[Вставка элементов 100](#_TOC_250244)

[Удаление элементов 101](#_TOC_250243)

[Ненулевые нижние пределы 101](#_TOC_250242)

[Двумерные массивы 101](#_TOC_250241)

[Массивы высокой размерности 102](#_TOC_250240)

[Треугольные массивы 105](#_TOC_250239)

[Массивы с разрывом 108](#_TOC_250238)

[Нахождение строки и столбца 110](#_TOC_250237)

[Получение значения 111](#_TOC_250236)

[Установка значения 111](#_TOC_250235)

[Удаление значения 113](#_TOC_250234)

[Матрицы 115](#_TOC_250233)

[Резюме 117](#_TOC_250232)

[Упражнения 117](#_TOC_250231)

[Глава 5. Стеки и очереди 119](#_TOC_250230)

[Стеки 119](#_TOC_250229)

[Стеки связных списков 120](#_TOC_250228)

[Стеки массивов 121](#_TOC_250227)

[Двойные стеки 123](#_TOC_250226)

[Алгоритмы с использованием стеков 124](#_TOC_250225)

[Очереди 129](#_TOC_250224)

[Очереди связных списков 130](#_TOC_250223)

[Очереди массивов 130](#_TOC_250222)

[Специализированные очереди 133](#_TOC_250221)

[Резюме 134](#_TOC_250220)

[Упражнения 134](#_TOC_250219)

[Глава 6. Сортировка 136](#_TOC_250218)

[Алгоритмы O(N2) 136](#_TOC_250217)

[Сортировка вставкой в массивах 136](#_TOC_250216)

[Сортировка выбором в массивах 138](#_TOC_250215)

[Пузырьковая сортировка 139](#_TOC_250214)

[Алгоритмы O(N х log N) 142](#_TOC_250213)

[Пирамидальная сортировка 142](#_TOC_250212)

[Быстрая сортировка 148](#_TOC_250211)

[Сортировка слиянием 155](#_TOC_250210)

[Алгоритмы быстрее O(N х log N) 157](#_TOC_250209)

[Сортировка подсчетом 157](#_TOC_250208)

[Блочная сортировка 159](#_TOC_250207)

[Резюме 160](#_TOC_250206)

[Упражнения 161](#_TOC_250205)

[Глава 7. Поиск 163](#_TOC_250204)

[Линейный поиск 163](#_TOC_250203)

[Бинарный поиск 164](#_TOC_250202)

[Интерполяционный поиск 165](#_TOC_250201)

[Резюме 166](#_TOC_250200)

[Упражнения 167](#_TOC_250199)

[Глава 8. Хеш-таблицы 168](#_TOC_250198)

[Основы хеш-таблиц 168](#_TOC_250197)

[Прямое связывание 170](#_TOC_250196)

[Открытая адресация 171](#_TOC_250195)

[Удаление элементов 172](#_TOC_250194)

[Линейное пробирование 173](#_TOC_250193)

[Квадратичное пробирование 174](#_TOC_250192)

[Псевдослучайное пробирование 176](#_TOC_250191)

[Двойное хеширование 176](#_TOC_250190)

[Упорядоченное хеширование 176](#_TOC_250189)

[Резюме 178](#_TOC_250188)

[Упражнения 179](#_TOC_250187)

[Глава 9. Рекурсия 181](#_TOC_250186)

[Базовые алгоритмы 181](#_TOC_250185)

[Факториал 181](#_TOC_250184)

[Числа Фибоначчи 183](#_TOC_250183)

[Ханойская башня 184](#_TOC_250182)

[Графические алгоритмы 187](#_TOC_250181)

[Кривые Коха 187](#_TOC_250180)

[Кривая Гильберта 189](#_TOC_250179)

[Кривая Серпинского 190](#_TOC_250178)

[Салфетки 193](#_TOC_250177)

[Алгоритмы с возвратом 194](#_TOC_250176)

[Задача о восьми ферзях 195](#_TOC_250175)

[Ход коня 198](#_TOC_250174)

[Сочетания и размещения 200](#_TOC_250173)

[Сочетания с циклами 201](#_TOC_250172)

[Сочетания с повторениями 202](#_TOC_250171)

[Сочетания без повторений 204](#_TOC_250170)

[Размещения с повторениями 204](#_TOC_250169)

[Размещения без повторений 205](#_TOC_250168)

[Удаление рекурсии 206](#_TOC_250167)

[Удаление хвостовой рекурсии 206](#_TOC_250166)

[Хранение промежуточных значений 208](#_TOC_250165)

[Удаление общей рекурсии 209](#_TOC_250164)

[Резюме 212](#_TOC_250163)

[Упражнения 212](#_TOC_250162)

[Глава 10. Деревья 215](#_TOC_250161)

[Терминология 215](#_TOC_250160)

[Свойства бинарного дерева 219](#_TOC_250159)

[Представление деревьев 220](#_TOC_250158)

[Общие правила построения деревьев 220](#_TOC_250157)

[Построение завершенных деревьев 223](#_TOC_250156)

[Обход дерева 223](#_TOC_250155)

[Обход в прямом порядке 224](#_TOC_250154)

[Симметричный обход 226](#_TOC_250153)

[Обход в обратном порядке 227](#_TOC_250152)

[Обход в ширину 228](#_TOC_250151)

[Время выполнения обхода 229](#_TOC_250150)

[Упорядоченные деревья 229](#_TOC_250149)

[Добавление вершин 230](#_TOC_250148)

[Поиск вершин 231](#_TOC_250147)

[Удаление вершин 232](#_TOC_250146)

[Связные деревья 235](#_TOC_250145)

[Построение связных деревьев 236](#_TOC_250144)

[Использование связных деревьев 238](#_TOC_250143)

[Специализированные алгоритмы 240](#_TOC_250142)

[Игра «Животные» 240](#_TOC_250141)

[Расчет математических выражений 241](#_TOC_250140)

[Деревья квадрантов 243](#_TOC_250139)

[Префиксные деревья 248](#_TOC_250138)

[Резюме 252](#_TOC_250137)

[Упражнения 253](#_TOC_250136)

[Глава 11. Сбалансированные деревья 257](#_TOC_250135)

[АВЛ-деревья 257](#_TOC_250134)

[Добавление значений 258](#_TOC_250133)

[Удаление значений 260](#_TOC_250132)

[2-3-деревья 261](#_TOC_250131)

[Добавление значений 262](#_TOC_250130)

[Удаление значений 264](#_TOC_250129)

[B-деревья 266](#_TOC_250128)

[Добавление значений 267](#_TOC_250127)

[Удаление значений 268](#_TOC_250126)

[Разновидности сбалансированных деревьев 270](#_TOC_250125)

[Иерархически организованные B-деревья 270](#_TOC_250124)

[B+-деревья 270](#_TOC_250123)

[Резюме 272](#_TOC_250122)

[Упражнения 272](#_TOC_250121)

[Глава 12. Деревья принятия решений 274](#_TOC_250120)

[Поиск по деревьям игры 274](#_TOC_250119)

[Минимакс 275](#_TOC_250118)

[Начальные ходы и реакции 279](#_TOC_250117)

[Эвристика дерева игры 279](#_TOC_250116)

[Поиск по деревьям принятия решений 281](#_TOC_250115)

[Задачи оптимизации 282](#_TOC_250114)

[Метод полного перебора 282](#_TOC_250113)

[Метод ветвей и границ 284](#_TOC_250112)

[Эвристика дерева принятия решений 285](#_TOC_250111)

[Другие задачи дерева принятия решений 290](#_TOC_250110)

[Резюме 294](#_TOC_250109)

[Упражнения 295](#_TOC_250108)

[Глава 13. Основные сетевые алгоритмы 296](#_TOC_250107)

[Терминология 296](#_TOC_250106)

[Разные представления сети 299](#_TOC_250105)

[Обход сети 302](#_TOC_250104)

[Обход в глубину 302](#_TOC_250103)

[Обход в ширину 304](#_TOC_250102)

[Проверка связности 305](#_TOC_250101)

[Остовные деревья 307](#_TOC_250100)

[Минимальные остовные деревья 308](#_TOC_250099)

[Поиск путей 309](#_TOC_250098)

[Поиск произвольного пути 309](#_TOC_250097)

[Поиск кратчайшего пути с помощью установки меток 310](#_TOC_250096)

[Поиск кратчайшего пути с помощью коррекции меток 313](#_TOC_250095)

[Поиск кратчайшего пути между всеми парами вершин 315](#_TOC_250094)

[Резюме 320](#_TOC_250093)

[Упражнения 320](#_TOC_250092)

[Глава 14. Дополнительные сетевые алгоритмы 324](#_TOC_250091)

[Топологическая сортировка 324](#_TOC_250090)

[Поиск циклов 327](#_TOC_250089)

[Раскрашивание карты 328](#_TOC_250088)

[Закрашивание двумя цветами 328](#_TOC_250087)

[Закрашивание тремя цветами 330](#_TOC_250086)

[Закрашивание четырьмя цветами. 331](#_TOC_250085)

[Закрашивание пятью цветами 331](#_TOC_250084)

[Другие алгоритмы закрашивания карт 335](#_TOC_250083)

[Максимальный поток 336](#_TOC_250082)

[Распределение рабочих мест 338](#_TOC_250081)

[Минимальный разрез в потоке 340](#_TOC_250080)

[Резюме 342](#_TOC_250079)

[Упражнения 343](#_TOC_250078)

[Глава 15. Строковые алгоритмы 345](#_TOC_250077)

[Парные скобки 345](#_TOC_250076)

[Вычисление арифметических выражений 347](#_TOC_250075)

[Синтаксические деревья 347](#_TOC_250074)

[Сопоставление с шаблоном 348](#_TOC_250073)

[Детерминированные конечные автоматы 349](#_TOC_250072)

[Построение ДКА для регулярных выражений 351](#_TOC_250071)

[Недетерминированные конечные автоматы 354](#_TOC_250070)

[Поиск строк. 355](#_TOC_250069)

[Вычисление редакционного расстояния 359](#_TOC_250068)

[Резюме 361](#_TOC_250067)

[Упражнения 362](#_TOC_250066)

[Глава 16. Криптография 365](#_TOC_250065)

[Терминология 366](#_TOC_250064)

[Перестановочные шифры 367](#_TOC_250063)

[Перестановка строк/столбцов 367](#_TOC_250062)

[Перестановка столбцов 369](#_TOC_250061)

[Маршрутные шифры 371](#_TOC_250060)

[Шифры подстановки 372](#_TOC_250059)

[Шифр Цезаря 372](#_TOC_250058)

[Шифр Виженера 373](#_TOC_250057)

[Простая подстановка 375](#_TOC_250056)

[Схема одноразовых блокнотов 375](#_TOC_250055)

[Блочные шифры 376](#_TOC_250054)

[Подстановочно-перестановочные сети 377](#_TOC_250053)

[Шифр Фейстеля 378](#_TOC_250052)

[Шифрование с открытым ключом и RSA 380](#_TOC_250051)

[Функция Эйлера 381](#_TOC_250050)

[Обратные величины 381](#_TOC_250049)

[Пример использования RSA 382](#_TOC_250048)

[Практические соображения 383](#_TOC_250047)

[Другие области применения криптографии 383](#_TOC_250046)

[Резюме 384](#_TOC_250045)

[Упражнения 385](#_TOC_250044)

[Глава 17. Теория вычислительной сложности 387](#_TOC_250043)

[Обозначения 388](#_TOC_250042)

[Классы сложности 388](#_TOC_250041)

[Сведение 391](#_TOC_250040)

[3SAT 393](#_TOC_250039)

[Паросочетание в двудольном графе 393](#_TOC_250038)

[NP-сложность 394](#_TOC_250037)

[Задачи обнаружения, сообщения и оптимизации 394](#_TOC_250036)

[Обнаружение p Сообщение 395](#_TOC_250035)

[Обнаружение p Оптимизация 395](#_TOC_250034)

[Сообщение p Обнаружение 396](#_TOC_250033)

[Оптимизация p Сообщение 396](#_TOC_250032)

[NP-полные задачи 397](#_TOC_250031)

[Резюме 399](#_TOC_250030)

[Упражнения 400](#_TOC_250029)

[Глава 18. Распределенные алгоритмы 402](#_TOC_250028)

[Виды параллелизма 402](#_TOC_250027)

[Систолические массивы 403](#_TOC_250026)

[Распределенные вычисления 405](#_TOC_250025)

[Многопроцессорные вычисления 407](#_TOC_250024)

[Состояние гонки 407](#_TOC_250023)

[Взаимная блокировка 411](#_TOC_250022)

[Квантовые вычисления 412](#_TOC_250021)

[Распределенные алгоритмы 413](#_TOC_250020)

[Отладка распределенных алгоритмов 413](#_TOC_250019)

[Чрезвычайно параллельные алгоритмы 414](#_TOC_250018)

[Сортировка слиянием 416](#_TOC_250017)

[Задача обедающих философов 416](#_TOC_250016)

[Задача двух генералов 419](#_TOC_250015)

[Задача византийских генералов 420](#_TOC_250014)

[Согласование 423](#_TOC_250013)

[Выбор лидера 426](#_TOC_250012)

[Снимок 427](#_TOC_250011)

[Синхронизация часов 428](#_TOC_250010)

[Резюме 429](#_TOC_250009)

[Упражнения 429](#_TOC_250008)

[Глава 19. Головоломки, встречающиеся на собеседованиях 432](#_TOC_250007)

[Как задавать вопросы с подвохом 433](#_TOC_250006)

[Как отвечать на вопросы с подвохом 435](#_TOC_250005)

[Резюме 439](#_TOC_250004)

[Упражнения 440](#_TOC_250003)

Приложение А. Собрание алгоритмических понятий 442

[Приложение Б. Решения к упражнениям 453](#_TOC_250002)

[Глоссарий 522](#_TOC_250001)

[Алфавитный указатель 536](#_TOC_250000)

**ОБ АВТОРЕ**

Род Стивенс начинал свою карьеру как математик, однако во время учебы в Массачусетском технологическом институте он всерьез увлекся алгоритмами, прошел все связанные с ними курсы и занялся написанием довольно сложных программ. Ему довелось применить свои знания об алгоритмах во многих сферах: коммутация телефонов, учет вызовов и времени разговора абонента, оперативное управление ремонтными работами, налогообложение, водоочистка, картография и даже подготовка профессиональных футболистов.

Кроме того, Род Стивенс — один из ведущих специалистов по Microsoft Visual Basic и преподаватель вводного курса программирования в технологическом ин- ституте ITT, а также автор более 20 книг, переведенных на несколько иностран- ных языков, и свыше 250 журнальных статей о C#, Visual Basic, Visual Basic for Applications, Delphi и Java.

У Стивенса есть несколько сайтов: VB Helper (vb-helper.com)1, куда ежемесячно заходят несколько миллионов пользователей и где содержатся подсказки, секре- ты, а также примеры программ на Visual Basic, а также C# Helper (csharphelper.com) с аналогичного рода материалами для программистов на C#. Связаться с ним мож- но по электронному адресу [RodStephens@vb-helper.com](mailto:RodStephens@vb-helper.com) или [RodStephens@csharphelper.com.](mailto:RodStephens@csharphelper.com)

# БЛАГОДАРНОСТИ

Автор выражает свое признание Бобу Элиоту, Тому Динсу, Гейлу Джонсону и Дэниэлу Скрибнеру за те усилия, которые они приложили, чтобы эта книга вышла в свет. Большое спасибо хочется сказать техническим редакторам Джорджу Кокуру, Дэйву Колмену и Джеку Жинхиу Хао за точность изложенной здесь информации. В любых ошибках, которые могли остаться в книге, виноват я сам, а не эти люди.

1Все указанные в книге сайты англоязычные. Издательство не несет ответственности за их содержимое и напоминает, что со времени написания книги сайты могли измениться или вовсе исчезнуть. — *Примеч. ред.*

# ВВЕДЕНИЕ

Алгоритмы — это наборы команд, способствующие эффективному программи- рованию. Они объясняют, как сортировать записи, искать элементы, рассчитывать числовые значения (например, простые множители), находить кратчайший путь между двумя точками на карте, определять максимально возможный поток ин- формации по сети и т. д. Хороший алгоритм способен решить задачу в считаные секунды, плохой потребует на это часы или же не решит ее вовсе.

Изучение алгоритмов поможет вам овладеть различными полезными метода- ми и научиться выбирать те из них, что наилучшим образом подходят к вашему сценарию. Ведь для решения какой-либо задачи можно использовать разную методику, но только определенная окажется идеальной. Кроме того, действия, показавшие отличные результаты с одним набором данных, с другим могут прос- то не сработать.

И, что еще более важно, вы познакомитесь с некоторыми общими способами решения тех или иных задач. И даже если приведенный алгоритм не сможет быть полезен в конкретном случае, это позволит по-другому взглянуть на стоящую перед вами проблему, а также даст возможность проанализировать собственные алгоритмы и найти неожиданный способ их применения.

Изученные методы будут нелишними и в вашей профессиональной деятельно- сти. Многие крупные компании, занимающиеся информационными технологиями, такие как Microsoft, Google, Yahoo!, IBM и др., хотят, чтобы их программисты по- нимали алгоритмы и связанные с ними способы решения задач. На собеседовании в подобных корпорациях вам вполне могут предложить похожее тестовое задание. Даже если у вас не получится его выполнить, огромное значение будет иметь то, как вы попытаетесь получить результат и как станете решать незнакомую проблему. Не исключено, что в процессе рассуждений вам откроется, какие алгоритмические подходы следует использовать по отношению к задаче. «Я не знаю... Может быть, поищу что-то похожее в Интернете» — плохой ответ. А вот пример хорошего ответа:

«Возможно, здесь мог бы помочь рекурсивный метод разобщения».

Данная книга — своеобразная инструкция, изложенная простым языком. В ней рассматривается множество важнейших классических алгоритмов; рассказывается, в каких случаях и какие из них стоит использовать; объясняется, на каких прин- ципах строится их работа; сообщается о методах, которые можно применять для поиска самостоятельных решений.

Вот часть той полезной информации, которая приводится в этом издании:

* численные алгоритмы (рандомизация, разложение на множители, работа с простыми числами, численное интегрирование);
* методы работы с общими структурами данных (массивами, связными спи- сками, деревьями, сетями);
* использование более совершенных структур данных (неупорядоченных мас- сивов, сбалансированных деревьев, *B*-деревьев);
* сортировка и поиск;
* сетевые алгоритмы (кратчайший путь, связующее дерево, топологическая сортировка, расчеты потока).

Ниже перечислены некоторые из общих методов решения задач, объяснение которых вы найдете на страницах книги:

* поиск с помощью прямого подбора;
* метод «разделяй и властвуй»;
* перебор с возвратами;
* рекурсия;
* метод ветвей и границ;
* жадные алгоритмы и поиск экстремума;
* алгоритмы сочетания ресурсов;
* сжимание пределов;
* эвристика.

Для более легкого усвоения материала в конце каждой главы приводятся упраж- нения, позволяющие изменить алгоритм или использовать его в новых ситуациях. Они также помогут закрепить продемонстрированные методы.

Наконец, в книге содержатся некоторые советы, которые окажутся полезны- ми на собеседовании. Методы алгоритмизации помогут решить многие тестовые задания. И даже если вы не справитесь с какой-то конкретной головоломкой, то сможете доказать, что знакомы со способами решения других задач.

## Выбор алгоритма

Каждый алгоритм включен в книгу по нескольким причинам.

* Он полезен, поэтому опытный программист должен понимать принцип его работы и уметь использовать в программах.
* Он демонстрирует важные технологии алгоритмического программирова- ния, которые можно применить к другим задачам.
* Он предлагается для изучения студентам ИТ-специальностей и может встре- титься на техническом собеседовании.

Ознакомившись с предлагаемым материалом и поработав над упражнениями, вы будете располагать достаточной базой знаний по алгоритмам и методам, кото- рые используются для решения многих задач программирования.

## Для кого предназначена книга

Это пособие в основном рассчитано на три категории читателей: профессио- нальных программистов, тех, кто только готовится к собеседованию при устрой- стве на работу, и студентов.

Первой категории будут полезны алгоритмы и методы решения, связанные непосредственно с рабочими задачами. Даже если вы столкнетесь с проблемой, которая здесь не описана, полученные знания позволят взглянуть на нее по-новому.

Вторая категория читателей сможет отточить навыки алгоритмизации. Есть вероятность, что на собеседовании приведенные примеры вам не попадутся, но рассматриваемые методики вполне пригодятся для решения похожих вопросов.

Третьей категории следует помнить, что принципы, описанные в книге, явля- ются простыми и эффективными. Не с каждым из них вам придется столкнуться в реальной жизни, но без таких, как, например, рекурсия, «разделяй и властвуй», метод ветвей и границ, программисту просто не обойтись.

**ЗАМЕЧАНИЕ**

Лично я считаю, что алгоритмы – сплошное удовольствие! Они заменяют мне кроссворды и судоку. Я обожаю то чувство, когда сложный алгоритм состав- лен, в него введены данные и видно, как появляется кривая, построенная по множеству найденных точек, или какая-нибудь трехмерная фигура.

## Как извлечь наибольшую пользу из книги

Конечно, вы можете просто читать это пособие, знакомясь с предлагаемыми алгоритмами и методами, но, чтобы действительно хорошо овладеть материалом, вам стоит применить все на практике, то есть попытаться использовать полученные знания в каком-либо языке программирования. Не ленитесь экспериментировать — изменять алгоритмы, находить новые варианты для решения старых задач. Не остав- ляйте без внимания приведенные упражнения и вопросы для собеседований — они могут натолкнуть вас на необычное использование уже знакомого метода.

Кроме того, я настоятельно рекомендую проработать как можно большее чис- ло алгоритмов и желательно на разных языках программирования. Это позволит увидеть, как разные языки влияют на решение одной и той же задачи. Тщательным образом изучайте упражнения и анализируйте принципы их решения, в идеале вы должны уделить внимание каждому из них. Часто важность включенного в книгу алгоритма становится понятной только после того, как с его помощью вам удастся решить конкретную задачу.

Еще один полезный совет. Поищите в Интернете задачи, предлагаемые на собеседованиях, и подумайте, как бы вы с ними справились. Возможно, вам не предложат что-то подобное, но иметь хотя бы общее представление о решении не помешает. Время, затраченное на практику, не пропадет зря — так вы укрепи- те свои знания.

Изучить алгоритмы можно только в действии, поэтому не бойтесь отложить книгу, открыть компилятор и написать код!

## Сайты с материалами книги

Код рассматриваемых в этом издании алгоритмов вы можете загрузить с двух сайтов. Один из них принадлежит издательству ЭКСМО, выпустившему пере- вод книги, и находится по адресу [http://eksmo.ru/upload/Algorithms\_examples.zip.](http://eksmo.ru/upload/Algorithms_examples.zip) Другой представляет собой веб-страницу самого автора. Ее вы можете открыть по ссылке CSharpHelper.com/algoritms.html.

## Структура книги

**Глава 1 «Основы алгоритмов»** содержит основные сведения, необходимые для анализа алгоритмов. В ней рассказывается о разнице между алгоритмами и струк- турами данных, об асимптотических обозначениях, а также описываются случаи, в которых практические соображения важнее, чем теоретические расчеты.

В **главе 2 «Численные алгоритмы»** речь идет об алгоритмах, которые рандо- мизируют номера и массивы, находят наибольший общий делитель и наименьшее общее кратное, осуществляют быстрое возведение в степень, а также определяют, является ли число простым. Здесь вы также познакомитесь с важными методами адаптивной квадратуры и моделированием по методу Монте-Карло.

В **главе 3 «Список с указателями»** рассматриваются гибкие структуры данных, имеющие вид связного списка. Они хороши тем, что могут использоваться для хранения списков, размер которых со временем увеличивается. Изложенные здесь понятия пригодятся и для построения других структур данных, например таких, как деревья и сети.

**Глава 4 «Массивы»** посвящена алгоритмам, которые предназначаются для работы с разного рода массивами, в том числе треугольными и разреженными, и помогают экономить время и память программы.

В **главе 5 «Стеки и очереди»** идет речь об алгоритмах и структурах данных, которые позволяют программе хранить и отыскивать элементы, организованные по принципу «первым вошел — первым вышел» и «последним вошел — первым вы- шел». Эта информация окажется полезной для моделирования некоторых реальных сценариев, например очереди в кассу магазина.

В **главе 6 «Сортировка»** уделяется внимание многочисленным методам сор- тировки. Вы узнаете, какие алгоритмы с какими данными лучше справляются и ка- ковы их рабочие циклы. Примечательно то, что в этой главе приводятся одни из немногих кодов, для которых известны точные теоретические пределы произво- дительности.

**Глава 7 «Поиск»** содержит сведения об алгоритмах, которые могут исполь- зоваться программой для поиска по отсортированным спискам. На их примере наглядно демонстрируются такие важные методы, как бинарное разделение и ин- терполяция.

В **главе 8 «Хеш-таблицы»** изучаются структуры данных, использующие до- полнительную память для ускоренного поиска отдельных элементов. Они наглядно демонстрируют очень важное во многих программах распределение пространства и времени.

**Глава 9 «Рекурсия»** посвящена алгоритмам, вызывающим самих себя. В не- которых случаях это может быть полезно, но иногда становится причиной проблем. Здесь вы узнаете, как убрать рекурсию из алгоритма, если это возникла такая необ- ходимость.

В **главе 10 «Деревья»** рассматриваются древовидные структуры данных с вы- сокой рекурсивностью. Они полезны для хранения и изучения иерархически ор- ганизованной информации, а еще работы с ней, например в таких неожиданных случаях, как вычисление значений арифметических выражений.

В **главе 11 «Сбалансированные деревья»** вы узнаете о том, как избежать чрез- мерного разветвления данных. Это очень важно, поскольку массивные древовидные структуры замедляют эффективность работы алгоритмов.

В **главе 12 «Деревья решений»** рассказывается о методах работы со сложными задачами, которые представляют собой серию последовательных решений. Часто такие алгоритмы дают лишь приблизительный ответ, не всегда лучший из воз- можных. Тем не менее они очень гибкие и пригодны для широкого применения.

В **главе 13 «Основные сетевые алгоритмы**» анализируется работа алгоритмов, обеспечивающих посещение всех узлов сети, обнаружение циклов, создание свя- зующих деревьев, а также нахождение путей через сеть.

**Глава 14 «Другие сетевые алгоритмы»** продолжает знакомство с сетевыми алгоритмами на примере топологической сортировки обусловленных задач, рас- краски графов, клонирования сетей и распределения работы сотрудников.

В **главе 15 «Криптографические алгоритмы»** идет речь об алгоритмах, управ- ляющих строками. Одни из них, например поиск подстрок, в готовом виде уже встроены в инструменты большинства языков программирования, другие, такие как подбор скобок и поиск в них отличий, требуют дополнительной работы.

В **главе 16 «Криптография»** рассказывается о том, как шифровать и дешиф- ровать информацию. Вы познакомитесь с основами криптографии, а также с не- которыми интересными методами (шифрование открытым ключом) и шифрами (Виженера, блочным). Такие алгоритмы шифрования, как DES (Data Encryption Standard — стандарт шифрования данных) и AES (Advanced Encryption Standard — усовершенствованный стандарт шифрования), в данной книге не рассматриваются, поскольку являются предметом специализированной литературы.

**Глава 17 «Теория сложности»** посвящена двум важнейшим в компьютерной науке классам задач: П (решаются в обусловленном полиномиальном времени) и НП (решаются в необусловленном полиномиальном времени). Здесь вы найдете их описание, узнаете о том как определить, к какому из них относится конкретная задача, а также изучите взаимозаменяемость П и НП.

В **главе 18 «Распределенные алгоритмы»** рассматривается работа со мно- жественными процессорами, которые сегодня устанавливаются почти на всех современных компьютерах. Данные алгоритмы важны тем, что помогают полно- стью использовать потенциальные вычислительные мощности компьютерной техники.

В **главе 19 «Вопросы на собеседовании»** приводятся полезные советы и из- учаются методы, которые можно использовать для решения задач, предлагаемых

программистам во время собеседования при устройстве на работу. Здесь также со- держится список сайтов с подобными головоломками, на которых можно попрак- тиковаться в применении полученных знаний.

В **приложении А «Основные понятия алгоритмов»** собраны идеи и стратегии, которые используются в алгоритмах, описанных в книге. Опираясь на них, вы мо- жете найти решение самых разных задач.

В **приложении Б «Ответы к упражнениям»** приведены ключи к решению за- даний, содержащихся в конце каждой главы.

В **глоссарии** даются определения важных алгоритмических понятий. Вероятно, вам будет полезно просмотреть их перед собеседованием.

## Что нужно для работы с книгой

Для чтения вам не понадобятся специальные устройства. Однако если вы дей- ствительно хотите усвоить материал, старайтесь отрабатывать его на том языке программирования, которым владеете. Это поможет вам не только разобраться с алгоритмами, но и лучше понять особенности языка.

Тем, кто собирается вплотную заняться программированием, потребуется ком- пьютер, а также любая подходящая среда разработки. Примеры из этой книги, ко- торые выложены на приведенных выше сайтах, написаны на C# в среде Visual Studio 2012. Если вы захотите воспользоваться ими, вам понадобится установить C# 2012 на компьютер, позволяющий работать с Visual Studio. Это должна быть довольно быстрая вычислительная машина с большим объемом жесткого диска и оперативной памяти. Лично мне вполне хватает компьютера, имеющего процес- сор Intel Core 2 с частотой 1,83 ГГц, 2 Гбайта оперативной памяти и 500 Гбайт на жестком диске. На самом деле этот объем несколько больше того, что нужен, но дисковое пространство является относительно дешевым, так почему бы не приоб- рести его с запасом?

Visual Studio можно запустить и на менее мощных системах, однако в этом слу- чае работа компьютера может существенно замедлиться, поскольку данная среда требует довольно большого объема памяти.

Программы будут загружаться и запускаться при поддержке C# Express Edition (подробнее об этой среде можно прочесть здесь: microsoft.com/visualstudio/eng/downloads#d- express-windows-desktop), так что нет необходимости в установке более дорогой вер- сии C#.

## Условные обозначения

Чтобы вам было легче усвоить материал, в книге использованы некоторые условные обозначения.

**ВСЕВОЗМОЖНЫЕ ВРЕЗКИ**

Дополнительная информация, а также более подробные сведения по рас- сматриваемой теме.

**ВНИМАНИЕ**

Важная информация, напрямую относящаяся к расположенному рядом тек- сту, на которую следует обратить внимание.

**ЗАМЕЧАНИЕ**

Разного рода замечания, советы, подсказки, тонкости и отступления от темы.

Несколько слов о стилях.

* Важные слова и термины при первом употреблении выделены курсивом. Многие их них вы сможете найти в глоссарии в конце книги.
* Используемые в работе комбинации клавиш обозначаются следующим об- разом: Ctrl+A. Эта запись говорит о том, что нужно удерживать Ctrl и лишь за- тем нажать клавишу A.

Адреса интернет-ресурсов и электронной почты выделяются следующим об- разом: CSharpHelper.com, [RodStephens@CSharpHelper.com.](mailto:RodStephens@CSharpHelper.com)

Для набора кода используется один из двух способов:

моноширинный шрифт без выделения для большинства примеров кода;

**моноширинный полужирный шрифт для выделения кода, важного в данном контексте.**

## Обратная связь

Если у вас возникнут вопросы либо вы захотите что-то прокомментировать или предложить, пишите на электронный адрес [RodStephens@CSharpHelper.com.](mailto:RodStephens@CSharpHelper.com) Я не обещаю решить все алгоритмические задачи, но обязательно укажу правильное направление.

# Глава 1

**ОСНОВЫ АЛГОРИТМИЗАЦИИ**

Прежде чем приступить к изучению алгоритмов, рассмотрим несколько важных моментов. Для начала вы должны знать, что *алгоритм —* это набор команд, необхо- димых для решения той или иной задачи. Он определяет шаги, согласно которым она будет выполняться.

Такое определение кажется довольно простым, однако никто не станет писать алгоритмы для выполнения примитивных задач или создавать инструкции, что- бы получить доступ к четвертому элементу массива (подразумевается, что вы уже владеете минимальными навыками программирования и знаете, как это делается). Как правило, алгоритмы пишутся только для сложных задач, например в том слу- чае, когда нужно найти кратчайший путь через сеть из сотен улиц или отыскать наилучший вариант инвестиций для оптимизации прибыли.

В этой главе объясняются некоторые основные положения алгоритмизации, о которых вам следует иметь полное представление, особенно если есть желание извлечь максимальную пользу от чтения книги. Возможно, вы захотите пропустить приведенную здесь информацию и сразу приступить к изучению специальных алгоритмов. Однако лучше хотя бы поверхностно ознакомиться с изложенным ма- териалом. Подробно изучите подраздел «Асимптотическая сложность алгоритма» в разделе «Алгоритм и структура данных» текущей главы, поскольку понимание времени существенно для выбора нужного алгоритма. Очень важно, справится он с заданием за секунды, часы или не справится вообще.

## Метод

Чтобы понять работу алгоритма, недостаточно просто рассмотреть его шаги.

Нужно выяснить еще несколько важных моментов.

* **Поведение алгоритма.** Находит ли он наилучшее из возможных или просто хорошее решение? Может ли быть несколько наилучших решений? Есть ли смысл отдать предпочтение одному из них?
* **Скорость алгоритма.** Быстрый он или нет? Замедляет ли работу только с не- которыми входными данными?
* **Требования к памяти алгоритма**. Сколько компьютерных ресурсов ему необ- ходимо? Является ли такой объем приемлемым? Нужны ли алгоритму мил- лионы терабайтов памяти, которыми компьютер не располагает (по край- ней мере, сейчас)?
* **Основные методы, используемые в алгоритме.** Можно ли задействовать их повторно для решения подобных задач?

В данном издании затрагиваются все вышеперечисленные темы. Вместе с тем я не ставлю целью детально рассмотреть каждый алгоритм — используется преи- мущественно интуитивный подход, без излишне детального анализа его работы. Подробное изучение отнимает много времени, к тому же не требуется большинству программистов. Тем не менее книга ориентирована в первую очередь на профес- сионалов, которым нужны действенные и эффективные решения.

Все приведенные алгоритмы классифицированы определенным образом. Объеди- няющим признаком может выступать, например, задача, которую они выполняют (сортировка, поиск), используемые структуры данных (связные списки, массивы, хеш-таблицы, деревья) или методы (рекурсия, деревья решений, распределяемые ал- горитмы). Иногда объединение алгоритмов в группы может показаться странным, но по мере накопления информации вы убедитесь, что такой выбор сделан не случайно.

Многие алгоритмы включают несколько общих моментов, которые упомина- ются в разных главах. Например, древовидные алгоритмы (главы 10–12) обычно имеют высокую степень рекурсивности (глава 9). Связные списки (глава 3) могут использоваться для формирования массивов (глава 4), хеш-таблиц (глава 8), сте- ков (глава 5) и очередей (глава 5). Ссылки и указатели применяются для постро- ения связных списков (глава 3), деревьев (главы 10–12) и сетей (главы 13 и 14). При чтении книги обращайте внимание на подобную связь. В приложении А обоб- щены используемые в программах стратегии. Это сделано для того, чтобы следо- вать им было легче.

## Алгоритм и структура данных

Как сказано выше, алгоритм представляет собой набор команд для выполнения какой-либо задачи. При этом все данные, необходимые для ее решения, организуют- ся особым образом в так называемую *структуру*. Это может быть массив, связный список, дерево, граф, сеть или что-то более замысловатое.

Алгоритмы не могут существовать без структур данных. Например, алгоритм редакторского расстояния, который описывается в главе 15 и устанавливает схо- жесть двух строк, тесно связан с сетью и без нее не работает. Точно так же нет смысла строить структуру данных, если вы не планируете использовать ее вместе с алгоритмом.

## Псевдокод

Для описания алгоритмов в книге используются интуитивно понятные англий- ские термины. Это сделано для того, чтобы рассматриваемые примеры можно было применять в большинстве языков программирования. Однако часто использование алгоритма связано с некоторыми нюансами, которые приводятся в псевдокоде. *Псевдокод —* это текст, похожий на язык программирования, но не являющийся таковым. Он описывает структуры и детали, которые вам понадобятся для при- менения алгоритма в полноценном коде вне зависимости от используемого языка программирования.

В следующем фрагменте показан пример псевдокода для вычисления наиболь- шего общего делителя (НОД) двух целых чисел.

// Находим наибольший общий делитель для a и b.

// GCD(a, b) = GCD(b, a Mod b). Integer: Gcd(Integer: a, Integer: b)

While (b != 0)

// Вычисляем остаток. Integer: remainder = a Mod b

// Находим наибольший общий делитель для b и остатка. a = b

b = remainder End While

// Наибольший общий делитель для a и 0 — это a. Return a

End Gcd

**ОПЕРАТОР MOD**

Оператор mod используется для нахождения остатка от деления по модулю. Например, 13 mod 4 = 1, поскольку если 13 разделить на 4, получится целое число 3 и 1 в остатке. Само уравнение обычно читается как «13 мод 4» или

«13 по модулю 4».

Псевдокод начинается с комментария, который предваряется символом //.

В первой строке кода происходит объявление алгоритма. В нашем случае это алгоритм Gcd, который возвращает результат в виде целого числа и прини- мает два параметра с именами a и b, каждый из которых также является це- лым числом.

**ЗАМЕЧАНИЕ**

 Части кода, которые выполняют задачу и по требованию возвращают результат, обычно называются подпрограммами, методами, процедурами, субпроцеду-

рами или функциями.

Далее следует сам метод. Его начинает строка While, открывающая цикл. Код, идущий после этого оператора, будет исполняться до тех пор, пока действует за- данное условие.

Цикл заканчивается оператором End While. Использовать последний не обязательно — на конец цикла уже указывает отступ, но все же лучше обозначить блок команд.

О завершении метода свидетельствует оператор Return. Рядом с ним указана величина, которую должен вернуть приведенный алгоритм. Если целью алгорит- ма является не получение конкретного значения, а, например, систематизация ве- личин или построение структуры данных, то после оператора Return возвращае- мая величина не ставится.

Рассмотренный код ничем не отличается от обычного программного. В последующих примерах вы можете встретить описание инструкции либо

величины, приведенное на русском языке в угловых скобках (< >). Это зна- чит, что программный код на основе инструкций вам нужно дописать само- стоятельно.

После объявления параметра или переменной (в алгоритме Gcd это параметры a и b, а также переменная remainder) задается их тип с обязательным использо- ванием двоеточия. Например, Integer: remainder. Для простых циклических переменных с целыми числами подобной процедурой можно пренебречь, достаточно написать For i = 1 To 10.

Еще одно отличие псевдокода от других языков программирования — возмож- ность использовать цикл For вместе с оператором Step, указывающим на вели- чину, по которой изменяется циклическая переменная. В конце цикла не забудьте написать Next i (где i — переменная цикла), чтобы напомнить о его окончании.

Обратите внимание на следующий псевдокод:

For i = 100 To 0 Step -5

// Вычисляем… Next i

Он равнозначен нижеприведенному коду C#:

for (int i = 100; i >= 0; i -= 5)

{

// Вычисляем…

}

В псевдокодах из этой книги вы встретите If-Then-Else, Case и некоторые другие выражения. Они являются базовыми для всех языков программирования и должны быть вам знакомы. Остальное, что может понадобиться для кода, запи- сывается в комментариях.

Структура данных, которая может быть вам незнакома (в некоторых языках программирования ее нет), — это List. Она подобна саморасширяющемуся мас- сиву. С помощью метода Add к концу имеющегося списка данных можно добавить еще один элемент. Например, следующий псевдокод создает List Of Integer, в котором содержатся числа от 1 до 10.

List Of Integer: numbers For i = 1 To 10

numbers.Add(i) Next i

После того как список объявлен, псевдокод может работать с ним, как с обыч- ным массивом, и обращаться к любому его элементу. Однако в отличие от массивов, списки позволяют удалять какой бы то ни было элемент и добавлять его в какую угодно позицию.

Многие алгоритмы в книге написаны в виде методов и функций, возвращающих результат. В этом случае в объявлении функции нужно указать тип получаемых данных. Если функция выполняется без возвращения результата, тип данных не задается.

В следующем псевдокоде содержатся два метода.

// Возвращает удвоенное значение введенного числа. Integer: DoubleIt(Integer: value)

Return 2 \* value End DoubleIt

// Следующая функция выполняется без возвращения результата. DoSomething(Integer: values[])

// Код.

...

End DoSomething

Первая функция DoubleIt получает и возвращает целое число, удваивая его значение в процессе работы. Вторая функция DoSomething использует в качестве параметра массив целых чисел, называемых величинами. Она также выполняет задание (например, перемешивает или сортирует величины), но результат не воз- вращает. Обратите внимание и на то, что в книге индексация элементов внутри массива начинается с 0, то есть если массив состоит из трех элементов, индексы у них будут 0,1 и 2.

Приведенные псевдокоды должны быть интуитивно понятны. В случае каких- либо затруднений вы всегда можете зайти на форум [www.wiley.com/go/essentialalgorithms](http://www.wiley.com/go/essentialalgorithms) и задать интересующий вас вопрос либо написать мне на электронный адрес [RodStephens@CSharpHelper.com](mailto:RodStephens@CSharpHelper.com) — я постараюсь дать подсказку.

Пожалуй, единственный существенный недостаток псевдокода — отсутствие компилятора, способного обнаружить ошибки. Большинство алгоритмов, реали- зованных на C#, а также приведенные в книге упражнения можно скачать на ука- занных во введении сайтах. Это позволит вам проверить собственную версию кода или получить нужную подсказку.

## Свойства алгоритма

Хороший алгоритм должен быть правильным, надежным и эффективным. Если он не справляется с поставленной задачей, от него мало пользы. Нет смысла в нем и тогда, когда он дает неправильное решение.

**ЗАМЕЧАНИЕ**

 Некоторые алгоритмы срабатывают нужным образом не всегда, а лишь вре- мя от времени, но даже это делает их полезными. Благодаря им вы можете

получить нужную информацию с определенной долей вероятности. Чтобы убедиться, что ответ правильный, вам придется выполнить код несколько раз. Примером такого алгоритма является тест простоты Ферма, описанный в главе 2.

Если алгоритм не является надежным, использовать его в программе стано- вится небезопасно. По-настоящему хороший код должен быть простым, интуитив- но понятным и изящным. Только в этом случае вы получите правильный резуль- тат или сможете внести необходимые изменения в структуру программы. Если алгоритм сложный и запутанный, у вас возникнут проблемы не только с его ре- ализацией, но и с устранением возможных ошибок. Под сомнение можно будет

поставить и сам результат: ведь если алгоритм непонятен, как узнать, правиль- но он работает или нет?

Большинство разработчиков много времени уделяют эффективности алго- ритмов. И не зря. Правильный результат, легкость реализации и возможность исправить ошибки — все это сведется к нулю, если алгоритм окончит свою ра- боту только через семь лет или потребует объем памяти больше того, что есть в компьютере.

**ЗАМЕЧАНИЕ**

Вышесказанное не означает, что сложные и запутанные алгоритмы не нужно изучать. Даже если у вас возникают трудности с их реализацией, попытка ра- зобраться в предлагаемом коде уже многого стоит. Спустя некоторое время вы улучшите навыки владения алгоритмами, и работать станет гораздо лег- че. Тем не менее всегда тщательно проверяйте алгоритмы, чтобы убедиться в том, что они дают правильные результаты.

Чтобы проверить эффективность алгоритма, программисты задаются вопросом: как изменится производительность кода, если скорректировать размерность зада- чи? Если удвоить количество переменных в алгоритме, не удвоится ли время его работы? А если возвести начальное количество переменных в четвертую степень, не станет ли полученный алгоритм выполняться несколько лет?

В таком же ключе следует рассуждать о расходе оперативной памяти и других ресурсах, необходимых для работы алгоритма. Стоит озаботиться и производитель- ностью алгоритма при различных условиях. Каков наихудший вариант? Сколь велика вероятность его возникновения? Если запустить алгоритм с использованием большого объема произвольных данных, то какова будет его средняя производительность?

Чтобы получить представление о том, как сложность задачи влияет на произво- дительность, программисты пользуются понятием «асимптотическая сложность». Об этом мы и поговорим ниже.

#### Асимптотическая сложность алгоритма

*Асимптотическая сложность (производительность)* определяется функцией, которая указывает, насколько ухудшается работа алгоритма с усложнением по- ставленной задачи. Такую функцию записывают в круглых скобках, предваряя прописной буквой *О*.

Например, *O*(*N*2) означает, что по мере увеличения количества входных дан- ных время работы алгоритма (использование памяти либо другой измеряемый параметр) возрастает квадратично. Если данных станет вдвое больше, произ- водительность алгоритма замедлится приблизительно в четыре раза. При уве- личении количества входных данных в три раза она станет меньше в девять раз.

**ЗАМЕЧАНИЕ**

 Выражение *O*(*N*2) читается как «порядок *N* в квадрате». Например, алгоритм быстрой сортировки, описанный в главе 6, имеет наихудшую производитель-

ность порядка *N*2.

Существуют пять основных правил для расчета асимптотической сложности алгоритма.

1. Если для математической функции *f* алгоритму необходимо выполнить опре- деленные действия *f*(*N*) раз, то для этого ему понадобится сделать *O*(*f*(*N*)) шагов.
2. Если алгоритм выполняет одну операцию, состоящую из *O*(*f*(*N*)) шагов, а за- тем вторую операцию, включающую *O*(*g*(*N*)) шагов, то общая производитель- ность алгоритма для функций *f* и *g* составит *O*(*f*(*N*) + *g*(*N*)).
3. Если алгоритму необходимо сделать *O*(*f*(*N*) + *g*(*N*)) шагов и область значений *N* функции *f*(*N*) больше, чем у *g*(*N*), то асимптотическую сложность можно упростить до выражения *O*(*f*(*N*)).
4. Если алгоритму внутри каждого шага *O*(*f*(*N*)) одной операции приходится выполнять еще *O*(*g*(*N*)) шагов другой операции, то общая производитель- ность алгоритма составит *O*(*f*(*N*)  *g*(*N*)).
5. Постоянными множителями (константами) можно пренебречь. Если *C* явля- ется константой, то *O*(*C*  *f*(*N*)) или *O*(*f*(*C*  *N*)) можно записать как *O*(*f*(*N*)).

Приведенные правила кажутся немного формальными из-за абстрактных функ- ций *f*(*N*) и *g*(*N*), но на самом деле ими очень легко пользоваться на практике. Ниже приведено несколько примеров, которые облегчат понимание.

#### Правило 1

*Если для математической функции f алгоритму необходимо выполнить опреде- ленные действия f(N) раз, то для этого ему понадобится сделать O(f(N)) шагов.*

Рассмотрим псевдокод с алгоритмом по нахождению наибольшего целого числа в массиве.

Integer: FindLargest(Integer: array[]) Integer: largest = array[0]

For i = 1 To <наибольший индекс>

If (array[i] > largest) Then largest = array[i]

Next i

Return largest End FindLargest

В качестве входного параметра алгоритм FindLargest использует массив целых чисел, результат возвращается в виде одного целого числа. В самом начале переменной largest присваивается значение первого элемента массива. Затем алгоритм перебирает оставшиеся элементы и сравнивает значение каждого из них с largest. Если он находит большую величину, то приравнивает largest к ней и по окончании цикла возвращает наибольшее найденное значение.

Алгоритм проверяет каждый из *N* элементов массива всего один раз, поэтому его производительность составляет *O*(*N*).

**ЗАМЕЧАНИЕ**

 Как правило, основная часть работы алгоритма приходится на циклы. Другого способа, позволяющего совершить более *N* шагов с фиксированным числом

строк кода, нет. Изучите циклы, чтобы выяснить, сколько времени они занимают.

#### Правило 2

*Если алгоритм выполняет одну операцию, состоящую из O(f(N)) шагов, а затем вторую, требующую O(g(N)) шагов, то общая производительность алгоритма для функций f и g будет O(f(N) + g(N)).*

Вернемся к алгоритму FindLargest. На этот раз обратите внимание, что несколько строк в действительности не включены в цикл. В следующем псевдокоде в комментариях справа приведен порядок времени выполнения все тех же шагов.

Integer: FindLargest(Integer: array[])

Integer: largest = array[0] // O(1) For i = 1 To <наибльший индекс> // O(N)

If (array[i] > largest) Then largest = array[i]

Next i

Return largest // O(1)

End FindLargest

Итак, приведенный алгоритм выполняет один шаг отладки перед циклом и еще один после него. Каждый из них имеет производительность *O*(1) (это однократное действие), поэтому общее время работы алгоритма составит *O*(1 + *N* + 1). Если ис- пользовать обычную алгебру и преобразовать выражение, то получится *O*(2 + *N*).

#### Правило 3

*Если алгоритму необходимо сделать O(f(N) + g(N)) шагов и область значений N функции f(N) больше, чем у g(N), то асимптотическую сложность можно упрос- тить до выражения O(f(N)).*

В предыдущем примере мы выяснили, что время работы алгоритма FindLargest определяется выражением *O*(2 + *N*). Если параметр *N* начнет возрастать, его зна- чение превысит постоянную величину 2 и предыдущее выражение можно будет упростить до *O*(*N*).

Игнорирование меньших функций позволяет пренебречь небольшими задачами отладки и очистки, чтобы сосредоточить внимание на асимптотическом поведении алгоритма, которое обнаруживается при усложнении задачи. Другими словами, время, затраченное алгоритмом на построение простых структур данных перед выполнением объемного вычисления, является несущественным по сравнению с длительностью основных расчетов.

#### Правило 4

*Если алгоритму внутри каждого шага O(f(N)) одной операции приходится вы- полнять еще O(g(N)) шагов другой операции, то общая производительность алго- ритма составит O(f(N)*  *g(N)).*

Рассмотрим алгоритм, который определяет, содержатся ли в массиве повто- ряющиеся элементы. (Стоит отметить, что это не самый эффективный способ обнаружения дубликатов.)

Boolean: ContainsDuplicates(Integer: array[])

// Цикл по всем элементам массива. For i = 0 To <наибольший индекс>

For j = 0 To <наибольший индекс>

// Проверяем, являются ли два элемента дубликатами. If (i != j) Then

If (array[i] == array[j]) Then Return True

Next i

Next j

End If

// Если мы дошли до этой строки, то дубликатов нет. Return False

End ContainsDuplicates

Алгоритм содержит два цикла, один из которых является вложенным. Внешний цикл перебирает все элементы массива *N*, выполняя *O*(*N*) шагов. Внутри каждого такого шага внутренний цикл повторно пересматривает все *N* элементов массива, совершая те же *O*(*N*) шагов. Следовательно, общая производительность алгоритма составит *O*(*N*  *N*) = *O*(*N*2).

#### Правило 5

*Постоянными множителями (константами) можно пренебречь. Если C явля- ется константой, то O(C*  *f(N)) или O(f(C*  *N)) можно записать как O(f(N)).*

Снова посмотрите на алгоритм ContainsDuplicates из предыдущего при- мера и обратите внимание на внутренний цикл, который представлен условием If. В рамках этого условия определяется, равны ли друг другу индексы i и j. Если нет — тогда сравниваются величины array[i] и array[j], в случае их совпадения возвращается значение True.

Пренебрегая дополнительным шагом в выражении Return (как правило, он вы- полняется один раз), предположим, что срабатывают оба оператора If (а так и про- исходит в большинстве случаев), тогда внутренний цикл будет пройден за *O*(2*N*) шагов. Следовательно, общая производительность алгоритма составит *O*(*N*  2*N*) =

= *O*(2*N*2). Последнее правило позволяет пренебречь коэффициентом 2 и записать производительность алгоритма в виде *O*(*N*2).

На самом деле мы возвращаемся к сути асимптотической сложности: нужно выяснить, как поведет себя алгоритм, если *N* начнет возрастать. Предположим, вы увеличите *N* в два раза, то есть будете оперировать значением 2*N*. Теперь если под- ставить фразу в выражение 2*N*2, получится следующее: 2  (2*N*)2 = 2  4*N*2 = 8*N*2. Это и есть наша величина 2*N*2, только умноженная на 4. Таким образом, время работы алгоритма увеличится в четыре раза.

Теперь давайте оценим производительность алгоритма, используя упрощенное по правилу выражение *O*(*N*2). При подстановке в него 2*N* получим следующее: (2*N*)2 = 4*N*2. То есть наша изначальная величина *N*2 возросла в четыре раза, как и время работы алгоритма.

Из всего вышесказанного следует, что независимо от того, будете вы ис- пользовать развернутую формулу 2*N*2 или ограничитесь просто *N*2, результат останется прежним: увеличение сложности задачи в два раза замедлит работу алгоритма в четыре раза. Таким образом, важной здесь является не константа 2,

а тот факт, что время работы увеличивается вместе с увеличением количества вводов *N*2.

**ЗАМЕЧАНИЕ**

 Важно помнить, что асимптотическая сложность дает представление о теоре- тическом поведении алгоритма. Практические результаты могут отличаться.

Предположим, производительность алгоритма равна *O*(*N*); если не прене- брегать целыми числами, реальное количество выполняемых шагов составит приблизительно 100 000 000 + *N*. В этом случае, пока *N* не слишком велико, значение константы лучше учитывать.

#### Обычные функции рабочего цикла

При изучении алгоритмов вы часто будете встречать одни и те же функции. Рассмотрим некоторые из них, чтобы получить общее представление и возмож- ность с ходу определять, является ли, например, алгоритм с производительностью *O*(*N*3) уместным.

#### 1

Алгоритм с производительность *O*(1) выполняется за один и тот же отрезок времени, независимо от сложности задачи. Как правило, речь идет об ограничен- ном круге команд, поскольку за *O*(1) времени невозможно даже просмотреть все входные данные.

Например, в какой-то момент алгоритму быстрой сортировки необходимо выбрать из массива чисел то, которое будет равно среднему значению. То есть если у нас есть массив целых чисел от 1 до 100, наилучшим значением окажется число 50. В приве- денном ниже алгоритме показан общий подход к решению этой задачи.

Integer: DividingPoint(Integer: array[]) Integer: number1 = array[0]

Integer: number2 = array[<последний индекс массива>] Integer: number3 = array[<последний индекс массива> / 2]

If (<number1 между number2 и number3>) Return number1 If (<number2 между number1 и number3>) Return number2 Return number3

End MiddleValue

Подобный алгоритм выбирает три величины (в начале, конце и середине масси- ва), сравнивает их между собой и возвращает среднее значение. Не факт, что выбор окажется наилучшим, но вполне вероятно, что худшим также не будет.

Поскольку алгоритм выполняет лишь несколько определенных шагов, его про- изводительность равняется *O*(1), а время работы не зависит от количества вводов *N*. (Конечно, рассмотренный фрагмент является лишь малой частью более сложного алгоритма.)

#### Log (N)

Алгоритм с производительностью *O*(log (*N)*) делит количество рассматривае- мых им элементов на фиксированный коэффициент при каждом шаге.

4

9

**ЛОГАРИФМЫ**

Логарифм числа по определенному основанию — это степень, в которую нужно возвести основание, чтобы получить данное число. Например, log2 8 = 3, по- скольку 23 = 8, где 2 — основание логарифма.

Как правило, в алгоритмах основание равно 2, поскольку входные данные делятся на две группы. Как вы скоро увидите, основание логарифма не важно для асимптотической сложности, поэтому его обычно опускают.

На рисунке 1.1 показано упорядоченное полное бинарное дерево. *Бинарное* оно потому, что каждая вершина имеет не более двух ветвей. *Полное* потому, что каж- дый уровень (кроме последнего) целиком заполнен, а все вершины на последнем уровне сгруппированы в левой части. И, наконец, *упорядоченное* потому, что зна- чение каждой вершины сравнимо по величине со значением левого дочернего узла и не больше значения правого.

7

3

1

2

6

10

8

**Рис. 1.1.** Пример упорядоченного полного бинарного дерева

Следующий псевдокод описывает поиск определенного элемента по дереву, изображенному на рисунке 1.1.

5

Node: FindItem(Integer: target\_value) Node: test\_node = <корень дерева>

Do Forever

// Если мы выпадаем из дерева, значит, искомой величины нет. If (test\_node == null) Return null

If (target\_value == test\_node.Value) Then

// test\_node содержит искомую величину.

// Это и есть та вершина, которую мы искали. Return test\_node

Else If (target\_value < test\_node.Value) Then

// Переходим по левой ветке. test\_node = test\_node.LeftChild

End Do

Else

End If

// Переходим по правой ветке. test\_node = test\_node.RightChild

End FindItem

Далее в главе 10 вы найдете подробное описание этого алгоритма, но сейчас вам нужно понять его суть. Алгоритм объявляет переменную test\_node и при- сваивает ей начальное значение, расположенное вверху дерева. (В отличие от на- стоящих деревьев, у компьютерных корни находятся вверху.) Затем запускается бесконечный цикл.

Если переменная test\_node равняется null, искомой величины на дереве нет — и алгоритм возвращает null.

**ЗАМЕЧАНИЕ**

 Null — особая величина, которую можно использовать в качестве перемен- ной для указания на объект, например на вершину или на дерево. Ее значение

можно перевести как «эта переменная ни на что не указывает».

Если test\_node содержит искомую величину, то переменная является вер- шиной, которую мы ищем, и алгоритм возвращает ее.

Когда искомая величина target\_value меньше величины test\_node, алго- ритм приравнивает test\_node к дочерней вершине слева, то есть к LeftChild. (При test\_node, находящемся внизу дерева, LeftChild равняется null, а ал- горитм исправляет положение во время следующего прохождения цикла.)

Если величина test\_node не равна target\_value и не меньше ее, значит, она больше. В этом случае алгоритм считает значение test\_node равным значе- нию дочерней вершины справа, то есть RightChild. (И снова, если test\_node располагается внизу дерева, то RightChild равняется null, а алгоритм исправляет положение при следующем прохождении цикла.)

Таким образом переменная test\_node постепенно проходит по дереву сверху вниз и, наконец, либо обнаруживает искомую величину, либо выпадает из дерева (test\_node равняется null).

В этом алгоритме важно знать, насколько низко переместится test\_node, пока не найдет target\_value либо не выпадет из дерева. Если повезет, искомая величина отыщется сразу — например, число 7 (см. рис. 1.1) обнаружится за один шаг. Проверка небольшой части дерева может потребоваться и в том случае, когда нужное значение располагается не в корневой вершине, как, например, число 4. Самым худшим вариантом считается тот, при котором алгоритму придется про- изводить поиск по всему дереву — от корневой вершины до низа.

Если присмотреться, становится заметно: почти половина вершин дерева сосре- доточена внизу, причем у них отсутствуют дочерние ответвления. В *полном* дереве, где каждая вершина содержит по две дочерние вершины или не имеет их вовсе, на нижний уровень приходится ровно половина всех вершин. В этом случае при по- иске случайно выбранных величин большую часть времени алгоритм затратит на то, чтобы пройти почти по всей высоте дерева.

Следующий важный вопрос — насколько высоким является дерево. Полное завершенное двоичное дерево высотой *H* включает примерно 2*H* вершины. Если

исходить из обратного, то *N* вершин составят дерево высотой log2 (*N)*. Посколь- ку алгоритм движется по дереву сверху вниз в худшем (и близком к нему) случае, а само дерево имеет высоту приблизительно log2 (*N)*, работа алгоритма займет вре- мя, равное *O*(log2 (*N)*).

На этом этапе обнаруживается интересная особенность. Вы можете преобра- зовать логарифм с основанием *A* в логарифм с основанием *B,* используя формулу log*B x* = log*A x* / log*A B*.

Если представить, что *B* = 2, то замена основания *A* в логарифме из выражения

*O*(log2 (*N)*) будет выглядеть следующим образом: *O*(log2 (*N)*) = *O*(log*A* (*N)* / log*A* 2).

Значение 1/log*A* 2 постоянно при любом *А*, и поскольку асимптотическое обо- значение пренебрегает величинами, умноженными на константу, то *O*(log2 (*N)*) бу- дет равен *O*(log*A* (*N)*) для какого угодно логарифма с основанием А. По этой при- чине время работы можно записать как *O*(log (*N)*) без указания основания и без

внутренних скобок.

Данный алгоритм — типичный пример тех, что имеет производительность *O*(log (*N)*). На каждом шаге он делит количество рассматриваемых элементов при- мерно надвое. Но для асимптотической сложности основание логарифма не име- ет значения, поэтому неважна и доля, которую использует алгоритм для деления. Производительность *O*(log (*N)*) сохранилась бы даже в том случае, если бы остав- шиеся элементы делились на коэффициент 1/10 и каждый шаг алгоритма был бы более прогрессивен либо на 9/10 и каждый шаг был бы прогрессивен куда меньше.

Логарифмическая функция log (*N)* с увеличением *N* возрастает сравнительно медленно, потому алгоритмы с производительностью *O*(log (*N)*) обычно достаточ- но быстры. В этом и заключается их главная польза.

#### sqrt (N)

Алгоритмы с производительностью *O*(sqrt (*N*)), где sqrt — функция извлече- ния квадратного корня, не являются общими и не рассматриваются в данной кни- ге. Эта функция возрастает очень медленно, хотя и несколько быстрее, чем log (*N)*.

#### N

Алгоритм FindLargest (см. пункт «Правило 1» подраздела «Асимптотиче- ская сложность алгоритма» в разделе «Свойство алгоритма» текущей главы) имеет производительность *O*(*N*). Функция *N* возрастает быстрее, чем log (*N)* и sqrt (*N*), но все же не так быстро, поэтому большинство подобных алгоритмов демонстри- рует на практике хорошую производительность.

#### N log N

Предположим, что алгоритм перебирает все элементы в поставленной задаче, а затем в отдельном цикле выполняет с элементом какую-то операцию *O*(log (*N)*).

В этом случае производительность алгоритма определяется выражением *O*(*N*  log *N*) или *O*(*N* log *N*). Как вариант, задача может выглядеть так: алгоритм выполняет опе- рацию *O*(log (*N)*) и на каждом ее шаге что-то делает с каждым элементом.

Предположим, у вас есть отсортированное дерево, содержащее *N* элементов (как описано выше) и такой же размерности массив. Вам надо узнать, какие из элементов массива присутствуют в дереве. Один из способов выяснить это — осу- ществить циклическое прохождение по величинам массива, задействовав описан- ный ранее метод поиска по дереву. В процессе работы алгоритм проверит *N* эле- ментов и выполнит log (*N)* шагов для каждого из них, так что общая производи- тельность будет *O*(*N*  log *N*).

Для многих алгоритмов сортировки, основанных на сравнении элементов, вре- мя работы составляет *O*(*N*  log *N*). На самом деле можно доказать, что любой по- добный алгоритм проходит как минимум такое количество шагов, что очень удоб- но для определения асимптотической сложности. Однако некоторые алгоритмы являются все же более быстрыми благодаря целым числам, которые игнорируют- ся асимптотическим обозначением.

**N2**

Алгоритм, который изначально перебирает все входные данные, а затем пере- сматривает их еще раз для каждого единичного значения, имеет производитель- ность *O*(*N*2). Например, алгоритм ContainsDuplicates (см. пункт «Правило 4» подраздела «Асимптотическая сложность алгоритма» в разделе «Свойство ал- горитма» текущей главы) выполняется именно за такое время. Возможны и дру- гие степени *N*, предположим *O*(*N*3) или O(*N*4). Однако они являются значительно более медленными, чем *O*(*N*2).

Говорят, алгоритм имеет *многочленное* время работы, если оно включает в себя многочлен *N*: например *O*(*N*), *O*(*N*2), *O*(*N*6) и даже *O*(*N*4000). В любом случае подоб- ные задачи можно решить. А вот экспоненциальное и факториальное время ра- боты, описанное ниже, возрастает очень быстро, поэтому алгоритмы с указанной производительностью будут применимы только для небольшого количества вход- ных данных.

**2N**

Экспоненциальные функции, такие как 2*N*, возрастают молниеносно и поэтому полезны для решения ограниченного круга задач. Обычно посредством алгоритмов с подобным временем работы ищется оптимальный набор входных данных.

В качестве примера рассмотрим следующую задачу. Допустим, у вас есть рюкзак определенной грузоподъемности и набор принадлежностей, каждый из которых имеет свою массу и значение. Вам необходимо наполнить рюкзак предметами так, чтобы их общее значение оказалось наибольшим.

Задача может показаться простой, но все известные алгоритмы для нахожде- ния наилучшего решения потребуют изучить каждую возможную комбинацию предметов. Если учесть, что любой предмет способен пребывать только в двух со- стояниях (находиться в рюкзаке или вне его), то, умножив количество состояний на количество предметов, получим 2  2  ...  2 = 2*N* вариантов выбора.

Иногда нет необходимости проверять все возможные комбинации. Например, если первый предмет заполняет собой рюкзак, то добавлять к нему еще что-то уже не имеет смысла. Однако в большинстве случаев вам не удастся исключить доста- точное количество вариантов, чтобы значительно сузить область поиска.

При решении задач с экспоненциальным временем работы часто пользуются *эвристиками* — алгоритмами, которые дают хорошие результаты, но не гарантируют, что они будут наилучшими.

#### N!

Функция *N*! (читается как «*N* факториал») рекомендуется для работы с целыми числами больше 0 и определяется формулой *N*! = 1  2  3  ...  *N*. Она возрастает намного быстрее, чем экспоненциальная функция 2*N*.

В алгоритмах с факториальным временем работы, как правило, ищется опти- мальное распределение входных данных. Например, у торгового представителя имеется список городов. Его задача — составить маршрут таким образом, чтобы посетить каждый населенный пункт один раз и вернуться в отправную точку, пре- одолев минимальное расстояние.

Если городов немного, то все просто, но если список длинный, задача услож- няется на порядок. Наиболее очевидный способ решения — перебрать все воз- можные варианты маршрута. Так, с помощью алгоритма можно выбрать *N* близ- лежащих населенных пунктов для первого города. После этого у вас останет- ся *N* – 1 возможных пунктов, которые еще нужно посетить, затем *N* – 2 и т. д. Следовательно, общее количество вариантов маршрутов составит *N*  (*N* – 1)   (*N* – 2)  ...  1 = *N*!.

#### Визуализация функций

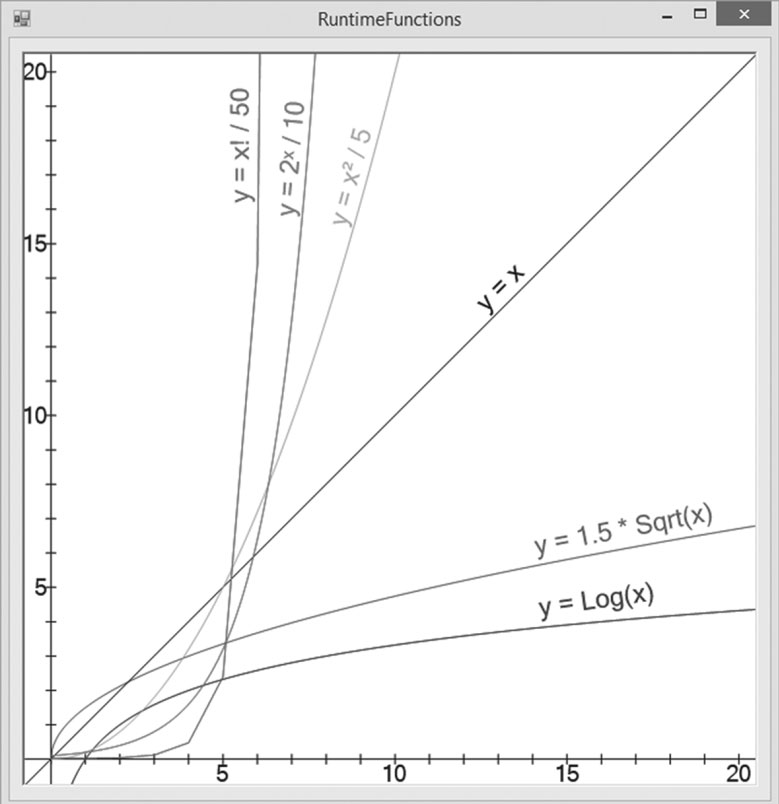
В таблице 1.1 представлены значения для функций, которые определяют время работы алгоритмов, описанных в предыдущих разделах. Она поможет оценить, на- сколько быстро возрастают эти функции.

**Òàáëèöà 1.1.** Âåëè÷èíû ôóíêöèé äëÿ ðàçëè÷íûõ âõîäíûõ äàííûõ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***N*** | **log2 *N*** | **sqrt (*N*)** | ***N*** | ***N2*** | **2*N*** | ***N*!** |
| 1 | 0 | 1 | 1 | 1 | 2 | 1 |
| 5 | 2,32 | 2,23 | 5 | 25 | 32 | 625 |
| 10 | 3,32 | 3,16 | 10 | 100 | 1 024 | 1  109 |
| 15 | 3,90 | 3,87 | 15 | 225 | 3,3  04 | 2,9  1016 |
| 20 | 4,32 | 4,47 | 20 | 400 | 1  106 | 5,24  1024 |
| 50 | 5,64 | 7,07 | 50 | 2500 | 1,1  1015 | 1,8  1083 |
| 100 | 6,64 | 10,00 | 100 | 1  104 | 1,3 1 030 |   10198 |
| 1 000 | 9,96 | 31,62 | 1 000 | 1  106 | 1,1  10301 | – |
| 10 000 | 13,28 | 100,00 | 1  104 | 1 108 | – | – |
| 100 000 | 16,60 | 316,22 | 1 105 | 1  1010 | – | – |

На рисунке 1.2 приведены соответствующие графики. Для наглядности масштаб некоторых из них был изменен, но и так понятно, какая они себя ведут при увели- чении *x*: функции десятичного логарифма, квадратного корня, линейных и даже

многочленных уравнений возрастают с умеренной скоростью, в то время как экс- поненциальные и факториальные делают это чрезвычайно быстро.



**Рис. 1.2.** Графики возрастания различных функций

## Практические рекомендации

Теория позволяет понять, как изменяется рабочий цикл алгоритма, однако не менее важны и практические моменты. Например, при анализе алгоритма считается, что все шаги занимают одинаковое количество времени, даже если это не соответ- ствует действительности. Взять хотя бы создание и удаление новых объектов — на них требуется намного больше времени, чем на перемещение целых чисел из одной части массива в другую. В таком случае алгоритм с массивами будет работать лучше, чем тот, что использует солидное количество объектов и выглядит лучше в асимптотическом обозначении.

Кроме того, многие среды программирования предоставляют доступ к функ- циям операционной системы, которые являются более эффективными, чем основ- ные технологии алгоритмизации. Например, часть алгоритма, который выполняет сортировку, требует перемещения элементов массива на одну позицию вниз, чтобы перед ними можно было поставить новый элемент. Этот довольно медленный про- цесс приводит к тому, что работа алгоритма определяется как *O*(*N*2). Однако мно-

гие программы могут использовать особую функцию, перемещающую блоки памя- ти (например, RtlMoveMemory в .NET-программах и MoveMemory в программах Windows C++). Таким образом, вместо перемещения одного элемента при прохож- дении через массив программа заставляет функции передвигать последовательно- сти величин, значительно ускоряя процесс.

Даже если алгоритм имеет оптимальную асимптотическую сложность, для увеличения производительности допустимо использовать инструменты, кото- рые предлагает среда программирования. Некоторые из них способны выпол- нять те же задачи, которые рассматриваются в этой книге. Например, многие библиотеки содержат программы сортировки и отлично подходят для работы с массивами данных. Так, Microsoft .NET Framework, используемая в C# и Visual Basic, включает метод Array.Sort, реализацию которого вам вряд ли удастся превзойти в собственном коде. Хотя в особых случаях вы сможете это сделать, если будете располагать дополнительной информацией о данных (см. главу 6).

Подобные специфические библиотеки могут быть общедоступны и весьма по- лезны при выполнении определенных задач. Например, эффективно использовать библиотеку сетевого анализа, вместо того чтобы самому разрабатывать некоторые инструменты, и сэкономить время, предположим, на построении деревьев или сортировке. Конечно, ничто не мешает выполнять эти задачи собственноручно, но обратиться к готовой базе данных намного проще.

Если в ваших инструментах программирования есть функции, заменяющие работу какого-либо алгоритма, обязательно воспользуйтесь ими. Так вы добьетесь лучшей производительности и затратите меньше сил на отладку.

Наконец, для очень больших задач самый эффективный алгоритм не всегда является самым быстрым. Допустим, вы сортируете длинный список чисел. В этом случае алгоритм быстрой сортировки сработает хорошо. Но если у вас всего три числа, простой ряд операторов If, возможно, проявит себя куда лучше. Вряд ли будет иметь значение, справится программа с задачей за 1 или за 2 мс. Если вы не собираетесь выполнять сортировку многократно, разумнее прибегнуть к простому алгоритму, который легче поддается отладке, а не к сложному, который сэкономит всего 1 мс.

Если вы пользуетесь библиотеками, которые аналогичны тем, что описаны выше, возможно, вам не придется разрабатывать все алгоритмы самостоятельно, но понимать принцип их действия крайне необходимо, чтобы извлечь как можно больше пользы из инструментов, которые применяются для их реализации. На- пример, если вам известно, что реляционные базы данных работают с *B*-деревьями и чем-то подобным для хранения индексов, то будет ясно, насколько важны коэф- фициенты предварительного распределения и заполнения. Если же вы разбирае- тесь в алгоритме быстрой сортировки, то вас вряд ли удивит, что некоторые люди не считают метод Array.Sort .NET Framework криптостойким (см. пункт «Ис- пользование быстрой сортировки» подраздела «Быстрая сортировка» раздела «Ал- горитмы O(*N*  log *N*)» в главе 6).

Понимание алгоритмов может пригодиться и в других ситуациях. Возможно, вам не понадобится сам алгоритм сортировки слияния, но будет полезен его ме- тод «разделяй и властвуй» для решения задач со множественными процессорами.

## Резюме

Чтобы получить максимальную пользу от алгоритмов, необходимо понимать принцип их действия и иметь представление об основных характеристиках произ- водительности. В этой главе мы рассмотрели асимптотическую сложность и узнали, как на ее основе предсказать поведение того или иного алгоритма при изменении размерности задачи, а также познакомились с некоторыми типичными алгоритмами и общими функциями, которые описывают их рабочий цикл.

Рисунок 1.2 позволил оценить, насколько быстро растет график каждой такой функции. На практике алгоритмы с многочленным временем работы оказываются относительно быстрыми, поэтому к ним удобно обратиться для решения умеренно сложных вопросов, в отличие от алгоритмов с экспоненциальными или факториаль- ными рабочими циклами, у которых производительность изменяется очень быстро и к которым лучше апеллировать в нетрудоемких процессах.

Теперь, когда у вас есть некоторое представление о том, как анализировать ско- рость алгоритмов, вы можете приступить к изучению частных случаев, например численных алгоритмов. Обычно они имеют несложную структуру данных, поэтому работают сравнительно быстро.

## Упражнения

Звездочкой обозначены задачи повышенной сложности.

1. В пункте «Правило 4» подраздела «Асимптотическая сложность алгорит- ма» в разделе «Свойство алгоритма» текущей главы был описан алгоритм ContainsDuplicates, имеющий время работы *O*(*N*2). Рассмотрим его улуч- шенную версию.

Boolean: ContainsDuplicates(Integer: array[])

// Цикл по всем элементам массива, кроме последнего. For i = 0 To <наибольший индекс> - 1

// Цикл по элементам после элемента i. For j = i + 1 To <наибольший индекс>

// Проверяем, являются ли два элемента дубликатами. If (array[i] == array[j]) Then Return True

Next j

Next i

// Если вы дошли до этой строки, то дубликатов нет. Return False

End ContainsDuplicates

Определите время работы для новой версии алгоритма.

1. В таблице 1.1 показана взаимосвязь между сложностью задачи *N* и произво- дительностью алгоритмов, описанных разными функциями. Эту же зависи- мость можно продемонстрировать еще одним способом — определить макси- мальную сложность задачи, которую способен выполнить компьютер с опре- деленной скоростью в течение конкретного времени.

Предположим, компьютер проделывает 1 млн шагов алгоритма за 1 с и алго- ритм выполняется в течение времени *O*(*N*2). В таком случае через 1 ч машина решит задачу, у которой *N* = 60 000 (поскольку 60 0002 = 3 600 000 000 — это количество произведенных шагов за указанный промежуток).

Создайте таблицу, где будет представлен размер наибольшей задачи *N*, ко- торую сможет выполнить компьютер для каждой из функций, приведенных в таблице 1.1, за секунду, минуту, час, день, неделю, год.

1. Иногда константы, которыми пренебрегают в асимптотическом обозначе- нии, играют очень важную роль. Представьте, что два алгоритма выполня- ют одинаковую задачу. Для работы первого требуется 1500*N* шагов, второй справляется с тем же заданием за 30*N*2 шагов. При каких значениях *N* вы от- дадите предпочтение тому или иному алгоритму?

4\*. У вас есть два алгоритма. Один состоит из *N*3/75 – *N*2/4 + *N* + 10 шагов, дру- гой — из *N*/2 + 8 шагов. Каково должно быть значение *N*, чтобы вы выбрали первый/второй алгоритм?

1. Программа берет в качестве входных параметров *N* букв и генерирует из них все возможные пары. Например, из букв *ABCD* выстраиваются следующие комбинации: *AB*, *AC*, *AD*, *BC*, *BD* и *CD* (подразумевается, что *AB* и *BA* — одна и та же пара). Каково время работы алгоритма?
2. Алгоритм с числом входных данных *N* генерирует величины для каждого единичного квадрата на поверхности куба *N**N**N*. Найдите время работы алгоритма.
3. У вас есть алгоритм с числом входных данных *N*, который генерирует вели- чины для каждого элементарного куба в составе куба *N**N**N* (рис. 1.3). Ка- ково время работы алгоритма?



*N* = 3 

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

*N* = 4

*N* = 5

|  |  |
| --- | --- |
|  |  |
|  |  |

**Рис. 1.3.** Сгенерированные кубы

8\*. Алгоритм получает *N* входных данных и генерирует значение для каждо- го элементарного куба в пределах определенных фигур (рис. 1.4), добавляя еще один уровень при увеличении исходного параметра. Найдите время ра- боты алгоритма (учтите, что некоторые кубы присутствуют на заднем пла- не, но на изображении не видны).

*N* = 1

*N* = 2

*N* = 3

*N* = 4

**Рис. 1.4.** Сгенерированные фигуры

1. Как вы думаете, может ли быть алгоритм без структуры данных? Возможно ли существование структуры данных без алгоритма?
2. Рассмотрим два алгоритма покраски забора.

Algorithm1()

For i = 0 To <количество досок в заборе> - 1

<Красим доску под номером i.>

Next i

End Algorithm1

Algorithm2(Integer: first\_board, Integer: last\_board) If (first\_board == last\_board) Then

// Имеется только одна доска. Красим только ее.

<Красим доску под номером first\_board.>

Else

End If

// Досок больше одной, делим их на две группы и красим рекурсивно. Integer: middle\_board = (first\_board + last\_board) / 2 Algorithm2(first\_board, middle\_board) Algorithm2(middle\_board, last\_board)

End Algorithm2

Каково время работы этих алгоритмов (*N* — количество досок в заборе)? Какой вариант лучше?

11\*. Числа Фибоначчи можно определить рекурсивно с помощью следующих правил.

Fibonacci(0) = 1

Fibonacci(1) = 1

Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)

Последовательность Фибоначчи начинается с чисел 1, 1, 2, 3, 5, 8, 13, 21, 34,

55, 89.

Каким образом можно сравнить функцию Фибоначчи с функциями време- ни работы алгоритма на рисунке 1.2?

# Глава 2 ЧИСЛЕННЫЕ АЛГОРИТМЫ

Численные алгоритмы используются для работы с числами. Они распола- гают величины в случайном порядке, разбивают числа на простые множители, находят наибольший общий делитель и определяют геометрические площади. И хотя такие алгоритмы применимы лишь в некоторых случаях, лежащие в их основе технологии очень полезны, например адаптивные алгоритмы, моделиро- вание по методу Монте-Карло или использование таблиц для хранения проме- жуточных результатов.

## Рандомизация данных

Рандомизация играет важную роль во многих приложениях. Она позволяет программе моделировать некоторые процессы, проверять поведение алгоритмов при работе со случайными входными величинами, а также искать ответы для слож- ных задач. Интеграция Монте-Карло, описанная в разделе «Численное интегриро- вание» текущей главы, использует случайно выбранные точки для оценки площа- ди сложной геометрической фигуры.

Первый шаг в любом алгоритме рандомизации — генерирование случайных чисел.

#### Генерирование случайных величин

В действительности ни один алгоритм, используемый компьютером для полу- чения чисел, не является «случайным». Если бы вы были знакомы с его внутренней структурой, то наверняка смогли бы предвидеть те значения, которые он генери- рует. Чтобы добиться абсолютной непредсказуемости, необходимо использовать нечто отличное от компьютерной программы. Одним из таких вариантов мог бы служить детектор излучения, измеряющий частицы, которые испускает радио- активный образец. Поскольку нельзя точно предсказать, когда появятся частицы, генерирование чисел на самом деле будет происходить случайно. Другими воз- можными источниками так называемой хаотичности выступают игральные кости, анализ статического электричества в радиоволнах, а также броуновское движение. Сайт random.org, например, позволяет генерировать случайные величины на основе измерений атмосферного шума.

К сожалению, генераторы истинно случайных чисел являются довольно слож- ными и медленными, поэтому в большинстве задач используется более быстрый генератор псевдослучайных чисел (ГПСЧ). Если подобранные с его помощью ве- личины в определенном смысле достаточно «случайны», на их основе программа демонстрирует хорошие результаты.

#### Генерирование величин

Примером простого и общего метода создания псевдослучайных чисел является *линейный конгруэнтный генератор*, использующий следующую зависимость для формирования величин:

*Xn* + 1 = (*A*  *Xn* + *B*) mod *M*, где *A*, *B*, и *M* — постоянные.

Величина *X*0 называется *начальным числом*. Она инициализирует генератор, по- этому различные величины *X*0 дают неодинаковые комбинации чисел.

Поскольку все значения в числовой последовательности берутся по модулю *M*, как только генератор достигает максимума, он производит число, полученное им ранее, а последовательность чисел повторяется с того места.

Рассмотрим пример. Дано *A* = 7, *B* = 5 и *M* = 11. Если начать с *X*0 = 0, предыдущее уравнение приведет к следующей последовательности чисел.

*X*0 = 0

*X*1 = (7  0 + 5) mod 11 = 5

*X*2 = (7  5 + 5) mod 11 = 40 mod 11 = 7

*X*3 = (7  7 + 5) mod 11 = 54 mod 11 = 10

*X*4 = (7  10 + 5) mod 11 = 75 mod 11 = 9

*X*5 = (7  9 + 5) mod 11 = 68 mod 11 = 2

*X*6 = (7  2 + 5) mod 11 = 19 mod 11 = 8

*X*7 = (7  8 + 5) mod 11 = 61 mod 11 = 6

*X*8 = (7  6 + 5) mod 11 = 47 mod 11 = 3

*X*9 = (7  3 + 5) mod 11 = 26 mod 11 = 4

*X*10 = (7  4 + 5) mod 11 = 33 mod 11 = 0

Поскольку *X*10 = *X*0 = 0, последовательность повторяется.

Величины 0, 5, 7, 10, 9, 2, 8, 6, 3, 4 выглядят относительно случайными, но те- перь, когда известен используемый в программе метод, зная текущее число, вы сможете правильно спрогнозировать все последующие.

Некоторые алгоритмы ГПСЧ используют множественные линейные конгруэнт- ные генераторы с разными постоянными и производят выборку из величин, сгене- рированных на каждом шаге. Это делается для того, чтобы полученные числа ка- зались «более случайными», а период повторения последовательности увеличил- ся. Однако и такие методы не являются истинно случайными.

**ЗАМЕЧАНИЕ**

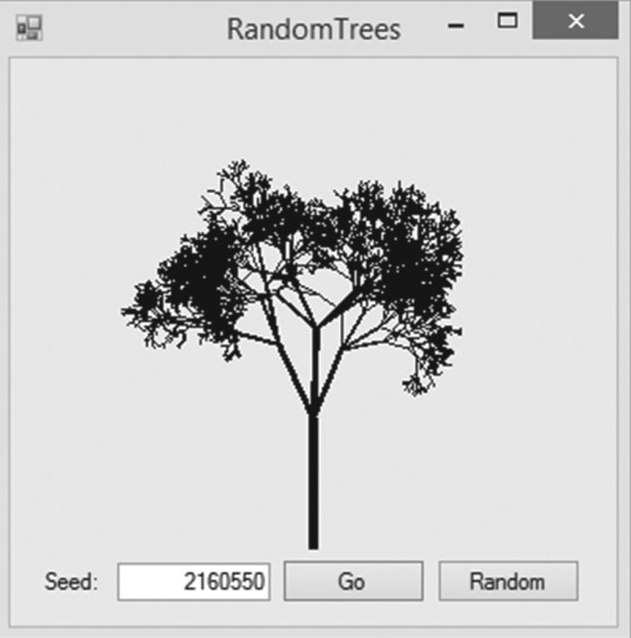
 Большинство языков программирования обладает встроенными ГПСЧ. Эти генераторы в основном довольно быстрые и производят очень длинные по-

следовательности чисел, прежде чем повториться. По такой причине их удобно использовать вместо того, чтобы писать собственные.

Одной из особенностей ГПСЧ является то, что можно воспользоваться определенным начальным числом для формирования такой же последователь- ности «случайных» чисел несколько раз. Это легко расценить как недостаток, поскольку означает, что числа более предсказуемы, однако возможность ис- пользовать одинаковые числа несколько раз значительно облегчает отладку некоторых программ.

Способность повторять последовательности чисел также разрешает неко- торым приложениям хранить сложные данные в очень сжатой форме. Пред- положим, программе нужно заставить объект пройти долгий и сложный псев- дослучайный путь по карте. Программа может сгенерировать путь и сохранить его координаты, чтобы позже перерисовать маршрут, или же оставить в памяти только значение начального числа, чтобы на его основе провести повторную инициализацию ГПСЧ.

На рисунке 2.1 представлено окно специальной программы RandomTrees, которая рисует дерево, используя заданное исходное значение. Введите произ- вольное число в окно Seed и нажмите кнопку Go, чтобы увидеть результат. Если начальные значения будут отличаться хотя бы на единицу, деревья получатся абсолютно разными.



**Рис. 2.1.** Окно программы RandomTrees

Вводимое вами число программа RandomTrees использует для генерации таких параметров, как количество ветвей, формируемых деревом при каждом шаге, угол их наклона относительно материнской ветви и отличие от нее в размерах. Если вы зададите одно и то же число два раза подряд, то получите два одинаковых дерева. (Программу можно скачать с одного из сайтов с материалами книги, которые упо- мянуты во введении.)

**КРИПТОСТОЙКИЕ ГПСЧ**

Любой линейный конгруэнтный генератор имеет период повторения, что дела- ет его бесполезным для криптографии. Например, вы пытаетесь зашифровать текст с использованием ГПСЧ и для этого генерируете случайные величины, а затем добавляете их к каждой букве в сообщении. Допустим, буква *А* + 3 бу- дет *D*, поскольку *D* находится через три позиции от *А* в английском алфавите. Если вы доберетесь до *Z*, то вернетесь к *А*. Так, *Y* + 3 = *B*.

Эта технология довольно хорошо работает, пока последовательность чисел случайна, однако линейный конгруэнтный генератор имеет ограниченное чис- ло начальных чисел. Все, что вам нужно, чтобы взломать код, — попробовать дешифровать сообщение с помощью любого начального числа. С целью де- шифровки программа оценивает распределение букв, чтобы увидеть, выглядит ли результат как реальный текст. Если начальное число выбрано неправильно, каждая буква должна появляться с приблизительно одинаковой частотой. Если начальное число правильное, одни буквы, допустим, *E* и *T*, будут появляться намного чаще, чем другие, например *J* и *X*. Если буквы распределены неравно- мерно, возможно, вы угадали начальное число.

Может показаться, что такая задача займет много времени, однако для совре- менных компьютеров это не так уж и сложно. Если начальное число – 32-битное целое, возможно лишь 4 млн вариантов. Современный компьютер может про- верить каждое начальное число за несколько минут или даже секунд.

*Криптостойкий генератор псевдослучайных чисел* (КГПСЧ) использует бо- лее сложные алгоритмы для генерации чисел, которые труднее спрогнози- ровать, а также для производства более длинных последовательностей без введения цикла. Они обычно обладают большими начальными значениями. Простой КГПСЧ может использовать 32-битное начальное число. КГПСЧ могут в состоянии работать с ключами длиной до 1000 бит для инициали- зации алгоритма.

КГПСЧ интересны и довольно «случайны», однако у них есть некоторые недостатки. Они сложнее и поэтому медленнее простых алгоритмов. Они также могут не позволить совершить всю инициализацию вручную, так что, возможно, вам не удастся сгенерировать повторяемую последова- тельность. Если вы хотите использовать ту же самую последовательность более одного раза, вам нужно задействовать простые ГПСЧ. К счастью, многие алгоритмы не нуждаются в КГПЧ, поэтому можно работать с более простыми алгоритмами.

#### Обеспечение равноправия

Обычно в программах нужно использовать равноправные ГПСЧ. *Равноправ- ным* ГПСЧ называется тот, который производит все возможные выходные данные с одинаковой вероятностью. ГПСЧ, являющийся неравноправным, называют *не- правильным*. Например, случай с монетой, падающей орлом в 2/3 случаев, являет- ся неправильным.

Многие языки программирования располагают методами, производящими слу- чайные числа в любых желаемых пределах, но если требуется написать код для пре- образования величин ГПСЧ в особый диапазон, нужно быть осторожным при со- блюдении равноправия.

Линейный конгруэнтный генератор производит число между 0 (включитель- но) и *M* (исключительно), где *M* — модуль, используемый в уравнении генератора *Xn* + 1 = (*A*  *Xn* + *B*) mod *M.*

Для программы, как правило, нужно произвольное число, находящееся за пределами диапазона от 0 до *M*. Простым, но неправильным способом отобра- жения числа, произведенного генератором в границах от min до max, является использование следующего уравнения: результат = min + число mod (max –

* min + 1). Например, чтобы получить величину между 1 и 100, вам необходимо произвести вот такой расчет: результат = 1 + число mod (100 – 1 + 1). Проблема заключается в том, что некоторые результаты могут быть куда более вероятны- ми, чем другие.

Чтобы понять, почему это так, рассмотрим пример, где *M* = 3, min = 0 и max = 1. Если генератор работает правильно, он производит числа 0, 1 и 2 с приблизительно одинаковой вероятностью. Если подставить эти значения в предыдущее уравнение, получатся величины, представленные в таблице 2.1.

**Òàáëèöà 2.1.** Ïîêàçàíèÿ ÃÏÑ×

|  |  |
| --- | --- |
| **Вводимое значение** | **Результат** |
| 0 | 0 |
| 1 | 1 |
| 2 | 0 |

Число 0 возникает в два раза чаще, чем 1, так что финальный результат явля- ется неправильным. В настоящих ГПСЧ, где модуль *M* очень большой, проблема не столь очевидна, но тем не менее существует.

Лучше всего конвертировать произведенную ГПСЧ величину в дробь между 0 и 1, а затем умножить ее на желаемый диапазон, как в следующей формуле: резуль- тат = min + (число  *M*)  (max – min).

Методом преобразования псевдослучайной величины из одного диапазона в другой является и просто игнорирование любых результатов, выпадающих за желаемые пределы. В предыдущем примере вы могли воспользоваться ограни- ченным ГПСЧ для генерирования величины между 0 и 2. Если получается 2, на- ходящееся за границами желаемого диапазона, его не учитывают и рассчитывают другое число.

Приведем пример, приближенный к реальности. Предположим, вы хотите дать печенье одному из четырех друзей и у вас есть игральная кость. Вам стоит просто несколько раз бросить кость, пока не получится величина между 1 и 4.

#### Получение равноправия от неправильных источников

Даже если ГПСЧ является неправильным, есть способ генерирования равно- правных чисел. Предположим, вы считаете монету неправильной. Вам сложно спрогнозировать вероятность того, выпадет у вас орел или решка, однако можно предположить, что эта вероятность не равна 0,5. Следующий алгоритм представляет подбрасывание правильной монеты.

Подбрасываем неправильную монету дважды.

Если результат представляет собой {Heads, Tails}, возвращаем Heads. Если результат представляет собой {Tails, Heads}, возвращаем Tails. Если результат выглядит как-то иначе, повторяем действие.

Чтобы понять, почему это работает, предположим, что для неправильной мо- неты вероятность выпадения орла равна *P*, а решки — 1 – *P*. Тогда вероятность выпадения орла, а затем решки рассчитывается как *P*  (1 – *P*), и наоборот, реш- ки, а затем орла как (1 – *P*)  *P*. Две указанные величины одинаковы, поэтому ве- роятность возвращения алгоритмом орла или решки тоже одинакова, а значит, результат равноправный.

Если неправильная монета выпадает орлом, а затем орлом или решкой, нужно повторить алгоритм. Возможно, придется проделать это много раз, прежде чем вы получите равноправный результат. Например, если *P* = 0,9, то орел будет выпадать два раза подряд в 81 % случаев, а в 1 % случаев два раза подряд выпадет решка.

**ВНИМАНИЕ**

Получение равноправного выпадения с помощью неправильной монеты вряд ли будет полезным в реальной программе. Тем не менее это хороший пример использования вероятностей, который может пригодиться вам на собеседо- вании при приеме на работу.

Подобную технологию можно использовать для расширения диапазона ГПСЧ. Предположим, вы хотите угостить печеньем одного из своих пяти друзей и вашим единственным источником случайности является правильная монета. В таком случае можно подбросить монету три раза и обработать результаты как двоичные числа с орлом для 1 и решкой для 0. Например, орел, решка, орел соответствует значению 101 в двоичной системе и 5 в десятичной. Если по- лучается результат, находящийся за пределами желаемого диапазона (в этом примере орел, орел, орел дает результат 111 в двоичной или 8 в десятичной системе исчисления, что больше количества друзей), вы сбрасываете результат и пробуете заново.

В заключение необходимо отметить, что для большинства случаев вам будет достаточно встроенного в язык программирования ГПСЧ. Если же вы нуждаетесь в большей степени случайности, можете обратить внимание на КГПСЧ. Использо- вание равноправной монеты с целью выбора случайного числа между 1 и 100 либо неправильного источника информации для генерирования равноправных чисел является более полезным при невыясненных обстоятельствах.

#### Рандомизация массивов

Довольно распространенной задачей в программах является рандомизация элементов массива. Допустим, программе календарного планирования нужно на- значить смены для работников организации. Если программа организует смены в алфавитном порядке по мере их появления в базе данных или в каком-либо другом статистическом порядке, сотрудники, которых постоянно записывают на ночную смену, будут недовольны.

В некоторых алгоритмах также может использоваться случайность с целью предотвращения наихудшей ситуации. Например, стандартный алгоритм быст- рой сортировки обычно работает хорошо, однако, если величины, которые он должен упорядочить, уже были отсортированы, производительность снижает- ся. Одним из способов выхода из положения будет рандомизация величин пе- ред их сортировкой.

В следующем алгоритме предлагается способ рандомизации массива.

RandomizeArray(String: array[])

Integer: max\_i = <верхняя граница массива> For i = 0 To max\_i - 1

// Выбираем элемент для i-й позиции в массиве.

Integer: j = <псевдослучайный номер между i и max\_i включительно>

<Меняем местами значения величин array[i] и array[j].>

Next i

End RandomizeArray

Представленный алгоритм проходит через каждую позицию в массиве один раз, так что время его работы равно *O(N)*, что является довольно быстрым для большей части приложений.

Стоит отметить, что повторение данного алгоритма не делает массив «более случайным». Когда вы тасуете карточную колоду, находящиеся рядом карты по преимуществу так и остаются вместе, так что вам нужно перетасовать колоду не- сколько раз, чтобы получить достаточно случайный результат. Текущий алгоритм полностью рандомизирует массив за одно прохождение, поэтому его повторный запуск является бесполезной тратой времени.

**РАВНОПРАВНО РАНДОМИЗИРОВАННЫЙ МАССИВ**

Другим важным вопросом, касающимся данного алгоритма, является такой: дает ли он равноправный порядок? Другими словами, есть ли вероятность того, что элемент, занимающий данную позицию, останется в ней? Напри- мер, плохо, если элемент, установленный в первую позицию, останется в ней в половине случаев.

Во введении я говорил, что в книге не будет длинных математических дока- зательств. При желании вы вправе пропустить следующее рассуждение и по- верить мне на слово — алгоритм рандомизации равноправен. Если вам из- вестно что-либо о вероятности, вполне возможно, что рассуждения вызо- вут интерес.

Для определенного элемента в массиве рассмотрим вероятность его разме- щения в позиции *k*. Чтобы занять ее, он не должен стать в позиции 1, 2, 3,… , *k* – 1, а затем в *k*.

Определим *P–i* как вероятность неразмещения элемента в позиции *i* при условии, что он не был ранее в позициях 1, 2, …, *i* – 1. Также определим *Pk* как вероятность размещения элемента в позиции *k* при условии, что он не был в позициях 1, 2, …, *k* – 1. Тогда общая вероятность того, что элемент займет позицию *k,* равняется *P*–1  *P*–2  *P*–3  ...  *P*–(*k* – 1)  *Pk*.

Если *P*1 равняется 1/*N*, то *P*–1 равняется 1 – *P*1 = 1 – 1 / *N* = (*N* – *1*)/*N*.

После того как задан первый элемент, элементы *N* – 1 могут быть поставлены в позицию 2, так что *P*2 = 1 / (*N* – 1), а *P*–2 = 1 – *P*2 = 1 – 1 / (*N* – 1) = (*N* – 2) / (*N* –1).

В общих чертах имеем: *P*i = 1 / (*N* – (*i* – 1)) и *P–i* = 1 – *Pi* = 1 – 1 / (*N* – (*i* – 1)) =

= (*N* – (*i* –1) – 1) / (*N* – (*i* – 1)) = (*N* – *i*) / (*N* – *i* + 1).

Если перемножить вероятности друг на друга, *P*–1  *P*–2  *P*–3  ...  *P*–(*k* – 1)  *Pk*

даст следующее уравнение:

*N — 1 × N — 2 × N — 3 × ... N — (k— 1) × N — 1*

*N N — 1 N — 2 N — (k — 1) + 1 N — (k — 1)*

Видно, что числитель каждого члена и знаменатель следующего члена со- кращаются. Когда вы совершите все сокращения, уравнение упростится до 1/*N*.

Это означает, что вероятность расположения элемента в позиции *k* являет- ся 1/*N* вне зависимости от значения *k*, так что размещение равноправно.

Еще одна задача, подобная рандомизации массива, — выбор определенного количества случайных элементов из массива без удвоения. Например, у вас есть список из 100 человек, которым вы хотите отправить экземпляр своей книги. Один из способов выбрать пять имен — это ввести всех претендентов в массив, рандо- мизировать его, а затем отправить книги первым пяти людям из сформированного списка. Вероятность того, что какое-либо имя находится в одной из пяти выигра- вших позиций, одинакова — следовательно, схема равноправна.

#### Генерирование неравномерных распределений

В некоторых программах нужно генерировать псевдослучайные числа, рас- пределяемые неравномерно. Нередко такие программы имитируют другие фор- мы генерации случайных чисел. Например, программе может понадобиться сгенерировать числа между 2 и 12, чтобы сымитировать бросок двух шести- гранных костей.

Вы не можете просто выбрать псевдослучайные числа между 2 и 12, поскольку вероятность получения каждого числа при бросании двух костей отсутствует. Для решения задачи нужно имитировать бросание костей, чтобы сгенерировать два числа между 1 и 6, а затем сложить полученные результаты.

## Нахождение наибольшего общего делителя

*Наибольший общий делитель* (НОД) двух целых чисел — наибольшее целое число, на которое они оба делятся без остатка. Например, НОД (60, 24) равен 12. Данная функция может показаться довольно странной, но на самом деле она очень полезна в криптографических задачах, которые широко используются в деловой сфере, в частности для защиты финансовых коммуникаций.

**ЗАМЕЧАНИЕ**

 Если НОД (*A*, *B*) = 1, говорят, что *A* и *B взаимно простые*.

В общем случае чтобы получить НОД, достаточно разложить два числа на мно- жители и найти общие коэффициенты. Однако еще около 300 г. до н. э. греческий математик Евклид в своем трактате «Начала» описал более быстрый способ, кото- рый представлен в следующем псевдокоде. Поскольку алгоритм создан на основе работы Евклида, он называется *евклидовым*, или *алгоритмом Евклида*.

Integer: GCD(Integer: A, Integer: B) While (B != 0)

Integer: remainder = A Mod B

// НОД(A, B) = НОД(B, остаток). A = B

B = remainder End While

Return A End GCD

В качестве примера рассмотрим нахождение НОД (4851, 3003). В таблице 2.2 приведены значения для *A*, *B* и *A* mod *B* для каждого шага.

**Òàáëèöà 2.2.** Çíà÷åíèÿ, èñïîëüçóåìûå äëÿ ðàñ÷åòà ÍÎÄ (4851, 3003)

|  |  |  |
| --- | --- | --- |
| ***A*** | ***B*** | ***A* mod *B*** |
| 4 851 | 3 003 | 1 848 |
| 3 003 | 1 848 | 1 155 |
| 1 848 | 1 155 | 693 |
| 1 155 | 693 | 462 |
| 693 | 462 | 231 |
| 462 | 231 | 0 |
| 231 | 0 |  |

При *B* = 0 переменная *A* и есть искомый НОД. В нашем случае это число 231. Проверить правильность результата можно следующим образом: 4851= 231  21 и 1848 = 231  8. Получается, что оба числа делятся на 231 без остатка, а величины 21 и 8 не имеют общих множителей.

**САМЫЙ БОЛЬШОЙ ИЗ ОБЩИХ ДЕЛИТЕЛЕЙ**

Дадим некоторое математическое пояснение, которое при желании вы може- те пропустить. Суть алгоритма Евклида сводится к тому, что НОД (*А*, *B*) = НОД (*B*, *A* mod *B*). Чтобы понять, почему это так, нужно знать, что представляет собой оператор деления по модулю. Предположим, остаток *R* = *A* mod *B*, тог- да *A* = *m*  *B* + *R*, где *m —* некоторое целое значение. Если *g* — это НОД для *A* и *B*, то *B* должно делиться на него без остатка, как и *m*  *B*. Аналогичным об- разом должно делиться и *А*, определяемое формулой *m*  *B* + *R*. Отсюда сле- дует, что раз *g* делит нацело *m*  *B*, то оно способно делить и *R*. Таким обра- зом, вышесказанное доказывает, что *g* делит без остатка *B* и *R*.

Чтобы считать *g* равным НОД (*B*, *R*), по-прежнему нужно знать, что оно явля- ется самым большим целым числом, на которое делятся *B* и *R*. Предположим, что есть некое значение *G*, на которое делятся *B* и *R* и которое больше, чем *g*.

Тогда *G* должно делить нацело *m*  *B* + *R*, то есть само *А*. Но это говорит о том, что *g* не является НОД (*А*, *B*) и противоречит нашему условию. Делаем вывод: такого *G* не существует, а *g* и есть НОД для *А* и *B*.

Приведенный алгоритм является довольно быстрым, поскольку величина *B* уменьшается как минимум на коэффициент 0,5 для каждых двух прохождений че- рез цикл While. Таким образом, максимальное время работы алгоритма составля- ет *O*(log (*B)*).

**ЖАЖДА СКОРОСТИ**

Попробуем разобраться, почему значение *B* в евклидовом алгоритме умень- шается как минимум на коэффициент 0,5 для каждых двух прохождений через

цикл While. Для этого обозначим с помощью *Ak*, *Bk* и *Rk* величины *A*, *B* и *R k*-й итерации и будем считать, что *A*1 = *m*1  *B*1 + *R*1 для целого числа *m*1, а во вто- рой итерации *A*2 = *B*1 и *B*2 = *R*1. Если *R*1  *B*1 / 2, тогда *B*2  *B*1 / 2.

Предположим, что *R*1 > *B*1 / 2 и в третьей итерации *A*3 = *B*2 = *R*1, а *B*3 = *R*2. По определению *R*2 = *A*2 mod *B*2, что аналогично записи *B*1 mod *R*1. Если до- пустить, что *R*1 > *B*1 / 2, тогда в *B*1 величина *R*1 укладывается ровно один раз, сохраняя в остатке *B*1 – *R*1.

Поскольку *R*1 > *B*1 / 2, то *B*1 – *R*1  *B*1 / 2. Теперь вернемся к уравнениям: *B*1 – *R*1 =

= *B*1 mod *R*1 = *A*2 mod *B*2 = *R*2 = *B*3. Следовательно, *B*3  *B*1 / 2, что и требовалось доказать.

## Возведение в степень

Иногда для решения задачи требуется возвести число в целую степень. Если степень низкая, это не сложно. Например, 73 легко вычислить путем простого умно- жения 7  7  7 = 343. Для более высоких значений, таких как 7102 187 291, данный способ является очень медленным.

**ЗАМЕЧАНИЕ**

 Возведение чисел в высокую степень нередко занимает довольно много времени. Этому факту можно было бы не придавать большого значения, если

бы данная математическая операция не использовалось в некоторых важных видах криптографии.

К счастью, есть способ ускорить процесс. Он основан на двух ключевых фор- мулах:

* + *A*2  *M* = (*AM*)2;
  + *AM* + *N* = *AM*  *AN*.

Первая позволяет быстро вычислить степень числа *A*, возводя в квадрат это же число в исходной степени; вторая помогает комбинировать степени любым удобным для вас образом.

Рассмотрим следующий псевдокод для возведения числа в степень.

// Возведим число А в степень Р.

Float: RaiseToPower(Float: A, Integer: P)

<Используем первую формулу, чтобы быстро вычислить A, A2, A4, A8 и т.д., пока не получим значение AN, где N + 1 > P.>

<Используем эти степени A и вторую формулу, чтобы вычислить AP.> Return AP

End RaiseToPower

Предположим, вы хотите с его помощью рассчитать значение для 76. Сначала алгоритм определит, сколько будет 71, 72 и 74. На этом он остановится, поскольку следующее значение степени (число 8) превысит нужное вам (число 6):

71 = 7;

72 = (71)2 = 72 = 49;

74 = (72)2 = 492 = 2401.

Затем алгоритму понадобится вторая ключевая формула, чтобы получить чис- ло 6 из уже имеющихся исходных степеней, возведенных в квадрат, то есть 6 = 2 + 4. Таким образом, 76 = 72  74 = 49  2401 = 117 649.

Чтобы получить искомое значение, нам понадобилось выполнить два умно- жения для расчета 72 и 74 и еще одно — для нахождения итогового произведения. В результате шагов получилось меньше, чем при обычном умножении 7  7  7   7  7  7, хотя в данном примере разница небольшая.

В общих чертах, имея экспоненту *P*, алгоритм рассчитывает степени log (*P*) для *A*. Затем он проверяет двоичные цифры *P*, чтобы увидеть, какие из полученных сте- пеней ему нужно перемножить для нахождения результата. (Если двоичная циф- ра *P* равна 1, конечный результат должен включать соответствующую степень 2. В предыдущем примере двоичное представление 6 равно 110. Таким образом, вклю- чены вторая и третья степени 2: 22 и 24.)

В двоичном исчислении величина *P* включает log2 *P* цифр. Следовательно, об- щее время работы составляет: *O*(log (*P*)) + *O*(log (*P*)) = *O*(log (*P*)). Даже если *P* = 1 000 000, значение log (*P*) получится около 20, поэтому алгоритм использует око-

ло 20 шагов (до 40 умножений), что намного меньше, чем 1 000 000.

Действия ограничивает лишь то, что величины, возведенные в высокие степени, возрастают до огромных значений. Даже такая «скромная» величина, как 7300, содержит 254 десятичные цифры. Это значит, что умножение больших чисел при расчете высоких степеней занимает много времени и места.

К счастью, чаще всего высокие степени вычисляются в криптографических алгоритмах, где все операции выполняются в абсолютном значении, которое огра- ничивает размер числа. Например, если в абсолютном значении число содержит 100 цифр, то в произведении двух таких чисел цифр может быть не более 200. Затем результат сводят с абсолютным значением, чтобы снова получить число длиной не более 100 цифр. При сведении каждого числа с абсолютным значением процесс вычисления замедляется, зато можно рассчитывать величины практически неограниченного размера.

## Работа с простыми числами

Как известно, *простое число* — это натуральное (целое положительное) число больше единицы, которое делится на единицу и на само себя. Все остальные на- туральные числа больше единицы являются *составными.*

Простые числа играют важную роль: они могут как облегчать, так и усложнять отдельные операции. Например, в некоторых видах криптографии, чтобы сделать алгоритм более надежным, используется произведение двух больших простых чисел.

Далее мы поговорим об обычных алгоритмах, работающих с такими числами.

#### Нахождение простых множителей

Самый легкий способ найти простые множители для числа — попытаться раз- делить его на все числа в промежутке между 2 и числом, которое меньше исходного значения на единицу. Если возможный множитель делит число нацело, сохраните его. Затем возьмите частное от деления и продолжите поиск с ним. Обратите вни- мание: перед тем как взять следующую цифру из ряда простых чисел, вы снова должны попробовать тот же самый множитель, поскольку исходное число может содержать более одного такого множителя.

Например, чтобы найти простые множители для числа 127, попробуйте раз- делить его на 2, 3, 4, 5 и т. д., пока не дойдете до 126.

Рассматриваемый алгоритм можно представить следующим псевдокодом.

List Of Integer: FindFactors(Integer: number) List Of Integer: factors

Integer: i = 2 While (i < number)

// Проверяем делимость на i. While (number Mod i == 0)

// i является множителем. Добавляем его в список. factors.Add(i)

// Делим число на i. number = number / i

End While

// Проверяем следующий возможный множитель. i = i + 1

End While

// Если от числа что-то осталось, остаток — тоже множитель. If (number > 1) Then factors.Add(number)

Return factors End FindFactors

Если число равно *N*, то время работы алгоритма составит *O*(*N*).

Чтобы усовершенствовать данный метод, воспользуемся тремя важными на- блюдениями.

* + Не стоит проверять, делится ли число на любое четное, кроме 2, поскольку четные числа уже сами по себе кратны двум. Это означает, что нужно рас- смотреть лишь делимость на 2 и на нечетные числа, вместо того, чтобы пе- ребирать все возможные множители. В таком случае время работы сокра- тится вдвое.
  + Следует проверять множители только до квадратного корня числа. Если *n* =

= *p*  *q*, то *p* или *q* должно быть меньше либо равно sqrt (*n*). (Если *p* и *q* боль- ше sqrt (*n*), их произведение превысит *n*.) Проверив возможные множители до sqrt (*n*), вы найдете наименьший среди них, а, поделив *n* на такой множи- тель, определите еще один. Это сократит время работы до *O*(sqrt (*n*)).

* + Всякий раз при делении числа на множитель вы можете обновить максималь- ное количество потенциальных множителей, которые необходимо проверить.

Усовершенствуем алгоритм согласно вышеизложенному.

List Of Integer: FindFactors(Integer: number) List Of Integer: factors

// Проверяем делимость на 2. While (number Mod 2 == 0)

factors.Add(2) number = number / 2

End While

// Ищем нечетные множители. Integer: i = 3

Integer: max\_factor = Sqrt(number) While (i <= max\_factor)

// Проверяем делимость на i. While (number Mod i == 0)

// i является множителем. Добавляем его в список. factors.Add(i

// Делим число на i. number = number / i

// Устанавливаем новую верхнюю границу. max\_factor = Sqrt(number)

End While

// Проверяем следующий возможный нечетный множитель. i = i + 2

End While

// Если от числа что-то осталось, остаток тоже множитель. If (number > 1) Then factors.Add(number)

Return factors End FindFactors

**ЗАМЕЧАНИЕ**

 Алгоритм, который разлагает число на простые множители, имеет время рабо- ты *O*(sqrt (*N*)) (где *N* — исходное число) и является относительно быстрым для

малых чисел. Для действительно больших *N* эта функция работает уже значи- тельно медленнее. Например, если *N* имеет длину 100 цифр, то sqrt (*N*) будет содержать их 50. При условии что *N* — простое число, даже быстрый компью- тер не сможет опробовать все возможные множители за разумный отрезок времени. Это поможет защитить некоторые алгоритмы шифрования.

#### Нахождение простых элементов

Предположим, вашей программе нужно выбрать большое простое число (по- добные задания не редкость в алгоритмах шифрования). Один из способов сделать это — использовать код, описанный ранее, и проверить, являются ли числа просты- ми. С разумно малыми значениями такой вариант сработает хорошо, но для очень больших чисел он может оказаться чересчур медленным.

Еще один метод нахождения простых чисел для указанного значения — *решето Эратосфена*. Он также хорош для достаточно малых величин, поскольку требует создания таблицы с записями для каждого анализируемого числа. Если значения будут слишком велики, понадобится неоправданно большой объем памяти.

Идея заключается в том, чтобы создать таблицу с одной записью для каждого числа между 2 и указанным пределом. Для начала вычеркните все четные числа, не считая 2. Затем просмотрите таблицу, чтобы найти следующее невычеркнутое число (в данном случае это 3), и исключите все значения, кратные ему (не считая самого числа). Обратите внимание: некоторые значения уже могут быть вычеркнутыми, поскольку делились на 2. Повторите те же действия со следующим невычеркну- тым числом и т. д., пока не дойдете до квадратного корня из верхнего предела. Все оставшиеся числа и будут искомыми простыми.

Продемонстрируем описанный выше алгоритм.

// Находим простые числа между 2 и max\_number (включительно). List Of Integer: FindPrimes(long max\_number)

// Определяем массив для чисел.

Boolean: is\_composite = new bool[max\_number + 1]

// Исключаем числа, кратные 2.

For i = 4 to max\_number Step 2 is\_composite[i] = true

Next i

// Исключаем числа, кратные найденным простым числам. Integer: next\_prime = 3

Integer: stop\_at = Sqrt(max\_number) While (next\_prime <= stop\_at)

// Исключаем числа, кратные данному простому числу. For i = next\_prime \* 2 To max\_number Step next\_prime Then is\_composite[i] = true

Next i

// Переходим к следующему простому числу, пропуская четные числа. next\_prime = next\_prime + 2

While (next\_prime <= max\_number) And (is\_composite[next\_prime]) next\_prime = next\_prime + 2 End While

End While

// Заносим простые числа в список. List Of Integer: primes

For i = 2 to max\_number

If (Not is\_composite[i]) Then primes.Add(i)

Next i

// Возвращаем простые числа. Return primes

End FindPrimes

Данный алгоритм обладает временем работы *O*(*N*  log (log *N*)), однако дока- зательство этого не входит в задачу книги.

#### Проверка на простоту

Алгоритм, который раскладывает числа на множители (см. подраздел «На- хождение простых элементов» раздела «Работа с простыми числами» в те- кущей главе) можно использовать и для определения простых чисел. Идея заключается в следующем: если алгоритм не выполнит свою задачу, значит, число простое.

Как уже говорилось, производительность данной программы хороша для отно- сительно малых чисел. Для числа, состоящего из 100 цифр, количество шагов алго- ритма будет уже 50-значным. Даже самые быстрые компьютеры не способны вы- полнять такой объем операций в разумный промежуток времени. Машине, совер- шающей 1 трлн операций в секунду, понадобится более чем 3  1030 лет. В силу это- го для некоторых алгоритмов шифрования, нуждающихся в использовании боль- ших чисел, рассмотренный метод проверки простых чисел не сработает. Но, к сча- стью, есть альтернатива, например *тест простоты Ферма*.

Малая теорема Ферма гласит: если *p* — простое число, а 1  *n* < *p*, то *np* – 1 mod *p* = 1. Другими словами, если вы возведете *n* в степень *p* – 1, а затем возьмете результат по модулю *p*, то ответом станет 1.

Предположим, что *p* = 11, а *n* = 2. Тогда *n p* – 1 mod *p* = 210 mod 11 = 1024 mod 11. В свою очередь 1024 = 11  93 + 1, следовательно, 1024 mod 11 = 1, что и требова- лось доказать.

Обратите внимание: *n p* – 1 mod *p* = 1 справедливо, даже если *p* не является про- стым числом. В таком случае величина *n* называется *обманщиком Ферма*, поскольку она ошибочно указывает на то, что число *p* — простое. Если же *n p* – 1 mod *p*  1, то *n* называют *свидетелем Ферма,* и эта величина показывает, что *p* не является про- стым числом.

Строго говоря, для натурального числа *p* минимум половина значений *n* меж- ду 1 и *p* — свидетели Ферма. Получается, если *p* — это не простое число и вы вы- бираете любое значение между 1 и *p*, то с вероятностью 50 % *n* окажется свидетелем Ферма и *np* – 1 mod *p*  1. Вам может не повезти, и вы возьмете в качестве *n* обманщика Ферма. Но если повторить тест многократно, то шансы выбрать свидетеля (если он существует) возрастут.

Если учесть, что в каждом тесте в половине случаев вы можете столкнуться со свидетелем Ферма и *p* проходит *k* тестов, то вероятность выбора обманщиков Ферма каждый раз будет равна 1/2*k*. Другими словами, существует 1/2*k* случаев, что *p* лишь кажется простым, но на самом деле является составным.

Например, *p* проходит тест 10 раз, тогда вероятность, что число не простое, составит 1/210 � 0,00098. Для большей уверенности повторите тест 100 раз, тогда значение вероятности изменится на 1/2100 � 7,8  10– 31.

Приведем псевдокод для реализации метода, принимающего решение о том, какое число (возможно) является простым.

// Возвращаем true, если число p (возможно) простое. Boolean: IsPrime(Integer: p, Integer: max\_tests)

// Проводим проверку max\_tests раз. For test = 1 To max\_tests

<Выбираем случайное число n между 1 и p (эксклюзивно).> If (np-1 Mod p != 1) Then Return false

Next test

// Возможно, число простое.

// С вероятностью 1/2 max\_tests число не является простым. Return true

End IsPrime

**ЗАМЕЧАНИЕ**

 Приведенный выше алгоритм называется *вероятностным*, поскольку дает правильный результат с определенной вероятностью. При работе с ним

следует учитывать тот факт, что он нередко оказывается неправильным. Од- нако вы можете повторять тесты до тех пор, пока не достигнете желаемого уровня уверенности.

Если число *p* очень большое (это единственный по-настоящему интересный случай), то расчет *np* – 1 с использованием умножения может занять много времени. К счастью, вы знаете, как ускорить работу алгоритма, имея дело с возведением в степень (см. раздел «Возведение в степень» текущей главы).

Научившись определять, является ли число (возможно) простым, вы можете составить алгоритм для выбора простых чисел.

// Возвращаем (возможно) простое число а длиной max\_digits. Integer: FindPrime(Integer: num\_digits, Integer: max\_tests)

Repeat

<Выбираем случайное число p с num\_digits digits.> If (IsPrime(p, max\_tests)) Then Return p

End FindPrime

## Численное интегрирование

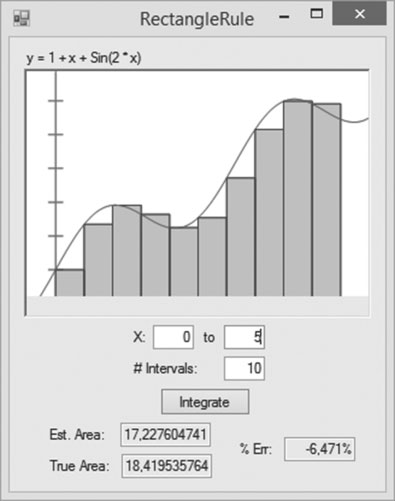
*Численное интегрирование* (*квадратура*, или *числовая квадратура*) — использо- вание численных методов для аппроксимации области под кривой, что определяется некоторой функцией. Чаще всего это функция одной переменной *y* = *F*(*x*), которая образует двумерную область. В некоторых случаях область может быть трехмерной и определяться функцией *z* = *F*(*x*, *y*). Допустимы и более многомерные варианты.

Чем проще функция, тем точнее расчет занимаемой области. Впрочем, есть ве- роятность, что вам не удастся найти для нее неопределенный интеграл. Например, слишком сложное или неизвестное уравнение, в случае если данные получены от

конкретного физического процесса. Тогда вместо расчетов лучше воспользовать- ся численным интегрированием. Оно выполняется несколькими способами. Са- мый простой из них предполагает работу с формулой Ньютона — Котеса, где ап- проксимация функции происходит с помощью нескольких многочленов. Частными случаями этого способа являются *формула прямоугольников* и *формула трапеций*.

#### Формула прямоугольников

В этом способе для аппроксимации области под кривой используется несколько прямоугольников одинаковой ширины. На рисунке 2.2 показано окно программы RectangleRule (ее легко найти на сайте с материалами книги), работающей по ука- занной формуле. Кроме того, вы можете видеть расчет точной области под кривой и сравнить, насколько далек приближенный результат от правильного.



**Рис. 2.2.** Окно программы RectangleRule (аппроксимация области под кривой для функции *y* = 1 + *x* + sin2*x*)

Ниже представлен псевдокод с алгоритмом, реализующим формулу прямо- угольников.

Float: UseRectangleRule(Float: function(), Float: xmin, Float: xmax, Integer: num\_intervals)

// Вычисляем ширину прямоугольника. Float: dx = (xmax - xmin) / num\_intervals

// Добавляем области прямоугольников. Float: total\_area = 0

Float: x = xmin

For i = 1 To num\_intervals

total\_area = total\_area + dx \* function(x) x = x + dx

Next i

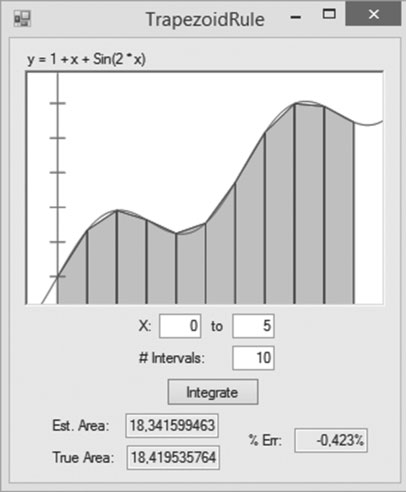
Return total\_area End UseRectangleRule

Этот алгоритм просто делит область на прямоугольники постоянной ширины и высотой, равной величине функции с левого ребра прямоугольника. Затем он описывает петлю над прямоугольниками, добавляя их области.

#### Формула трапеций

На рисунке 2.2 видно, что прямоугольники не подходят к кривой вплотную, за счет чего появляется погрешность в расчетах области. Ошибку можно уменьшить, если использовать более узкие прямоугольники. Так, в вышеприведенном приме- ре увеличение количества прямоугольников с 10 до 20 снизит погрешность с 6,5 до 3,1 %.

Альтернативная стратегия для аппроксимации кривой — заменить прямо- угольники трапециями. На рисунке 2.3 показано окно программы TrapezoidRule, работающей по этому принципу (вы можете скачать ее с сайта с материалами книги).



**Рис. 2.3.** Окно программы TrapezoidRule

Следующий псевдокод представляет алгоритм применения формулы трапеций.

Float: UseTrapezoidRule(Float: function(), Float: xmin, Float: xmax, Integer: num\_intervals)

// Вычисляем ширину трапеции.

Float: dx = (xmax - xmin) / num\_intervals

// Добавляем области трапеций. Float: total\_area = 0

Float: x = xmin

For i = 1 To num\_intervals

total\_area = total\_area + dx \* (function(x) + function(x + dx))/2

x = x + dx

Next i

Return total\_area End UseTrapezoidRule

Единственная существенная разница между этим алгоритмом и тем, который работает с формулой прямоугольников, заключается в операторе, добавляющем область к каждой части. В данном случае используется формула для построения трапеций: область = ширина  среднее длин параллельных сторон.

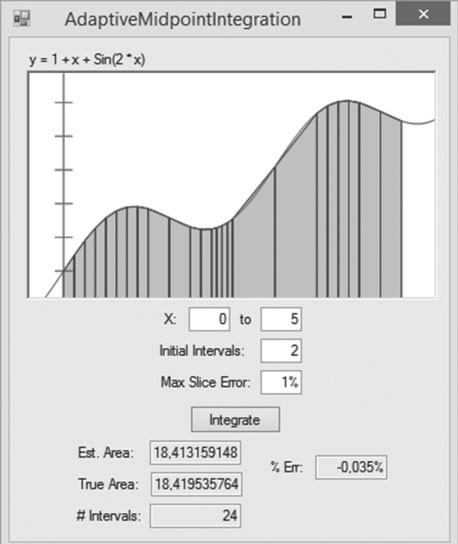
Вы можете рассматривать правило прямоугольников как построение кривой с помощью ступенчатой функции, перепрыгивающей с одной величины на другую на каждом верхнем ребре прямоугольника, а правило трапеций — как выстраивание кривой посредством ломаных линий.

Еще одним примером формулы Ньютона — Котеса является *формула Симпсона*, использующая для аппроксимации кривой многочлены второй степени. В других методах улучшения результата добиваются с помощью многочленов большей степени.

#### Адаптивная квадратура

Такая разновидность численной интеграции обнаруживает области, в которых аппроксимация имеет наибольшую погрешность, и совершенствует используемый для работы метод. Еще раз взгляните на рисунок 2.3: там, где кривая приближается к прямой линии, трапеции подходят к ней очень близко, но там, где она резко из- гибается, трапеции отстоят от нее. Программа, использующая адаптивную квадра- туру, ищет области, где фигуры не примыкают вплотную к кривой, и увеличивает их количество на этих участках.

На рисунке 2.4 показано окно программы AdaptiveMidpointIntegration, ра- ботающей на основе формулы трапеций с адаптивной квадратурой. Ее суть за- ключается в следующем: часть аппроксимирующей области вначале заполняется одной трапецией, а затем разделяется на две меньшие фигуры. Если разница между областью, занимаемой большой трапецией, и суммой областей меньших трапеций превышает определенный процент, программа выбирает разбиение и переходит к аналогичному расчету каждой отдельной области.



**Рис. 2.4.** Окно программы AdaptiveMidpointIntegration

Описанный алгоритм представляет следующий псевдокод.

// Интегрируем, используя адаптивную формулу трапеции. Float: IntegrateAdaptiveMidpoint(Float: function(),

Float: xmin, Float: xmax, Integer: num\_intervals, Float: max\_slice\_error)

// Вычисляем ширину начальных трапеций. Float: dx = (xmax - xmin) / num\_intervals double total = 0

// Добавляем области трапеций. Float: total\_area = 0

Float: x = xmin

For i = 1 To num\_intervals

// Добавляем кусочек области. total\_area = total\_area +

SliceArea(function, x, x + dx, max\_slice\_error) x = x + dx

Next i

Return total\_area End IntegrateAdaptiveMidpoint

// Возвращаем площадь данной области.

Float: SliceArea(Float: function(),Float: x1, Float: x2, Float: max\_slice\_error)

// Вычисляем значение функции на конечных и центральной точках. Float: y1 = function(x1)

Float: y2 = function(x2) Float: xm = (x1 + x2) / 2 Float: ym = function(xm)

// Вычисляем площадь большой трапеции и двух меньших трапеций. Float: area12 = (x2 - x1) \* (y1 + y2) / 2.0

Float: area1m = (xm - x1) \* (y1 + ym) / 2.0 Float: aream2 = (x2 - xm) \* (ym + y2) / 2.0 Float: area1m2 = area1m + aream2

// Оцениваем, насколько мы близко.

Float: error = (area1m2 - area12) / area12

// Оцениваем погрешность.

If (Abs(error) < max\_slice\_error) Then Return area1m2

// Погрешность слишком большая. Делим трапецию и пробуем еще раз. Return

SliceArea(function, x1, xm, max\_slice\_error) + SliceArea(function, xm, x2, max\_slice\_error)

End SliceArea

Если запустить программу AdaptiveMidpointIntegration и выполнить после- довательное разделение области, начиная с 2 частей до 24 (см. рис. 2.4), а затем оценить погрешность, то она составит всего 0,035 %. Если же вы воспользуетесь программой TrapezoidRule, где все 24 части будут равной ширины, то вероятность ошибки окажется примерно в два раза больше — 0,072 %. Таким образом, при равном количестве частей адаптивный алгоритм более эффективен.

В программе AdaptiveTrapezoidIntegration решение о том, когда следует раз- бивать область на подобласти, принимается на основе особого метода. Сперва вычисляется вторая производная функции в начальной точке *x* на отрезке, затем интервал разбивается на одну часть плюс одно значение производной в секунду. Например, если вторая производная равна 2, то программа делит интервал на три части (формула была выбрана произвольно, для улучшения результатов вы можете использовать другую).

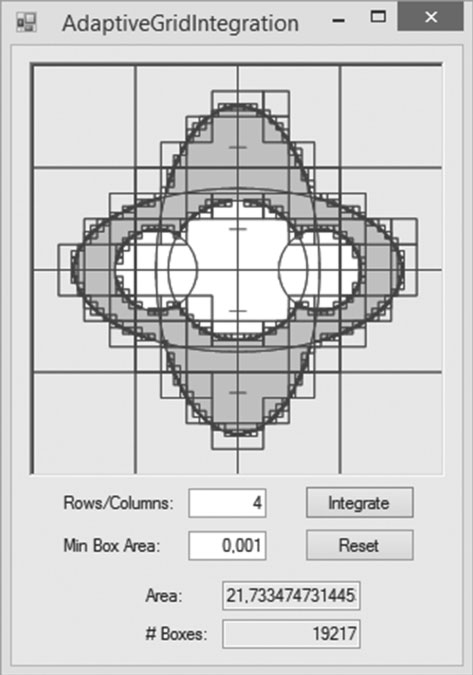
**ЗАМЕЧАНИЕ**

 Для случая когда все расчеты выполняются приблизительно, производная функции сообщает о своем отклонении в каждой точке. Вторая производная

говорит о степени искривления кривой и о том, насколько быстро это проис- ходит. Производные более высоких степеней означают, что кривая изгибается еще сильнее, поэтому программа AdaptiveTrapezoidIntegration разбивает об- ласть на большее количество частей.

Описанная методика не сработает, если вы не сможете рассчитать вто- рую производную для кривой. А технология, используемая в программе AdaptiveMidpointIntegration, должна показать хорошие результаты в любом случае, так что вам следует прибегнуть к ней.

Адаптивные технологии применяются во многих алгоритмах, поскольку позволяют получить отличные результаты без лишних усилий. В программе AdaptiveGridIntegration (рис. 2.5) с их помощью рассчитывается площадь закрашен- ной области, которая образуется из двух перпендикулярных друг другу эллипсов (за вычетом трех внутренних кругов).



**Рис. 2.5.** Окно программы AdaptiveGridIntegration

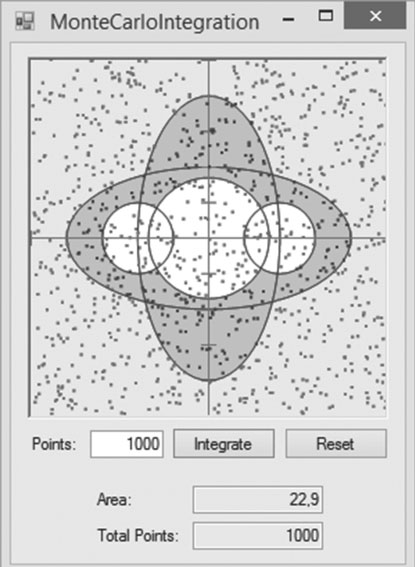
Программа заключает сложную фигуру в рамку и разбивает на сегменты с по- мощью решетки. На рисунке 2.5 решетка выстроена из четырех вертикальных и четырех горизонтальных рядов. Для каждого сегмента программа определяет, располагается ли он внутри или снаружи закрашенной области. Так, если ни одна из точек сегмента не входит в закрашенную область, весь сегмент игнорируется. В случае когда в закрашенную область попадает каждая точка, программа добав- ляет площадь сегмента к площади оцениваемой области. При условии что внутри заштрихованной фигуры находятся лишь некоторые точки, программа разбивает исходный сегмент на более мелкие части и использует ту же технологию для вы- числения остальной площади фигуры.

Взглянув на рисунок 2.5 еще раз, вы увидите, каким образом программа AdaptiveGridIntegration разделила фигуру на сегменты. Легко заметить, что коли- чество этих сегментов по краям закрашенной области намного больше, чем внутри или снаружи. Всего же их получилось 19 217.

#### Интеграция Монте-Карло

Это еще одна разновидность численной интеграции, при которой програм- ма генерирует множество равномерно распределенных псевдослучайных точек, определяет, какие из них находятся внутри целевой области, а затем оценивает искомую площадь.

Предположим, точки генерируются внутри квадрата 2020, площадь которо- го составляет 400 квадратных единиц. При условии что 37 % псевдослучайных точек попадут внутрь искомой области, площадь нужной нам фигуры определит- ся по формуле 0,37  400 = 148 квадратных единиц. В программе MonteCarlo- Integration (рис. 2.6) рассчитывается площадь той же фигуры, что и в программе AdaptiveGridIntegration.



**Рис. 2.6.** Окно программы MonteCarloIntegration (точки внутри закрашенной области обозначены черным цветом, за ее пределами — серым)

Интеграция Моне-Карло менее совершенна, чем интеграция трапеции или адап- тивная интеграция. Тем не менее иногда проще использовать именно ее, поскольку здесь не требуется, чтобы вы хорошо представляли фигуру, площадь которой со- бираетесь рассчитывать. Достаточно сгенерировать точки и посмотреть, сколько из них попадает в нужную вам область.

**ЗАМЕЧАНИЕ**

 В данной главе псевдослучайные величины используются для расчета площа- дей, однако подобные технологии применимы и для решения других задач.

В *моделировании по методу Монте-Карло* выбранные псевдослучайные ве- личины помогают вычислить процент, который удовлетворяет определенному критерию, и на его основе спрогнозировать общее количество подходящих величин.

## Нахождение нулей

Иногда программе нужно выяснить, где уравнение пересекает ось *x*. Другими словами, имея уравнение *y* = *f*(*x*), требуется найти *x*, при котором *f*(*x*) = 0. Такие величины называются *корнями* уравнения.

*Метод Ньютона* (иногда его называют *методом Ньютона — Рафсона*) позволяет определить корни уравнения путем последовательной аппроксимации. Первым

делом выбирается некое начальное приближение *X*0 к корню. Если функция *f*(*X*0) оказывается недостаточно близка к 0, алгоритм движется по касательной, выстро- енной из точки начального приближения, и находит пересечение с осью *x*. Эта новая

точка и будет следующим предположительным приближением *X*1 для корня. Так продолжается до тех пор, пока не отыщется величина *Xk*, при которой функция *f*(*Xk*) будет достаточно близка к 0.

Единственная сложность в понимании этого метода связана с тем, как заставить алгоритм двигаться по касательной. Если у вас есть формула для нахождения про- изводной функции *f*'(*x*), записываемой также как *dfdx*(*x*), то благодаря следующему уравнению программа сможет обновить приближение для корня:

*X* = *X* – *f*(*Xi*) .

**ЗАМЕЧАНИЕ**

*i+*1 *i f'*(*Xi*)

В рамках данной книги нахождение производных функций не рассматривает- ся. Если вам нужна дополнительная информация, поищите ее в Интернете или учебниках по математике.

На рисунке 2.7 описанный выше процесс представлен графически. Цифрой 1 обозначена точка, соответствующая начальному приближению для корня. Значение функции *y* в ней далеко от 0, поэтому алгоритм следует вдоль касательной, пока та не пересечет ось *x*. Затем он рассчитывает функцию для нового приближения и получает точку 2, в которой значение функции *y* также далеко от 0. Весь процесс повторяется и находится точка 3, а после 4. У точки 4 ордината достаточно близка к 0, следовательно, алгоритм останавливается.

1

Ось *X*

4

3

2

**Рис. 2.7.** Метод Ньютона: корни функции определяются путем следования вдоль касательных, проведенных к графику

Работу рассмотренного алгоритма можно представить следующим псевдокодом.

// Используем метод Ньютона для определения корня функции f(x). Float: FindZero(Float: f(), Float: dfdx(), Float: initial\_guess,

Float: maxError)

float x = initial\_guess

// Останавливаемся после 100 повторений, если что-то не так.

// Вычисляем значение в данной точке. float y = f(x)

// Если погрешность достаточно мала, останавливаемся. if (Math.Abs(y) < maxError) break

// Обновляем х.

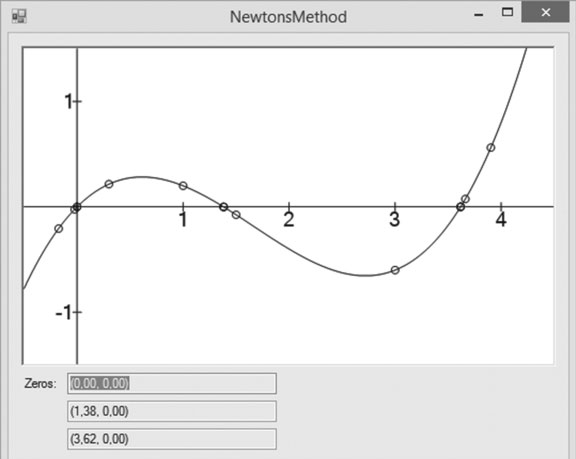
x = x  y / dfdx(x)

Next i

Return x End NewtonsMethod

В качестве исходных параметров алгоритм берет функцию *y* = *f*(*x*), ее про- изводную *dfdx*, начальное приближение для корня и допустимую погрешность. Переменная *x* приравнивается к начальному приближению и используется в цикле For, который может срабатывать до 100 раз. Обычно алгоритм быстро находит решение, но если функция имеет совсем небольшую кривизну, алгоритм нередко отклоняется в сторону и «забывает» о решении или же «застревает», выбирая то одно, то другое начальное приближение. Предустановленное количество итераций позволит программе выйти из цикла.

В цикле For алгоритм рассчитывает функцию *f*(*x*) и, если результат недоста- точно близок к нулю, обновляет *x* для совершения следующей попытки. Обратите внимание, что некоторые функции имеют больше одного корня. В таком случае вам следует запустить алгоритм FindZero несколько раз, используя новое начальное приближение для обнаружения нового корня.



**Рис. 2.8.** Окно программы NewtonsMethod

На рисунке 2.8 показано окно программы NewtonsMethod (вы можете скачать ее на сайте с материалами книги), которая задействует метод Ньютона для поиска трех корней функции *y* = *x*3 / 5 – *x*2 + *x*. Точки на графике — это начальные при- ближения, выбранные программой.

## Резюме

По сравнению с рандомизацией данных такие численные алгоритмы, как раз- ложение на множители и нахождение наибольшего общего делителя, в чистом виде имеют довольно ограниченную сферу применения. Тем не менее используемые в них методы и идеи в отдельных ситуациях оказываются довольно ценными. На- пример, мысль о том, что алгоритм может быть вероятностным, пригодится для написания других алгоритмов, не работающих с отличной достоверностью (по- добный вопрос может встретиться и на собеседовании).

В этой главе вы получили представление об идее равноправия и предвзятости — двух очень важных понятиях для любых алгоритмов рандомизации, например для алгоритма интеграции Монте-Карло. Кроме того, узнали, что такое адаптивная квадратура, и познакомились с технологиями, позволяющими программе сосре- дотачиваться на более значимых моментах решения задачи и меньше обращать внимания на то, с чем легко справиться. Подобные программы, способные под- страиваться под обстоятельства, встречаются во многих алгоритмах.

Еще одна особенность численных алгоритмов (поиск НОД, тест простоты Фер- ма, интеграция Монте-Карло, формулы прямоугольников и трапеций) заключается в том, что они не нуждаются в сложных структурах данных. Однако большинство других программ, описанных в книге, все же потребуют специализированных струк- тур, таких как, например, связные списки. Именно о них пойдет речь в следующей главе. Это далеко не самый сложный способ организации данных среди тех, с ко- торыми вы познакомитесь, но, безусловно, очень полезный. Кроме того, понятие

«связные списки» пригодится вам и при работе с другими структурами, например с деревьями или сетями.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

* 1. Напишите алгоритм, в котором игральная кость будет генерировать под- брасывания монеты.
  2. В пункте «Получение равноправия от неправильных источников» (см. под- раздел «Генерирование случайных величин» раздела «Рандомизация дан- ных» в текущей главе) объяснялось, как дважды подкинутая неправильная монета помогает получить справедливое подбрасывание монеты. Но иногда такое действие не дает результата, и вам нужно повторить процесс. Предпо- ложим, монета выпадает орлом в 3/4 случаев и решкой в 1/4. Какова веро- ятность, что вы не получите нужный результат после двух подбрасываний и вам придется пробовать снова?
  3. Вернемся к монете из упражнения 2. Предположим, вы ошибались и монета на самом деле является правильной, но алгоритм все еще используется для получения равноправных подбрасываний неправильной монеты. Какова вероятность, что вы не получите результат после двух подбрасываний и вам придется попробовать снова?
  4. Разработайте алгоритм с использованием неправильной игральной кости для генерации равноправных величин между 1 и 6. Какова его эффективность?
  5. Напишите алгоритм для выбора случайных величин *M* из массива, содер- жащего *N* элементов (где *M*  *N*). Определите время его работы и приме- ните к примеру, где необходимо раздать книги пяти людям, выбранным из 100 (или 10 000) записей.
  6. Продумайте алгоритм для игры в покер, где каждый игрок получает по пять карт. Имеет ли значение, как раздаются карты: по одной каждому игроку (пока у каждого не окажется нужное количество) или все пять сразу?
  7. Напишите программу, которая имитирует бросание двух игральных костей и рисует гистограмму или график, показывающие, сколько раз выпала каждая величина. Сравните количество выпадений величин с ожидаемым распре- делением бросков двух правильных костей при том же количестве попыток. Сколько попыток вам понадобится осуществить, чтобы результаты совпали с ожидаемым распределением?
  8. Что случится с евклидовым алгоритмом, если *A* < *B*?
  9. Наименьшее общее кратное для *A* и *B* — это наименьшее целое число, на ко- торое *A* и *B* делятся без остатка. Как можно использовать НОД для расчета наименьшего общего кратного?
  10. Алгоритм быстрого возведения в степень, представленный в этой главе, яв- ляется высокоуровневым. Напишите низкоуровневый алгоритм.
  11. Как изменить алгоритм из предыдущего упражнения, чтобы применить бы- строе возведение в степень по модулю?

12\*. Напишите программу, которая рассчитывает НОД для нескольких пар псев- дослучайных чисел и отображает в виде графика количество шагов, требуе-

мых алгоритмом НОД в сравнении с алгоритмом нахождения среднего ариф- метического двух чисел. Выглядит ли результат логарифмическим?

13. Следующий псевдокод показывает, как решето Эратосфена вычеркивает чис- ла, кратные простому next\_prime.

// Вычеркиваем числа, кратные данному простому числу.

For i = next\_prime \* 2 To max\_number Step next\_prime Then is\_composite[i] = true

Next i

Первое вычеркнутое число — это next\_prime \* 2. Однако вы знаете, что оно уже исключено, поскольку кратно 2 (алгоритм первым делом вычеркнул такие числа). Как изменить цикл For, чтобы избежать повторного пересмо- тра исключенных чисел?

14\*. В бесконечном ряде составных чисел (так называемые числа Кармайкла) каж- дое относительно простое малое число является обманщиком Ферма. Дру- гими словами, *p* — число Кармайкла, если *p* — нечетное и каждое *n —* обман- щик Ферма, при условии что 1 < *n* < *p* и НОД (*p*, *n*) = 1. Напишите алгоритм со списком чисел Кармайкла от 1 до 10 000 и их простыми множителями.

1. При использовании формулы прямоугольников части некоторых фигур ока- зываются над или под кривой, увеличивая или уменьшая рассчитываемую площадь соответственно. Что произойдет, если для высоты прямоугольника вы возьмете значение функции из центра прямоугольника, а не из его левого ребра? Напишите программу для проверки своей гипотезы.
2. Возможно ли создать программу, использующую адаптивную интеграцию Монте-Карло? Будет ли она эффективной?
3. Напишите высокоуровневый алгоритм, который ищет объем трехмерной фигуры с помощью интеграции Монте-Карло.
4. Используйте метод Ньютона для нахождения точек пересечения двух функций.

# Глава 3 СВЯЗНЫЕ СПИСКИ

Связные списки — возможно, самые простые структуры данных, которые вам придется создавать. Тем не менее некоторые методы их построения применимы для формирования более сложных структур, описанных в книге. Для использова- ния связных списков вам необходимо иметь представление о ссылках и ячейках, включая способы их нахождения, вставки и удаления. Эти же понятия фигури- руют при построении сложных сетей и деревьев, в том числе сбалансированных.

В данной главе вы познакомитесь с основами, необходимыми для работы со связными списками. В последующих главах (4, 5, 8, а также 10–14) мы будем воз- вращаться к рассмотренному здесь материалу.

## Основные положения

*Связный список* построен из объектов, обычно называемых *ячейками*. Этот класс содержит все данные, которые должны храниться в списке, и ссылку на другую ячейку. *Ссылка* представляет собой справку или указатель на объект такого же класса. Поле типа «указатель» в ячейке часто называется Next.

Например, приведенный ниже код представляет определение класса IntegerCell в C#. В ячейке содержатся целое число и указатель на следующий объект IntegerCell в связном списке.

class IntegerCell

{

public int Value;

public IntegerCell Next;

}

Графически связные списки представляют, как правило, в виде квадратиков (ячеек) и стрелок (ссылок). Ссылку, которая ни на что не указывает, обозначают квадратиком поменьше с косым перекрестием внутри (на языке программирова- ния значением указателя, соответствующего такой ссылке, будет ничто, нуль либо другое специальное значение, которое имеет аналогичный смысл).

Чтобы код программы сумел найти список, требуется определенная переменная, которая будет на него указывать. Нередко она называется top (поскольку пред- ставляет вершину списка) и может быть переменной класса ячейки или указателем на первую ячейку в списке.

На рисунке 3.1 показаны два связных списка, в которых содержатся цифры 31, 72, 47 и 9. Верхняя переменная top — это указатель на первую ячейку списка,

а нижняя — первая ячейка в списке. Оба списка заканчиваются квадратиком с косым перекрестием — так обозначается нулевой указатель.

**top**

31

72



9

47

top

31

72

**Рис. 3.1.** Примеры связных списков

Связные списки — хороший способ хранить элементы, количество которых может со временем увеличиваться или уменьшаться. Чтобы присовокупить новую ячейку, нужно всего лишь добавить ее в начало или в конец связного списка. Мас- сив располагает фиксированным размером, поэтому его сложно увеличить, если требуется включить новые элементы.



9

47

В следующем разделе рассматриваются некоторые алгоритмы, которые мож- но использовать для управления связными списками. Многие из них легче всего описывать с помощью цифр, показывающих список до и после выполнения операции.

## Однонаправленные связные списки

В *однонаправленном связном списке* (см. рис. 3.1) каждая ячейка связана со следующей с помощью одинарной ссылки. Чтобы использовать такой список, вам понадобятся алгоритмы для передвижения по списку, поиска, добавления и уда- ления элементов. Рассмотрим некоторые из них.

#### Передвижение по спискам

Если программа содержит встроенные или связные списки, переход по ее ячей- кам осуществляется относительно просто. Следующий алгоритм показывает, как это сделать и как работать со значениями в ячейках. В приведенном ниже примере используется метод Print (он выводит значения ячеек), однако вы можете заме- нить его любым другим.

Iterate(Cell: top) While (top != null)

Print top.Value top = top.Next

End While End Iterate

**ЗАМЕЧАНИЕ**

 В подобных алгоритмах предполагается, что значение проходит параметр top. Таким образом, код может изменять его, сохраняя значение top в коде вызова.

Данный алгоритм начинается с цикла While, который работает до тех пор, пока верхний указатель ячейки не станет null. Внутри цикла алгоритм сперва вызывает метод Print, чтобы показать значение ячейки top, а затем с ее помощью указыва- ет на следующую ячейку связного списка. Процесс продолжается до тех пор, пока top не станет указывать на null в конце списка и цикл While не остановится.

Приведенный алгоритм изучает каждую ячейку в связном списке, поэтому если их количество равно *N*, то время работы составит *O*(*N*).

#### Нахождение ячеек

Поиск ячейки выглядит так: алгоритм передвигается по связному списку и оста- навливается, найдя нужную.

Cell: FindCell(Cell: top, Value: target) While (top != null)

If (top.Value == target) Then Return top top = top.Next

End While

// Если мы дошли до этой строки, искомого значения нет в списке. Return null

End FindCell

Алгоритм вводит цикл While, который работает, пока top не станет равным null. Внутри этого цикла сравнивается значение ячейки top с искомым значе- нием. Если они совпадают, алгоритм возвращает top; если нет, top превращается в указатель на следующую ячейку в списке.

Если top становится равным null после того, как будет пересмотрен весь спи- сок, — искомого значения нет. Возможно также, что в алгоритме вызвано исключение или возникла какая-то ошибка (зависит от используемого языка программирования).

В следующих разделах вы увидите, что легче всего работать с ячейкой в связном списке, если есть указатель на ту, что стоит перед ней. Приведенный ниже алгоритм находит ячейку, предшествующую целевой.

Cell: FindCellBefore(Cell: top, Value: target)

// Если список пуст, искомого значения нет. If (top == null) Return null

// Поиск искомого значения. While (top.Next != null)

If (top.Next.Value == target) Then Return top top = top.Next

End While

// Если мы дошли до этой строки, искомого значения нет в списке. Return null

End FindCellBefore

Код подобен предыдущему, но в нем есть два отличия. Первое заключается в том, что перед началом работы проверяется, принимает ли top значение null. Это помогает понять, можно ли рассматривать top.Next без риска. Ведь если top.Next окажется неопределенным, программа, использующая такой алгоритм, зависнет. В случае когда top не является null, алгоритм вводит цикл While, од- нако на этот раз вместо top.Value рассматривается top.Next.Value. Если уда- ется отыскать искомое значение, top указывает на ячейку перед той, которая его содержит, и алгоритм возвращает top.

#### Использование ограничителей

Если внимательно изучить приведенный выше алгоритм, легко представить случай, при котором он даст сбой. Например, когда искомое значение содержится в первой ячейке связного списка, то перед ней не окажется ячейки и алгоритм не сможет ничего вернуть. Первое значение, которое он станет рассматривать, будет находиться во второй ячейке списка, а алгоритм никогда не возвращается назад.

Один из способов справиться с описанной проблемой — добавить специальный код, который отыщет искомое значение непосредственно в первой ячейке. Но если программе придется рассматривать эту ситуацию в качестве особого случая, она может запутаться.

Другой подход — создать *ограничитель* в начале списка. Он представляет собой ячейку, которая является частью связного списка, но не содержит какие-либо зна- чимые данные. Ограничитель используется только в качестве метки-заполнителя, поэтому алгоритмы могут свободно обращаться к нему.

Следующий псевдокод представляет приведенный выше алгоритм

FindCellBefore, но уже с использованием ограничителя.

Cell: FindCellBefore(Cell: top, Value: target)

// Поиск искомого значения. While (top.Next != null)

If (top.Next.Value == target) Then Return top top = top.Next

End While

// Если мы дошли до этой строки, искомого значения нет в списке. Return null

End FindCellBefore

Эта версия не нуждается в проверке того, принимает ли top значение null, по- скольку в связном списке всегда есть хотя бы один ограничитель. В силу указанной причины цикл While может начаться сразу. Проверка значения в первой ячейке списка также осуществляется, поэтому алгоритм будет работать и в том случае, когда искомое значение находится в ней.

Данная версия кода может возвращать ячейку с ограничителем перед первой реальной (верхней) ячейкой, если это целесообразно. Таким образом, программе не требуется специальный код для работы с особым случаем, когда искомое значение размещено в начале списка.

При поиске нужного значения вам может повезти — оно отыщется сразу. В худшем случае алгоритму придется пройтись по большей части связного списка.

Если искомого значения не окажется, в поле зрения попадет каждая ячейка. То есть если список содержит *N* ячеек, время работы алгоритма будет определяться как *O*(*N*).

На первый взгляд, может показаться, что ограничитель только занимает место. На самом деле он отменяет необходимость использовать специальный код и упро- щает алгоритм, а также делает его более изящным.

В следующих разделах предполагается, что связные списки имеют ограничители и верхний указатель.

#### Добавление ячеек в начало списка

Связные списки используются для создания такой структуры данных, которая может хранить элементы подобно массиву, но способна расширяться, если потре- буется больше места. Чтобы добавить ячейку к такому списку, достаточно вставить ее в начало, сразу после ограничителя.

AddAtBeginning(Cell: top, Cell: new\_cell) new\_cell.Next = top.Next

top.Next = new\_cell End AddAtBeginning

Алгоритм устанавливает новый указатель ячейки Next таким образом, чтобы он указывал вначале на ту ячейку, которая идет первой после ограничителя в списке, а затем на новую. Получается, что новая ячейка размещается после ограничителя и становится первой в связном списке.

На рисунке 3.2 представлен связный список до и после добавления новых ячеек в начало списка (закрашенный квадрат — ограничитель).

Новая ячейка

Ограничитель



Новая ячейка

Ограничитель



**Рис. 3.2.** Добавление элемента в начало связного списка

Указанный алгоритм состоит всего из двух шагов, поэтому время его работы равно *O*(1), независимо от количества ячеек в списке.

#### Добавление ячеек в конец списка

Вставить ячейку в конец списка немного сложнее, чем в начало, поскольку алгоритм сначала должен пройти по списку, чтобы добраться до последней ячейки.

AddAtEnd(Cell: top, Cell: new\_cell)

// Находим последнюю ячейку. While (top.Next != null)

top = top.Next End While

// Добавляем новую ячейку в конец. top.Next = new\_cell

new\_cell.Next = null End AddAtEnd

Код проходит по связному списку вплоть до последней ячейки и делает так, чтобы ее ссылка указывала сначала на новую ячейку, а та в свою очередь на null. Алгоритм выглядел бы более беспорядочным, если бы в списке отсутствовал огра- ничитель. Вы можете использовать специальный код в том случае, если список пуст, а top указывает на null.

На рисунке 3.3 этот процесс представлен графически.

Новая ячейка

Ограничитель



Новая ячейка

Ограничитель

**Рис. 3.3.** Добавление элемента в конец связного списка

Алгоритм должен пройти через весь список, поэтому, если в списке содержится

*N* ячеек, время его работы составит *O*(*N*).

#### Вставка ячеек

В предыдущих разделах мы рассмотрели, как добавить ячейки в начало или конец связного списка. А что если их требуется вставить в середину? Допустим, у вас есть переменная after\_me, указывающая на ячейку, после которой нужно добавить элемент.

InsertCell(Cell: after\_me, Cell: new\_cell) new\_cell.Next = after\_me.Next after\_me.Next = new\_cell

End InsertCell

Согласно алгоритму, ссылка новой ячейки указывает на ячейку, следующую за

after\_me, а ссылка after\_me — на новую ячейку (рис. 3.4).

after\_me

Новая ячейка

Ограничитель



after\_me



Новая ячейка

Ограничитель

**Рис. 3.4.** Вставка новой ячейки после отмеченной ячейки

Поскольку алгоритм проходит всего два шага, он работает в течение *O*(1) вре- мени, хотя может понадобиться и *O(N)* времени, чтобы найти ячейку after\_me. Например, если вы хотите вставить ячейку после ячейки, содержащей искомое значение, для начала придется ее отыскать.

#### Удаление ячеек

Чтобы удалить целевую ячейку, нужно установить предыдущую ссылку на ячейку, которая идет за целевой. Следующий псевдокод представляет алгоритм, определяющий ячейку after\_me.

DeleteAfter(Cell: after\_me) after\_me.Next = after\_me.Next.Next

End DeleteAfter

На рисунке 3.5 алгоритм показан графически.

after\_me

Ограничитель



after\_me

Ограничитель



**Рис. 3.5.** Удаление ячейки из связного списка

В C# и Visual Basic для управления памятью используется метод «сборки мусора», который автоматически утилизирует удаленные ячейки, если программе требуется больше памяти. Однако в некоторых языках программирования могут понадобиться дополнительные действия. Следующая версия алгоритма демонстрирует освобож- дение целевой ячейки в C++.

DeleteAfter(Cell: after\_me)

Cell: target\_cell = after\_me.Next after\_me.Next = after\_me.Next.Next free(target\_cell)

End DeleteAfter

От языка программирования зависит и то, как вы будете удалять связный спи- сок. В C# и Visual Basic достаточно приравнять все ссылки на список к значению null, и «сборка мусора» его утилизирует. А в C++, например, нужно пройти по списку и освободить каждую ячейку непосредственно, как показано в следующем псевдокоде.

DestroyList(Cell: top) While (top != null)

// Сохраняем указатель на следующую ячейку. Cell: next\_cell = top.Next

// Освобождаем ячейку top. free(top)

// Переходим к следующей ячейке. top = next\_cell

End While End DestroyList

Итак, запомните, что освобождение ресурсов зависит от языка программиро- вания и при удалении ячейки или другого объекта из структуры данных вам могут понадобиться дополнительные действия.

## Двунаправленные связные списки

В двунаправленных списках ссылки ячеек указывают на следующие и предыду- щие ячейки. Вторые часто называют Prev, или Previous.

В подобных структурах данных удобно иметь верхний и нижний ограничители (рис. 3.6), чтобы программа легко управляла списком с обоих концов, например добав- ляла и удаляла элементы. Такой подход позволяет это сделать за время, равное *O*(1).

Верхний ограничитель



Нижний ограничитель

**Рис. 3.6.** Пример двунаправленного списка

Алгоритмы для работы с двунаправленными и однонаправленными списками очень похожи, за исключением того, что первые должны выполнять дополнительные действия для управления еще одним набором ссылок. Вот как выглядит псевдокод, добавляющий ячейку после выбранной.

InsertCell(Cell: after\_me, Cell: new\_cell)

// Обновляем ссылки на следующую ячейку. new\_cell.Next = after\_me.Next after\_me.Next = new\_cell

// Обновляем ссылки на предыдущую ячейку. new\_cell.Next.Prev = new\_cell new\_cell.Prev = after\_me

End InsertCell

Основная трудность подобных алгоритмов заключается в отслеживании ссылок, способных обновляться в любой момент времени. В приведенном выше алгоритме третий оператор с конца устанавливает ссылку Prev, которая должна указывать на новую ячейку.

after\_me.Next.Prev = new\_cell

Однако когда оператор запущен, after\_me.Next уже обновлен и указывает на новую ячейку, поэтому вместо него следует использовать new\_cell.Next.

На рисунке 3.7 алгоритм представлен графически.



Новая ячейка

Верхний ограничитель

after\_me

Нижний ограничитель

Верхний ограничитель

after\_me

**Рис. 3.7.** Обновление двунаправленного списка



Новая ячейка

Нижний ограничитель

## Сортированные списки

Иногда элементы связного списка удобно хранить в отсортированном виде. В этом случае чтобы добавить элемент, нужно найти позицию, к которой он при-

надлежит, и обновить соответствующие ссылки. Рассмотрим, как выполняется по- добное действие для сортированного однонаправленного списка.

// Вставляем новую ячейку в сортированный однонаправленный список.

InsertCell(Cell: top, Cell: new\_cell)

// Находим ячейку перед той, в которую будем вставлять новый элемент. While (top.Next != null) And (top.Next.Value < new\_cell.Value)

top = top.Next End While

// Вставляем новую ячейку после ячейки top. new\_cell.Next = top.Next

top.Next = new\_cell End InsertCell

В худшем случае алгоритму понадобится пересмотреть весь список, чтобы найти верное местонахождение нового элемента. Таким образом, если ячеек *N*, время работы составит *O*(*N*), и, хотя теоретически его улучшить невозможно, на практике алгоритм получится ускорить, добавив нижний ограничитель. Когда зна- чение Value нижнего ограничителя будет превышать любое аналогичное значение из ячеек, тест top.Next != null можно удалить. Такое допустимо, поскольку код все равно отыщет местоположение для новой ячейки, даже если оно будет на- ходиться прямо перед нижним ограничителем.

Предположим, в ячейках содержатся имена, использующие символы ASCII. Вы можете установить Value нижнего ограничителя на ~, поскольку этот символ следует после любого действительного имени. Если речь идет о целых числах, то в качестве Value нижнего ограничителя лучше использовать наибольшее целое число (для большинства 32-битных систем это 2 147 483 647).

Ниже представлена исправленная версия псевдокода. Предполагается, что в списке находится нижний ограничитель со значением большим, чем любое другое значение в ячейках.

// Вставляем новую ячейку в сортированный однонаправленный список. InsertCell(Cell: top, Cell: new\_cell)

// Находим ячейку перед той, в которую будем вставлять новый элемент. While (top.Next.Value < new\_cell.Value)

top = top.Next End While

// Вставляем новую ячейку после ячейки top. new\_cell.Next = top.Next

top.Next = new\_cell End InsertCell

## Алгоритмы для работы со связными списками

До сих пор в этой главе рассматривались алгоритмы, с помощью которых стро- ятся связные списки и происходит управление ими (добавляются элементы в на- чало, конец и середину, а также находятся и удаляются). Далее будут описываться немного другие коды.

#### Копирование

С помощью алгоритмов списки можно перегруппировывать. Этот и следующие разделы будут посвящены сортировке элементов. Если вы хотите оставить началь- ный список в целостности и сохранности, то предварительно сделайте его копию.

Приведенный ниже псевдокод демонстрирует, как можно скопировать одно- направленный список.

// Копируем список.

Cell: CopyList(Cell: old\_sentinel)

// Создаем ограничитель нового списка. Cell: new\_sentinel = new Cell()

// Отслеживаем последний добавленный элемент. Cell: last\_added = new\_sentinel

// Пропускаем ограничитель.

Cell: old\_cell = old\_sentinel.Next

// Копируем элементы. While (old\_cell != null)

// Создаем новый элемент. last\_added.Next = New Cell

// Переходим к новому элементу. last\_added = last\_added.Next

// Устанавливаем значение нового элемента. last\_added.Value = old\_cell.Value

// Готовимся копировать следующую ячейку. old\_cell = old\_cell.Next

End While

// Заканчиваем ячейкой null. last\_added.Next = null

// Возвращаем ограничитель нового списка. Return new\_sentinel

}

Алгоритм выглядит довольно просто, но стоит упомянуть об одной его особен- ности — переменной last\_added. Она нужна для того, чтобы следить за ячейкой, которую недавно добавили к копии списка. Чтобы скопировать новый элемент, алгоритм приравнивает его к last\_added.Next и переводит в конец списка. За- тем обновляется переменная last\_added (теперь она будет указывать на новый элемент), и в нее копируется оригинальное значение ячейки.

Таким образом, список увеличивается снизу, а не сверху. Это похоже на добав- ление элементов в конец списка из упражнения 1 в конце текущей главы.

#### Сортировка вставкой

Алгоритмам сортировки посвящена глава 6, но о двух из них — сортировке вставкой и методом выбора — мы поговорим уже сейчас.

В основе первого упомянутого алгоритма лежит следующая идея: из списка ввода выбирается элемент и вставляется в соответствующую позицию в отсорти- рованном списке вывода, который вначале пуст. Приведенный ниже псевдокод представляет реализацию этого процесса на примере однонаправленного списка с верхним ограничителем.

// Используем функцию Insertionsort для сортировки списка. Cell: Insertionsort(Cell: input)

// Устанавливаем ограничитель для списка вывода. Cell sentinel = new Cell()

sentinel.Next = null

// Пропускаем ограничитель списка ввода. input = input.Next

// Повторяем до тех пор, пока не вcтавим все элементы в новый список. While (input != null)

// Берем следующую ячейку для добавления в список. Cell: next\_cell = input

// Заменяем input на input.Next для следующего прохождения цикла. input = input.Next

// Смотрим, куда добавить следующий элемент в список вывода. Cell: after\_me = sentinel

While (after\_me.Next != null) And

(after\_me.Next.Value < next\_cell.Value) after\_me = after\_me.Next

End While

// Вставляем элемент в список вывода. next\_cell.Next = after\_me.Next after\_me.Next = next\_cell

End While

// Возвращаем отсортированный список. return sentinel

End Insertionsort

В самом начале алгоритм выстраивает пустой список, который будет содержать отсортированный вывод. Затем в цикле пересматривается неотсортированный список вводных ячеек и для каждой из них определяется новое место. Код можно упростить, применив алгоритм InsertCell из подраздела «Вставка ячеек» раз- дела «Однонаправленные связные списки» в текущей главе.

Если элементы в списке ввода сортируются в порядке возрастания, в алгоритме каждый из них вставляется в начало нового списка за несколько шагов. При усло- вии что в списке *N* ячеек, для добавления всех элементов потребуется около *O*(*N*) шагов — это наилучший вариант работы.

Если элементы сортируются в порядке убывания, то они добавляются в конец выводного списка. Каждый раз, чтобы переместиться туда, потребуется выполнить еще один дополнительный шаг. Таким образом, для вставки всех элементов пона- добится 1 + 2 + 3 + ... + *N* = *N*  (*N* – 1)/2 = *O*(*N*2) шагов.

При случайной сортировке элементов одни из них могут добавляться быстро, другие отнимут больше времени. В результате работа алгоритма будет определяться как *O*(*N*2), хотя на практике она не превысит время худшего случая.

Многие алгоритмы сортировки занимают *O*(*N* log *N*) времени, потому произ- водительность алгоритма *O*(*N*2) является относительно низкой. Это делает алго- ритм неэффективным для больших списков, но подходящим для работы с малыми и связными списками.

#### Сортировка методом выбора

В данном случае алгоритм ищет по списку ввода наибольший элемент, а затем добавляет его в начало увеличивающегося списка вывода. Следующий псевдокод представляет алгоритм сортировки для однонаправленного списка, содержащего целые числа.

// Используем функцию Selectionsort для сортировки списка. Cell: Selectionsort(Cell: input)

// Устанавливаем ограничитель для списка вывода.

Cell: sentinel = new Cell sentinel.Next = null

// Повторяем до тех пор, пока список ввода не будет пуст. While (input.Next != null)

// Находим наибольший элемент в списке ввода.

// Ячейка after\_me предшествует ячейке с наибольшим элементом. Cell: best\_after\_me = input

Integer: best\_value = best\_after\_me.Next.Value

// Начинаем искать следующий элемент. Cell: after\_me = input.Next

While (after\_me.Next != null)

If (after\_me.Next.Value > best\_value) Then best\_after\_me = after\_me best\_value = after\_me.Next.Value

End If

after\_me = after\_me.Next End While

// Удаляем лучшую ячейку из списка ввода. Cell: best\_cell = best\_after\_me.Next best\_after\_me.Next = best\_cell.Next

// Добавляем лучшую ячейку в начало списка вывода. best\_cell.Next = sentinel.Next

sentinel.Next = best\_cell End While

// Возвращаем отсортированный список. Return sentinel

End Selectionsort

Этот алгоритм можно несколько упростить, если выделить код, который нахо- дит наибольшую ячейку в списке ввода, а затем поместить его в другой алгоритм и запускать уже из него.

Если в списке ввода содержится *K* элементов, то для поиска наибольшего из них понадобится пройти *K* шагов. По мере работы алгоритма список будет уменьшаться. Таким образом, если изначально в нем было *N* элементов, общее количество шагов составит *N* + (*N* – 1) + (*N* – 2) + ... + 2 + 1 = *N*  (*N* – 1)/2 = *O*(*N*2). Это совпадает со временем работы алгоритма сортировки.

## Многопотоковые связные списки

В однонаправленном связном списке каждая ячейка содержит ссылку на сле- дующую ячейку, в двунаправленном — на следующую и предыдущую. Это необхо- димо для того, чтобы обеспечить два способа перемещения — прямой и обратный. Однако к ячейкам списка можно добавлять и другие ссылки, чтобы предусмотреть иные способы перемещения.

Предположим, вы построили класс Planet для хранения информации о плане- тах Солнечной системы и назначили ему поле с именем NextDistance, в котором находятся сведения о расстояние до Солнца. С учетом этого список будет органи- зован в следующем порядке: Меркурий, Венера, Земля, Марс, Юпитер, Сатурн, Уран, Нептун (и Плутон, если захотите включить и его). Аналогичным образом можно добавить и другие поля, чтобы отсортировать планеты по их массе, диаметру и т. д. Каждое такое прохождение через ячейки, определенные набором ссылок, называется *потоком*.

С одним потоком работать легко, если представить его как простой связный список, но визуализировать все потоки одновременно довольно сложно. На ри- сунке 3.8 показан связный список планет с тремя потоками. Ссылки, обозначен- ные тонкой линией, указывают на сортировку по удаленности от Солнца, пунктир- ной — по массе, а толстой — по диаметру.



Меркурий

Земля

Марс

Нептун

Уран

Сатурн

Юпитер

Венера

Ограни- читель

**Рис. 3.8.** Визуализация многопотокового связного списка

**ЗАМЕЧАНИЕ**

Потоки могут иметь и другие структуры данных, например дерево. В этом слу- чае оно позволяет программе посетить все его вершины в нетипичном порядке.

## Связные списки с циклами

*Кольцевой связный список* — это связный список, в котором последняя ссылка указывает на первый элемент списка (рис. 3.9). Он может быть полезен в том слу- чае, когда в рамках цикла нужно бесконечно проходить через последовательность элементов. Так, операционная система способна повторять цикл процессов, чтобы запустить каждый из них. Если начался новый процесс, он может быть добавлен в любое место списка, например за ограничителем, что позволит ему сразу запу- ститься.

Ограничитель



**Рис. 3.9.** Пример кольцевого связного списка

Еще одним примером служит игра, которая бесконечно циклически проходит по списку объектов, позволяя каждому из них передвигаться по экрану. И снова новые объекты можно добавлять в список куда угодно.

На рисунке 3.10 показан кольцевой связный список, в который включены не все ячейки. В этом случае возникают два интересных вопроса. Во-первых, как определить, содержит ли связный список такой цикл? Во-вторых, если связный список содержит подобный цикл, как узнать, где он начинается и как его прервать? Последний вопрос, по сути, связан с определением конца списка.

Ограничитель



*A*

*B*

*C*

*D*

*I E*

*H*

*F*

*G*

**Рис. 3.10.** Кольцевой связный список и ячейки, не включенные в цикл

Взглянув еще раз на рисунок 3.10, вы можете определить, что конец списка — это ячейка *I*, поскольку ее посещают последней перед тем, как начнется очередной по- втор действий. В следующих разделах описаны некоторые из наиболее интересных алгоритмов, помогающие ответить на поставленные вопросы.

#### Маркировка ячеек

Возможно, самый легкий способ определить, имеет ли связный список цикл, — пройти через его ячейки и пометить каждую. Если очередная рассматриваемая ячейка окажется помеченной, значит, в списке есть цикл и он начинается в этом месте. Приведенный ниже алгоритм представляет следующий псевдокод.

// Возвращаем true, если в списке есть цикл.

// Если в списке есть цикл, разрываем его. Boolean: Has\_loopMarking(Cell: sentinel)

// Предполагаем, что цикла нет. Boolean: has\_loop = false

// Проходим по всему списку. Cell: cell = sentinel

While (cell.Next != null)

// Смотрим, были ли мы в следующей ячейке. If (cell.Next.Visited)

// Это начало цикла.

// Разрываем цикл. cell.Next = null has\_loop = true

<Разрываем цикл While.>

End If

// Переходим к следующей ячейке. cell = cell.Next

// Помечаем ячейку как пройденную. cell.Visited = true

End While

// Еще раз проходим по списку, чтобы снять флажки. cell = sentinel

While (cell.Next != null) cell.Visited = false cell = cell.Next

End While

// Возвращаем результат. Return has\_loop

End Has\_loopMarking

Подобный принцип работы реализован в программе BreakLoopMarking, кото- рую вы можете загрузить на сайте с материалами книги. Алгоритм проходит через цикл дважды: вначале, чтобы присвоить флажку Visited в каждой ячейке значение true, а затем, чтобы поменять его на false. Если в списке *N* ячеек, то алгоритм совершает 2*N* шагов и работает в течение времени *O*(*N*).

Кроме того, алгоритм требует, чтобы для каждой ячейки было добавлено поле Visited, поэтому ему понадобится *O*(*N*) времени. Поскольку список уже зани- мает *O*(*N*) места для хранения ячеек и их ссылок, это не должно стать проблемой. Однако следует помнить, что алгоритм все же требователен к памяти.

**ЗАМЕЧАНИЕ**

 Маркировку ячеек удобно использовать и при работе с другими структурами данных, в частности с сетями. Этот метод реализован в некоторых алгоритмах,

описанных в главах 13 и 14.

Нередко в подобных задачах есть дополнительное условие, запрещающее до- бавлять поле Visited. Этому ограничению соответствуют рассмотренные ниже алгоритмы.

#### Использование хеш-таблиц

Более подробно хеш-таблицы описываются в главе 8. Все, что вам необходимо знать о них сейчас, — такие таблицы помогают очень быстро сохранять и находить элементы, а также выявлять их наличие в принципе.

Представленный ниже алгоритм перемещается по списку и добавляет каждую ячейку в хеш-таблицу, предварительно проверяя, нет ли ее уже там. Если он до- ходит до ячейки, которая присутствует в хеш-таблице, значит, с нее в списке на- чинается цикл.

// Возвращаем true, если в списке есть цикл.

// Если в списке есть цикл, разрываем его. Boolean: HasLoopHashTable(Cell: sentinel)

// Создаем хеш-таблицу. Hashtable: visited

// Проходим по всему списку. Cell: cell = sentinel

While (cell.Next != null)

// Смотрим, были ли мы в следующей ячейке.

If (visited.Contains(cell.Next))

// Это начало цикла.

// Разрываем цикл и возвращаем true. cell.Next = null

Return true

End If

// Добавляем ячейку в хеш-таблицу. visited.Add(cell)

// Переходим к следующей ячейке. cell = cell.Next

End While

// Если мы дошли до этой строки, цикла в списке нет. Return false

End HasLoopHashTable

Работу приведенного кода демонстрирует программа BreakLoopHashtable, кото- рую вы можете загрузить на сайте с материалами книги. Алгоритм проходит через ячейки списка один раз, поэтому если в списке содержится *N* ячеек, то выполняется *N* шагов, а время работы составляет *O*(*N*).

Для этого алгоритма также нужна хеш-таблица. Лучше всего, если ее размер бу- дет превышать количество действительных значений. В таком случае хеш-таблица должна располагать пространством для более чем *N* записей, например 1,5*N*. Это повысит производительность и займет *O*(*N*) места.

Рассмотренный код придерживается ограничения (нельзя изменять класс ячейки), но использует дополнительную память. В следующих разделах описаны алгоритмы, которые определяют циклы, обходясь без нее.

#### Повторная трассировка списка

Представленный ниже алгоритм проводит по списку один объект, а затем в его поисках через каждую посещенную ячейку проходит второй объект. Это слегка запутанное объяснение легче понять, если взглянуть на следующий псевдокод.

// Возвращаем true, если в списке есть цикл.

// Если в списке есть цикл, разрываем его. Boolean: HasLoopRetracing(Cell: sentinel)

// Проходим по всему списку. Cell: cell = sentinel

While (cell.Next != null)

// Смотрим, были ли мы в следующей ячейке. Cell: tracer = sentinel

While (tracer != cell)

If (tracer.Next == cell.Next)

// Это начало цикла.

// Разрываем цикл и возвращаем true. cell.Next = null

Return true

End If

tracer = tracer.Next End While

// Переходим к следующей ячейке. cell = cell.Next

End While

// Если мы дошли до этой строки, цикла в списке нет. Return false

End HasLoopRetracing

Работу этого алгоритма демонстрирует программа BreakLoopHashtable, кото- рую можно скачать на сайте с материалами книги.

Предположим, что в списке *N* ячеек. Когда объект алгоритма cell проверяет *K*-ю ячейку, объект tracer должен пройти вверх по списку до этой точки. Таким образом, он выполнит *K* шагов. Это значит, что общее время работы алгоритма со- ставит 1 + 2 + 3 + ... + *N* = *N*  (*N* – 1)/2 = *O*(*N*2). Оно меньше, чем у предыдущих алгоритмов, но зато не требуется дополнительной памяти.

Следующий алгоритм также не нуждается в дополнительных ресурсах, но ра- ботает в течение времени *O*(*N*).

#### Реверсирование списка

Алгоритм проходит по ячейкам списка, меняя каждую ссылку таким образом, чтобы она указывала на предыдущую ячейку, а не на последующую. Если алгоритм обнаруживает значение null до ограничителя, цикла в списке нет. Конечно, опи- санное действие преобразует ссылки, поэтому алгоритм опять проходит по списку и восстанавливает их, чтобы они снова указывали на изначальные ячейки.

Прояснит подобный принцип рисунок 3.11. Верхний фрагмент — это ориги- нальный список, а закрашенная ячейка — та, через которую проходит алгоритм в настоящее время, реверсируя ссылки. На среднем фрагменте видно, что алгоритм достиг ячейки *I* (измененные ссылки показаны толстыми стрелками) и должен проследовать в ячейку *D*, а из нее по реверсированным ссылкам в ячейки *C*, *B* и *A*. В результате совершенных действий ссылки изменятся снова, что отражено на нижнем фрагменте пунктиром. В этом месте алгоритм возвращается к пер- вой ячейке и определяет, что список содержит цикл. Обратите внимание: новый список выглядит так же, как и старый, только ссылки в цикле имеют обратное направление.

Ограничитель



*B*

*C*

*D*

*I E*

Ограничитель

*H*

*F*

*A*

*B*

*C*

*G D*

*I E*

Ограничитель

*H*

*F*

*B*

*C*

*G*

*D*

*I E*

*H*

*F*

*G*

*A*

*A*

**Рис. 3.11.** Обнаружение цикла в связном списке с помощью реверсирования ссылок

Поскольку такой алгоритм должен реверсировать список дважды, он начинается со следующего метода.

// Реверсируем цикл и возвращаем новое начало списка. Cell: ReverseList(Cell: sentinel)

Cell: prev\_cell = null Cell: curr\_cell = sentinel While (curr\_cell != null)

// Реверсируем ссылку данной ячейки. Cell: next\_cell = curr\_cell.Next curr\_cell.Next = prev\_cell

// Переходим к следующей ячейке. prev\_cell = curr\_cell

curr\_cell = next\_cell End While

// Возвращаем последнюю пройденную ячейку. Return prev\_cell

End ReverseList

Приведенный псевдокод перемещается по списку, меняет ссылки и возвращает последний посещенный им элемент, который теперь становится первым.

Следующий алгоритм использует предыдущий псевдокод, чтобы определить наличие цикла.

// Возвращаем true, если в списке есть цикл. Boolean: HasLoopReversing(Cell: sentinel)

{

// Если список пуст, цикла в нем нет.

If (sentinel.Next == null) Then Return false

// Проходим по всему списку, реверсируя ссылки. Cell: new\_sentinel = ReverseList(sentinel)

// Еще раз проходим по всему списку, чтобы восстановить ссылки. ReverseList(new\_sentinel)

// Если реверсированный список начинается с той же ячейки,

// что и оригинальный список, значит, в нем есть цикл.

// Возвращаем результат.

If (new\_sentinel == sentinel) Then Return true Return false

End HasLoopReversing

Указанный алгоритм вызывает реверсирующий метод ReverseList дважды: сперва, чтобы получить первую ячейку измененного списка, а затем, чтобы восста- новить начальные значения ссылок. Если при этом ограничитель окажется таким же, как и в первой ячейке нового списка, алгоритм вернет значение true; если нет — false. В целом же выполняется 2*N* = *O*(*N*) шагов.

Алгоритм работает за время *O*(*N*) и не требует дополнительных ресурсов. Но у него есть и недостаток, который заключается в том, что он способен лишь обнаружить цикл, но не остановить его. Следующий алгоритм решает и эту задачу, зато является, возможно, наиболее сложным из всех описанных в текущем разделе.

#### Черепаха и кролик

*Алгоритм кролика и черепахи*, который также называется *алгоритмом на- хождения циклов Флойда*, придуман Робертом Флойдом в 1960-х гг. Лежащий в его основе принцип не слишком сложный, но объяснить его непросто, поэто- му если вы не любите математические формулы, то нижеприведенное описа- ние можно пропустить.

Алгоритм запускает два объекта: «черепаху» и «кролика». Они двигаются из на- чала списка, но с разными скоростями: «черепаха» преодолевает за один шаг одну ячейку, а «кролик» — две. Когда последний достигает ссылки со значением null — список окончен и цикла в нем нет. Если же цикл есть, «кролик» входит в него и на- чинает бегать по кругу. Тем временем «черепаха» ползет, пока не дойдет до цикла и в какой-то момент не окажется в нем вместе с «кроликом».

Пусть *L* — количество ячеек внутри цикла, *T* — количество шагов, пройденных

«черепахой», чтобы попасть в цикл, а *H* — расстояние от начала цикла до того по- ложения, в котором за это же количество шагов оказался «кролик». Тогда, согласно рисунку 3.12, *L* = 5, *T* = 4, *H* = 4.

*T*



*A*

*H*



*E*

*H*

*F*

*C*

*B*

*G*

*D*

**Рис. 3.12.** Движение «кролика» и «черепахи»:

*T* — расстояние, которое проходит «черепаха», чтобы дойти до цикла,

*H* — расстояние от начала цикла до «кролика» за то же время

Поскольку «кролик» перемещается в два раза быстрее «черепахи», он доходит до цикла через *T* ячеек и преодолевает еще *T* ячеек внутри цикла, чтобы оказаться в позиции, как на рисунке 3.12. Это приводит к факту № 1.

**ФАКТ № 1**

Если вы перемещаетесь через *T* ячеек внутри цикла, то заканчиваете свой путь за *H* ячеек от того места, где его начали.

Обратите внимание, что «кролик» мог пробежать несколько кругов по циклу, если *L* намного меньше *T*. Например, если *L* = 5, а *T* = 102, черепаха дойдет до цик- ла за 102 шага. «Кролик» же добирается до цикла за 51 шаг, совершает 50 шагов (преодолевает 100 ячеек), пробегая 20 кругов по циклу, а затем перемещается еще на один шаг (2 ячейки) внутри цикла. В таком случае *H* = 2.

Возникает вопрос: когда же «кролик» догонит «черепаху»? Как только «чере- паха» попадет в цикл, «кролик» окажется на *H* шагов впереди нее (см. рис. 3.12). Однако поскольку «черепаха» и «кролик» пребывают в цикле, вы можете рассма- тривать последнего как отстающего на *L* – *H* ячеек. Из-за того, что «кролик» пере- мещается на две ячейки, а «черепаха» на одну, он нагоняет по одной ячейке за шаг. Это означает, что «кролик» доберется до «черепахи» за *L* – *H* шагов.

На рисунке 3.12, где *H* = 4, а *L* = 5, «кролик» догонит «черепаху» за 5 – 4 =

= 1 шаг. Они встретятся в ячейке *E*. В этот момент «черепаха» передвинется в ци- кле на *L* – *H* ячеек, и оба объекта будут на *L* – (*L* – *H*) = *H* ячеек ближе к началу цикла. Отсюда факт № 2.

**ФАКТ № 2**

Когда «кролик» догонит «черепаху», они будут находиться за *H* ячеек от начала цикла.

Если бы вы могли переместить объект «черепаха» на *H* ячеек от точки встречи, он оказался бы как раз в начале цикла. К сожалению, это значение вам неизвестно. Тем не менее из факта № 1 вы знаете, что если черепаха проходит через *T* ячеек по циклу, она закончит свой путь на расстоянии *H* ячеек до того места, откуда стар- товала, то есть остановится в начале цикла!

Значение *T* тоже для вас тайна, поэтому переместить «черепаху» на указан- ное расстояние вы опять не можете. Однако если запустить «кролика» от начала связного списка и позволить ему проходить за раз только одну ячейку вместо двух (возможно, он устал, бегая по циклу), он также окажется в начале цикла после того, как пересечет *T* ячеек, то есть оба объекта встретятся снова.

Следующий псевдокод демонстрирует алгоритм высокого уровня.

* 1. Запускаем «черепаху» из начала списка со скоростью одна ячейка за шаг и «кролика» со скоростью две ячейки за шаг.
  2. Если «кролик» найдет ссылку null, список не содержит цикла.
  3. Если «кролик» догонит «черепаху», перезапускаем его из начала списка со скоростью одна ячейка за шаг, в то время как «черепаха» продолжает дви- гаться с в прежнем темпе.
  4. Когда «кролик» и «черепаха» снова встретятся, они будут находиться в на- чале цикла. Оставляем «кролика» в этом месте, чтобы он мог «отдохнуть», пока «черепаха» движется по циклу. Момент, когда указатель Next «чере- пахи» покажет на ячейку, где ждет «кролик», и будет означать конец цикла.
  5. Чтобы прервать цикл, устанавливаем указатель «черепахи» Next на null.

**ВНИМАНИЕ**

Мне никогда не встречалась программа, в которой действительно было нужно использовать алгоритм кролика и черепахи. Если вы внимательны в процессе работы, вашим связным спискам не смогут повредить случайные циклы. Тем не менее обнаружение циклов — довольно популярный вопрос на собеседовании, поэтому об этом методе полезно знать.

#### Циклы в двунаправленных связных списках

Обнаружить циклы в двунаправленных списках очень просто: если они есть, указатель Next возвращается к ранней части списка. При этом Prev будет указы- вать на уже существующую ячейку, а не на ту, которую создал цикл. Таким образом, чтобы выявить цикл, просто пройдите через список и проверьте cell для каждой ячейки: Next.Prev == cell.

Все это предполагает, что ячейки образуют двунаправленный связный спи- сок и цикл, если он существует, является обычным. Если списки Next и Prev полностью несогласованы, метод позволяет обнаружить путаницу, но не помогает исправить ее. Это больше похоже на случай с двумя потоками через одни и те же ячейки, чем на двунаправленный список с циклом.

## Резюме

В этой главе мы рассмотрели связные списки и некоторые операции, которые можно с ними совершать. Вы узнали, что такое однонаправленные, двунаправлен- ные и многопотоковые списки, а также познакомились с основными алгоритмами, позволяющими находить, добавлять и удалять элементы, обнаруживать циклы и избавляться от них.

Работа с указателями — это своего рода подспорье для понимания последующих глав, где рассматриваются деревья (в том числе сбалансированные), сети и другие связные структуры данных. Очередная глава посвящена использованию связных структур данных для работы с разреженными массивами.

## Упражнения

1. В подразделе «Добавление ячеек в конец списка» раздела «Однонаправлен- ные связные списки» текущей главы приводится алгоритм *O*(*N*). Если вы сохраните другую переменную — bottom, — указывающую на последнюю ячейку в списке, то сможете добавлять элементы к концу списка за время *O*(1). Напишите код. Каким образом он усложняет другие алгоритмы, кото- рые находят, удаляют или добавляют элемент в начало (конец) списка? Ре- ализуйте алгоритм для удаления элемента из такого списка.
2. Разработайте алгоритм для нахождения наибольшего элемента в несортиро- ванном однонаправленном списке с ячейками, содержащими целые числа.
3. Напишите алгоритм для добавления элемента в начало двунаправленного списка.
4. Создайте алгоритм для добавления элемента в конец двунаправленного списка.
5. Если вы сравните алгоритмы из упражнений 3 и 4 с алгоритмом InsertCell из раздела «Двунаправленные связные списки» текущей главы, то замети- те, что они очень похожи. Перепишите алгоритмы для упражнений 3 и 4 так, чтобы они вызывали алгоритм InsertCell, вместо того чтобы обновлять ссылки списка напрямую.
6. Напишите алгоритм, удаляющий определенную ячейку из двунаправленного списка. Представьте данный процесс в виде рисунка.
7. Предположим, у вас есть сортированный двунаправленный список с имена- ми. Можно ли улучшить алгоритм, начав поиск не с верхнего, а с нижнего ограничителя? Изменит ли это время его работы?
8. Напишите алгоритм вставки элемента в сортированный двунаправленный список, где верхний и нижний ограничители содержат минимально и мак- симально возможные значения.
9. Реализуйте алгоритм, который определяет, отсортирован ли связный список.
10. У алгоритмов сортировки вставкой и методом выбора теоретическое время работы одинаково — *O*(*N*2). Объясните, почему на практике второй алгоритм работает дольше.
11. Напишите программу, создающую многопотоковый список планет (см. раз- дел «Многопотоковые связные списки» текущей главы). Разрешите поль- зователю выбрать нужный переключатель или пункт выпадающего списка, чтобы отсортировать планеты по разным потокам. (Подсказка: создайте класс Planet c полями Name, DistanceToSun, Mass, Diameter, NextDistance, NextMass и NextDiameter. Затем используйте метод AddPlanetToList, чтобы добавить планету к потокам в отсортированном порядке.)
12. Создайте программу, реализующую алгоритм черепахи и кролика.

# Глава 4 МАССИВЫ

*Массивы* — обыкновенные структуры данных, интуитивно понятные, легкие в использовании и поддерживаемые большинством языков программирования. Они настолько просты, что закономерно усомниться, стоит ли их рассматривать в книге об алгоритмах. В большинстве случаев массивы используются по прямо- му назначению, но иногда им находят особое применение, поэтому все же стоит рассказать о них.

В текущей главе описываются технологии алгоритмизации, которые можно за- действовать для ускоренной работы с массивами, а также для создания массивов с ненулевыми нижними пределами и экономии объема памяти.

## Основные положения

*Массив* — это большое количество непрерывной памяти, доступ к которой осу- ществляется с помощью индексов (их количество указывает на размерность мас- сива). Его легко представить в виде совокупности ящиков, в которых программа хранит значения (рис. 4.1).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Рис. 4.1.** Примеры одно-, двух- и трехмерного массива

При создании массива программа, как правило, задает его размерность и гра- ницы этой размерности. Например, в следующем коде на языке C# объявлен мас- сив numbers, имеющий 10 строк и 20 столбцов.

int[,] numbers = new int[10, 20];

В C# границы массива отсчитываются с нуля, поэтому индексы строк могут принимать значения от 0 до 9, а индексы столбцов — от 0 до 19.

Кроме всего прочего, программа негласно назначает достаточное количество непрерывной памяти для хранения данных массива. Она выглядит как длинный ряд байтов, с каждым из которых сопоставлен определенный индекс массива. В одномерном массиве назначение индекса происходит просто: индекс *i* соответ- ствует записи *i*. В двумерном аналогичная операция осуществляется одним из двух способов: путем развертывания по строкам или по столбцам (рис. 4.2).

* При развертывании по строкам программа привязывает первую строку за- писей массива к первому набору ячеек памяти, вторую — к следующему на- бору ячеек и т. д. по одной строке за раз, пока все записи не будут приведе- ны в соответствие.
* При развертывании по столбцам с первым набором ячеек памяти будет со- относится первый столбец записей массива, со вторым — второй и т. д.



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Строка 0 |  |  | Строка 1 |  |  | Строка 2 |  |

|  |  |  |
| --- | --- | --- |
|  | Строка 0 |  |
|  | Строка  1 |  |
|  | Строка 2 |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Столбец 0 |  |  | Столбец 1 |  |  | Столбец  2 |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Столбец 0 | Столбец 1 | Столбец 2 |
|  |  |  |

**Рис. 4.2.** Соотнесение записей массива с ячейками памяти: по строкам *(вверху)*и столбцам



Эту же мысль о развертывании данных можно перенести и на массивы с боль- шей размерностью. Например, для хранения трехмерного массива в построчном порядке программе понадобится сопоставить первый двумерный слой массива с нулевым третьим индексом. Назначение происходит в построчном порядке, то есть второй слой будет соответствовать третьему индексу под номером 1 и т. д.

Рассмотрим алгоритм назначения трехмерного массива. Предположим, вы определили метод Map2DArray, который создает двумерный массив, и на его основе пытаетесь получить трехмерный массив.

For i = 0 To <верхний предел третьей координаты массива> Map2DArray(<массив, у которого третья координата установлена на i>)

Next i

Приведенный выше код поможет сопоставлять с ячейками памяти массивы и с куда большей размерностью.

Обычно привязка записей массива к ячейкам памяти не зависит от какой бы то ни было программы, поэтому переживать на этот счет не стоит. Достаточно того, что код работает с записями массива, а как они хранятся, вам знать необязатель- но. Тем не менее понимание процесса полезно для создания собственных структур данных с последующим применением в треугольных массивах (о них рассказыва- ется в разделе «Треугольные массивы» текущей главы).

## Одномерные массивы

Алгоритмы, в которых используются *одномерные*, или *линейные*, массивы, как правило, кажутся тривиальными. Их часто предлагают к обсуждению на со- беседованиях по программированию, поэтому остановится на этом вопросе все же стоит. Операции с линейными массивами своего рода подспорье для рабо- ты с более интересными структурами данных, такими как связные списки, сте- ки и очереди.

#### Нахождение элементов

В главе 7 приводится несколько интересных алгоритмов, занимающихся поис- ком целевого элемента в отсортированном массиве. Если же элементы массива не отсортированы, то используется *линейный*, или *исчерпывающий*, *поиск*. При этом рассматривается каждый элемент массива, пока не находится целевой либо не становится очевидным его отсутствие.

Integer: IndexOf(Integer: array[], Integer: target) For i = 0 to array.Length - 1

If (array[i] == target) Return i

Next i

// Целевого элемента в массиве нет. Return -1

End IndexOf

В наихудшем случае целевой элемент может быть самым последним в массиве. Если всего элементов *N*, то алгоритм закончит поиск, пройдясь по каждому из них. Таким образом, время его работы составит *O*(*N*). То же самое произойдет, если целевого элемента в массиве нет. Чтобы проверить каждый элемент в массиве по очереди, в среднем понадобится *N*/2 шагов, что также равняется *O*(*N*).

#### Нахождение минимальной, максимальной и средней величин

В числовом массиве нередко требуется найти минимальную, максимальную или среднюю величину.

Integer: FindMinimum(Integer: array[]) Integer: minimum = array[0]

For i = 1 To array.Length - 1

If (array[i] < minimum) Then minimum = array[i]

Next i

Return minimum End FindMinimum

Integer: FindMaximum(Integer: array[]) Integer: maximum = array[0]

For i = 1 To array.Length - 1

If (array[i] > maximum) Then maximum = array[i]

Next i

Return maximum End FindMaximum

Float: FindAverage(Integer: array[]) Integer: total = 0

For i = 0 To array.Length - 1 total = total + array[i]

Next i

Return total / array.Length End FindMaximum

Как и в случае с нахождением определенного значения в массиве, алгоритм должен просмотреть каждый элемент, поэтому время его работы составит *O*(*N*).

Аналогичным образом можно рассчитать другие статистические величины, скажем, стандартное отклонение или дисперсию. Наибольшую трудность вызывает поиск срединного значения, то есть такого, которое находится посередине в ряду всех значений массива. Например, среди чисел 1, 3, 4, 7, 8, 8, 9 срединным окажется 7,

поскольку есть три числа меньше его (1, 3, 4) и три больше (8, 8, 9). За один проход по массиву вы не получите всей информации, необходимой для определения сре- динного значения, так как для этого нужна более общая информация о величинах.

Один из способов решить подобную задачу — сравнить с каждым отдельно взятым значением оставшиеся значения из массива и проследить за тем, меньше они или больше тестового. Если количество меньших величин окажется равно количеству больших, тестовое значение и будет срединным. Далее приводится основной алгоритм.

Integer: FindMedian(Integer: array[]) For i = 0 To array.Length - 1

// Находим количество элементов больших и меньших, чем array[i]. Integer: num\_larger = 0

Integer: num\_smaller = 0

For j = 0 To array.Length - 1

If (array[j] < array[i]) Then num\_smaller = num\_ smaller + 1

If (array[j] > array[i]) Then num\_larger = num\_larger

+ 1

Next j

If (num\_smaller = num\_larger) Then Return array[i]

End If

Next i

End FindMedian

В приведенном алгоритме есть несколько недостатков. Например, он не срабо- тает в случае, когда в массиве будут присутствовать одинаковые значения. Предпо- ложим такие: 1, 2, 3, 3, 4. Он неэффективен и для массивов с четным количеством элементов: в подобной ситуации срединное значение определяется как среднее двух центральных элементов. Например, в ряду 1, 4, 6, 9 таковым станет число

(4 + 6)/2 = 5.

Несмотря на описанные недостатки алгоритма, проанализируем время его работы. Если в массиве *N* значений, внешний цикл For i выполняется *N* раз. Для каждой из этих итераций внутренний цикл For i срабатывает также *N* раз. Стало быть, во внутреннем цикле совершается *N*  *N* = *N*2 шагов, что делает время работы алгоритма равным *O*(*N*2).

Еще более быстрые алгоритмы занимаются поиском срединного значения в предварительно отсортированном числовом ряду. В главе 6 вы познакомитесь с различными способами сортировки массива, содержащего *N* элементов, за вре- мя *O*(*N*  log *N*) и убедитесь, что оно эффективнее, чем *O*(*N*2).

#### Вставка элементов

Добавить элемент в конец линейного массива довольно просто при условии, что язык программирования может расширить массив на один элемент. Гораздо слож- нее произвести вставку в произвольную позицию. Рассмотрим алгоритм, который добавляет новый элемент в позицию position в линейном массиве.

InsertItem(Integer: array[], Integer: value, Integer: position)

<Расширяем массив, чтобы добавить элемент в конец.>

// Смещаем элементы после целевой позиции,

// чтобы осовбодить место для нового элемента.

For i = array.Length - 1 To position + 1 Step -1 array[i] = array[i - 1]

Next i

// Вставляем новый элемент. array[position] = value

End InsertItem

Обратите внимание, что цикл алгоритма For запускается в конце массива и переходит к началу. Таким образом он заполняет сперва новую позицию в кон- це массива, а затем каждую предыдущую сразу после того, как ее значение было скопировано в другое место.

Если в исходном массиве было *N* элементов, цикл For сработает *N* – position раз. В наихудшем случае, когда элемент понадобится вставить в начало массива (то есть position = 0), цикл будет вызван *N* раз, поэтому время работы такого ал- горитма определяется как *O*(*N*).

**ЗАМЕЧАНИЕ**

Многие языки программирования располагают более совершенными мето- дами удаления блоков памяти. Они заставляют элементы сместиться вниз на одну позицию намного быстрее.

На практике вставка элементов в линейный массив не так уж распространена. Однако используемый метод смещения, который позволяет освободить место для нового элемента, широко применяется в других алгоритмах.

#### Удаление элементов

Удаление *k*-го элемента из массива похоже на вставку. Код сперва смещает все следующие за целевым элементы на одну позицию ближе к началу, а затем изменяет размер массива, чтобы ликвидировать последнюю ненужную запись.

В наихудшем случае, когда нужно удалить первый элемент массива, алгоритму придется сдвинуть все записи. Это означает, что время его работы составит *O*(*N*).

**ЗАМЕЧАНИЕ**

 В некоторых случаях запись можно не удалять, а пометить как неиспользуемую. Например, если в массиве содержатся ссылки или указания на объекты, им

достаточно присвоить значение null. Подобный метод особенно полезен в хеш-таблицах — изменение размеров массива позволяет сэкономить время.

Однако если постоянно помечать записи как неиспользуемые, массив в ко- нечном счете переполнится и, чтобы найти конкретный элемент, вам придется проверять много пустых позиций. В такой ситуации в определенный момент времени нужно произвести сжатие массива и удалить все ненужное.

## Ненулевые нижние пределы

Во многих языках программирования нижним пределом массива любой раз- мерности является 0. Например, линейный массив, состоящий из 10 элементов, может иметь индексы от 0 до 9, но не от 1 до 10. Однако иногда размерность массива удобно рассматривать так, будто его нижние пределы не равны нулю.

Предположим, вы создаете программу, записывающую объемы продаж с 2000 по 2010 г. для десяти работников с идентификационными номерами от 1 до 10. В этом случае лучше объявить массив следующим образом.

Double: sales[1 to 10, 2000 to 2010]

Если нулевой индекс является обязательным для языка программирования, можно перевести удобные для вас пределы в такие, которые начинаются с 0. В сле- дующих двух подразделах объясняется, как использовать ненулевые нижние преде- лы для массивов с двойной и большей размерностью.

#### Двумерные массивы

Работа с массивами, имеющими ненулевые нижние пределы, не так уж сложна. Снова рассмотрим пример, где в качестве исходных данных фигурирует иденти- фикационный номер работника (от 1 до 10) и год (с 2000-го по 2010-й). Поскольку у нас в наличии два диапазона — 10 идентификационных номеров и 11 лет, назначим массив с 10 строками и 11 столбцами.

Double: sales[10, 11]

Чтобы получить доступ к записи для работника e за год y, необходимо рас- считать строку и столбец в текущем массиве.

row = e - 1 column = y – 2000

Сейчас программа работает с записью array[row, column]. Но вы можете упростить задачу: заключить массив в класс и создать конструктор, устанавливающий объекту нужные пределы, на основе которых будут рассчитываться строки и столбцы.

Некоторые языки программирования позволяют сделать методы get и set ин- дексами класса и работать с объектами почти так же, как если бы они были масси- вами. Например, в C# можно использовать следующий код, чтобы устанавливать и получать значения в массиве.

array[6, 2005] = 74816;

MessageBox.Show(

«In 2005 employee 6 had» + array[6, 2005].ToString() + « in sales.»

Поскольку каждый язык программирования специфичен, более подробный код привести нереально. На сайте с материалами книги вы можете загрузить пример программы TwoDArray, чтобы увидеть полную реализацию алгоритма на языке C#.

#### Массивы высокой размерности

Метод, описанный в предыдущем подразделе, хорошо использовать, когда из- вестна размерность массива. В противном случае возникает сложность, связанная с тем, что для *N* размерностей вам понадобится назначить *N*-мерный массив, чтобы он мог содержать все данные. Конечно, вы можете создать отдельные классы для работы с двумя, тремя, четырьмя и более размерностями, но лучше найти более общий способ.

Вместо того чтобы хранить значения в двумерном массиве, попробуйте запако- вать их в одномерный в построчном порядке. Последний массив при этом должен быть достаточно большим, чтобы вместить все элементы. Так, если имеется *N* строк в *M* столбцах, то лучше задать массив с количеством записей *N**M*.

Double: values[N \* M]

Чтобы найти положение элемента в массиве, рассчитайте строку и столбец, как показывалось выше. Если элемент отвечает работнику e и году y, то расчет будет выглядеть следующим образом.

row = e - <нижняя граница массива с ИД работников> column = y - <нижняя граница массива со значениями года>

Теперь когда известны строка и столбец элемента, необходимо найти его ин- декс в массиве values. Первым делом нужно выяснить, сколько полных строк помещается в массиве перед необходимым элементом. Если элемент находится в строке r, значит, перед ним r полных строк, пронумерованных как 0, 1, ..., r – 1. Поскольку в каждой строке <row size> элементов, то перед нужной нам записью их r<row size>. На следующем этапе стоит определить количество элементов, которые предшествуют целевому значению и находятся с ним в одной строке. Если

номер столбца с нужной нам записью равен c, то перед ней в той же строке стоит c элементов: 0, 1, ..., c – 1. Таким образом, общее количество предшествующих элементов определяется следующей формулой.

index = row  <row size> + column

Теперь вы можете найти элемент values[index].

Предложенная технология немного сложнее той, что описана в предыдущем подразделе, зато применима для любой размерности. Предположим, вы хотите соз- дать массив размерности *N*, сохранив нижние пределы в массиве lower\_bounds, а верхние — в массиве upper\_bounds. Первый шаг, который понадобится сде- лать, — создать одномерный массив с достаточным количеством места для хране- ния всех значений. Для этого просто вычтите каждый нижний предел из каждого верхнего предела, чтобы увидеть, насколько «широким» должен быть массив этой размерности, а затем перемножьте полученные «ширины».

Integer: ArraySize(Integer: lower\_bounds[], Integer: upper\_bounds[]) Integer: total\_size = 0

For i = 0 To lower\_bounds.Length - 1

total\_size = total\_size \* (upper\_bounds[i] - lower\_bounds[i])

Next i

Return total\_size End ArraySize

Следующий этап — привести в соответствие строки и столбцы с позициями в одномерном массиве. Сделать это немного сложнее. Вспомните, как в предыду- щем примере строка и столбец привязывались к индексу в массиве values. Сна- чала код определял количество полных строк в массиве, а затем умножал его на ко- личество элементов в строке. Затем он добавлял единицу к каждой позиции пред- шествующего элемента в той строке, где находилась нужная запись.

Проделать то же самое для трехмерного массива не намного слож- нее. На рисунке 4.3 представлен трехмерный массив размерностью 443 (высотастрокастолбец). Нужный нам элемент с координатами (1, 1, 3) обо- значен серым цветом.

1, 3)

(1,

} Слой

Высота

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Столбец

**Рис. 4.3.** Пример трехмерного массива

Чтобы соотнести элемент (1, 1, 3) с индексом в массиве values, вначале стоит определить количество целых слоев, которые ему предшествуют. В представленных

координатах высота равна 1, поэтому перед нужной нам записью есть один пол- ный слой. Его размер равен <row size><column size>. Если переобозначить координаты как (h, r, c), то количество предшествующих элементов в имеющихся слоях можно найти по следующей формуле.

index = h  <row size>  <column size>

Далее необходимо определить, сколько элементов стоит перед нужным нам в полных строках. Согласно второй координате, полная строка всего одна. С уче- том принятых обозначений необходимо прибавить к индексу r длин строк:

index = index + r  <row size>

Нужно добавить все элементы, предшествующие целевому в его столбце.

index = index + c

Метод можно усовершенствовать и приспособить для работы с еще больши- ми размерностями. Чтобы облегчить расчет индексов в массиве values, создай- те массив slice\_sizes, сохранив в него размер слоя для каждой размерности. В трехмерном массиве этими величинами будут <row size><column size>,

<column size> и 1.

При работе с более сложно организованными массивами размер слоя можно найти, умножив размер следующего слоя на размер текущей размерности. Напри- мер, для четырехмерного массива расчет будет выглядеть так: <height size> <row size><column size>.

Теперь, зная эти особенности, вы готовы увидеть полный алгоритм. Предполо- жим, массив bounds содержит изменяемые нижний и верхний пределы для жела- емого *N*-мерного массива. Следующий псевдокод инициализирует массив.

InitializeArray(Integer: bounds[])

// Устанавливаем пределы.

Integer: NumDimensions = bounds.Length / 2 Integer: LowerBound[NumDimensions] Integer: SliceSize[NumDimensions]

// Инициализируем LowerBound и SliceSize. Integer: slice\_size = 1

For i = NumDimensions - 1 To 0 Step -1 SliceSize[i] = slice\_size

LowerBound[i] = bounds[2 \* i]

Integer: upper\_bound = bounds[2 \* i + 1]

Integer: bound\_size = upper\_bound - LowerBound[i] + 1 slice\_size \*= bound\_size

Next i

// Определяем место для всех элементов. Double: Values[slice\_size]

End InitializeArray

Чтобы рассчитать количество измерений, данный код делит количество зна- чений в массиве bounds на 2. Затем создается массив LowerBound для хранения нижних пределов и массив SliceSize с размерами слоев в разных размерностях.

После код задает значение slice\_size, равное 1. Это размер слоя в наивысшей размерности, являющийся столбцом в предыдущем примере.

На следующем этапе в работу вступает цикл, который проходит по размер- ностям, начиная с наивысшей и заканчивая нулевой (это соответствует пути от столбца к строке по высоте в прошлом случае). Код устанавливает текущий размер слоя на slice\_size и сохраняет нижний предел размерности, а затем умножает данное значение на показатель нынешней размерности, чтобы получить величину слоя для размерности поменьше.

По окончании цикла в slice\_size будут содержаться перемноженные зна- чения всех размерностей массива. Это и есть общее количество элементов, кото- рые код использует для объявления массива Values, где будут размещаться дан- ные начального массива.

В следующем псевдокоде задействуются массивы LowerBound и SliceSize для приведения в соответствие индексов из массива indices с индексами из мас- сива Values.

Integer: MapIndicesToIndex(Integer: indices[]) Integer: index = 0

For i = 0 to indices.Length - 1 index = index +

(indices[i] - LowerBound[i]) \* SliceSize[i]

Next i Return index

End MapIndicesToIndex

Код задает начальное значение index, равное 0, а затем проходит циклом по размерностям массива. Для каждой размерности он умножает количество находя- щихся в ней слоев на размер слоя и прибавляет результат к index. После того как код пройдет по всем размерностям, в index будет размещаться индекс элемента из массива Values.

Вы можете усовершенствовать данный алгоритм, заключив его в класс. Конструк- тор укажет объекту, какие размерности нужно использовать. В зависимости от языка программирования вы сможете создать методы доступа get и set, так называемые *аксессоры*. Таким образом, программа будет расценивать объект как массив.

Чтобы увидеть применение этого алгоритма на языке C#, загрузите програм- му NDArray с сайта с материалами книги.

## Треугольные массивы

В некоторых случаях пространство можно сэкономить, используя вместо обыч- ных прямоугольных массивов *треугольные* (рис. 4.4). В *треугольном массиве* эле- менты, расположенные над диагональю (там, где индекс столбца больше, чем индекс строки), имеют значения по умолчанию: 0, null или пустое место.

Возьмем, например, *матрицу смежности*, которая представляет связи между точками в сети. В качестве последней могут выступать авиамаршруты, проложен- ные между разными аэропортами. Если совершается полет от аэропорта i до аэро- порта j, то запись массива connected[i, j] приравнивается к 1. Предположим

также, что существует обратный маршрут от аэропорта j до аэропорта i, это дает нам запись connected[i, j] = connected[j, i]. Получается, что нет не- обходимости хранить connected[i, j] и connected[j, i], поскольку они одинаковы. И следовательно, программа может сэкономить пространство, преоб- разовав матрицу смежности в треугольный массив.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Рис. 4.4.** Пример треугольного массива

**ЗАМЕЧАНИЕ**

Возможно, не стоит создавать треугольный массив 33 или 100100, посколь- ку в этом случае вы сохраните всего лишь 3 или 4960 записей соответствен- но. Так вам удастся сэкономить совсем не много памяти, а работать будет сложнее, чем с обычным массивом, а вот треугольный массив размерностью 10 00010 000 сохранит порядка 50 млн записей, поэтому может быть полезен.

Построить треугольный массив несложно: достаточно упаковать все исходные значения в одномерный массив, пропустив ненужные записи. Трудности могут возникнуть при определении того, насколько большим должен быть одномерный массив и как назначить и связать строки и столбцы с его индексами.

В таблице 4.1 приведено количество записей, необходимое для построения треугольных массивов различных размеров.

**Òàáëèöà 4.1.** Ïîñòðîåíèå òðåóãîëüíûõ ìàññèâîâ

|  |  |
| --- | --- |
| **Количество записей** | **Количество строк в массиве** |
| 1 | 1 |
| 3 | 2 |
| 6 | 3 |
| 10 | 4 |
| 15 | 5 |
| 21 | 6 |
| 28 | 7 |

Если проанализировать таблицу 4.1, можно найти закономерность: количество ячеек для *N* строк равно количеству ячеек, необходимому для *N* – 1 строк плюс *N*.

Взглянув на треугольный массив более внимательно, вы поймете, что почти по- ловина его записей умещается в квадратном массиве с таким же количеством строк. Если строк *N*, то в квадратном массиве будет *N*2 записей, поэтому очевидно, что их количество в соответствующем треугольном массиве будет также включать величи- ну *N*2. Взяв обычное квадратное уравнение *A*  *N*2 + *B*  *N* + *C* и подставив значения из таблицы 4.1, вы определите *A*, *B* и *C* и придете к выводу, что уравнение сводится к (*N*2 + *N*)/2. Таким образом, чтобы построить треугольный массив с количеством строк *N*, нужно назначить одномерный массив, содержащий (*N*2 + *N*)/2 элементов.

На следующем этапе следует выяснить, как привести в соответствие строки и столбцы с индексами одномерного массива. Чтобы узнать индекс для записи со строкой r и столбцом c, нужно определить, сколько элементов ей предшествует в одномерном массиве.

Посмотрите на рисунок 4.5 и обратите внимание на количество записей, идущих перед ячейкой с координатами (3, 2). Полные строки выделены жирной линией. Количество записей в них равноценно количеству записей в треугольном массиве с тремя строками, а вы уже знаете, как его рассчитывать.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  | (3, 2) |  |  |
|  |  |  |  |  |

**Рис. 4.5.** Поиск индекса записи в треугольном массиве

Элементы, идущие перед целевой записью в ячейке (3, 2) и не входящие в полную строку, размещаются слева. В данном примере нужная нам запись рас- положена в столбце 2, а перед ней в той же строке стоят еще два элемента. В общем случае формула индекса для записи со строкой r и столбцом c будет выглядеть так: ((*r* – 1)2 + (*r* – 1))/2 + c.

Эти две формулы облегчают работу с треугольными массивами. Используйте первую, чтобы узнать, сколько элементов должен содержать массив с количеством строк *N*, а вторую, чтобы привести в соответствие строки и столбцы с индексами в одномерном массиве.

Integer: FindIndex(Integer: r, Integer: c)

Return ((r - 1) \* (r - 1) + (r - 1)) / 2 + c End FindIndex

Приведенный алгоритм легко упростить, заключив треугольный массив в класс. Создав методы get и set, вы сможете рассматривать данный массив как обычный.

Последнее, что стоит рассмотреть, — как класс треугольного массива восприни- мает запросы к несуществующим элементам. Что, например, должно происходить, если программа попытается получить доступ к записи (1, 4) в верхней половине треугольника? Все зависит от обстоятельств: вы можете вернуть значение по умол- чанию, поменять местами строку и столбец и вернуть полученное значение или же сгенерировать исключение.

## Массивы с разрывом

Безусловно, треугольные массивы позволяют программе существенно эконо- мить память. Но если вы знаете, что записей будет совсем мало, эти расходы можно сократить еще более значительно.

Вернемся к матрице смежностей с авиалиниями, где 1 в записи [i, j] обо- значает полет между городами i и j. Полетов может быть всего 600, и соединять они будут только 200 городов. В таком случае в массиве из 40 000 записей окажет- ся 600 ненулевых значений. Даже если полеты будут симметричными (на каждый полет из i в j будет приходится полет из j в i) и вы сохраните их в треугольном массиве, из 20 100 записей ненулевыми окажутся только 300. Это значит, что поч- ти 99 % массива не используется.

*Массив с разрывом* позволяет сэкономить пространство, поскольку не пред- ставляет недостающие записи. Если программа находит элемент в таком массиве, она возвращает его значение; если элемент отсутствует, возвращается значение по умолчанию (в примере матрицы смежности это 0).

Массив с разрывом используется для создания связного списка из связных списков. В таком случае новый список хранит информацию о строках. В нем каж- дый элемент указывает на другой связный список, содержащий записи из столбцов массива для этой строки.

Вы можете построить массив с разрывом с двумя классами ячеек: ArrayRow для представления самой строки и ArrayEntry для представления ее значений. Первый класс будет хранить номер ячейки, ссылку на следующий ArrayRow и на первый ArrayEntry в этой строке. Ниже показано расположение данно- го класса.

ArrayRow:

Integer: RowNumber ArrayRow: NextRow ArrayEntry: RowSentinel

Класс ArrayEntry содержит количество столбцов в записи, вне зависимости от значения в массиве, а также ссылку на объект ArrayEntry в этой строке. Ниже показано расположение класса ArrayEntry, где T — любой тип данных, которые могут содержаться в массиве.

ArrayEntry:

Integer: ColumnNumber T: Value

ArrayEntry: NextEntry

Чтобы было легче добавлять и удалять строки, их список может начинаться с ограничителя, как и список значений в конкретной строке. На рисунке 4.6 показан массив с разрывом, ограничители закрашены серым цветом.

Запись массива

Запись массива

Строка массива

Строка массива

Запись массива

Запись массива

Строка массива

Запись массива

Запись массива

Строка массива

Запись массива

Запись массива

Запись массива

Строка массива

Запись массива

Запись массива

**Рис. 4.6.** Пример массива с разрывом

Хранение объектов ArrayRow в возрастающем порядке RowNumber помога- ет определить, когда значение в массиве отсутствует. Например, если вы ищете в списке конкретный номер строки и находите объект ArrayRow, имеющий ниж- ний RowNumber, то знайте, что искомого номера строки в массиве нет. По этой же причине объекты ArrayEntry хранятся в возрастающем порядке ColumnNumber.

Стоит отметить, что объекты RowEntry, расположенные на рисунке 4.6 по одной вертикали, не обязательно представляют одни и те же столбцы. Так, первый объект RowEntry в первой строке может находиться в столбце 100, а аналогичный ему по счету объект во второй строке — в столбце 50.

На первый взгляд схема на рисунке 4.6 выглядит несколько запутанной, но ей легко пользоваться. Чтобы найти определенное значение, отыщите нужную стро- ку, спустившись сверху вниз, а затем желаемый столбец в ней. Если вам не удает- ся выявить строку или столбец, значения в массиве нет.

Хотя в массиве с разрывом присутствуют объекты ArrayRow и ограничите- ли, не содержащие значений, он все равно является более эффективным, чем треу- гольный. В наихудшем случае каждая строка будет содержать одно значение. Тогда при размерности массива *N**N* задействуется *N* + 1 объектов ArrayRow и 2*N* объ- ектов ArrayEntry. Среди них только *N* будут содержать фактические величи- ны, а остальные станут ограничителями или послужат для навигации по массиву. В общем случае доля объектов со значениями окажется *N*/(*N* + 1 + 2*N*) = *N*/(3*N* +

1. или приблизительно 1/3. Сравните это с описанным ранее треугольным масси- вом, который был пуст почти на 99 %.

В случае со структурой данных, показанной на рисунке 4.6, вам все же понадо- бится написать алгоритмы, чтобы осуществить три типа операций.

* 1. Получить значение в строке и столбце или вернуть значение по умолчанию, если таковых нет.
  2. Установить значение для строки и столбца.
  3. Удалить значение в строке и столбце.
  4. Составить эти алгоритмы будет немного легче, если сначала определить ме- тоды для нахождения нужных строки и столбца.

#### Нахождение строки и столбца

Чтобы отыскать значение удобно использовать метод FindRowBefore. Он ищет объект ArrayRow *перед* местом, где должна быть целевая строка. Если ее в массиве нет, возвращается ArrayRow перед тем местом, где она могла бы быть.

ArrayRow: FindRowBefore(Integer: row, ArrayRow: array\_row\_sentinel) ArrayRow: array\_row = array\_row\_sentinel

While (array\_row.NextRow != null) And

(array\_row.NextRow.RowNumber < row) array\_row = arrayRow.NextRow

End While

Return array\_row End FindRowBefore

Этот алгоритм приравнивает переменную array\_row к ограничителю стро- ки массива, затем продвигает ее к следующему объекту array\_row в списке, пока тот не станет равен null или RowNumber следующего объекта не будет как мини- мум таким же, как число целевой строки.

Ситуация, когда следующий объект равняется null, говорит о том, что про- грамма просмотрела весь список строк и не нашла нужной. Если бы такая строка была, она располагалась бы после текущего объекта array\_row.

Алгоритм найдет целевую строку при условии, что значение следующего объ- екта RowNumber равно ей. Если оно больше, значит, такой строки в массиве нет. В случае присутствия она размещалась бы за текущим объектом array\_row.

Подобным образом можно определить метод FindColumnBefore для нахож- дения объекта ArrayEntry перед местом, где должен быть целевой столбец. Здесь в качестве параметров выступают число целевого столбца и ограничитель строки, в котором нужно это число искать.

FindColumnBefore(Integer: column, ArrayEntry: row\_sentinel) ArrayEntry: array\_entry = row\_sentinel

While (array\_entry.NextEntry != null) And

(array\_entry.NextEntry.ColumnNumber < column)) array\_entry = array\_entry.NextEntry;

Return array\_entry End FindColumnBefore

Если в массиве находится *N* объектов ArrayRow, то время работы метода

FindRowBefore составит *O*(*N*). При условии что строка содержит *M* элементов,

наиболее отличных от заданных по умолчанию, время метода FindColumnBefore окажется равным *O*(*M*). Более точная эффективность методов рассчитывается исходя из количества и распределения значений в массиве, отличных от заданных по умолчанию.

#### Получение значения

Получить значение из массива помогут методы FindRowBefore

и FindColumnBefore.

GetValue(Integer: row, Integer: column)

// Находим строку.

ArrayRow: array\_row = FindRowBefore(row) array\_row = array\_row.NextRow

If (array\_row == null) Return default

If (array\_row.RowNumber > row) Return default

// Находим столбец в целевой строке. ArrayEntry: array\_entry =

FindColumnBefore(column, array\_row.RowSentinel) array\_entry = array\_entry.NextEntry

If (array\_entry == null) Return default

If (array\_entry.ColumnNumber > column) Return default Return array\_entry.Value

End GetValue

Этот алгоритм использует FindRowBefore, чтобы установить array\_row на строку перед целевой. Затем он продвигает array\_entry к следующему столбцу, возможно, к целевому. Если array\_entry равен null либо ссылается на непра- вильную колонку, метод GetValue возвращает значение массива по умолчанию. В случае когда алгоритм доходит до строки, он обнаруживает правильный объект ArrayEntry и возвращает значение объекта.

В приведенном коде вызываются методы FindRowBefore и FindColumnBefore. Если в массиве *N* строк с *M* значениями, отличными от заданных по умолчанию, то общее время работы метода GetValue составит *O*(*N* + *M*). Это намного боль- ше времени *O*(1), которое требуется для получения значения из обычного масси- ва, однако массив с разрывом использует гораздо меньше места.

#### Установка значения

Установка значения подобна нахождению, за исключением того, что алгоритм должен вставить новую строку или столбец в массив, если это необходимо.

SetValue(Integer: row, Integer: column, T: value)

// Если добавляемое значение является значением по умолчанию,

// мы не устанавливаем его, а удаляем. If (value == default)

DeleteEntry(row, column) Return

End If

// Находим строку перед целевой строкой. ArrayRow: array\_row = FindRowBefore(row)

// Если целевой строки нет, добавляем ее.

If (array\_row.NextRow == null) Or (array\_row.NextRow.RowNumber > row)

ArrayRow: new\_row

new\_row.NextRow = array\_row.NextRow array\_row.NextRow = new\_row

End If

ArrayEntry: sentinel\_entry new\_row.RowSentinel = sentinel\_entry sentinel\_entry.NextEntry = null

// Переходим к целевой строке. array\_row = array\_row.NextRow

// Находим столбец перед целевым столбцом. ArrayEntry: array\_entry =

FindColumnBefore(column, array\_row.RowSentinel)

// Если целевого столбца нет, добавляем его.

If (array\_entry.NextEntry == null) Or

(array\_entry.NextEntry.ColumnNumber > column)

ArrayEntry: new\_entry

new\_entry.NextEntry = array\_entry.NextEntry array\_entry.NextEntry = new\_entry

End If

// Переходим к целевому элементу. array\_entry = array\_entry.NextEntry

// Устанавливаем значение. array\_entry.Value = value

End SetValue

Первым делом алгоритм проверяет значение, которое устанавливается в масси- ве. Если оно задано по умолчанию, то удаляется из массива, чтобы минимизировать его размер (с этой целью вызывается метод DeleteEntry, описанный в подразделе

«Удаление значения» текущего раздела), и код прекращает работу.

Если новое значение не является значением по умолчанию, алгоритм обраща- ется к методу FindRowBefore, чтобы найти строку перед целевой. Если строка, размещенная за той, что возвращается с помощью FindRowBefore, не является целевой, значит, алгоритм достиг конца списка либо следующая строка идет за це- левой. В обоих случаях алгоритм добавляет новый объект ArrayRow между пред- ыдущей и следующей за ним строками.

Описанный процесс показан на рисунке 4.7. В списке слева целевая строка от- сутствует, она обозначена пунктирным овалом.

Чтобы вставить объект ArrayRow, алгоритм создает новый объект и устанавли- вает его ссылку NextRow на значение array\_row объекта NextRow, а затем при- сваивает новому объекту новый ограничитель строки. Когда список сформирован, он выглядит подобно правой части рисунка 4.7, где ссылка NextRow указывает на новый объект.

После того как алгоритм находит целевую строку или создает ее, он вызывает метод FindColumnBefore, чтобы отыскать объект ArrayEntry, который пред-

ставляет целевой столбец. Если такого объекта не существует, алгоритм создает его и вставляет в связный список объекта ArrayEntry аналогично тому, как до- бавлял ArrayRow. Наконец, алгоритм перемещает переменную array\_entry в ArrayEntry, соответствующую строке, и устанавливает ее значение.

array\_row

Строка массива

Строка массива

**Рис. 4.7.** Вставка новой строки в массиве с разрывом при отсутствии целевой строки



array\_row

new\_row

Строка массива

Строка массива



Алгоритм SetValue может обратиться к алгоритму DeleteEntry, описан- ному в подразделе «Удаление значения» текущего раздела, а тот в свою очередь — к методам FindRowBefore и FindColumnBefore. Если же алгоритм SetValue не вызывает DeleteEntry, методы FindRowBefore и FindColumnBefore прямо или косвенно срабатывают все равно.

Предположим, массив имеет *N* строк с *M* значениями, отличными от за- данных по умолчанию. В таком случае с учетом методов FindRowBefore и FindColumnBefore время работы алгоритма SetValue составит *O*(*N* + *M*).

#### Удаление значения

Алгоритм удаления значения работает по тому же принципу, который исполь- зуется для его получения или установки.

DeleteEntry(Integer: row, Integer column)

// Находим строку перед целевой строкой. ArrayRow: array\_row = FindRowBefore(row)

// Если целевой строки нет, нам не нужно ничего удалять.

If (array\_row.NextRow == null) Or (array\_row.NextRow.RowNumber > row)

Return

// Находим элемент перед целевым столбцом в следующей строке. ArrayRow: target\_row = array\_row.NextRow

ArrayEntry: array\_entry =

FindColumnBefore(column, target\_row.RowSentinel)

// Если целевого элемента нет, нам не нужно ничего удалять.

If (array\_entry.NextRow == null) Or (array\_entry.NextRow.ColumnNumber > column)

Return

// Удаляем целевой столбец.

array\_entry.NextColumn = array\_entry.NextColumn.NextColumn

// Если в целевой строке остались столбцы, мы закончили. If (target\_row.RowSentinel.NextColumn != null) Return

// Удаляем пустую целевую строку. array\_row.NextRow = array\_row.NextRow.NextRow

End DeleteEntry

Приведенный алгоритм вызывает FindRowBefore, чтобы найти строку перед целевой. Если таковой не существует, алгоритму не надо ничего удалять, поэтому он останавливается. Затем алгоритм вызывает FindColumnBefore, чтобы найти столбец перед целевым в нужной строке. Если такового не существует, значит, снова ничего удалять не требуется и код прекращает работу.

Потом алгоритм находит объект ArrayEntry перед целевой записью в связном списке записей строки и удаляет целевую запись, перенаправив ссылку NextColumn с предыдущей записи на ту, что идет после целевой.

Изложенная выше процедура показана на рисунке 4.8. Верхний список является исходным, а переменная array\_entry относится к записи, стоящей перед целевой. Чтобы удалить целевую запись, алгоритм делает так, чтобы ссылка NextColumn указывала на следующую запись.



Запись массива

target\_entry

array\_entry

Запись массива

Строка массива

Строка массива

**Рис. 4.8.** Удаление целевой записи в массиве с разрывом

Запись массива

target\_entry

array\_entry



Запись массива

Алгоритм не меняет ссылку целевой записи NextColumn — она все еще отно- сится к следующей записи. Но поскольку ссылки на целевую запись больше нет, для программы она потеряна.

После того как программа удалила целевую запись из связного списка строки, она проверяет ограничитель строки ArrayRow. Если ссылка объекта NextColumn не null, строка все еще содержит другие записи столбцов, поэтому алгоритм останавли-

вается. Если в целевой строке больше нет записей, алгоритм удаляет ее из связного списка объектов ArrayRow подобно тому, как он делал это для целевого столбца.

**ЗАМЕЧАНИЕ**

 Как только алгоритм удалил строку или столбец, нужно освободить соответ- ствующий участок памяти. Количество совершаемых для этого действий зависит

от используемого языка программирования. Например, программа, написанная на C++, должна явно вызывать функцию free для удаленного объекта, чтобы сделать память пригодной к использованию. У других языков программирования иной подход. Например, в C# и Visual Basic применяется метод сборки мусора. Поэтому в следующий раз, когда уборщик мусора запустится, он автоматически удалит любые объекты, к которым программа больше не имеет доступа.

Алгоритм DeleteEntry вызывает FindRowBefore и FindColumnBefore. Если в массиве имеется *N* строк с *M* значениями, отличными от заданных по умол- чанию, общее время работы метода DeleteEntry составит *O*(*N* + *M*).

## Матрицы

Массивы широко применяются для представления матриц и выполнения с ними различного рода операций. Например, чтобы сложить две матрицы 33, нужно сложить соответствующие записи. Ниже представлен алгоритм, демонстрирующий эту операцию для двух двумерных массивов.

**ЗАМЕЧАНИЕ**

Если вы не знакомы с матрицами и операциями над ними, вы можете прочи- тать статью «Матрица (математика)» на ru.wikipedia.org/wiki/Ìàòðèöà\_(ìàòåìàòèêà).

AddArrays(Integer: array1[], Integer: array2[], Integer: result[]) For i = 0 To <максимальный предел для размерности 1>

For j = 0 To <максимальный предел для размерности 2> result[i, j] = array1[i, j] + array2[i, j]

Next i

Next i

End AddArrays

Следующий алгоритм показывает умножение двух обычных двумерных матриц.

MultiplyArrays(Integer: array1[], Integer: array2[], Integer: result[]) For i = 0 To <максимальный предел для размерности 1>

For j = 0 To <максимальный предел для размерности 2>

// Вычисляем результат [i, j]. result[i, j] = 0

For k = 0 To <максимальный предел для размерности 2> result[i, j] = result[i, j] + array1[i, k] \* array2[k, j]

Next i

Next j

Next k

End MultiplyArrays

Эти алгоритмы работают с треугольными массивами и массивами с разрывом, однако они неэффективны, поскольку проверяют каждый элемент в обоих вход- ных массивах, даже если там записей нет. Например, в треугольном массиве отсут- ствуют все значения [i, j], где j > i, поэтому при сложении или умножении та- ких записей нужно учитывать некоторые особенности. Если предположить, что не- достающие записи содержат 0, их прибавление или умножение не влияет на резуль- тат. Когда записи имеют другое значение по умолчанию, результатом тех же опера- ций будет уже не треугольный массив. Тогда вам может потребоваться полное сло- жение или умножение массивов.

Вместо того чтобы рассматривать каждую запись, алгоритм должен уделить внимание только тем записям, которые действительно существуют. В случае с тре- угольными массивами это не очень сложно. Их сложение и умножения рассматри- вается в упражнениях 12 и 13 соответственно. Если вы не сможете с ними спра- виться, поищите ответ в приложении Б.

Ситуация осложняется, когда речь заходит о массивах с разрывом, хотя по- тенциальная экономия времени при этом имеет большее значение. Например, при сложении двух таких матриц нет необходимости проводить итерацию по строкам и столбцам, которые не существуют в обоих входных массивах. Следующий алго- ритм высокого уровня складывает две матрицы с разрывом.

AddArrays(SparseArray: array1[], SparseArray: array2[],

SparseArray: result[])

// Устанавливаем указатели на списки строк в матрицах. ArrayRow: array1\_row = array1.Sentinel.NextRow ArrayRow: array2\_row = array2.Sentinel.NextRow ArrayRow: result\_row = result.Sentinel

// Повторяем действия, пока в обеих входных строках имеются элементы. While (array1\_row != null) And (array2\_row != null)

If (array1\_row.RowNumber < array2\_row.RowNumber) Then

// Количество строк в array1\_row меньше. Копируем значение.

<Копируем строку array1\_row в результат.> array1\_row = array1\_row.NextRow

Else If (array2\_row.RowNumber < array1\_row.RowNumber) Then

// Количество строк в array2\_row меньше. Копируем значение.

<Копируем строку array2\_row в результат.> array2\_row = array2\_row.NextRow

Else

End If End While

// Количество строк в обеих матрицах одинаковое.

// Складываем значения из обеих матриц.

<Добавляем значения из array1\_row and array2\_row в результат.>

array1\_row = array1\_row.NextRow array2\_row = array2\_row.NextRow

// Копируем все оставшиеся элементы входных матриц. If (array1\_row != null) Then

<Копируем оставшиеся строки array1\_row’s в результат.>

End If

If (array2\_row != null) Then

<Копируем оставшиеся строки array2\_row’s в результат.>

End If End AddArrays

Подобным образом можно написать и алгоритм умножения двух матриц с раз- рывом, не учитывая все недостающие строки и столбцы. Это задание рассматривается в упражнении 15. Если вы не справитесь с ним, то можете найти ответ в приложении Б.

**МАТРИЦЫ С РАЗРЫВОМ, ОТСОРТИРОВАННЫЕ ПО СТОЛБЦАМ**

В некоторых алгоритмах удобнее получить доступ к записям в матрице с разры- вом по столбцам, а не по строкам. Например, при умножении двух трехмерных матриц вы умножаете записи в строках первой матрицы на записи в столбцах второй. Чтобы упростить эту процедуру, можно применять связные списки для представления столбцов вместо строк. Если вам нужно получить доступ к матрице с разрывом в порядке строк и столбцов, воспользуйтесь обоими представлениями.

## Резюме

Обычные массивы интуитивно понятны и легки в использовании, однако в не- которых случаях проще работать с массивом, имеющим ненулевые нижние пределы. Для этого подойдут методы, описанные в одноименном разделе текущей главы. Бывает и так, что обычные массивы оказываются неэффективными. Например, если записи присутствуют только в нижней половине, лучше использовать треугольный массив, чтобы сэкономить приблизительно половину расходуемой памяти, а когда записей еще меньше — обратиться к массивам с разрывом. Конечно, массивы с не- нулевыми нижними пределами, треугольные и с разрывом сложнее обыкновенных, которые привычны для множества языков программирования. Однако в некоторых случаях они более удобны и позволяют экономить память.

Еще одна важная особенность массивов заключается в том, что они предостав- ляют доступ к любому элементу. Можно получить или добавить любую запись, если известен ее индекс. В следующей главе речь пойдет о двух видах контейнеров: стеках и очередях. Подобно массивам эти структуры данных содержат наборы элементов, но методы их вставки и удаления весьма ограничены.

## Упражнения

Звездочкой обозначены задачи повышенной сложности.

1. Напишите алгоритм расчета выборочной дисперсии для одномерного масси- ва, содержащего *N* элементов, которая определяется следующим уравнением:

*N*–1



1

*i*

*S= N*

*i*=0

*(x ––x)*2,

где *–x* — средняя величина значений в массиве, а  — сумма всех значений

*xi* для *i* от 0 до *N* – 1.

1. Создайте алгоритм расчета стандартного отклонения выборки для одномерно- го массива чисел, которое определяется как квадратный корень из дисперсии.
2. Разработайте алгоритм для нахождения медианы отсортированного одно- мерного массива (проверьте, с каким количеством элементов вам приходится иметь дело — с четным или нечетным).
3. В подразделе «Удаление элементов» раздела «Одномерные массивы» теку- щей главы было описано, как удалять элемент из линейного массива. На- пишите соответствующий псевдокод.
4. Треугольные массивы иногда называются нижними треугольными, посколь- ку все значения хранятся в нижней левой половине. Попробуйте получить верхний треугольный массив со значениями в верхнем правом углу.
5. Попытайтесь из нижнего треугольного массива создать верхний, в котором записи будут хранится в верхней левой половине. Какая связь между строкой и столбцом для каждого элемента такого массива?
6. Предположим, что вам нужно провести диагональ в прямоугольном (не ква- дратном) массиве из левого верхнего угла в правый нижний. Напишите алгоритм, который приравнивает записи вдоль диагонали или под ней к 1, а над диагональю к 0.
7. Диагональ в прямоугольном массиве начинается с правого верхнего и идет к левому нижнему. Разработайте алгоритм, присваивающий записям вдоль диагонали или над ней 1, а под диагональю 0.
8. Напишите алгоритм, где значение каждого элемента прямоугольного масси- ва — это расстояние до ближайшего края массива.

10\*. Преобразуйте метод построения треугольных массивов для трехмерных че- тырехгранных, содержащих записи value[i, j, k], где j  i и k  j. Как бы вы усовершенствовали этот метод для еще больших размерностей?

1. Как создать треугольный массив с разрывом?
2. Напишите алгоритм для сложения двух треугольных массивов.
3. Разработайте алгоритм для умножения двух треугольных массивов.
4. Приведенный алгоритм сложения двух матриц с разрывом является высоко- уровневым. Допишите необходимые команды вместо инструкций внутри угловых скобок. (Подсказка: для копирования записей из одного списка в другой можно создать метод CopyEntries, а для комбинирования запи- сей в двух строках с одинаковым номером — метод AddEntries.)
5. Напишите высокоуровневый алгоритм для умножения двух матриц с раз- рывом, которые имеют значение по умолчанию, равное 0.

# Глава 5 СТЕКИ И ОЧЕРЕДИ

Стеки и очереди — относительно простые структуры данных, которые хранят объекты в порядке «первым пришел, первым вышел» или «последним пришел, пер- вым вышел». По мере необходимости их можно расширять, чтобы добавить допол- нительные элементы, подобно связным спискам (см. главу 3). По сути, для созда- ния стеков и очередей используются связные списки.

Стеки и очереди идеально подходят для моделирования реальных ситуаций — каналов обслуживания в банке или супермаркете. Однако чаще всего в них хранят объекты, которые будут в дальнейшем обрабатываться с помощью других алгорит- мов, например сетевых алгоритмов кратчайшего пути.

В этой главе вводятся понятия «стек» и «очередь», а также даются сопутству- ющая терминология и методы, которые стоит применять для их создания.

## Стеки

*Стек* — это структура данных, в которой элементы добавляются и удаляются в порядке «последним пришел, первым ушел». Из-за такого поведения их иногда называют *списками LIFO*, или просто *LIFO* (от англ. *last in first out*).

Стек легко представить в виде стопки книг, лежащей на столе: можно доба- вить или убрать книгу сверху, но нельзя вынуть ее из середины или снизу, не раз- рушив всю конструкцию. Еще один хороший пример — стопка тарелок с пружин- ной фиксацией, которая используется в кафетериях (рис. 5.1). Если добавить та- релку сверху, то пружина сожмется и вся стопка окажется вровень со столешни- цей. То же самое произойдет при снятии тарелки, когда пружина ослабнет.

Вдавливание

Выталкивание

**Рис. 5.1.** Представление стека в виде стопки тарелок

Такую структуру данных иногда еще называют *стеком магазинного типа*; до- бавление объекта — *вдавливанием* в стек, а удаление — *выталкиванием* из стека. Класс стека обычно предусматривает методы Push и Pop для добавления элементов и их удаления соответственно.

В следующем разделе описывается несколько общепринятых методов создания стека.

#### Стеки связных списков

Стек легко реализовать с помощью связного списка: метод Push добавит в на- чало ячейку, а метод Pop удалит ее оттуда. В следующем псевдокоде представлен алгоритм вставки элемента в стек на основе связного списка.

Push(Cell: sentinel, Data: new\_value)

// Создаем ячейку для записи нового значения.

Cell: new\_cell = New Cell new\_cell.Value = new\_value

// Добавляем новую ячейку в связный список. new\_cell.Next = sentinel.Next

sentinel.Next = new\_cell End Push

Ниже приводится алгоритм удаления элемента из стека на основе связного списка.

Data: Pop(Cell: sentinel)

// Проверяем, есть ли элемент для удаления.

If (sentinel.Next == null) Then <Формируем исключение.>

// Получаем значение верхней ячейки. Data: result = sentinel.Next.Value

// Удаляем верхнюю ячейку из связного списка. sentinel.Next = sentinel.Next.Next

// Возвращаем результат. Return result

End Pop

Рассмотренный процесс показан на рисунке 5.2. Верхний фрагмент отра- жает состояние стека после того, как программа вставила в него буквы *A*, *P*, *P*, *L* и *E*; средний — тот же стек после добавления буквы *S*, нижний — после ее же удаления.

**ЗАМЕЧАНИЕ**

Более подробно о связных списках рассказано в главе 3.

При работе со связными списками как вставка, так и удаление элементов зани- мают время *O*(1), то есть обе операции являются довольно быстрыми. Кроме того, список не требует дополнительной памяти — достаточно ссылок между ячейками, поэтому существенно экономит ресурсы.

Ограничитель



*A*

*P*

*L*

*Е*

*P*

Вдавливание



S

Ограничитель



*P*

*L*

*Е*

*P*



*A*

*S*

Выталкивание



*S*

Ограничитель



*A*

*P*

*L*

*Е*

*P*

**Рис. 5.2.** Стек со связным списком

#### Стеки массивов

Реализация стека на основе массива не сложнее, чем на базе связного списка. Назначьте достаточное место для массива, чтобы сохранить элементы, которые собираетесь включить в стек, и используйте переменную, чтобы отслеживать сле- дующую пустую позицию в стеке.

Ниже представлен алгоритм вставки элемента в стек на основе массива.

Push(Data: stack\_values [], Integer: next\_index, Data: new\_value)

// Проверяем, есть ли место для добавления нового элемента.

If (next\_index == <длина stack\_values>) Then <Формируем исключение.>

// Добавляем новый элемент. stack\_values[next\_index] = new\_value

// Увеличиваем next\_index. next\_index = next\_index + 1

End Push

Следующий псевдокод содержит алгоритм удаления элемента из стека на основе массива.

Data: Pop(Data: stack\_values[], Integer: next\_index)

// Проверяем, есть ли элемент для удаления.

If (next\_index == 0) Then <Формируем исключение.>

// Уменьшаем next\_index. next\_index = next\_index – 1

// Возвращаем верхнее значение. Return stack\_values[next\_index]

End Pop

Описанный процесс показан на рисунке 5.3. Верхний фрагмент — это стек, в который добавлены буквы *A*, *P*, *P*, *L* и *E*; средний и нижний фрагменты — стек после вставки и удаления буквы *S* соответственно.

Nextlndex



Nextlndex

Вдавливание

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *A* | *P* | *P* | *L* | *E* |  |  |  |



*S*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *A* | *P* | *P* | *L* | *E* | *S* |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *A* | *P* | *P* | *L* | *E* |  |  |  |

**Рис. 5.3.** Создание стека на основе массива



Nextlndex

Выталкивание

*S*

В случае с массивами операции вдавливания и выталкивания элементов занима- ют время *O*(1) и являются довольно быстрыми. Кроме того, установка и получение значения из массива происходят быстрее, чем создание новой ячейки в связном списке, а потому такой метод более эффективен.

Основанный на массиве стек не нуждается и в дополнительной памяти для хранения ссылок между ячейками, но ему требуется место для новых элементов. Объем такого места зависит от используемого приложения и от того, известно ли вам заранее количество записей. Если нет — при необходимости можно изменить размер массива, но это займет дополнительное время. Если в массиве *N* элементов, то при изменении его размера понадобится выполнить *O*(*N*) шагов, чтобы скопи- ровать уже имеющиеся записи.

В зависимости от того, как используется стек, резервирование места для до- полнительных элементов может быть крайне неэффективным. Предположим, что в какой-то момент алгоритму понадобится сохранить 1000 записей в стеке, хотя большую часть времени ему нужно всего несколько. В таком случае массив будет тратить неоправданно много ресурсов памяти. Если же вы знаете, что в стеке всег- да будет храниться всего нескольких элементов, то имеет смысл выстраивать его на основе массива.

#### Двойные стеки

Предположим, алгоритм нуждается в использовании двух стеков с ограничен- ным общим размером. Вы можете сохранить оба в одном массиве с разных сторон таким образом, чтобы они расширялись в середину (рис. 5.4).

Nextlndex1 Nextlndex2



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *A* | *P* | *P* | *L* | *E* |  |  |  |  |  | *A* | *N* | *A* | *N* | *A* | *B* |

**Рис. 5.4.** Массив с двумя стеками, имеющими ограниченный общий размер

В следующем псевдокоде показаны алгоритмы вдавливания и выталкивания элементов из двух стеков, содержащихся в одном массиве. Чтобы упростить запись, массив Values и переменные NextIndex1 и NextIndex2 хранятся за пределами методов Push.

Data: StackValues[<max items>] Integer: NextIndex1, NextIndex2

// Инициализируем массив. Initialize()

NextIndex1 = 0

NextIndex2 = <длина StackValues> - 1 End Initialize

// Добавляем элемент в начало стека. Push1(Data: new\_value)

// Проверяем, есть ли место для добавления нового элемента. If (NextIndex1 > NextIndex2) Then <Формируем исключение.>

// Добавляем новый элемент. StackValues[NextIndex1] = new\_value

// Увеличиваем NextIndex1. NextIndex1 = NextIndex1 + 1

End Push1

// Добавляем элемент в конец стека. Push2(Data: new\_value)

// Проверяем, есть ли место для добавления нового элемента. If (NextIndex1 > NextIndex2) Then <Формируем исключение.>

// Добавляем новый элемент. StackValues[NextIndex2] = new\_value

// Уменьшаем NextIndex2. NextIndex2 = NextIndex2 - 1

End Push2

// Удаляем элемент из начала стека. Data: Pop1()

// Проверяем, есть ли элемент для удаления.

If (NextIndex1 == 0) Then <Формируем исключение.>

// Уменьшаем NextIndex1. NextIndex1 = NextIndex1 – 1

// Возвращаем верхнее значение. Return StackValues[NextIndex1]

End Pop1

// Удаляем элемент из конца стека. Data: Pop2()

// Проверяем, есть ли элемент для удаления. If (NextIndex2 == <длина StackValues> - 1) Then <Формируем исключение.>

// Увеличиваем NextIndex2. NextIndex2 = NextIndex2 + 1

// Возвращаем верхнее значение. Return StackValues[NextIndex2]

End Pop2

#### Алгоритмы с использованием стеков

Стеки используются во многих алгоритмах, например для нахождения крат- чайшего пути (см. главу 13). Описание некоторых их них приводится в следующих подразделах.

#### Реверсирование массива

Перевернуть массив с помощью стека довольно просто: добавьте все элементы в стек, а затем вытолкните их обратно. Благодаря принципу «последним пришел, первым ушел» они вернутся в обратном порядке.

ReverseArray(Data: values[])

// Добавляем элементы из массива в стек. Stack: stack = New Stack

For i = 0 To <количество значений> - 1 stack.Push(values[i])

Next i

// Выталкиваем элементы из стека в массив.

For i = 0 To <количество значений> - 1 values[i] = stack.Pop()

Next i

End ReverseArray

Если в массиве *N* элементов, алгоритм пройдет 2*N* шагов. Таким образом, время его работы составит *O*(*N*).

#### Сортировка вагонов поезда

Представьте, что из депо выехал поезд, к которому прицеплены вагоны, иду- щие в разные пункты назначения. Вы должны воспользоваться перегонами, чтобы правильно сформировать состав.

В левой части рисунка 5.5 показан поезд с вагонами, направляющимися в го- рода 3, 2, 1, 3, 2. Локомотив может подойти к перегону и прицепить свой крайний правый вагон к любому стоящему там крайнему левому вагону. Его цель — вы- полнить сортировку.

Перегоны

2

3

1

2

Входной путь Выходной путь

3

**Рис. 5.5.** Моделирование сортировки вагонов на основе стека

Задачу удобно решать с помощью нескольких стеков. Один из них будет представлять прибывающий поезд. Используя метод Pop, он удалит вагон с правой стороны, а благодаря методу Push прицепит его к правому краю локомотива.

Другие стеки представляют собой перегон и отходящий путь. Их методы

Push переводят вагон на левый конец пути, а методы Pop убирают его оттуда.

В следующем псевдокоде программа использует стеки для формирования со- става, показанного на рисунке 5.5. Здесь train — это прибывающий поезд, track1 и track2 — верхний и нижний перегоны, output — отходящий путь.

holding1.Push(train.Pop()) // Шаг 1: Переводим вагон 2 на верхний перегон. holding2.Push(train.Pop()) // Шаг 2: Переводим вагон 3 на нижний перегон. output.Push(train.Pop()) // Шаг 3: Переводим вагон 1 на выходной путь. holding1.Push(train.Pop()) // Шаг 4: Переводим вагон 2 на верхний перегон. train.Push(holding2.Pop()) // Шаг 5: Прицепляем вагон 3 к поезду. train.Push(holding1.Pop()) // Шаг 6: Прицепляем вагон 2 к поезду. train.Push(holding1.Pop()) // Шаг 7: Прицепляем вагон 2 к поезду. train.Push(output.Pop()) // Шаг 8: Прицепляем вагон 1 к поезду.

Описанный процесс показан на рисунке 5.6. Перемещаемый вагон обведен толстой линией, стрелка указывает направление движения.

**ЗАМЕЧАНИЕ**

В настоящем депо, как правило, формируется сразу нескольких составов с еще бол'ьшим числом вагонов и перегонов, поэтому реальная задача гораздо слож- нее, чем та, что описана в примере. А поскольку перестановка вагонов на каж- дом шаге занимает несколько минут, важно найти решение с наименьшим ко- личеством возможных действий.

1

2



5

**1**

3

3

2

2

1

2

3

3



2

3 2 1

3

2



6

2

3

3

1

2



3

3 2

1

3

2



7

1

2

2

3

3



4

2

2

1

3

3



8

1

2

2

3

3

**Рис. 5.6.** Шаги по формированию состава

с использованием двух перегонов и выходного пути

#### Ханойская башня

В головоломке «Ханойская башня» (рис. 5.7) есть три колышка. На один из них нанизаны диски, отсортированные по размеру — от самого большого до самого маленького. Ваша задача — переместить их с одного колышка на другой, сохранив порядок. При этом нельзя брать больше одного диска за раз и класть бол'ьший диск на меньший.

**Рис. 5.7.** Головоломка «Ханойская башня»

Головоломку легко смоделировать с помощью трех стеков — по одному на каждый колышек. Объектами в них будут выступать радиусы дисков.

Следующий псевдокод моделирует перемещение дисков с левого колышка на средний.

peg2.Push(peg1.Pop()) peg3.Push(peg1.Pop()) peg3.Push(peg2.Pop()) peg2.Push(peg1.Pop()) peg1.Push(peg3.Pop()) peg2.Push(peg3.Pop()) peg2.Push(peg1.Pop())

Приведенные шаги показаны на рисунке 5.8.

**Рис. 5.8.** Решение головоломки «Ханойская башня»

**ЗАМЕЧАНИЕ**

Для облегчения восприятия в примере на рисунке 5.8 используется всего три диска. Чтобы переместить *N* дисков, понадобится 2*N* – 1 шагов, причем их число будет быстро увеличиваться с возрастанием *N*. Если бы у вас была стопка из 35 дисков и вы перекладывали по одному диску в секунду, решение головоломки заняло бы более 1000 лет.

Еще один хороший способ решить головоломку «Ханойская башня» — исполь- зовать рекурсию (см. главу 9).

#### Сортировка стеков вставкой

В главе 6 мы еще вернемся к алгоритмам сортировки, а пока вспомните сор- тировку связных списков методом вставки, приведенную в главе 3. Основная ее идея заключалась в том, что из списка ввода выбирался элемент и помещался

в соответствующую позицию в отсортированном списке вывода (который изна- чально был пустым). Этот же способ применим и к стекам.

В таком случае элементы разносятся по обеим сторонам стека: в начало по- падают неотсортированные, в конец — отсортированные. По умолчанию ни один из элементов не отсортирован и все они находятся в соответствующем разделе.

Алгоритм создает второй, временный, стек. Для каждого элемента алгоритм забирает верхний элемент из стека и хранит его в переменной. Затем он пере- мещает во временный стек все неотсортированные элементы, а далее отсортиро- ванные, пока не найдет ту позицию, к которой относится элемент. В этот момент алгоритм вставляет новый элемент в исходный стек и перемещает все элементы из временного стека обратно в исходный. Процесс повторяется до тех пор, пока все элементы не будут добавлены в отсортированный раздел.

В следующем псевдокоде представлен высокоуровневый алгоритм сортировки вставкой.

// Сортируем элементы в стеке. StackInsertionsort(Stack: items)

// Создаем временный стек. Stack: temp\_stack = New Stack

Integer: num\_items = <количество элементов> For i = 0 To num\_items - 1

// Ищем позицию следующего элемента.

// Забираем первый элемент. Data: next\_item = items.Pop()

<Переносим еще не отсортированные элементы

во временный стек temp\_stack. На этом этапе у нас (num\_items - i - 1) неотсортированных элементов.>

<Перемещаем отсортированные элементы во второй стек, пока не найдем место для next\_item.>

<Добавляем next\_item в эту позицию.>

<Возвращаем элементы из временного стека temp\_stack в исходный.>

Next i

End StackInsertionsort

Для каждого элемента алгоритм перемещает во временный стек сначала неот- сортированные элементы, затем некоторые отсортированные, а после возвращает все в исходный стек. На разных этапах количество неотсортированных элементов, которые нужно переместить, равно *N*, *N* – 1, *N* – 2, ..., 2, 1. Таким образом, общее количество перемещенных элементов: *N* + (*N* – 1) + (*N* – 2) + ... + 2 + 1 = *N*   (*N* + 1)/2 = *O*(*N*2). Это означает, что алгоритм обладает временем работы *O*(*N*2).

#### Сортировка стеков выбором

В главе 3 также описывалась сортировка выбором. Она предполагала поиск наименьшего среди неотсортированных элементов и его перемещение в конец от- сортированного списка. Этот же метод подходит для стеков.

Как и в предыдущем случае, элементы исходного стека располагаются в двух разделах: ближе к концу — отсортированные, в самом начале — неотсортированные. По умолчанию ни один из элементов не отсортирован.

Алгоритм создает второй, временный, стек и перемещает в него все неотсорти- рованные элементы для каждого элемента, отслеживая наибольший и вдавливая его. Затем все неотсортированные элементы из временного стека возвращаются в исходный стек. Процесс повторяется до тех пор, пока все элементы не окажутся в отсортированном разделе.

В следующем псевдокоде представлен высокоуровневый алгоритм сортировки выбором.

// Сортируем элементы в стеке. StackSelectionsort(Stack: items)

// Создаем временный стек. Stack: temp\_stack = New Stack

Integer: num\_items = <количество элементов> For i = 0 To num\_items - 1

// Ищем позицию следующего элемента.

// Находим элемент, принадлежащий позиции i.

<Переносим еще не отсортированные элементы во временный

стек temp\_stack, отслеживая наибольший. Сохраняем наибольший элемент в переменную largest\_item. На этом этапе у нас (num\_items - i - 1) неотсортированных элементов.>

<Добавляем largest\_item в оригинальный стек в конец предварительно отсортированных элементов.>

<Возвращаем неотсортированные элементы из временного стека temp\_stack в исходный, минуя найденный largest\_item.>

Next i

End StackSelectionsort

Для каждой записи алгоритм перемещает неотсортированные элементы вначале во временный стек и добавляет наибольший в отсортированный раздел оригинального стека, а затем возвращает их обратно. Получается, неотсортированные элементы пере- носятся дважды, причем на разных этапах их насчитывается *N*, *N* – 1, *N* – 2, ..., 1. Отсюда общее число перемещенных элементов *N* + (*N* – 1) + (*N* – 2) + ... + 1 = *N*  (*N* +1)/2 =

= *O*(*N*2), а время работы алгоритма *O*(*N*2).

## Очереди

*Очередь* — это структура данных, в которой элементы добавляются и удаляют- ся в порядке «первым пришел, первым ушел». Из-за такого поведения стеки ино- гда называют *списками FIFO*, или просто *FIFO* (от англ. *first in first out*).

Очереди подобны магазинным очередям в кассу: вы подходите к концу и про- двигаетесь к началу, где кассир берет у вас деньги и выдает чек.

Метод, добавляющий элементы к очереди, называется Enqueue, а метод, уда- ляющий элементы из очереди, — Dequeue.

В следующих разделах описывается несколько общепринятых методов работы с очередями.

#### Очереди связных списков

Работу с очередью облегчают связные списки. Чтобы удалить из нее последний элемент, список должен быть двойным. Метод Enqueue добавляет ячейку в нача- ло списка, а метод Dequeue удаляет ее из конца.

Приведем псевдокод для постановки элемента в очередь.

Enqueue(Cell: top\_sentinel, Data: new\_value)

// Создаем ячейку для хранения нового значения.

Cell: new\_cell = New Cell new\_cell.Value = new\_value

// Добавляем новую ячейку в связный список. new\_cell.Next = top\_sentinel.Next top\_sentinel.Next = new\_cell

new\_cell.Prev = top\_sentinel End Enqueue

В следующем псевдокоде элемент удаляется из очереди.

Data: Dequeue(Cell: bottom\_sentinel)

// Проверяем, есть ли элемент для удаления.

If (bottom\_sentinel.Prev == top\_sentinel) Then <Формируем исключение.>

// Получаем значение последней ячейки. Data: result = bottom\_sentinel.Prev.Value

// Удаляем последнюю ячейку из связного списка. bottom\_sentinel.Prev = bottom\_sentinel.Prev.Prev bottom\_sentinel.Prev.Next = bottom\_sentinel

// Возвращаем результат. Return result

End Dequeue

**ЗАМЕЧАНИЕ**

Более подробно о связных списках читайте в главе 3.

В связных списках постановка элемента в очередь и удаление из нее происходят за время *O*(1), то есть обе операции являются довольно быстрыми. Кроме того, поскольку для списков не требуется дополнительной памяти (достаточно ссылок между ячейками), они экономят ресурсы.

#### Очереди массивов

Создание очереди в массиве в некотором смысле сложнее, чем в связном списке. Для отслеживания позиций понадобятся две переменные: Next (чтобы отметить

следующую открытую позицию) и Last (чтобы отметить позицию, которая ис- пользовалась дольше всего). Если вы будете сохранять элементы на одной стороне массива и удалять их с другой, занятое пространство просто сместится вниз.

Предположим, очередь создается в массиве, имеющем восемь записей. Рассмот- рим постановку элемента в очередь и его удаление из очереди.

Enqueue(M) Enqueue(O) Enqueue(V)

Dequeue() // Удаляем M. Dequeue() // Удаляем O. Enqueue(I)

Enqueue(N) Enqueue(G)

Dequeue() // Удаляем V. Dequeue() // Удаляем I.

Последовательность приведенных операций отражена на рисунке 5.9. Изначаль- но Next и Last относятся к одной и той же записи. Получается, что очередь пу- ста. После цикла операций Enqueue и Dequeue для добавления новых элементов доступными окажутся лишь две ячейки. Далее это действие станет невозможным.

Последний Следующий



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Последний



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *M* | *O* | *V* |  |  |  |  |  |

Последний



Следующий

Следующий

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | *V* |  |  |  |  |  |

Последний



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | *V* | *I* | *N* | *G* |  |  |

Последний



Следующий

Следующий

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | *N* | *G* |  |  |

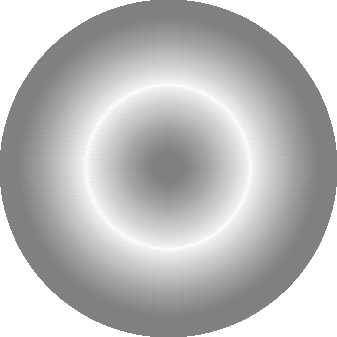
**Рис. 5.9.** Постановка элементов в очередь и удаление из нее

Решить поставленную задачу поможет увеличение массива в тот момент, когда из него выпадет Next. К сожалению, со временем массив сильно разрастется, а про- странство перед записью Last окажется бесполезным. Можно попробовать также перемещать записи массива к началу каждый раз, когда из него выпадает Last. Этот способ будет работать относительно медленно.

Более эффективно будет построить *кольцевой массив*. В нем работа с последним элементом осуществляется так, будто он находится прямо перед первым. В этом случае, когда Next выпадает из массива, он как бы возвращается к первой позиции и у программы снова есть место для хранения новых элементов.

На рисунке 5.10 показана кольцевая очередь, содержащая значения *M*, *O*, *V*, *I*, *N* и *G*.

**Рис. 5.10.** Пример кольцевой очереди



Однако появляется новая проблема: когда очередь пуста, Next совпадает с Last. Если вы добавите достаточное количество элементов в очередь, Next со- вершит полный круг и снова дойдет до Last. Таким образом, невозможно опреде- лить, является очередь пустой или полной.

Справиться с задачей можно несколькими способами: отслеживать общее ко- личество элементов, неиспользуемые промежутки либо элементы, добавляемые в очередь и удаляемые из нее. Программа CircularQueue (ее вы можете скачать на сайте с материалами книги), например, постоянно оставляет один из промежут- ков свободным. В этом случае, если добавить в очередь на рисунке 5.10 другой эле- мент, она будет считаться полной, когда Next стоит перед Last, даже несмотря на одну пустую запись.

Следующий псевдокод показывает алгоритм, используемый демонстрационной версией программы для добавления элемента в очередь.

// Переменные для управления очередью.

Data: Queue[<размер очереди>] Integer: Next = 0

Integer: Last = 0

// Добавляем элемент. Enqueue(Data: value)

// Проверяем, есть ли место для добавления нового элемента. If ((Next + 1) Mod <размер очереди> == Last)

Then <Формируем исключение.>

Queue[Next] = value

Next = (Next + 1) Mod <размер очереди> End Enqueue

Приведенный ниже псевдокод выводит элемент из очереди.

// Удаляем элемент.

Data: Dequeue()

// Проверяем, есть ли элемент для удаления.

if (Next == Last) Then <Формируем исключение.> Data: value = Queue[Last]

Last = (Last + 1) Mod <размер очереди> Return value

End Dequeue

Проблемы в кольцевой очереди возникают и тогда, когда она полностью за- полнена. В таком случае необходимо добавить элементы, назначить больший мас- сив, скопировать данные в новый массив или использовать новый массив вместо старого. Это может занять некоторое время, поэтому попытайтесь сделать массив достаточно вместительным с самого начала.

#### Специализированные очереди

Любая очередь — это уже довольно специализированная структура данных. Но есть еще такие интересные разновидности, как очереди с приоритетом и деки.

#### Очереди с приоритетом

В рассматриваемых очередях каждый элемент имеет свой приоритет и удаляется тот, у которого этот приоритет наивысший. Дело в том, что элементы с высоким приоритетом обрабатываются первыми.

В структурах подобного рода элементы можно хранить отсортированными по приоритету. При добавлении новой записи очередь пересматривается и находится нужная позиция, в которой та будет размещена. Удаление элементов осуществля- ется с самого верхнего. В таком случае постановка элемента в очередь занимает *O*(*N*) времени, а удаление из нее — *O*(1).

Элементы можно хранить и в произвольном порядке — том, в котором они до- бавляются в очередь. Тогда при удалении будет происходить поиск элемента с наи- высшим приоритетом. При таком подходе постановка элемента в очередь займет *O*(1) времени, а удаление из нее — *O(N)*.

Оба метода являются довольно простыми, и их легко использовать при работе со связными списками.

Пирамидальная структура данных, описанная в подразделе «Пирамидальная сортировка» раздела «Алгоритмы *O*(*N*  log *N*)» главы 6, предлагает более эффек- тивный способ создания очереди с приоритетом. Он основан на пирамидальной сортировке, поэтому может добавлять и удалять элементы за время *O*(log (*N*)).

#### Деки

*Дек* — это очередь с двусторонним доступом. Она позволяет добавлять элементы и удалять их с обоих концов.

Деки полезны в алгоритмах, где информация о приоритете элементов известна лишь частично. То есть вы обладаете сведениями, что некоторые элементы имеют

высокий приоритет, а иные — низкий, но какой точно — вам знать не обязательно. В таком случае вы можете добавлять элементы с высоким приоритетом к одному концу дека, а элементы с низким приоритетом — к другому.

Деки легко реализовать с помощью двунаправленных списков.

## Резюме

В этой главе описывались стеки и очереди — две структуры, часто используе- мые другими алгоритмами для хранения данных. В стеке добавление и удаление элементов осуществляется с одного и того же конца в порядке «последним пришел, первым вышел», в очереди — добавление происходит в один конец, а удаление из другого в порядке «первым пришел, первым вышел».

Построить стек можно на основе массива или связного списка. В первом слу- чае вам нужно зарезервировать достаточно места, во втором это условие не имеет никакого значения. Массив подходит и для построения очереди, правда, его размер придется менять, если элемент достигнет конца. Чтобы избежать такой ситуации, удобнее пользоваться двунаправленными списками.

Стеки и очереди позволяют отсортировать элементы за *O*(*N*2) времени, но та- кие алгоритмы пригодны, скорее, для выполнения упражнений, чем для эффектив- ной работы. В следующей главе рассматриваются способы сортировки, обеспечи- вающие значительно большую производительность — *O*(*N*  log *N*) и даже *O(N)*.

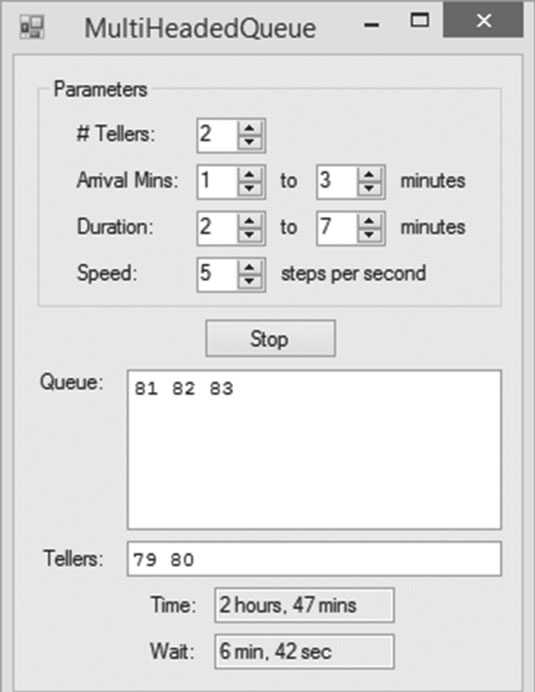
## Упражнения

Звездочкой обозначены задачи повышенной сложности.

1. Каково соотношение между переменными NextIndex1 и NextIndex2 для двух стеков, если один из них полный?
2. Разработайте алгоритм для получения выходного стека, в котором элемен- ты входного стека расположены в обратном порядке.
3. Напишите программу, которая производит сортировку вставкой для стека.
4. Выполняя сортировку вставкой в стеке, алгоритм для каждого элемента перемещает во временный стек сначала неотсортированные элементы, за- тем некоторые из отсортированных, а далее возвращает все в исходный стек. Действительно ли ему нужно возвращать все элементы в исходный стек? Можно ли усовершенствовать работу алгоритма, изменив данный шаг? Как это повлияет на время работы алгоритма?
5. Как применить алгоритм сортировки вставкой для стека к задаче по форми- рованию состава поезда?
6. Напишите программу, осуществляющую сортировку выбором для стеков.
7. Как применить алгоритм сортировки выбором для стека к задаче по форми- рованию состава поезда?
8. Разработайте программу, которая реализует очередь с приоритетом.
9. Создайте программу, реализующую дек.

10\*. Представьте, что в банке клиенты выстраиваются в одну очередь, которая об- служивается несколькими кассирами. Новый клиент занимает место в кон- це очереди и продвигается к ее началу, где необходимую операцию прово- дит освободившийся на данный момент кассир. Смоделируйте очередь «с не- сколькими головами» на основе обычной, которую обслуживают нескольки- ми кассирами.

Создайте программу, подобную той, которая изображена на рисунке 5.11, чтобы сымитировать очередь «с несколькими головами». Разрешите поль- зователю устанавливать количество кассиров, время между клиентами, вре- мя нахождения каждого клиента в очереди, а также скорость симуляции. За- пустите модель с введенными параметрами, чтобы увидеть, как ведет себя очередь. Как количество кассиров повлияет на среднее время ожидания?



**Рис. 5.11.** Окно пограммы MultiHeadedOueue

1. Напишите программу, осуществляющую сортировку вставкой для очереди.
2. Создайте программу, осуществляющую сортировку выбором для очереди.

# Глава 6 СОРТИРОВКА

Алгоритмы сортировки нередко описываются в книгах по алгоритмизации, и на то есть несколько причин. Во-первых, они интересны и на их основе хорошо изу- чать такие технологии, как рекурсия, разделение задач на подзадачи, кучи, а так- же деревья. Во-вторых, эти алгоритмы являются одними из немногих, для кото- рых известно точное время работы. Наиболее быстрые из них, например, выпол- няются за время *O*(*N*  log *N*), если количество сортируемых элементов равно *N*. Такой уровень производительности в отдельных случаях считается оптимальным. И в-третьих, алгоритмы сортировки полезны, поскольку позволяют упорядочить данные определенным образом.

В этой главе описано несколько различных алгоритмов сортировки. Одни из них — сортировка вставкой, выбором и слиянием — более сложные, но зато быстро- действенные. Другие, например сортировка подсчетом, не используют сравнение элементов и могут преодолевать ограничение *O*(*N*  log *N*), работая удивительно эффективно при правильных условиях.

В следующем разделе алгоритмы распределены по категориям с учетом их производительности.

**ЗАМЕЧАНИЕ**

Многие программные библиотеки содержат встроенные быстродействую- щие инструменты сортировки, которые удобно использовать при написании и отладке собственного кода. Но понимать принципы работы таких алгорит- мов все же необходимо, поскольку иногда самостоятельно созданные инстру- менты могут оказаться лучше встроенных. Например, для очень малых спи- сков пузырьковая сортировка будет эффективнее любой сложной библиотеки, как и сортировка счетом, если данные имеют определенные характеристики.

## Алгоритмы O(N2)

Алгоритмы *O*(*N*2) являются относительно медленными, зато простыми. По- следнее качество позволяет им иногда превосходить сравнительно быстрые, но более сложные алгоритмы для очень малых массивов.

#### Сортировка вставкой в массивах

Алгоритм сортировки вставкой мы уже рассматривали в главе 3, когда речь шла о связных списках, и в главе 4, когда изучали стеки и очереди. Основная идея такого метода заключается в выборе элемента из списка ввода и его вставке в соот- ветствующую позицию отсортированного списка вывода, который изначально пуст.

Аналогичную технологию можно использовать и для сортировки массива.

Insertionsort(Data: values[])

For i = 0 To <количество значений> - 1

// Перемещаем элемент i в соответствующую позицию

// в отсортированной части массива.

<Находим первый индекс j, при котором j < i и values[j] > values[i].>

<Помещаем элемент в позицию j.>

Next i

End Insertionsort

При циклическом прохождении по элементам массива на основе индекса i на- блюдается следующее разделение: элементы с индексом менее i признаются отсор- тированными, а с индексом более или равным i — неотсортированными. После того как код пересмотрит все индексы массива (от 0 до последнего), он перемещает элемент с индексом i в соответствующую позицию во второй части массива. Чтобы обнару- жить эту позицию, перебираются уже отсортированные элементы и находится пер- вый элемент, который больше, чем values[i]. Перемещение может занять какое-то время. Ведь если новый индекс элемента должен быть j, код должен сдвинуть все элементы между индексами j и i на одну позицию вправо, чтобы освободить место.

На рисунке 6.1 показаны основные шаги алгоритма. Верхний фрагмент пред- ставляет собой исходный неотсортированный массив. На среднем фрагменте первые четыре элемента (они обведены толстой линией) уже отсортированы, а алгоритм готовится добавить к ним следующий со значением 3. Код пересматривает от- сортированные элементы, пока не определит, что число 3 нужно вставить перед числом 4. На нижнем фрагменте видно, что алгоритм передвинул значения 5, 6 и 7 вправо, чтобы освободить место для значения 3. Вставив его, он продолжит цикл For, чтобы упорядочить следующий элемент со значением 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 5 | 6 | 1 | 3 | 2 | 8 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 6 | 7 | 3 | 2 | 8 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  | 5 | 6 | 7 | 2 | 8 | 0 |

3

**Рис. 6.1.** Сортировка вставкой

Приведенный алгоритм сортирует элементы в исходном массиве, поэтому ему не требуется дополнительной памяти (исключение составляют нескольких пере- менных для контроля циклов и перемещения элементов).

Если в массиве содержится *N* элементов, алгоритм рассматривает все. Для каждой позиции i он должен найти предыдущие отсортированные элементы

и определить новое место для сортируемого значения. Затем ему нужно передви- нуть элементы между этим положением и положением индекса i на одну позицию вправо. Так, если элемент i должен сместиться на позицию j, потребуется вначале j шагов, а потом еще i – j для перестановки элементов. В результате общее коли- чество шагов будет равно i.

Если суммировать шаги, требуемые для позиционирования всех элементов, общее время работы составит 1 + 2 + 3 + ... + *N* = (*N*2 + *N*)/2, отсюда произво- дительность алгоритма — *O*(*N*2). Это не очень хороший показатель, однако для относительно малых массивов (менее 10 000 элементов) он является достаточно быстрым. К тому же алгоритм довольно простой, поэтому для совсем малых мас- сивов может работать эффективнее, чем более сложные алгоритмы. Насколько малым должен быть такой массив, зависит от вашей системы. Обычно речь идет о 5–10 элементах.

#### Сортировка выбором в массивах

Подобная сортировка рассматривалась в главе 3 для связных списков и в главе 4 для стеков и очередей. Ее суть состоит в том, чтобы найти в списке ввода наиболь- ший элемент и добавить его в конец отсортированного списка. Как вариант, можно искать наименьший элемент и перемещать его в начало отсортированного списка.

В следующем псевдокоде представлен алгоритм, работающий с массивами.

Selectionsort(Data: values[])

For i = 0 To <количество значений> - 1

// Находим элемент, принадлежащий позиции i.

<Находим наименьший элемент, у которого индекс j >= i.>

<Меняем местами values[i] и values[j].>

Next i

End Selectionsort

Код проходит в рамках цикла по массиву и выбирает наименьший элемент, ко- торый еще не добавлен в отсортированную часть списка, затем меняет его местами с элементом в позиции i.

Основные шаги алгоритма представлены на рисунке 6.2. На верхнем фрагмен- те виден исходный неотсортированный массив. На среднем первые три элемента уже отсортированы (они обведены толстой линией), а алгоритм готовится по- менять позицию следующего элемента. Он ищет неотсортированные элементы, чтобы выявить среди них тот, который имеет наименьшее значение, — в данном случае это 3. Найденное число перемещается в следующую неотсортированную позицию. На нижнем фрагменте показан массив после того, как новый элемент был добавлен в отсортированную часть. Цикл For приступает к следующему элементу со значением 5.

Как и в случае сортировки вставкой, алгоритм располагает упорядоченные эле- менты в исходном массиве, поэтому дополнительной памяти ему не требуется (кро- ме нескольких переменных для контроля над циклами и перемещения элементов). Если в массиве содержится *N* элементов, алгоритм изучает каждый из них. Для на- чала он должен пересмотреть *N* – *i* еще не отсортированных элементов, чтобы найти принадлежащий позиции i, а затем передвинуть сортируемый элемент на послед-

нюю позицию за малое количестве шагов. Таким образом, для перемещения всех элементов потребуется (*N* – 1) + (*N* – 2) + ... + 2 + 1 = (*N*2 + *N*)/2 шагов. Это значит, что алгоритм обладает временем работы *O*(*N*2), как и алгоритм сортировки вставкой.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 5 | 6 | 1 | 3 | 2 | 8 | 0 |

Наименьшее

*i* значение

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 8 | 7 | 3 | 6 | 5 |

Обмен

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 7 | 8 | 6 | 5 |

**Рис. 6.2.** Сортировка выбором

Сортировка выбором также является достаточно быстрой для относительно малых массивов (менее 10 000 элементов). Кроме того, она проста и если элементов совсем немного (от 5 до 10), работает эффективнее, чем более сложные алгоритмы.

#### Пузырьковая сортировка

Пузырьковая сортировка предполагает следующее: если массив не отсортиро- ван, любые два смежных элемента в нем находятся в неправильном положении. Из-за этого алгоритм должен проходить по массиву несколько раз, меняя местами все неправильные пары.

Bubblesort(Data: values[])

// Повторяем, пока не отсортируем массив. Boolean: not\_sorted = True

While (not\_sorted)

// Предполагаем, что неправильных пар нет. not\_sorted = False

// Ищем смежные элементы массива, стоящие в неправильном порядке. For i = 0 To <количество значений> - 1

// Проверяем, стоят ли элементы i и i – 1

// в правильном порядке.

If (values[i] < values[i - 1]) Then

// Меняем их местами. Data: temp = values[i] values[i] = values[i - 1] values[i - 1] = temp

Next i End While

End Bubblesort

End If

// Массив еще не отсортирован. not\_sorted = True

В коде используется булевская переменная not\_sorted, которая отслеживает перемещение элементов при прохождении через массив. Пока она истинна, цикл работает — ищет неправильные пары элементов и перестраивает их.

Иллюстрацией этого алгоритма может служить рисунок 6.3. Первый массив выглядит по большей части отсортированным, но, пройдясь по нему, алгоритм выявит, что пара 6 — 3 находится в неправильном положении (число 6 должно следовать после 3). Код поменяет найденные элементы местами и получит вто- рой массив, в котором в неправильном положении окажется пара 5 — 3. После ее исправления образуется третий массив, где неверна пара 4 — 3. Поменяв ее эле- менты местами, алгоритм сформирует четвертый массив, совершит еще один ко- нечный проход и, не найдя пар, стоящих в неправильном положении, остановится.

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 4 |
| 5 |
| 6 |
| 3 |
| 7 |
| 8 |
| 9 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 4 |
| 5 |
| 3 |
| 6 |
| 7 |
| 8 |
| 9 |

|  |
| --- |
| 0 |
| 1 |
| **2** |
| 4 |
| 3 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |

**Рис. 6.3.** Пузырьковая сортировка

В пузырьковой сортировке неупорядоченный элемент 3 как бы медленно

«всплывает» на правильную позицию, отсюда и специфическое название метода. Каждое прохождение через массив ставит на нужное место как минимум один эле- мент. В массиве, приведенном на рисунке 6.3, при первом прохождении на правиль- ной позиции оказывается число 6, при втором — 5, при третьем — 3 и 4.

Если предположить, что в массиве содержится *N* элементов и хотя бы один из них занимает свое место в результате однократного пересмотра значений, то алго- ритм может совершить не более *N* прохождений. (Все *N* понадобятся, когда массив изначально отсортирован в обратном порядке.) Каждое такое прохождение вклю- чает *N* шагов, отсюда общее время работы алгоритма — *O*(*N*2).

Как и две предыдущие сортировки, пузырьковая является довольно медлен- ной, но может показать приемлемую производительность в малых списках (менее

1000 элементов). Она также быстрее, чем более сложные алгоритмы для очень ма- лых списков (около 5 элементов).

Вы можете несколько усовершенствовать пузырьковую сортировку. В масси- ве, приведенном на рисунке 6.3, элемент со значением 3 находится ниже своей ко- нечной позиции. А что если он будет выше нее? Алгоритм определит, что элемент не там, где ему положено находиться, и поменяет местами со следующим элемен- том, затем снова переставит, обнаружив неверный распорядок. Так продолжится вниз по списку до тех пор, пока не отыщется конечная позиция. Чередуя прохож- дения вверх и вниз по массиву, вы можете ускорить работу алгоритма: первые бы- стрее поставят на место те элементы, которые находятся слишком низко в списке, вторые — те, что слишком высоко.

Еще один способ усовершенствования — выполнять несколько перестановок за проход. Например, при движении вниз по массиву элемент (назовем его *K*) может сменить свою позицию не один раз, прежде чем займет нужное место. Вы сэконо- мите время, если не будете передвигать его по массиву, а сохраните во временной переменной и, сместив другие элементы вверх, найдете целевую позицию для *K*, вставите его туда и продолжите прохождение.

Предположим, что в нашем массиве содержится наибольший элемент *L*, который стоит не на своем месте. Двигаясь вниз, алгоритм добирается до него (возможно, совершая другие перестановки) и затем перемещает вниз по списку, пока тот не достигнет конечной позиции. Во время последнего прохождения по массиву ни один элемент не может встать после *L*, поскольку этот элемент уже находится на нужном месте. Значит, алгоритм может остановиться, когда достигнет элемента *L*.

Если обобщить вышесказанное, получается, что алгоритм завершает про- хождение через массив, добравшись до позиции последней перестановки, кото- рую выполнил во время предыдущего прохождения. Таким образом, отследив последние перестановки при движении вниз и вверх по массиву, вы можете сократить путь.

Рассмотренные усовершенствования проиллюстрированы на рисунке 6.4. На крайнем левом фрагменте видно, что во время первого прохождения вниз по массиву алгоритм вставляет элемент 7 во временную переменную и меняет местами с элементами 4, 5, 6 и 3. Другими словами, алгоритму не требуется хранить элемент 7 в массиве до тех пор, пока он не займет конечную позицию.

Разместив 7 должным образом, алгоритм продолжает движение по массиву и не находит других элементов для перестановки. Теперь ему известно, что 7 и следу- ющие за ним элементы стоят в своих конечных позициях и рассматривать их больше не нужно. Если бы какой-нибудь элемент, расположенный ближе к верху массива, оказался больше 7, то при первом же прохождении он переместился бы вниз, минуя 7. На среднем фрагменте элементы на конечных позициях закрашены серым — во время последующих прохождений они не нуждаются в проверке.

На крайнем правом фрагменте алгоритм совершает второе прохождение по мас- сиву вверх, начиная с элемента, стоящего перед 7. В нашем случае это 3. Алгоритм меняет его местами с элементами 6, 5 и 4, удерживая во временной переменной, пока не поставит на конечную позицию. Теперь 3 и предшествующие ему элементы находятся на своих местах — они закрашены серым цветом.

Последнее прохождение осуществляется вниз по массиву, начиная со значения 4 и заканчивая значением 6. Во время него перестановки не выполняются, и алгоритм останавливается.

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 7 |
| 4 |
| 5 |
| 6 |
| 3 |
| 8 |
| 9 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 4 |
| 5 |
| 6 |
| 3 |
| 7 |
| 8 |
| 9 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |

**Рис. 6.4.** Усовершенствованная пузырьковая сортировка

Такие усовершенствования на практике ускоряют пузырьковую сортировку (в ходе теста 10 000 элементов были отсортированы обычным образом за 2,5 с, а усовершенствованным за 0,69 с). Но все же время работы подобного алгоритма остается *O*(*N*2), то есть он применим для списков с ограниченным размером.

## Алгоритмы O(N  log N)

Алгоритмы *O*(*N*  log *N*) намного быстрее алгоритмов *O*(*N*2), по крайней мере, при работе с большими массивами. Например, если *N* = 1000, то время работы *O*(*N*  log *N*) покажет результат 1  104, а *N*2 — 1  06, то есть примерно в 100 раз больше. Такая раз- ница в скорости делает алгоритмы первого рода более полезными на практике.

#### Пирамидальная сортировка

Пирамидальная сортировка подходит для работы с полными бинарными дере- вьями в массиве. В ней используется структура данных, называемая *кучей*.

#### Хранение полных бинарных деревьев в массивах

*Бинарное дерево* — это дерево, в котором каждая вершина соединена с хотя бы двумя дочерними записями. Если все его уровни заполнены, кроме, возможно, по- следнего (где все вершины сдвинуты влево), то такое дерево называется *полным*.

На рисунке 6.5 показано полное бинарное дерево с 12 вершинами: первые три уровня полные, четвертый содержит пять вершин, сдвинутых влево.

7

1

10

4

6

9

2

11

3

5

12

8

**Рис. 6.5.** Полное бинарное дерево

Полные бинарные деревья легко сохранить в массив: присвойте корневому узлу индекс 0, а дочерним записям любого узла с индексом i — индексы 2i + 1 и 2i + 2 и т. д.

Если узлу присвоен индекс j, то родительская запись для него будет иметь ин- декс (j – 1)/2 , где означают округление результата до следующего меньшего

целого числа. Например, l2,9J = 2 и l2J = 2.

l J l J

На рисунке 6.6 показано дерево (см. рис. 6.5), переведенное в массив. Сверху указаны входные индексы. Так, узлу 6 присвоен индекс 4. Значит, его дочерние записи 5 и 12 будут иметь индексы 4  2 + 1 = 9 и 4  2 + 2 = 10 соответственно.

0 1 2 3 4 5 6

7 8 9

10 11

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 1 | 10 | 4 | 6 | 9 | 2 | 11 | 3 | 5 | 12 | 8 |

**Рис. 6.6.** Полное бинарное дерево, переведенное в массив



Если индекс какой-либо дочерней записи превышает наибольший индекс масси- ва — такой записи в дереве нет. Например, узел 9 имеет индекс 5, тогда его дочерняя запись справа должна носить индекс 2  5 + 2 = 12. Но это число выходит за пределы массива, и если посмотреть на рисунок 6.5, то можно убедиться, что у элемента со значением 9 справа дочерней записи нет. Рассмотрим также получение индекса родительской записи для элемента со значением 12, которому присвоен индекс 10.

J l J

(10 – 1)/2 = 4,5 = 4. В массиве ему соответствует элемент 6. На нашем дереве (см. рис. 6.5) это и есть родительский узел для вершины со значением 12.

#### Куча

*Двоичная куча* (рис. 6.7) — это полное бинарное дерево, где каждая вершина содержит значение, которое не меньше тех, что находятся в его дочерних записях.

На рисунке 6.5 изображена не куча. Если взять для примера корневой узел 7, можно заметить, что его правая дочерняя запись имеет значение 10, то есть она больше родительской.

15

11

10

7

5

9

2

6

4

3

1

8

**Рис. 6.7.** Двоичная куча

Вы можете выстраивать кучу, добавляя по одной вершине за раз. Начните с дерева, состоящего из одной вершины. Поскольку у него нет дочерних за- писей, оно уже является кучей. Новый узел нужно добавлять в конец дерева. Чтобы оно оставалось полностью бинарным, это должен быть правый край нижнего уровня.

Теперь сравните новое значение со значением родительской записи. Если первое больше, поменяйте их местами. Поскольку дерево изначально формировалось как куча, текущее родительское значение должно превышать уже имеющееся дочернее (если оно есть). Поменяв местами родительский и новый дочерний элементы, вы сохраните нужное отношение в данном узле, но можете нарушить его в узле вы- шестоящем. Поднимитесь на уровень вверх, оцените располагающуюся там роди- тельскую запись и, если нужно, произведите замену. Продолжайте это действие до тех пор, пока не дойдете до корневой записи. Так вы получите кучу.

На рисунке 6.8 показан процесс, в ходе которого к дереву, изображенному на рисунке 6.7, добавляют новую запись 12. Обновленную кучу вы можете увидеть на рисунке 6.9.



15

11

10

7

5

9

2

6

4

3

1

8

12

**Рис. 6.8.** Добавление нового значения к куче

15

11

12

7

5

10

2

6

4

3

1 8

9

**Рис. 6.9.** Обновленная куча

Если куча хранится в массиве, ее обновление выполнить еще проще. Очередной элемент, присоединяемый к концу дерева, автоматически занимает соответству- ющую позицию в массиве.

Следующий псевдокод превращает массив в кучу.

MakeHeap(Data: values[])

// Добавляем каждый элемент в кучу (по одному). For i = 0 To <количество значений> - 1

// Начинаем с нового элемента и работаем до корня. Integer: index = i

While (index != 0)

// Находим индекс родительской записи. Integer: parent = (index - 1) / 2

// Если дочерняя запись меньше или равна родительской,

// мы закончили и выходим из цикла While.

If (values[index] <= values[parent]) Then Break

// Меняем местами родительскую и дочернюю записи. Data: temp = values[index]

values[index] = values[parent] values[parent] = temp

// Переходим к родительской записи. index = parent

End While

Next i

End MakeHeap

Кучи хороши для создания очередей с приоритетом, поскольку чем больше элемент в дереве, тем ближе он к корневому узлу. Следовательно, при удалении элемента из очереди достаточно убрать тот, что является корневым. К сожалению, такая операция разрушает пирамидальную структуру кучи. Но ситуацию легко ис- править, если переместить последний элемент дерева в корень. На следующем этапе понадобится восстановить основное свойство кучи. Для этого можно использовать тот метод, который фигурирует и при построении: если новое значение меньше какого-то дочернего, поменяйте их местами. Исправив текущий узел, проверьте

тот, что идет на уровень ниже. Продолжайте перестановку узлов вниз по дереву, пока не найдете точку, в которой свойство кучи уже будет выполненным, либо не достигните конца дерева.

В следующем псевдокоде представлен алгоритм удаления элемента из кучи и восстановления ее основного свойства.

Data: RemoveTopItem (Data: values[], Integer: count)

// Сохраняем верхний элемент, чтобы вернуться к нему позднее. Data: result = values[0]

// Перемещаем последний элемент к корню. values[0] = values[count - 1]

// Восстанавливаем свойство кучи. Integer: index = 0

While (True)

// Находим индексы дочерних записей. Integer: child1 = 2 \* index + 1 Integer: child2 = 2 \* index + 2

// Если индекс дочерней записи выпадает из дерева,

// используем индекс родительской записи. If (child1 >= count) Then child1 = index If (child2 >= count) Then child2 = index

// Если свойство кучи выполнено,

// мы закончили и выходим из цикла While.

If ((values[index] >= values[child1]) And (values[index] >= values[child2])) Then Break

// Получаем индекс дочерней записи с большим значением. Integer: swap\_child

If (values[child1] > values[child2]) Then swap\_child = child1

Else

swap\_child = child2

// Меняем местами с большим дочерним элементом. Data: temp = values[index]

values[index] = values[swap\_child] values[swap\_child] = temp

// Переходим на дочернюю ветку. index = swap\_child

End While

// Возвращаем значение, которое удалили из корня. return result

End RemoveTopItem

Приведенный алгоритм в качестве параметра берет размер дерева, поэтому он может найти место, где заканчивается куча внутри массива. Значение со- храняется в корневом узле, чтобы потом была возможность к нему вернуться. Затем последний элемент дерева перемещается к корневому узлу. Для этого алгоритм назначает индексу корневого узла переменную index и вводит бес- конечный цикл While.

Внутри цикла алгоритм рассчитывает индексы дочерних записей текущего узла. Если они оба выпадают из дерева, то их значения приравниваются к родитель- скому. В таком случае дальнейшее сравнение индекса происходит с самим собой. А поскольку любое значение больше или равно самому себе, подобный подход удовлетворяет свойству кучи и недостающий узел не заставляет алгоритм менять значения местами.

После того как алгоритм рассчитает дочерние индексы, он проверяет, сохраня- ется ли свойство кучи в текущем месте. Если да, то алгоритм покидает цикл While. То же самое происходит, если нет обеих дочерних записей или в отсутствии одной из них другая соответствует свойству кучи.

Если свойство кучи не выполняется, алгоритм устанавливает swap\_child на индекс дочерней записи, содержащий большее значение, и меняет местами значения родительского и дочернего узлов. Затем он обновляет переменную index, чтобы сдвинуть ее вниз к переставленному дочернему узлу, и продолжает спускаться по дереву.

#### Осуществление пирамидальной сортировки

Теперь, когда вы знакомы с построением кучи и знаете, как с ней работать, осу- ществить пирамидальную сортировку для вас не составит труда. Приведенный ниже алгоритм строит кучу. Он несколько раз меняет ее первый и последний элементы местами и перестраивает все дерево, исключая последний элемент. При каждом прохождении один элемент удаляется из кучи и добавляется в конец массива, где расположены элементы в отсортированном порядке.

Heapsort(Data: values)

<Преобразуем массив в кучу.>

For i = <количество значений> - 1 To 0 Step -1

// Меняем местами корневой и последний элементы. Data: temp = values[0]

values[0] = values[i] values[i] = temp

<Определяем элемент в позиции i, который нужно удалить из кучи. Теперь куча содержит i - 1 элементов. Опускаем значение нового узла вниз, чтобы восстановить свойство кучи.>

Next i

End Heapsort

Представленный алгоритм начинает свою работу с того, что превращает мас- сив в кучу. Затем он несколько раз удаляет верхний элемент, самый большой и перемещает его в конец кучи. Количество элементов в куче при этом сокращает- ся и восстанавливается ее основное свойство, а переставленный элемент занимает правильную позицию в конце кучи.

На завершающем этапе алгоритм удаляет элементы из кучи в порядке от наи- большего к наименьшему и помещает их в конец постоянно сокращающейся кучи. Массив продолжает содержать значения в порядке возрастания.

Объем памяти, требуемый для пирамидальной сортировки, легко рассчи- тать. Алгоритм хранит все данные внутри исходного массива и использует лишь

определенное количество дополнительных переменных для расчета значений и их перестановки. Если в массиве содержится *N* значений, алгоритм использует *O*(*N*) памяти.

Сложнее определить время работы алгоритма. Чтобы построить исходную кучу, ему приходится прибавлять каждый элемент к растущей куче. Всякий раз он разме- щает его в конце дерева и просматривает структуру данных до самого верха, чтобы убедиться, что она является кучей. Поскольку дерево полностью бинарное, количе- ство его уровней равно *O*(log *N*). Таким образом, передвижение элемента вверх по дереву может занять максимум *O*(log *N*) шагов. Алгоритм осуществляет шаг, при котором добавляется элемент и восстанавливается свойство кучи, *N* раз, поэтому общее время построения исходной кучи составит *O*(*N*  log *N*).

Для завершения сортировки алгоритм удаляет каждый элемент из кучи, а за- тем восстанавливает ее свойства. Для этого он меняет местами последний элемент кучи и корневой узел, а затем передвигает новый корень вниз по дереву. Дерево имеет высоту *O*(log *N*) уровней, поэтому процесс может занять *O*(log *N*) време- ни. Алгоритм повторяет такой шаг *N* раз, отсюда общее количество нужных ша- гов — *O*(*N*  log *N*).

Суммирование времени, необходимого для построения исходной кучи и окон- чания сортировки, дает следующее: *O*(*N*  log *N*) + *O*(*N*  log *N*) = *O*(*N*  log *N*).

*Пирамидальная сортировка* представляет собой так называемую сортировку на месте и не нуждается в дополнительной памяти. На ее основе хорошо видно, как работают кучи и как происходит сохранение полного бинарного дерева в мас- сиве. И хотя производительность *O*(*N* log *N*) высока для алгоритма, сортирующе- го путем сравнений, далее пойдет речь об алгоритме, который показывает еще бо- лее быстрые результаты.

#### Быстрая сортировка

Приведенный ниже высокоуровневый алгоритм делит массив на две части и впоследствии рекурсивно вызывает сам себя для их сортировки.

Quicksort(Data: values[], Integer: start, Integer: end)

<Выбираем элемент из массива. Называем его разделяющим элементом.>

<Переносим элементы, которые меньше разделяющего, в начало массива.>

<Переносим элементы, которые больше разделяющего или равны ему, в конец массива. >

<Пусть серединой будет индекс, где помещен разделяющий элемент.>

// Рекурсивно сортируем две части массива. Quicksort(values, start, middle - 1) Quicksort(values, middle + 1, end)

End Quicksort

В верхней части рисунка 6.10 показан массив для сортировки. В качестве раз- деляющего элемента выбрано первое значение 6.

На среднем фрагменте значения, которые меньше, чем разделяющий эле- мент, были перемещены в начало массива, а значения, которые больше или рав- ны ему, — в конец. Разделяющий элемент с индексом 6 закрашен темно-серым.

Обратите внимание, что еще один элемент массива со значением 6 следует за разделителем.



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 7 | 5 | 9 | 5 | 3 | 8 | 4 | 0 | 3 | 6 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 5 | 3 | 4 | 0 | 3 | 6 | 7 | 9 | 8 | 6 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 3 | 4 | 5 | 5 | 6 | 5 | 7 | 8 | 9 |

**Рис. 6.10.** Быстрая сортировка



Далее алгоритм рекурсивно вызывает сам себя, чтобы отсортировать части массива до и после разделяющего элемента. Результат этого процесса показан на нижнем фрагменте рисунка 6.10.

Перед тем как приступить к подробному анализу алгоритма, изучим время его работы.

#### Анализ времени работы алгоритма

Для начала рассмотрим особый случай, когда при каждом шаге разделяющий элемент делит анализируемый массив пополам (рис. 6.11).



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

**Рис. 6.11.** Разделение массива поровну



В каждой вершине дерева, изображенного на рисунке 6.11, происходит обра- щение к алгоритму быстрой сортировки. Толстыми линиями отмечено разделение массива надвое. Стрелки символизируют алгоритм быстрой сортировки, он вызы-

вает себя дважды — для каждой половины массива. Внизу дерева вызов осущест- вляется единожды (и возвращается, ничего не сделав), поскольку список из одного элемента уже отсортирован. Достигнув последнего уровня в пирамидальной струк- туре данных, вызовы начинают возвращаться к собственным методам — так про- исходит контроль снизу вверх.

Если изначально в массиве содержится *N* элементов и они разделены четко по- полам, дерево вызовов быстрой сортировки имеет высоту log *N* уровней. При каж- дом вызове проверяются элементы в той части массива, которая подвергается сортировке. Например, в массиве из четырех элементов будут исследованы все четыре элемента с целью последующего разделения их значений.

Все элементы из исходного массива представлены на каждом уровне дерева. Та- ким образом, любой уровень содержит *N* элементов. Если вы добавите те, которые должен проверять каждый вызов быстрой сортировки на определенном уровне де- рева, получится *N* элементов. Это значит, что вызовы быстрой сортировки потре- буют *N* шагов. Поскольку дерево имеет высоту log *N* уровней и для каждого из них нужно *N* шагов, общее время работы алгоритма составит *O*(*N*  log *N*).

Подобный алгоритм предполагает разделение массива на две равные части при каждом шаге, что выглядит не очень правдоподобно. Однако в большинстве случаев разделяющий элемент будет отстоять недалеко от середины — не точно по центру, но и не с самого края. Например, на рисунке 6.10 на среднем фрагменте разделяю- щий элемент 6 находится хотя и не ровно посередине, но близко к ней. В этом слу- чае алгоритм быстрой сортировки все еще обладает временем работы *O*(*N*  log *N*).

В худшем случае разделяющий элемент может оказаться меньше любого другого в той части массива, которую делит, либо все они будут иметь равные значения. Тогда ни один из элементов не перейдет в левую часть массива — все, кроме раз- деляющего, окажутся в правой. Первый рекурсивный вызов вернется немедленно, поскольку сортировка не понадобится, зато в ходе второго потребуется обработать почти каждый элемент. Если первому вызову быстрой сортировки надо отсорти- ровать *N* элементов, то рекурсивному — *N* – 1.

Если разделяющий элемент всегда меньше остальных в сортируемой части массива, алгоритм вызывается для сортировки вначале *N* элементов, затем *N* – 1, *N* – 2 и т. д. В таком случае дерево вызовов, изображенное на рисунке 6.11, является очень тонким и имеет высоту *N*.

Вызовы быстрой сортировки на уровне i в дереве должны проверить *N* – i элементов. Суммирование всех элементов, проверяемых на всех вызовах, даст *N* + (*N* – 1) + (*N* – 2) + ... + 1 = *N*  (*N* + 1)/2, что равняется *O*(*N*2). Таким образом, в худшем случае время работы алгоритма составит *O*(*N*2). Кроме вышесказанного, следует обратить внимание на требуемый объем памяти. Он частично зависит от метода, с помощью которого массив делится на части, а также от глубины рекурсии алгоритма. Если последовательность рекурсивных вызовов слишком глубока, про- грамма расходует стековое пространство и зависнет.

В примере с деревом, изображенным на рисунке 6.11, алгоритм быстрой сор- тировки рекурсивно вызывает сам себя до глубины *N*. Таким образом, в ожидае- мом случае стек вызова программы будет иметь глубину *O*(log *N*) уровней. Для большинства компьютеров это не проблема. Даже если в массиве содержится 1 млрд элементов, в log *N* их всего 30, а стек вызова должен обладать возможностью

работать с 30 рекурсивными вызовами. Однако в худшей ситуации, когда дерево высокое и тонкое, глубина рекурсии составит *N*. Немногие программы способны создать стек вызова с 1 млрд рекурсивных вызовов.

Вы можете избежать наихудшего сценария, если заставите алгоритм работать в течение разумного времени и с разумной глубиной рекурсии путем тщательного выбора разделяющего элемента. В следующих подразделах рассматриваются необ- ходимые для этого стратегии, приводятся два метода разделения массива на части, а также резюмируются сведения по использованию быстрой сортировки на практике.

#### Выбор разделяющего элемента

В качестве разделяющего элемента можно использовать первый элемент из той части массива, которая сортируется. Это простой и, как правило, эффективный спо- соб. К сожалению, если массив изначально отсортирован в прямом или обратном порядке, ситуация окажется наихудшей. Лучше всего, если элементы располага- ются в случайном порядке, но чаще сортировка является частичной или полной.

Решить проблему поможет предварительная рандомизация массива перед вы- зовом быстрой сортировки. Если элементы расположены в случайном порядке, маловероятно, что описанный метод каждый раз будет выбирать плохой разделя- ющий элемент, и мы получим худший вариант его работы. В главе 2 рассказыва- лось, как рандомизировать массив за время *O*(*N*) и не увеличить при этом ожида- емую производительность алгоритма быстрой сортировки *O*(*N*  log *N*). Однако на практике для работы с большим массивом требуется довольно много времени, по- этому опытные программисты не используют такой метод.

Можно прибегнуть к другому варианту: определить первый, последний и средний элементы в сортируемой части массива и использовать в качестве разделителя эле- мент, чье значение находится между любыми двумя значениями из этого трио. Нет абсолютной гарантии, что результат не будет приближаться к наибольшему или наи- меньшему элементу в данной части массива, однако незначительная вероятность есть.

И наконец, последний метод заключается в том, чтобы выбрать случайный индекс из сортируемой части массива, а затем использовать в качестве разделителя значение элемента с данным индексом. Маловероятно, что каждый такой выбор приведет к плохому результату и худшему варианту работы алгоритма.

#### Осуществление быстрой сортировки со стеками

Когда разделитель выбран, нужно распределить элементы по обеим сторонам массива. Это довольно просто сделать, если разбросать элементы по двум стекам, в зависимости от того, больше или меньше они разделителя. Следующий псевдо- код представляет реализацию рассматриваемого шага.

Stack of Data: before = New Stack of Data Stack of Data: after = New Stack of Data

// Собираем элементы до и после разделяющего элемента.

// Предполагается, что разделяющий элемент хранится в переменной start. For i = start + 1 To end

If (values[i] < divider) Then before.Push(values[i]) Else after.Push(values[i])

Next i

<Перемещаем обратно в массив элементы стека «до».>

<Добавляем разделяющий элемент в массив.>

<Перемещаем обратно в массив элементы стека «после».>

На этом месте алгоритм готов рекурсивно вызывать сам себя, чтобы отсорти- ровать две части массива по обе стороны от разделяющего элемента.

#### Осуществление быстрой сортировки на месте

Применять стеки для разбиения массива на две части, безусловно, удобно, но для этого требуется выделить дополнительную память. Вы можете сэкономить время и назначить стеки в начале алгоритма, а затем при каждом вызове работать с ними, а не создавать собственные. Однако все равно потребуется объем памяти *O*(*N*) для их первого использования.

Поработав немного, вы сможете разбить элементы на две группы без задей- ствования дополнительной памяти. В следующем высокоуровневом псевдокоде иллюстрируется реализация этого метода.

<Перемещаем разделяющий элемент в начало массива.>

<Удаляем разделяющий элемент из массива. В начале массива образуется

«дырка», куда можно поместить другой элемент.>

Повторяем:

<Пересматриваем массив с конца, чтобы найти последний элемент, который меньше разделяющего.>

<Перемещаем найденный элемент в «дырку». Теперь возникает «дырка» на его месте.>

<Пересматриваем массив с самого начала, чтобы найти первый элемент, который больше разделяющего или равен ему.>

<Перемещаем найденный элемент в «дырку». Теперь возникает «дырка» на его месте.>

Поиск по массиву в обратном и прямом направлениях, а также перемещение элементов в «дырку» нужно вести до тех пор, пока две области, которые вы ищете, не сойдутся где-то посередине. Затем следует разместить разделяющий элемент в «дырке», образовавшейся между двумя частями, и отсортировать эти части, вы- звав алгоритм рекурсивно. Это довольно трудный шаг, но, изучив внимательно на- стоящий код, вы сможете понять его работу.

В следующем псевдокоде представлен низкоуровневый алгоритм быстрой сор- тировки.

// Сортируем данную часть массива.

Quicksort(Data: values[], Integer: start, Integer: end)

// Если в массиве не более одного элемента, значит, он отсортирован. If (start >= end) Then Return

// Используем первый элемент как разделяющий. Integer: divider = values[start]

// Перемещаем те элементы, которые меньше разделяющего,

// в начало массива, а те, которые больше или равны ему, – в конец.

Integer: lo = start Integer: hi = end While (True)

// Просматриваем массив с конца от hi, чтобы найти

// последний элемент, значение которого меньше разделяющего.

// Перемещаем найденный элемент в «дырку».

// Теперь «дырка» возникает на его месте. While (values[hi] >= divider)

hi = hi - 1

If (hi <= lo) Then <Выходим из внешнего цикла While.> End While

If (hi <= lo) Then

// Левая и правая части массива сходятся посередине,

// значит, мы закончили. Помещаем сюда разделяющий

// элемент и выходим из цикла While. values[lo] = divider

<Выходим из внешнего цикла While.>

End If

// Перемещаем найденное значение в нижнюю половину. values[lo] = values[hi]

// Пересматриваем массив с начала от lo в поисках первого

// элемента, значение которого больше или равно разделяющему.

// Перемещаем найденный элемент в «дырку».

// Теперь «дырка» возникает на его месте. lo = lo + 1

While (values[lo] < divider) lo = lo + 1

If (lo >= hi) Then <Выходим из внешнего цикла While.> End While

If (lo >= hi) Then

// Левая и правая части массива сходятся посередине,

// значит, мы закончили. Помещаем сюда разделяющий

// элемент и выходим из цикла While. lo = hi

values[hi] = divider

<Выходим из внешнего цикла While.>

End If

// Перемещаем найденное значение в верхнюю половину. values[hi] = values[lo]

End While

// Рекурсивно сортируем две части. Quicksort(values, start, lo - 1) Quicksort(values, lo + 1, end)

End Quicksort

В самом начале алгоритм проверяет, сколько элементов содержит данная часть массива. Если не более одного, значит, массив отсортирован и код завершает работу. Если элементов как минимум два, алгоритм сохраняет первый как разделяющий. Вы можете выбрать разделитель любым другим способом, просто переместите его в начало части, чтобы алгоритм мог воспользоваться им во время выполнения по- следующих шагов.

Далее вводятся переменные lo и hi, которые сохраняют высший индекс в ниж- ней части массива и низший в верхней части соответственно. Алгоритм использует их, чтобы отслеживать размещение элементов в обеих половинах и выяснять, где остается «дырка» после каждого шага.

Затем начинается бесконечный цикл While и работает до тех пор, пока ниж- няя и верхняя части массива не сойдутся. Внутри внешнего цикла While алгоритм движется по массиву с конца, от индекса hi, пока не найдет элемент, который дол- жен находиться в нижней части массива. Он перемещает его в «дырку», оставлен- ную разделяющим элементом. Затем алгоритм ведет поиск с самого начала, от ин- декса lo, пока не отыщет элемент, который должен располагаться в верхней ча- сти массива. Этот элемент помещается в «дырку», оставленную предыдущим пе- реставленным элементом.

Как только две части массива сходятся, в место соединения добавляется разде- ляющий элемент, и алгоритм рекурсивно вызывает сам себя для сортировки частей.

#### Использование быстрой сортировки

Если элементы делятся без использования стеков, быстрая сортировка не по- требует дополнительной памяти, разве что для хранения нескольких переменных. Подобно пирамидальной сортировке, она имеет предположительное время работы *O*(*N*  log *N*), в худшем случае — *O*(*N*2). У пирамидальной сортировки производи- тельность всегда равна *O*(*N*  log *N*), поэтому в некотором смысле такие алгорит- мы более безопасны и изящны. Хотя на практике быстрая сортировка оказывает- ся быстрее, и именно ей отдают предпочтение программисты. Подобный алгоритм также присутствует в большинстве готовых библиотек. (В Java иногда использу- ется сортировка слиянием. В следующем разделе вы узнаете больше о ее особен- ностях, а из врезки «Устойчивая сортировка» поймете, чем обусловлен такой ис- ключительный выбор.)

Помимо скорости, быстрая сортировка имеет еще одно преимущество — ее можно распараллелить. Предположим, в компьютере более одного процессора — такое не редкость в наши дни. Каждый раз, когда алгоритм разбивает массив на две части, он может отсортировать их с помощью различных процессоров. Тео- ретически многоядерный компьютер способен задействовать *O*(*N*) процессоров для сортировки списка в течение *O*(log *N*) времени, но на практике их, как пра- вило, два или четыре. Таким образом, истинное время работы нужно разделить на количество процессоров плюс дополнительные затраты на организацию раз- личных потоков вычислений. Это не изменит поведение алгоритма, но повысит его производительность.

Поскольку в худшем случае быстрая сортировка работает за время *O*(*N*2), за- имствовать ее из готовых библиотек небезопасно с криптографической точки зре- ния. Если алгоритм использует простую стратегию разделения элементов (на- пример, выбор первого элемента), взломщик может спровоцировать отказ от об- служивания, так называемую DoS-атаку: сформировать массив, который содер- жит элементы в порядке, дающем наихудшую производительность, и заставить вашу программу пройти по нему. Большинство программистов не задумываются о таком развитии событий, но если это для вас важно, лучше использовать случай- ные стратегии выбора разделяющего элемента.

#### Сортировка слиянием

Подобно быстрой сортировке, сортировка слиянием использует стратегию

«разделяй и властвуй». Если в первом случае выбирается разделитель, а остав- шиеся элементы делятся на две группы — бо'льшие и меньшие его, то во втором разделение происходит на две равные части, а затем алгоритм рекурсивно вызы- вает сам себя для их сортировки. Потом отсортированные половины сливаются в комбинированный отсортированный список. Описанный алгоритм представлен в следующем псевдокоде.

Mergesort(Data: values[], Data: scratch[], Integer: start, Integer: end)

// Если в массиве только один элемент — он отсортирован. If (start == end) Then Return

// Разбиваем массив на левую и правую половины. Integer: midpoint = (start + end) / 2

// Вызываем Mergesort для сортировки двух половин. Mergesort(values, scratch, start, midpoint) Mergesort(values, scratch, midpoint + 1, end)

// Соединяем отсортированные половины. Integer: left\_index = start

Integer: right\_index = midpoint + 1 Integer: scratch\_index = left\_index

While ((left\_index <= midpoint) And (right\_index <= end))

If (values[left\_index] <= values[right\_index]) Then scratch[scratch\_index] = values[left\_index] left\_index = left\_index + 1

Else

End If

scratch[scratch\_index] = values[right\_index] right\_index = right\_index + 1

scratch\_index = scratch\_index + 1 End While

// Завершаем копирование из непустой половины.

For i = left\_index To midpoint scratch[scratch\_index] = values[i] scratch\_index = scratch\_index + 1

Next i

For i = right\_index To end scratch[scratch\_index] = values[i] scratch\_index = scratch\_index + 1

Next i

// Копируем значения в исходный массив.

For i = start To end

values[i] = scratch[i]

Next i End Mergesort

Кроме сортируемых массивов, начального и конечного индексов, алгоритм так- же берет в качестве параметра рабочий массив, который использует для объедине- ния отсортированных половин.

Код начинает работу с проверки общего количества элементов в части массива. Если элемент один, значит, массив отсортирован и алгоритм прекращает выполнять-

ся. Если элементов несколько, алгоритм рассчитывает индекс того, что находится в середине части, и рекурсивно вызывает сам себя для сортировки половин. Затем отсортированные половины объединяются. Они пересматриваются в цикле, при этом меньший элемент копируется в рабочий массив. Если одна половина пуста, алгоритм копирует остальные элементы из другой половины. На завершающем этапе объединенные элементы из рабочего массива переносятся обратно в исходный массив values.

**ЗАМЕЧАНИЕ**

 Можно объединить сортируемые половины и без рабочего массива, но это более сложный и медленный процесс, поэтому программисты такой массив

используют крайне редко.

В дереве, представленном на рисунке 6.11, изображены вызовы быстрой сор- тировки, которые осуществляются, когда значения в массиве сбалансированы наи- лучшим образом, другими словами, алгоритм делит элементы на равные части при каждом шаге. Как мы уже знаем, аналогичный подход используется в сортировке слиянием, поэтому можно утверждать, что рисунок 6.11 в большей степени отно- сится именно к ней.

Анализ времени работы для быстрой сортировки применим и для сортировки слиянием. Таким образом, производительность рассмотренного выше алгоритма составит *O*(*N*  log *N*). Как и в случае пирамидальной сортировки, она не зависит от изначального расположения элементов, поэтому всегда одинакова. Нет здесь и худшего варианта, как в быстрой сортировке.

Сортировку слиянием тоже можно распараллелить. Когда алгоритм рекурсив- но вызывает сам себя, он вправе передать один из таких вызовов другому процес- сору. Однако это требует некоторой координации: исходный вызов должен подо- ждать, пока оба рекурсивных вызова закончатся, чтобы объединить их результа- ты. Быстрая сортировка, напротив, может приказать рекурсивным вызовам отсо- ртировать определенную часть массива и не ждать их возвращения.

Сортировка слиянием особенно полезна, когда данные не содержатся в па- мяти единовременно. Например, программе нужно отсортировать 1 млн запи- сей о клиентах, каждая из которых занимает 1 Мбайт. Для комплексной загруз- ки данных ей понадобится 1018 байт памяти, или 1000 Тбайт, что намного боль- ше, чем в основной массе компьютеров. Сортировка слиянием не требует тако- го количества ресурсов, алгоритму даже не нужно обращаться к элементам мас- сива, пока не вернутся его рекурсивные вызовы. Он проходит через отсортиро- ванные половины линейным способом и объединяет их, что сокращает необходи- мость разбивать память компьютера на страницы. В противоположность выше- сказанному быстрая сортировка, перемещая элементы в разные половины мас- сива, перепрыгивает с одного места на другое, увеличивая разбивку на страницы и очень сильно замедляя алгоритм.

Сортировка слиянием была особенно полезна во времена, когда большие объе- мы данных хранились на ленточных накопителях. Они работали тем эффективнее, чем меньше требовалось совершить перемоток для продолжения записи. (Сор- тировка данных, которые не поместились в памяти, называется *внешней.*) Специ- ализированные версии сортировки слиянием по-своему интересны, но, поскольку они больше не используются, рассматривать их не имеет смысла.

Более распространенный способ сортировки больших объемов данных — сор- тировка ключей элементов. Например, запись о клиенте может занимать 1 Мбайт, а его имя — только 100 байт. В этом случае проще создать отдельный индекс, ко- торый вначале подберет имена к номерам записей, а затем отсортирует исключи- тельно их. В дальнейшем даже если у вас появится 1 млн клиентов, для сортиров- ки имен потребуется лишь 100 Мбайт памяти — объем, который компьютер может легко хранить. (В главе 11 описываются деревья B и B+, которые часто использу- ются системами баз данных для хранения и сортировки ключей.)

**УСТОЙЧИВАЯ СОРТИРОВКА**

*Алгоритм устойчивой сортировки* сохраняет относительное расположение эквивалентных значений. Например, программа сортирует объекты Car по свойству Cost, причем объекты A и B имеют одинаковые значения данного свойства. Если объект A изначально предшествует объекту B в массиве, то в алгоритме устойчивой сортировки он сохранит это положение и в отсорти- рованном массиве.

Когда в качестве сортируемых элементов выступают типы значений, например целые числа, даты или строки, то при их совпадении две записи окажутся экви- валентными. Другими словами, абсолютно не важно, является ли сортировка устойчивой. Так, если в массиве есть две записи со значением 47, то все рав- но, какое из них станет первым в отсортированном массиве. Если же для вас важно, чтобы объекты были перераспределены нужным образом, устойчивая сортировка позволяет пересматривать массив много раз для получения ре- зультата, отсортированного по нескольким ключам, например Maker и Cost в случае с Car.

Сортировку слиянием легко использовать как устойчивую (это демонстрирует описанный ранее алгоритм), именно к ней обращается метод Arrays.sort в библиотеке Java. Ее также можно распараллелить и реализовать на компью- терах с более чем одним процессором (см. главу 18). И хотя в ряде случаев быстрая сортировка оказывается эффективнее, сортировка слиянием все же имеет свои преимущества.

## Алгоритмы быстрее O(N  log N)

В текущей главе уже говорилось, что самому быстрому алгоритму, использу- ющему сравнение, для сортировки *N* элементов требуется как минимум *O*(*N*  log *N*) времени. Рассмотренные сортировки (пирамидальная, слиянием и быстрая) в ожидаемом случае достигают такого предела, поэтому может показаться, что тема исчерпана. Но если заострить внимание на словах «алгоритму, использу- ющему сравнения» и воспользоваться другим методом, то указанная произво- дительность не предел. В следующих подразделах вы познакомитесь с двумя алгоритмами, где сортировка осуществляется за время, которое значительно меньше, чем *O*(*N*  log *N*).

#### Сортировка подсчетом

Подобный метод удобно использовать в том случае, если сортировать прихо- дится целые числа, лежащие в относительно небольшом диапазоне. Например, вам нужно упорядочить 1 млн целых чисел от 0 до 1000. Основная идея заключается

в том, чтобы установить количество элементов массива с определенным значени- ем, а затем скопировать это значение по порядку нужное количество раз обратно в массив. Реализацию такого метода показывает следующий код.

Countingsort(Integer: values[], Integer: max\_value)

// Создаем массив счетчиков. Integer: counts[0 To max\_value]

// Инициализируем массив счетчиков

// (требуется не во всех языках программирования).

For i = 0 To max\_value

counts[i] = 0

Next i

// Считаем количество элементов для каждого значения. For i = 0 To <количество значений> - 1

// Прибавляем 1 к счетчику данного значения. counts[values[i]] = counts[values[i]] + 1

Next i

// Копируем значения в исходный массив. Integer: index = 0

For i = 0 To max\_value

// Копируем значение i в массив counts[i] раз. For j = 1 To counts[i]

values[index] = i index = index + 1

Next j

Next i

End Countingsort

Параметр max\_value содержит наибольшее значение из массива (если оно не задано, алгоритм определит его, просмотрев массив); *M* — общее количество эле- ментов в массиве counts (*M* = max\_value + 1), а *N* — в массиве values. Если ис- пользуемый вами язык программирования не инициализирует массив counts так, чтобы он содержал нули, алгоритм затратит на это *М* шагов, после чего выполнит еще *N* шагов, чтобы посчитать значения.

В завершение работы алгоритм копирует значения обратно в исходный массив (каждое один раз), поэтому данная часть процесса займет *N* шагов. Если какая-либо из записей в массиве counts все еще равна 0, программа потратит определенное время на перепрыгивание через нее. В худшем случае, когда все значения одинако- вы, а массив counts содержит в основном нули, для аналогичного действия пона- добится *M* шагов. Отсюда общее время работы алгоритма: *O*(2*N* + *M*) = *O*(*N* + *M*). Если *M* относительно мало по сравнению с *N*, полученная производительность ока- жется лучше *O*(*N*  log *N*), которую демонстрирует пирамидальная сортировка и дру- гие описанные выше алгоритмы.

На практике для тестового массива с 1 млн элементов и диапазоном значений от 0 до 1000, быстрая сортировка заняла 4,29 с, а сортировка подсчетом — всего 0,03 с. Следует отметить, что для быстрой сортировки выбранный пример оказался не са- мым лучшим, поскольку среди значений было множество дубликатов (приблизи- тельно 1000), а данный алгоритм справляется с ними плохо. Пирамидальная со- ртировка аналогичного массива длилась около 1,02 с. Она показала себя эффектив- нее быстрой сортировки, но все равно значительно уступила сортировке подсчетом.

#### Блочная сортировка

Алгоритм *блочной*, или, как ее еще называют, *корзинной*, *сортировки* делит элементы на блоки, а затем сортирует их с помощью рекурсивного вызова блочной сортировки либо другого алгоритма и присоединяет содержимое блоков к исходно- му массиву. В следующем псевдокоде представлен высокоуровневый алгоритм, демонстрирующий этот метод.

Bucketsort(Data: values[])

<Создаем блоки.>

<Распределяем элементы по блокам.>

<Сортируем блоки.>

<Собираем элементы блоков в исходный массив.> End Bucketsort

Если при использовании *M* блоков значения в массиве из *N* элементов распре- делены относительно равномерно и их диапазон делится поровну, можно ожидать, что в каждый блок попадет *N*/*M* элементов.

В качестве примера возьмем массив, изображенный вверху на рисунке 6.12. Он содержит 10 элементов со значениями от 0 до 99. На первом этапе алгоритм делит элементы на блоки, которые в нашем случае состоят из 20 значений: от 0 до 19, от 20 до 39 и т. д. На втором этапе алгоритм сортирует каждый блок, а на третьем объединяет их значения для построения результата сортировки.

Блоками могут выступать стеки, связные списки, очереди, массивы или любые другие структуры данных.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 29 | 85 | 19 | 10 | 91 | 9 | 63 | 7 | 97 | 61 |
|  | | | | | Разбиение | | | | |

0–19 20–39 40–59 60–79 80–99

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 19 | 29 |  |  | 63 | 85 |
| 10 |  |  |  | 61 | 91 |
| 9 |  |  |  |  | 97 |
| 7 |  |  |  |  |  |
|  | | Сортировка | | | |

0–19 20–39 40–59 60–79 80–99

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7  9  10  19 | 29 |  | 61  63 | 85  91  97 |

Объединение

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 9 | 10 | 19 | 29 | 61 | 63 | 85 | 91 | 97 |

**Рис. 6.12.** Этапы блочной сортировки

Чтобы разбить *N* равномерно распределенных элементов по блокам, алгорит- му понадобится *N* шагов, если не учитывать перенос элемента в блок. Обычно такое преобразование занимает фиксированный отрезок времени. Предположим,

в массиве находятся целые числа от 0 до 99, как в примере на рисунке 6.12. Вам надо перенести элемент со значением v в блок номер v/20 . Этот номер можно рассчитать за определенное время, таким образом, для распределения элементов потребуется *O*(*N*) шагов.

l J

Если таких блоков *M*, для сортировки каждого из них понадобится *F*(*N*/*M*) шагов, где *F* — функция времени работы алгоритма, который вы используете при сортировке блоков. Получается, что общее время сортировки всех блоков составит *O*(*M*  *F*(*N*/*M*)).

Чтобы собрать все отсортированные значения назад в массив, надо выполнить *N* основных шагов и, возможно, *O*(*M*) дополнительных, чтобы пропустить пустые блоки. Но если *M* < *N*, для всей операции потребуется *O*(*N*) шагов.

При суммировании всех этапов получим следующее: *O*(*N*) + *O*(*M*  *F*(*N*/*M*)) +

+ *O*(*N*) = *O*(*N* + *M*  *F*(*N*/*M*)). Если *M* является фиксированной долей *N*, то *N*/*M*, а следовательно, и *F*(*N*/*M*) — константы. Тогда формула упрощается до *O*(*N* + *M*).

На практике число *M* должно составлять относительно большую долю *N*, чтобы алгоритм мог хорошо работать. Если у вас 10 млн записей и 10 блоков, то в каждом из них будет в среднем по 1 млн записей. В отличие от сортировки подсчетом, произ- водительность блочной сортировки не зависит от количества используемых блоков.

## Резюме

Алгоритмы сортировки, описанные в текущей главе, используют разные ме- тоды и обладают различными характеристиками. Все они сведены в таблицу 6.1.

**Òàáëèöà 6.1.** Õàðàêòåðèñòèêè àëãîðèòìîâ ñîðòèðîâêè

|  |  |  |  |
| --- | --- | --- | --- |
| **Сортировка** | **Время работы** | **Метод** | **Область использования** |
| Вставкой | *O*(*N*2) | Вставка | Очень малые массивы |
| Выбором | *O*(*N*2) | Выбор | Очень малые массивы |
| Пузырьковая | *O*(*N*2) | Двусторонние прохожде- ния, ограничения рассма- триваемых пределов | Очень малые и частично сортированные массивы |
| Пирамидальная | *O*(*N* log *N*) | Кучи, хранение полных деревьев в массиве | Крупные массивы с неиз- вестным распределением |
| Быстрая | *O*(*N* log *N*) ожидаемое,  *O*(*N*2)  в худшем случае | «Разделяй и властвуй», перемещение элементов в позицию, рандомиза- ция во избежание худше- го случая | Крупные массивы без большого количества дубликатов, параллель- ная сортировка |
| Слиянием | *O*(*N* log *N*) | «Разделяй и властвуй», объединение, внешняя сортировка | Крупные массивы с неиз- вестным распределением, большие объемы данных, параллельная сортировка |
| Подсчетом | *O*(*N* + *M*) | Счет | Крупные массивы с до- статочно единообразным распределением значений |

Рассмотренные алгоритмы демонстрируют набор полезных методов и обеспе- чивают хорошую производительность для решения широкого спектра задач, но это

далеко не все. Существуют десятки других способов сортировки. Некоторые из них являются модификациями приведенных алгоритмов, в других используются совершенно иные методы. Так, в главе 10 вы можете познакомиться с деревьями, которые также задействуются при сортировке данных, дополнительную информа- цию легко найти в Интернете.

В этой главе приводится только описание способов сортировки, но не поясняет- ся, зачем она нужна, а тем временем самая простая сортировка может быть очень по- лезна. Например, просмотр информации о банковских клиентах с учетом сортировки по балансу поможет определить, кому из них нужно уделить особое внимание. Еще одно преимущество упорядочивания данных — поиск. Если вы расположите имена клиентов в алфавитном порядке, то найти конкретного человека станет намного про- ще. О методах поиска в отсортированных данных мы и поговорим в следующей главе.

## Упражнения

1. Напишите программу, которая реализует сортировку вставкой.
2. Цикл For i, который используется в алгоритме, сортирующем вставкой, пе- ресматривает индексы от 0 до последнего в массиве. Что произойдет, если начальный индекс будет равен 1? Изменит ли это время работы алгоритма?
3. Создайте программу, которая реализует сортировку выбором.
4. Какие изменения, наподобие тех, что приводятся в упражнении 2, вы бы внесли в сортировку выбором? Повлияет ли это на время работы алгоритма?
5. Напишите программу, которая реализует пузырьковую сортировку.
6. Добавьте к программе из предыдущего упражнения первое и третье улуч- шения пузырьковой сортировки из подраздела «Пузырьковая сортировка» в разделе «Алгоритмы *O*(*N*2)» текущей главы (прохождение вниз и вверх, отслеживание последней перестановки).
7. Постройте очередь с приоритетом на основе кучи, используя массив, в ко- тором не нужно изменять размеры. Назначьте ему фиксированный размер, возможно, 100 элементов, а затем отслеживайте те, с которыми работает про- грамма. (Подсказка: используйте два массива (один для строковых значений, другой для соответствующих приоритетов), расположите элементы соглас- но их приоритетам.)
8. Определите время работы, необходимое для добавления элементов в оче- редь с приоритетом на основе кучи. Проанализируйте, сколько времени по- надобится для удаления.
9. Напишите программу, которая реализует пирамидальную сортировку.
10. Попробуйте обобщить метод пирамидальной сортировки применительно к полному бинарному дереву так, чтобы в нем можно было хранить полное дерево степени *d*. Если дан индекс вершины *p*, каковы будут его дочерние и родительские индексы?
11. Напишите программу, которая реализует быструю сортировку со стеками (воспользуйтесь готовыми стеками из вашей среды программирования либо создайте собственные).
12. Реализуйте алгоритм быстрой сортировки, заменив стеки на очереди (вос- пользуйтесь готовыми очередями из вашей среды программирования или постройте собственные). Проанализируйте преимущества и недостатки та- кого преобразования.
13. Напишите программу, которая реализует быструю сортировку на месте.
14. Для быстрой сортировки наихудшим случаем является тот, при котором элементы отсортированы в прямом или обратном порядке либо содержат множество дубликатов. От первых двух проблем можно избавиться, если выбрать случайные разделяющие элементы. Как избежать третьей?
15. Напишите программу, которая реализует сортировку подсчетом.
16. Представьте, что значения массива counts находятся в диапазоне от 100 000 до 110 000, в нем 110 001 запись и количество элементов относительно мало. Как модифицировать сортировку подсчетом, чтобы добиться хорошей про- изводительности в этой ситуации?
17. Если массив содержит *N* элементов, охватывающих диапазон от 0 до *M* – 1, что случится с блочной сортировкой при использовании *M* блоков?
18. Напишите программу, которая реализует блочную сортировку. Позвольте пользователю определить количество элементов, их максимальное значе- ние и количество блоков.
19. Какой из алгоритмов сортировки будет хорошо работать, а какой нет для следующих наборов данных:
    * 10 значений с плавающей запятой;
    * 1000 целых чисел;
    * 1000 имен;
    * 100 000 целых чисел со значениями от 0 до 1000;
    * 100 000 целых чисел со значениями от 0 до 1 млрд;
    * 100 000 имен;
    * 1 млн значений с плавающей запятой;
    * 1 млн имен;
    * 1 млн целых чисел с равномерным распределением;
    * 1 млн целых чисел с неравномерным распределением?

# Глава 7 ПОИСК

В предыдущей главе мы узнали о том, как упорядочивать данные, и выясни- ли, что алгоритмы быстрой и древовидной сортировки позволяют легко справить- ся с большим объемом значений, а блочная сортировка и сортировка подсчетом показывают неплохую производительность только при определенных условиях. В этой главе речь пойдет об алгоритмах, которые помогают находить конкретные данные в отсортированном массиве. Благодаря им, например, легко отыскать нуж- ную фамилию в телефонной книге всего за 1–2 мин, если все десятки тысяч запи- сей расположены в алфавитном порядке. Представьте, что было бы, если бы спи- сок был неотсортированным.

**ЗАМЕЧАНИЕ**

Приведенные ниже алгоритмы работают с простыми массивами, а не со спе- циализированными структурами данных. Хотя последние, к числу которых от- носятся и деревья, также быстро находят элемент с определенным значением (читайте об этом в главе 10).

**ЗАМЕЧАНИЕ**

Некоторые языки программирования предоставляют свои инструменты для поиска элементов в сортированном массиве, например в классе Array в .NET Framework предусмотрен метод BinarySearch. Как правило, результаты их работы весьма неплохие, поэтому для экономии времени и отладки сор- тировочного кода можно пользоваться заложенными возможностями языка. Но понимать принцип действия поисковых алгоритмов все же нелишне: впол- не вероятно, что ваш вариант окажется лучше, чем предлагаемые инструмен- ты. Например, интерполяционный поиск в некоторых случаях показывает себя намного быстрее бинарного.

## Линейный поиск

*Линейный, или исчерпывающий, поиск* циклически проходит по массиву, пытаясь отыскать целевой элемент. На рисунке 7.1 таким элементом является число 77.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 21 | 22 | 39 | 41 | 47 | 49 | 52 | 57 | 60 | 65 | 71 | 77 | 90 | 91 |

**Рис. 7.1.** Линейный поиск

В отличие от бинарного и интерполяционного поиска, линейный работает не с массивами, а со связными списками, где нельзя запросто перепрыгнуть из одной части в другую. Списки могут быть и несортированными, но с отсортированными алгоритм экономит значительную часть времени: он останавливается, дойдя до элемента со значением, превышающим целевое, и не ведет дальнейший поиск того, чего заведомо не существует.

В следующем псевдокоде представлен линейный алгоритм поиска для массива.

// Находим индекс целевого элемента в отсортированном массиве.

// Если элемента в массиве нет, возвращаем -1. Integer: LinearSearch(Data values[], Data target)

For i = 0 To <количество значений> - 1

// Проверяем, является ли элемент целевым. If (values[i] == target) Then Return i

// Проверяем, прошли ли мы возможную позицию целевого элемента. If (values[i] > target) Then Return -1

Next i

// Если мы дошли до этой строки, то целевого элемента в массиве нет. Return -1

End LinearSearch

Приведенный алгоритм, возможно, циклически пройдет по всему массиву, прежде чем обнаружит, что целевого элемента нет. Это будет худшим случаем *O*(*N*). В любой другой ситуации время работы составит *O*(*N*). Если суммировать количество шагов, необходимых для поиска каждого элемента в массиве, получится 1 + 2 + 3 + ... + *N* = *N*  (*N* + 1)/2. Разделив сумму на *N*, чтобы получить среднее время поиска для всех элементов, получим (*N* + 1)/2, а это и есть *O*(*N*).

Линейный поиск намного медленнее, чем бинарный или интерполяционный. Главное его преимущество заключается в том, что он работает со связными и не- сортированными списками.

## Бинарный поиск

В бинарном поиске алгоритм отслеживает наименьший и наибольший индек- сы элементов массива — min и max. Изначально они равны первому (нулевому) и последнему индексу соответственно. Затем алгоритм рассчитывает индекс, на- ходящийся между ними, то есть mid. Если целевое значение меньше mid, алго- ритм сбрасывает max, чтобы начать новый поиск в левой половине массива; если оно больше — сбрасывается min и новый поиск ведется в правой половине. Если же целевое значение равно mid, алгоритм возвращает его индекс.

На рисунке 7.2 изображен бинарный поиск числа 77.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 21 | 22 | 39 | 41 | 47 | 49 | 52 | 57 | 60 | 65 | 71 | 77 | 90 | 91 |

**Рис. 7.2.** Бинарный поиск



Описанный алгоритм представлен в следующем псевдокоде.

// Находим индекс целевого элемента в отсортированном массиве.

// Если элемента в массиве нет, возвращаем -1. Integer: BinarySearch(Data values[], Data target)

Integer: min = 0

Integer: max = <количество значений> - 1 While (min <= max)

// Находим разделяющий элемент. Integer: mid = (min + max) / 2

// Проверяем, в какой половине вести поиск – левой или правой. If (target < values[mid]) Then max = mid - 1

Else If (target > values[mid]) Then min = mid + 1 Else Return mid

End While

// Если мы дошли до этой строки, то целевого элемента в массиве нет. Return -1

End BinarySearch

На каждом шаге данный алгоритм делит элементы, среди которых может содержаться целевой, пополам. Если всего элементов *N*, то после прохождения *O*(log *N*) шагов в части массива, где может располагаться целевой элемент, останется только одно значение. Таким образом алгоритм найдет искомое или обнаружит, что действия не принесли должного результата. Это значит, что алгоритм обладает временем работы *O*(log *N*).

## Интерполяционный поиск

Бинарный поиск каждый раз ищет целевой элемент массива в середине раздела, интерполяционный пытается ускорить процесс — он старается угадать расположе- ние целевого элемента в массиве по его значению. Предположим, в массиве содер- жится 1000 элементов со значениями от 1 до 100. Если наша цель — найти число 30, то его нужно искать в районе первой трети массива, где-то рядом с индексом 300. Общее распределение чисел не всегда позволяет получить результат с прицельной точностью, но он может оказаться довольно близким.

На рисунке 7.3 изображен интерполяционный поиск числа 77.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 21 | 22 | **39** | 41 | 47 | 49 | 52 | 57 | 60 | 65 | 71 | 77 | 90 | 91 |

**Рис. 7.3.** Интерполяционный поиск

Следующий псевдокод представляет собой высокоуровневый алгоритм опи- санного метода.

Integer: InterpolationSearch(Data values[], Data target) Integer: min = 0

Integer: max = values.Length - 1 While (min <= max)

// Находим разделяющий элемент.

Integer: mid = min + (max - min) \*

(target - values[min]) / (values[max] - values[min]) If (values[mid] == target) Then Return mid

<Устанавливаем min или max для поиска в левой или правой части.> End While

Return -1

End InterpolationSearch

В данном алгоритме некоторые проблемы остаются нерешенными. Например, расчет mid может привести к выходу за пределы массива или к тому, что его зна- чение не будет находиться между min и max. Эти задачи вам предлагается решить в упражнении 6.

Самая сложная часть приведенного кода — оператор для расчета mid. Что- бы найти его значение, к текущему значению min добавляется расстояние от min до max, масштабируемое по ожидаемой доле расстояния от values[min] до values[max], где должен находиться target.

Например, если значение values[min] равно 100, values[max] — 200, а target — 125, определить место для поиска целевого элемента помогут следу- ющие расчеты.

(target - values[min]) / (values[max] - values[min]) = (125 - 100) / (200 - 100) =

25 / 100 =

0.25

Это значит, что новое значение mid должно располагаться на четверти пути от min до max.

Если данные распределены очень неравномерно и вы ищете наихудшее целе- вое значение, данный алгоритм обладает производительностью *O*(*N*). Если рас- пределение относительно равномерное, ожидаемая производительность составит *O*(log (log *N*)). Однако доказательство этого не входит в рамки данной книги.

## Резюме

В таблице 7.1 представлены расчеты log *N* и log (log *N*) для различных значений *N*. На их основе легко сравнить скорости линейного, бинарного и интерполяционного поиска.

**Òàáëèöà 7.1.** Õàðàêòåðèñòèêè àëãîðèòìîâ

|  |  |  |
| --- | --- | --- |
| ***N*** | **log2 *N*** | **log2(log2 *N*)** |
| 1 000 | 10 | 3,3 |
| 1 000 000 | 19,9 | 4,3 |
| 1 000 000 000 | 29,9 | 4,9 |
| 1 000 000 000 000 | 39,9 | 5,3 |

Линейный поиск наилучшим образом проявляет себя для относительно малых массивов. Бинарный хорошо работает даже с очень большими массивами — он мо- жет найти нужный элемент среди 1 трлн записей всего за 40 шагов.

Интерполяционный поиск подходит для массивов любого размера, которые целесообразно обрабатывать на компьютере. Ему удается найти целевой элемент среди 1 трлн записей всего за пять шагов. По сути, чтобы ожидаемых шагов было более девяти, массив должен состоять из 10154 элементов. В общем случае в интер- поляционном поиске все зависит от распределения значений: иногда алгоритму

«везет» и он находит целевой элемент за один-два шага, а в некоторых случаях могут понадобится и четыре-пять шагов. Но по большому счету именно этот поиск является самым быстрым.

## Упражнения

Если вы еще не знакомы с рекурсией, пропустите упражнения 2, 5 и 7 и верни- тесь к ним после того, как ознакомитесь с главой 8.

1. Напишите программу, реализующую линейный поиск.
2. Создайте программу, применяющую рекурсивный линейный поиск. Каковы ее преимущества и недостатки по сравнению с нерекурсивной версией?
3. Реализуйте алгоритм линейного поиска в отсортированных связных списках.
4. Напишите программу, использующую бинарный поиск.
5. Разработайте программу, реализующую рекурсивный бинарный поиск. Ка- ковы ее преимущества и недостатки по сравнению с нерекурсивной версией?
6. Создайте программу, применяющую интерполяционный поиск.
7. Реализуйте программу рекурсивного интерполяционного поиска. Каковы ее преимущества и недостатки по сравнению с нерекурсивной версией?
8. В каком алгоритме сортировки из главы 6 используется техника, напоми- нающая интерполяционный поиск?
9. Если в массиве содержатся дубликаты, описанные в этой главе алгоритмы би- нарного и интерполяционного поиска не гарантируют возвращения первого элемента. Попробуйте изменить их, чтобы они могли вернуть первый целевой элемент. Определите время работы измененных версий.

# Глава 8 ХЕШ-ТАБЛИЦЫ

В предыдущей главе вы познакомились с бинарным поиском — алгоритмом, ко- торый имеет производительность *O*(log *N*) и ищет элемент в сортированном спи- ске. Он определяет разделяющий элемент в середине той части списка, где, по его мнению, находится целевой элемент, и сравнивает оба значения. Если разделяю- щий элемент оказывается больше целевого, осуществляется рекурсивная провер- ка левой части массива, если меньше — правой.

Еще один поиск, рассмотренный в главе 7, — интерполяционный — пытается спрогнозировать местонахождение целевого элемента путем математических расче- тов. Он обладает временем работы *O*(log (log *N*)), что значительно превышает ана- логичный показатель бинарного поиска. Причина такой высокой скорости кроет- ся в том, что для нахождения значений используется специальная структура дан- ных. (Подобным образом работают алгоритмы блочной сортировки и сортировки подсчетом, описанные в главе 6.)

Хеш-таблицы также позволяют быстро искать значения, поскольку не хранят их в специальном отсортированном списке, а организуют особым образом и легко могут рассчитать местонахождение нужного элемента. Предположим, у вас есть небольшая компания из 20 сотрудников и вам хотелось бы получать информацию о каждом из них по идентификационному номеру (ИН). Один из способов сохра- нить данные — сформировать массив из 100 элементов, в котором ИН будут соот- ветствовать позициям *N* mod 100. Например, сотрудник с ИН 2190 будет размещен в позиции 90, с ИН 2817 — в позиции 17, а с ИН 3078 — в позиции 78. Чтобы най- ти определенного работника, достаточно воспользоваться формулой ИН mod 100 и просмотреть соответствующую запись в массиве. Подобная операция займет вре- мя *O*(1), то есть на проверку окажется более быстрой, чем интерполяционный поиск.

На практике не все так просто. Если сотрудников окажется достаточно много, то среди них найдутся двое, у которых ИН будет соответствовать одному и тому же значению. Скажем, ИН 2817 и 1317 займут в таблице позицию 17. Но даже не- смотря на этот недостаток, выполненное преобразование значений — хорошее на- чало. Оно является отправной точкой для работы с хеш-таблицами. Далее вы по- знакомитесь с этими структурами более подробно и рассмотрите способы их при- менения в различных программах.

## Основы хеш-таблиц

*Хеш-таблица* соединяет данные с ячейками. Зачастую такая связь образует- ся между значением ключа, например ИН или фамилией, и гораздо большей за-

писью, скажем, информацией о сотруднике или клиенте. В силу этой особенности хеш-таблицы иногда называют *ассоциативными массивами*, или, выражаясь менее официальным языком, *словарями*.

Преобразование значений ключей в вид, пригодный для использования в хеш- таблице, называется *хешированием* и описывается особыми хеш-функциями. Эти значения часто похожи, поэтому хорошие функции распределяют их таким образом, чтобы они располагались в разных позициях в таблице.

Предположим, вы хотите находить записи о клиентах в хеш-таблице по фамили- ям. Если у вас работают два сотрудника — Ричардс и Ричардсон, в идеале функция хеширования должна связать их фамилии с двумя разными ячейками. В подобных случаях часто генерируются несколько бессмысленные значения, которые выглядят так, будто истинное значение ключа было разбито на куски.

Если в хеш-таблице размещается достаточно много значений, рано или поздно найдутся два ключа, которые окажутся связанными с одной и той же позицией. Такая ситуация называется *коллизией*. Если она произойдет, вам по- надобится *политика разрешения коллизий*, определяющая, что нужно делать. Как правило, связь между ключом и позицией ищется до тех пор, пока не найдется свободная ячейка.

*Коэффициент заполнения* хеш-таблицы отражает то, насколько много в ней за- писей. Этот показатель влияет на вероятность возникновения коллизий. Больше шансов, что они будут иметь место при добавлении ключа в таблицу, заполненную на 95 %, нежели на 10 %.

Таким образом, для хеш-таблицы нужны:

* структура для хранения данных;
* функция хеширования для связывания ячеек со структурой данных;
* политика разрешения коллизий, определяющая, что нужно делать, когда ключи вступают в противоречие.

Чтобы быть полезной, хеш-таблица должна как минимум добавлять новые эле- менты и находить сохраненные ранее. Пригодится и способность удалять хеширо- ванный ключ, которая, к сожалению, иногда отсутствует.

**ИЗМЕНЕНИЕ РАЗМЕРОВ ХЕШ-ТАБЛИЦ**

По мере работы хеш-таблица может целиком заполниться или стать на- столько полной, что коллизии будут очень вероятны и пострадает произ- водительность. В таком случае понадобится усовершенстовать алгоритм, чтобы определить, где и как изменяется размер хеш-таблицы, и увеличить ее. Иногда доработки потребуются, чтобы таблицу уменьшить и освободить неиспользованное пространство, например если она рассчитана на 1 млн записей, а у вас их всего 10.

Самый простой способ изменить размеры хеш-таблицы — создать новую желаемого размера и рехешировать в нее все элементы из исходной струк- туры. Он применим почти ко всем таблицам, хотя есть и другие варианты осуществить то же действие, например прямое связывание. В следующих разделах вы узнаете более подробно о часто используемых способах по- строения хеш-таблиц.

## Прямое связывание

В хеш-таблице с прямым связыванием (рис. 8.1) значения ключей хранятся в специальных наборах записей, называемых *блоками*. Каждый из них является вершиной связного списка, в котором находятся привязанные к блоку элементы.

0 1 2 3 4 5 6 7 8 9



920



681

291

281



102



424



507

917



689

189

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | |  | |  |  | |  |  |  | |  |  | |
|  |  | |  | |  | | |  | | | |  | | |  |

**Рис. 8.1.** Хеш-таблица с прямым связыванием

Обычно блоки расположены в массиве таким образом, что можно использовать простую функцию хеширования для определения их ключа. Например, если у вас *N* блоков, а ключи численные, свяжите ключ *K* с блоком под номером *K* mod *N*.

Чтобы добавить ключ к хеш-таблице, его нужно вначале соотнести с блоком через функцию хеширования, а затем вставить новую ячейку в верхнюю часть связ- ного списка. Каждая из двух операций потребует *O*(1) шагов, то есть займет очень мало времени. По определению хеш-таблица не должна содержать дублированных значений. Если в ней *B* блоков и *N* элементов, которые распределены достаточно равномерно, то каждый связный список блока будет включать приблизительно *N*/*B* элементов. Таким образом, чтобы проверить, присутствует ли новый элемент в блоке, понадобится *O*(*N*/*B*) шагов. Это значит, что для добавления элемента в хеш-таблицу нужно всего *O*(1) + *O*(*N*/*B*) = *O*(*N*/*B*) шагов.

**ЗАМЕЧАНИЕ**

Поиск элементов в хеш-таблице пройдет быстрее, если связные списки будут содержать ключи в отсортированном порядке. В этом случае алгоритм сделает вывод, что ключа нет, если дойдет до значения больше ключевого, и не станет просматривать список до конца. Теоретически время его работы составит *O*(*N*/*B*), но на практике оно будет немного меньше.

Чтобы найти нужный элемент, программе необходимо хешировать ключ и определить, в каком из блоков он может содержаться, а затем двигаться по связному списку до тех пор, пока не будет достигнут его конец или не обнару- жится искомое. Если вы доберетесь до конца списка, значит, запрашиваемого элемента в хеш-таблице нет. Как и в случае с добавлением элемента, предстоит выполнить *O*(*N*/*B*) шагов.

Удалить элемент из хеш-таблицы с прямым связыванием также несложно: требуется хешировать ключ элемента, чтобы найти нужный блок, и выполнить со- ответствующую операцию в связном списке. Для хеширования понадобится *O*(1) шагов, непосредственно удаление займет *O*(*N*/*B*) шагов. Таким образом, общее время составит *O*(*N*/*B*).

Хеш-таблица с прямым связыванием может расширяться и сжиматься по мере необходимости, поэтому вам не нужно специально изменять ее размер. Однако если связные списки станут слишком длинными, поиск и удаление элементов займут много времени. В этом случае вам понадобится увеличить таблицу, чтобы создать больше блоков. Поскольку при рехешировании таблицы не надо проводить по- иск дубликатов до конца связного списка в каждом блоке, полностью справиться с операцией можно за время *O*(*N*).

## Открытая адресация

Безусловно, прямое связывание имеет свои преимущества. Основное из них за- ключается в том, что общее количество значений не зависит от количества блоков. Однако есть у таких хеш-таблиц и некоторые недостатки. Например, если элемен- тов в блоках накопится слишком много, поиск нужного займет продолжительное время. Чтобы сократить временные затраты, число блоков можно увеличить, но тогда вы рискуете получить ряд пустых, которые потребуют места в памяти и не будут использоваться таблицей.

Еще один способ реализации хеш-таблиц — *открытая адресация*. В этом случае значения хранятся в массиве, а функция хеширования представляет собой некото- рые расчеты. Например, в массиве с *M* записями простая функция хеширования может связать значение ключа *K* с позицией *K* mod *M*.

В разных видах открытой адресации используются различные функции хеши- рования. Неодинакова и политика разрешения коллизий, но в общем случае она выглядит так: для каждого значения в массиве подбирается несколько ячеек, и если первая уже занята, алгоритм пробует использовать вторую, затем третью и так до тех пор, пока не найдет свободную или не придет к выводу, что таковой нет.

Серия ячеек, которую алгоритм подбирает для значения, называется *пробной последовательностью*. По ее средней длине хорошо оценивать наполненность хеш- таблицы. В идеале пробная последовательность должна равняться 1 или 2, большие цифры говорят о полной таблице.

Иногда политика разрешения коллизий такова, что для элемента может не найтись свободной ячейки, даже когда она есть. Если пробная последовательность повторяет саму себя перед тем, как проверить очередную запись, некоторые записи могут остаться неиспользованными.

Чтобы найти элемент в хеш-таблице, алгоритм следует за пробной последова- тельностью, пока не произойдет одно из трех событий.

1. Если пробная последовательность сумела отыскать элемент, задача выполнена.
2. Если пробная последовательность находит пустую запись в массиве, эле- мента нет.
3. Пробная последовательность проверяет *M* записей (по размеру массива) — и алгоритм приходит к выводу, что значение отсутствует. Последователь- ность может перебрать не все элементы, но если пройдет по всем, вы бу- дете знать, что они точно пересмотрены или что целевой элемент не най- ден. Она также может проверить в цикле одну и ту же позицию несколько раз. В любом случае значение не должно присутствовать, поскольку ина- че оно бы добавлялось к массиву с использованием той же пробной после- довательности.

При разумном заполнении хеш-таблицы открытая адресация работает очень быстро. Если длина пробной последовательности равна 1 или 2, добавление и на- хождение элементов выполняются за время *O*(1). Но если массив из *N* элементов существенно переполнен, производительность снижается. В наихудшем случае ал- горитм придет к выводу, что элемента в массиве нет, за время *O*(*N*). Поиск присут- ствующих элементов также будет выполняться крайне медленно.

Вы можете увеличить размер массива, чтобы уменьшить коэффициент напол- ненности хеш-таблицы. Для этого создайте новый массив и рехешируйте элементы в нем. Для каждого из них операция займет *O*(1) времени, а общая производитель- ность алгоритма составит *O*(*N*).

В следующем подразделе описывается еще одна важная задача, связанная с от- крытой адресацией.

#### Удаление элементов

Открытая адресация позволяет относительно быстро добавлять и находить элементы, если массив не слишком полон, но не дает возможности удалять их так, как это делается при прямом связывании. Дело в том, что один элемент массива может являться частью пробной последовательности для другого элемента, нарушив ее, вы уже не сможете найти другой элемент.

Предположим, элементы массива *A* и *B* связаны с одним и тем же индексом *IA*. Первоначально вы добавите в указанную позицию элемент *A*, а затем используете ее в пробной последовательности, чтобы перейти ко второй позиции *IB* и добавить в нее

элемент *B*. Теперь представьте, что элемент *A* удален и нужно найти элемент *B*. Об-

ратившись последовательно к индексу *IA*, вы увидите уже пустую запись и ошибоч- но решите, что элемент *B* отсутствует.

Если обозначить элемент как удаленный, а не делать запись массива пустой, ука- занной проблемы удастся избежать. Скажем, в массиве, где содержатся 32-битные целые числа, запись, не имеющую значения, можно представить как –2 147 483 648, а удаленную — как 2 147 483 647. Тогда, добавляя новый элемент в хеш-таблицу, вы разместите его в пустую запись, если найдете таковую в пробной последова- тельности, а занимаясь поиском удаленного элемента, будете вести его до тех пор, пока не обнаружите соответствующее значение.

Недостаток такого подхода — наполнение таблицы удаленными записями. Это не замедлит вставку элементов, но отрицательно скажется на их поиске. В худшем случае, если массив окажется целиком заполнен текущими и удаленными элемен- тами, чтобы найти нужный элемент или сделать вывод о его отсутствии, вам может понадобиться поиск по всему массиву.

При большом количестве удаленных элементов целесообразно рехешировать текущие значения и сбросить удаленные ячейки массива, чтобы они содержали специальное пустое значение. Если в массиве с разумным коэффициентом запол- нения содержится *N* элементов, это должно занять *O*(*N*) времени.

#### Линейное пробирование

В *линейном пробировании* политика разрешения коллизий добавляет к каждой ячейке постоянное число (чаще всего 1), называемое шагом по индексу, которое ге- нерирует пробную последовательность. При каждом очередном добавлении берется размер массива по модулю, стало быть, при необходимости последовательность возвращается к началу массива.

Предположим, в хеш-таблице 100 элементов, а правило хеширования звучит следующим образом: *N* связано с ячейкой *N* mod 100. Тогда пробная последователь- ность для значения 2197 проверяет ячейки 97, 98, 99, 0, 1, 2 и т. д.

На рисунке 8.2 представлен массив из 10 записей, который уже содержит не- сколько значений. Чтобы добавить в него новое значение 71, используя линейную пробную последовательность, нужно связать его с ячейкой 71 mod 10 = 1. Но эта ячейка уже занята значением 61, поэтому алгоритм переходит к ячейке 2, которая тоже заполнена. Следующей должна быть ячейка 3 — она свободна, и алгоритм размещает там 71.

71

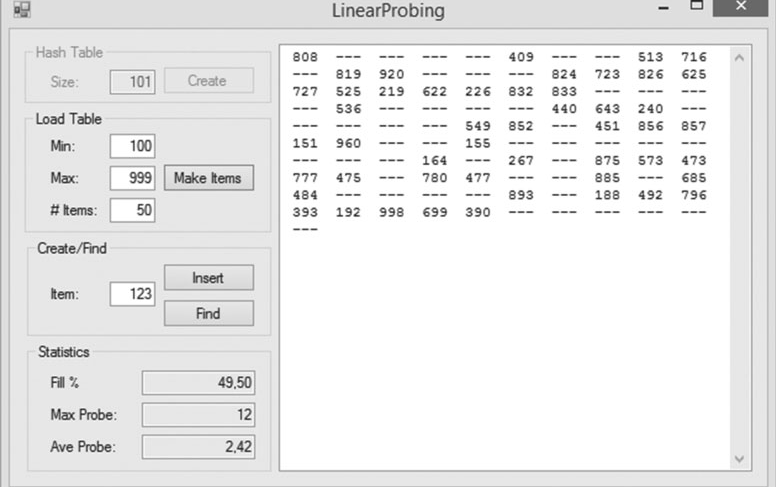


|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 61 | 22 |  | 34 | 84 | 14 |  | 38 |  |

**Рис. 8.2.** Линейная пробная последовательность

Преимущество данного метода — в его простоте. Если необходимо, пробная последовательность пройдет по каждой ячейке массива и вставит элемент в сво- бодное место, если оно еще осталось. Но есть и сопутствующий недостаток — так называемая *первичная кластеризация*, которая проявляется в образовании больших блоков смежных записей и приводит к длинным пробным последовательностям. В результате при добавлении нового элемента и его хешировании к какой-либо за- писи в кластере пробная последовательность вынуждена пройти через весь кластер, чтобы найти свободную ячейку.

Хороший пример первичной кластеризации демонстрирует программа LinearProbing. В ее окне (рис. 8.3) представлена хеш-таблица, состоящая из 101 ячей- ки, причем 50 из них являются заполненными. В общем случае при равномерном распределении элементов внутри таблицы пробная последовательность для каждого будет иметь длину, равную 1. Для новых элементов длина последовательности составит 1 или 2, в зависимости от того, свяжет начальное хеширование элемент со свободной или уже занятой ячейкой. В нашем примере средняя длина пробной последовательности хеш-таблицы составляет 2,42, что немного больше значения, которое можно получить при равномерном распределении. При более высоких коэффициентах заполнения дела станут еще хуже.



**Рис. 8.3.** Окно программы LinearProbing

**ЗАМЕЧАНИЕ**

Хеш-таблица, изображенная на рисунке 8.3, является ответом к упражне- нию 8.3 из приложения Б.

Чтобы понять, как формируются кластеры, рассмотрим пример пустой хеш- таблицы. Предположим, в ней содержится *N* записей и существует вероятность 1/*N*, что она закончится в любой данной позиции при добавлении случайного числа. А теперь представим, что *K* — конечная позиция таблицы. Существует вероятность 1/*N*, что новое случайное число попадет в позицию *K* и линейное пробирование по- пытается расположить элемент в позиции *K* + 1. Но та же вероятность характерна и для случая, при котором новый элемент будет связан с позицией *K* + 1 напрямую. Значит, существует вероятность 2/*N*, что элемент займет конечную позицию *K* + 1 и сформируется малый кластер.

Через какое-то время таких кластеров станет много. Чем больше они будут, тем больше вероятность того, что новый элемент добавится в конец одного из них. В итоге более мелкие кластеры сольются в более крупные, массив заполнится ими — и возникнут длинные пробные последовательности.

В следующих двух подразделах описаны способы, позволяющие бороться с пер- вичной кластеризацией.

#### Квадратичное пробирование

Возникновение больших кластеров при линейном пробировании связано с тем, что новые элементы связываются с ячейками, стоящими в конце группы, и по- степенно увеличивают ее. Предотвратить подобную ситуацию помогает *квадра- тичное пробирование*. Для создания пробной последовательности в качестве шага по индексу берется квадрат количества ячеек. Другими словами, если в линейном пробировании существует последовательность *K*, *K* + 1, *K* + 2, *K* + 3, ... то в квадра- тичном варианте она будет выглядеть так: *K*, *K* + 12, *K* + 22, *K* + 32, ... В этом случае,

если два элемента окажутся связанными с разными позициями в одном и том же кластере, они не обязательно будут придерживаться одной пробной последователь- ности и попадут в конец кластера.

На рисунке 8.4 показана хеш-таблица, в начале которой есть группа из пяти элементов. Новое значение 71 получает пробную последовательность 1, 1 + 12 =

= 2, 1 + 22 = 5, 1 + 32 = 10 и не добавляется к существующему кластеру. Значение 93 поначалу связано с тем же кластером, но согласно собственной пробной последо- вательности 3, 3 + 12 = 4, 3 + 22 = 7 также не попадает в него.

71



98

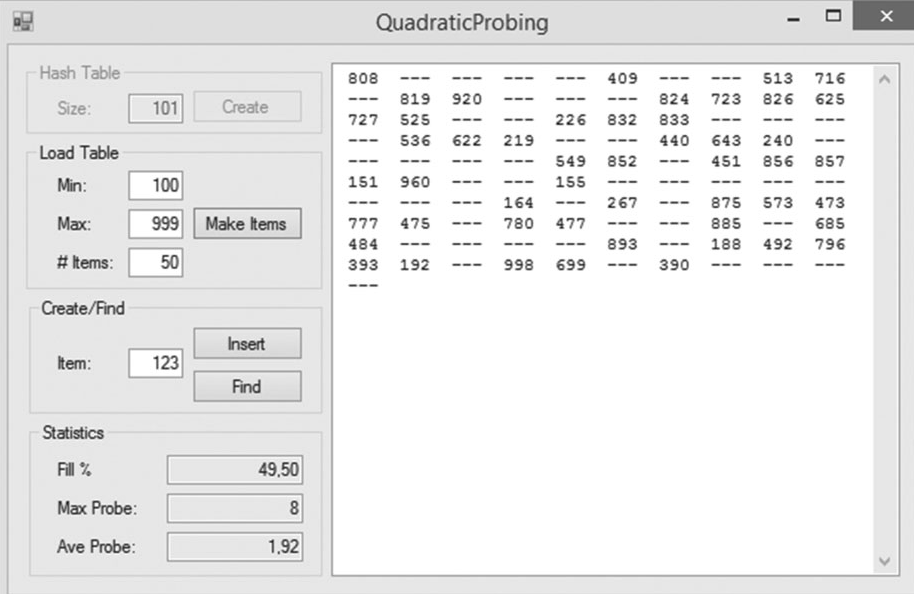
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 61 | 22 | 43 | 34 | 55 |  |  |  |  |  |  |  |  |  |

**Рис. 8.4.** Квадратичное пробирование

В окне программы QuadraticProbing на рисунке 8.5 используется квадратичное пробирование. Если сравнить хеш-таблицу из текущего примера с той, что пред- ставлена на рисунке 8.3, можно увидеть, что квадратичное пробирование дает более короткую длину пробной последовательности (1,92), чем линейное (2,42).

**ЗАМЕЧАНИЕ**

Программа, представленная на рисунке 8.5, является частью ответа к упраж- нению 8.4 из приложения Б.



**Рис. 8.5.** Окно программы QuadraticProbing

Квадратичное пробирование предотвращает первичную, но не *вторичную кла- стеризацию*, при которой значения, связанные с одинаковой начальной позицией в массиве, получают одну и ту же пробную последовательность, иногда очень длин- ную. В результате образуется точно такая же группа элементов, но уже не собранных вместе, а распределенных по всему массиву.

Еще один недостаток квадратичного пробирования связан с тем, что оно может не найти свободную позицию, даже если в хеш-таблице их несколько. Дело в том, что с каждым разом перемещение по массиву происходит все дальше и дальше, и незаполненная ячейка попросту пропускается.

#### Псевдослучайное пробирование

Это пробирование подобно линейному, за исключением того, что шаг по ин- дексу формирует псевдослучайная функция изначально связанной ячейки. Пред- положим, что это ячейка *K*, тогда пробная последовательность будет выглядеть следующим образом: *K*, *K* + *p*, *K* + 2*p*, ... где *p* определяется псевдослучайной функцией.

Подобно квадратичному пробированию, псевдослучайное предотвращает только первичную кластеризацию, но страдает от вторичной: значения, связанные с одной и той же начальной позицией, размещаются в хеш-таблице согласно одной и той же пробной последовательности. Точно так же псевдослучайное пробирование может пропускать некоторые неиспользуемые записи.

#### Двойное хеширование

Чтобы избавиться от вторичной кластеризации значения, связанные с одной и той же начальной ячейкой, должны получать разные пробные последовательно- сти. И здесь пригодится двойное хеширование. Оно похоже на псевдослучайное пробирование, только шаг для индекса задается не псевдослучайной функцией начальной ячейки, а второй функцией хеширования.

Допустим, значения *A* и *B* связаны с позицией *K*. В псевдослучайном пробирова- нии шаг по индексу *p* генерируется функцией *F*1(*K*), затем оба значения используют пробную последовательность *K*, *K* + p, *K* + 2*p*, *K* + 3*p,* ... В двойном хешировании для

связывания начальных значений *А* и *В* применяется функция псевдослучайного хеширования *F*2. В итоге при одном и том же начальном *K* образуются две пробные последовательности с различными шагами по индексу: *pA* = *F*2(*A*) и *pB* = *F*2(*B*) со-

ответственно.

Несмотря на то что двойное хеширование хорошо справляется с первичной и вторичной кластеризациями, оно точно так же, как и псевдослучайное пробиро- вание, может пропускать неиспользуемые записи.

#### Упорядоченное хеширование

Есть такие приложения, в которых значения хешируются всего один раз, а поиск ведется многократно. Речь идет, например, о программах, работающих со словарями, адресными книгами или каталогами товаров. В этом случае гораздо важнее быстро находить значения, нежели их добавлять.

Как вы уже знаете, в хеш-таблице с прямым связыванием поиск элементов за- нимает намного меньше времени, если ее связные списки отсортированы: алгоритм останавливается, если находит элемент со значением большим, чем целевое и не тратит время на проверку всех данных. Сортировку можно использовать и для составления самих хеш-таблиц. В этом случае пробная последовательность для значения *K* будет проверять ячейки массива со значениями *V*1, *V*2 и т. д., то есть все

*Vi*, которые меньше *K*.

Обратите внимание, что значения не должны находиться в строго возраста- ющем порядке. Например, пробная последовательность со значениями 61, 32, 71 будет абсолютно нормальной для 71, но не для 32, поскольку перед ним окажется большее число 61.

Создав массив подобным образом, вы ускорите поиск элемента: алгоритм оста- новится, если найдет значение больше целевого. Следующий высокоуровневый псевдокод ищет элемент в отсортированной хеш-таблице.

// Возвращаем положение ключа в массиве или -1, если такового нет. Integer: FindValue(Integer: array[], Integer: key)

Integer: probe = <начальная позиция в пробной последовательности ключа>

// Повторяем.

While true

// Проверяем, найден ли элемент.

If (array[probe] == key) Then Return probe

// Проверяем, есть ли пустое место.

If (array[probe] == EMPTY) Then Return -1

// Проверяем, не проходили ли мы уже это положение. If (array[probe] > key) Then Return -1

// Пробуем следующее положение из пробной последовательности. probe = <следующая позиция в пробной последовательности ключа>

End While End FindValue

Точное расположение данных в хеш-таблицах, которые описывались ранее, за- висело от порядка, в котором эти данные добавлялись. Так, если в массиве из 10 за- писей функция хеширования помещала элемент *K* в позицию *K* mod 10, то значе- ния 11, 21, 31, 41 занимали позиции от 1 до 4 именно в таком порядке. Они могли

бы разместиться и в обратной последовательности (41, 31, 21, 11), при условии что вы добавляли их именно так.

Предположим, элементы в хеш-таблицу вносятся в порядке от наименьшего к наибольшему. В этом случае пробная последовательность для нового значения должна содержать значения меньше его, ведь они уже находятся в таблице. Только так вы можете проводить быстрый поиск. К сожалению, подобные действия не всег- да осуществимы — нужно заранее знать порядок расположения данных в таблице. А теперь представьте, что вы лишь изредка добавляете в нее по несколько элементов.

К счастью, есть способ создать отсортированную хеш-таблицу и без оглядки на существующий порядок. Согласно ему, для вновь добавляемого элемента ищет- ся пустая позиция с помощью пробной последовательности. Если в предлагаемой алгоритмом позиции обнаруживается элемент с большим значением, он уступает

место новому элементу, а сам подвергается рехешированию. В ходе процесса мо- жет случиться так, что будет найдено еще бол'ьшее значение, тогда уже оно лишит- ся собственной позиции и алгоритм займется его перемещением. Так продолжит- ся до тех пор, пока не отыщется абсолютно пустое место.

Описанные действия представлены в следующем высокоуровневом псевдокоде.

AddItem(Integer: array[], Integer: key)

Integer: probe = <начальная позиция в пробной последовательности>

// Повторяем.

While true

// Проверяем, есть ли пустое место.

If (array[probe] == EMPTY) Then array[probe] = key Return

End If

// Проверяем, является ли найденное значение больше ключа. If (array[probe] > key) Then

// Помещаем сюда ключ и рехешируем другой элемент. Integer: temp = array[probe]

array[probe] = key key = temp

End If

// Пробуем следующее положение из пробной последовательности. probe = <следующая позиция в пробной последовательности>

End While End AddItem

Последний шаг внутри цикла While приравнивает probe к следующему по- ложению в текущей пробной последовательности. В линейном и псевдослучайном пробировании, а также в двойном хешировании вы сумеете определить его, даже если поменяли рехешируемое значение key на бол'ьшее. Так, при двойном хеширо- вании можно применить функцию второго хеширования к новому значению key, чтобы определить шаг по индексу в новой пробной последовательности и двигать- ся по ней с указанного места. В квадратичном пробировании подобный метод не сработает, поскольку вам, вероятно, понадобится знать, как долго алгоритм шел по новой пробной последовательности, чтобы добраться до этого места.

Приведенный метод действен потому, что вы заменяете найденные большие значения меньшими, то есть пробная последовательность так и остается отсортиро- ванной. Единственным сомнительным значением оказывается новое бо'льшее зна- чение, которое рехешируется. Оно занимает позицию, которая делает его пробную последовательность отсортированной.

## Резюме

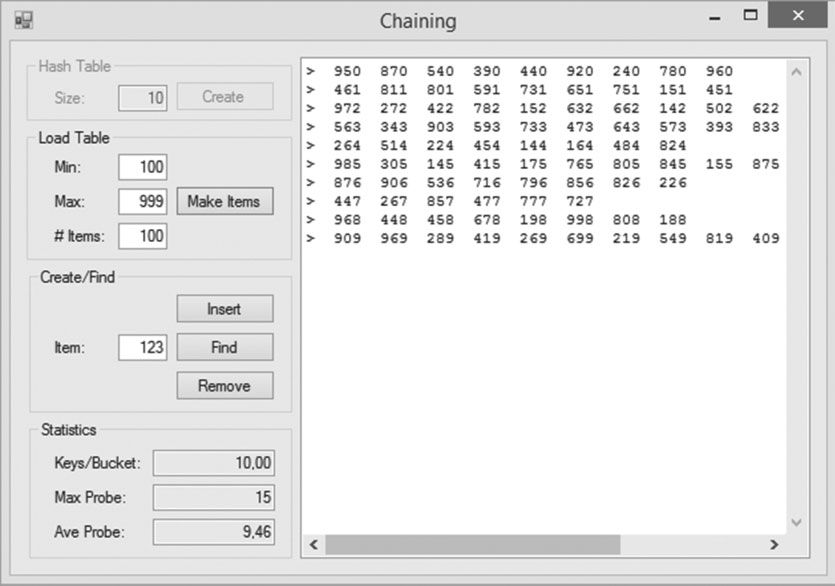
Хеш-таблица позволяет очень быстро хранить и находить значения. Если она обладает разумным коэффициентом заполнения, то отыскать в ней нужный элемент несложно. Когда данных становится слишком много, скорость поиска снижается. Если же их очень мало, то ресурсы памяти расходуются неэкономно.

Возрастающий коэффициент заполнения хеш-таблицы — признак того, что не- обходимо изменить ее размеры. Это потребует определенных временных затрат и до- полнительной памяти, однако подобный временно-пространственный компромисс вполне обычен для алгоритмов и приводит к улучшению их производительности.

Упорядоченное хеширование предполагает другой подход к ускорению по- иска данных. В этом случае понадобится потратить дополнительное время на по- строение хеш-таблицы. Если при вставке нового значения алгоритм столкнется с бо'льшим значением, ему придется поменять их местами и заняться рехеширова- нием бо'льшего. Здесь удобно использовать рекурсию, то есть заставить алгоритм вставки вызвать самого себя. В следующей главе вы узнаете, как это сделать, и по- знакомитесь с достоинствами и недостатками указанного метода. Кроме того, по- лучите сведения о том, как удалить рекурсию из программы, если глубокие стеки вызовов или частый пересчет значений вызывают проблемы.

## Упражнения

Для упражнений, где необходимо создать хеш-таблицу, используйте интер- фейс, подобный тому, что показан на рисунке 8.6. Тут кнопка Create создает новую хеш-таблицу, Make Items разрешает пользователю единовременно добавить в нее много случайных элементов, а Insert è Find соответственно вставляют и находят один элемент. После каждого изменения в таблице или ее данных нужно отобразить количество ключей на один блок для алгоритмов прямого связывания или коэф- фициент заполнения для алгоритмов открытой адресации. Кроме того, следует продемонстрировать максимальную и среднюю длину пробной последовательности при попытке найти все значения между минимальным и максимальным, которые фигурируют в таблице.



**Рис. 8.6.** Окно программы Chaining, позволяющей создавать и тестировать хеш-таблицы

Во всех программах хеширования используйте единый псевдослучайный ге- нератор чисел, формирующий одни и те же значения. Это позволит вам сравнить поведение разных алгоритмов.

1. Напишите программу для создания хеш-таблицы с прямым связыванием.
2. Измените программу из предыдущего упражнения так, чтобы можно было использовать сортированные связные списки. Сравните средние длины проб- ных последовательностей двух программ для случая, когда в хеш-таблицах задействуются 10 блоков и содержится 100 элементов.
3. Начертите график средней длины пробной последовательности для про- грамм из упражнений 1 и 2, где речь идет о хеш-таблицах с 10 блоками и 50, 100, 150, 200 и 250 элементами. Какой вывод можно сделать на основе по- лученных данных?
4. Напишите программу для создания хеш-таблицы с использованием открытой адресации и линейного пробирования.
5. Разработайте программу для создания хеш-таблицы с использованием от- крытой адресации и квадратичного пробирования.
6. Напишите программу для создания хеш-таблицы с использованием открытой адресации и псевдослучайного пробирования.
7. Разработайте программу для создания хеш-таблицы с использованием от- крытой адресации и двойного хеширования.
8. Линейное пробирование всегда находит пустое место для значения, если оно существует. Однако квадратичное и псевдослучайное пробирование, а также двойное хеширование могут пропускать пустые записи и приходить к лож- ному выводу о том, что таблица заполнена. Как выбрать размер таблицы *N*, чтобы предотвратить вывод о том, что хеш-таблица укомплектована, если это не так?
9. Напишите программу для создания хеш-таблицы с использованием откры- той адресации и упорядоченного квадратичного пробирования.
10. Воспользуйтесь своим любимым языком программирования для построении хеш-таблицы, которая использует открытую адресацию с упорядоченным двойным хешированием.
11. Чтобы сравнить различные алгоритмы открытой адресации, начертите гра- фик средней длины пробной последовательности для программ из упраж- нений 4–7, 9 и 10. Сделайте выводы на основе таблицы со 101 записью и 50, 60, 70, 80 и 90 значениями.

# Глава 9 РЕКУРСИЯ

В ходе *рекурсии* метод вызывает сам себя. Если он делает это непосредствен- но, то рекурсия называется *прямой*, если через другой метод — *косвенной*. Она так- же может быть *единичной* (однократный вызов) либо *множественной* (вызов осу- ществляется несколько раз).

На первый взгляд понятие кажется несколько сложным, поскольку человек не мыслит рекурсивно. Например, занимаясь покраской забора, вы, скорее всего, начнете с одного края и будете двигаться к другому. Вряд ли вам придет в голову разделить забор на две части (левую и правую), а затем красить каждую половину.

Тем не менее есть задачи, которые рекурсивны по своей природе, а их структура и решение легко отслеживаются с помощью алгоритма. Таковы, например, про- граммы, выстраивающие деревья и проводящие по ним поиск.

В этой главе вам предстоит познакомиться с некоторыми рекурсивными ал- горитмами. И хотя они полезны уже сами по себе, гораздо важнее понимать за- ложенный в них принцип. Уяснив его, вы начнете замечать рекурсию во многих ситуациях, имеющих отношение к программированию. Однако не всегда это наилучший способ решения задач, в некоторых случаях он снижает произво- дительность программы, в связи с чем в книге будут рассмотрены еще и способы удаления рекурсии.

## Базовые алгоритмы

В следующих подразделах приводятся три рекурсивных алгоритма для расчета факториала, чисел Фибоначчи, а также решения задачи о ханойской башне. Они относительно просты, но раскрывают важные понятия. Как только вы с ними раз- беретесь, сможете перейти к более сложным алгоритмам.

#### Факториал

Факториал числа *N* определяется через функцию factorial, которую можно представить рекурсивно следующими формулами.

0! = 1

*N*! = *N*  (*N* – 1)!

Тогда расчет 3! будет выглядеть так: 3! = 3  2! = 3  2  1! = 3  2  1  0! =

= 3  2  1  1.

Это определение приводит к простому рекурсивному алгоритму.

Integer: Factorial(Integer: n) If (n == 0) Then Return 1 Return n \* Factorial(n - 1)

End Factorial

Если входное значение *n* = 0, то алгоритм возвращает 1, что соответству- ет первому уравнению, которое определяет функцию factorial. В против- ном случае код возвращает число *n* раз с факториалом *n* – 1, согласно второ- му уравнению.

Отсюда вытекают две важные характеристики, которыми должны обладать ре- курсивные алгоритмы.

1. Каждый раз при запуске метод снижает сложность задачи, а затем вызыва- ет себя еще раз, чтобы решить ее упрощенный вариант. В приведенном при- мере расчет *n*! преобразовался в расчет (*n* – 1)!, а затем в умножение на *n*.
2. Рекурсия должна быть конечной. В рассмотренном случае входной параметр *n* уменьшается с каждым вызовом, пока не становится равным 0. В этот момент алгоритм возвращает 1 — и процесс прекращается.

Обратите внимание на недостаток вышеприведенного алгоритма: при вызове метода Factorial с входным параметром –1, рекурсия никогда не остановится. Алгоритм будет выполнять нижеприведенные расчеты.

–1! =

–1  –2! =

–1  –2  –3! =

–1  –2  –3  –4! = ...

Чтобы избежать этого, нужно изменить первый оператор в алгоритме на If (n <= 0) Then Return 1. Тогда при вызове с отрицательным параметром вернется 1.

**ЗАМЕЧАНИЕ**

 С точки зрения разработки программного обеспечения, рассмотренное выше решение считается не самым лучшим, поскольку в программе, вызывающей

алгоритм, скрывается кое-какая проблема. Значение 1 возвращается, даже в том случае если истинный факториал отрицательного числа не определен, что может ввести в заблуждение. Правильнее осуществить проверку в коде вызова и убедиться в том, что число как минимум равно 0, или же вызвать исключение.

В целом анализировать время работы рекурсивных алгоритмов непросто, но не в данном случае. При входном значении *N* алгоритм вызывает сам себя *N* + 1, чтобы выполнить расчет *N*!, (*N* – 1)!, (*N* – 2)!, ..., 0!. Здесь нет ничего сложного, поэтому общая производительность составляет *O*(*N*).

Поскольку алгоритм вызывает сам себя *N* + 1 раз, максимальная глубина ре- курсии также равна *O*(*N*). В некоторых средах программирования она может быть ограничена, а это влечет за собой определенные проблемы.

**ВАЖНОЕ СТЕКОВОЕ ПРОСТРАНСТВО**

Как правило, для работы программы в памяти компьютера выделяются две области памяти: стек и хип. Когда часть кода вызывает метод, информация о вызове находится в стеке (см. главу 5). Как только метод возвращается — информация из стека выталкивается и программа возобновляет свою работу с места, на котором остановилась. Список методов, необходимых для того, чтобы дойти до определенного шага в программе, называется *стеком вызова*.

*Хип* используется для создания переменных и проведения расчетов. Обычно он больше, чем стек, так как в объемный код часто не включаются методы, вызываю- щие другие методы. Тем не менее в алгоритмах иногда может быть очень глубокая рекурсия, что приводит к большому расходу стекового пространства и зависанию программы. В силу этого важно оценивать не только время работы алгоритма и требование к памяти, но и максимальную глубину используемой рекурсии.

Функция factorial возрастает очень быстро, поэтому значение *N* не беспре- дельно. Например, 20! � 2,4  1018, а 21! окажется настолько длинным, что превысит 64 бита — и программа не справится с расчетами. Во избежание проблем глубина рекурсии не должна превосходить 20. Для работы с большими факториалами лучше использовать другие типы данных, например, 64-битное число двойной точности с плавающей запятой, которое справится с 170! � 7,3  10306, или BigInteger в .NET, позволяющий содержать произвольно большие числа. Однако в подобном случае глубокая рекурсия может спровоцировать расход стекового пространства и зависание программы. О том, как решить эту проблему, мы поговорим в подраз- деле «Удаление хвостовой рекурсии» раздела «Удаление рекурсии» текущей главы.

#### Числа Фибоначчи

Числа Фибоначчи определяются приведенными ниже уравнениями. Fibonacci (0) = 0

Fibonacci (1) = 1

Fibonacci (*n*) = Fibonacci (*n* – 1) + Fibonacci (*n* – 2) для *n* > 1

Например, первые 12 чисел Фибоначчи — это 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 и 89.

**ЗАМЕЧАНИЕ**

Иногда Fibonacci (0) и Fibonacci (1) приравнивают к 1. В результате получаются те же значения, как и в вышеприведенном определении, только пропускается 0.

Согласно определению, получаем следующий алгоритм.

Integer: Fibonacci(Integer: n) If (n <= 1) Then Return n

Return Fibonacci(n - 1) + Fibonacci(n - 2); End Fibonacci

Если входное значение *n* равно 0 или 1, на выходе возвращается 0 или 1. (Если оно меньше 1, возвращается само входное значение.) В случае когда *n* больше 1, ал- горитм вызывает сам себя для входных значений *n* – 1 и *n* – 2, суммирует их, а за- тем возвращает результат.

Приведенный код легок для понимания, но работает довольно медленно. Так, для расчета Fibonacci (6) программа может определить Fibonacci (5) и Fibonacci (4), но перед расчетом Fibonacci (5) ей понадобится установить Fibonacci (4) и Fibonacci (3), то есть Fibonacci (4) определяется дважды. При больших *N* рекурсия Fibonacci (*N*) становится достаточно глубокой, а одни и те же значения приходится находить огром- ное количество раз, заставляя программу работать в течение долгого времени.

На рисунке 9.1 показано дерево вызовов алгоритма Фибоначчи для определения Fibonacci (6). Каждая вершина — это и есть вызов, а стоящее в ней число — входной параметр. В корневом узле алгоритм обращается к Fibonacci (5) и Fibonacci (4). Обратите внимание, что внизу дерево заполнено дублированными вызовами: на- пример, Fibonacci (0) рассчитывается пять раз, а Fibonacci (1) — восемь.



6

5

4

4

3

3

2

3

2

2

1

2

1 0

1

2

1 0

1 0

1

0

1

0

1

**Рис. 9.1.** Дерево вызовов алгоритма Фибоначчи

Оценить время работы этого алгоритма сложнее, чем факториального, посколь- ку он является многократно рекурсивным. Предположим, что *T*(*N*) — произво- дительность алгоритма Фибоначчи с входным значением *N*. Если *N* > 1, то код рассчитывает Fibonacci (*N* – 1) и Fibonacci (*N* – 2), затем на дополнительном шаге суммирует их и возвращает результат. Таким образом, *T*(*N*) = *T*(*N* – 1) + *T*(*N* – 2) + 1.

Это немного больше, чем *T*(*N* – 1) + *T*(*N* – 2). Но если вы проигнорируете константу 1 в конце формулы, то получите определение функции Fibonacci, то есть именно от нее зависит итоговое время работы алгоритма. Функция Fibonacci возрастает довольно быстро, хотя и уступает функции factorial. Так, Fibonacci (92) � 7,5  1018, а вот Fibonacci (93) уже не укладывается в длину це- лого числа. Это значит, что максимальная глубина рекурсии равна 92 и для боль- шинства сред программирования не является проблемной. На практике время ра- боты алгоритма Фибоначчи возрастает очень быстро. На моем компьютере расчет Fibonacci (44) занимает свыше 1 мин, а использование бо'льших значений стано- вится непрактичным.

#### Ханойская башня

Эта головоломка уже приводилась в главе 5 (см. рис. 5.7). Напомним, что в ней фи- гурирует стопка дисков, уложенных по размеру диаметра (от большего к меньшему) на один из трех колышков. Ваша цель — переместить стопку на другой колышек, при этом разрешается брать лишь по одному диску за раз и класть меньший на больший.

Рассматривать поставленную задачу как единое целое тяжело, но ее можно упростить и рекурсивным способом решить только часть.

// Перемещаем верхние n дисков с колышка from\_peg на колышек to\_peg,

// используя при необходимости колышек other\_peg.

TowerOfHanoi(Peg: from\_peg, Peg: to\_peg, Peg: other\_peg, Integer: n)

// Рекурсивно перемещаем верхние n – 1 дисков с колышка from\_peg

// на колышек to\_peg.

If (n > 1) Then TowerOfHanoi(from\_peg, other\_peg, to\_peg, n - 1)

<Перемещаем верхний диск с колышка from\_peg на колышек to\_peg.>

// Рекурсивно перемещаем верхние n – 1 дисков с колышка other\_peg

// на колышек to\_peg.

If (n > 1) Then TowerOfHanoi(other\_peg, to\_peg, from\_peg, n - 1) End TowerOfHanoi

Вначале вам придется полностью довериться алгоритму TowerOfHanoi, хотя пока и непонятно, как он работает. На первом этапе верхние *N* – 1 дисков перено- сятся с исходного колышка на тот, который не является целевым. Для этого алго- ритм рекурсивно вызывает сам себя. Но как узнать, что он работает и справится с меньшей задачей?

Ответ прост: при необходимости алгоритм рекурсивно вызывает сам себя, чтобы перемещать все меньшие и меньшие стопки дисков. В определенный момент он переносит только один диск и останавливается. Другими словами, каждый рекур- сивный вызов используется для решения более простой задачи, и в конце концов, она оказывается настолько примитивной, что в вызове уже нет необходимости.

На рисунке 9.2 изображены общие действия, которые требуются для решения головоломки из дисков. В самом начале с первого колышка на третий рекурсивно перемещаются все диски, кроме самого большого. Затем на второй колышек нани- зывается самый большой диск. И наконец, с третьего колышка на второй рекур- сивно перемещаются оставшиеся диски.

**Рис. 9.2.** Общая схема рекурсивного решения головоломки «Ханойская башня»

Вся последовательность шагов для перемещения стопки из трех дисков с пер- вого колышка на второй представлена на рисунке 9.3.

**Рис. 9.3.** Детальная схема рекурсивного решения головоломки «Ханойская башня»

Проанализируем время работы алгоритма. Пусть *T*(*N*) — количество шагов, необходимых для перемещения *N* дисков с одного колышка на другой. Очевид- но, что *T*(1) = 1, ведь для перемещения одного диска нужен один шаг. Для *N* > 0, *T*(*N*) = *T*(*N* – 1) + 1 + *T*(*N* – 1) = 2*T*(*N* – 1) + 1. Если проигнорировать константу 1, то *T*(*N*) = 2*T*(*N* – 1), получается, что функция обладает экспоненциальным вре- менем работы *O*(2*N*).

Еще один способ провести анализ — создать таблицу наподобие таблицы 9.1 и рассчитать количество шагов для разных значений *N*. Здесь для каждого *N* > 1 использовалась формула *T*(*N*) = 2*T*(*N* – 1) + 1. Изучив результаты, вы увидите, что *T*(*N*) = 2*N* – 1.

**Òàáëèöà 9.1.** Âðåìÿ ðàáîòû àëãîðèòìà

äëÿ ðåøåíèÿ ãîëîâîëîìêè «Õàíîéñêàÿ áàøíÿ»

|  |  |
| --- | --- |
| ***N*** | ***T*(*N*)** |
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 4 | 15 |
| 5 | 31 |
| 6 | 63 |
| 7 | 127 |
| 8 | 255 |
| 9 | 511 |
| 10 | 1 023 |

Как и в случае с алгоритмом Фибоначчи, максимальная глубина рекурсии при входном значении *N* равна *N*. С увеличением входного параметра время работы про- граммы будет возрастать очень быстро, другими словами, применять алгоритм для решения головоломки станет нецелесообразно еще до того, как будет достигнута максимальная глубина рекурсии.

## Графические алгоритмы

Некоторые алгоритмы используют рекурсию, чтобы создавать сложные графи- ческие изображения. И хотя код у них очень компактный, на самом деле они ничуть не проще тех базовых алгоритмов, что описаны в предыдущем разделе.

#### Кривые Коха

Кривая Коха — хороший пример *самоподобного фрактала*, где часть кривой напоминает ее общую форму. Построение таких фракталов начинается с *инициа- тора* — участка, определяющего основные очертания будущей фигуры. В ходе рекурсии все инициаторы или некоторые из них преобразуются (правильно мас- штабируются, поворачиваются и т. д.), превращаясь в своего рода *генераторы*, которые затем замещаются новыми версиями генераторов.

Простейшая кривая Коха в качестве инициатора использует линию, которая на каждом уровне рекурсии разбивается на четыре части. Из них формируются три сегмента длиной в 1/3 от исходного отрезка. Первая и четвертая части нахо- дятся в направлении базовой линии, вторая поворачивается на –60°, третья — на 120° (рис. 9.4). На следующем уровне рекурсии программа заменяет каждый сег- мент в генераторе новой копией генератора. Взглянув на рисунок 9.5, вы поймете, почему кривая называется самоподобной: ее часть выглядит как уменьшенная ко- пия целой фигуры.

Пусть pt1, pt2, pt3, pt4 и pt5 — точки, связанные с сегментами в гене- раторе на рисунке 9.4. Начертим кривую Коха с помощью приведенного ниже псевдокода.

// Чертим кривую Коха заданной сложности, начиная с точки p1

// и прокладывая расстояние length в направлении angle. DrawKoch(Integer: depth, Point: pt1, Float: angle, Float: length)

If (depth == 0) Then

<Чертим сегмент.>

Else

End If

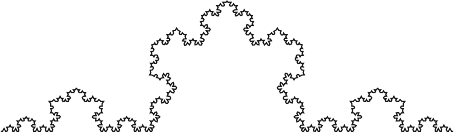
<Находим точки pt2, pt3, and pt4.>

// Рекурсивно чертим части кривой. DrawKoch(depth - 1, pt1, angle, length / 3);

DrawKoch(depth - 1, pt2, angle - 60, length / 3); DrawKoch(depth - 1, pt3, angle + 60, length / 3); DrawKoch(depth - 1, pt4, angle, length / 3);

End DrawKoch

**Рис. 9.4.** Кривая Коха: инициатор *(слева)* и генератор *(справа)*

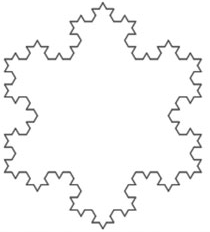


**Рис. 9.5.** Кривая Коха с уровнями рекурсии от 0 до 5

Если depth = 0, то алгоритм чертит сегмент от точки p1 длиной length, следуя в направлении angle. (То как осуществляется рисование, зависит от ис- пользуемой вами среды программирования.) При depth > 0 алгоритм определяет точки pt2, pt3 и pt4, а затем выстраивает четыре сегмента длиною 1/3 от ис- ходного отрезка: вначале он следует из точки pt1 в направлении angle до точки pt2, потом разворачивается на 60° влево и идет до точки pt3, совершает еще один поворот на 120° вправо (на угол, превышающий исходный на 60°) и двигается до точки pt4, и наконец, из нее, придерживаясь начального угла, преодолевает по- следнюю часть пути.

Если глубина больше 0, то алгоритм рекурсивно вызывает себя четыре раза. Предположим, что *T*(*N*) — количество шагов, которые он совершает для глубины n, тогда *T*(*N*) = 4  *T*(*N* – 1) + *C*, где *C* — константа. Если проигнорировать послед- нюю, то *T*(*N*) = 4  *T*(*N* – 1), отсюда время работы — *O*(4*N*). Максимальная глубина рекурсии, необходимая для построения кривой Коха сложностью *N*, определяется только *N* и не должна вызывать проблем, ведь, как и в предыдущих алгоритмах (для нахождения чисел Фибоначчи и решения головоломки «Ханойская башня»), время работы возрастает невероятно быстро.

Если соединить края трех кривых Коха таким образом, чтобы их инициаторы образовывали треугольник, то получится так называемая снежинка Коха (рис. 9.6).



**Рис. 9.6.** Снежинка Коха третьего уровня

#### Кривая Гильберта

Как и в случае с кривой Коха, построение кривой Гильберта начинается с про- стого инициатора. Чтобы перейти к более глубокому уровню рекурсии, алгоритм разбивает инициатор на части и, повернув соответствующим образом уменьшенную копию целой кривой, выстраивает конечную фигуру.

На рисунке 9.7 показаны кривые Гильберта уровней 0, 1 и 2. На последних двух фрагментах соединяющие линии выделены серым цветом, чтобы вы понимали, как образуются фигуры высокого порядка.

**Рис. 9.7.** Кривые Гильберта разного уровня

Следующий псевдокод выстраивает кривую Гильберта.

// Чертим кривую Гильберта, изначально двигаясь в направлении <dx, dy>. Hilbert(Integer: depth, Float: dx, Float: dy)

If (depth > 0) Then Hilbert(depth - 1, dy, dx) DrawRelative(dx, dy)

If (depth > 0) Then Hilbert(depth - 1, dx, dy) DrawRelative(dy, dx)

If (depth > 0) Then Hilbert(depth - 1, dx, dy) DrawRelative(-dx, -dy)

If (depth > 0) Then Hilbert(depth - 1, -dy, -dx) End Hilbert

Данный алгоритм предполагает, что местоположение рисунка уже опреде- лено. Метод DrawRelative прокладывает линию из текущей позиции в новую и обновляет координаты. Так, если текущая позиция (10, 20), то оператор DrawRelative(0, 10) начертит сегмент (10, 20) - (10, 30) и обозначит

текущую позицию как (10, 30).

Если глубина рекурсии больше 0, то алгоритм рекурсивно вызывает сам себя, чтобы начертить кривую самого низкого порядка с параметрами dx и dy, пере- ключаемыми таким образом, чтобы меньшая кривая поворачивалась на 90°. Вер- нувшись к рисунку 9.7, вы увидите, что кривая уровня 0 поворачивается в начале кривой уровня 1. На следующем этапе программа рисует сегмент, связывающий первую кривую низкого уровня со следующей, а после вызывает сама себя, чтобы начертить следующую часть кривой. Теперь алгоритм сохраняет dx и dy в их ис- ходных позициях, чтобы вторая часть кривой не поворачивалась. В завершение выстраивается следующий сегмент соединяющей линии и происходит послений рекурсивный вызов. В этот раз dx меняется на -dy, а dy — на -dx, чтобы меньшая кривая поворачивалась на –90°.

#### Кривая Серпинского

Кривая Серпинского1 также воспроизводит саму себя на более высоком уровне из кривых более низкого уровня. Однако в отличие от предыдущих примеров в по- строении простейшего фрагмента используются четыре косвенные рекурсивные функции, вызывающие самих себя. На рисунке 9.8 показаны кривые Серпинского уровней 0, 1, 2 и 3, серым цветом выделены соединяющие сегменты.

Кривая Серпинского уровня 1 (рис. 9.9) состоит из четырех частей, отрисован- ных четырьмя разными функциями. Эти функции чертят кривые, перемещая теку- щую позицию по направлениям вправо, вниз, влево и вверх. Например, функция Right формирует сегменты, для которых текущая позиция смещается вправо. На рисунке 9.9 серым цветом выделены соединяющие сегменты.

Чтобы начертить более сложную кривую, алгоритм разбивает ее часть на низ- коуровневые фрагменты. На рисунке 9.10, например, показано, как создать часть уровня 2 из четырех частей уровня 1. Проанализировав рисунок 9.8, вы поймете, как образуется фигура в целом.

1 Польский математик Вацлав Франциск Серпинский (1882–1969) изучал многие виды фракталов, поэтому его имя вы встретите в книге еще не раз.

Уровень 0 Уровень 1

Уровень 2 Уровень 3

**Рис. 9.8.** Кривая Серпинского разных уровней

**Рис. 9.9.** Кривая Серпинского уровня 1

Вправо

Вниз

Вверх

Вправо Вправо

**Рис. 9.10.** Правая часть кривой Серпинского уровня 2

Вниз

Вверх

В следующем псевдокоде представлен основной алгоритм.

// Чертим кривую Серпинского. Sierpinski(Integer: depth, Float: dx, Float: dy)

SierpRight(depth, dx, dy) DrawRelative(dx, dy) SierpDown(depth, gr, dx, dy) DrawRelative(-dx, dy) SierpLeft(depth, dx, dy) DrawRelative(-dx, -dy) SierpUp(depth, dx, dy) DrawRelative(dx, -dy)

End Sierpinski

Для формирования части кривой алгоритм вызывает методы SierpRight, SierpDown, SierpLeft и SierpUp. Между ними он ссылается на еще один метод DrawRelative, чтобы начертить соединяющие сегменты. Как и в случае с кри- вой Гильберта, соединяющая линия прокладывается из текущей позиции в новую точку, затем текущая позиция обновляется. Вызовы DrawRelative — единствен- ный шаг в алгоритме, где на самом деле выполняется рисование.

В следующем псевдокоде представлен алгоритм SierpRight.

// Чертим часть кривой, идущую вправо. SierpRight(Integer: depth, Float: dx, Float: dy)

If (depth > 0) Then

depth = depth – 1

SierpRight(depth, gr, dx, dy) DrawRelative(gr, dx, dy) SierpDown(depth, gr, dx, dy) DrawRelative(gr, 2 \* dx, 0) SierpUp(depth, gr, dx, dy) DrawRelative(gr, dx, -dy) SierpRight(depth, gr, dx, dy)

End If

End SierpRight

Наглядно работа данного метода представлена на рисунке 9.10. Сначала про- исходит вызов SierpRight, чтобы начертить часть кривой с меньшей глубиной рекурсии, идущую вправо. Затем формируется сегмент внизу справа, чтобы соеди- ниться со следующей частью кривой. Далее вызывается метод SierpDown, чтобы образовалась часть кривой с меньшей глубиной рекурсии, идущая вниз, и прокла- дывается соединительный сегмент вправо для следующей части с меньшей глуби- ной рекурсии. Та в свою очередь формируется методом SierpUp и идет вверх. Да- лее появляется соединительный сегмент, направленный вверх и вправо. В заверше- ние вызывается метод SierpRight для создания конечной части кривой с мень- шей глубиной рекурсии, идущей вправо.

Другие методы построения частей кривой Серпинского рассматриваются в упражнениях.

Поскольку методы Серпинского вызывают друг друга много раз, они являют- ся многократно и косвенно рекурсивными. Нерекурсивный способ решения зада- чи сложнее.

**ПРИБЛИЗИТЕЛЬНАЯ МАРШРУТИЗАЦИЯ**

Кривые заполнения пространства — Гильберта и Серпинского — предполагают использование простого метода приблизительной маршрутизации. Предпо- ложим, вам нужно посетить несколько мест в городе. Если вы начертите кривую Гильберта или Серпинского на карте, то сможете посетить нужные пункты в том порядке, в котором через них проходит кривая. (Не нужно прокладывать маршрут в точности по кривой — она используется лишь для генерирования упорядоченности). Скорее всего, результат будет не оптимальным, но и не самым плохим. Его можно использовать в качестве отправной точки для задаче про коммивояжера, описанной в главе 17.

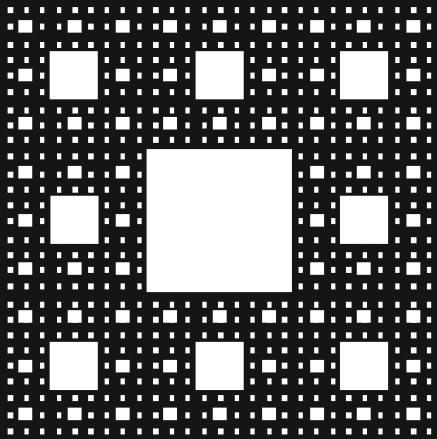
#### Салфетки

*Салфетка* — разновидность самоподобного фрактала. Ее построение начина- ют с такой геометрической фигуры, как треугольник или квадрат. Если заданная глубина рекурсии равна 0, создается примитивная форма. При глубине рекурсии больше 0, метод делит начальную форму на меньшие подобия и рекурсивно вы- зывает сам себя, чтобы начертить некоторые из них (но не все).

На рисунке 9.11 представлены треугольные салфетки уровня 0, 1, 2 и 3, часто называемые салфетками (иногда ситечками или треугольниками) Серпинского. Они создаются следующим образом: исходный треугольник делится на четыре ча- сти, три из которых, находящиеся в углах, рекурсивно окрашиваются.

***Рис. 9.11.*** Салфетки Серпинского разного уровня

На рисунке 9.12 изображена квадратная салфетка, или ковер Серпинского. Что- бы ее получить, нужно разделить квадрат на девять частей, удалить центральную часть, а затем рекурсивно окрасить остальные.



**Рис. 9.12.** Ковер Серпинского

Написать псевдокоды низкого уровня для создания салфетки и ковра Серпин- ского предлагается в качестве упражнений.

## Алгоритмы с возвратом

Рекурсия используется также для поиска лучшего решения сложных задач. В алгоритмах с возвратом создаются тестовые варианты, которые просчитывают, возможность справится с поставленной задачей. Если программа приходит к выво- ду, что предлагаемый способ не приведет к нужному результату, данный тестовый вариант отбрасывается и вверху по стеку вызовов ищется другой.

Такой метод очень удобен: он позволяет строить частичное решение и опреде- лять, приведет ли оно к полному. Благодаря такому подходу вы можете прекратить дальнейшее усовершенствование пробного варианта и вернуться на несколько шагов назад, чтобы продолжить работу оттуда.

В следующем высокоуровневом псевдокоде представлен общий способ возврата.

// Исследуем тестовое решение.

// Возвращаем false, если его нельзя развить до полного решения.

// Возвращаем true, если рекурсивный вызов LeadsToSolution

// приводит к полному решению.

Boolean: LeadsToSolution(Solution: test\_solution)

// Если можно сделать вывод, что данное частичное решение

// не приводит к полному, возвращаем false.

If <test\_solution не может решить проблему> Then Return false

// Если это полное решение, возвращаем true.

If <test\_solution — полное решение> Then Return true

// Расширяем частичное решение.

Loop <Проходим по всем возможным расширениям в test\_solution.>

<Расширяем test\_solution.>

// Рекурсивно проверяем, ведет ли это к решению.

If (LeadsToSolution(test\_solution)) Then Return true

// Это расширение не ведет к решению. Отменяем расширение.

<Отменяем расширение.> End Loop

// Если мы дошли до этой строки, данное частичное решение

// не приводит к полному. Return false

End LeadsToSolution

Алгоритм LeadsToSolution берет в качестве параметра любые необходимые данные, чтобы проработать частичное решение, и возвращает true, если оно при- водит к полному.

Первым делом осуществляется проверка — имеет ли частичное решение право на существование. Если оно не приводит к полному решению, алгоритм возвращает false и вызывает метод LeadsToSolution, который завершает текущий тест и приступает к новому. Если предлагаемое решение допустимо, алгоритм циклически проходит по всем возможным его расширениям до получения конечного результата. Для каждого расширения алгоритм рекурсивно ссылается на самого себя, чтобы определить, будет ли оно работать. Если рекурсивный вызов возвращает false, расширение не годится: оно отменяется и осуществляется следующая попытка с новым расширением. Если ал- горитм перепробовал все возможные расширения и не нашел среди них целесообраз- ного, он возвращает false, чтобы вызов LeadsToSolution закрыл тестовое решение.

Поиск нужного варианта можно представить в виде продвижения по дереву, где каждая ветвь соответствует конкретному решению, которое пытается справится с за- дачей. Например, в дереве для шахматной партии ветви будут представлять собой возможные ходы на определенном этапе игры. Относительно быстрые тесты помо- гают понять, приведет ли частичное решение к полному и, если нет, отрубить соот- ветствующую ветвь дерева, не проводя по ней исчерпывающий поиск. Это позволяет сэкономить время. (Более подробно о дереве принятия решений читайте в главе 12.)

В следующих подразделах описываются две задачи, использующие алгоритмы с возвратом: задача о восьми ферзях и о ходе коня. Их изучение поможет вам разо- браться в представленном общем решении.

#### Задача о восьми ферзях

Цель задачи — расположить на шахматной доске восемь ферзей так, чтобы они не могли атаковать друг друга (рис. 9.13). Другими словами, два ферзя не могут находиться в одном горизонтальном, вертикальном или диагональном ряду. Если попытаться разместить 8 ферзей во всех возможных вариантах на 64 клетках шах- матной доски, то придется перепробовать 4 426 165 368 комбинаций. Это займет невероятно много времени.

**ПОДСЧЕТ КОМБИНАЦИЙ**

Огромное количество возможных вариантов решения объясняется тем, что каждый из 8 ферзей может занять любой из 64 квадратов доски. Фигуры равнозначны, поэтому не имеет значения, куда и какую из них вы поставите. Таким образом, можно считать, что из 64 квадратов шахматного поля, вам нужно выбрать любые 8.

.

Количество вариантов, при котором из множества *n* элементов выбирается *k* элементов без повторений, передается с помощью биномиального коэффи- циента (*n/k*) (читается как «из *n* по *k*») и рассчитывается по формуле:

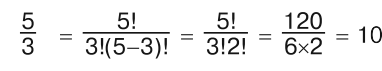


( )

( )

.

Приведенные ниже расчеты отражают количество возможных оригинальных выборок в случае, когда из пяти допустимых элементов берутся три:



( )

Если в множестве из *n* элементов один и тот же элемент может присутствовать

несколько раз, количество допустимых комбинаций определяется по формуле.

.



(

)

Например, для множества из пяти элементов, содержащего дубликаты, коли- чество допустимых комбинаций из трех элементов будет рассчитываться так:



(

) ( )

.



( )

В задаче о восьми ферзях нужно выбрать восемь разных клеток (более одного ферзя в одну клетку поставить нельзя), поэтому количество возможных вариантов составит:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

**Рис. 9.13.** Решение задачи о восьми ферзях

При решении данной задачи алгоритм с возвратом позволяет не рассматривать некоторые комбинации. Для начала, например, можно разместить одного ферзя в верхнем левом углу доски и попробовать добавить другого справа. Но поскольку фигуры запрещается располагать в одном ряду, вы можете сразу исключить любое решение, при котором первые два ферзя находятся рядом в верхнем левом углу. Программа вернется к тому моменту, перед которым она добавила второго ферзя, и сосредоточится на поиске потенциально успешных решений.

Если вам кажется, что от этого мало толку, представьте, сколько комбинаций могло включать неверное расположении двух таких ферзей: (62/6) = 61 474 519. Именно столько решений вы исключите. Таким образом, всего один шаг возвра- та сэкономит усилия, необходимые для проверки более 61 млн бесполезных воз- можностей.

По сути, если первая фигура займет верхний левый угол, уже ни одна другая не сможет стать в том же ряду по вертикали, горизонтали или диагонали. Это значит, что непригодной становится 21 клетка шахматной доски. Исключив все связанные с ними частичные решения, вы отбросите почти 1,3 млрд бесперспективных вари- антов. В ходе последующих проверок возникнут и другие лишенные смысла ком- бинации. Например, после установки второго ферзя на одной из допустимых кле- ток доски размещение третьего получит свои ограничения.

В следующем псевдокоде показано, как использовать возврат для решения за- дачи о восьми ферзях.

Boolean: EightQueens(Boolean: spot\_taken[,], Integer: num\_queens\_positioned)

// Проверяем, допустимо ли тестовое решение. If (Not IsLegal(spot\_taken)) Then Return false

// Проверяем, все ли ферзи размещены.

If (num\_queens\_positioned == 8) Then Return true

// Расширяем частичное решение.

// Проверяем все позиции для следующего ферзя. For row = 0 to 7

For col = 0 to 7

// Проверяем, не занята ли эта позиция. If (Not spot\_taken[row, col]) Then

// Помещаем сюда ферзя. spot\_taken[row, col] = true

// Рекурсивно проверяем, ведет ли это

// к решению.

If EightQueens(spot\_taken, num\_queens\_positioned + 1))

Then Return true

// Это расширение не ведет к решению.

// Отменяем расширение. spot\_taken[row, col] = false

End If Next col

Next row

// Если мы дошли до этой строки, мы не смогли найти

// подходящее решение. Return false

End EightQueens

Данный алгоритм в качестве параметра берет двумерный массив булевых зна- чений spot\_taken. Запись spot\_taken[row, col], имеющая значение true, говорит о том, что в ряду row и столбце col присутствует ферзь. Второй параметр алгоритма num\_queens\_positioned определяет, сколько ферзей фигурирует в тестовом решении.

В самом начале работы алгоритм вызывает метод IsLegal для проверки до- пустимости тестового решения. Этот метод не показан, но суть его заключается в циклическом прохождении через массив spot\_taken и поиске двух ферзей в одном ряду, столбце или диагонали. Затем алгоритм сравнивает num\_queens\_ positioned с общим количеством ферзей (напомним, что их восемь). Если все фигуры задействованы, то тестовое решение является полным и алгоритм возвра- щает true. (После этого массив spot\_taken не меняется, поэтому при возвра- щении первого вызова EightQueens в нем будет находится ответ.)

В неполном решении алгоритм циклом проходит по всем рядам и столбцам и для каждой пары ряд — столбец проверяет spot\_taken, чтобы определить, не занято ли это место. Если оно свободно, то алгоритм размещает в нем следующего ферзя и рекур- сивно вызывает себя, чтобы определить, приводит ли расширенное решение к полному.

Когда рекурсивный вызов возвращает true, это значит, что он нашел полное решение и данный вызов также возвращает true. Если же рекурсивный вызов воз- вращает false, то расширенное решение не приводит к полному, алгоритм удаля- ет ферзя из его текущей позиции и пробует найти следующую. В случае когда пе- репробованы все возможные положения для следующего ферзя и ни одно из них не подходит, тестовое решение еще до добавления новой фигуры приходит к вы- воду, что желаемый ответ получить невозможно, и алгоритм возвращает false.

Вы можете усовершенствовать производительность данного алгоритма несколь- кими способами, о них говорится в упражнениях 13 и 14 в конце данной главы.

#### Ход коня

Цель задачи — заставить коня ступить на каждую клетку шахматной доски всего один раз. Ход считается завершенным, если в конечной позиции фигура находится в одном шаге от начальной позиции, — тогда конь может немедленно двигаться дальше. Незавершенный ход считается открытым.

**ЗАМЕЧАНИЕ**

 За один ход конь преодолевает две клетки по горизонтали или вертикали и еще одну перпендикулярно текущей позиции (рис. 9.14).



**Рис. 9.14.** Возможные восемь ходов коня без выхода за пределы шахматной доски

Общее решение задачи представлено в следующем высокоуровневом алго- ритме.

// Перемещаем коня в позицию [row, col].

// Затем рекурсивно пробуем сделать следующие ходы.

// Возвращаем true, если найдено подходящее решение. Boolean: KnightsTour(Integer: row, Integer: col,

Integer: move\_number[,], Integer: num\_moves\_taken)

// Перемещаем коня в эту позицию. num\_moves\_taken = num\_moves\_taken + 1 move\_number[row, col] = num\_moves\_taken

// Проверяем, сделали ли мы все требующиеся ходы. If (num\_moves\_taken == 64) Then Return true

// Создаем массив, чтобы определить, допустимы ли ходы

// в данной позиции.

Integer: dRows[] = { -2, -2, -1, 1, 2, 2, 1, -1 }

Integer: dCols[] = { -1, 1, 2, 2, 1, -1, -2, -2 }

// Пробуем все допустимые позиции для следующего хода. For i = 0 To 7

Integer: r = row + d\_rows[i] Integer: c = col + d\_cols[i]

If ((r >= 0) And (r < NumRows) And

(c >= 0) And (c < NumCols) And (move\_number[r, c] == 0))

Next i

Then

End If

// Данный ход допустим и возможен.

// Делаем его, а затем рекурсивно пробуем другие.

If (KnightsTour

(r, c, move\_number, num\_moves\_taken)) Then Return true

// Этот ход не работает. Отменяем его. move\_number[row, col] = 0

// Если мы дошли до этой строки, подходящее решение не найдено. return false

End KnightsTour

В качестве входных параметров алгоритм берет ряд и столбец, по которым конь продолжает движение, массив move\_number, который нумерует движение коня по каждой клетке, также количество уже сделанных ходов.

В самом начале программа записывает хода коня на текущую клетку и увеличи- вает количество сделанных шагов. Если последнее равно 64, то фигура завершила перемещение по доске и алгоритм возвращает true (указание на успешное выпол- нение задачи). Если ход не окончен, алгоритм инициализирует два массива, что- бы показать шаги, которые можно выполнить из текущей клетки. Например, пер- вые записи в массивах — это –2 и –1. Они указывают на то, что конь может пере- меститься из клетки (row, col) на (row – 2, col – 1), если таковая присутству- ет на шахматной доске.

Далее алгоритм проверяет в цикле все возможные ходы из позиции (row, col). Если ход допустим и не был совершен в тестовом решении, алгоритм передвигает фигуру и рекурсивно вызывает себя сам, чтобы проверить, приводит ли это к пол- ному решению. Если ни один из возможных ходов в текущей позиции не приво- дит к решению, алгоритм устанавливает move\_number[row, col] равным 0, что- бы отменить текущий шаг, и возвращает false (указание на то, что перемещение коня в клетку (row, col) бесполезно).

К сожалению, в задаче о ходе коня найти ограничения нелегко. Расставляя на доске восемь ферзей, мы довольно просто определяли, в каких позициях одна фи- гура подвергнется атаке другой, и не рассматривали их. Для коня любая клетка, на которую он может перейти и на которой он еще не был, дает новое тестовое реше- ние. В некоторых случаях определить, что тестовое решение не сработает, не со- ставляет труда. Например, если на доске есть клетка, на которой конь еще не стоял, и она находится на расстоянии одного шага от любой другой не пройденной клетки. Однако, распознать такую ситуацию сложно.

В силу этой причины на раннем этапе алгоритм часто подолгу следует за тесто- вым решением до того, как узнает, что оно невыполнимо. Поскольку конь может совершить до восьми ходов, количество потенциальных передвижений составляет 864, или в грубом округлении 6,3  1057. Изучив позиции фигуры на доске, вы точнее определите данное число (например, в углу у коня остается лишь два потенциаль- ных хода), но в любом случае оно будет огромно.

Все вышесказанное означает, что решить задачу о ходе коня на обычной шах- матной доске размером 88 клеток чрезвычайно сложно. В ходе тестов програм- ма справилась с тем же заданием на доске размерностью 66 почти сразу, а на до- ске 76 за 2 с. Зато для доски 77 решения не нашлось даже за час.

Справиться с задачей поможет *эвристический алгоритм*. Он часто дает хороший результат, хотя и не гарантирует, что тот будет наилучшим из возможных. Напри- мер, эвристический алгоритм может на 10 % увеличить предполагаемое время по- ездки, чтобы учесть задержки транспорта. Это не всегда гарантирует своевременное прибытие в пункт назначения, но повышает шансы попасть туда без опоздания.

В 1823 г. эвристический алгоритм для задачи о ходе коня предложил Варн- сдорф. Согласно его теории, следующий возможный шаг подбирался таким обра- зом, чтобы общее количество шагов получалось наименьшим. Предположим, конь находится в позиции, из которой допускается сделать всего два хода. Выбрав пер- вый, вы получите еще пять возможных, выбрав второй — всего один. В таком слу- чае эвристический алгоритм отдаст предпочтение второму ходу.

Предложенное Варнсдорфом решение является настолько эффективным, что находит полный ход без возвратов даже для досок размером 7575 клеток. (В моем тесте программа нашла решение на доске размером 5757 почти мгновенно, а за- тем зависла из-за переполнения стека на доске размером 5858.)

#### Сочетания и размещения

*Сочетание* — набор элементов, выбранных из множества. Например, для мно- жества {*A*, *B*, *C*} сочетаниями будут являться {*A*, *B*}, {*A*, *C*} и {*B*, *C*}. Причем в одном и том же множестве порядок элементов не имеет значения, то есть {*A*, *B*} считается

равным {*B*, *A*}. Можно провести аналогию с заказом в кафе. Неважно, попросите ли вы бутерброд и кофе или же кофе и бутерброд.

*Размещение* — тот же набор элементов множества, но только упорядоченный. В том же множестве {*A, B, C*} можно выделить следующие размещения из двух эле- ментов: (*A, B*), (*A, C*), (*B, A*), (*B, C*), (*C, A*) и (*C, B*). (Обратите внимание, что соче- тание обозначается фигурными скобками, а размещение круглыми.)

Группы элементов в выборке зависят еще и от того, возможны ли в ней повто- рения. Например, для множества {*A, B, C*} к уже приведенным выше сочетаниям добавятся {*A, A*}, {*B, B*} и {*C, C*}.

Особым случаем размещений являются перестановки. В них используются все элементы множества и не допускаются повторения. Так, в случае множества {*A, B, C*} перестановками будут (*A, B, C*), (*A, C, B*), (*B, A, C*), (*B, C, A*), (*C, A, B*) и (*C, B, A*).

Многие представляют себе размещения в виде множества, которое должно стать набором перестановок, а не в виде более общего случая, когда нельзя брать все элементы множества, но разрешены повторения.

В следующих подразделах описаны алгоритмы, которые используются для генерирования сочетаний и размещений с повторениями и без них.

#### Сочетания с циклами

Если вам нужно выбрать из множества определенное количество элементов, то при написании программы проще всего воспользоваться циклами For. Приве- денный ниже псевдокод формирует все возможные сочетания из трех элементов с повторениями на основе множества из пяти элементов.

// Генерируем сочетания по три элемента с повторениями. List<string>: Select3WithDuplicates(List<string> items)

List<string>: results = New List<string>

For i = 0 To <наибольший индекс в массиве items>

For j = i To <наибольший индекс в массиве items>

For k = j To <наибольший индекс в массиве items> results.Add(items[i] + items[j] + items[k])

Next i

Next j

Next k

Return results

End Select3WithDuplicates

Алгоритм берет в качестве параметра список string и использует три цик- ла For для выбора трех букв, входящих в формируемое сочетание. Причем в каж- дом новом цикле начальное значение счетчика приравнивается к текущему значе- нию из предыдущего цикла. Так, второй цикл начинается с *j*, равного *i*. Это зна- чит, что в формируемом сочетании вторая буква будет отличаться от тех букв, ко- торые предшествуют первой в самом множестве. Например, если для множества

{*A, B, C, D*, *E*} первой буквой в сочетании является *C*, то в качестве второй нельзя выбрать ни *A*, ни *B*. Благодаря такому подходу сохраняется алфавитный порядок и предотвращается образование сочетаний по типу {*A, B, C*} или {*A, C, B*}, которые, по сути, одинаковы. В третьем, внутреннем, цикле алгоритм объединяет все вы- бранные элементы в единое выходное значение.

Чтобы запретить повторения, в каждом новом цикле используйте для счетчи- ка текущее значение предыдущего цикла, увеличенное на 1. В результате внутрен- ний цикл не сможет выбрать тот же элемент, что и внешний, — и вы добьетесь же- лаемого эффекта.

// Генерируем сочетания по три элемента без повторений. List<string>: Select3WithoutDuplicates(List<string> items)

List<string>: results = new List<string>()

For i = 0 To <наибольший индекс в массиве items>

For j = i + 1 To <наибольший индекс в массиве items>

For k = j + 1 To <наибольший индекс в массиве items> results.Add(items[i] + items[j] + items[k])

Next i

Next j

Next k

Return results

End Select3WithoutDuplicates

О том, как рассчитывать количество возможных сочетаний для данного мно- жества и количество элементов в сочетании, рассказывалось во врезке «Подсчет комбинаций» из подраздела «Задача о восьми ферзях» раздела «Алгоритмы с воз- вратом» текущей главы.

#### Сочетания с повторениями

Алгоритмы, описанные в предыдущем подразделе предполагают, что вы знаете, сколько элементов должно быть в выборке (общее количество элементов в исходном множестве программа может подсчитать самостоятельно). Но что если размер выборки вам неизвестен и невозможно запрограммировать нужное количество циклов For?

Вы можете решить проблему рекурсивно, добавляя при каждом вызове один элемент к сочетанию. Если выборка будет неполной, алгоритм сошлется сам на себя, чтобы добавить еще одно значение. Собрав нужное количество элементов, программа что-нибудь с ними сделает, например распечатает полученный список.

В следующем псевдокоде представлена генерация сочетаний с повторениями:

// Генерируем сочетания с повторениями. SelectKofNwithDuplicates(Integer: index, Integer: selections[],

Data: items[], List<List<Data>> results)

// Проверяем, был ли это последний вызов. If (index == <длина сочетания>) Then

// Добавляем результат к списку элементов. List<Data> result = New List<Data>()

For i = 0 To <наибольший индекс в сочетании> result.Add(items[selections[i]])

Else

Next i results.Add(result)

// Получаем наименьшее значение,

// которое мы можем использовать для следующего выбора. Integer: start = 0

// Используем данное значение, если это первый индекс. If (index > 0) Then start = selections[index - 1]

// Выполняем следующий вызов.

For i = start To <наибольший индекс в массиве items>

// Добавляем i к сочетанию. selections[index] = i

End If

Next i

// Рекурсивно выполняем другие вызовы.

SelectKofNwithDuplicates

(index + 1, selections, items, results)

End SelectKofNwithDuplicates

Приведенный алгоритм использует следующие параметры:

* index — индекс элемента в сочетании, который устанавливает рекурсив- ный вызов алгоритма. Если index равен 2, то вызов алгоритма заполняет selections[2];
* selections — массив, в котором хранятся индексы элементов из сочета- ния. Так, если в selections содержатся две записи — 2 и 3, то в сочетание включены элементы с индексами 2 и 3;
* items — массив элементов, из которых формируется сочетание;
* results — список комбинаций элементов, представляющий полные соче- тания. Например, для сочетания {*A*, *B*, *D*}, results будет хранить список, включающий индексы *A*, *B* и *D*.

В самом начале алгоритм проверяет индекс элемента в формируемом соче- тании. Если значение больше длины массива selections, то сочетание полное и алгоритм добавляет его к списку results. В противном случае сочетание не- полное, и программа определяет наименьший индекс в массиве items, который она может использовать для следующего выбора. Если данный вызов алгоритма заполняет первую позицию в массиве selections, он вправе взять любое зна- чение в массиве items, другими словами, start установлен на 0. Если элемент в сочетании не первый, то алгоритм приравнивает start к индексу последнего выбранного значения.

Предположим, у нас есть множество {*A*, *B*, *C*, *D*, *E*} и алгоритм выбирает тре- тий элемент сочетания. При этом первые два имели индексы 0 и 2, стало быть, те- кущую выборку можно представить как {*A*, *C*}. В данном случае алгоритм прирав- нивает start к 3 и рассматривает в качестве возможного элемента тот, у которо- го индекс больше либо равен 3. Согласно представленному множеству таковыми будут элементы *D* и *E.*

Используемая установка start сохраняет порядок элементов в сочетании (в рассмотренном выше примере он соответствует алфавиту) и предотвращает ге- нерацию сочетаний из одинаковых элементов с разным расположением (напри- мер, {*A, C, D*} и {*A, D, C*}).

Приняв start за начальное значение счетчика, алгоритм циклически прохо- дит до последнего индекса в массиве items и размещает каждое его значение в мас- сиве selections, чтобы добавить соответствующий элемент к выборке. Затем он рекурсивно назначает величины другим записям в массиве selections.

#### Сочетания без повторений

Чтобы получить сочетания без повторений, в предыдущий алгоритм необходимо внести одно небольшое изменение: не просто приравнять переменную start к по- следнему индексу, добавленному в массив selections, но и увеличить ее на 1. В приведенном ниже псевдокоде преобразованная строка выделена полужирным шрифтом.

// Генерируем сочетания без повторений. SelectKofNwithoutDuplicates(Integer: index, Integer: selections[],

Data: items[], List<List<Data>> results)

// Проверяем, был ли это последний вызов. If (index == <длина сочетания>) Then

// Добавляем результат к списку элементов. List<Data> result = New List<Data>()

For i = 0 To <наибольший индекс в сочетании> Result.Add(items[selections[i]])

Else

Next i results.Add(result)

// Получаем наименьшее значение, которое мы можем

// использовать для следующего выбора. Integer: start = 0

// Используем данное значение, если это первый индекс.

**If (index > 0) Then start = selections[index - 1] + 1**

// Выполняем следующий вызов.

For i = start To <наибольший индекс в массиве items>

// Добавляем i к выборке. selections[index] = i

End If

Next i

// Рекурсивно выполняем другие вызовы. SelectKofNwithoutDuplicates(

index + 1, selections, items, results)

End SelectKofNwithoutDuplicates

Алгоритм работает, как и раньше, только в этот раз элемент выбирается из тех, что идут за последним, включенным в сочетание. Например, если из множества {*A*, *B*, *C*, *D*} уже сформировано сочетание {*A*, *B*}, то, занимаясь поиском третьего эле- мента, программа будет рассматривать исключительно буквы, стоящие после *B*, в данном случае *С* и *D*.

#### Размещения с повторениями

Генерация размещений с повторениями напоминает аналогичную операцию для сочетаний.

// Генерируем размещения с повторениями. PermuteKofNwithDuplicates(Integer: index, Integer: selections[],

Data: items[], List<List<Data>> results)

// Проверяем, был ли это последний вызов. If (index == <длина размещения>) Then

// Добавляем результат к списку элементов.

List<Data> result = New List<Data>()

For i = 0 To <наибольший индекс в размещении> Result.Add(items[selections[i]])

Else

Next i results.Add(result)

// Выполняем следующий вызов.

For i = 0 To <наибольший индекс в массиве items>

// Добавляем i к выборке. selections[index] = i

End If

Next i

// Рекурсивно осуществляем другие вызовы. PermuteKofNwithDuplicates(index + 1, selections, items, results)

End PermuteKofNwithDuplicates

Основное отличие приведенного кода в том, что в цикле не устанавливается на- чальное значение, поэтому пересматриваются все элементы. Кроме того, они ис- пользуются в любом порядке.

**ПОДСЧЕТ РАЗМЕЩЕНИЙ С ПОВТОРЕНИЯМИ**

Предположим, вы формируете размещения с повторениями из *n* элементов по *k*. Каждая позиция в размещении может быть заполнена любым элементом из множества *n*, то есть совершается *k* независимых выборов (каждый после- дующий не зависит от предыдущих). Таким образом, существует *n*  *n*  ...  *n* =

= *nk* возможных размещений. В особом случае, если вы хотите сформировать размещение из всех имеющихся элементов *n* (то есть выполнить перестанов- ку), общее число результатов будет равняться *nn*.

#### Размещения без повторений

Размещения без повторений формируются подобно сочетаниям без повторе- ний. Чтобы их получить, в предыдущий алгоритм достаточно внести лишь одно небольшое изменение: при выборе элементов исключить те, которые уже были использованы. В следующем псевдокоде исправленные строки выделены полу- жирным шрифтом.

// Генерируем размещения без повторений. PermuteKofNwithoutDuplicates(Integer: index, Integer: selections[],

Data: items[], List<List<Data>> results)

// Проверяем, был ли это последний вызов. If (index == <длина размещения>) Then

// Добавляем результат к списку элементов. List<Data> result = New List<Data>()

For i = 0 To <наибольший индекс в размещении> Result.Add(items[selections[i]])

Else

Next i results.Add(result)

// Выполняем следующий вызов.

For i = 0 To <наибольший индекс в массиве items>

**// Убеждаемся, что элемент i еще не использовался. Boolean: used = false**

**For j = 0 To index - 1**

**Next j**

**If (selections[j] == i) Then used = true**

**If (Not used) Then**

// Добавляем i к выборке. selections[index] = i

End If

Next i

End If

// Рекурсивно осуществляем другие вызовы. PermuteKofNwithoutDuplicates(

index + 1, selections, items, results)

End PermuteKofNwithoutDuplicates

**ПОДСЧЕТ РАЗМЕЩЕНИЙ БЕЗ ПОВТОРЕНИЙ**

Предположим, вы формируете размещения без повторений из *n* элементов по *k*. В качестве первого элемента выборки алгоритм может взять любой из *n* имеющихся, второй ему придется искать из оставшихся *n* – 1 элементов. Перемножив все проделанные шаги, вы получите общее количество возможных размещений: *n*  (*n* – 1)  (*n* – 2)  ...  (*n* – *k* + 1).

В случае когда *k* = *n* и вы, по сути, совершаете перестановку всех элементов без повторений, формула примет вид *n*  (*n* – 1)  (*n* – 2)  ... 1 = *n*! Это число многие отождествляют с количеством размещений во множестве.

Большинство людей представляют себе размещения в виде перестановок из *n*

элементов по *n* без повторений.

## Удаление рекурсии

Рекурсия способствует лучшему пониманию некоторых проблем, например для головоломки «Ханойская башня» она дает простое и изящное решение. К со- жалению, есть у рекурсии и свои недостатки: иногда ее использование очевидно, но неэффективно.

Так, рекурсивный алгоритм генерирования чисел Фибоначчи требует от про- граммы многократного расчета одних и тех же величин. Это замедляет работу на- столько, что расчет более 50 значений становится непрактичным. Другие алгоритмы используют очень глубокие уровни рекурсии и приводят к тому, что программа ис- черпывает свой стек вызовов, как, например, ход коня с эвристическим алгоритмом Варнсдорфа для шахматной доски размером более 5757.

Чтобы предотвратить подобные проблемы, можно предпринять несколько шагов. В следующих подразделах вы познакомитесь с несколькими методами реструктуризации и удаления рекурсии, которые помогут улучшить произво- дительность.

#### Удаление хвостовой рекурсии

Удаление хвостовой рекурсии происходит в том случае, когда в последней операции перед возвратом алгоритм вызывает сам себя. В качестве примера рас- смотрим применение факториального алгоритма.

Integer: Factorial(Integer: n) If (n == 0) Then Return 1

Integer: result = n \* Factorial(n - 1) Return result

End Factorial

Первым делом алгоритм проверяет, надо ли ему вызывать самого себя или можно вернуть значение 1. Если рекурсивный вызов нужен, программа умножает текущий результат на *n* и возвращает его.

Рекурсивную версию алгоритма можно преобразовать в нерекурсивную с по- мощью цикла, внутри которого будут выполняться любые необходимые задания из исходного кода. Перед окончанием цикла алгоритму потребуется установить для параметров значения, используемые во время рекурсивного вызова. В случае факториального алгоритма это значение нужно отследить с помощью перемен- ной. Когда цикл повторяется, параметры устанавливаются для рекурсивного вы- зова — и алгоритм выполняет то, что делал рекурсивный вызов.

Конец цикла связан с условием, которое изначально заканчивало рекурсию. Для факториального алгоритма это *n* = 0. Когда алгоритм рекурсивно вызывает сам себя, он уменьшает параметр *n* на 1. Также должна поступать и нерекурсивная версия до окончания цикла.

В следующем псевдокоде представлена нерекурсивная версия факториально- го алгоритма.

Integer: Factorial(Integer: n)

// Создаем переменную для отслеживания возвращаемого значения.

// Приравниваем ее значение к 1, чтобы можно было умножить на нее результат.

// (Результат равен 1, если мы не входим в цикл.) Integer: result = 1

// Запускаем цикл, контролируемый условием остановки рекурсии. While (n != 0)

// Сохраняем результат для данного «рекурсивного» вызова. result = result \* n

// Готовимся к рекурсии. n = n - 1

Loop

// Возвращаем накопленный результат. Return result

End Factorial

Алгоритм кажется длиннее только из-за комментариев.

Удалить хвостовую рекурсию довольно просто, поэтому некоторые компили- рующие программы поступают подобным образом автоматически, чтобы сокра- тить требования к стековому пространству.

Конечно, главная проблема факториального алгоритма связана не с глубиной рекурсии, а с тем, что результаты становятся слишком большими и их тяжело хранить в типах данных с фиксированным размером. Для некоторых алгоритмов хвостовая рекурсия является полезной и обычно улучшает производительность,

поскольку проверка выходного условия цикла While осуществляется быстрее, чем вызов рекурсивного метода.

#### Хранение промежуточных значений

В алгоритме Фибоначчи хвостовая рекурсия не используется. Недостаток этой программы заключается в том, что она повторно рассчитывает слишком много промежуточных значений, а это отнимает время. Справиться с проблемой можно, если запоминать значения по мере их вычисления.

// Вычисляем значения. Integer: FibonacciValues[100]

// Наибольшее вычисленное значение на данный момент. Integer: MaxN

// Устанавливаем значения Fibonacci[0] и Fibonacci[1]. InitializeFibonacci()

FibonacciValues[0] = 0

FibonacciValues[1] = 1

MaxN = 1

End InitializeFibonacci

// Возвращаем n-е число Фибоначчи. Integer: Fibonacci(Integer: n)

// Если мы еще не вычисляли данное значение, находим его. If (MaxN < n) Then

FibonacciValues[n] = Fibonacci(n - 1) + Fibonacci(n - 2) MaxN = n

End If

// Возвращаем полученное значение. Return FibonacciValues[n]

End Fibonacci

В самом начале алгоритм объявляет глобально видимый массив FibonacciValues для хранения рассчитываемых величин. Переменная MaxN от- слеживает наибольшее значение *n*, для которого Fibonacci (*n*) находится в массиве.

Затем инициализируется метод InitializeFibonacci. Он устанавли- вает первые два числа Фибоначчи перед тем, как программа обратится к функ- ции Fibonacci. Эта функция сравнивает MaxN со своим входным параметром

1. Если *n*-е число Фибоначчи еще не определено, алгоритм рекурсивно вызывает

сам себя, чтобы его рассчитать, потом сохраняет полученное значение в массиве

FibonacciValues и обновляет MaxN.

В конце программа возвращает значение, которое хранится в массиве FibonacciValues. К этому моменту она уже знает, что оно было там ранее либо его поместили туда предыдущие строки кода.

В данной версии алгоритма каждое число Фибоначчи рассчитывается единож- ды, а затем просто извлекается из массива. В результате нет необходимости опре- делять промежуточные величины огромное количество раз. Если на моем компью- тере исходный алгоритм находил Fibonacci (44) приблизительно за 1 мин (и это был предел его возможностей), то усовершенствованная версия способна взяться за большие значения, например Fibonacci (92) изменяется почти мгновенно. Однако

рассчитать Fibonacci (93) даже у нее не получится, поскольку результат превышает целое число длиной 64 бита. (Если ваш язык программирования имеет доступ к бо- лее вместительным типам данных, то алгоритм легко справится с Fibonacci(1000) и более трудоемкими расчетами.)

Сохранение промежуточных величин повышает производительность програм- мы, но не удаляет рекурсию. Если вы хотите это сделать, разберитесь с тем, как ра- ботает следующий алгоритм.

Для расчета конкретного значения Fibonacci (*n*) программа сначала рекурсив- но определяет Fibonacci (*n* – 1), Fibonacci (*n* – 2) и т. д., вплоть до Fibonacci (2). Зна- чения Fibonacci (1) и Fibonacci (0) она находит в массиве FibonacciValues. По окончании каждого рекурсивного вызова полученный результат заносится в массив FibonacciValues, чтобы его можно было использовать выше по стеку. Для этого за- писи сохраняются в порядке возрастания: Fibonacci (2), Fibonacci (3), …, Fibonacci (*n*). Учитывая вышесказанное, можно удалить рекурсию и сделать так, чтобы алгоритм проходил похожие шаги для создания чисел Фибоначчи в порядке возрастания.

// Возвращаем n-е число Фибоначчи. Integer: Fibonacci(Integer: n)

If (n > MaxN) Then

// Вычисляем значения Fibonacci(MaxN) и Fibonacci(n). For i = MaxN + 1 To n

FibonacciValues[i] = Fibonacci(i - 1) + Fibonacci(i - 2)

Next i

// Обновляем MaxN. MaxN = n

End If

// Возвращаем полученное значение. Return FibonacciValues[n]

End Fibonacci

Представленная версия алгоритма начинается с вычисления всех чисел Фи- боначчи вплоть до необходимого числа. Затем программа возвращает это зна- чение. Можно даже поместить весь код вычисления в метод инициализации InitializeFibonacci, а затем заставить метод Fibonacci вернуть правиль- ное значение из массива.

#### Удаление общей рекурсии

Из предыдущих подразделов вы уже узнали, как удалить хвостовую рекурсию и рекурсию из алгоритма Фибоначчи. Однако как избавиться от нее в других си- туациях, для вас пока остается загадкой. Возьмем, например, многократно рекур- сивную кривую Гильберта, здесь удаление хвостовой рекурсии не решит проблему. Возможно, поработав над алгоритмом достаточно долго, вы сможете получить его нерекурсивную версию, но это будет сложно.

В общем случае, чтобы избавиться от рекурсии, надо попытаться имитировать ее действия в программе. Перед совершением рекурсивного вызова алгоритм хра- нит информацию о своем текущем состоянии в стеке. Когда вызов возвращается, информация из стека выталкивается, чтобы код возобновил работу с того места, на котором остановился.

Чтобы воспроизвести подобные действия, представьте, что в алгоритме перед каждым рекурсивным вызовом находятся отдельные части. Назовем их 1, 2, 3 и т. д. Создайте переменную section, которая будет указывать на то, какую часть нужно запустить, и приравняйте ее к 1, чтобы отсчет начался с первой части кода.

Для организации работы используйте цикл While, он будет выполняться до тех пор, пока section > 0. Переместите весь код алгоритма в тело цикла и добавьте в него операторы If-Else. Сделайте так, чтобы каждый оператор If сравнивал переменную section с номером части: если они совпадают, должен выполняться соответствующий код. (Вместо операторов If-Else можно использовать оператор Switch или Select Case.)

При работе с определенной частью не забудьте увеличить переменную section, чтобы алгоритм знал, какая часть будет следующей в ходе цикла. Когда програм- ма обычным образом рекурсивно вызовет сама себя, вставьте все текущие значе- ния параметров в стеки. Добавьте туда и section, чтобы коду было понятно, ка- кую часть выполнять по возвращении из псевдорекурсии. Обновите параметры, ко- торые должны использоваться псевдорекурсией. Наконец, приравняйте section к 0, чтобы начать псевдорекурсивный вызов в первой части кода.

В следующем псевдокоде исходный алгоритм кривой Гильберта разбит на части после каждой рекурсии.

Hilbert(Integer: depth, Float: dx, Float: dy)

// Часть 1.

If (depth > 0) Then Hilbert(depth - 1, dy, dx)

// Часть 2. DrawRelative(dx, dy)

If (depth > 0) Then Hilbert(depth - 1, dx, dy)

// Часть 3. DrawRelative(dy, dx)

If (depth > 0) Then Hilbert(depth - 1, dx, dy)

// Часть 4. DrawRelative(-dx, -dy)

If (depth > 0) Then Hilbert(depth - 1, -dy, -dx)

// Часть 5.

End Hilbert

Ниже представлена нерекурсивная версия алгоритма.

// Чертим кривую Гильберта.

Hilbert(Integer: depth, Float: dx, Float: dy)

// Создаем стеки для хранения информации до рекурсии. Stack<Integer> sections = new Stack<int>(); Stack<Integer> depths = new Stack<int>(); Stack<Float> dxs = new Stack<float>();

Stack<Float> dys = new Stack<float>();

// Определяем, какая часть кода будет выполняться следующей. Integer: section = 1

While (section > 0)

If (section == 1) Then

section = section + 1 If (depth > 0) Then

sections.Push(section) depths.Push(depth) dxs.Push(dx) dys.Push(dy)

// Hilbert(depth - 1, gr, dy, dx) depth = depth - 1

float temp = dx dx = dy

dy = temp section = 1

End If

Else If (section == 2) Then DrawRelative(gr, dx, dy) section = section + 1

If (depth > 0) Then

sections.Push(section) depths.Push(depth) dxs.Push(dx) dys.Push(dy)

// Hilbert(depth - 1, gr, dx, dy) depth = depth - 1

section = 1

End If

Else If (section == 3) Then DrawRelative(gr, dy, dx) section = section + 1

If (depth > 0) Then

sections.Push(section) depths.Push(depth) dxs.Push(dx) dys.Push(dy)

// Hilbert(depth - 1, gr, dx, dy) depth = depth - 1

section = 1

End If

Else If (section == 4) Then DrawRelative(gr, -dx, -dy) section = section + 1

If (depth > 0) Then

sections.Push(section) depths.Push(depth) dxs.Push(dx) dys.Push(dy)

// Hilbert(depth - 1, gr, -dy, -dx) depth = depth - 1

float temp = dx dx = -dy

dy = -temp section = 1

End If

Else If (section == 5) Then

// Возвращаемся из рекурсии.

// Если выталкивать нечего, мы наверху.

If (sections.Count == 0) Then section = -1 Else

// Выталкиваем предыдущие параметры. section = sections.Pop()

End Hilbert

End If End While

depth = depths.Pop() dx = dxs.Pop()

dy = dys.Pop()

Данный вариант алгоритма немного длиннее, поскольку в нем содержатся не- сколько копий кода для вставки значений в стеки, обновления параметров и вы- талкивания значений из стеков.

## Резюме

Рекурсия представляет собой мощную технологию, и некоторые задачи уже сами по себе требуют ее применения, например головоломка «Ханойская башня». Кроме того, рекурсия позволяет создавать интересные фигуры (самоподобные кривые и салфетки с малым количеством кода), а также применять алгоритмы с возвратом. С ее помощью удобно выполнять задания, где требуется повторять конкретные шаги неопределенное количество раз. Так, сгенерировать сочетание или размещение элементов довольно легко, главное — знать заранее, сколько их будет в итоговой выборке. Но если это неизвестно, легче воспользоваться рекурсией.

Несмотря на неоспоримую пользу, рекурсия может вызывать некоторые проб- лемы. Иногда ее бесконтрольное применение вынуждает алгоритм многократно повторять одни и те же расчеты, как в случае с расчетом чисел Фибоначчи. Кроме того, глубокие уровни рекурсии способствуют быстрому расходованию стекового пространства и приводят к зависанию программы. В подобных случаях рекурсию целесообразно удалить, чтобы улучшить производительность.

Перечисленные недостатки не умаляют всей мощи и пользы рассмотренной тех- нологии. Еще один вариант ее использования — работа с естественно рекурсивными структурами данных, например с деревьями, о которых пойдет речь в следующих трех главах.

## Упражнения

Для выполнения некоторых упражнений потребуется опыт графического про- граммирования. Такие задания отмечены звездочками, что говорит об их повышен- ной сложности. То, как вы будете их выполнять, зависит от используемой среды программирования. Обратите также внимание на задачи о восьми ферзях и о ходе коня, они подходят для графической реализации.

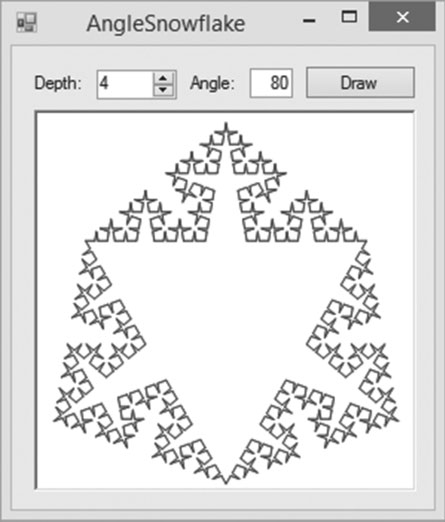
* 1. Напишите программу, реализующую исходный рекурсивный факториаль- ный алгоритм.
  2. Создайте программу с использованием исходного рекурсивного алгорит- ма Фибоначчи.
  3. Реализуйте программу по решению головоломки «Ханойская башня». Вам нужно получить ряд ходов вида *A*➔*B*, которые представляют собой переме-

щение верхнего диска с колышка *А* на колышек *B*. Вот примерный результат перемещения трех дисков: *A*➔*B A*➔*C B*➔*C A*➔*B C*➔*A C*➔*B A*➔*B*.

4\*. Напишите программу для решения головоломки «Ханойская башня» и по- кажите ходы, изобразив передвижения дисков между колышками. (Ищите подсказку в приложении Б).

5\*. Создайте программу для рисования снежинок Коха.

6\*.В стандартной снежинке Коха генератор создает углы, равные 60. Р но мож- но использовать и другие значения для получения интересных результатов. Напишите программу, которая разрешит пользователю задавать угол в ка- честве входного параметра и представит результат, подобный тому, что изо- бражен на рисунке 9.15.



**Рис. 9.15.** Снежинка Коха, построенная с использованием угла 80

1. Реализуйте программу для черчения кривых Гильберта. (О том, как устанав- ливать dx, читайте в приложении Б.)
2. Напишите псевдокод для алгоритмов, которые чертят части кривой Серпин- ского вниз, влево и вверх.

9\*. Создайте программу, которая чертит кривые Серпинского. (О том, как уста- навливать dx, читайте в приложении Б.)

1. Напишите псевдокод низкого уровня, чтобы начертить салфетку Серпинского.
2. Напишите псевдокод низкого уровня, чтобы начертить ковер Серпинского.
3. Реализуйте программу для решения задачи о восьми ферзях.
4. Усовершенствуйте программу из предыдущего упражнения: отследите, сколько ферзей может атаковать определенную позицию на доске. Добав- ляя очередную фигуру, игнорируйте позиции, для которых значение не рав- но 0. Как проделанные модификации повлияют на расстановку одного фер- зя и на общее время работы алгоритма?
5. Обратите внимание, что в задаче о восьми ферзях в каждом ряду на шахмат- ной доске должна стоять лишь одна фигура. Усовершенствуйте программу из упражнения 12 так, чтобы каждый вызов метода EightQueens искал по- зицию для нового ферзя только в следующем ряду. Как это повлияет на ко- личество возможных положений фигуры и общее время работы алгоритма?
6. Напишите программу, которая решает задачу о ходе коня методом возвра- та и в которой пользователю разрешается задавать размеры шахматной до- ски. Каковы будут наименьшие размеры доски, чтобы ход конем оказался возможен?
7. Используйте любимый язык программирования, чтобы решить задачу о ходе коня с использованием эвристического алгоритма Варнсдорфа.
8. Какова связь между выборкой без повторений и размещением без повторений?
9. Попробуйте написать программу, которая будет реализовывать алгорит- мы SelectKofNwithDuplicates и SelectKofNwithoutDuplicates.
10. Постарайтесь написать программу, которая будет работать с алгоритмами

PermuteKofNwithDuplicates и PermuteKofNwithoutDuplicates.

1. Реализуйте нерекурсивный факториальный алгоритм.
2. Разработайте рекурсивный алгоритм Фибоначчи с сохраняемыми значе- ниями.
3. Напишите программу, применяющую нерекурсивный алгоритм Фибоначчи.
4. Нерекурсивный алгоритм Фибоначчи рассчитывает числа до необходимого предела, а затем находит требуемое значение в массиве. На самом деле ал- горитм может рассчитывать меньшие значения Фибоначчи, лишь когда они нужны. Это займет немного больше времени, но зато вам не потребуется содержать полноценный массива. Напишите программу, реализующую не- рекурсивный алгоритм Фибоначчи с использованием данного метода.
5. Разработайте нерекурсивный алгоритм построения кривой Гильберта.

# Глава 10 ДЕРЕВЬЯ

В этой главе рассматриваются такие высоко рекурсивные структуры данных, как деревья. Они могут использоваться для отражения иерархии и моделирования процессов принятия решений. В виде дерева удобно хранить структуру организа- ции компании или каталог частей, из которых состоит, скажем, автомобиль. В рам- ках приведенного ниже текста вы узнаете, как формируются относительно простые деревья, и усвоите базовые знания, необходимые для работы с более сложными де- ревьями, о которых пойдет речь в главах 11 и 12.

## Терминология

В программировании при описании деревьев часто пользуются терминами, ко- торые связаны с генеалогией, садоводством и вычислительной техникой. Большин- ство из них будет вам интуитивно понятно, поскольку вы, вероятно, уже сталкива- лись с ними в другом контексте.

Итак, дерево состоит из *вершин (узлов)*, которые содержат данные и соединя- ются *ветвями*. Последние изображают в виде стрелок, ведущих от *родительской вершины* к *дочерней*.

**ЗАМЕЧАНИЕ**

 Дерево — особый вид сети, или графа, поэтому по аналогии ветви, соеди- няющие вершины, иногда называют *ссылками* или *линиями*. Более подробно

* сетях рассказано в главах 13 и 14.

У каждой вершины, за исключением *корневой*, есть одна родительская вершина. Две вершины с общим родителем иногда называют *узлами-сестрами*, дочерние вер- шины дочерних вершин — *потомками*, а родительские вершины родительских вер- шин — *предками*. Все эти термины, описывающие связи, очень близки к генеалогии. Можно даже использовать такие определения, как двоюродный брат, племянник, дедушка, — они не внесут путаницы, хотя и покажутся несколько непривычными.

В зависимости от типа дерева у вершины может быть разное количество по- томков. Это число называется *степенью вершины*. Его максимальное значение определяет *степень дерева* в целом. Например, дерево второй степени является *би- нарным*, у каждой его вершины может быть максимум два дочерних узла. Вершина,

не имеющая дочерних узлов, называется *терминальной (листовой)*; содержащая хотя бы один дочерний узел — *внутренней*.

У деревьев в программировании корень обычно располагается вверху, а ветви направлены вниз (рис. 10.1).

Корень



Ветви Внутренние

4 вершины

2 5

1 3 7

6 8

Терминальные вершины

**Рис. 10.1.** Древовидная структура данных

Согласно всему вышесказанному, можно утверждать, что деревом является:

* вершина с одним корнем;
* корневая вершина, соединенная ветвями с одним или несколькими мень- шими деревьями.

*Уровень (глубина*) вершины — это расстояние от вершины до корня. У корня оно будет равно 0.

*Высота вершины* — это длина нисходящего пути от текущей вершины до ли- стовой, то есть до низа дерева. Высота дерева равна высоте корневой вершины.

*Поддеревом* дерева *T* с корнем *R* является вершина *R* со всеми ее потомками. Так, на рисунке 10.1 поддерево с корнем в вершине 5 — это дерево, содержащее вершины 5, 7, 6 и 8.

В *упорядоченном дереве* расположение дочерних деревьев имеет значение. Мно- гие алгоритмы по-разному рассматривают левую и правую дочерние вершины. В *неупорядоченном* расположение дочерних деревьев не играет роли. (Как правило, деревья упорядоченны, даже если это не особо важно для алгоритма. Все дело в том, что дочерние вершины или ветви хранятся в массиве или другой совокупности, предполагающей сортировку.)

Для любых двух вершин *первый (наименьший) общий предок* — это узел-предок, расположенный наиболее близко к ним. Чтобы его обнаружить, нужно проследовать от одной из вершин к корню до нахождения общего родительского узла с другой вершиной. На рисунке 10.1 для вершин 3 и 5 первым общим предком будет корень 4. Следует отметить, что таким предком может оказаться и одна из двух вершин. На том же рисунке для вершин 5 и 6 это будет 5.

Обратите также внимание, что между любыми двумя вершинами дерева су- ществует уникальный путь, который не следует по одной ветви дважды. Он берет начало из первой вершины, проходит вверх по дереву к наименьшему общему предку, а затем спускается ко второй вершине.

В *полном дереве* вершина не имеет дочерних вершин либо их количество равно степени дерева. Например, в полном бинарном дереве у каждого узла может быть два дочерних или ни одного. Дерево, изображенное на рисунке 10.1, неполное, по- скольку у вершины 5 имеется только одна дочерняя вершина 7.

У *завершенного дерева* каждый уровень является полным, за исключением, возможно, нижнего, где все вершины сдвигаются влево. На рисунке 10.2 показано завершенное бинарное дерево. Оно неполное, поскольку третья вершина на втором уровне имеет лишь одну дочернюю.



**Рис. 10.2.** Завершенное бинарное дерево

*Идеальное дерево* — полное дерево, все листья которого находятся на одном уровне. Другими словами, в нем присутствуют все вершины, допустимые для его высоты.

На рисунке 10.3. представлены примеры различных типов деревьев.



Полное дерево Завершенное дерево Идеальное дерево

**Рис. 10.3.** Примеры деревьев с возрастающим количеством вершин для данной высоты

Чтобы облегчить понимание материала, все упомянутые термины сведены в та- блицу 10.1.

**Òàáëèöà 10.1.** Òåðìèíîëîãèÿ, ñâÿçàííàÿ ñ äåðåâüÿìè

|  |  |
| --- | --- |
| **Термин** | **Значение** |
| Брат (сестра) | Две вершины дерева, имеющие общего родителя |
| Вершина (узел) | Объект дерева, содержащий данные и соединенный с другими такими же объектами с помощью ветвей |
| Вершина внутренняя | Вершина, содержащая как минимум одну дочернюю вершину |
| Вершина дочерняя | Вершина, соединенная с родительской вершиной и, как правило, расположенная на уровень ниже ее |
| Вершина терминальная (листовая) | Вершина, не содержащая дочерних вершин |
| Ветвь | Часть дерева, соединяющая вершины |
| Высота | Применительно к вершине самый длинный путь, ведущий от нее вниз по дереву к терминальной (листовой) вершине.  Применительно к дереву то же, что и высота корня |
| Глубина | Уровень |
| Дерево бинарное | Дерево второго уровня |
| Дерево завершенное | Дерево, в котором каждый уровень является полным, кроме, воз- можно, самого нижнего, где вершины сдвинуты влево |
| Дерево идеальное | Полное дерево, в котором все листья находятся на одном уровне |
| Дерево полное | Дерево, в котором каждая вершина содержит нуль дочерних вер- шин либо их количество равно степени дерева |
| Дерево упорядоченное | Дерево, для которого важна сортировка всех дочерних узлов |
| Корень | Самая верхняя вершина дерева, не имеющая родительских вершин |
| Лист | Терминальная вершина |
| Поддерево | Вершина дерева со всеми ее потомками |
| Потомок | Дочерние вершины дочерних вершин и т. д. вниз по дереву до листьев |
| Предок | Родительская вершина родительской вершины и т. д. вверх по древу до корня |
| Предок первый (наименьший) общий | Один и тот же предок для двух любых вершин, который располо- жен к ним ближе всего |
| Родитель | Вершина, соединенная с дочерними вершинами с помощью вет- вей и, как правило, расположенная на уровень выше них. Каж- дый узел дерева, кроме корня, имеет только одного родителя |
| Степень | Применительно к вершине общее количество ее дочерних вершин. По отношению к дереву максимальная степень любой из ее вершин |
| Уровень | Расстояние между данной вершиной и корнем дерева |

Теперь, когда вам знакомы все необходимые термины, можно приступить к из- учению некоторых свойств и функций деревьев.

## Свойства бинарного дерева

Бинарные деревья используются во многих алгоритмах. Во-первых, они легки для понимания, а во-вторых, большинство задач основано на двойном выборе. Вот несколько полезных фактов о бинарных деревьях.

* Количество ветвей *B* в бинарном дереве из *N* вершин находится по формуле

*B* = *N* – 1.

* Количество вершин *N* в идеальном бинарном дереве высотой *H* рассчитыва- ется следующим образом: *N* = 2*H* + 1 – 1.
* Если в идеальном бинарном дереве *N* вершин, его высота определяется как log2 (*N* + 1) – 1.
* Количество терминальных вершин *L* в идеальном бинарном дереве высотой *H* вычисляется по формуле *L* = 2*H* + 1 – 1, а количество внутренних вершин *I* — по формуле *I* = *N* – *L* = (2*H* + 1 – 1) – 2*H* = (2*H* + 1 – 2*H*) – 1 = 2*H*  (2 – 1) – 1 = 2*H* – 1. Отсюда следует, что в идеальном бинарном дереве количество терминальных и внутренних вершин практически одинаково: *I* = *L* – 1.
* Количество недостающих ветвей *M* (места, в которые можно добавить дочер- нюю вершину) в бинарном дереве из *N* вершин рассчитывается как *M* = *N* + 1.
* Если в бинарном дереве *N*0 терминальных вершин и *N*2 вершин второго уров- ня, то *N*0 = *N*2 + 1. Другими словами, терминальных вершин всегда на одну больше, чем вершина второго уровня.

**ТЕРМИНАЛЬНЫЕ И ПОЛНЫЕ ВЕРШИНЫ**

Последнее утверждение требует некоторых пояснений.

* 1. Пусть *N* — общее количество вершин, *B* — общее количество ветвей, а *N*0, *N*1 и *N*2 — количество вершин степени 0, 1 и 2 соответственно.
  2. От каждой вершины к родительской вершине ведет единственная ветвь, поэтому *B* = *N* – 1.
  3. Вершины *N*0 не имеют исходящих ветвей, у вершин *N*1 по одной такой ветви, у вершин *N*2 — по две. Таким образом, общее количество ветвей определяется как *B* = *N*1 + 2*N*2.
  4. Приравняв оба уравнения для *B*, получим *N* – 1 = *N*1 + 2*N*2. Затем при- бавим к обеим частям по единице: *N* = *N*1 + 2*N*2 + 1.
  5. Общее количество вершин дерева — это сумма всех трех видов его вершин, то есть *N* = *N*0 + *N*1 + *N*2.
  6. Объединение двух уравнений для *N* даст следующее: *N*1 + 2*N*2 + 1 = *N*0 +

+ *N*1 + *N*2. На завершающем этапе вычтем *N*1 + *N*2 из обеих частей. В ре- зультате получим формулу *N*2 + 1 = *N*0.

Благодаря приведенным фактам производительность алгоритмов, связанных с деревьями, рассчитывать намного легче. Так, если программе понадобится пройти по всему идеальному бинарному дереву из *N* вершин от корня до листьев, то ей придется совершить *O*(log *N*) шагов.

**ИНДУКТИВНОЕ РАССУЖДЕНИЕ**

Многие из указанных фактов можно доказать методом математической ин- дукции. Для этого сначала нужно установить базис для малой задачи, а затем предпринять индуктивный переход, который гласит: если свойство истинно для определенного значения *K*, то оно истинно и для значения *K* + 1, получается, свойство справедливо для всех значений *K*.

Рассмотрим в качестве примера второй факт, утверждающий, что количество вершин *N* в идеальном бинарном дереве высотой *H* находится по формуле *N* = 2*H* + 1 – 1. (Далее в индуктивном доказательстве в роли *K* выступает *H*.)

**База индукции.** Рассмотрим идеальное дерево высотой *H* = 0 с одной кор- невой вершиной и без ветвей. В таком случае количество вершин *N* равно 1. Обратите внимание: 2*H* + 1 – 1 = 20 + 1 – 1 = 21 – 1 = 2 – 1 = 1, поэтому *N* = 2*H* + 1 – 1, что и требовалось получить.

**Шаг индукции.** Предположим, что свойство истинно для идеальных бинарных деревьев высотой *H*. Пусть есть идеальное бинарное дерево высотой *H* + 1, у которого корневая вершина соединена с двумя идеальными бинарными под- деревьями высотой *H*. Тогда в каждом поддереве содержится 2*H* + 1 – 1 вершин. Добавив корневую вершину, получим формулу для расчета общего количество вершин в дереве новой высоты: 2  (2*H* + 1 – 1) + 1. Небольшая перестановка даст следующее: (2(*H* + 1) + 1 – 2) + 1 = 2(*H* + 1) + 1 – 1. По сути, это и есть первоначальная формула, только вместо *H* подставлено *H* + 1.

Значит, свойство истинно, а количество вершин в идеальном бинарном дереве высотой *H* равно 2*H* + 1 – 1 для всех *H*.

Если бинарное дерево из *N* вершин является толстым, другими словами, не очень высоким и тонким, например полным, то для него прогнозы относительно *О* будут близки к идеальному бинарному дереву: высота — *O*(log *N*), количество листов — *O*(*N*/2) = *O*(*N*), количество внутренних вершин — *O*(*N*/2) = *O*(*N*), коли- чество недостающих ветвей — *O*(*N*).

Вышесказанное будет справедливо и для толстых деревьев более высоких степе- ней, но изменится основание логарифма. Так, высота толстого дерева степени 10 из *N* вершин окажется равной *O*(log10 *N*). Поскольку для *О* все основания логарифмов

одинаковы, это то же, что *O*(log *N*). Однако на практике игнорируемые константы

могут иметь значение.

В главе 11 описываются сбалансированные деревья, которые не вырастают слиш- ком высокими или тонкими. Это гарантирует, что все свойства будут истинными.

## Представление деревьев

Вершины деревьев можно представить с помощью классов, а для хранения пол- ных деревьев использовать массив. В следующих подразделах описаны оба метода.

#### Общие правила построения деревьев

В случае деревьев классы используются для создания вершин аналогично тому, как в связных списках они используются для создания ячеек. Назначьте классу

любые свойства, необходимые для хранения данных (в полном бинарном дереве это могут быть LeftChild и RightChild), и ссылки на объекты для представления ветвей, ведущих к дочерним вершинам.

Ниже в псевдокоде создается класс бинарной вершины. В зависимости от ис- пользуемого вами языка программирования в нем могут быть те или иные изменения.

Class BinaryNode String: Name

BinaryNode: LeftChild BinaryNode: RightChild Constructor(String: name)

Name = name End Constructor

End Class

Класс начинается с объявления открытого свойства Name — имя вершины. За- тем определяются свойства LeftChild и RightChild для ссылок на дочерние узлы. На следующем этапе конструктор класса берет строку в качестве параметра и сохраняет ее в свойстве вершины Name. Приведенный ниже алгоритм с помощью класса выстраивает дерево, изображенное на рисунке 10.1.

BinaryNode: root = New BinaryNode(«4») BinaryNode: node1 = New BinaryNode(«1») BinaryNode: node2 = New BinaryNode(«2») BinaryNode: node3 = New BinaryNode(«3») BinaryNode: node5 = New BinaryNode(«5») BinaryNode: node6 = New BinaryNode(«6») BinaryNode: node7 = New BinaryNode(«7») BinaryNode: node8 = New BinaryNode(«8») root.LeftChild = node2

root.RightChild = node5 node2.LeftChild = node1 node2.RightChild = node3 node5.RightChild = node7 node7.LeftChild = node6 node7.RightChild = node8

Первый созданный объект BinaryNode представляет корень дерева, после- дующие — остальные вершины. После того как все узлы определены, между ними устанавливаются ссылки. Если степень дерева больше 2 или оно не упорядоченно, дочерние ссылки удобнее разместить в массиве, списке или другой совокупности данных. Это позволит программе циклически пройти по дочерним вершинам и вы- полнить необходимые действия, а вам не придется писать для каждого дочернего узла отдельную строку кода.

В следующем псевдокоде создается класс TreeNode, допускающий присутствие любого количества дочерних вершин при каждой родительской.

Class TreeNode String: Name

List Of TreeNode: Children

Constructor(String: name) Name = name

End Constructor End Class

Этот код отличается от предыдущего тем, что дочерние вершины содержатся не в отдельных свойствах, а в списке ссылок List на объекты TreeNode.

Обратите внимание: в представлениях присутствуют ссылки только на дочерние вершины, на родительские ссылок нет. Большинство работающих с деревьями алго- ритмов придерживается иерархии, поэтому передвигается вниз по дереву: от более высокого уровня к более низкому. Для определения родительской вершины добавьте к классу свойство Parent. Еще одна особенность подобных алгоритмов — они хранят данные в вершинах и лишь некоторые — в ветвях. Чтобы реализовать последнюю идею, необходимо добавить информацию в родительскую вершину либо создать отдельный класс Branch. Первый метод представлен в следующем псевдокоде.

Class TreeNode String: Name

List Of TreeNode: Children List Of Data: ChildData

Constructor(String: name) Name = name

End Constructor End Class

В этом классе добавление дочерней вершины должно сопровождаться добавле- нием данных для ведущей к ней ветви. Получается, что ChildData[i] содержит данные для ветви, направленной к Children[i].

Ниже представлен класс Branch, который можно использовать для хранения информации о ветвях.

Class Branch

Data: BranchData TreeNode: Child

End Class

В классе содержатся все необходимые данные и ссылка на дочерний объект. Если для хранения информации используется ветвь, класс вершины нужно заме- нить. В следующем псевдокоде показано, как это сделать по отношению к TreeNode.

Class TreeNode String: Name

List Of Branch: Branches

Constructor(String: name) Name = name

End Constructor End Class

Теперь, чтобы проверить дочерние узлы вершины, следует пройти циклом по списку Branches и использовать каждое свойство Child объекта Branch.

**XML И ДЕРЕВЬЯ**

XML (от англ. *eXtensible Markup Language*) — расширяемый язык разметки для представления данных. Созданные на его основе документы иерархичны: знаки определяются внутри других знаков. Все это хорошо подходит для постоянного хранения деревьев и их передачи из одной программы (компьютера) в другую.

Например, можно в файле XML сформировать структуру вашей компании, а за- тем переслать его какому-нибудь пользователю. Получить дополнительную информацию об XML можно по ссылкам ru.wikipedia.org/wiki/XML или w3schools.com/ xml/xml\_whatis.asp либо из книги Дэвида Хантера, Джеффа Рафтера, Джо Фаусетта и др. «Работа с XML» (4-е изд., М.: Диалектика, 2009).

#### Построение завершенных деревьев

В алгоритме пирамидальной сортировки (см. подраздел «Пирамидальная сор- тировка» раздела «Алгоритмы *O*(*N*  log *N*)» в главе 6), используется завершенное бинарное дерево, которое хранится в массиве для представления кучи. Напомним, что значение каждой родительской вершины в куче должно быть сопоставимо со значением ее дочерних вершин (рис. 10.4).



15

11

10

7

5

9

**2**

**6** 4 3 1 8

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 11 | 10 | 7 | 5 | 9 | 2 | 6 | 4 | 3 | 1 | 8 |

**Рис. 10.4.** Куча в виде дерева, хранящегося в массиве



Если вершина хранится в массиве с индексом *i*, ее дочерние вершины будут иметь индексы 2*i* + 1 и 2*i* + 2. Для вершины с индексом *j* родительская вершина получит индекс (*j* – 1) / 2 , где запись означает округление результата до бли-

жайшего меньшего целого числа. Например, l2,9J и l2J равняются одному и тому же

l J l J

числу 2. Так образуется сжатый формат для хранения любого завершенного бинар- ного дерева в массиве.

Работа с подобным деревом осложняется, когда приходится менять размер массива, поэтому, возможно, лучше отдать предпочтение классам.

**ЗАМЕЧАНИЕ**

Познакомившись с деревьями более детально, вернитесь в подраздел «Пира- мидальная сортировка» раздела «Алгоритмы *O*(*N* log *N*)» главы 6, чтобы понять, как использовать классы вместо массива для построения кучи.

## Обход дерева

Важнейшая операция, связанная с деревьями, — *обход*. Ее цель — пройти по всем вершинам в дереве в определенном порядке и выполнить некоторые действия (в самом простом случае только перечислить).

**ОБХОД И ПОИСК**

Многие алгоритмы проводят поиск определенной вершины по дереву. В его основе также лежит обход. В главе 11 вы узнаете об эффективных способах проведения поиска.

Для бинарных деревьев существует четыре вида обходов: в прямом порядке, в обратном порядке, симметричный и в ширину.

#### Обход в прямом порядке

При таком обходе алгоритм вначале обрабатывает вершину, затем ее левый до- черний узел, а после правый. Рассмотрим дерево, представленное на рисунке 10.5, и постараемся показать его вершины в прямом порядке.



*D*

*B*

*E*

*A*

*C*

**Рис. 10.5.** Исходное дерево для обхода в прямом порядке

Как уже было сказано, первым делом алгоритм обратится к корню и выведет значение *D*. Затем он переместится к левой дочерней вершине корня, выведет *B* и рассмотрит уже ее левый дочерний узел, то есть *A*. Больше дочерних вершин нет, поэтому алгоритм вернется в узел *B* и проследует к его правой дочерней верши- не, в нашем случае к *С*. У нее тоже нет потомков, следовательно, произойдет еще один возврат к вершине *B*. Раз дочерних узлов у этой вершины больше не имеет- ся, то программа поднимется вверх по дереву к корню *D* и проследует к его право- му дочернему узлу *E*. Не обнаружив и у него потомков, алгоритм вновь обратится к корню, констатирует отсутствие любых других дочерних вершин и завершит об- ход. В результате порядок обхода будет выглядеть так: *D*, *B*, *A*, *C*, *E*.

Примите во внимание, что алгоритм изучает (проходит) вершины в одном порядке, а итоги проделанной работы выводит в другом. В следующем списке представлены шаги, которые выполняются во время прямого обхода для дерева, изображенного на рисунке 10.5.

1. Пройти *D*.
2. Вывести *D*.
3. Пройти *B*.
4. Вывести *B*.
5. Пройти *A*.
6. Вывести *A*.
7. Пройти *B*.
8. Пройти *C*.
9. Вывести *C*.
10. Пройти *B*.
11. Пройти *D*.
12. Пройти *E*.
13. Вывести *E*.
14. Пройти *D*.

Рекурсивная реализация данного алгоритма представлена ниже.

TraversePreorder(BinaryNode: node)

<Обрабатываем вершину.>

If (node.LeftChild != null) Then TraversePreorder(node.LeftChild) If (node.RightChild != null) Then TraversePreorder(node.RightChild)

End TraversePreorder

Короткая и простая программа фактически иллюстрирует определение обхода в прямом порядке. Она начинается с обработки текущей вершины, но вместо этого действия можно вставить любой нужный код (добавьте, например, текущую метку вершины к строке вывода или изучите вершину, чтобы определить, был ли найден целевой элемент). Далее алгоритм определяет, содержит ли вершина левый до- черний узел. Если да — программа рекурсивно вызывает саму себя, чтобы обойти левое дочернее поддерево, затем повторяет тот же шаг для правого дочернего узла и заканчивает работу. Таким образом, чтобы исследовать все дерево, нужно вызы- вать алгоритм TraversePreorder и передать ему в качестве параметра корневую вершину.

Представленный код нужно разместить в главной программе, модуле кода или вспомогательном классе. Удобнее всего управлять деревом внутри класса вершины. В следующем псевдокоде это класс BinaryNode.

Class BinaryNode String: Name

BinaryNode: LeftChild BinaryNode: RightChild

Constructor(String: name) Name = name

End Constructor TraversePreorder()

<Обрабатываем вершину.>

If (LeftChild != null) Then TraversePreorder(LeftChild) If (RightChild != null) Then TraversePreorder(RightChild)

End TraversePreorder End Class

Алгоритм имеет много общего с предыдущей версией, за исключением того, что код работает в объекте BinaryNode и обладает прямым доступом к свойствам LeftChild и RightChild данного объекта. Это немного упрощает запись и позво- ляет сохранить ее в инкапсулированном виде в классе BinaryNode. Теперь, чтобы обойти все дерево, нужно вызвать метод корневой вершины TraversePreorder.

**МЕТОДЫ ПРОХОЖДЕНИЯ**

В большинстве языков программирования ссылку методу можно передать в ка- честве параметра. В данном примере это означает, что TraversePreorder, дойдя до шага <Обрабатываем вершину.>, способен вызвать метод, ко- торый сделает с деревом все, что нужно, обработав вершины должным об- разом.

Обход в прямом порядке используется не только для бинарных деревьев, но и для тех, что имеют более высокую степень. Неизменным остается общее правило: первым делом изучается сама вершина, а затем ее дочерние узлы.

#### Симметричный обход

При симметричном обходе алгоритм обрабатывает левый дочерний узел вер- шины, затем ее саму и только после этого правый дочерний узел.

Работая с деревом, изображенным на рисунке 10.5, программа начнет двигать- ся с корня, переместится сразу к левому дочернему узлу *B*, а через него к левому дочернему узлу *А*. У этой вершины уже нет левого потомка, поэтому программа выведет *А* и, не обнаружив правого потомка, вернется к родительской вершине *B*, чтобы вывести ее. После этого алгоритм проследует к правому дочернему узлу *С*, у которого также отсутствует левый потомок. Выведя *С*, программа удостове- рится, что нет и правого потомка, и снова вернется к родительской вершине *B*. Поскольку работа с левой частью дерева уже закончена, алгоритм поднимется к корню *D*, выведет его и обратится к правой части с дочерней вершиной *E*. У нее нет левого дочернего узла, значит, алгоритм выведет *E* и, не найдя правого дочер- него узла, вернется к корню *D*. Итоговый порядок симметричного обхода будет таким: *A*, *B*, *C*, *D*, *E*.

Обратите внимание, что в выводе вершины располагаются в отсортированном порядке. Упорядоченные деревья как раз и строятся таким образом, чтобы симмет- ричный обход дал на выходе отсортированные значения.

Рекурсивное применение описанного алгоритма представлено в следующем псевдокоде.

TraverseInorder(BinaryNode: node)

If (node.LeftChild != null) Then TraverseInorder(node.LeftChild)

<Обрабатываем вершину.>

If (node.RightChild != null) Then TraverseInorder(node.RightChild) End TraverseInorder

Запись полностью соответствует определению симметричного обхода. Програм- ма рекурсивно обрабатывает левый дочерний узел вершины, если он существует, затем саму вершину, а после правый дочерний узел, если таковой есть.

Чтобы обойти целое дерево, достаточно вызвать метод TraverseInorder и передать ему корневую вершину в качестве параметра. Следующий псевдокод представляет такой же алгоритм, внедренный внутри класса BinaryNode.

Class BinaryNode String: Name BinaryNode: LeftChild BinaryNode: RightChild

Constructor(String: name) Name = name

End Constructor

TraverseInorder()

If (LeftChild != null) Then TraverseInorder(LeftChild)

<Обрабатываем вершину.>

If (RightChild != null) Then TraverseInorder(RightChild) End TraverseInorder

End Class

В отличие от обхода в прямом порядке, для дерева со степенью больше 2 опреде- лить симметричный обход непросто. Можно заставить алгоритм обработать одну половину дочерних узлов, а затем вершину и оставшиеся дочерние узлы, но такое решение будет нетипичным.

#### Обход в обратном порядке

В данном случае алгоритм обрабатывает сначала левый дочерний узел вершины, затем правый и только после саму вершину. Как и в предыдущих обходах, алгоритм начнет рассматривать дерево, изображенное на рисунке 10.5, с корня, переместит- ся в левую часть к дочернему узлу *B*, а через него к левому дочернему узлу *A*. Не найдя дальнейших потомков, он выведет *A*, вернется к родительской вершине *B* и перейдет к правому дочернему узлу *C*. У него тоже нет отходящих ветвей, поэто- му программа выведет саму вершину *C* и снова обратится к родительскому узлу *B*. Поскольку работа с дочерними вершинами на этом уровне закончена, алгоритм выведет *B*, поднимется к корню *D* и проследует к правому дочернему узлу *E*. Не обнаружив связанных с ним потомков, он выведет *E* и вернется к *D*. Раз изучение дочерних узлов окончено, остается вывести *D* и завершить обход. В целом порядок следования алгоритма будет таким: *A*, *C*, *B*, *E*, *D*.

В следующем псевдокоде представлена рекурсивная реализация процесса.

TraversePostorder(BinaryNode: node)

If (node.LeftChild != null) Then TraversePostorder(node.LeftChild) If (node.RightChild != null) Then TraversePostorder(node.RightChild)

<Обрабатываем вершину.> End TraversePostorder

Алгоритм обрабатывает правую и левую дочерние узлы данной вершины, если они существуют, а затем саму вершину.

Чтобы обойти все дерево, программа вызывает TraversePostorder, переда- вая ему в качестве параметра корень. В следующем псевдокоде алгоритм внедрен внутрь класса BinaryNode.

Class BinaryNode String: Name

BinaryNode: LeftChild BinaryNode: RightChild Constructor(String: name)

Name = name End Constructor

TraverseInorder()

If (LeftChild != null) Then TraversePostorder(LeftChild) If (RightChild != null) Then TraversePostorder(RightChild)

<Обрабатываем вершину.> End TraversePostorder

End Class

Обход в обратном порядке также легко применить к деревьям со степенью больше 2: алгоритм должен пройти по всем дочерним узлам вершины перед тем, как обратиться к ней самой.

#### Обход в ширину

Совершая обход в ширину, алгоритм обрабатывает все вершины дерева на те- кущем уровне в порядке слева направо, а затем переходит к вершинам следующего уровня. В случае с деревом, изображенном на рисунке 10.5, программа первым делом обратится к корню и выведет *D*, потом перейдет к следующему уровню и выведет вершины *B* и *E*, а на завершающем этапе опустится на нижний уровень и выведет *A* и *С*. Полный обход будет выглядеть так: *D*, *B*, *E*, *A*, *C*.

Данный алгоритм не придерживается структуры дерева, в отличие от алгорит- мов, рассмотренных выше. В используемом примере нет дочерней ссылки от вер- шины *E* к вершине *A*, поэтому неясно, как алгоритм к ней перейдет. Чтобы решить эту задачу, нужно добавить дочерние узлы в очередь, а затем обработать ее, когда завершится обработка родительского уровня.

TraverseDepthFirst(BinaryNode: root)

// Создаем очередь для хранения дочерних вершин при дальнейшей обработке. Queue<BinaryNode>: children = New Queue<BinaryNode>()

// Помещаем корень в очередь. children.Enqueue(root)

// Обрабатываем очередь, пока она не станет пустой. While (children Is Not Empty)

// Получаем следующую вершину в очереди. BinaryNode: node = children.Dequeue()

// Обрабатываем вершину.

<Обрабатываем вершину.>

// Добавляем дочернюю вершину в очередь.

If (node.LeftChild != null) children.Enqueue(node.LeftChild) If (node.RightChild != null) children.Enqueue(node.RightChild)

End While

End TraverseDepthFirst

Сперва алгоритм создает очередь и размещает в ней корневую вершину. Затем начинается цикл, который работает до тех пор, пока очередь не опустеет. Внутри него программа удаляет из очереди первую вершину, обрабатывает ее, а после до- бавляет в очередь дочерние узлы. В таком случае все элементы рассматриваются в порядке «первым пришел, первым вышел», то есть вершины текущего уровня будут обработаны полностью, перед тем как начнется обработка любых дочерних узлов. Поскольку алгоритм ставит в очередь сразу левый дочерний узел вершины, а затем правый, на определенном уровне все вершины обрабатываются в порядке слева направо. (Этот факт можно доказать с помощью индукции.)

#### Время выполнения обхода

Алгоритмы, выполняющие обход дерева симметрично, в прямом и обратном порядках, следуют вниз до терминальных вершин, возвращаясь вверх к корню по мере разворачивания рекурсивных вызовов. После того как алгоритм прошел по вершине и вернулся к родителю, он больше к ней не обращается, то есть посещает ее всего один раз. Таким образом, для дерева из *N* вершин время работы этих ал- горитмов составит *O*(*N*).

Упомянутые три обхода не требуют дополнительного пространства на диске, поскольку они используют структуру дерева для отслеживания своего местополо- жения в нем. Тем не менее их глубина рекурсии равна высоте дерева, и если дерево окажется очень высоким, вероятно переполнение стека.

При обходе в ширину вершины обрабатываются по мере их постановки в оче- редь. Поскольку каждая вершина помещается туда единожды, для дерева из *N* вершин время работы алгоритма будет равным *O*(*N*).

Этот алгоритм не использует рекурсию, а значит, не зависит от ее глубины, но нуждается в дополнительном пространстве для построения очереди. В наихудшем случае, если дерево является идеальным бинарным, то есть на его нижнем уровне находится практически половина всех вершин (см. раздел «Свойства бинарного дерева» текущей главы), в очереди окажется *O*(*N*/2) = *O*(*N*) вершин.

В самом общем смысле можно представить, что дерево произвольного уровня со- стоит из одного корневого узла, а все другие являются дочерними. Тогда очередь об- разуется из *N* – 1 вершин, но требование к пространству остается прежним — *O*(*N*).

## Упорядоченные деревья

Как отмечалось ранее, вершины упорядоченных деревьев (рис. 10.6) структу- рированы таким образом, что симметричный обход обрабатывает их в отсортиро- ванном порядке. Другими словами, значение каждого родительского узла больше значения его левого дочернего узла и меньше правого (или равно ему).



5

2

6

1

4

9

3

7

10

8

**Рис. 10.6.** Упорядоченное дерево

Чтобы работать с упорядоченным деревом, вам понадобятся три алгоритма для добавления, удаления и нахождения его вершин.

#### Добавление вершин

Построить упорядоченное дерево просто. Чтобы добавить дочернее значение к вершине, сравните его со значением родителя и рекурсивно переместитесь вниз по нужной ветке (левой или правой). Если таковой ветки нет, она появится вместе с новым значением.

В следующем псевдокоде представлен алгоритм для класса BinaryNode. Пред- полагается, что данные из вершины хранятся в свойстве Value.

// Добавляем вершину к упорядоченному поддереву вершины. AddNode(Data: new\_value)

// Сравниваем новое значение со значением вершины. If (new\_value < Value) Then

// Если новое значение меньше, размещаем его в левом поддереве. If (LeftChild == null) LeftChild = New BinaryNode(new\_value) Else LeftChild.AddNode(new\_value)

Else

// Если новое значение не меньше, размещаем его в правом поддереве.

If (RightChild == null) RightChild = New BinaryNode(new\_value) Else RightChild.AddNode(new\_value)

End If End AddNode

Алгоритм сравнивает новое значение со значением вершины. Если первое меньше, программа перемещается в левое поддерево. Ссылка на левый дочерний узел, равная null, говорит о том, что левый дочерний узел текущей вершины нужно создать и поместить в него новое значение. В противном случае програм- ма рекурсивно вызывает метод дочерней вершины AddNode и дополняет левое поддерево.

Если новое значение не меньше значения вершины, алгоритм переходит в пра- вое поддерево. Ссылка на правый дочерний узел, равная null, указывает программе на то, что его нужно создать и разместить в нем новое значение. В противном случае алгоритм рекурсивно вызывает метод дочерней вершины AddNode и достраивает правое поддерево.

**ЗАМЕЧАНИЕ**

Как и в случае со связными списками (см. главу 3), в верхней части дерева по- лезно использовать ограничитель. Если для упорядоченного дерева значение корня сделать меньше любого возможного значения, то, добавляя вершины, уже не надо проверять, пустое ли дерево. Все новые узлы будут заканчивать правое поддерево под корнем.

Время работы приведенного алгоритма зависит от порядка, в котором добав- ляются вершины. Если это происходит произвольно (без предварительной сорти- ровки), дерево вырастает относительно низким и широким. В таком случае при *N* вершинах высота дерева составит *O*(log *N*). Добавляя новую вершину, необхо- димо провести поиск до самого низа структуры, то есть совершить *O*(log *N*) шагов. При *N* вершинах полное построение дерева займет *O*(*N* log *N*).

**ВЫСОТА СЛУЧАЙНОГО ДЕРЕВА**

При построении красивого широкого упорядоченного дерева высотой *O*(log *N*) не всегда заметно, что к нему добавляется *O*(*N*) вершин. Но как быть, если дерево низкое и половина вершин уже сосредоточена в верхней части, а вам нужно разместить оставшиеся? Вспомните раздел «Свойства бинарного де- рева» текущей главы. В нем говорилось, что в идеальном бинарном дереве почти половина всех вершин находится в нижней части. Значит, когда в дереве задействована уже половина имеющихся вершин, в нем построены все уровни, кроме последнего. Таким образом, высота дерева составляет log *N* – 1. Теперь нужно добавить оставшиеся вершины на глубину log *N* – 1. Отсюда общее ко- личество шагов *N*/2  log *N* – 1 = *O*(*N* log *N*).

Если вершины предварительно отсортировать, то выстроенное на их основе дерево получится высоким и тонким. В худшем случае, если все значения изна- чально будут находиться в прямом или обратном порядке, каждая вершина при- обретет один дочерний узел и сформируется структура из *N* вершин высотой *N*. Тогда, добавив к дереву *N*/2 вершин, мы получим высоту *N*/2. Это означает, что размещение оставшихся *N*/2 вершин потребует *N*/2 шагов, а общее время работы алгоритма составит *O*(*N*/2  *N*/2) = *O*(*N*2).

Алгоритм AddNode можно использовать и для пирамидальной сортировки. Сформировав на его основе упорядоченное дерево, используйте симметричный обход для выведения элементов в сортированном порядке. Ожидаемое время ра- боты алгоритма составит *O*(*N* log *N*), симметричный обход в свою очередь зай- мет *O*(*N*). Таким образом, общая производительность будет найдена по формуле *O*(*N* log *N* + *N*) = *O*(*N* log *N*). В худшем случае построение упорядоченного дерева потребует *O*(*N*2) времени, а симметричный обход — *O*(*N*), тогда общее время со- ставит *O*(*N*2 + *N*) = *O*(*N*2).

#### Поиск вершин

Построив упорядоченное дерево, вы можете задействовать его для поиска кон- кретных элементов. Например, если вершины представляют собой записи о сотруд- никах, то используемыми для сортировки значениями станут их идентификационные номера. В следующем псевдокоде рассматривается метод, предоставленный классом BinaryNode, который проводит поиск целевого значения в поддереве вершины.

// Находим вершину с целевым значением. BinaryNode: FindNode(Key: target)

// Если мы нашли целевое значение, возвращаем эту вершину. If (target == Value) Then Return <текущая вершина>

// Смотрим, где целевое значение — в левом или правом поддереве. If (target < Value) Then

// Ищем в левом поддереве.

If (LeftChild == null) Then Return null Return LeftChild.FindNode(target)

Else

End If

// Ищем в правом поддереве.

If (RightChild == null) Then Return null Return RightChild.FindNode(target)

End FindNode

Сначала алгоритм проверяет текущее значение вершины и, если оно равно целевому, возвращает его. Обнаружив, что целевое значение меньше текущего, программа приходит к выводу, что нужная вершина располагается в левом под- дереве. Если левая дочерняя ветвь отсутствует, алгоритм возвращает null — целевого элемента в дереве нет. В противном случае рекурсивно вызывается метод FindNode, чтобы изучить поддерево. В ситуации когда целевое значение больше текущего, аналогичные шаги поиска предпринимаются по правому поддереву.

Если в дереве *N* вершин и оно сбалансированное (не высокое и не тонкое), его высота составляет *O*(log *N*). Тогда максимальное количество шагов для поиска — *O*(log *N*).

#### Удаление вершин

Убрать вершину из упорядоченного дерева несколько сложнее, чем в него до- бавить, ведь первым делом ее нужно найти (о том, как это сделать, рассказано в предыдущем подразделе). Следующий шаг будет зависеть от того, где целевая вершина находится. Чтобы разобраться с различными ситуациями, рассмотрим дерево, изображенное на рисунке 10.7.



60

35

70

21

42

71

89

17

24

68

11

23

63

69

**Рис. 10.7.** Упорядоченное бинарное дерево

Терминальную вершину можно удалить не задумываясь — дерево все равно останется упорядоченным. Таков, например, узел 89 в дереве на рисунке 10.7; его удаление не нарушит общего порядка организации данных (рис. 10.8).

Если целевая вершина не является терминальной и у нее есть всего один до- черний узел, ее можно заменить этим дочерним узлом. Например, если удалить вершину 71 из дерева на рисунке 10.8, получится структура, изображенная на рисунке 10.9.



60

35

76

21

42

71

17

24

68

11

23

63

69

**Рис. 10.8.** Упорядоченное бинарное дерево после удаления терминальной вершины



60

35

76

21 42

68

17 24

63

69

11

23

**Рис. 10.9.** Упорядоченное бинарное дерево

с удаленной внутренней вершиной, имевшей один дочерний узел

Самый сложный случай — когда у целевой вершины есть два дочерних узла. Общая стратегия предполагает поставить вместо родительской вершины левую до- чернюю. Но здесь возникают две ситуации. Если у левого дочернего узла нет право- го потомка, то данный узел можно передвинуть на место целевого. Например, уда- лив вершину 21 из дерева на рисунке 10.9, получим дерево, как на рисунке 10.10.



60

35

76

17 42

68

11 24

63

69

23

**Рис. 10.10.** Упорядоченное бинарное дерево с удаленной внутренней вершиной, где левый дочерний узел не имел правого потомка

Если же у левого дочернего узла целевой вершины есть правый потомок, при- дется спуститься от этого узла вниз до крайней правой вершины. В случае когда у нее нет своих дочерних узлов, именно она и заменит целевую. Удалось обнару- жить левый дочерний узел — поставьте его вместо крайней правой вершины, а ту в свою очередь вместо целевой.

Попробуем удалить вершину 35 из дерева на левом фрагменте рисунка 10.11. Как видно, у нее два дочерних узла, причем левый узел 17 имеет правого потомка. Алгоритм перемещается вниз от 17 настолько, насколько это возможно, следуя по правым дочерним ссылкам. В нашем примере конечной целью станет 24, но, обычно крайняя правая дочерняя вершина располагается гораздо глубже.



60

35

76

17 42

68

11 24

63 69

23



60

24

76

17 42

68

11 23

63 69

**Рис. 10.11.** Удаление из бинарного упорядоченного дерева внутренней вершины, у которой левый дочерний узел

имеет правого потомка *(слева)*, и результат этой операции *(справа)*

Поскольку у найденной вершины есть дочерняя, алгоритм выполняет следу- ющие перестановки: вместо крайней правого узла помещает дочерний, а вместо це- левого — крайний правый. Применительно к нашему случаю вершина 24 заменит- ся вершиной 23, а вершина 35 — вершиной 24 (см. рис. 10.11, правый фрагмент).

## Связные деревья

*Тред* — это серия ссылок, которая позволяет перемещаться по дереву или сети особым способом, отличным от привычного следования по ветвям и ссылкам. Де- рево, содержащее один тред или несколько, называется *связным*. Пример такой структуры изображен на рисунке 10.12, в ней каждая вершина связана тредами с вершинами, которые предшествуют ей или следуют за ней в симметричном об- ходе. Это позволяют алгоритму осуществлять симметричный или обратный обход быстрее, чем по ветвям.



5

2

6

**1**

4

9

7

**Рис. 10.12.** Связное дерево (пунктирными стрелками выделены треды)

**ЗАМЕЧАНИЕ**

 В связном дереве можно определить и другие треды, но представленные на рисунке 10.12 являются самыми распространенными. Поскольку они указывают

на прямое и обратное направления в симметричном обходе, такой вид дерева иногда называют *симметрично связным*.

**ЗАМЕЧАНИЕ**

Обратите внимание, что все вершины дерева на рисунке 10.12 имеют левую ветвь и левый тред или же правую ветвь и правый тред. Если вы можете как-то отличить ветвь и тред, то достаточно использовать всего одну ссылку. Напри- мер, создайте в классе вершины две булевские переменные HasLeftBranch и HasRightBranch и сохраняйте треды в дочерних ссылках, когда перемен- ные примут значение True. Можно даже уложить две булевские переменные в один байт и использовать операции с байтами, чтобы понять, установлены ли они. Однако если вы работаете с небольшим деревом, экономия памяти не оправдает подобных сложностей.

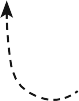
Чтобы использовать связные деревья, вам нужно знать две вещи: как их строить и как совершать по ним обход.

#### Построение связных деревьев

Изначально в связном дереве имеется всего одна вершина, нет ветвей, а треды установлены на null. Создать эту вершину легко, тяжелее добавить последующие. В зависимости от того, будут они левыми или правыми дочерними узлами, дей- ствуют разными способами.

Предположим, нужно создать левый дочерний узел. Пусть он имеет значение 3 и формируется для вершины 4 в дереве, изображенном на рисунке 10.12. Местопо- ложение новой вершины определяется путем сравнения со значением родительской вершины. В нашем примере число 3 — следующее наименьшее по отношению к чис- лу 4. Получается, что вершина, которая теперь стала в обходе перед новой вершиной, ранее располагалась перед родительской. Применительно к нашему случаю вершина, размещенная перед 3, находилась перед 4, то есть это 2. Выходит, что при добавлении нового узла его левый тред нужно приравнять к левому треду родителя.

Предшественник родителя в обходе — это теперь новая вершина, именно на нее указывает левая ветвь родителя. Таким образом, родительская вершина больше не нуждается в левом треде и его нужно приравнять к null. Правый тред новой вершины должен вести к следующей вершине в обходе дерева, в нашем случае расположения — к родительской. Отсюда вытекает, что правый тред вершины 3 обязан указывать на вершину 4 (рис. 10.13).



5

2

6

1

4

9

3

7

**Рис. 10.13.** Обновленное дерево с добавленной вершиной 3

При добавлении новой вершины в качестве правого дочернего узла шаги будут похожими, только левая и правая ветви и треды поменяются ролями. Правый тред новой вершины позаимствует значение у правого треда родителя (тот в свою оче- редь получит значение ), а левый тред станет указывать на родительскую вер- шину. На рисунке 10.14 изображено дерево с добавленным правым узлом. В сле- дующем псевдокоде представлен алгоритм вставки вершины в связное упорядо- ченное дерево.

null

// Добавляем вершину к упорядоченному поддереву данной вершины. AddNode(Data: new\_value)

// Сравниваем новое значение с имеющимся. If (new\_value < this.Value)

// Если новое значение меньше, добавляем его к левому поддереву. If (this.LeftChild != null)

Then this.LeftChild.AddNode(new\_value) Else

Else

End If

// Добавляем новую дочернюю вершину сюда. ThreadedNode child = new ThreadedNode(new\_value) child.LeftThread = this.LeftThread child.RightThread = this

this.LeftChild = child this.LeftThread = null

// Если новое значение не меньше,

// добавляем его к правому поддереву. If (this.RightChild != null)

Then this.RightChild.AddNode(new\_value) Else

End If End AddNode

End If

// Добавляем новую дочернюю вершину сюда. ThreadedNode child = new ThreadedNode(new\_value) child.LeftThread = this

child.RightThread = this.RightThread this.RightChild = child this.RightThread = null



**5**

2

6

1

4

9

3

7

8

**Рис. 10.14.** Обновленное дерево с добавленной вершиной 8

В начале работы алгоритм сравнивает новое значение со значением текущей вершины. Если первое меньше, то программа обращается к левому поддереву и при наличии у вершины левого дочернего узла рекурсивно вызывает метод AddNode. Когда левого дочернего узла нет, алгоритм создает новую вершину, при этом при- равнивает ее левый тред к левому треду текущей вершины, а правый ведет к текущей вершине, затем у текущей вершины направляет левую ветвь на новую вершину, левый же тред приравнивает к null. Если новое значение больше текущего, про- делываются похожие шаги, но только в правом поддереве.

Приведенный алгоритм очень похож на рассмотренный ранее, который добавлял вершины к упорядоченному дереву. Оба рекурсивно спускаются по дереву, чтобы найти местоположение новой вершины. Разница лишь в том, что в последней версии программы нужно осуществить дополнительные действия по установке тредов.

Как и в предыдущем случае, построение связного упорядоченного дерева из *N* вершин займет *O*(*N*  log *N*) времени, если значения добавляются произвольно. В худшей ситуации, когда значения отсортированы в прямом или обратном поряд- ке, производительность алгоритма составит *O*(*N*2).

#### Использование связных деревьев

В следующем псевдокоде треды используются для осуществления симметрич- ного обхода.

InorderWithThreads(BinaryNode: root)

// Начинаем с корня. BinaryNode: node = root

// Запоминаем, по ветке или по треду мы идем.

// Предположим, что мы дошли до корня по ветке, поэтому идем налево. Boolean: via\_branch = True

// Повторяем до конца обхода. While (node != null)

// Если мы дошли до этой вершины по ветке, идем вниз налево. If (via\_branch) Then

While (node.LeftChild != null) node = node.LeftChild

End While

End If

// Обрабатываем данную вершину.

<Обрабатываем вершину.>

// Находим следующую вершину для обработки. If (node.RightChild == null) Then

// Используем тред. node = node.RightThread via\_branch = False

Else

End If End While

// Используем правую ветку. node = node.RightChild via\_branch = True

End InorderWithThreads

В начале работы алгоритм инициализирует переменную node для корневой вершины и устанавливает переменную via\_branch на True как указатель на то, что он добрался до текущего места по ветке. Это заставляет программу обратиться к крайнему левому узлу дерева в следующем шаге. Далее идет цикл, который про- должается, пока переменная node не выпадет из дерева в конце обхода.

Добравшись до текущей вершины, алгоритм не обязательно приступает к ее об- работке. Если он обнаруживает исходящую левую ветвь, то приходит к выводу, что расположенные ниже в поддереве узлы имеют значения, меньшие текущего, и их нужно рассмотреть в первую очередь. По этой причине программа перемещается вниз по ветви настолько далеко, насколько возможно, и лишь затем обрабатывает

текущую вершину (например, в дереве, изображенном на рисунке 10.12, она про- следует от вершины 6 к вершине 9 только после того, как рассмотрит 7).

Далее, если у вершины нет правой ветви, алгоритм следует по правому тре- ду. В случае когда правый тред равен null, вершина также приравнивается к null — и цикл While заканчивает свою работу. Затем алгоритм устанавливает via\_branch на False, чтобы показать, что он добрался до новой вершины по тре- ду, а не по ветви. (Для дерева на рисунке 10.12 подобное произойдет не единожды, например на пути из вершины 4 к 5. Поскольку via\_branch установлен на False, алгоритм будет следующей обрабатывать вершину 5.)

Если правая ветвь текущей вершины не null, алгоритм следует по ней и уста- навливает via\_branch на True, чтобы спуститься по поддереву данной вершины во время следующего прохождения через цикл While.

Ниже приводится описание шагов, которые предпринимает алгоритм для обхода дерева, изображенного на рисунке 10.12.

1. Начинаем работу у корня и устанавливаем переменную via\_branch на True.
2. Поскольку via\_branch равно True, следуем по левым ветвям к 2, а затем к 1. Обрабатываем вершину 1.
3. Движемся по правому треду к 2, а затем устанавливаем via\_branch на

False.

1. Поскольку via\_branch равно False, обрабатываем вершину 2.
2. Следуем по правой ветви к 4, а затем устанавливаем via\_branch на True.
3. Поскольку via\_branch равно True, пытаемся переместиться вниз по левым ветвям. Левой ветви нет, значит, остаемся у вершины 4 и обрабатываем ее.
4. Для начала следуем по правому треду к 5, а затем устанавливаем перемен- ную via\_branch на False.
5. Поскольку via\_branch равна False, обрабатываем вершину 5.
6. Движемся по правой ветви до 6, а затем устанавливаем via\_branch на True.
7. Поскольку via\_branch равна True, пытаемся переместиться вниз по левым ветвям. Левой ветви нет, значит, остаемся у вершины 6 и обрабатываем ее.
8. Следуем по правой ветви до 9, а затем устанавливаем переменную via\_ branch на True.
9. Поскольку via\_branch равно True, идем по правой ветви к 7 и обрабаты- ваем эту вершину.
10. Движемся по правому треду к 9, а затем устанавливаем переменную via\_ branch на False.
11. Поскольку via\_branch равно False, обрабатываем вершину 9.
12. Следуем по правому треду до значения null, а затем устанавливаем пере- менную via\_branch на False.
13. Теперь переменная node равна null, заканчиваем работу цикла While.

Приведенный алгоритм по-прежнему проходит по всем ветвям вершины и че- рез каждую вершину, поэтому обладает временем работы *O*(*N*), однако ему не тре- буется разрешение для рекурсивных вызовов и подъема по дочерним ветвям. Та- ким образом, он экономит немного времени по сравнению с обычным обходом, не использует рекурсию, а значит, не зависит от ее глубины. Кроме того, в отличие от обхода в ширину он не нуждается в дополнительном пространстве на диске.

## Специализированные алгоритмы

Программисты разработали множество алгоритмов для решения специфиче- ских задач на основе деревьев. Описать абсолютно все невозможно, но в следующих подразделах мы коснемся четырех из них, которые представляют особый интерес. Благодаря им вы изучите такие полезные технологии, как обновление дерева для включения новых данных, расчет рекурсивных выражений, а еще разделение гео- метрических областей.

В последнем подразделе описываются префиксные деревья, которые хорошо известны в науке об алгоритмах.

#### Игра «Животные»

Суть игры заключается в следующем: программа пытается угадать задуман- ное пользователем животное и при этом самообучается, то есть постепенно со- вершенствуется. Вся необходимая информация хранится в бинарном дереве, каждая вершина которого содержит вопрос, предполагающий ответ «да» или

«нет». Последовательный выбор одного из этих ответов ведет программу вниз по соответствующим ветвям до терминальной вершины, где и находится название животного. Если программа совершает ошибку, она просит пользователя ввести уточняющий вопрос, который позволил бы ей прийти к верному решению, затем добавляет его в новую внутреннюю вершину и создает для нее свои листья.

Предположим, алгоритм программы выглядит так, как на рисунке 10.15, и поль- зователь загадал змею. В таблице 10.2 приводятся список вопросов, которые задает программа, и полученные ответы пользователя.



Собака

Кошка

Птица

Рыба

Змея

Оно дышит в воде?

Оно покрыто чешуей?

Оно лает?

Это млекопитающее?

**Рис. 10.15.** Исходное дерево знаний для игры «Животные»

**Òàáëèöà 10.2.** Èãðà «Æèâîòíûå» ïûòàåòñÿ óãàäàòü çìåþ

|  |  |
| --- | --- |
| **Вопрос программы** | **Ответ пользователя** |
| Это млекопитающее? | Нет |
| Оно покрыто чешуей? | Да |
| Оно дышит в воде? | Нет |
| Это змея? | Да |

Если задуман жираф, вопросы и ответы прозвучат иначе (табл. 10.3), а програм- ма добавить в дерево уточняющую информацию и нужное животное (рис. 10.16).

**Òàáëèöà 10.3.** Èãðà «Æèâîòíûå» ïûòàåòñÿ óãàäàòü æèðàôà



Собака

Птица

Жираф

Кошка

Рыба

Змея

Оно дышит в воде?

Оно с длинной шеей?

Оно покрыто чешуей?

Оно лает?

Это млекопитающее?

|  |  |
| --- | --- |
| **Вопрос программы** | **Ответ пользователя** |
| Это млекопитающее? | Да |
| Оно лает? | Нет |
| Это кошка? | Нет |
| Какое животное вы загадали? | Жираф |
| Какой вопрос поможет отличить кошку от жирафа? | У него длинная шея? |
| Для жирафа ответ утвердительный? | Да |

**Рис. 10.16.** Дополненное дерево знаний для игры «Животные»

#### Расчет математических выражений

Деревья хороши для моделирования математических выражений. Для каждого опе- ратора создается внутренний код, а для каждого численного значения — терминальная вершина. Это значит, что единое сложное выражение разбивается на части, а получен- ные с их помощью результаты затем используются для поиска итогового значения.

Попробуем решить следующий пример: (6  14) / (9 + 12). Чтобы получить для него ответ, нужно сначала рассчитать 6  14 и 9 + 12, а затем разделить их ре- зультаты. Учитывая вышесказанное, моделирование дерева будет выглядеть так: необходимо выстроить поддеревья для нахождения частных выражений и потом соединить их с корневой вершиной, которая представляет собой комбинирующую операцию — в нашем случае деление (рис. 10.17).



/



+

6

14

9

12

**Рис. 10.17.** Дерево для расчета математического выражения (6  14) / (9 + 12)

Каждая внутренняя вершина образованного дерева имеет дочерние узлы, в ко- торых находятся операнды. Например, двоичные операторы сложения и деления (+ и /) должны комбинировать значения, расположенные в их левой и правой дочер- них вершинах. Терминальные узлы можно представить в виде специальных опе- раторов, преобразующих текстовое значение в числовое.

Единственное, что отсутствует в классе арифметической вершины, — это метод расчета. Ему следует изучить тип вершины и вернуть соответствующий результат. Например, в случае оператора сложения (+) метод должен рекурсивно рассчитать частные выражения и суммировать полученные значения.

В следующем псевдокоде приводится список, на основе которого определяется тип оператора: текстовое значение (например, 8), сложение, вычитание, умножение, деление или унарное отрицание (например, –5). При необходимости его можно дополнить и другими операторами, такими как квадратный корень, возведение в степень, синус, косинус и т. д.

Enumeration Operators Literal

Plus Minus Times Divide Negate

End Enumeration

Ниже рассматривается класс ExpressionNode, представляющий вершину в дереве математического выражения.

Class ExpressionNode Operators: Operator

ExpressionNode: LeftOperand, RightOperand String: LiteralText

// Рассчитываем выражение.

Float: Evaluate()

Case Operator

Literal:

Return Float.Parse(LiteralText)

Plus:

Minus:

Return LeftOperand.Evaluate() + RightOperand.Evaluate()

Return LeftOperand.Evaluate() - RightOperand.Evaluate()

Times:

Divide:

Negate: End Case

End Evaluate End ExpressionNode

Return LeftOperand.Evaluate() \* RightOperand.Evaluate()

Return LeftOperand.Evaluate() / RightOperand.Evaluate()

Return -LeftOperand.Evaluate()

В самом начале объявляются свойства класса:

* Operator — выбранный тип оператора из приведенного списка Operators;
* LeftOperand и RightOperand — ссылки на левые и правые дочерние узлы (если вершина содержит унарный оператор, например отрицание, то исполь- зуется только левый дочерний узел; если текстовое значение — не использу- ется ни один из дочерних узлов);
* LiteralText — это свойство используется только для вершин с текстовым значением, например 12.

Затем метод Evaluate совершает нужное действие, опираясь на свойство Operator. Например, если Operator равен Plus, он складывает значения лево- го и правого дочерних узлов, а потом возвращает сумму.

Таким образом, рассчитать все выражение на основе выстроенного дерева до- вольно легко — достаточно вызвать метод корневой вершины Evaluate. Сложно сформировать само дерево на основе строки — в нашем случае это (6  14) / (9 + 12). О том, как это сделать, читайте в главе 15, посвященной строкам.

#### Деревья квадрантов

*Деревья квадрантов*, или *квадродеревья*, — это древовидные структуры дан- ных, которые помогают находить объекты в двумерном пространстве. Предпо- ложим, есть приложение, которое отображает на карте несколько тысяч объек- тов. Как только пользователь щелкнет в нужном ему месте, программа должна провести полноценный поиск по объектам и найти ближайший к указанному пункту назначения. Если объекты хранятся в простом списке, программа будет вынуждена последовательно проверить каждый из них, чтобы определить под- ходящий.

Дерево квадрантов значительно ускоряет подобные операции. В нем каждая вершина — это прямоугольник в двумерном пространстве, который содержит список элементов. Если элементов слишком много, они перераспределяются по четырем дочерним вершинам, представляющим собой северо-западный, северо-восточный, юго-восточный и юго-западный квадранты родительской вершины.

Первым делом элемент с заданными координатами *X* и *Y* ищется в корневой вершине. Если у дерева квадрантов есть дочерние вершины, то по координатам элемента определяется нужная, а затем рекурсивный поиск ведется уже в ней. Дойдя до вершины, которая не имеет дочерних узлов, алгоритм пересматривает ее элементы, пытаясь найти целевой.

Предположим, что вышеупомянутое приложение содержит 1500 объектов. Для линейного поиска по списку потребуется произвести приблизительно 750 сравнений. А теперь представьте, что вы храните информацию в дереве квадрантов, каждая вершина которого вмещает 100 элементов. Если вершины распределены по карте относительно равномерно, то получится следующая картина (рис. 10.18).



**Рис. 10.18.** Дерево квадрантов

с равномерно распределенными элементами

Корневая вершина разделена на четыре квадранта, каждый из которых вклю- чает еще четыре меньших квадранта. Терминальные вершины представляют собой наименьший квадрант, чья вместимость составляет 100 элементов.

Чтобы найти целевой элемент, нужно определить сначала бо'льший квадрант, в котором он содержится, затем внутри него более мелкий квадрант и, наконец, провести поиск по элементам терминальной вершины. Результатом станет среднее арифметическое из двух сравнений квадрантов и около 50 сравнений элементов. Относительная скорость сравнений квадрантов и элементов может меняться в за- висимости от поставленной задачи, но обычно она намного быстрее, чем 750 срав- нений из простого списка.

Если дерево квадрантов содержит *N* вершин, в каждой из которых может нахо- дится до *K* элементов, относительно равномерно распределенных на одной площади, то его высота будет около log4 (*N*/*K*). В нашем примере *N* = 1500, а *K* = 100, значит,

высота дерева, изображенного на рисунке 10.8, равна log4 (1500/100) = log4 15 �

� 1,95, что близко к 2.

На рисунке 10.19 представлена визуализация дерева квадрантов из 200 элемен- тов, каждая вершина которого включает около 10 элементов (на практике последнее число может быть больше, чтобы деление происходило не так часто). Обозначен- ные рамки показывают, как расчерчена область в целом: карта разбита на четыре основных квадранта, в каждом из которых выделяются четыре квадранта поменьше, затем некоторые из них дробятся еще раз и т. д.

Для работы с деревом квадрантов вам понадобятся алгоритмы, которые умеют находить нужный элемент или добавлять его к вершине поддерева. Кроме того, может пригодится алгоритм для нанесения элементов на карту.

В следующем псевдокоде представлено базовое определение класса для дерева квадрантов.

Class QuadtreeNode

// Максимальное количество элементов в одной вершине дерева квадрантов. Integer: MaxItems = 10

// Элементы данной вершины дерева квадрантов. List Of Data: Items

// Область, которую представляет данная вершина дерева квадрантов. Rectangle: Area

// Значения срединных координат X и Y. Float: Xmid, Ymid

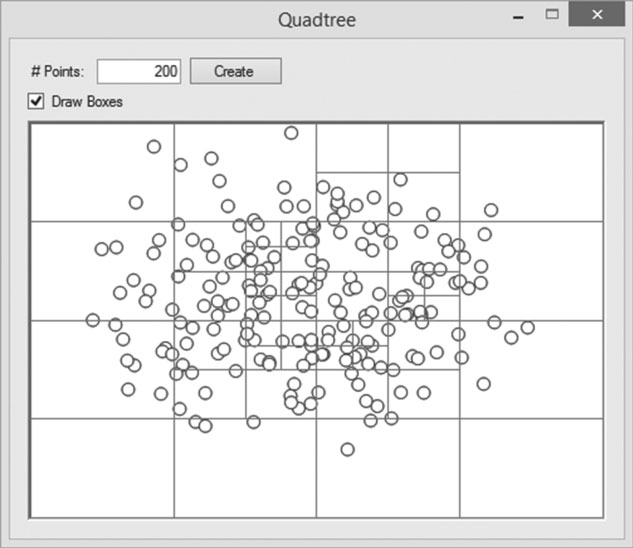
// Дочерние вершины дерева квадрантов. QuadtreeNode: NWchild, NEchild, SEchild, SWchild

// Инициализация конструктора. Constructor(Rectangle: area)

Area = area

Xmid = (Area.Left + Area.Right) / 2 Ymid = (Area.Top + Area.Bottom) / 2

End Constructor End QuadtreeNode



**Рис. 10.19.** Дерево квадрантов из 200 элементов

Значение MaxItems указывает на максимальное количество элементов, кото- рые может содержать вершина, не разбиваясь на квадранты.

Свойство Items — это элементы вершины. Если вершина внутренняя, то Items

равно null, а элементы хранятся в дочерних поддеревьях.

Свойство Area — область, которую представляет вершина в двумерном про- странстве. Значения Xmid и Ymid — срединные координаты *X* и *Y*; они используются для определения нужного квадранта.

В классе есть конструктор, инициализирующий свойство вершины Area, а так- же Xmid и Ymid. По сути, последние два значения можно рассчитать в любой мо- мент, но поскольку они часто используются (по крайней мере для нетерминаль- ных вершин), инициализация позволяет сэкономить время.

Ниже в псевдокоде QuadtreeNode в поддерево добавляется новый элемент.

// Добавляем элемент в поддерево вершины. AddItem(Item: new\_item)

// Проверяем, полная ли вершина.

If ((Items != null) And (Items.Count + 1 > MaxItems)) Then

// Разбиваем данную вершину.

Float: wid = (Area.Right - Area.Left) / 2 Float: hgt = (Area.Bottom - Area.Top) / 2 NWchild = New QuadtreeNode(

New Rectangle(Area.Left, Area.Top, wid, hgt)) NEchild = New QuadtreeNode(

New Rectangle(Area.Left + wid, Area.Top, wid, hgt)) SEchild = New QuadtreeNode(

New Rectangle(Area.Left + wid, Area.Top + hgt, wid, hgt)) SWchild = New QuadtreeNode(

New Rectangle(Area.Left, Area.Top + hgt, wid, hgt))

// Сдвигаем точки в подходящее поддерево. For Each item In Items

If (item.Y < Ymid) Then

If (item.X < Xmid) Then NWchild.AddItem(item) Else NEchild.AddItem(item)

Else

End If Next item

If (item.X < Xmid) SWchild.AddItem(item) Else SEchild.AddItem(item)

// Удаляем список Items данной вершины.

Items = null

End If // Заканчиваем, если вершина полная.

// Добавляем элемент в подходящее поддерево. If (Items != null) ThenItems.Add(new\_item) Else If (new\_item.Y < Ymid) Then

If (new\_item.X < Xmid) Then NWchild.AddItem(new\_item) Else NEchild.AddItem(new\_item)

End AddItem

Else

End If

If (new\_item.X < Xmid) Then SWchild.AddItem(new\_item) Else SEchild.AddItem(new\_item)

Если текущая вершина терминальная и добавление еще одного элемента приво- дит к превышению их допустимого количества, алгоритм разбивает вершину на че- тыре дочерние. Затем он циклически проходит по элементам и, рекурсивно вызывая метод AddItem, перемещает их в соответствующее поддерево. После этого список Items устанавливается на null, чтобы указать, что вершина является внутренней.

Если свойство Items не равно null, это говорит о том, что вершина терми- нальная, и алгоритм добавляет новый элемент к списку Items. В противном случае алгоритм выполняет ряд тестов If-Then, чтобы определить, в каком поддереве со-

держится элемент, а затем рекурсивно вызывает соответствующий метод дочерней вершины AddItem.

В следующем высокоуровневом псевдокоде показано, как QuadtreeNode на- ходит элемент в данной точке и возвращает ответ с помощью параметра result. Значение True сообщает о том, что целевой элемент найден, False — о том, что целевого элемента нет.

// Ищем элемент в данной точке. Возвращаем true, если находим его. Boolean: FindItem(Point: target, Item: result)

// Проверяем, есть ли дочерние вершины. If (Items == null) Then

// Дочерние вершины есть, ищем подходящее поддерево. If (target.Y < Ymid) Then

If (new\_item.X < Xmid) Then

Return NWchild.FindItem(target, result) Else Return NEchild.FindItem(target, result)

Else

Else

End If

If (new\_item.X < Xmid) Then

Return SWchild.FindItem(target, result) Else Return SEchild.FindItem(target, result))

// Дочерних вершин нет, ищем элементы текущей вершины. For Each item In Items

// Проверяем, тот ли это элемент.

If ((item.X == target.X) And (item.Y == target.Y)) Then result = item

Return True

End If Next item

// Элемент не найден. Return False

End If

End FindItem

Сначала алгоритм проверяет свойство Items, чтобы выяснить, имеет ли теку- щая вершина дочерние узлы. Если таковые есть, он определяет, в каком поддереве содержится элемент, и рекурсивно вызывает метод FindItem соответствующей дочерней вершины. В противном случае выполняется поиск целевого элемента по списку вершины Items. Не обнаружив элемент в указанном месте, программа возвращает False.

Рассмотренный код кажется относительно простым, но на практике все оказы- вается сложнее. Хранящиеся в дереве элементы отличны от обычных окружностей, прямоугольников или сегментов линий. Отсюда возникает вопрос: к какой из двух из вершин дерева квадрантов отнести элемент, расположенный на границе? Как вариант, можно поместить его в обе вершины, чтобы поиск оказался удачным независимо от того, на какой площади щелкнет пользователь. Но в этом случае необходимо изменить алгоритмы таким образом, чтобы они могли взаимодейство- вать с двумерными элементами. Например, при поиске элемента нужно не просто сравнивать целевую точку с местом, в котором он находится, но и разработать

определенный метод, который позволит выяснить, располагается ли целевая точка внутри элемента.

Подобное решение приводит к дублированию элементов — они представляют один и тот же объект в различных вершинах дерева квадрантов. Это в свою очередь провоцирует нецелесообразное использование пространства. Кроме всего прочего, вершины дерева квадрантов заполняются гораздо быстрее, чем при другом способе, поэтому должны разделяться чаще, как следствие — дерево становится более глубоким.

Еще одно возможное решение — представить каждый элемент с помощью осо- бой точки, расположенной в центре или в верхнем левом углу. Тогда при поиске элемента анализируются вершины дерева квадрантов с областями, частично по- крывающими область вокруг целевой точки, которая достаточно велика, чтобы включать в себя наибольший элемент.

Например, изображенные на рисунке 10.19 элементы — это окружности с радиу- сом 5, которые представлены центральными точками. При поиске элемента в точке (*A*, *B*) программа проверяет любое дерево квадрантов с областью, пересекающей прямоугольник *A* – 5  *X*  *A* + 5 и *B* – 5  *Y*  *B* + 5.

Все описанные изменения внести в алгоритм несложно, правда, они удлинят код.

**ВОСЬМЕРИЧНЫЕ ДЕРЕВЬЯ**

Октодерево подобно квадродереву, за исключением одного — в нем хра- нятся трехмерные объекты. Восьмеричная вершина также трехмерна. Когда в ней накапливается много элементов, она делится на восемь октантов по числу дочерних вершин, а элементы распределяются между поддеревьями.

#### Префиксные деревья

*Префиксное дерево*, или *луч*, — это дерево, содержащее строки. Каждая вну- тренняя вершина представляет собой одну букву, терминальная — часто несколь- ко. Путь от корня до терминальной вершины соответствует строке.

Частичный путь от корня к внутренней вершине создает префикс для более длин- ных путей, поэтому лучи часто называют *префиксными деревьями*. Путь, представля- ющий собой строку ключа, имеет присваиваемое значение, независимо от того, закан- чивается он у внутренней или у терминальной вершины. На рисунке 10.20 изображе- но префиксное дерево, содержащее ключи и значения, перечисленные в таблице 10.4.



*A*

*N*

*W*

*E*

29

*T*

*ISP*

36

72

**Рис. 10.20.** Префиксное дерево: путь через него определяет строку

**Òàáëèöà 10.4.** Äàííûå äëÿ òèïîâîãî ïðåôèêñíîãî äåðåâà

|  |  |
| --- | --- |
| **Ключ** | **Значение** |
| *WANE* | 29 |
| *WISP* | 72 |
| *WANT* | 36 |

Рассмотрим путь от корня до вершины *E*. Пройденные узлы соответствуют бук- вам *W*, *A*, *N* и *E*, поэтому вершина представляет собой ключ *WANE*, значение кото- рого равно 29. Путь к вершине *T* ведет через буквы *W*, *A*, *N* и *T*, отсюда ключ *WANT* со значением 36. А вот путь к вершине *N* создает строку *WAN*, которая является префиксом для *WANE* и *WANT*.

Обратите внимание, что терминальная вершина может содержать более одной буквы, например в вершине *ISP* их сразу три. Путь от корня до данной вершины представляет собой ключ *WISP*, имеющий значение 72.

Для добавления нового ключа к префиксному дереву используйте его буквы, чтобы пройти по соответствующему пути. Если вы доберетесь до терминальной вершины, а в ключе все еще будут содержаться буквы, добавьте новую дочернюю вершину — она представит оставшуюся часть пути. Поясним на примере. Пред- положим, нужно добавить к префиксному дереву ключ *WANTED*. Пройдя через вершины *W*, *A*, *N* и *T*, вы обнаружите, что у вас еще остались буквы *ED*. Все, что вам нужно сделать, — добавить новую вершину, в которой они будут храниться (рис. 10.21).



*A*

*N*

*W*

*ISP*

*E* 29 *T*

36

*ED*

10

72

**Рис. 10.21.** Префиксное дерево с добавленной терминальной вершиной

Иногда новый ключ в префиксном дереве легко найти. Например, на рисун- ке 10.21 уже присутствуют все необходимые вершины для представления ключа *WAN*. В таком случае достаточно добавить значение к соответствующей вершине, как показано на рисунке 10.22.



*A*

*W*

*N*

18

*ISP*

*E* 29 *T*

36

*ED*

10

72

**Рис. 10.22.** Префиксное дерево с добавленным значением

Букву можно не хранить во внутренней вершине, а определить по пути, прой- денному до нее от корня. Представьте себе, например, массив с дочерними верши- нами, где Children[0] — это ветвь для буквы *A*, Children[1] — ветвь для бук- вы *B* и т. д.

На рисунке 10.23 изображено видоизмененное префиксное дерево с рисун- ка 10.22, у которого буквы из внутренних вершин перемещены на ветви. Обратите внимание, что для вершины со значением 29 не нужна дополнительная информа- ция, поскольку ключ, который она представляет, полностью определяется путем, и напротив, путь к вершине со значением 10 задействует только буквы *W*, *A*, *N*, *T* и *E*, поэтому необходимо сохранить конечную *D*.



*N*

*E*

29

18

*T*



*D E*

36 10

72



*A*

*W*

*I*

*SP*

**Рис. 10.23.** Префиксное дерево,

в котором буквы располагаются не в вершинах, а на ветвях

В следующем псевдокоде показано, как добавить элемент к префиксному де- реву. Под фразой «оставшийся ключ вершины» подразумевается та часть клю- ча, которая хранится в терминальной вершине, подобно *D* и *SP* на рисунке 10.23.

AddValue(string new\_key, string new\_value)

<Если new\_key не пуст и соответствует оставшемуся ключу вершины, помещаем значение в эту вершину и останавливаемся.>

<Если new\_key пуст, так же как и оставшийся ключ вершины, помещаем сюда значение и останавливаемся.>

<Если new\_key пуст, а оставшийся ключ вершины нет,

переносим последний (за исключением первой буквы) в дочернюю вершину, помещаем сюда значение и останавливаемся.>

// Если мы дошли до этой строки, нам нужна дочерняя вершина. If <дочерний массив обнулен> Then

// Создаем массив Children. Children = New TrieNode[26]

<Если оставшийся ключ вершины не пуст, переходим к соответствующей дочерней вершине.>

End If

// Конвертируем букву в целое число, не считая А. Integer: index = new\_key[0] - ‘A’

// Ищем соответствующее поддерево. If (Children[index] == null)

// Этой дочерней вершины не существует, поэтому создаем ее.

// Она будет представлять оставшуюся часть нового ключа. Children[index] = New TrieNode() Children[index].RemainingKey = new\_key.Substring(1) Children[index].Value = new\_value

Return

End If

// Ищем соответствующее поддерево. Children[index].AddValue(new\_key.Substring(1), new\_value)

End AddValue

Алгоритм довольно сложный. Его легче понять, если начертить дерево и вслед за кодом провести в нем необходимые обновления.

Перемещаясь по дереву, алгоритм удаляет буквы из нового ключа в со- ответствии с ветвями, которые пересекает. Затем рассматривается текущее значение нового ключа. Если оно не пустое и совпадает с ключом вершины, новое значение размещается в текущей вершине. На рисунке 10.23 данное по- ложение совпадает с установкой значения *WANTED*. Когда алгоритм дойдет до последней вершины, новым значением ключа станет *D*, а это и есть оставшийся ключ вершины.

Если новый ключ и оставшийся ключ вершины пусты, алгоритм размещает значение в текущей вершине. Таким образом на рисунке 10.23 устанавливается значение *WAN*. Когда алгоритм пересекает ветвь *N*, новый ключ представляет собой пустую строку. У вершины в конце ветви нет оставшегося ключа (он мо- жет быть только у терминальных вершин), поэтому здесь должно разместиться значение *WAN*.

Если пуст только новый ключ, но не оставшийся ключ вершины, тогда алгоритм перемещает последний в дочерний узел. Так происходит, когда префиксное дерево содержит *WANE* и *WANTED*, но не *WANT*. В этом случае путь для *WANTED* будет выглядеть как *W*, *A*, *N*, *T, ED*. При добавлении *WANT* и пресечении ветви *T* новое значение ключа пусто, поскольку путь представляет полностью новый ключ *WANT*. Тем не менее данный ключ имеет значение *ED*, поэтому алгоритм перемещает ED вниз к дочерней вершине, а затем создает новую ветвь *E* и новую вершину с *D* в качестве оставшегося ключа.

Совершив все предыдущие шаги, программа должна переместиться в поддерево дочерней вершины. Перед тем как это сделать, она определяет, был ли инициали- зирован массив вершины Children. Если нет, то создается массив Children. В случае когда не пуст еще и оставшийся ключ вершины, алгоритм перемещает его (без первой буквы) в соответствующую дочернюю вершину.

Затем проверяется дочерняя вершина, которая должна содержать новый ключ. Если ее не существует, она создается и в нее сохраняется оставшаяся часть нового ключа и новое значение. Если дочерняя вершина есть, алгоритм рекурсивно вы- зывает сам себя, чтобы добавить новый ключ (без первой буквы) к поддереву этой вершины.

Поиск значения в префиксном дереве осуществляется по тому же принципу, но с использованием более простого алгоритма, поскольку здесь меньше особых случаев.

// Находим значение в поддереве вершины. Data: FindValue(String: target\_key)

// Если целевой ключ соответствует оставшемуся ключу вершины,

// возвращаем значение этой вершины.

If (target\_key == RemainingKey) Then Return Value

// Ищем соответствующую дочернюю вершину. If (Children == null) Then Return null Integer: index = target\_key[0] - ‘A’

If (Children[index] == null) Then Return null

Return Children[index].FindValue(target\_key.Substring(1)) End FindValue

Сначала алгоритм сравнивает целевой ключ с оставшимся ключом вершины. Если они одинаковы, возможны два варианта. Согласно первому, программа могла истощить целевой ключ и дойти до вершины, не имеющей оставшегося значения (как при поиске *WAN* на рисунке 10.22). В соответствии со вторым код дошел до вершины, на которой целевое значение подходит к оставшемуся ключу (как при поиске *WANTED* на рисунке 10.22). В обоих случаях код соответствует целевому ключу, поэтому алгоритм возвращает его значение.

Если оставшийся целевой ключ не подходит к оставшемуся ключу вершины, алгоритм выполняет поиск в дочерних узлах. Когда у текущей вершины их нет, это значит, что целевого ключа в префиксном дереве не существует и алгоритм возвра- щает null. В противном случае алгоритм рассчитывает индекс дочерней вершины целевого ключа и, не обнаружив ее, также делает вывод, что ключа в префиксном дереве нет, и возвращает null. Если же дочерняя вершина целевого ключа есть, то программа рекурсивно вызывает себя, чтобы эта вершина нашла целевой ключ (без первой буквы).

## Резюме

Деревья позволяют хранить данные в иерархическом порядке и облегчают работу с ними. Выстроив единожды подобную структуру, вы можете затем пере- бирать ее значения разными способами и осуществлять поиск нужного элемента.

Производительность данных алгоритмов очень часто зависит от высоты дере- ва. Если последнее содержит *N* вершин и является относительно коротким и широ- ким, то его высота составит *O*(log *N*), а алгоритм окажется относительно быстрым. Высокое и тонкое дерево высотой *O*(*N*) даст плохую производительность. Напри- мер, построение упорядоченного бинарного дерева в лучшем случае займет время *O*(*N* log *N*), а в худшем — *O*(*N*2).

Приняв во внимание вышесказанную зависимость, программисты разработа- ли специальные деревья, которые самостоятельно исправляют свою структуру, чтобы не расти слишком высокими или слишком тонкими. В следующей главе описано несколько видов таких деревьев, включая *B*-деревья и *B+*-деревья, используемые во многих системах баз данных для хранения и эффективного поиска информации.

## Упражнения

Звездочками отмечены задачи повышенной сложности.

1. Может ли идеальное бинарное дереве содержать четное количество вершин?
2. Идеальное дерево считается полным и завершенным, но не все полные и за- вершенные деревья идеальны. Начертите дерево, которое является полным и завершенным, но не идеальным.
3. Докажите, что количество ветвей *B* в бинарном дереве из *N* вершин вычис- ляется по формуле *B* = *N* – 1. Используйте индукцию.
4. Докажите факт из упражнения 3 без использования индукции.

5\*. Докажите, что количество терминальных вершин *L* в идеальном двоичном дереве высотой *H* определяется выражением *L* = 2*H*. Используйте индукцию.

6\*\*. Докажите, что в двоичном дереве из *N* вершин количество недостающих вет- вей *M* (мест, куда может быть добавлена дочерняя вершина) находится по формуле *M* = *N* + 1. Используйте индукцию.

1. Как будет выглядеть обход в прямом порядке для дерева, изображенного на рисунке 10.24?



*E*

*B*

*F*

*A*

*D*

*I*

*C*

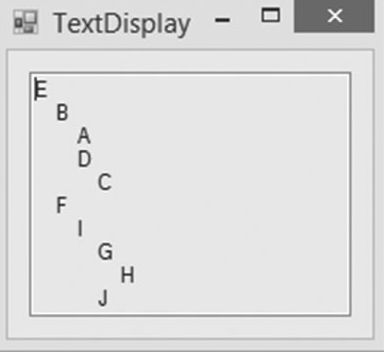
*G*

*J*

*H*

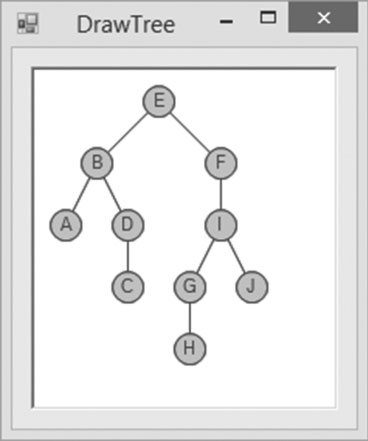
**Рис. 10.24.** Дерево значений

1. Совершите симметричный обход дерева, изображенного на рисунке 10.24.
2. Проанализируйте обход в обратном порядке для дерева, представленного на рисунке 10.24.
3. Каков обход в ширину для дерева, изображенного на рисунке 10.24?
4. Напишите программу, которая совершает все виды обхода из упражне- ний 7–10 для дерева, представленного на рисунке 10.24.
5. Что случится, если использовать стек вместо очереди в алгоритме обхода в ширину из подраздела «Обход в ширину» раздела «Обход дерева» текущей главы? Попробуйте сгенерировать такой обход рекусивно.
6. Напишите программу, подобную той, чье окно представлено на рисунке 10.25, где отображаются значения, полученные в результате прямого обхода дерева на рисунке 10.24.



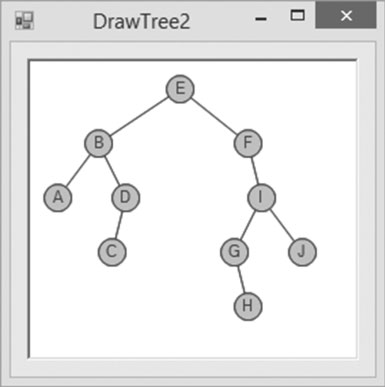
**Рис. 10.25.** Окно программы TextDispaly (значения дерева выведены наподобие иерархии папок в Ïðîâîäíèêå ОС Windows)

14\*\*. Напишите программу с более наглядным изображением дерева, подобную той, чье окно представлено на рисунке 10.26. (Подсказка: задайте метод клас- са для вершины PositionSubtree, чтобы позиционировать ее поддерево. Возьмите в качестве параметров координаты *x* и *y*, которые может занимать поддерево, и рассчитайте нужный для этого прямоугольник. Рекурсивным вызовом метода PositionSubtree определите размеры для возможных левых и правых дочерних поддеревьев, чтобы узнать размер исходного под- дерева. Задайте также методы класса, чтобы рекурсивно начертить ссылки и вершины дерева).



**Рис. 10.26.** Окно программы DrawTree

15\*\*. Изображенная на рисунке 10.26 структура годится для представления неупорядоченного дерева. В случае с упорядоченным бинарным по ней будет трудно определить разновидность дочернего узла — левый он или правый по отношению к родителю. Измените программу из упражне- ния 14 так, чтобы она формировала структуру, подобную той, что пред- ставлена на рисунке 10.27. Здесь на отсутствие дочернего узла указы- вает пустое место.

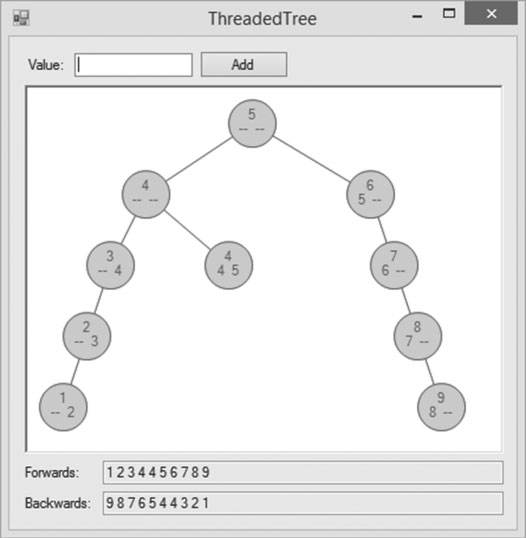


**Рис. 10.27.** Окно программы DrawTree2

16. Напишите псевдокод, выполняющий обратный симметричный обход в связ- ном сортированном дереве.

17\*. Напишите программу, которая создает связное упорядоченное дерево и со- вершает по нему симметричный и обратный симметричный обходы.

18\*\*. Усовершенствуйте программу из упражнения 17 так, чтобы она отображала дерево, подобное тому, что представлено на рисунке 10.28. В каждой окруж- ности содержится значение текущей вершины и тех вершин, к которым ведут ее треды. Например, левый тред вершины 4 установлен на null (обозначается как «- -»), а правый указывает на вершину 5.



**Рис. 10.28.** Окно программы ThreadedTree

1. Идентифицируйте вид дерева знаний из игры «Животные». Оно полное, за- вершенное, идеальное, представляет собой комбинацию всех трех видов или же не относится ни к одному из них?
2. В игре «Животные» для хранения информации может использоваться сле- дующий класс вершин.

Class AnimalNode String: Question

AnimalNode: YesChild, NoChild End Class

Если им воспользоваться, то как отличить, что содержит вершина — вопрос или животное?

1. Напишите программу, реализующую игру «Животные».
2. Начертите деревья для следующих математических выражений: 1) (15  3) + (24  6);

2) 8  12 – 14  32;

3) 1  2 + 1  4 + 1  20.

1. Напишите программу для расчета математических выражений. Поскольку информация о их разбиении рассматривается в главе 15, сделайте так, что- бы программа создавала математические выражения на основе примеров из упражнения 22.
2. Начертите деревья для следующих математических выражений: 1) (36  2)  (9  32);

2) 5!  ((5 – 3)!  3!);

3) sin2 (45°).

25\*. Усовершенствуйте программу из упражнения 23 для расчета выражений из упражнения 24.

26\*\*. Напишите программу, подобную той, что представлена на рисунке 10.19. Раз- решите пользователю выбирать нужную окружность кнопкой мыши (если щелчок выполняется за пределами окружностей, ни одна из них не должна быть выбрана). При воспроизведении карты выделите целевую окружность (если она есть) другим цветом.

1. Начертите префиксное дерево, представляющее ключи и значения из сле- дующей таблицы.

|  |  |  |  |
| --- | --- | --- | --- |
| **Ключ** | **Значение** | **Ключ** | **Значение** |
| *APPLE* | 10 | *ANT* | 40 |
| *APP* | 20 | *BAT* | 50 |
| *BEAR* | 30 | *APE* | 60 |

28\*\*. Напишите программу, которая добавляет и находит элементы в префиксном дереве.

# Глава 11 СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

В главе 10 уже приводилось общее описание деревьев и некоторых программ, в которых они используются. Вы уже знаете, что при совершении обхода время работы алгоритмов зависит от общего размера дерева, а при добавлении вершины к упорядоченному дереву — от его высоты. К тому же если упорядоченное дере- во из *N* вершин является относительно коротким и широким, для вставки новой вершины потребуется *O*(log *N*) шагов. Если же древовидная структура получает- ся высокой и тонкой (вершины добавляются в отсортированном порядке), число шагов рассчитывается как *O*(*N*), что намного больше, чем в предыдущем случае.

В текущей главе описываются *сбалансированные деревья*, то есть такие, кото- рые перестраивают свои вершины, чтобы не стать слишком высокими или тонки- ми. Подобные структуры данных могут быть не идеальными с точки зрения фор- мирования или обладать минимальной возможной высотой для определенного ко- личества вершин, но они достаточно оптимальны, чтобы проходящие по ним алго- ритмы имели время работы *O*(log *N*).

**ЗАМЕЧАНИЕ**

В текущей главе не используется псевдокод, поскольку работу алгоритмов со сбалансированными деревьями гораздо легче объяснить на основе рисунков.

В следующих разделах рассматриваются три вида сбалансированных структур данных: АВЛ-, 2-3- и *B*-деревья.

## АВЛ-деревья

*АВЛ-дерево* — упорядоченная бинарная структура, в любой вершине которой высота двух поддеревьев отличается максимум на единицу. При добавлении или удалении вершины дерево повторно балансируют по необходимости, чтобы со- хранить данное свойство.

**ЗАМЕЧАНИЕ**

АВЛ-деревья стали самыми первыми сбалансированными структурами данных. Они описаны еще в 1962 г. и названы в честь разработчиков Георгия Максимо- вича Адельсона-Вельского и Евгения Михайловича Ландиса.

АВЛ-дерево является упорядоченным бинарным, поэтому поиск целевых значе- ний в нем выполняется довольно легко (см. главу 10). Немного сложнее происходит их добавление и удаление. Об этих двух операциях мы и поговорим в следующих подразделах.

#### Добавление значений

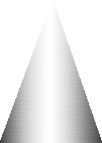
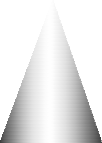
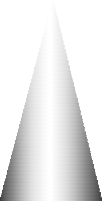
Рассматривая вершину АВЛ-дерева, пользуются показателем сбалансирован- ности. Он указывает на то, каковы относящиеся к ней поддеревья — тяжелые слева или справа либо же сбалансированные. Исходя из этого, показатель сбалансирован- ности можно определить как разность между высотами левого и правого поддере- вьев. Значение –1 скажет о том, что вершина утяжелена справа, 0 — что поддеревья равной высоты, +1 — что вершина утяжелена слева.

Добавление новой вершины к АВЛ-дереву происходит за счет рекурсивного спуска по дереву до места, где должна находиться новая вершина. По мере его вы- полнения программа обновляет показатели сбалансированности у каждой вершины, возвращаясь к корню. Если выявляется вершина, у которой показатель сбаланси- рованности меньше –1 или больше +1, программа совершает одно вращение или несколько, чтобы устранить дисбаланс.

В общем случае поведение алгоритма зависит от того, куда именно добавляется новая вершина. Возможны четыре случая:

* 1. левое поддерево левой дочерней вершины;
  2. левое поддерево правой дочерней вершины;
  3. правое поддерево левой дочерней вершины;
  4. правое поддерево правой дочерней вершины.

На рисунке 11.1 новое значение позиционируется в левое поддерево *А*1 левой дочерней вершины. Этот случай имеет особое название — *«левый — левый»*. Изо- браженные треугольники представляют собой сбалансированные АВЛ-поддеревья, которые могут содержать большое количество вершин. У вершины *В* дерево не сбалансировано, поскольку поддерево *А*1 в вершине *А* на два уровня выше, чем поддерево *В*2.



*B*

*A*

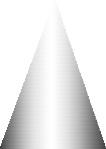
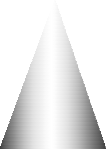
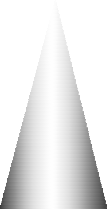
*B*2

*A*2

*A*1

**Рис. 11.1.** Случай «левый — левый»

Чтобы сбалансировать дерево, необходимо совершить *правое вращение*: заме- нить вершину *В* вершиной *А* и сделать поддерево *А*2 левым поддеревом вершины *В* (рис. 11.2).



*A*

*B*

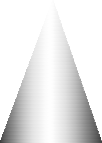
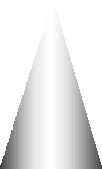
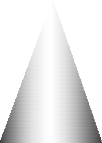
*A*1

*A*2

*B*2

**Рис. 11.2.** Сбалансированное дерево после правого вращения

Случай *«правый — правый»* похож на предыдущий, только дерево балансируется с помощью *левого вращения* (рис. 11.3).



*B*

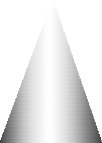
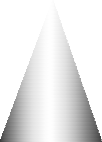
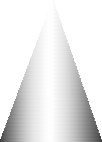
*A*

*A*1

*B*1

*B*2

**Рис. 11.3.** Случай «правый — правый» *(слева)*



*A*

*B*

*A*1

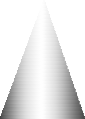
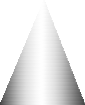
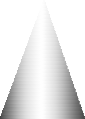
*B*1

*B*2

и сбалансированное дерево после левого вращения *(справа)*

В случае *«левый — правый»* (рис. 11.4, первый фрагмент) новое значение со- держится в правом поддереве левой дочерней вершины, которое в свою очередь состоит из вершины *А* и еще двух поддеревьев — *А*1 и *А*2. В каком из последних находится добавленная вершина, абсолютно неважно — поддерево с корнем *А* все равно на два уровня ниже, чем поддерево *В*2. Это значит, что поддерево с корнем *С* на два уровня глубже, чем поддерево *В*2, получается, дерево не сбалансировано у вершины *В*.

Чтобы сбалансировать дерево, нужно использовать вначале левое, а затем пра- вое вращение. На втором фрагменте рисунка 11.4 после перемены вершин *А* и *С* образуется случай «левый — левый»: поддерево левой дочерней вершины смещается на два уровня ниже правого поддерева с корнем *В*. Теперь можно сбалансировать всю структуру с помощью правого вращения (рис. 11.4, третий фрагмент).



*B*

*C*

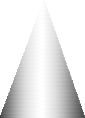
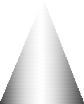
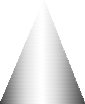
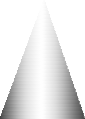
*A*

*B*2

*C*1

*A*1

*A*2



*B*

*A*

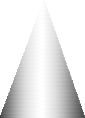
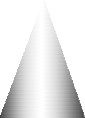
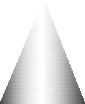
*C*

*B*2

*A*2

*C*1

*A*1



*A*

*C*

*B*

*C*1

*A*1

*A*2

*B*2

**Рис. 11.4.** Случай «левый — правый» *(сверху)*

и дерево после левого *(посередине)* и правого вращений *(снизу)*

Подобным образом балансируются деревья и в случае «правый — левый», толь- ко вначале используется правое вращение (чтобы выйти на случай «правый — пра- вый»), а затем левое (чтобы сбалансировать все дерево).

#### Удаление значений

Вращения помогают не только добавлять вершины в АВЛ-дерево, но и удалять их. На первом фрагменте рисунка 11.5 показана исходная структура данных. Уда- ление вершины 1 приводит к тому, что изображенное на втором фрагменте дерево становится несбалансированным у вершины 3: ее левое поддерево имеет высоту 1,

а правое — 3. Чтобы сбалансировать структуру, нужно применить левое вращение (рис. 11.5, третий фрагмент).



3

2

5

1

4

7

6

8



3

2

5

4

7

6

8



5

3

7

2

4

6

8

**Рис. 11.5.** Удаление значения: исходное *(вверху)*, несбалансированное *(посередине)* и сбалансированное дерево

Во всех случаях максимальная высота АВЛ-дерева из *N* вершин равна *O*(log *N*), потому оно является относительно коротким и широким. Это значит, что операции, в ходе которых происходит подъем по дереву (например, поиск значения), требуют *O*(log *N*) времени. Столько же занимает и балансировка. Таким образом, добавление или удаление значения происходит за время *O*(log *N*).

## 2-3-деревья

Чтобы АВЛ-дерево оставалось сбалансированным, его структура должна быть масштабной — поддеревья при любой вершине отличаются друг от друга максимум на один уровень по высоте. Для определения этой высоты вам пришлось бы про- смотреть все поддерево вплоть до терминальной вершины.

Сбалансированное *2-3-дерево* не столь масштабно, поэтому можно не проводить полный анализ поддеревьев у данной вершины, а лишь рассмотреть количество

имеющихся у нее дочерних узлов. Согласно названию, их бывает два или три. Вершина с двумя дочерними узлами называется *2-вершиной*, с тремя — *3-вершиной*.

Если представить, что каждая внутренняя вершина имеет минимум два до- черних узла, то максимальная высота дерева из *N* вершин составит log2 *N*. В этом случае структура данных подобна обычному бинарному дереву, где в каждой

вершине содержится значение. Поиск нужного значения проводится сверху вниз по левой или правой ветви — в зависимости от того, меньше оно или больше значения вершины.

В вершинах с тремя дочерними узлами находятся два значения. При поиске спускаются по левой ветви дерева, если искомое значение меньше первого значения вершины, по средней — если оно расположено между первым и вторым значением или по правой ветви — если оно больше второго значения.

На практике и тот и другой вид вершины можно представить одним и тем же классом или структурой. Достаточно лишь указать количество возможных дочерних узлов, а затем добавить свойство, которое покажет, сколько значений используются. (У терминальной вершины может быть одно или два значения, но нет дочерних узлов.)

Чтобы найти значение 76 в 2-3-дереве, изображенном на рисунке 11.6, требуется сравнить его со значением корня 42. Поскольку искомое число больше, нам нужно двигаться по правой дочерней ветви. У следующей вершины значение 76 сравни- вается с 69 и 81, и раз оно находится между ними, то дальше надо спуститься по средней ветви. Там, в терминальной вершине, и завершается поиск.



42

27 41

69 81

11

38 40 50

76

94

**Рис. 11.6.** 2-3-дерево с двумя и тремя дочерними узлами при внутренних вершинах

Поиск значений в 2-3-дереве осуществляется просто, чего не скажешь об их добавлении и удалении. Последние две операции немного сложнее, чем при работе с обычным бинарным деревом.

#### Добавление значений

Чтобы вставить новое значение, как и в случае с упорядоченным деревом, не- обходимо найти ту терминальную вершину, к которой это значение относится. В зависимости от того, заполнена ли вершина, действуют по-разному.

Если в терминальной вершине менее двух значений, новое добавляется с со- хранением принятого порядка. Когда терминальная вершина заполнена (у нее уже есть два значения), действуют следующим образом: меньшее значение переносят в новую левую вершину, большее — в новую правую, а среднее перемещают вверх к родительской вершине. Это называется *разделением вершины*.

На рисунке 11.7 к исходному дереву добавляется значение 42. Сравнив его с корнем 27, приходим к выводу, что нам нужна нижняя правая ветвь. Вершина под этой ветвью терминальная (то есть в идеале в ней и должно находиться новое значение) и к тому же полная — к 32 и 57 необходимо добавить 42. Чтобы освобо- дить для нового значения место, узел следует разделить: меньшее и большее числа (32 и 57) остаются в терминальных вершинах, а среднее (47) перемещается вверх к родительской вершине.



27

19

32 57



27 42

19

32

57



42

**Рис. 11.7.** Добавление нового значения к полной терминальной вершине: исходное *(слева)* и дополненное деревья *(справа)*

При разделении узла может получиться так, что в родительской вершине ока- жется слишком много значений и ее также придется делить. В худшем случае такую операцию понадобится повторить вверх по всему дереву, что спровоцирует *разде- ление корня* и сделает дерево большим. Однако это единственное стечение обстоя- тельств, при котором 2-3-дерево станет высоким.

Для упорядоченного бинарного дерева худший сценарий — добавление зна- чений в отсортированном порядке. Это приведет к тому, что дерево получится высоким и тонким. При *N* вершинах оно будет иметь высоту *N*.

На рисунке 11.8 изображено формирование 2-3-дерева, в которое добавляются числа от 1 до 7 в порядке их следования.



1 2



2

1

3 4



2

1

3



2 4

1

3

5

3 4 5 6



4

2

6

1

3

5

7

7



2 4

1

3

5 6

**Рис. 11.8.** Разделение корня 2-3-дерева

На первом фрагменте в терминальной вершине содержатся значения 1 и 2, далее показано, как происходит добавление остальных чисел и поиск их месторасполо- жения. Проанализировав рисунок, вы увидите, что добавление значения 4 привело к разделению вершины, а 7 — к разделению корня.

#### Удаление значений

Теоретически удаление значения из 2-3-дерева должно напоминать добавление, только с точностью до наоборот: вместо разделения вершины следует ожидать слияние вершин. Однако на практике все гораздо сложнее.

Чтобы упростить задачу, представим, будто значение удаляется из терминаль- ной вершины. Если оно находится не в терминальной вершине, то, как и в любом упорядоченном дереве, его следует заменить крайним правым значением, располо- женным слева. Заменяемая вершина будет находиться в терминальной, поэтому далее ситуацию можно рассматривать так, словно крайнее правое значение было удалено из левой вершины.

Вершина, в которой произведено удаление, остается с одним значением или вовсе без них. В первом случае задачу можно считать выполненной. Во втором нужно позаимствовать одно из двух значений у вершины-сестры и переместить его в пустую вершину.

Обратимся к рисунку 11.9. Предположим, вы хотите удалить значение 4 из корня дерева на верхнем фрагменте. Для начала переместите число 3 в удаленную позицию, чтобы получить дерево, изображенное на втором фрагменте.



4

2

6 8

1

3

5

7

9



3

*A*

*B*

2 6 8

1

5

7

9



3

2

6

8

1

5

7

9



6

2

8

1

3 5 7

9

**Рис. 11.9.** Удаление значения из вершины в 2-3-дереве

Новая структура больше не является 2-3-деревом, поскольку у внутренней вер- шины *А* есть только один дочерний узел. Зато ее сестра *В* содержит три дочерних узла, один из которых можно позаимствовать: сделайте вершину со значением 5 дочерней для вершины *А*, как на третьем фрагменте. На следующем этапе нужно удалить значение 6, которое помогало решить, когда перемещаться от вершины *В* к вершине с числом 5.

Теперь у значения 6 нет вершины и числа расположены не в отсортированном порядке. Поскольку 6 больше любого другого значения в поддереве *А*, его следует переместить в корень. Значение 3, которое до этого в нем находилось, больше всех исходных значений в поддереве *А* и меньше позаимствованного значения 5. Таким образом, его нужно расположить слева от 5. На четвертом фрагменте рисунка 11.9 представлен конечный результат операции.

Рассмотрим еще такую ситуацию, когда значение удаляется из внутренней вершины и у нее остается только один дочерний узел, а у вершины-сестры — всего одно значение. В таком случае у вершины-сестры нельзя позаимствовать вершину, но можно присоединить ее саму целиком. Это называется *слиянием вершин*.

При слиянии вершин у их родителя теряется один дочерний узел. Если до этого детей было двое, нарушается условие, согласно которому каждая внутренняя вер- шина в 2-3-дереве должна иметь два или три дочерних узла. В таком случае нужно перейти вверх по дереву и изменить баланс у родителя, перераспределив значения либо объединив родителя с его вершиной-сестрой.

В качестве примера возьмем дерево, изображенное на верхнем фрагменте рисун- ка 11.10, и предположим, что нам нужно удалить из него значение 3. В результате получится структура, как на втором фрагменте. Она уже не является 2-3-деревом, поскольку у внутренней вершины *А* имеется только один дочерний узел.



7 20

2

10

30

1

3

8

11

25

31



7 20

*A*

2

10

*B*

30

1

8

11

25

31



20

2 10

30

1

7 8 11

25

31

**Рис. 11.10.** Объединение вершин при удалении значения из 2-3-дерева

У вершины *В* тоже только два дочерних узла, поэтому позаимствовать у нее дочерний узел нельзя, но можно объединить ее с вершиной *А*. В результате у одной общей вершины появится два значения и три дочерних узла, что отлично впишет- ся в нашу пространственную структуру. Однако родительская вершина потеряет дочерний узел, а следовательно, и значение. Последнее можно переместить в под- дерево присоединенной вершины.

По окончании вышеперечисленных перестановок значения потеряют отсор- тированный порядок, следовательно, их придется переставить. Результирующее дерево, которое получится в ходе операции, изображено на третьем фрагменте рисунка 11.10.

В нашем примере верхняя вершина, где по итогу осталось значение 20, имеет два дочерних узла. Если бы подобная структура не образовалась, пришлось бы по- вторно балансировать дерево на уровне вершины, позаимствовать дочерний узел либо объединить текущую вершину с вершиной-сестрой.

В худшем случае слияния могут протянуться по всему дереву и коснуться корней. Однако это единственное стечение обстоятельств, в результате которого 2-3-дерево может вырасти более коротким.

## B-деревья

*B*-деревья являются расширением 2-3-деревьев, или наоборот, 2-3-деревья мож- но считать особым случаем *B*-деревьев.

В 2-3-дереве каждая внутренняя вершина содержит одно или два значения и имеет две либо три ветви. В *B*-дереве порядка *K* каждая внутренняя вершина (кроме, возможно, корневой) содержит от *K* до 2*K* значений и имеет от *K* + 1 до 2*K* + 1 ветвей.

Количество значений, определяющее *порядок дерева*, может быть огромно, поэтому внутренние вершины в *B*-дереве часто называют *ведрами*. *B*-дерево по- рядка *K* обладает следующими свойствами:

* каждая его вершина содержит максимум 2*K* значений и минимум *K* значе- ний (кроме, возможно, корневой вершины);
* внутренняя вершина, включающая *M* значений, имеет *M* + 1 ветвей, которые ведут к *M* + 1 дочерним вершинам;
* все терминальные вершины находятся на одном уровне.

Поскольку от каждой внутренней вершины отходит как минимум *M* + 1 ветвь, *B*-дерево не может вырасти слишком высоким или тонким. Например, в *B*-дереве девятого порядка у каждой внутренней вершины может быть 10 ветвей. Таким об- разом, дерево, содержащее 1 млн значений, должно иметь log10 1 000 000 = 6 уров-

ней в высоту. (Для полного бинарного дерева при том же количестве значений вы-

сота составит 20 уровней.)

**ЗАМЕЧАНИЕ**

*Степень дерева* — это максимальное количество ветвей, которым может рас- полагать любая его вершина, то есть *B*-дерево порядка *K* имеет степень 2*K* + 1.

Поиск по *B*-дереву проводится так же, как и по 2-3-дереву: рассматриваются значения каждой вершины, выясняется, между какими из них лежит целевое, а за- тем происходит спуск по соответствующей ветви.

На рисунке 11.11 показана вершина *B*-дерева второго порядка. Чтобы найти значение 35, нужно спуститься по ветви *B*, поскольку оно располагается между значениями 27 и 36. Если ведется поиск числа 50, превышающего все имеющиеся значения вершины, то надо пройти вниз по последней ветви *D*.



42

36

27

*А В С D*

**Рис. 11.11.** Пример *B*-дерева (внутренние вершины содержат несколько значений, между которыми располагаются ветви)

Добавление и удаление значений в *B*-дереве подобно аналогичным операциям в 2-3-дереве.

#### Добавление значений

Чтобы включить новое значение в *B*-дерево, нужно найти соответствующую терминальную вершину. Если в ней менее 2*K* значений, новый элемент просто займет положенное ему место. В случае когда текущая терминальная вершина заполнена, но ее сестра содержит менее 2*K* значений, добавляемый элемент по- мещается в вершину-сестру с учетом существующего порядка.

В качестве примера рассмотрим рисунок 11.12. Предположим, что вам необходи- мо добавить значение 17 в дерево, изображенное на первом фрагменте. Подходящая терминальная вершина уже заполнена, поэтому меняется порядок в вершине-сестре, как на втором фрагменте, и новое значение становится в нее. (Обратите внимание, что разделяющее значение в родительской вершине также изменилось.)



18

16

15

13

25

11

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 32 | 37 |  |  |

17



17

16

15

13

18

11

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 25 | 32 | 37 |  |

**Рис. 11.12.** Добавление значения к полной вершине *B*-дерева: исходная *(вверху)* и перераспределенная *(внизу)* структуры

Если все вершины-сестры заполнены (или вы не хотите менять в них порядок значений, что довольно сложно), то текущую терминальную вершину можно раз- делить на две по 2*K* значений в каждой. В этом случае, чтобы добавить новое зна- чение к уже имеющимся в вершине, необходимо переместить среднее значение вверх, чтобы оно стало разделяющим в родительском узле, а оставшиеся разнести по двум новым вершинам.

Предположим, вам нужно добавить значение 34 к дереву, изображенному на первом фрагменте рисунка 11.13. Терминальная вершина и ее сестры заполнены, поэтому перераспределить элементы нельзя, зато можно разделить саму терми- нальную вершину, как показано втором фрагменте.



|  |  |  |  |
| --- | --- | --- | --- |
|  | 11 | 25 |  |
| 12 | 14 | 15 | 18 |
|  |  |  |  |
|  | 11 | 25 | 37 |
| 12 | 14 | 15 | 18 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 7 | 8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 32 | 37 | 41 | 42 |

34



|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 7 | 8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 32 | 34 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 41 | 42 |  |  |

**Рис. 11.13.** Добавление значения к полной вершине *B*-дерева: исходная структура *(вверху)* и с разделенной терминальной вершиной *(внизу)*

Перемещая новое значение вверх, вы можете обнаружить, что и родительская вершина уже заполнена. Тогда необходимо повторить процесс — перераспреде- лить значения между сестрами родителя или же разделить его самого, передвинув значения еще выше.

В худшем случае процесс пройдет по всей структуре и спровоцирует разделе- ние корня, но это единственный вариант, когда *B*-дерево может вырасти высоким.

#### Удаление значений

Чтобы удалить значение из внутренней вершины *B*-дерева, поменяйте его ме- стами с крайним правым значением в левой части, как в случае с упорядоченными деревьями, а затем действуйте так, будто имеете дело с терминальной вершиной.

Если после проведенной операции терминальная вершина содержит минимум *K* значений, ваша задача выполнена. Когда значений меньше, необходимо повторно сбалансировать дерево: обратите внимание на сестер вершины, где аналогичный показатель больше *K*, и перераспределите значения нужным образом.

Предположим, требуется удалить значение 32 из дерева, представленного на первом фрагменте рисунка 11.14. В результате ваших действий останется терми- нальная вершина всего с одним значением, что неприемлемо для *B*-дерева второй

степени. Чтобы избежать подобной ситуации, значения в целевой вершине и ее сестрах были перераспределены, как на втором фрагменте, — в каждой из них осталось по два значения.



|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 7 |  |

|  |  |  |
| --- | --- | --- |
|  | 11 | 25 |
| 12 | 14 |  |
|  |  |  |
|  | 7 | 14 |
| 11 | 12 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 32 | 37 |  |  |

**Рис. 11.14.** Удаление значения из вершины *B*-дерева: исходная *(вверху)* и сбалансированная *(внизу)* структуры

Если в вершинах-сестрах не содержится более *K* значений, можно объединить одну из них с текущей вершиной, чтобы в ней оказалось 2*K* значений.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 25 | 37 |  |  |

Допустим, вы хотите удалить значение 12 из дерева на верхнем фрагменте ри- сунка 11.15. И терминальная вершина, и ее сестры содержат по *K* значений, поэтому перераспределение тут не поможет. Зато можно объединить терминальную вершину с одной из сестер, как показано во втором фрагменте.



|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 7 | 14 | 40 |
| 11 | 12 |  |  |
|  |  |  |  |
|  | 7 | 40 |  |
| 11 | 14 | 25 | 37 |

|  |  |  |  |
| --- | --- | --- | --- |
| 25 | 37 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 49 | 50 |  |  |

**Рис. 11.15.** Удаление значения из вершины *B*-дерева: исходная структура *(вверху)* и после объединения *(внизу)*

При слиянии двух вершин в их общем родителе может не оказаться *K* значе- ний, тогда процесс нужно повторить на более высоком уровне: перераспределить значения среди вершин-сестер или объединить одну из них с родителем.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 49 | 50 |  |  |

В худшем случае объединение пройдет по всему дереву и вызовет так назы- ваемое слияние корней, однако это единственный случай, когда *B*-дерево может вырасти коротким.

## Разновидности сбалансированных деревьев

Помимо уже описанных видов деревьев существуют еще некоторые модифика- ции. О двух из них, имеющих отношение к *B*-деревьям, мы поговорим в следующих подразделах. К слову, эти же модификации могут быть применимы и к другим видам сбалансированных структур, например к 2-3-деревьям (вы ведь помните, что это все те же *B*-деревья, только первого порядка).

#### Иерархически организованные B-деревья

При добавлении элемента к *B*-дереву сначала рекурсивно проходят по его вы- соте, чтобы найти нужную терминальную вершину. Если вершина полная, веро- ятно, ее понадобится разбить и перераспределить значение к родителю. В резуль- тате по мере возвращения рекурсивные вызовы могут то и дело добавлять пере- мещенные вверх значения к текущей вершине и, если она тоже разбивается, пере- ставлять уже другие элементы ближе к корню. Поскольку разделение ведер про- исходит в то время, как алгоритм движется вверх по структуре данных, подобное *B*-дерево иногда называют *восходящим*.

Альтернативная стратегия — сделать так, чтобы алгоритм разбивал любые полные вершины, спускаясь по дереву. Тогда в родительской вершине создается пространство, в которое при необходимости можно поднять «лишнее» значение. И если целевая терминальная вершина является полной, то алгоритм знает, что у ее родителя есть место (не было бы его, родительский узел был бы уже разбит). Поскольку такие разбивки ведра случаются при перемещении рекурсии вниз по структуре данных, подобное *B*-дерево иногда называют *нисходящим*.

В нисходящем *B*-дереве разбивки ведра происходят раньше, чем в любом другом случае. Полные вершины разделяются даже тогда, когда их дочерние узлы содер- жат много неиспользованных записей. А поскольку пустых записей в таком дереве больше, чем необходимо, то оно выше, чем могло бы быть восходящее *B*-дерево. Тем не менее свободное пространство сокращает вероятность того, что добавление нового значения повлечет многочисленные разбивки ведер.

К сожалению, для слияния ведер не существует нисходящего алгоритма. Спус- каясь по дереву, программа не может определить, потеряет ли вершина дочерний узел и нужно ли ее объединить с вершиной-сестрой.

#### B+-деревья

*B*-деревья часто используются для хранения объемных записей, например ин- формации о сотрудниках. Каждая такая запись, как правило, занимает несколько килобайтов памяти, а при наличии фотографии и несколько мегабайтов. *B*+-дерево организует данные с применением некоего ключевого значения. Им может стать *идентификационный номер* (ИН) сотрудника.

В этом случае перераспределение элементов в ведре сильно замедлится, по- скольку программе, скорее всего, придется иметь дело с большим объемом данных. Чтобы их переместить, понадобится каскадная разбивка ведра. Во избежание это- го во внутренних вершинах *B*-деревьев располагаются лишь ключевые значения и указатели на остальные данные записи. Теперь, чтобы изменить порядок ведер,

алгоритму необходимо переместить только их, а не целую запись. Такой тип струк- туры данных называется *B+-деревом*. Его пример показан на рисунке 11.16. Здесь пунктирные линии указывают на ссылки (указатели), которые ведут от ключа к соответствующим данным, заключенным в квадратные рамки.



12

27

4

9

16 19 20

29 30

30

29

20

19

16

9

4

|  |  |  |  |
| --- | --- | --- | --- |
| 12 | 27 |  |  |

**Рис. 11.16.** Пример *B*+-дерева

Помимо того что *B*+-деревья ускоряют перестраивание значений, они имеют еще несколько преимуществ. Благодаря им облегчается работа с разными ключа- ми для одних и тех же данных. Например, программа может использовать первое *B*+-дерево для систематизации записей о сотрудниках по их ИН, а второе — по номеру паспорта. При этом обращение к нужной записи происходит по одинаковым указателям. Так, чтобы найти сотрудника по ИН или номеру паспорта, нужно осу- ществить поиск по соответствующему дереву и перейти по правильному указателю к актуальным данным.

Другим преимуществом *B*+-деревьев является то, что их вершины могут со- держать множество величин. Это значит, вам по силам увеличить степень дерева и сделать его более коротким. Предположим, вы строите *B*-дерево второго порядка таким образом, чтобы каждая его вершина имела от 3 до 5 дочерних узлов. Что-

бы вместить 1 млн записей, понадобится структура высотой от log5 1 000 000 до log3 1 000 000 или от 9 до 13. Тогда для поиска нужного значения придется проверить 13 вершин. Маловероятно, что все записи дерева уместятся в памяти одновременно,

поэтому, скорее всего, потребуется 13 доступов к диску, что существенно замедлит процесс. А теперь представьте, что вы храните тот же 1 млн записей в *B*+-дереве, где вершины имеют одинаковый размер в килобайтах. Поскольку в этих вершинах содержатся только ключевые значения, их количество может быть огромно.

Возьмем, например, *B*+-дерево, где в одной вершине умещается до 20 ИН со- трудников (в действительности эта цифра может быть намного больше). В этом случае у каждого узла может быть от 11 до 21 дочерних узлов. Таким образом, дерево распределит уже упомянутый 1 млн значений по высоте от log21 1 000 000 до

log11 1 000 000 или от 5 до 6. Это значит, что для поиска нужного элемента программа

проверит максимум шесть вершин и совершит шесть доступов к диску, сократив

время операции приблизительно вдвое.

**ЗАМЕЧАНИЕ**

Поскольку *B+*-деревья сокращают время поиска при нескольких доступах к дис- ку, их часто используют в реляционных базах данных для хранения индексов.

## Резюме

Подобно другим упорядоченным структурам данных сбалансированные деревья позволяют программе хранить и быстро находить нужные значения. Организация элементов в АВЛ-, 2-3-, *B-* и *B+*-деревьях гарантирует, что такие деревья не вы- растут слишком высокими и тонкими, что может негативно сказаться на работе алгоритмов.

Добавление и удаление значений в сбалансированном дереве чуть более трудо- емко, чем в обычном (несбалансированном) дереве. Тем не менее такие операции занимают лишь *O*(log *N*) времени, то есть теоретически для обоих видов деревьев этот показатель почти одинаков. Дополнительное время, затраченное в действитель- ности, позволяет алгоритму гарантировать, что оба процесса не возрастут линейно.

В главе 8 описываются линейные хеш-таблицы, в которых можно хранить и ис- кать значения даже быстрее, чем в сбалансированных деревьях. Однако таблицы не обладают другим преимуществом, например они не позволяют быстро отображать все значения в отсортированном порядке.

На данном этапе мы рассмотрели типовые алгоритмы, которые позволяют лишь строить и обходить обычные и сбалансированные деревья. В следующей главе вы столкнетесь с моделированием и решением более широкого спектра задач.

## Упражнения

1. Начертите схему наподобие той, что представлена на рисунке 11.4, чтобы повторно сбалансировать АВЛ-дерево в случае «правый — левый».
2. Схематично изобразите добавление в АВЛ-дерево чисел от 1 до 8 в порядке их следования.
3. Сбалансируйте АВЛ-дерево, изображенное на рисунке 11.17, после удаления вершины 33.



33

12

47

5

17 35

65

8

57

71

**Рис. 11.17.** Пример АВЛ-дерева

1. По примеру рисунка 11.7 повторно сбалансируйте 2-3-дерево, изображенное на рисунке 11.18, добавив к нему значение 24.
2. Опираясь на рисунок 11.9, изобразите, как удалить значение 20 из 2-3-дерева, представленного на рисунке 11.18.
3. Начертите схему наподобие рисунка 11.13, чтобы добавить значение 56 к *B*-дереву, изображенному на рисунке 11.19.



30

10 20 37

5

13

22 23

**Рис. 11.18.** Пример 2-3-дерева



42

48

47

44

60

49

35

21

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 7 | 16 |

|  |  |  |  |
| --- | --- | --- | --- |
| 26 | 27 | 30 | 33 |

|  |  |  |  |
| --- | --- | --- | --- |
| 52 | 54 | 55 | 58 |

|  |  |  |  |
| --- | --- | --- | --- |
| 62 | 67 | 70 | 73 |

**Рис. 11.19.** Пример *B*-дерева

1. Возьмите за основу рисунок 11.14 и покажите, как удалить значение 49 из

*B*-дерева, полученного в предыдущем упражнении.

1. Схематично изобразите *B*-дерево второго порядка, к которому последова- тельно добавляются числа 1, 2, 3 и т. д. до тех пор, пока корневая вершина не получит четыре дочерних узла. Сколько величин окажется в дереве в этот момент?
2. Как правило, компьютеры считывают данные с жесткого диска блоками. Предположим, вам нужно построить *B*- или *B*+-дерево, в котором хранятся записи о клиентах в ведрах, состоящих из четырех блоков по 2 Мбайта. Пусть каждая запись имеет размер 1 Кбайт, ключевое значение — это фамилия клиента, занимающая 100 байт, а каждый указатель, находящийся между вершинами (или ведущий к данным в *B*+-дереве), требует для хранения 8 байт. Каков наивысший порядок для *B*- или *B*+-дерева при использовании ведер с четырьмя блоками? Какова наибольшая высота *B*- и *B*+-дерева, если в них хранятся 10 000 записей?

# Глава 12

**ДЕРЕВЬЯ ПРИНЯТИЯ РЕШЕНИЙ**

В главах 10 и 11 вы познакомились с алгоритмами, которые работают с дере- вьями в целом и со сбалансированными деревьями в частности. Вы узнали, как построить данного вида структуры и как работать с ними. Но пока неясно, как ис- пользовать деревья для решения конкретных задач.

В текущей главе описываются древовидные структуры, которые позволяют моделировать ситуации, где допустима множественность решений, а каждая ветвь представляет собой определенный выбор. В итоге каждая терминальная вершина содержит полный набор действий, приводящих к конечному результату, а целью становится поиск наилучшего возможного варианта или самой подходящей тер- минальной вершины в дереве.

Предположим, вам нужно разделить разные предметы на две группы так, чтобы вес каждой из них оказался одинаков. Задачу легко смоделировать с помощью би- нарного дерева: на уровне *K* в левой ветви предмет *K* включается в одну группу, а в правой — в другую. Прохождение по всему дереву соответствует полному распре- делению предметов. Ваша цель — найти путь, позволяющий ровно разделить вес.

Деревья принятия решений подходят для моделирования всех видов ситуаций, когда для получения результата нужно совершить несколько шагов. Единственный их недостаток заключается в том, что они вырастают огромными. Например, би- нарное дерево, которое делит вышеописанные *N* объекты на две группы, должно состоять из 2*N* терминальных вершин, а это может сделать поиск по всему дереву безуспешным. Например, для разделения 50 предметов понадобится древовидная структура из 1,13  1015 терминальных вершин. В этом случае, даже если проверять по 1 млн вершин в секунду, потребуется более 2100 лет.

В текущей главе рассказывается лишь о некоторых видах деревьев принятия решений. Из нее вы узнаете о методах эффективного поиска для больших задач, где нельзя перебрать все значения полностью, а также об эвристических методах нахождения приблизительных решений, когда изучение всего дерева невозможно. Для начала познакомимся с поиском по деревьям игры.

## Поиск по деревьям игры

Деревья игры хорошо подходят для моделирования игры в шахматы, шашки, го и крестики-нолики. В них каждая ветвь отражает действие одного из игроков, другими словами, если в какой-то момент у игрока появится 10 возможных ходов, то и у дерева будет 10 возможных ветвей. Полное следование по дереву соответ- ствует законченной игре.

Как и все остальные деревья принятия решений, эти структуры растут очень быстро. Например, если в игре в шахматы состоялось 40 ходов (оба игрока совер- шили по 20) и на каждом этапе возможно в среднем около 30 вариантов действий, то общее количество прохождений по дереву будет 3040 � 1,2  1059. Исчерпывающий компьютерный поиск, осуществляющий 1 млн прохождений в секунду, продлится около 2,3  1044 лет. (Об оценке неповторяющихся шахматных партий можно про- читать по ссылке ru.wikipedia.org/wiki/×èñëî\_Øåííîíà.)

Игра в крестики-нолики легче поддается решению. На первом шаге у крести- ков есть девять возможных вариантов выбора, на втором шаге у ноликов таких вариантов уже восемь. В дальнейшем у каждого участника становится на одно возможное действие меньше, чем у соперника на предыдущем шаге. Таким об- разом, общее количество прохождений в дереве игры вычисляется как 9  8  7   ...  1 = 9! = 362 880.

Некоторые из действий становятся недопустимыми. Например, когда крести- ки занимают три верхние клетки после первых пяти ходов — игра заканчивается. Значит, любые прохождения по дереву, начинающиеся с того, что крестики стоят в указанных клетках, не достигают девятого уровня. Если удалить все подобные прохождения, которые рано заканчиваются, в дереве так или иначе сохранится еще около 250 000 терминальных вершин, то есть структура по-прежнему останется относительно разветвленной.

В следующих подразделах описываются алгоритмические методы, которые го- дятся для сравнительно небольшой игры в крестики-нолики, но они также приме- нимы к любой похожей игре, например к шахматам или шашкам.

#### Минимакс

Чтобы выбрать предпочтительный ход, нужно оценить доступные позиции на доске. Если, разместив крестик в определенной клетке, вы выиграете, у этой позиции большое значение, и наоборот: если, совершив ход, позволите выиграть сопернику, стало быть, у нее малое значение. В различных играх на подобную ценность позиции влияют разные факторы, например лидируете ли вы в партии, какую часть доски занимают ваши фигуры, могут ли они представлять угрозу для фигур соперника и т. д. В крестиках-ноликах предполагается четыре воз- можные позиции:

* 4 — игра закончится победой данного игрока;
* 3 — неясно, чем завершится игра (победой, проигрышем или ничьей);
* 2 — игра закончится ничьей;
* 1 — игра завершится проигрышем данного игрока.

На рисунке 12.1 изображены все возможные позиции. В верхнем левом фраг- менте крестики выиграют на следующем шаге; в верхнем правом — проиграют, по- скольку нолики одержат победу независимо от того, каким будет очередной шаг. Нижний левый фрагмент демонстрирует неопределенную ситуацию при условии, что поиск ведется только на нескольких уровнях в дереве игры. Наконец, нижний правый приведет к ничьей, при этом абсолютно неважно, куда будет поставлен оче- редной крестик или нолик.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Выигрывают крестики Проигрывают крестики

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Неопределенный результат

Ничья

**Рис. 12.1.** Возможные позиции в игре крестики-нолики

Из приведенных позиций вытекают очевидные выводы.

* Если выигрывает первый игрок — второй проигрывает.
* Если для первого участника партия заканчивается ничьей, то она заканчи- вается ничьей и для второго.
* Если исход игры неизвестен для первого игрока, то он неизвестен и для вто- рого.

В сложных играх оценить позицию на доске иногда сложно, поскольку про- грамма не в состоянии провести тщательный поиск по дереву, чтобы просчитать все возможные варианты исхода партии. В таких случаях позициям назначаются приблизительные значения, чтобы программа могла выбрать наилучший вариант.

На достаточно быстром компьютере в игре крестики-нолики поиск можно провести по всему дереву, поэтому значение 3 не является обязательным. Оно приведено, скорее, для того, чтобы понять, как поступать в более сложных случаях. (Аналогичную ситуацию в крестиках-ноликах можно смоделировать, если ограни- чить поиск в дереве несколькими уровнями.)

**ЗАМЕЧАНИЕ**

Поскольку в игре в крестики-нолики поиск можно вести по всему дереву, оче- видно, что уже на первом шаге один из участников может ускорить свой вы- игрыш либо ничью. Если оба игрока полностью понимают алгоритм приня- тия решений, то результат, по сути, становится предсказуемым. Единствен- ный случай, при котором исход партии сомнителен, — если кто-то из сопер- ников совершит ошибку. В более сложных играх, например в шахматах, такая ситуация маловероятна. Здесь дерево принятия решений становится слиш- ком большим, чтобы его можно было целиком охватить, что, в свою очередь, делает игру интересной.

*Минимакс* — поисковая стратегия, при которой на каждом ходе игры пытают- ся минимизировать выгоду противника. Так, если есть два возможных хода, где первый позволяет выиграть вашему сопернику, а второй сулит проигрыш, следует предпочесть второй.

Ниже представлен высокоуровневый алгоритм минимакса.

// Находим лучший ход для player1.

Minimax(Board: board\_position, Move: best\_move, Value: best\_value, Player: player1, Player: player2, Integer: depth, Integer: max\_depth)

// Проверяем, не превышена ли максимально допустимая

// глубина рекурсии.

If (depth > max\_depth) Then

// Максимально допустимая глубина рекурсии превышена.

// Исход игры для данной позиции не определен. best\_value = Unknown

Return

End If

// Определяем ход, который минимизирует шансы player2. Value: lowest\_value = Infinity

Move: lowest\_move

For Each <возможный тестовый ход>

<Обновляем board\_position, чтобы сделать пробный ход.>

// Оцениваем данную позицию.

If <Это выигрыш, проигрыш или ничья.> Then

<Устанавливаем соответствующие lowest\_value и lowest\_move.>

Else

// Рекурсивно пробуем другие возможные ходы. Value: test\_value

Move: test\_move

Minimax(board\_position, test\_move, test\_value, player2, player1, depth, max\_depth)

// Проверяем, нашли ли мы худший ход для player2. If (test\_value < lowest\_value) Then

// Этот ход лучше. Сохраняем его. lowest\_value = test\_value lowest\_move = test\_move

End If

End If

<Восстанавливаем board\_position, чтобы отменить пробный ход.>

Next <возможный пробный ход>

// Сохраняем лучший ход. best\_move = lowest\_move

// Преобразуем значение доски для player2 в значение для player1. If (lowest\_value == Win)

best\_value = Loss

Else If (lowest\_value == Loss) best\_value = Win

End Minimax

Else End If

...

Алгоритм начинает свою работу с проверки глубины рекурсии. Если макси- мально допустимое значение превышено, то определить исход игры по данной позиции на доске невозможно. Тогда для best\_value возвращается значение Unknown.

Чтобы выбрать лучший ход для player1, программа должна найти наихудшую позицию доски для player2. Для отслеживания последней создаются специальные переменные. Так, lowest\_value принимается равным Infinity и заменяется любым найденным значением.

Затем алгоритм Minimax циклически пересматривает все шаги, которые может совершить player1 и рекурсивно вызывает сам себя, чтобы установить лучший ход для player2 вслед за тем, как player1 сделает тестовый. После возвращения рекурсивного вызова возможный лучший результат для player2 сравнивается со значением, сохраненным в lowest\_value. Если тестовое значение меньше, то алгоритм обновляет lowest\_value и lowest\_move. Таким образом, ему становится известно, что этот ход является предпочти- тельным для player1.

По окончании проверки всех допустимых тестовых ходов алгоритм знает, какой из них подойдет player1, чтобы поставить в самую невыгодную позицию player2. Он сохраняет этот ход, а затем конвертирует значение доски для player2 в зна- чение для player1 на основании того, что лучшая позиция заставит проиграть player2, но даст возможность выиграть player1, и наоборот.

Когда player2 не выигрывает и не проигрывает, становится непонятно, как проводить конвертацию значений. Но для игры в крестики-нолики результаты Unknown и Draw сулят обоим участникам одно и то же. Так, если позиция на до- ске приводит к ничьей для player2, то такой же результат ожидает и player1.

Для более сложной игры, например в шахматы, положение на доске может при- нимать значения от –100 до +100, где +100 означает победу, –100 — поражение. В этом случае позиция для player2 может быть просто отрицательным вариантом той же самой позиции для player1.

Побочный эффект стратегии минимакс проявляется в том, что она рассматри- вает все варианты решений с равным значением позиции одинаково предпочти- тельными. Чтобы понять, где кроется проблема, представьте, что игра близится к концу и проигрыш неизбежен независимо от предпринятых действий. В таком случае при поиске по дереву игры программа выберет первый же ход, поскольку любой из них приведет к одному и тому же результату. Иногда это может выглядеть крайне неразумно. Например, противник одержит победу уже на следующем шаге, хотя ее еще можно было отсрочить на два-три хода. Для человека логичнее было бы действовать иначе: затянуть игру в надежде, что противник допустит ошибку или не поймет, что партия почти закончилась. Кроме того, алгоритм может отдать предпочтение победе за шесть ходов, а не за два. С описанными проблемами ре- ально справиться, если в случае неизбежного поражения выбирать более длинные последовательности шагов, в случае победы — более короткие.

Чтобы выиграть в крестики-нолики, простой стратегии минимакс, как правило, достаточно. Однако для более сложных игр программа не может проводить поиск по всему дереву, для них также нужны другие стратегии, о которых пойдет речь в следующих подразделах.

#### Начальные ходы и реакции

Уменьшить размеры дерева игры помогает сохранение заранее рассчитанных начальных шагов и реакций. Если вы уже провели предварительный поиск воз- можного первого хода, то программа может сделать его сразу, а не тратить время на просчеты.

Следующий шаг выполняет пользователь, поэтому компьютеру придется по- дождать, пока не завершатся оба действия. Размер дерева в этот момент зависит от конкретных предпринимаемых ходов, однако он будет намного меньше, чем первоначальный. Так, вся игра в крестики-нолики содержит 255 168 возможных сценариев, но если крестик поставить в верхнюю левую клетку, а нолик — в верх- нюю среднюю, то число сценариев сократится до 2668. Этого много для пересчета вручную, но приемлемо для дерева и компьютерного поиска.

Когда пользователь выполняет ход в игре первым, дерево также сильно умень- шается. Если он, например, заполнит верхнюю левую клетку, то в оставшемся дереве окажется 27 732 возможных сценариев. Это намного больше, чем в случае второго хода, но меньше, чем в исходном дереве игры. Сократить полученное число поможет еще одно изменение.

Для постановки крестика в первом ходе есть девять возможных вариантов. Если предварительно просчитать все реакции на каждый из них, то можно за- ставить программу выбрать подходящую. Тогда вместо поиска по дереву игры из 27 732 возможных сценариев понадобится лишь найти реакцию.

Далее пользователь совершает очередной ход, и программе снова не нужно осуществлять поиск по дереву, пока не будут завершены все три шага: ход пользо- вателя, выбор предварительно рассчитанной реакции и еще один ход пользователя. Дерево игры становится еще меньше. Так, если первый крестик займет верхнюю левую клетку, нолик — верхнюю среднюю, а второй крестик — верхнюю правую, то останется лишь 592 возможных сценария. При большом желании вы даже можете просчитать их вручную.

В более сложных играх, таких как шахматы, дерево является бесконечно боль- шим и ликвидация нескольких верхних уровней не принесет особой пользы. Про- пуск трех ходов позволит сократить количество сценариев с 1,2  1059 до 4,5  1054, но число по-прежнему останется огромным для полного поиска. Тем не менее благодаря предварительному расчету ходов и реакций программа способна быстро совершать свои первые действия, экономить время для планирования последующих шагов и избегать заведомо невыгодных начальных положений.

#### Эвристика дерева игры

Для всех игр, кроме, пожалуй, самых простых, деревья получаются довольно большими и не позволяют вести полный поиск. Именно поэтому нет возможности узнать, какой конкретно ход приведет к лучшему решению. В таких ситуациях значение для шага определяют с помощью эвристики.

*Эвристика* — это алгоритм, который находит возможный хороший результат, но не гарантирует, что тот будет наилучшим. Она не осуществляет поиск по всему дереву игры, но помогает сформулировать правила для принятия решения — какие части дерева нужно пропустить, а какие заслуживают особого внимания.

Один из видов игровой эвристики — поиск шаблонных позиций на доске. На- пример, некоторые шахматные игроки пользуются негласным правилом: «Когда побеждаешь, меняй беспощадно». Это значит следующее: если у вас есть преимуще- ство и вы можете обменяться равнозначными фигурами — сделайте это. Подобным образом вам удастся укрепить свои шансы на победу и уменьшить дерево игры, что, в свою очередь, облегчит поиск по нему в будущем. К шаблонам также относятся длинная последовательность обменов, рокировка и ход, представляющий угрозу нескольким фигурам, королю или ферзю, вскрытый шах, превращение пешки и ее взятие на проходе и т. д.

Когда программа идентифицирует один из таких вариантов, она в состоянии изменить используемую стратегию для поиска по дереву. Например, если намеча- ется длинный ряд обменов, алгоритм может превысить принятую максимальную глубину рекурсии и просчитать обмен до конца, чтобы выяснить, приведет он к выигрышу партии или нет.

Другой вид эвристики — присвоить позициям определенные числовые значения и изменять их сумму для доски в целом на основе того, как расположены фигуры игрока или какие из них находятся под угрозой. Например, в игре в крестики- нолики каждую клетку можно пронумеровать, согласно количеству вероятных победных сценариев, в которых он фигурирует. Так, верхняя левая клетка получит значение 3, поскольку с ее помощью можно одержать победу тремя способами (рис. 12.2).

|  |  |  |
| --- | --- | --- |
| 3 | 2 | 3 |
| 2 | 4 | 2 |
| 3 | 2 | 3 |

**Рис. 12.2.** Эвристические значения клеток для игры в крестики-нолики

В шахматах три центральных квадрата доски самые важные, поэтому им следует придать наибольшее значение. Вам также понадобится ввести особые значения для квадратов, занятых фигурой или находящихся под угрозой. Обратите внимание и на то, что в большинстве игр со временем значимость позиций на доске меняется. Например, на ранних этапах шахматной партии важны четыре центральных ква- драта, а в самом конце определяющими являются те, благодаря которым игрок может поставить шах и мат.

**ЗАМЕЧАНИЕ**

 Весьма интересное упражнение для программиста — описать игру «Ревер- си». Ее дерево намного больше, чем для крестиков-ноликов, поэтому полный

поиск все равно невозможен, зато правила и фигуры проще тех, что приняты в шахматах, а шаблоны легче распознать. Используя только значения позиций на доске и поиск по дереву, можно создать довольно-таки мощную програм- му. Чтобы найти больше информации о «Реверси» (правила игры и использу- емые стратегии), пройдите по ссылке ru.wikipedia.org/wiki/Ðåâåðñè.

Еще больше об эвристике вы узнаете из подраздела «Эвристика дерева при- нятия решений» следующего раздела текущей главы.

## Поиск по деревьям принятия решений

В игре с помощью дерева выбирают наилучший ход. В общем случае это вы- глядит как поиск самого оптимального пути по дереву. Подобным образом можно смоделировать и некоторые другие процессы принятия решений.

Рассмотрим, например, задачу о разбиении, когда предметы с разным весом, ценой, значением или любыми другими отличительными характеристиками не- обходимо разделить на две группы, которые в сумме по представленному признаку окажутся одинаковыми. В некоторых случаях это легко. Так, если у вас есть четыре предмета, чьи весовые значения представлены цифрами 2, 4, 1 и 1, то решение ка- жется очевидным: вы можете разместить самый тяжелый предмет в первой группе, а остальные отправить во вторую. Не составит труда и задача, где фигурирует четное количество предметов с одинаковым весом — их просто следует разделить поровну.

Намного сложнее ситуация, когда имеется большое количество предметов с раз- личным весом. Тогда распределять их по группам удобнее с помощью бинарного дерева принятия решений, где на уровне *K* рассматривается судьба предмета *K*: левая ветвь относит его к первой группе, а правая — ко второй.

На рисунке 12.3 представлено полное дерево решений для задачи с четырьмя упомянутыми выше предметами весом 2, 4, 1 и 1 единицы. Путь по дереву — это окончательное разделение предметов на две группы. Так, если идти вначале по левой ветви корня, а затем по следующим трем правым ветвям, то первый предмет с весом 2 окажется в первой группе, а все остальные с весом 4, 1 и 1 — во второй. Числа под терминальными вершинами отражают суммарный весовой показатель каждой группы. В нашем случае это 2 и 6.

Элемент 1 1 2



Элемент 2 1 2 1 2

Элемент 3 1 2

###### 1 2 1 2 1 2

Эле- 1

мент 4

###### 2 1 2 1

2 1 2 1 2 1

2 1 2 1 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 |  | 7 |  | 7 |  | 6 |  | 4 |  | 3 |  | 3 |  | 2 |  | 6 |  | 5 |  | 5 |  | 4 |  | 2 |  | 1 |  | 1 |  | 0 |
| 0 |  | 1 |  | 1 |  | 2 |  | 4 |  | 5 |  | 5 |  | 6 |  | 2 |  | 3 |  | 3 |  | 4 |  | 6 |  | 7 |  | 7 |  | 8 |

**Рис. 12.3.** Дерево решений для моделирования задачи о разбиении

Следует отметить, что на рисунке 12.3 только две терминальные вершины (пя- тая и двенадцатая) соответствуют разделению общего веса предметов поровну. В сущности, это одно и то же решение, просто предметы попадают в разные группы.

**ЗАМЕЧАНИЕ**

Фактически любое найденное решение будет иметь два варианта, в которых группы меняются местами. Если перед началом поиска расположить произ- вольный элемент в первой группе, то можно сократить дерево на один уровень, удалив все случаи, где тот же элемент попадает во вторую группу. Тем не менее сходные решения останутся.

Дерево решений, изображенное на рисунке 12.3, достаточно велико, хотя в примере фигурирует только четыре предмета. Если бы их было больше, то структу- ра данных разрослась бы неимоверно. Например, для аналогичной задачи с 50 пред- метами понадобилось бы дерево, включающее 250 терминальных вершин, то есть количество возможных решений составило бы около 1,13  1015. При том что толь- ко несколько сценариев дадут нужное разделение веса, поиск наилучшего вариан- та становится слишком сложным.

Далее описана разница между разными версиями решения задача о разбиении. Одна из них — трудная, другая — очень трудная. Кроме того, вы познакомитесь с общими методами эффективного поиска по деревьям принятия решений.

#### Задачи оптимизации

Такие задачи, как задача о разбиении, зачастую принимают две формы: в первой устанавливается возможность того или иного решения, а во второй располагается лучшее. Применительно к вышеуказанному набору предметов первая форма должна определить, можно ли в принципе образовать две группы с равными суммарными весами, а вторая найти такое допустимое разделение, когда обе группы окажутся максимально близки по весу, — так называемая *задача оптимизации*. В одном слу- чае требуется строгий ответ — да или нет, в другом предоставляется приближенное решение, поэтому оптимизационная версия в некотором смысле легче.

Предположим, вам нужно разбить на две группы 100 элементов с общим весом 400 единиц. Если вы найдете в дереве решений строго равное разделение, ответом на первый вопрос будет «да». Но на подобные поиски можно потратить часы и даже дни. В этом случае нельзя сделать вывод о том, что идеального варианта разбиения не существует. К такому заключению можно прийти только, если вы его не найдете.

В оптимизационной версии задачи все выглядит проще. Найденное решение может быть не самым лучшим, однако оно есть и как минимум приближено к нему. При достаточно долгом поиске вполне вероятно, что это решение окажется отно- сительно хорошим, если не идеальным, а при определенной доле везения вы даже сможете разделить предметы строго поровну.

В следующих подразделах рассматривается поиск по деревьям принятия ре- шений: в первых двух описываются методы, которые применимы для оптимизаци- онной и неоптимизационной версии задачи, а далее, где речь идет об эвристике, — только для оптимизационной.

#### Метод полного перебора

Самый простой способ исследовать дерево принятия решений — проследовать по всем его вершинам и найти лучший ответ к задаче, то есть провести *исчерпывающий поиск*. Следует отметить, что для этого само дерево строить не нужно, необходим

лишь путь, который поможет определить, в каком месте структуры данных вы пре- бываете. Многие алгоритмы в таком случае используют рекурсию — она помогает выбирать ветви на разных уровнях, а сами вызовы отслеживают текущую позицию.

Представленный ниже высокоуровневый алгоритм работает на основе метода полного перебора решений для оптимизационной версии задачи о разбиении.

StartExhaustiveSearch()

<Инициализируем лучшее решение, которое заменяет первое тестовое.> ExhaustiveSearch(0)

End StartExhaustiveSearch

ExhaustiveSearch(Integer: next\_index)

// Проверяем, закончили ли мы. If <next\_index > max\_index>

// Все элементы назначены, значит, мы в терминальной вершине.

<Если тестовое решение лучше, чем последнее найденное, сохраняем его.>

Else

End If

// Назначены не все элементы — мы не в терминальной вершине.

<Относим элемент next\_index к группе 0.> ExhaustiveSearch(next\_index + 1)

<Отменяем отнесение элемента next\_index к группе 0.>

<Относим элемент at next\_index к группе 1.> ExhaustiveSearch(next\_index + 1)

<Отменяем отнесение элемента next\_index к группе 1.>

End ExhaustiveSearch

Метод StartExhaustiveSearch инициализирует лучшее найденное решение. Обычно он приравнивает начальное значение (в задаче о разбиении — это разница весов двух групп) к очень большому числу, поэтому первое же тестовое решение станет усовершенствованным. Затем вызывается метод ExhaustiveSearch, кото- рый и проделывает основную часть работы. Он берет в качестве параметра индекс элемента, который должен определить в группу, что, по сути, соответствует глубине рекурсии и уровню в дереве.

Распределив все элементы, ExhaustiveSearch сравнивает тестовое реше- ние с последним найденным лучшим. Если тестовое более совершенно, метод со- храняет его как новое лучшее. В противном случае ExhaustiveSearch пытается отнести элемент next\_index в группу 0, а затем рекурсивно вызывает сам себя, чтобы поработать с оставшимися элементами. Когда вызов возвращается, метод пробует определить элемент next\_index в группу 1, после также рекурсивно разбирается с прочими элементами. В итоге вызовы спускаются по дереву, пока не дойдут до терминальных вершин и при необходимости не обновят лучшее решение.

Приведенный базовый алгоритм относительно гибкий — его легко адаптировать для многих сложных задач. В случае разбиения, например, массив можно использо- вать для хранения тестового и последнего найденного лучшего решений. Запись *K* при- нимает значения 0 или 1, что указывает на принадлежность элемента к той или иной группе. Когда алгоритм доходит до терминальной вершины, он подсчитывает вес всех элементов в каждой группе и сравнивает разницу с найденным лучшим решением.

Хотя алгоритм и выглядит простым, тот факт, что он осуществляет поиск по всему дереву решений, делает его относительно медленным. Применяемый метод нельзя ускорить, но реально улучшить и значительно сократить время поиска. Когда алгоритм дойдет до терминальной вершины, где тестовое назначение создает две группы со строго равными суммарными весами, он может прекратить работу и не рассматривать оставшуюся часть дерева решений. Если оптимальных ответов множество, такое прерывание позволит справиться с задачей относительно быстро и отказаться от дополнительных временных затрат.

Так, при проведении исчерпывающего поиска попытка разделить 20 элемен- тов на две группы с равным весом потребует пересмотра 2 097 150 вершин. Если, отыскав оптимальное решение, алгоритм остановится, то он успеет пройти только по 4098 вершинам. Впрочем, эти результаты могут сильно отличаться в зависимо- сти от веса.

#### Метод ветвей и границ

Это более эффективный метод, чем исчерпывающий поиск. Когда алгоритм до- ходит до конца ветви, он рассчитывает лучший результат, который может достичь на данном пути. Если полученное значение не превосходит последнее найденное лучшее, ветвь игнорируется и поиска по ее поддереву не происходит. Иногда такой способ значительно экономит время.

Предположим, в задаче о разбиении отслеживается текущий суммарный вес предметов в каждой из двух групп, а также тех предметов, что еще никуда не вклю- чены. А теперь представьте, что алгоритм достиг того момента, когда группа 0 ве- сит 100 единиц, группа 1 — 50, а нераспределенные предметы — 20. При этом уже найдено решение, когда разница веса в двух группах составляет 20.

Если бы алгоритму пришлось поместить все оставшиеся элементы в группу 1, то ее вес стал бы 70, а разница в значениях с группой 0 оказалась бы равной 30. Однако алгоритм уже нашел решение с меньшим результатом 20. Это значит, что текущее тестовое решение нельзя усовершенствовать, программа может прекратить работу над ним и не распределять оставшиеся элементы.

В следующем высокоуровневом алгоритме осуществляется поиск методом ветвей и границ для оптимизационной версии задачи о разбиении.

StartBranchAndBound()

<Инициализируем лучшее решение, которое заменяет первое тестовое.> BranchAndBound(0)

End StartBranchAndBound

BranchAndBound(Integer: next\_index)

// Проверяем, закончили ли мы. If <next\_index > max\_index>

// Все элементы назначены, значит, мы в терминальной вершине.

<Если тестовое решение более совершенно, чем лучшее, сохраняем его.>

Else

// Назначены не все элементы — мы не в терминальной вершине.

<Относим элемент item next\_index в группу 0.> BranchAndBound(next\_index + 1)

<Отменяем отнесение элемента item next\_index в группу 0.>

<Относим элемент next\_index В группу 1.> BranchAndBound(next\_index + 1)

<Отменяем отнесение элемента next\_index в группу 1.>

End If

End BranchAndBound

Действия подобны тем, что имеют место при исчерпывающем поиске. Разни- ца заключается лишь в том, что алгоритм определяет, можно ли усовершенство- вать тестовое решение по сравнению с текущим лучшим, и, если нельзя, то рекур- сию не использует. Как правило, такой подход «срезает» многие ветви с дерева ре- шений, поэтому он более быстрый, чем исчерпывающий поиск.

Например, если попытаться разделить на две группы с равным весом все те же 20 элементов, то метод ветвей и границ пройдет лишь по 774 650 вершинам, а ис- черпывающий поиск, как мы помним, по 2 097 150. Используя разрешенное пре- рывание, описанное в предыдущем разделе, метод ветвей и границ успеет рассмо- треть 298 вершин, а исчерпывающий поиск — 4082.

Метод ветвей и границ очень полезен, но перед тем как перейти к эвристике, вспомним связанные с ним два важных факта. Во-первых, поиск происходит по любому пути дерева и может в результате дать более совершенное решение, чем то лучшее, что уже найдено. Это значит, что ответ в задаче будет оптимальным. Во- вторых, хотя в данном методе игнорируется большая часть дерева, оно может быть поистине огромным и процесс останется относительно медленным.

Так, в тестовом исчерпывающем поиске задача с разбиением 25 элементов была решена за 6,6 с, а метод ветвей и границ показал результат 2 с, что на первый взгляд выглядит весьма неплохо. Однако добавление нового элемента к задаче расширило дерево приблизительно вдвое и привело к тому, что метод ветвей и границ срабо- тал за 4 с. Еще один дополнительный элемент увеличил временные затраты до 7,9 с.

Метод ветвей и границ намного быстрее, чем исчерпывающий поиск, но и он не достаточно скор, чтобы справиться с действительно большим деревом приня- тия решений, например с тем, которое имеет 2,2 трлн вершин и иллюстрирует за- дачу о разбиении 40 элементов.

#### Эвристика дерева принятия решений

Исчерпывающий поиск и метод ветвей и границ находят наилучшее возмож- ное решение, но, к сожалению, работают только с относительно небольшими за- дачами. Для поиска в больших деревьях удобнее использовать эвристику. Она не обязательно найдет наилучший возможный ответ, но даст неплохую версию хотя бы для оптимизационной версии задачи. В следующих пунктах рассматриваются четыре эвристических метода применительно к задаче о разбиении.

#### Случайный поиск

Одна из наиболее простых эвристик для поиска по дереву решений — следо- вание по случайным путям. В этом случае в каждой вершине выбирается произ- вольная ветвь. Пересмотрев достаточное их количество, можно найти относительно неплохое решение. Описанный подход реализован в следующем псевдокоде.

RandomSearch()

<Инициализируем лучшее решение, которое заменяет первое тестовое.> For i = 1 To num\_trials

For index = 0 To max\_index

<Произвольно относим элемент с номером index в группу 0 или 1.>

Next index

// Проверяем, насколько совершенно это решение.

<Если тестовое решение превосходит лучшее, сохраняем его.>

Next i

End RandomSearch

Работа алгоритма начинается с инициализации лучшего решения. Затем вводит- ся цикл, задающий конкретное количество попыток. Для каждой из них алгоритм проходит по распределяемым элементам и случайным образом помещает их в одну из групп — 0 или 1.

По завершении распределения осуществляется проверка решения — превос- ходит ли оно последнее найденное лучшее. Если да, то новое решение сохраняется.

Когда вы пытаетесь разделить *N* предметов, для каждой попытки понадобится *N* шагов. Это доказывает быстроту, а следовательно, и эффективность эвристики в большом дереве. Ведь вероятность найти хорошее решение в таком случае может быть крайне мала и, очевидно, потребуется множество попыток.

Допустимо установить фиксированное количество попыток, например 100, — для малых деревьев этого вполне достаточно. Но все же лучше отталкиваться от размера дерева.

Еще одна стратегия — определить число попыток через функцию, связанную с количеством разделяемых предметов. Например, если предметов *N*, то можно вос- пользоваться такой формулой: num\_trials = 3*N*3. Конечно, по мере увеличения *N* функция 3*N*3 также быстро возрастает, но все же медленнее, чем 2*N*, поэтому поиск проводится лишь в небольшой части дерева принятия решений.

Вы также можете осуществлять попытки до тех пор, пока определенное коли- чество случайных путей в ряду не перестанет находить улучшения. Тогда алгоритм продолжит работу, если поиск усовершенствований будет оставаться относительно простым.

Вполне возможно, что в некоторых случаях идеальный вариант — заставить алгоритм непрерывно обновлять лучшее решение, пока вы его не остановите. Что ж, ваше право: если вам не нужно получить ответ быстро, почему бы не заставить программу поработать в течение многих часов или даже дней?

#### Усовершенствование путей

Предыдущий эвристический метод можно сделать более эффективным: нач- ните с выбора произвольного пути, затем случайным образом определите элемент и переместите его из текущей группы в другую. Если это улучшит разбиение, про- должайте обмен, если нет, отмените действия и предпримите следующую попытку. Повторяйте процесс, пока усовершенствование остается возможным.

У данного метода есть несколько вариантов. Например, вместо того чтобы пере- мещать элементы случайно, делайте это последовательно — по одному элементу

за раз. Процесс можно повторять, поскольку одно действие способно изменить со- отношение веса в двух группах и разрешить замену того элемента, который ранее было невозможно переместить. Описанный алгоритм представлен в следующем псевдокоде.

MakeImprovements()

<Инициализируем лучшее решение, которое заменяет первое тестовое.> For i = 1 To num\_trials

// Выбираем случайное начальное решение. For index = 0 To max\_index

<Произвольно относим элемент с номером index в группу 0 или 1.>

Next index

// Пытаемся улучшить решение. Boolean: had\_improvement = True While (had\_improvement)

// Предполагаем, что на этот раз улучшений нет. had\_improvement = False

// Пробуем переместить элементы. For index = 0 To max\_index

<Переносим элемент с номером index в другую группу.>

// Проверяем, улучшает ли это тестовое решение.

If <обмен улучшает решение> Then had\_improvement = True

Loop

Else

End If Next index

<Возвращаем элемент.>

// Проверяем, является ли решение более эффективным.

<Если тестовое решение превосходит лучшее, сохраняем его.>

Next i

End MakeImprovements

Алгоритм вводит цикл, чтобы предпринять определенное количество попыток, для каждой из которых выбирается случайное тестовое решение. В следующем цикле алгоритм пытается усовершенствовать текущее решение, переместив каждый элемент в ту группу, куда он не был распределен. Если это оправдано, алгоритм сохраняет результат, если нет — действие отменяется. Когда совершенствования более невозможны, тестовое решение сравнивается с последним найденным лучшим и сохраняется, если превосходит его.

Количество попыток определяется так же, как и в случайной эвристике: можно использовать фиксированное число, создавать зависимость от количества элемен- тов, вести поиск, пока не найдутся усовершенствованные лучшие решения или пока он не будет остановлен принудительно.

Улучшить путь, выполнив всего одно перемещение, практически невозможно. Предположим, вы распределяете предметы весом 6, 5, 5, 5, 3 и 3 единицы и соз-

даете две группы {6, 3, 3} и {5, 5, 5} весом 12 и 15 единиц соответственно. Разница между ними равна 3. Перемещение элемента из первой группы во вторую лишь увеличит разрыв, поэтому не приведет к эффективному решению. Так, если вы

переставите элемент весом 5 из второй группы в первую, то новые наборы будут выглядеть следующим образом: {6, 5, 3, 3} и {5, 5}, другими словами, их вес составит 17 и 10 единиц соответственно, что не является усовершенствованием. Зато если элемент весом 3 перенести из первой группы во вторую, а элемент весом 5 — из второй группы в первую, получатся наборы {6, 5, 3} и {5, 5, 3} весом 14 и 13 единиц. А это уже улучшенное решение по сравнению с исходным.

В других стратегиях усовершенствования делается попытка заменить сразу два элемента, хотя и этого не всегда достаточно (часто таких элементов должно быть три). Тем не менее предлагаемый подход считается несложным и может дать хороший результат, а потому является полезным.

#### Имитация отжига

Это улучшенная версия усовершенствования путей. В ходе нее в исходное решение вначале вносятся большие изменения, а затем все меньшие и меньшие, пока алгоритм не достигнет нужной тестовой версии, которая и сравнивается с по- следним найденным лучшим решением. Однако, как уже говорилось, и этот метод не идеален: могут быть случаи, когда необходимо переместить три элемента.

**ЗАМЕЧАНИЕ**

Идея имитации отжига заимствована из поведения молекул в охлаждающемся металле или минерале. Когда вещество нагревается слишком сильно, моле- кулы перемещаются быстро и беспорядочно. По мере снижения температуры это движение замедляется и формируются устойчивые структуры, которые благодаря излишкам энергии продолжают сливаться с другими образования- ми, формируя более стабильные связи. Когда вещество полностью остывает, избыточная энергия, которая смогла бы спровоцировать структурные разру- шения, сходит на нет. В результате, если процесс протекает достаточно мед- ленно, образуется лишь несколько больших кристаллов с очень устойчивым соединением молекул.

Другой вариант осуществить имитацию отжига — проанализировать случайные изменения любой степени сложности. Если такое изменение приводит к усовершен- ствованию, то алгоритм принимает его и продолжает работу. Если нет, алгоритм все равно соглашается с ним в надежде, что оно в итоге улучшит решение. Со временем он отказывается принимать изменения, которые не приводят к усовершенствова- нию, и сосредотачивается только на по-настоящему действенных.

#### Поиск максимума

Предположим, вы — заблудившийся в ночи турист, которому нужно взобраться на самую высокую вершину горы. Один из способов сделать это — двигаться по наиболее крутому склону. Если гора имеет относительно ровную форму, то в кон- це концов вы доберетесь до места назначения, но если нет — рискуете застрять на каком-нибудь боковом пике или холме.

В эвристике такой горный подъем означает следующее: алгоритм всегда делает выбор, который приближает его к лучшему решению. Применительно к задаче о раз- биении это означает, что очередным элементом в группе становится тот, который минимизирует разницу в суммарных весах. Другими словами, элемент добавляется к более «легкой» группе.

Предположим, у вас есть предметы весом 3, 4, 1, 5 и 6 единиц. Первый предмет может войти в любую группу, например в первую. Далее алгоритм рассматривает второй предмет весом 4. Если мы разместим его в той же группе, то образуются два набора — {3, 4} и { } с разницей в 7 единиц. Если отнести предмет во вторую группу — соотношение будет {3} и {4}, а разница — 1. Очевидно, что наилучшим выбором для алгоритма окажется последний вариант.

Затем выбирается место для третьего предмета весом 1. Размещение его в пер- вой группе даст наборы {3, 1} и {4} с весовой разницей 0, а во второй — {3} и {4, 1} с весовой разницей 2. Значит, лучшим местом для третьего предмета станет первая группа. Аналогичная работа продолжается до тех пор, пока не будут распределены все предметы.

В следующем псевдокоде представлен алгоритм поиска максимума.

HillClimbing()

For index = 0 To max\_index Integer: difference\_0 =

<Разница весов, если элемент index в группе 0.> Integer: difference\_1 =

<Разница весов, если элемент index в группе 1.>

If (difference\_0 < difference\_1)

<Размещаем элемент index в группе 0.>

Else

End If Next index

End HillClimbing

<Размещаем элемент index в группе 1.>

Если вы распределяете *N* предметов, то алгоритм совершает *N* шагов, поэтому является достаточно быстрым. Тем не менее в большом дереве принятия решений он вряд ли найдет лучший возможный ответ, хотя вполне вероятно, что пред- ложенная версия будет рациональной. В целом же поиск максимума настолько быстр, что его вполне можно усовершенствовать, используя методы, описанные в предыдущих разделах.

#### Сортированный поиск максимума

Легкий способ усовершенствовать алгоритм поиска максимума — сначала про- вести сортировку предметов, потом их анализ в порядке убывания. Смысл за- ключается в том, что на ранних этапах по группам распределяются более тяжелые предметы, а затем с помощью более легких производится балансировка суммарных весовых значений.

SortedHillClimbing()

<Сортируем элементы в порядке убывания веса.>

For index = 0 To max\_index Integer: difference\_0 =

<Разница весов, если элемент index в группе 0.> Integer: difference\_1 =

<Разница весов, если элемент index в группе 1.> If (difference\_0 < difference\_1)

Else End If

Next index

<Размещаем элемент index в группе 0.>

<Размещаем элемент index в группе 0.>

End SortedHillClimbing

Фактически это тот же алгоритм поиска максимума, только с добавлением сортировки.

На первый взгляд сделанная модификация незначительна, однако сортирован- ный поиск максимума, как правило, предоставляет более эффективное решение, чем несортированный.

Если вы разбиваете *N* предметов, алгоритм сортированного поиска максимума совершает *O*(*N* log *N*) шагов для упорядочения значений, а затем еще *N* для ге- нерирования решения. И хотя дополнительная сортировка несколько замедляет работу алгоритма, все равно он является очень быстрым, поэтому не ленитесь и попытайтесь усовершенствовать его точно так же, как и обычный алгоритм по- иска максимума.

#### Другие задачи дерева принятия решений

Деревья принятия решений пригодны для моделирования не только для за- дачи о разбиении, но и других сложных ситуаций. В следующих пунктах именно об этих алгоритмических задачах и пойдет речь. Чаще всего они идут парами: в одном случае спрашивается, возможно что-либо или нет, а во втором ищется оптимальный ответ.

#### Обобщенная задача о разбиении

В обычной задаче о разбиении набор предметов делится на две группы с рав- ным суммарным весом. В обобщенной версии аналогичное разделение проис- ходит на *K* групп. В дереве решений в этом случае у каждой вершины имеется *K* ветвей, соответствующих уровню, на котором располагается элемент на одном из *K* разбиений. При *N* элементах дерево будет иметь *N* уровней и *KN* терминаль- ных вершин.

Эвристики, применяемые к обычной задаче о разбиении, подходят и для обоб- щенного варианта, хотя и существенно усложняются. Например, попытка переме- стить предмет из одной группы в другую трансформируется в попытку переставить предмет в любую из групп *K* – 1.

Оптимизационная версия обобщенной задачи ищет способ разделить предме- ты на *K* групп наилучшим образом, например так, чтобы минимизировать сумму всех абсолютных значений, которые получаются при вычитании из реального веса группы усредненного показателя. Предположим, у вас есть четыре группы суммар- ным весом 15, 18, 22 и 25 единиц. Среднее значение приведенных весов равно 20,

поэтому абсолютные разности получатся равными 5, 2, 2 и 5, что в сумме даст 14. Вы также можете минимизировать сумму квадратов разностей. Для данного примера это будет выглядеть так: 25 + 4 + 4 + 25 = 58. Подобные приемы находят решения, в которых все веса групп близки к среднему значению.

#### Задача о сумме подмножеств

В этой задаче имеется набор чисел и нужно определить, есть ли подмножество, сумма элементов которого дает 0. В качестве примера возьмем числа {–11, –7, –5,

–3, 4, 6, 9, 12, 14}, для них таким подмножеством является {–7, –5, –3, 6, 9}. Опти- мизационная версия задачи потребует от вас найти результат, близкий к 0.

Для моделирования этой задачи дерево принятия решений напоминает таковое из задачи о разбиении. По сути, нужно разделить элементы на две группы: в одну собрать те, что в сумме дадут 0, а во вторую — все исключенные. Если всего эле- ментов *N*, то у дерева будет *N* уровней, у каждой его вершины — по две ветви (одна соответствует добавлению элемента в подмножество с нулевой суммой, а вторая его исключению) и в общей сложности 2*N* терминальных вершин.

С оптимизационной версией данной задачи можно использовать метод ветвей и границ, а также эвристику.

#### Задача об упаковке в контейнеры

В данной задаче имеется набор предметов разного веса, а также несколько контейнеров одинаковой емкости (в обобщенной версии емкости могут быть раз- ными). Ваша цель — разложить элементы так, чтобы задействовать минимальное количество контейнеров.

Моделируя ситуацию, представьте следующее: каждая ветвь в дереве принятия решений соответствует тому, что предмет помещается в определенный контейнер. Если таких предметов *N*, а контейнеров *K*, то в дереве будет содержаться *N* уровней и *K* ветвей в каждой вершине, то есть в целом в нем окажется *KN* терминальных вершин. Поскольку это задача оптимизации, в поисках хорошего решения можно использовать метод ветвей и границ, а также эвристику.

Смежная задача требует разложить предметы по контейнерам таким образом,

чтобы число контейнеров определялось как r<общий вес всех элементов>  <ем-

1 r 1

кость контейнера> , где означает округление. Например, если общий вес предме- тов равен 115 единицам, а емкость контейнеров — 20, то вам предстоит выяснить, можно ли упаковать весь набор в 6 контейнеров. Воспользуйтесь эвристикой, что- бы найти хорошее решение, но даже если вы не отыщите его, это не значит, что та- кового не существует.

#### Задача раскроя

По сути, это двумерная версия предыдущей задачи об упаковке в контейне- ры, где необходимо вырезать из досок, кусков ткани и т. д. определенные фигуры (как правило, прямоугольники). Цель — использовать минимальное количество материала.

Смоделировать задачу в виде дерева принятия решений намного сложнее, чем в случае с распределением предметов по контейнерам, поскольку общее количе- ство полученных форм зависит от способа их размещения на исходном материале. Просто привязать фигуру к материалу недостаточно, нужно назначить ее позицию. Если у вас имеется *K* кусков чего бы то ни было, то есть более *K* мест, где можно разместить форму, следовательно, у вершины дерева присутствует более *K* ветвей.

Тем не менее с учетом некоторых условностей дерево принятия решения исполь- зовать все же возможно. Например, если кусок материала имеет размер 3672 см

и вы допускаете размещение формы только на позиции (*X*, *Y*), где *X* и *Y* — целые числа, то существует 36  72 = 2592 вероятных положений для нее. Это означает, что у каждой вершины в дереве будет *K*  2592 ветвей.

К счастью, многие из этих ветвей легко «срезать», например те, что располагают форму очень близко к краю материала или вынуждают перекрывать другую форму. Проигнорировав подобные случаи, проводят поиск хотя бы какого-то решения. Об- ратите внимание, что дерево все равно останется очень большим, и для нахождения рациональных ответов придется использовать эвристику.

Также следует отметить, что принятые допущения могут исключать некоторые решения. Предположим, вы пытаетесь расположить квадраты размером 77 см на куске материала площадью 2020 см. Если края фигуры и материала будут параллельны друг другу, то получится уместить лишь четыре квадрата (рис. 12.4, слева); если же использовать вращение формы — квадратов ляжет уже пять (рис. 12.4, справа).

**Рис. 12.4.** Расположение фигур на материале: без поворота формы *(слева)* и с поворотом *(справа)*

В стандартном варианте задачи дается один очень длинный кусок материала, например рулон бумаги. Ваша цель — минимизировать число используемых по- гонных метров.

#### Задача о ранце

В этой задаче предметы с определенным весом и ценностью нужно сложить в ранец, рассчитанный на некоторую грузоподъемность. Причем сделать это следует так, чтобы содержимое ранца имело максимальное значение. Другими словами, можно наполнить его преимущественно тяжелыми предметами высокой ценности либо легкими предметами низкой ценности.

Рассматриваемая задача подобна задаче о разбиении, фактически предметы делятся на две группы: те, что войдут в ранец, и те, что не войдут. Цель — сделать так, чтобы первая группа оказалась максимально ценной и подходящей по весу. Дерево принятия решений будет иметь много общего с тем, что фигурирует в за- даче о разбиении, разница лишь в том, что накладывается ограничение по весу. Пригодятся и аналогичные методы решения — случайные и улучшенные, а также имитация отжига.

Работу метода ветвей и границ можно остановить, если суммарное значение нерассмотренных элементов окажется недостаточным для совершенствования текущего решения так, чтобы оно превосходило уже найденное лучшее решение.

Предположим, вы уже отыскали ответ, где итоговая ценность предметов составляет 100 руб., и анализируете частичное решение, в котором на данный момент ценность равна 50 руб. Если суммарное значение оставшихся предметов 20 руб., то вам уже никак не удастся превзойти текущее лучшее решение.

Эвристика поиска максимума на каждом шаге будет добавлять предмет с наи- высшим значением, который способен вместиться в ранец. Отсортированный ва- риант этого алгоритма рассмотрит предметы в порядке убывания веса, то есть на последних этапах постарается максимально использовать оставшееся пространство в ранце. А еще лучше анализировать соотношение вес — ценность, тогда в первую очередь в ранец попадут предметы, чей килограмм веса имеет большую стоимость в рублях.

Дерево решений для задачи о ранце при *N* элементах содержит 2*N* терминальных вершин, то есть найти ответ не легко, но эвристические методы опробовать стоит.

#### Задача коммивояжера

Предположим, вы разъездной торговый агент, которому нужно посетить опре- деленные места, а затем вернуться в исходную точку. Ваша цель — спланировать маршрут так, чтобы пройденное расстояние оказалось минимальным.

**ЗАМЕЧАНИЕ**

Задача коммивояжера имеет широкое практическое применение, например для служб доставки и почтовых учреждений. Рациональное планирование маршрута помогает сократить общее пройденное расстояние, уменьшить износ транспортных средств и сэкономить топливо.

При моделировании данной задачи каждый уровень *K* в дереве принятия ре- шений должен соответствовать выбору места *K* в итоговом маршруте. Если есть *N* пунктов назначения, то в корневой вершине находится *N* ветвей, каждая из которых соответствует посещению определенного места. На втором уровне ветвей будет уже *N* – 1 по количеству еще не посещенных мест, на третьем — *N* – 2 и т. д. вплоть до терминальных вершин. Общее число последних составит *N*  (*N* – 1)  (*N* – 2)  ...  1 = *N*!.

Итоговое дерево получится настолько большим, что для нахождения оптималь- ных решений вам потребуется использовать эвристику.

#### Задача выполнимости булевых формул

Если дано логическое утверждение, например «*A* и *B* или (*A* и не *С*)», то в те- кущей задаче требуется ответить на вопрос: можно ли присвоить значения true и false переменным *A*, *B* и *C* так, чтобы утверждение стало истинным? В смежной задаче необходимо найти это самое назначение.

При моделировании можно воспользоваться бинарным деревом принятия ре- шений: его левые ветви будет соответствовать установке значения true, а правые — false. Каждая терминальная вершина в этом случае ознаменует назначение всех переменных и определит, является ли все утверждение истинным или ложным.

Данная задача сложнее задачи о разбиении, поскольку в ней нет приблизи- тельных решений. Любая терминальная вершина дает конкретный ответ, ведь

утверждение не может считаться приблизительно истинным. (Тем не менее веро- ятностная логика, где переменные не являются абсолютно истинными или лож- ными, позволяет сделать утверждение возможно истинным.)

Случайный поиск по дереву позволит найти решение, но если вы его не отыще- те, то не сможете сделать вывод о том, что такого не существует. Пытаясь усовер- шенствовать решения, помните: внесенные изменения повлияют на истинность или ложность всего утверждения, то есть применять постепенные улучшения не полу- чится. Это значит, что вам не удастся воспользоваться стратегией усовершенство- вания пути, имитацией отжига или поиском максимума, описанными ранее. Кро- ме того, невозможно определить, станет ли утверждение ложным после частичных назначений, поэтому метод ветвей и границ также отпадает. Получается, что един- ственно доступными остаются исчерпывающий и случайный поиски, которые дей- ственны только для относительно малых задач.

**ЗАМЕЧАНИЕ**

 Учитывая огромное количество нюансов, вы можете спросить, зачем в прин- ципе нужна задача выполнимости булевых формул. Оказывается, к ней можно

свести многие другие задачи и судить об их решаемости по тому, выполнимо или невыполнимо построенное логическое выражение. Другими словами, если нет известного простого решения для задачи выполнимости булевых формул, то нет его и для другой задачи.

Задача выполнимости булевых формул связана с задачей выполнимости в 3-конъюнктивной нормальной форме, где логическое утверждение состоит из вы- сказываний, объединенных оператором And. Каждое высказывание в свою очередь содержит три переменные или их отрицания и оператор Or. Например, утверждение

«(*A* или *B* или не *C*) и (*B* или не *A* или *D*)» записано в формате задаче 3-выполни-

мости. При желании можно доказать, что задачи выполнимости и 3-выполнимости являются равноценными, то есть их одинаково трудно решать. Кроме того, следует отметить, что для обеих версий подойдет одно и то же дерево принятия решений.

## Резюме

Деревья принятия решений — мощный инструмент, с помощью которого лег- ко найти ответ в сложных задачах, пусть даже и не наилучший. Например, задачу о восьми ферзях, описанную в главе 9, можно смоделировать посредством дерева, состоящего из 8 уровней, 64 ветвей и 648 � 2,8  1014 терминальных вершин. В даль- нейшем с учетом условий игры большую часть ветвей допустимо срезать и иссле- довать только 113 оставшихся позиций на доске. Но надо сказать, такая интерпре- тация задачи может стать ошибочной — вполне вероятно, что вы пропустите упро- щения, которые позволят найти ответ более эффективным способом. Тем не менее деревья принятия решений предоставляют отличную технологию, весьма удобную для тех случаев, когда других идей решения задачи нет.

В главах 10–12 вы познакомились и с самими деревьями, и с алгоритмами, ко- торые их строят и производят поиск по ним. В следующих двух главах мы перей- дем к изучению сетей — еще одних связных структур данных, которые в отличие от деревьев не являются иерархичными. Это позволяет использовать циклы, что несколько усложняет работу в целом, зато помогает моделировать и решать дру- гие интересные задачи.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

1. Напишите программу, которая проводит исчерпывающий поиск по игре в крестики-нолики и подсчитывает количество партий, которые заканчивают- ся победой крестиков, ноликов или ничьей. Сколько партий в принципе возмож- но? Указывают ли полученные цифры на преимущество того или иного игрока?
2. Внесите изменения в программу из упражнения 1 из расчета, что пользова- тель совершил один начальный ход или несколько. Снова подсчитайте коли- чество возможных побед крестиков, ноликов или ничьих. Сколько игр мож- но провести с учетом каждого из девяти начальных ходов, совершенных кре- стиками? (Нужно ли рассчитывать все девять?)
3. Напишите программу игры в крестики-нолики с компьютером. Пусть поль- зователь выступает за любую из команд. Продумайте три уровня сложности:

«Случайный» (компьютер ходит произвольным образом), «Новичок» (ком- пьютер использует минимакс только с тремя уровнями рекурсии) и «Экс- перт» (компьютер использует минимакс во всей игре).

1. Напишите программу с исчерпывающим поиском для решения оптимиза- ционной задачи о разбиении. Разрешите пользователю задать вес предме- тов в установленных пределах и установить флажок, допускающий прину- дительно прервать работу алгоритма.
2. Модернизируйте программу из упражнения 4: разрешите использовать ме- тод ветвей и границ без принудительного прерывания.

6\*. На основе программы из упражнения 4 решите задачу о разбиении, исполь- зуя исчерпывающий поиск, метод ветвей и границ, а также значения от 1 до 5, от 1 до 6, от 1 до 7 и т. д. до 25. Отразите схематично количество прой- денных вершин и разбиение предметов для двух методов. Отдельно изобра- зите логарифмическую зависимость количества пройденных вершин от ко- личества весов. Какой вывод можно сделать о количестве пройденных вер- шин для двух методов?

1. Модернизируйте программу из упражнения 5, используя случайную эвристику.
2. Перепишите программу из упражнения 7 с учетом эвристики усовершен- ствования путей.
3. Какие группы найдет программа для разбиения, если она будет использо- вать эвристику поиска максимума для весов 7, 9, 7, 6, 7, 7, 5, 7, 5 и 6? Каковы суммарные веса групп и разница между суммарными весами?
4. Модернизируйте программу из упражнения 8, используя эвристику поис- ка максимума.
5. Проработайте упражнение 9, прибегнув к эвристике сортированного поис- ка максимума.
6. Проработайте упражнение 9, используя исчерпывающий поиск.
7. Усовершенствуйте программу из упражнения 10 за счет эвристики сортиро- ванного поиска максимума.

# Глава 13

**ОСНОВНЫЕ СЕТЕВЫЕ АЛГОРИТМЫ**

В главах 10–12 мы рассматривали деревья. В этой главе описывается родствен- ная структура данных — сеть. По аналогии с деревом сеть содержит узлы, которые соединяются посредством звеньев, но, в отличие от узлов в деревьях, сетевые узлы могут не иметь иерархической связи. В частности, звенья в сети формируют ци- клы, в результате чего путь, который следует через эти звенья, способен вернуть- ся в исходную точку.

Эта глава посвящена сетям и некоторым базовым сетевым алгоритмам, таким как нахождение циклов, поиск кратчайшего пути и построение дерева, которое яв- ляется частью сети, включающей в себя каждый узел.

## Терминология

У сетей терминология не настолько сложная, как у деревьев, поскольку она не заимствует так много понятий из генеалогии. Однако на рассмотрение соответству- ющих терминов все же стоит потратить несколько минут.

Сеть состоит из набора *узлов*, соединенных с помощью *звеньев* (иногда, осо- бенно в контексте математических алгоритмов и теорем, сеть называют *графом*, узлы — *вершинами*, а звенья — *ребрами*). Если узлы *A* и *B* связаны напрямую, они являются *смежными* и именуются *соседями*.

В отличие от деревьев, сеть не имеет корневого узла, хотя в некоторых сетях опре- деленные узлы могут обладать особым значением. Например, транспортная сеть мо- жет содержать специальные узлы-концентраторы, в которых начинают и заверша- ют свои маршруты автобусы, поезда, паромы или другие транспортные средства.

Звено может быть *направленным* (если переход по нему возможен только в одном направлении) или *ненаправленным* (если допускается любое направле- ние перехода). Сеть называется направленной или ненаправленной в зависимо- сти от типа звеньев, которые она содержит.

*Путь* — это последовательность чередующихся узлов и звеньев, проходящая по сети от одного узла к другому. Допустим, любые два соседних узла соединены только одним звеном (другими словами, между узлами *A* и *B* нет двух звеньев). В этом случае путь можно указать, перечислив узлы, которые он проходит, или звенья, которые использует.

*Цикл,* или *петля,* — это путь, который возвращается в свою исходную точку. Как и в случае с деревьями, *степенью* узла называется количество звеньев, ко-

торые нужно пройти до него. Степень сети равна наибольшей степени любого узла

внутри нее. В направленной сети входящая и исходящая степень узла определяют- ся количеством звеньев, которые входят и выходят из него.

С узлами и звеньями часто связаны какие-нибудь данные. Например, узлы не- редко имеют имя, идентификационный номер или физическое местоположение, такое как широта и долгота. Многие звенья характеризуются издержками или ве- сом, в частности временем, которое занимает проезд по звену в уличной сети. Они также могут иметь максимальную пропускную способность, например максималь- ный объем тока, который можно пропустить по проводу в электросети, или наи- большее количество автомобилей, которые проедут по улице за определенную еди- ницу времени.

*Доступным* называется узел, которого можно достигнуть из заданного узла, следуя по звеньям. В зависимости от сети некоторые узлы могут быть недоступны из определенных позиций.

В направленной сети узлы *A* и *B* называются *связанными*, если к *B* можно по- пасть из *A*. Стоит отметить, что если узел *A* связан с узлом *B*, а узел *B* — с узлом C, то узлы *A* и *C* также должны быть связанными.

*Связный компонент сети* — это набор всех взаимосвязанных узлов. Сеть назы- вается связной, если все ее узлы соеденены друг с другом.

Если все узлы в направленной сети сообщаются между собой, такая сеть назы- вается *сильно связной*. Если в связной направленной сети заменить направленные звенья на ненаправленные, такая сеть станет *слабо связной*.

На рисунке 13.1 показаны некоторые участки небольшой направленной сети. Звенья представлены стрелками, острия которых обозначают направления. Двуна- правленные стрелки изображают парные звенья, которые имеют разные направле- ния. Издержки звеньев указаны в виде чисел. В рассматриваемом примере, чтобы сделать изображение сети более простым, мы исходим из того, что противополож- ные звенья имеют одинаковые издержки. На практике это не всегда так.

Сеть, представленная на рисунке 13.1, является сильно связной, потому что с помощью ее звеньев можно найти путь от одного узла к любому другому.

Звенья

Узлы



*A* 9

*B*

10

*C*

10 15

11

*D*

10

*E*

11

*F*

12

13

*G*

12

*H*

Издержки

**Рис. 13.1.** В направленной сети стрелки обозначают направления звеньев

Обратите внимание, что пути между узлами могут не быть уникальными. На- пример, и *A*-*E*-*F*, и *A*-*B*-*C*-*F* являются путями между узлами *A* и *F*.

Термины, касающиеся деревьев, собраны в таблице 13.1 — так вам будет про- ще их запомнить.

**Òàáëèöà 13.1.** Êðàòêîå èçëîæåíèå ñåòåâîé òåðìèíîëîãèè

|  |  |
| --- | --- |
| **Термин** | **Значение** |
| Смежный | Если два узла соединены звеном, они являются смежными |
| Пропускная способность | Максимальный объем чего-либо, что может пройти через узел или звено. Например, максимальный ток, проходящий через провод в электросети, или наибольшее количество автомоби- лей, которые пересекут звено в уличной сети за единицу вре- мени |
| Связанный | В ненаправленной сети узлы *A* и *B* являются связанными, если в узел *B* можно попасть из узла *A* (и наоборот). Ненаправлен- ная сеть считается связной, если каждый узел доступен  из любого другого узла |
| Связный компонент | Набор взаимосвязанных узлов |
| Издержки | Со звеном могут быть связаны определенные издержки. То же самое, но в меньшей мере касается узлов |
| Цикл | Путь, который возвращается в свою исходную точку |
| Степень | В ненаправленной сети это количество звеньев, которые нужно пройти до узла. В направленной сети узел имеет входящую  и исходящую степень |
| Направленный | Звено является направленным, если переход по нему возможен только в одном направлении. Сеть считается направленной, если она содержит направленные звенья |
| Ребро | Звено |
| Граф | Сеть |
| Входящая степень | Количество звеньев, входящих в узел направленной сети |
| Звено | Объект в сети, представляющий отношение между двумя узла- ми. Звенья могут быть направленными и ненаправленными |
| Петля | Цикл |
| Сосед | Два узла являются соседями, если они смежные |
| Узел | Объект в сети, представляющий точкообразное местоположе- ние. Узлы соединяются с помощью звеньев |
| Исходящая степень | Количество звеньев, исходящих из узла в направленной сети |
| Путь | Последовательность чередующихся узлов и звеньев, проходя- щая по сети от одного узла к другому. Если два любых смеж- ных узла соединены только одним звеном, путь можно указать в виде списка узлов или звеньев, которые он в себя включает |
| Доступный узел | Узел *B* является доступным из узла *A*, если существует путь из узла *A* в узел *B* |
| Сильно связный | Направленная сеть является сильно связной, если каждый ее узел доступен из любого другого узла |

|  |  |
| --- | --- |
| **Термин** | **Значение** |
| Ненаправленный | Звено является ненаправленным, если переход по нему воз- можен в любом направлении. Сеть считается ненаправленной, если она содержит только ненаправленные звенья |
| Вершина | Узел |
| Слабо связный | Направленная сеть является слабо связной, если каждый ее узел доступен из любого другого узла после замены направлен- ных звеньев на ненаправленные |
| Вес | Затраты |

## Разные представления сети

Продемонстрировать сеть в виде объектов довольно просто. Можно обозначить узлы с помощью класса Node.

Представление звеньев зависит от того, как вы будете их использовать. Напри- мер, создавая направленную сеть, следует помнить, что звенья могут иметь вид ссы- лок, хранящихся в классе узла. Туда же добавляются затраты или другие данные, если они должны быть связаны со звеньями. Описываемый случай представлен на примере следующего псевдокода с простым классом узла.

Class Node

String: Name List<Node>: Neighbors List<Integer>: Costs

End Node

Это решение подходит для простых ситуаций, но часто бывает полезно соз- дать класс для представления звеньев в виде объектов. Например, отдельные ал- горитмы, такие как алгоритм минимального остовного дерева, который будет опи- сан далее в текущей главе, формируют список звеньев. Если звенья являются объ- ектами, не составляет никакого труда поместить в список определенные их экзем- пляры. Если же они представлены в виде ссылок, хранящихся в классе узла, сде- лать это будет сложнее.

В следующем псевдокоде показаны классы узла и звена, которые хранят звенья в виде отдельных объектов для ненаправленной сети.

Class Node

String: Name List<Link>: Links

End Node Class Link

Integer: Cost Node: Nodes[2]

End Link

Здесь класс Link содержит массив из двух объектов Node, представляющих соединяемые им узлы.

В ненаправленной сети объект Link выступает звеном, связывающим два узла, при этом порядок следования этих узлов не имеет значения. Если звено соединяет

узлы *A* и *B*, объект Link будет находиться в списке Neighbors обоих узлов, по- этому вы можете придерживаться его, двигаясь в любом направлении.

Поскольку порядок следования узлов в массиве Nodes внутри звена неважен, алгоритм, пытающийся найти соседа, должен сравнивать текущий узел с элемента- ми Nodes, чтобы определить, какой из них является смежным. Например, если алго- ритм пытается отыскать соседа узла *A*, ему нужно обратиться к массиву Nodes вну- три звена, чтобы понять, какой из его элементов — узел *A*, а какой — сосед.

В направленной сети классу звена по-настоящему необходимо знать только свой целевой узел. Классы для такой ситуации представлены в следующем псевдокоде.

Class Node

String: Name List<Link>: Links

End Node Class Link

Integer: Cost Node: ToNode

End Link

Тем не менее в целях удобства класс звена по-прежнему может содержать ссыл- ки на оба своих узла. Например, если узлы сети имеют пространственное размеще- ние, а звенья содержат ссылки на свои исходные и целевые узлы, это облегчит за- дачу прорисовки звеньев. Если звенья располагают ссылками только на свои целе- вые узлы, объектам узлов придется передавать звеньям дополнительную инфор- мацию, чтобы те могли себя прорисовать.

Если вы используете класс звена, содержащий массив Nodes, исходный узел можно хранить в первом элементе этого массива, а целевой — во втором.

**ЗАМЕЧАНИЕ**

В демонстрационных программах, доступных для загрузки на веб-сайте этой книги, используется первое представление — то, в котором класс звена имеет свойство Nodes, хранящее ссылки как на исходный, так и на целевой узел.

Оптимальный способ представления сети в файле зависит от того, какие ин- струменты доступны в вашей среде программирования. Например, некоторые XML-библиотеки могут сохранять и загружать сетевую информацию, хотя XML является иерархическим языком, который лучше подходит для иерархических структур данных.

Чтобы ничего не усложнять, в примерах, доступных для загрузки, используется структура простого текстового файла. Файл начинается с количества узлов в сети. Затем для каждого узла выделяется по одной строке текста.

y Каждая строка, представляющая узел, содержит его название и координаты (x,

). Дальше следует набор записей для звеньев узла. Каждая такая запись включает индекс целевого узла, затраты звена и его пропускную способность.

Формат показан на примере следующих строк.

number\_of\_nodes name,x,y,to\_node,cost,capacity,to\_node,cost,capacity,... name,x,y,to\_node,cost,capacity,to\_node,cost,capacity,... name,x,y,to\_node,cost,capacity,to\_node,cost,capacity,...

Ниже показан файл, представляющий сеть, которая описана на рисунке 13.2.

3

A,85,41,1,87,1,2,110,4 B,138,110,2,99,4

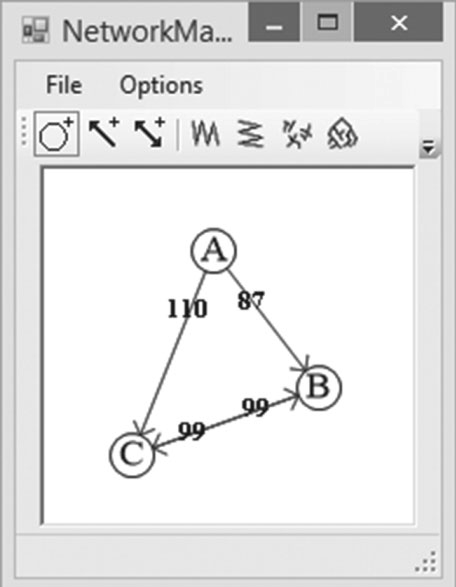
C,44,144,1,99,4

Файл начинается с количества узлов — 3. Затем следуют строки, представля- ющие каждый узел.

Строка для узла *A* начинается с его имени — A. Следующие две записи опреде- ляют координаты узла (x, y); в рассматриваемом примере узел расположен в точ- ке (85, 41).

Дальше строка содержит последовательность наборов значений, описывающих звенья. Первый набор относится к первому звену, ведущему к узлу *B* (индекс 1); он имеет издержки в размере 87 и пропускную способность 1. Второй набор отно- сится ко второму звену, ведущему к узлу *C* (индекс 2); он имеет издержки в раз- мере 110 и пропускную способность 4.

Остальные строки файла определяют узлы *B* и *C* вместе с их звеньями.



**Рис. 13.2.** Эта сеть содержит четыре звена: первые два соединяют узел *A*

с узлами *B* и *C*, два других — узел *B* с узлом *C*

**ЗАМЕЧАНИЕ**

 Чтобы программировать сетевые алгоритмы, вам нужно иметь возможность создавать сети. Вы можете написать код, который создает в сети по одному

узлу и звену за раз, но было бы удобно иметь программу, с помощью которой можно генерировать тестовые сети. Инструкции касательно того, что эта про- грамма должна делать, ищите в упражнении 1.

## Обход сети

Многие алгоритмы так или иначе занимаются обходом сети. Например, алго- ритмы остовного дерева и кратчайшего пути минуют узлы в дереве.

В следующих разделах описывается несколько алгоритмов, которые решают сетевые проблемы с помощью разных видов обхода.

#### Обход в глубину

Алгоритм обхода деревьев в ширину, приведенный в главе 10, почти подходит для сетей. Представленный ниже псевдокод демонстрирует немного видоизменен- ную версию этого алгоритма, адаптированную для класса сетевых узлов.

Traverse()

<Process node>

For Each link In Links

link.Nodes[1].Traverse Next link

End Traverse

Сначала этот метод обрабатывает текущий узел. Затем он циклически переби- рает его звенья и рекурсивно вызывает самого себя для обработки целевого узла каждого звена.

Этот код будет работать, если не считать одной серьезной проблемы. В отличие от деревьев, сети не являются иерархическими, вследствие чего они могут содер- жать петли. Если в сети есть петля, данный алгоритм в конце концов начнет по ней рекурсивное движение, войдя в бесконечный цикл.

Одно из решений подобной проблемы заключается в изменении алгоритма та- ким образом, чтобы он мог понимать, посещался ли узел ранее. Это можно сделать очень просто — добавить в класс узла свойство Visited. В следующем псевдокоде показан алгоритм, переписанный для использования такого свойства.

Traverse()

<Process node> Visited = True

For Each link In Links

If (Not link.Nodes[1].Visited) Then link.Nodes[1].Traverse

End If Next link

End Traverse

Теперь алгоритм, посещая текущий узел, присваивает свойству Visited зна- чение True. Затем он циклически перебирает звенья узла. Если свойство Visited целевого узла звена равно False, он рекурсивно вызывает самого себя, обрабатывая этот целевой узел.

И хотя рассмотренное решение работает, оно может привести к глубоким уров- ням рекурсии. Если сеть содержит *N* узлов, алгоритм способен вызвать себя *N* раз. В случае большого *N* это грозит переполнением стека и сбоем программы.

Вы можете избежать этой проблемы с помощью методик, описанных в главе 9, которые позволяют убрать рекурсию. В следующем псевдокоде показана разновид- ность алгоритма, в которой вместо рекурсии используется стек.

DepthFirstTraverse(Node: start\_node)

// Посещаем узел. start\_node.Visited = True

// Создаем стек и помещаем в него начальный узел.

Stack(Of Node): stack stack.Push(start\_node)

// Повторяем, пока стек не станет пустым. While <stack isn’t empty>

// Берем следующий узел из стека. Node node = stack.Pop()

// Обрабатываем звенья узла. For Each link In node.Links

// Используем звено, только если целевой узел еще не посещался.

If (Not link.Nodes[1].Visited) Then

// Отмечаем узел как посещенный. link.Nodes[1].Visited = True

// Заносим узел в стек. stack.Push(link.Nodes[1])

End If Next link

Loop // Продолжаем обработку стека, пока тот не иссякнет.

End DepthFirstTraverse

Этот алгоритм посещает начальный узел и помещает его в стек, а затем, пока стек не освободится, убирает из него следующий узел и обрабатывает его.

В процессе обработки узла алгоритм проверяет его звенья. Если целевой узел звена еще не посещался, алгоритм помечает его как посещенный и добавляет в стек для дальнейшей обработки.

Благодаря такому способу добавления узлов в стек этот алгоритм обходит сеть в глубину. Чтобы понять, почему так происходит, предположим, что алгоритм на- чинает с узла *A* и что *A* имеет соседей *B*1, *B*2 и т. д. При обработке узла *A* алгоритм помещает его соседей в стек. Позже, во время обработки *B*1, он добавляет в стек его смежные узлы, *C*1, *C*2 и т. д. Поскольку стек возвращает элементы по принципу

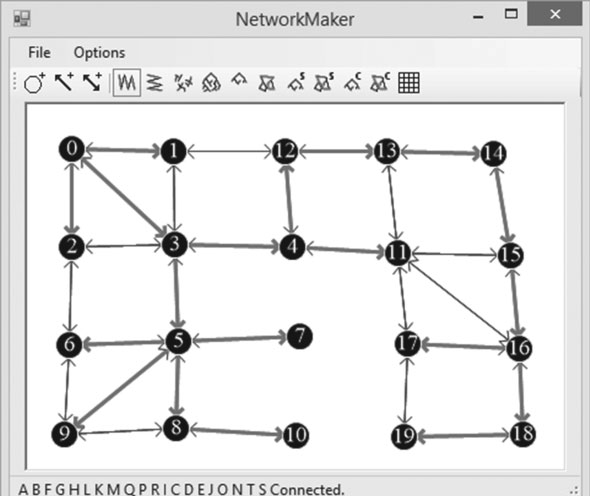
«последним пришел — первым вышел», алгоритм обрабатывает узлы *Ci* перед *Bi*.

Прежде чем приступить к обработке ближайших соседей начального узла *A*, алго-

ритм в процессе своей работы удаляется от него на большие расстояния, быстро проходя по сети.

Такой метод называется *обходом в глубину*, потому что перед посещением всех узлов, располагающихся ближе всего к корню, он удаляется от начальной точки на большое расстояние.

На рисунке 13.3 показан обход в глубину, где узлы помечены в соответствии с порядком, в котором их минуют.



**Рис. 13.3.** При обходе в глубину некоторые узлы, значительно удаленные от корневого, посещаются раньше, чем те, что находятся близко к нему

Приложив некоторые усилия, можно выяснить, каким образом узлы выбира- лись для обхода. Алгоритм начал с узла, помеченного как 0. Затем он добавил в стек узлы с метками 1, 2 и 3.

Поскольку узел 3 был помещен в стек после всех остальных, он обрабаты- вался следующим, затем в стек добавлялись узлы 4 и 5. Вставленный узел — 5 — шел следующим в очереди на обработку; потом алгоритм помещал в стек узлы 6, 7, 8 и 9.

Если хотите, можете продолжить изучение рисунка 13.3, чтобы разобраться, почему алгоритм посетил узлы именно в таком порядке. Но уже сейчас вам долж- но быть видно, что некоторые узлы, удаленные от начальной точки, обрабатыва- ются раньше тех, которые находятся ближе к ней.

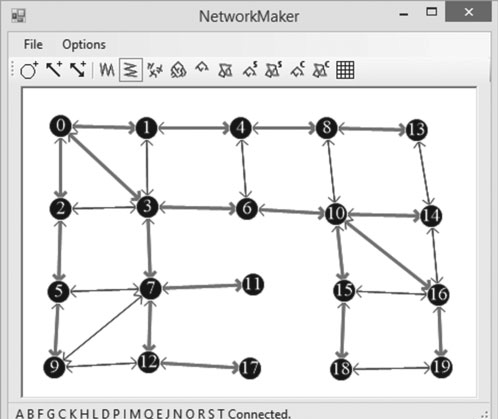
#### Обход в ширину

Иногда удобнее начинать обход с узлов, которые находятся ближе к началу, и только потом посещать более удаленные узлы. Предыдущий алгоритм рассма- тривал узлы, удаленные от начальной точки, раньше тех, которые располагаются ближе к ней, поскольку для их обработки использовался стек. Если вместо стека прибегнуть к очереди, узлы будут обрабатываться по принципу «первый зашел — первый вышел», в результате чего сначала обработке подвергнутся узлы, находя- щиеся ближе к начальной точке.

Рассматриваемый алгоритм, прежде чем двигаться дальше, посещает всех соседей узла, поэтому его называют поиском в ширину. На рисунке 13.4 пока- зан обход в ширину, где узлы помечены в соответствии с тем, в каком поряд- ке их минуют.

Как и в случае с обходом в глубину, вы можете изучить рисунок 13.4, чтобы увидеть, каким образом алгоритм посещал узлы сети. Начав с узла, помеченного как 0, он добавил в очередь узлы с метками 1, 2 и 3.

Поскольку очередь возвращает элементы по принципу «первый зашел — пер- вый вышел», следующим обрабатывается узел 1, соседи которого помещаются в ту же очередь. Единственный пока еще не посещенный сосед этого узла име- ет номер 4.



**Рис. 13.4.** При обходе в ширину узлы, находящиеся ближе к начальной точке, посещаются раньше узлов, которые располагаются дальше от нее

Дальше алгоритм удаляет из очереди узел 2 и добавляет в нее соседа этого узла, помеченного как 5. После из очереди убирается узел 3, а вместо него туда помеща- ются его соседи под номерами 6 и 7.

Если хотите, можете продолжить изучение рисунка 13.4, чтобы разобраться, по- чему алгоритм рассмотрел узлы именно в таком порядке. Но уже сейчас вам долж- но быть видно, что все узлы, находящиеся ближе всего к начальной точке, посеща- ются раньше любых узлов, более удаленных от нее.

#### Проверка связности

Из алгоритмов обхода, описанных в двух предыдущих разделах, логически вы- текает еще несколько алгоритмов с незначительными изменениями. Например, ал- горитм обхода посещает все узлы, доступные из начальной точки. В ненаправлен- ной сети это означает, что посещенными окажутся все ее узлы (при условии, что она является связной). Отсюда следует простой алгоритм для определения связ- ности ненаправленной сети.

Boolean: IsConnected(Node: start\_node)

// Обходим сеть, начиная со start\_node. Traverse(start\_node)

// Смотрим, не остался ли какой-то узел непосещенным. For Each node In <all nodes>

If (Not node.Visited) Then Return False Next node

// Все узлы были посещены, значит, сеть связная. Return True

End IsConnected

Здесь используется предыдущий алгоритм обхода; при этом у каждого узла проверяется свойство Visited, чтобы понять, посещался ли он ранее.

Вы можете расширить приведенный способ обхода для поиска всех связных компонентов в сети. Просто повторяйте обход, пока не посетите все узлы. В сле- дующем псевдокоде показан алгоритм, который использует обход в глубину для обнаружения связных компонентов в сети.

List(Of List(Of Node)): GetConnectedComponents

// Следим за количеством посещенных узлов. Integer: num\_visited = 0;

// Создаем результат в виде списка списков. List(Of List(Of Node)): components

// Повторяем, пока все узлы не попадут в связный компонент. While (num\_visited < <number of nodes>)

// Находим узел, который еще не посещали.

Node: start\_node = <first node not yet visited>

// Добавляем начальный узел в стек. Stack(Of Node): stack stack.Push(start\_node) start\_node.Visited = True num\_visited = num\_visited + 1

// Добавляем узел в новый связный компонент. List(Of Node): component components.Add(component) component.Add(start\_node)

// Обрабатываем стек, пока он не станет пустым. While <stack isn’t empty>

// Берем из стека следующий узел.

Node: node = stack.Pop()

// Обрабатываем звенья узла. For Each link In node.Links

// Используем звено, только если целевой узел еще не посещали.

If (Not link.Nodes[1].Visited) Then

// Помечаем узел как посещенный. link.Nodes[1].Visited = True

// Помечаем звено как часть дерева. link.Visited = True

num\_visited = num\_visited + 1

// Добавляем узел к текущему связному компоненту. component.Add(link.Nodes[1])

// Заносим узел в стек. stack.Push(link.Nodes[1])

End If Next link

End // While <stack isn’t empty>

Loop // While (num\_visited < <number of nodes>)

// Возвращаем компоненты. Return components

End GetConnectedComponents

Этот алгоритм возвращает список списков, каждый из которых хранит узлы в связном компоненте. Сначала он создает переменную num\_visited для отсле- живания количества посещенных узлов. Затем генерирует список списков, который будет возвращаться.

Дальше идет цикл, продолжающийся до тех пор, пока не будет посещен каж- дый узел. Внутри этого цикла программа ищет узлы, которые она еще не исследо- вала, и добавляет их сначала в стек (так же как в алгоритме обхода), а потом в но- вый список, представляющий связный компонент узла.

После этого программа входит в цикл, подобный тому, который использовался ранее в алгоритме обхода. В нем стек обрабатывается до тех пор, пока не станет пу- стым. Единственная разница заключается в том, что в данном случае рассмотренные узлы добавляются не только в стек, но и в список, который формируется в процессе.

Когда стек становится пустым, это говорит о том, что алгоритм побывал во всех узлах, связанных с начальной точкой. На этом этапе он находит другой узел, кото- рый еще не был посещен, и запускает процесс по новой.

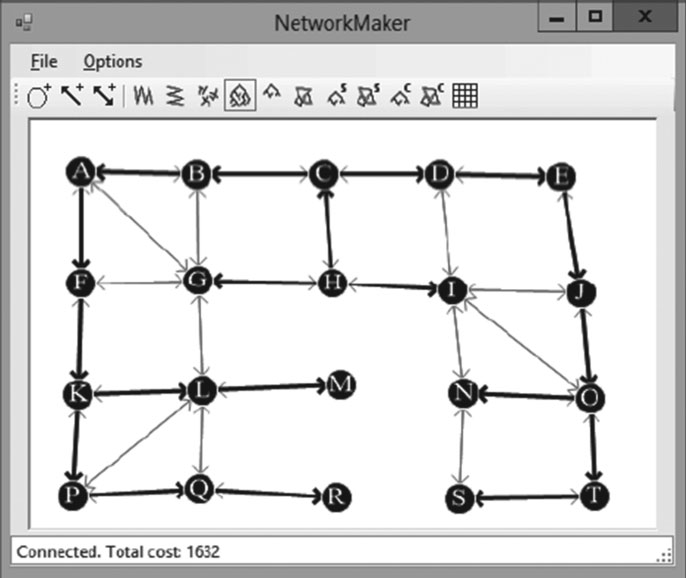
Исследовав все узлы, алгоритм возвращает список связных компонентов.

#### Остовные деревья

Если ненаправленная сеть является связной, вы можете создать дерево с корнем в любом узле, демонстрируя тем самым путь от корня к любому другому узлу сети. Такое дерево называется *остовным*, поскольку оно охватывает все сетевые узлы.

Например, на рисунке 13.5 показано остовное дерево для сети с корнем в узле

*H*. Двигаясь по темным звеньям, можно отследить путь от корневого узла *H* к лю- бой другой точке сети. Например, путь к узлу *M* проходит через точки *H*, *C*, *B*, *A*, *F*, *K*, *L* и *M*.



**Рис. 13.5.** Остовное дерево, соединяющее все узлы в сети

На самом деле алгоритмы обхода, описанные ранее, обнаруживают остовные де- ревья — они просто не запоминают, какие звенья дерева были использованы. Что- бы это исправить, достаточно дописать следующие строчки сразу после выраже- ния, которое помечает новый узел как посещенный.

// Помечаем звено как часть остовного дерева. link.Visited = True

В сущности, алгоритм начинает свою работу с корневого узла в остовном дере- ве. С каждым шагом он выбирает очередной узел, который является смежным по отношению к этому дереву, и добавляет его туда. Новый алгоритм просто записы- вает сведения о том, какие звенья были использованы при формировании остов- ного дерева, соединяющего узлы.

#### Минимальные остовные деревья

Алгоритм остовного дерева, рассмотренный в предыдущем разделе, позволяет использовать любой узел в качестве корневого, поэтому вариантов остовного де- рева может быть множество.

Остовное дерево, которое имеет наименьшие затраты из возможных, называет- ся *минимальным*. Стоит отметить, что сеть может иметь сразу несколько минималь- ных остовных деревьев. В сущности, если все звенья сети характеризуются одина- ковыми затратами, минимальным считается любое остовное дерево.

Ниже пошагово описан простой высокоуровневый алгоритм, позволяющий на- ходить минимальное остовное дерево с корневым узлом *R*.

1. Добавляем корневой узел *R* в исходное остовное дерево.
2. Повторяем, пока все узлы не окажутся в остовном дереве.
   1. Находим наименее затратное звено, соединяющее узел остовного дерева с узлом, который в это дерево пока не входит.
   2. Добавляем в остовное дерево целевой узел этого звена.

Такой алгоритм является «жадным», потому что на каждом этапе ищет звено с самыми низкими затратами из возможных. Делая оптимальный выбор на локаль- ном уровне, он достигает наилучшего результата в целом.

Ознакомимся, например, с сетью, показанной в левой части рисунка 13.6, и пред- положим, что звенья и узлы, выделенные толстыми линиями, являются частью формирующегося остовного дерева с корнем в узле *A*. На шаге 2a мы рассмотрим звенья, соединяющие узлы внутри дерева с теми, которые пока еще находятся за его пределами. В нашем примере эти звенья имеют затраты 15, 10, 12 и 11. С помо- щью «жадного» алгоритма добавим наименее затратное звено (в этом случае 10), чтобы получить дерево, представленное справа.

Больше всего времени занимает шаг 2a, где выполняется поиск следующего звена для дерева. То, насколько долго это будет происходить, зависит от исполь- зуемого вами подхода.



*C*



*C*



*A*

9

*B*

10

10

15

11

*D*

10

*E*

11

*F*

12

13

*G* 12

*H*



*A*

9

*B*

10

10

15

11

*D*

10

*E*

11

*F*

12

13

*G* 12

*H*

**Рис. 13.6.** На каждом шаге в остовное дерево добавляется наименее затратное звено, которое соединяет узел внутри дерева с узлом, находящимся за его пределами

Один из способов нахождения наименее затратного звена заключается в ци- клическом переборе узлов дерева и поиске среди их звеньев такого, которое ведет к узлу за пределами дерева и имеет минимальные затраты. Это достаточно мед- ленный подход, поскольку алгоритм вынужден проверять звенья узлов много раз, даже если они ведут к узлам, которые уже располагаются в дереве.

Оптимальнее всего хранить список потенциально подходящих звеньев. Присо- единяя узел к формирующемуся остовному дереву, алгоритм добавляет в список любые звенья этого узла, которые располагаются за пределами дерева. Поиск ми- нимального звена осуществляется внутри этого списка. Если в процессе алгоритм обнаруживает звено, ведущее к узлу, который уже находится внутри дерева (по- тому что узел мог стать частью дерева уже после того, как звено попало в список), он удаляет его из списка. Таким образом, алгоритму не придется повторно рассма- тривать это звено в дальнейшем. Когда список потенциально подходящих звеньев иссякнет, алгоритм завершит свою работу.

## Поиск путей

Поиск путей в сети является распространенной задачей. Обычный пример — на- хождение маршрута из одной точки транспортной сети в другую.

В следующих разделах описываются некоторые алгоритмы для поиска путей в рамках сети.

#### Поиск произвольного пути

Алгоритмы остовного дерева, рассмотренные ранее в этой главе, предостав- ляют средство для нахождения пути между любыми двумя узлами в сети. Следу- ющие шаги описывают простой высокоуровневый алгоритм для поиска пути из узла *A* в узел *B*.

1. Находим остовное дерево с корнем в узле *A*.
2. Проходим обратно по звеньям остовного дерева от узла *B* к узлу *A*.
3. Меняем порядок прохождения звеньев на противоположный. Приведенный алгоритм генерирует остовное дерево с узлом *A* в качестве кор-

ня, а затем переходит к узлу *B*. Для каждой точки на своем пути он обнаруживает звено в рамках остовного дерева, которое ведет к этой точке. Записав найденное звено, переходит к следующему узлу на своем пути.

К сожалению, поиск звена, которое ведет к определенному узлу в остовном де- реве, — довольно сложная задача. При использовании соответствующих алгорит- мов, описанных ранее, придется циклически перебрать каждое звено, чтобы узнать, является ли оно частью остовного дерева и отсылает ли к текущему узлу.

Для решения этой проблемы достаточно немного подкорректировать алгоритм остовного дерева. Вначале нужно добавить свойство FromNode в класс Node. Затем, когда алгоритм отмечает, что узел находится внутри дерева, необходимо присвоить его свойству FromNode ссылку на узел, чье звено использовалось для соединения нового узла с деревом.

Теперь, чтобы на шаге 2 найти путь из узла *B* в узел *A*, можно просто следовать по свойствам FromNode этих узлов.

#### Поиск кратчайшего пути с помощью установки меток

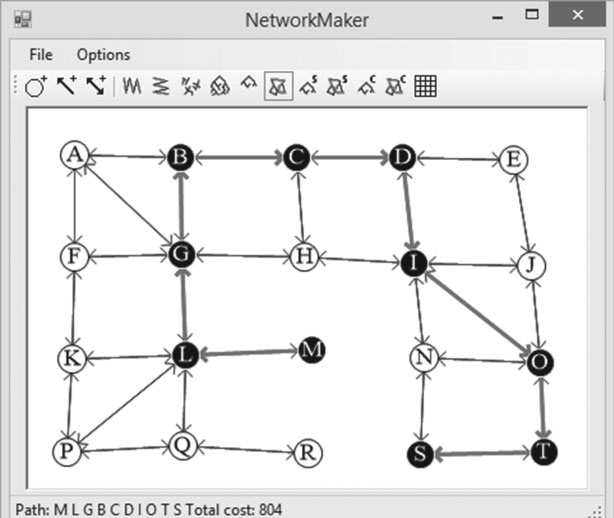
Алгоритм, описанный в предыдущем разделе, находит путь от начального к целевому узлу, однако этот путь может оказаться не самым оптимальным. Он берется из остовного дерева, что не гарантирует его высокую эффективность. На рисунке 13.7 показан путь от узла *M* к узлу *S*. Если представить затраты зве- ньев в виде их длины, можно довольно легко найти маршрут покороче, такой как *M* ➔ *L* ➔ *G* ➔ *H* ➔ *I* ➔ *N* ➔ *S*.

Для нахождения кратчайшего пути между двумя узлами понадобится более практичный алгоритм. Такие алгоритмы делятся на две категории: установочные и корректирующие. Алгоритм коррекции меток будет описан в следующем разде- ле, а пока что мы рассмотрим их установку.

Этот алгоритм начинает свою работу в корневом узле, создавая остовное дерево способом, который напоминает процесс формирования минимального остовного дерева, описанный ранее. На каждом шаге выбирается наименее за- тратное звено, которое соединяет новый узел с деревом. Алгоритм кратчайше- го пути действует иначе, выбирая звено, которое ведет к ближайшему от кор- невой точки узлу.

Чтобы определить, какой из узлов является наименее удаленным, алгоритм маркирует каждый из них с помощью расстояния до начальной точки. Проверяя звено, он добавляет расстояние до его исходного узла к его затратам и затем опре- деляет текущее расстояние до целевого узла звена.

Добавив все узлы в остовное дерево, алгоритм завершает свою работу. Пути внутри дерева представляют собой кратчайшие маршруты от начального узла к лю- бой точке сети, поэтому результат называется *деревом кратчайшего пути*.



**Рис. 13.7.** Путь, который проходит остовное дерево от одного узла к другому, может быть неоптимальным

Ниже приводится высокоуровневое описание этого алгоритма.

1. Устанавливаем расстояние начального узла равным 0 и помечаем его как часть дерева.
2. Добавляем звенья начального узла в список потенциально подходящих зве- ньев, которые могут быть использованы для расширения дерева.
3. Циклически перебираем и проверяем звенья внутри списка, пока тот не ока- жется пустым.
   1. Если звено ведет к узлу, который уже находится в дереве, убираем его из списка.
   2. Допустим, звено *L* соединяет узел *N*1 внутри дерева с узлом *N*2, который в это дерево пока что не входит. Если *D*1 — это расстояние до узла *N*1, вхо- дящего в дерево, а *C*1 — затраты звена, тогда вы можете добраться до узла *N*2, расположенного на расстоянии *N*1 + *CL*, пройдя сначала к узлу *N*1 и за- тем следуя по звену. Пусть *D*2 = *N*1 + *CL* будет потенциальным расстояни- ем для узла *N*1, который использует это звено. Перебирая звенья в спи- ске, мы будем отслеживать звено и узел (назовем их *Lbest* и *Nbest*), которые обеспечивают наименьшее расстояние из возможных (назовем его *Dbest*).
   3. Устанавливаем расстояние для *Nbest* равным *Dbest* и маркируем *Nbest* как часть дерева кратчайшего пути.
   4. Любое из звеньев *L*, исходящих из *Nbest*, которое ведет к узлу за пределами дерева, добавляется в список потенциально подходящих звеньев.

Рассмотрим, например, сеть, показанную на рисунке 13.8. Допустим, звенья и узлы, выделенные толстыми линиями, являются частью формирующегося дере- ва кратчайшего пути. Метки на узлах обозначают их расстояние до корневой точ- ки, которая помечена как 0.



19



19



0

9

9

10

10

15

11

10 10

15

11

*F*

12

13

*G* 12

*H*



0

9

9

10

10

15

11

10 10

15

11

*F*

12

13

22 12

*H*

**Рис. 13.8.** На каждом шаге к дереву кратчайшего пути добавляется звено, формирующее самое короткое итоговое расстояние от корня к узлу, который не входит в дерево

Чтобы добавить следующее звено в дерево, проверим звенья, которые ведут от дерева к узлам за его пределами, и вычислим расстояние до этих узлов. В данном примере имеются три потенциально подходящих звена.

Первое ведет от узла с меткой 19 к узлу *F*. Расстояние от начальной точки до узла, помеченного как 19, равно 19 (отсюда и метка), а затраты звена составля- ют 11. Следовательно, итоговое расстояние до узла *F* через это звено равняется 19 + 11 = 30.

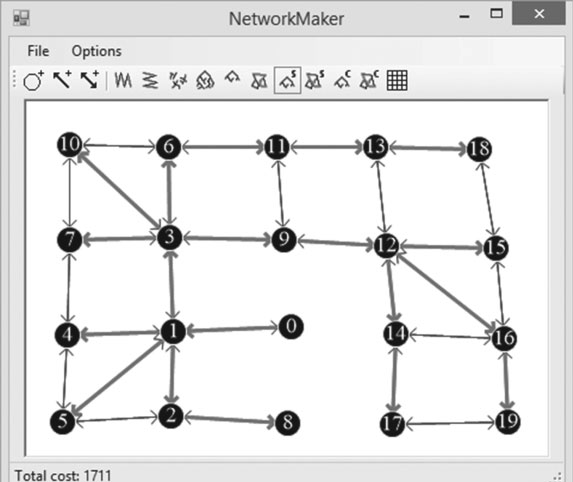
Второе звено ведет от узла с меткой 15 к узлу *F*. Расстояние от начальной точки до узла, помеченного как 15, равно 15, а затраты звена составляют 11. Таким обра- зом, итоговое расстояние до узла *F* через это звено равняется 15 + 11 = 26.

Третье звено ведет от узла с меткой 10 к узлу *G*. Его затраты составляют 12, по- этому итоговое расстояние через него будет равно 10 + 12 = 22. Именно это звено должно быть добавлено в дерево, поскольку оно обеспечивает наименьшее из трех вычисленных расстояний. Результат показан в правой части рисунка 13.8.

На рисунке 13.9 изображен весь кратчайший путь, сгенерированный этим алго- ритмом. В данной сети затраты ссылок выражены через их длину в пикселях. Каж- дый узел помечен порядковым номером, в соответствии с которым он был добав- лен в дерево. Корневой узел вставлялся первым, поэтому он имеет номер 0; узел слева от него был добавлен следующим, поэтому он отмечен номером 1 — и т. д.

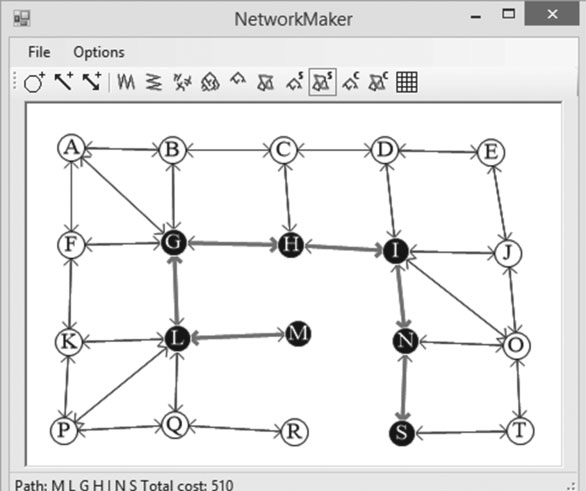
Обратите внимание на то, как значения меток увеличиваются по мере удаления от корневой точки. Это похоже на порядок добавления узлов в дерево при обходе в ширину. Разница лишь в том, что тогда алгоритм исходил из количества звеньев между узлами и корневой точкой, а в нашем случае учитывается расстояние, прой- денное по звеньям от корневой точки до узлов.

Построив дерево кратчайшего пути, вы можете проследовать по значениям FromNode, принадлежащим его узлам, чтобы найти обратный путь от целевого узла к начальной точке, как было описано в предыдущем разделе.



**Рис. 13.9.** Дерево кратчайшего пути, которое находит самый короткий маршрут от корневой точки до любого узла в сети

На рисунке 13.10 показан самый короткий путь из узла *M* в узел *S* в рамках ис- ходной сети. Он выглядит более рациональным, нежели маршрут, построенный с помощью остовного дерева (см. рис. 13.7).



**Рис. 13.10.** Путь по дереву кратчайшего пути — самый короткий маршрут из корня в определенный узел сети

#### Поиск кратчайшего пути с помощью коррекции меток

Больше всего времени в алгоритме кратчайшего пути, использующего установку меток, отнимает поиск следующего звена, которое нужно добавить к дереву. Для этого алгоритм должен перебрать все потенциально подходящие звенья, чтобы найти то, которое ведет к новому узлу с наименьшими затратами.

В качестве альтернативной стратегии к дереву кратчайшего пути можно доба- вить любое из потенциально подходящих звеньев, пометив как обычно его целевой узел величиной затрат, возникающих при проходе к корню.

Позже, при рассмотрении звеньев из списка потенциально подходящих, алго- ритм находит лучший путь к узлу, который уже является частью дерева кратчай- шего пути. В этом случае он обновляет расстояние до узла, помещает его звенья обратно в список (если их там еще нет) и продолжает работу.

В алгоритме установки меток расстояние до узла назначается только один раз и никогда не меняется. В алгоритме коррекции меток расстояние тоже устанавли- вается лишь единожды, но позже оно может быть неоднократно исправлено.

Ниже представлено высокоуровневое описание этого алгоритма.

1. Устанавливаем расстояние до начального узла равным 0 и помечаем его как часть дерева.
2. Добавляем звенья начального узла в список потенциально подходящих зве- ньев.
3. Пока список не окажется пустым:
   1. рассматриваем первое звено в списке потенциально подходящих звеньев;
   2. вычисляем расстояние до целевого узла звена: <расстояние> = <рассто- яние до исходного узла> + <затраты звена>;
   3. если новое расстояние оказывается лучше текущего:
   4. обновляем расстояние до целевого узла;
   5. добавляем все звенья целевого узла в список потенциально подходящих звеньев.

Данный алгоритм может показаться более сложным, но на самом деле его код выглядит компактнее, потому что вам не нужно перебирать список потенциально подходящих звеньев в поисках лучшего.

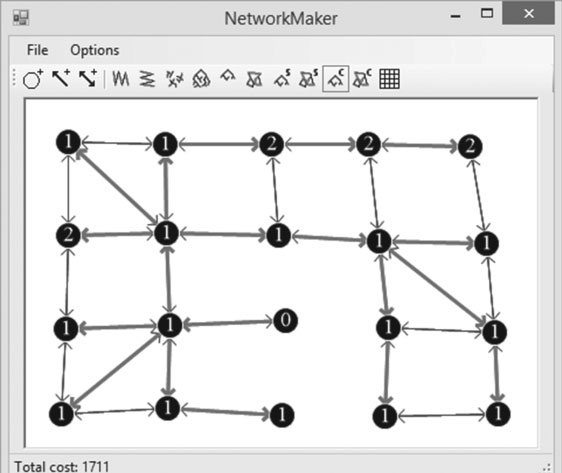
Поскольку этот алгоритм может несколько раз изменить звено, ведущее к узлу, у вас нет возможности отметить, что звенья используются деревом, и оставить их в таком виде. Если вы меняете путь к узлу, вам необходимо найти старое звено и убрать с него метку.

В качестве более простого решения можно было бы добавить в класс Node

свойство FromLink. Оно будет меняться вместе со звеном, ведущим к узлу.

Если вы все же хотите пометить звенья, используемые деревом кратчайшего пути, сначала постройте это дерево, а потом циклически переберите его узлы и при- свойте метки звеньям из соответствующих свойств FromLink.

На рисунке 13.11 показана сеть с деревом кратчайшего пути, найденным с по- мощью метода коррекции меток. Как и прежде, затраты звеньев в этой сети выра- жены через их длину в пикселах. Каждый узел содержит метку, показывающую, сколько раз было исправлено расстояние до него (значение свойства FromLink). Корневой узел обозначен числом 0, поскольку его значение было установлено в са- мом начале и никогда не менялось.



**Рис. 13.11.** В алгоритме коррекции меток расстояния до некоторых узлов могут корректироваться несколько раз

Многие из узлов на рисунке 13.11 помечены числом 1. Это означает, что рас- стояние к ним было установлено лишь один раз и никогда не корректировалось. Несколько узлов имеет метку 2, что для нас значит следующее: их значения изме- нялись всего один раз после установки.

В крупной и более сложной сети расстояние до узлов может корректировать- ся много раз, прежде чем дерево кратчайшего пути будет окончательно достроено.

#### Поиск кратчайшего пути между всеми парами вершин

Алгоритмы, которые были описаны до этого момента, находили дерево крат- чайшего пути от начальной точки ко всем остальным узлам в сети. Но существует еще один алгоритм этого типа, который ищет кратчайший путь между каждой парой сетевых узлов.

Алгоритм Флойда — Уоршелла начинается с двумерного массива под назва- нием Distance, где Distance[start\_node, end\_node] является самым ко- ротким расстоянием между узлами start\_node и end\_node.

Чтобы создать массив, его нужно инициализировать, устанавливая элементам по диагонали (представляющим расстояние от самого узла) значения 0. Элемен- там, которые выступают прямыми звеньями между двумя узлами, присваивают- ся величины затрат этих звеньев. Остальные элементы массива должны быть рав- ны бесконечности.

Предположим, что массив Distance частично заполнен. Рассмотрим внутри него путь от узла start\_node к узлу end\_node. Представим себе также, что для некоего значения via\_node путь использует только узлы 0, 1, 2…, via\_node – 1.

Добавление узла via\_node может сократить путь только в том случае, если этот узел находится где-то в середине улучшенного пути. Другими словами, марш- рут start\_node --+ end\_node превращается в start\_node --+ via\_node, за ко- торым идет via\_node --+ end\_node.

Чтобы обновить массив Distance, нужно проверить все пары узлов start\_node и end\_node. Если Distance[start\_node, end\_node] > Distance[start\_node, via\_node] + Distance[via\_node, end\_node], элемент Distance[start\_node, end\_node] обновляется путем присваивания ему более короткого расстояния.

Если сделать то же самое для via\_node = 0, 1, 2..., *N* – 1, где *N* — число узлов в сети, то массив будет хранить итоговое кратчайшее расстояние между любыми двумя узлами, используя другие узлы в качестве промежуточных точек в опти- мальных путях.

Пока что алгоритм не решает задачу нахождения самого короткого пути между двумя узлами. Он всего лишь объясняет, как искать оптимальное расстояние меж- ду ними. К счастью, информацию о пути можно добавить, создав еще один двумер- ный массив под названием Via.

Массив Via отслеживает определенный узел на пути от одной точки к другой. Иными словами, Via[start\_node, end\_node] хранит индекс узла, который вы должны посетить, проходя кратчайшее расстояние от start\_node до end\_node.

Если Via[start\_node, end\_node] равно end\_node, это означает, что от start\_node к end\_node ведет одно звено, поэтому кратчайший путь состоит только из узла end\_node.

Если Via[start\_node, end\_node] равно какому-то другому узлу via\_node, вы можете рекурсивно перебрать массив, чтобы найти путь сначала от start\_node к via\_node, а затем от via\_node к end\_node (если это кажется вам немного за- путанным, рассмотрение алгоритма для использования массива Via должно будет все прояснить).

Чтобы создать массив Via, его нужно инициализировать элементами со зна- чениями.

Затем, если узлы соединяются напрямую, Via[start\_ node, end\_node]

следует присвоить значение end\_node.

Теперь, когда мы создали массив Distance и оптимизировали расстояние start\_node --+ end\_node, заменив его путями start\_node --+ via\_node и via\_node --+ end\_node, нам нужно сделать еще кое-что. Мы должны при- своить Via[start\_node, end\_node] = via\_node, чтобы сигнализировать о том, что кратчайший путь от start\_node к end\_node проходит через про- межуточный узел via\_node.

Ниже приведено полное пошаговое описание алгоритма для построения мас- сивов Distance и Via (исходя из того, что сеть состоит из *N* узлов).

1. Инициализируем массив Distance:
   1. всем элементам Distance[i, j] присваивается бесконечность;
   2. для всех *i* от 1 до *N* – 1 присваиваем Distance[i, i] значение 0;
   3. если узлы *i* и *j* соединены звеном *i* ➔ *j*, присваиваем Distance[i, j] за- траты этого звена.
2. Инициализируем массив Via. Для всех *i* и *j*:
   1. если Distance[i, j] меньше бесконечности, присваиваем Via[i, j]

значение *j*, сигнализируя тем самым, что путь от *i* к *j* идет через узел *j*;

* 1. в противном случае присваиваем Via[i, j] значение –1, чтобы указать на то, что из узла *i* в узел *j* нет пути.

1. Выполняем следующие вложенные циклы, чтобы найти улучшения в пути.

For via\_node = 0 To N - 1

For from\_node = 0 To N - 1

For to\_node = 0 To N - 1

Integer: new\_dist =

Distance[from\_node, via\_node] + Distance[via\_node, to\_node]

If (new\_dist < Distance[from\_node, to\_node]) Then

// Это улучшенный путь. Обновляем его. Distance[from\_node, to\_node] = new\_dist Via[from\_node, to\_node] = via\_node

End If Next to\_node

Next from\_node Next via\_node

Цикл via\_node перебирает индексы узлов, которые могут оказаться проме- жуточными, и оптимизирует имеющиеся маршруты. Когда он закончит свою ра- боту, все кратчайшие пути будут готовы.

В следующем псевдокоде показано, как с помощью окончательных версий мас- сивов Distance и Via искать узлы в рамках кратчайшего пути от начального узла к целевому.

List(Of Integer): FindPath(Integer: start\_node, Integer: end\_node) If (Distance[start, end] == infinity) Then

// Между этими узлами нет пути. Return null

End If

// Получаем промежуточный узел для этого пути. Integer: via\_node = Via[start\_node, end\_node]

// Проверяем, есть ли прямое соединение. If (via\_node == end\_node)

// Прямое соединение есть.

// Возвращаем список, содержащий только end\_node. Return { end\_node }

Else

// Прямого соединения нет.

// Возвращаем start\_node --> via\_node плюс via\_node --> end\_ node.

Return

{

}

End If

FindPath(start\_node, via\_node] + FindPath(via\_node, end\_node]

End FindPath

Рассмотрим, например, сеть, представленную на рисунке 13.12. Верхние масси- вы показывают то, как со временем трансформируется значения Distance; ниж- ние массивы демонстрируют изменения значений Via. Элементы, подвергшиеся перестройке, выделены, чтобы их было легче заметить.

4



*A*



*B* 10 *C*

6 3

*D*

*A B C D*

*A B C D*

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 4 |  |  |
| 4 | 0 | 10 | 6 |
|  | 10 | 0 | 3 |
|  | 6 | 3 | 0 |

*A B C D*

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* |  |  |
| *A* | *B* | *C* | *D* |
|  | *B* | *C* | *D* |
|  | *B* | *C* | *D* |

*A B C D A B C D A B C D A B C D*

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 4 |  |  |
| 4 | 0 | 10 | 6 |
|  | 10 | 0 | 3 |
|  | 6 | 3 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 4 | 14 10 | |
| 4 | 0 | 10 | 6 |
| 14  10 | 10 | 0 | 3 |
| 6 | 3 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 4 | 14 | 10 |
| 4 | 0 | 10 | 6 |
| 14 | 10 | 0 | 3 |
| 10 | 6 | 3 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 4 | 13 | 10 |
| 4 | 0 | 9 | 6 |
| 13 | 9 | 0 | 3 |
| 10 | 6 | 3 | 0 |

Оптимизация с помощью узла *A*

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* |  |  |
| *A* | *B* | *C* | *D* |
|  | *B* | *C* | *D* |
|  | *B* | *C* | *D* |

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *B B* | |
| *A* | *B* | *C* | *D* |
| *B*  *B* | *B* | *C* | *D* |
| *B* | *C* | *D* |

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *B* | *B* |
| *A* | *B* | *C* | *D* |
| *B* | *B* | *C* | *D* |
| *B* | *B* | *C* | *D* |

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *D*  *D* | *B* |
| *A* | *B* | *D* |
| *D* | *D* | *C* | *D* |
| *B* | *B* | *C* | *D* |

Оптимизация с помощью узла *B*

Оптимизация с помощью узла *C*

Оптимизация с помощью узла *D*

**Рис. 13.12.** Кратчайшие пути между всеми парами узлов

в сети можно представить в виде массивов Distance (сверху) и Via (снизу)

Таблица в левом верхнем углу показывает исходные значения в массиве Distance. Расстояние от любого узла к самому себе равно 0. Расстояние между двумя узлами, соединенными звеном, устанавливается в размере затрат этого звена. Например, звено между узлами *A* и *B* имеет затраты 4, поэтому Distance[A, B] равно 4 (чтобы пример был более простым для понимания, имена узлов исполь- зуются в качестве индексов массива). Остальным элементам присваивается бес- конечность.

Таблица в левом нижнем углу показывает исходные значения в массиве Via.

Например, от узла *C* к узлу *B* ведет звено, поэтому Via[C, B] равно *B*.

После инициализации массивов алгоритм пытается выполнить оптимизацию. Сначала он ищет пути, которые удастся улучшить, используя узел *A* в качестве про- межуточной точки. Однако этот узел находится на краю сети, поэтому он не может оптимизировать никакой путь.

Затем алгоритм пытается улучшить пути с помощью узла *B* и находит четыре варианта для оптимизации. Например, взглянув на второй массив

Distance, легко заметить, что Distance[A, C] содержит бесконечность, од- нако Distance[A, B] и Distance[B, C] равны соответственно 4 и 10. Та- ким образом, путь *A* ➔ *C* может быть улучшен. Чтобы выполнить оптимиза- цию, алгоритм присваивает Distance[A, C] значение 4 + 10 = 14, а Via[A, C] делает равным узлу *B*.

Изменения можно проследить, глядя прямо на сеть. Для начального пути *A* ➔ *C* было установлено бесконечное расстояние. По сравнению с ним путь *A* ➔ *B* ➔ *C* является улучшением, и, посмотрев на сеть, вы можете заметить, что итоговая длина этого пути равна 14.

Точно так же можно пройтись по изменениям в путях *A* ➔ *D*, *C* ➔ *A* и *D* ➔ *A*.

Дальше алгоритм пытается улучшить пути, используя узел *C* в качестве про- межуточного. Этот узел не позволяет ничего оптимизировать, так как он находит- ся на краю сети.

Наконец, алгоритм пробует в качестве промежуточного узел *D*. С его помощью можно улучшить четыре пути: *A* ➔ *C*, *B* ➔ *C*, *C* ➔ *A* и *C* ➔ *B*.

Примером, иллюстрирующим поиск пути внутри окончательной версии мас- сива Via, служит таблица в правом нижнем углу (см. рис. 13.12). Допустим, вы хотите найти путь от узла *A* к узлу *C*. Пошаговый процесс определения этого пути описан ниже.

1. Via[A, C] равно *D*, поэтому *A* ➔ *C* = *A* ➔ *D* + *D* ➔ *C*.
2. Via[A, D] равно *B*, поэтому *A* ➔ *D* = *A* ➔ *B* + *B* ➔ *D*.
   1. Via[D, C] равно *C*, поэтому от узла *D* к узлу *C* ведет звено.
   2. Via[A, B] равно *B*, поэтому от узла *A* к узлу *B* ведет звено.
   3. И наконец, Via[B, D] равно *D*, поэтому от узла *B* к узлу *D* ведет звено. Заключительный маршрут проходит через узлы *B*, *D* и *C*, следовательно, ито-

говый путь будет выглядеть как *A* ➔ *B* ➔ *D* ➔ *C*.

На рисунке 13.13 представлены рекурсивные вызовы.

Создав массивы Distance и Via, вы можете быстро найти кратчайший путь между любыми двумя точками в сети. Недостаток состоит в том, что массивы не- редко занимают много места.

Например, сеть улиц в относительно большом городе может достигать 30 000 узлов, поэтому в массиве пришлось бы хранить 2  30 0002 = 1,8 млрд эле- ментов. Если элементы являются 4-байтовыми целыми числами, то массив зай- мет в памяти 7,2 Гбайт.

Даже если вы можете себе позволить такие объемы памяти, алгоритм для построения массивов использует три вложенных цикла, которые проходят от 1 до *N*, где *N* — количество узлов, поэтому общее время работы алгоритма будет рав- но *O*(*N*3). Если *N* достигает 30 000, это выливается в 2,7  1013 итераций. Чтобы сге- нерировать такие массивы, компьютеру, который выполняет 1 млн итераций в се- кунду, понадобится 10 месяцев.



*D**C*=*C*

*A**B*=*B*

*B**D*=*D*

*A**D*=*A**B*+*B**D*

*A**C*=*A**D*+*D**C*

**Рис. 13.13.** Для поиска пути от start\_node к end\_node через промежуточную точку

via\_node нужно рекурсивно найти маршрут от start\_node к via\_node

и от via\_node к end\_node

Этот алгоритм не получится использовать для по-настоящему больших сетей, поэтому при необходимости для нахождения маршрутов следует прибегнуть к дру- гим алгоритмам кратчайшего пути. Если нужно искать множество путей в рамках небольшой сети, состоящей всего лишь из нескольких сотен узлов, вы, вероятно, сможете сэкономить некоторое время, применив рассматриваемый алгоритм для предварительного вычисления всех кратчайших маршрутов.

## Резюме

Большинство алгоритмов, описанных в этой главе, занимается обходом сети. Алгоритмы обхода в глубину и ширину имеют разный порядок посещения сете- вых узлов. Алгоритмы связности, остовного дерева, минимального остовного дере- ва и кратчайшего пути тоже минуют сеть различными способами. Например, ми- нимальное остовное дерево перебирает звенья в порядке их затратности, а поиск кратчайшего пути с помощью установки меток учитывает расстояние до целево- го узла звена.

Кратчайший путь между всеми парами вершин вычисляется несколько иначе. Вместо перебора сети он формирует коллекцию из *N* деревьев кратчайшего пути, каждое из которых позволяет обойти сеть.

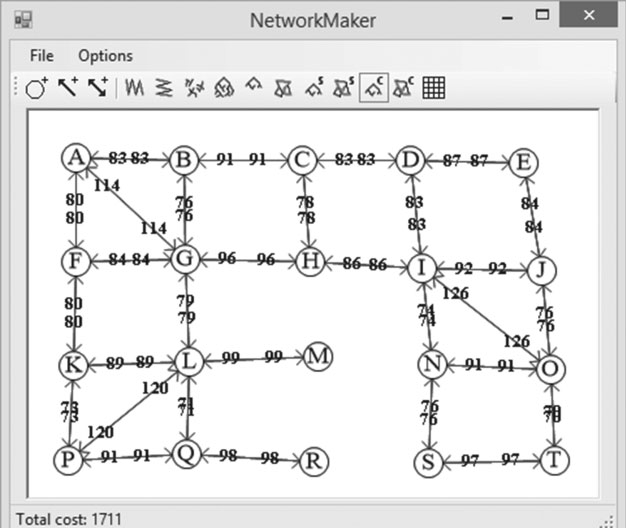
В следующей главе мы продолжим разговор о сетях. Будут рассмотрены более эффективные алгоритмы, позволяющие решать реальные проблемы — например, расстановку задач, раскрашивание карт и распределение рабочих мест.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

1\*. Напишите программу, похожую на ту, что показана на рисунке 13.14. Она должна позволять вам создавать, сохранять и загружать тестовые сети. Кноп- ки на панели инструментов дают возможность пользователю добавлять узлы,

одно- и двунаправленные звенья (на самом деле это два звена, соединяющие нажатые узлы в обоих направлениях). Меню File должно иметь пункты New, Open и Save As для создания, сохранения и загрузки сетей (если вы использу- ете язык C# и не хотите разрабатывать все приложение целиком, можете за- грузить пример программы с веб-сайта этой книги и заменить ее алгоритми- ческий код своим собственным).



**Рис. 13.14.** Демонстрационная программа NetworkMaker позволяет создавать, сохранять и загружать тестовые сети

1. Расширьте программу из упражнения 1 таким образом, чтобы пользователь мог выполнять обход сети. При выборе инструмента обхода и щелчке по узлу должна отображаться схема обхода.
2. Расширьте программу из упражнения 1, добавив в нее инструмент, который отображает связные компоненты сети.
3. Подходит ли алгоритм поиска связных компонентов для направленных се- тей? Почему?
4. Расширьте программу из упражнения 1, добавив в нее инструмент, что ищет и отображает остовное дерево с корнем в узле, по которому щелкнул поль- зователь.
5. В подразделе «Минимальные остовные деревья» раздела «Обход сети» в те- кущей главе сказано, что в сетях, где все звенья имеют одинаковые затраты, любое остовное дерево является минимальным. Почему это так?
6. Расширьте программу из упражнения 1, добавив в нее инструмент, что ищет и отображает минимальное остовное дерево с корнем в узле, по которому щелкнул пользователь.
7. Расширьте программу из упражнения 1, добавив в нее инструмент, который с помощью остовного дерева ищет и отображает пути между двумя узлами, выбранными пользователем.
8. Всегда ли дерево кратчайшего пути является минимальным остовным дере- вом? Если да, то почему? Если нет, нарисуйте обратный пример.
9. Расширьте программу из упражнения 1, добавив в нее инструмент, который ищет и отображает дерево кратчайшего пути с установкой меток. Его корень должен находиться в узле, по которому щелкнул пользователь.
10. Расширьте программу из упражнения 1, добавив в нее инструмент, который использует дерево кратчайшего пути с установкой меток для поиска и ото- бражения маршрута между двумя выбранными узлами.
11. Расширьте программу из упражнения 1, добавив в нее инструмент, который ищет и отображает дерево кратчайшего пути с коррекцией меток. Его корень должен находиться в узле, по которому щелкнул пользователь.
12. Расширьте программу из упражнения 1, добавив в нее инструмент, который использует дерево кратчайшего маршрута с коррекцией меток для поиска и отображения пути между двумя выбранными узлами.
13. Как поведет себя алгоритм поиска кратчайшего пути с коррекцией меток, если сеть содержит цикл с отрицательным общим весом? Как будет рабо- тать алгоритм установки меток?
14. Представьте, что вам нужно найти кратчайший путь между относитель- но близкими точками в крупной сети. Как это сделать с помощью алгорит- ма установки меток, не прибегая к построению целого дерева кратчайшего пути? Сэкономят ли время внесенные изменения?

16\*. Рассмотрим сценарий из упражнения 15. Как бы вы искали путь с помощью алгоритма коррекции меток, не создавая целое дерево кратчайшего пути? Помогло бы это сэкономить время?

17\*.Представьте, что вы едете в музей и ремонтные работы на дороге по- стоянно заставляют вас сворачивать с кратчайшего пути. При этом вам необходимо каждый раз вычислять новое дерево кратчайшего путь для нахождения оптимального маршрута к музею с учетом вашего нового местоположения. Каким образом можно было бы избежать этих повтор- ных вычислений?

18\*. Расширьте программу из упражнения 1, добавив в нее инструмент, который ищет и отображает для сети массивы Distance и Via. Проверьте ее на при- мере сети, похожей на ту, что показана на рисунке 13.12.

19\*. Расширьте программу из упражнения 18 так, чтобы она могла не только ис- кать кратчайший путь между всеми парами узлов в сети, но и отображать его. Проверьте ее на примере сети, похожей на ту, что показана на рисунке 13.12.

1. Допустим, при создании массивов Distance и Via с помощью алгоритма кратчайшего пути между всеми парами вершин ваш компьютер может вы- полнять 1 млн итераций в секунду. Сколько времени займет создание этих массивов для сети, содержащей 100 узлов? 1000 узлов? 10 000 узлов?
2. Используя сеть, показанную на рисунке 13.15, изобразите изменение масси- вов Distance и Via так, как это сделано на рисунке 13.13. Как выглядят ис- ходный и итоговый кратчайшие пути от узла *A* к узлу *C*?



*A*

10

18

*B*

6

*C*

11

20

*D*

**Рис. 13.15.** Массивы Distance и Via, которые необходимо изобразить для этой сети

# Глава 14

**ДОПОЛНИТЕЛЬНЫЕ СЕТЕВЫЕ АЛГОРИТМЫ**

Глава 13 была посвящена сетевым алгоритмам обхода, в том числе направ- ленным в глубину и ширину для поиска кратчайшего пути между двумя узлами. В этой главе мы продолжим рассматривать сетевые алгоритмы, начиная с относи- тельно простых, таких как топологическая сортировка и поиск циклов, и закан- чивая чуть более сложными, среди которых будут раскраска графа и вычисление максимального потока.

## Топологическая сортировка

Представьте, что вам нужно решить сложную задачу, состоящую из множе- ства действий, часть которых должна быть выполнена раньше других. Возьмем, например, ремонт кухни. Для начала вам может понадобиться разрешение мест- ных властей. Затем нужно будет заказать новую бытовую технику, перед установ- кой которой придется поменять проводку. Масштабный проект, такой как ремонт целого жилого или офисного помещения, обычно включает сотни шагов со слож- ным набором зависимостей.

В таблице 14.1 приведены некоторые зависимости, с которыми можно стол- кнуться во время ремонта кухни.

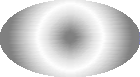
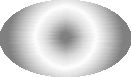
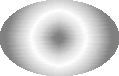
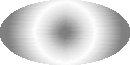
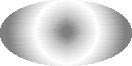
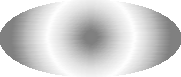
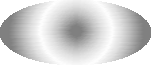
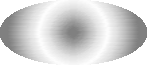
**Òàáëèöà 14.1.** Çàâèñèìîñòè, ñâÿçàííûå ñ ðåìîíòîì êóõíè

|  |  |
| --- | --- |
| **Задача** | **Предварительное требование** |
| Получение разрешения | — |
| Покупка бытовой техники | — |
| Установка бытовой техники | Покупка бытовой техники |
| Снос стен | Получение разрешения |
| Монтаж проводки | Снос стен |
| Установка гипсокартонных стен | Монтаж проводки |
| Прокладка труб | Снос стен |
| Начальная проверка | Монтаж проводки |
| Начальная проверка | Прокладка труб |
| Установка гипсокартонных стен | Прокладка труб |
| Установка гипсокартонных стен | Начальная проверка |
| Покраска стен | Установка гипсокартонных стен |
| Покраска потолка | Установка гипсокартонных стен |

|  |  |
| --- | --- |
| **Задача** | **Предварительное требование** |
| Отделка пола | Покраска стен |
| Отделка пола | Покраска потолка |
| Итоговая проверка | Отделка пола |
| Установка кафеля | Установка гипсокартонных стен |
| Установка освещения | Покраска потолка |
| Итоговая проверка | Установка освещения |
| Установка шкафов | Отделка пола |
| Итоговая проверка | Установка шкафов |
| Установка столешницы | Установка шкафов |
| Итоговая проверка | Установка столешницы |
| Отделка пола | Установка гипсокартонных стен |
| Установка бытовой техники | Отделка пола |
| Итоговая проверка | Установка бытовой техники |

Вы можете представить этот проект в виде сети, в которой звено ведет от зада- чи *A* к задаче *B*, если задача *B* должна быть выполнена перед *A*. На рисунке 14.1 по- казана сеть, представляющая задачи, которые перечислены в таблице 14.1.

*Частичное упорядочивание* — это набор зависимостей, который определяет по- рядок следования отдельных, но не обязательно всех содержащихся в нем объек- тов. Зависимости, которые перечислены в таблице 14.1 и показаны на рисунке 14.1, частично упорядочивают действия, связанные с ремонтом.



Получение разрешения

Прокладка труб

Плитка

Снос

Монтаж проводки

Установка гипсокартонных стен

Покупка бытовой техники

Начальная проверка

Покраска потолка

Покраска стен

Установка бытовой техники

Отделка пола

Установка шкафов

Установка освещения

Итоговая проверка

Установка столешницы

**Рис. 14.1.** Набор частично упорядоченных задач можно представить в виде сети

Если же вы хотите решить эти задачи, то необходимо превратить частичное упорядочивание в полное, чтобы действия шли друг за другом. Например, усло- вия, перечисленные в таблице 14.1, не содержат явного запрета на отделку пола пе- ред прокладкой труб, но если внимательно изучить таблицу или сеть, можно уви- деть, что эти задачи в такой последовательности выполнять нельзя (отделка пола должна проводиться после покраски стен, а та, в свою очередь, должна идти по- сле установки гипсокартона, которая требует предварительной прокладки труб).

*Топологическая сортировка* — это процесс превращения частичного упорядо- чивания сети в полное.

Есть один довольно простой алгоритм для расширения частичного упорядо- чивания. Если задачи можно выполнить в правильной последовательности, неко- торые из них должны быть такими, что не имеют предварительных условий и мо- гут быть решены в первую очередь. Найдите подобную задачу, добавьте ее к рас- ширенному упорядочиванию и удалите из сети. Затем повторите то же самое для аналогичных задач, пока все из них не будут выполнены.

Если на каком-то этапе у вас останутся только задачи с предварительными тре- бованиями, это будет означать, что они имеют циклическую зависимость и что ча- стичное упорядочивание нельзя расширить до полного.

Общий алгоритм показан на примере следующего псевдокода.

// Возвращаем узлы полностью упорядоченными. List(Of Node) ExtendPartialOrdering()

// Создаем список полностью упорядоченных узлов. List(Of Node): ordering

While <the network contains nodes>

// Находим узел без предварительных требований.

Node: ready\_node

ready\_node = <a node with no prerequisites> If <ready\_node == null> Then Return null

// Перемещаем узел в итоговый список.

<Add ready\_node to the ordering list>

<Remove ready\_node from the network> End While

Return ordering

End ExtendPartialOrdering

Основная идея этого алгоритма довольно проста. Сложность заключается в том, чтобы реализовать его эффективно. Если на каждом шаге вы будете просто искать в сети задачи без предварительных требований, количество таких шагов всякий раз может достигать *O*(*N*), что выльется в общее время работы продолжительно- стью *O*(*N*2).

Чтобы улучшить это решение, для каждого узла сети можно создать и инициа- лизировать новое свойство NumBeforeMe, которое будет хранить количество пред- варительных требований для узла. После этого при удалении узла из сети нужно проследовать по его звеньям, найти узлы, которые зависят от него, и уменьшить их значения NumBeforeMe на 1. Узел, чье свойство NumBeforeMe равно 0, готов стать частью расширенного упорядочивания.

Улучшенный алгоритм показан на примере следующего псевдокода.

// Возвращаем узлы полностью упорядоченными. List(Of Node) ExtendPartialOrdering()

// Создаем список полностью упорядоченных узлов. List(Of Node): ordering

// Создаем список узлов без предварительных условий. List(Of Node): ready

// Инициализация.

<Initialize each node’s NumBeforeMe count>

<Add nodes with no prerequisites to the ready list>

While <the ready list contains nodes>

// Расширяем упорядочивание за счет узла. Node: ready\_node = <First node in ready list>

<Add ready\_node to the ordering list>

// Обновляем счетчики NumBeforeMe. For Each link In ready\_node.Links

// Обновляем счетчик NumBeforeMe этого узла. ink.Nodes[1].NumBeforeMe = link.Nodes[1].NumBeforeMe - 1

// Проверяем, готов ли узел к перемещению.

If (link.Nodes[1].NumBeforeMe == 0) Then ready.Add(link.Nodes[1])

End If Next link

End While

If (<Any node has NumBeforeMe> 0>) Then Return null Return ordering

End ExtendPartialOrdering

В приведенном алгоритме мы исходим из того, что сеть является полностью связ- ной. Если это не так, алгоритм нужно применить к каждому связному компоненту.

## Поиск циклов

Процесс поиска циклов заключается в определении их наличия в составе сети. Другими словами, это определение того, возвращается ли путь, проходящий через сеть, в исходную точку.

Найти цикл довольно просто, если рассматривать этот процесс как разновид- ность топологической сортировки. Сеть содержит цикл лишь в том случае, если она не может быть отсортирована топологически. Проще говоря, с точки зрения топологической сортировки сеть содержит цикл, если последовательности задач *A*, *B*, *C*... *K* формируют петлю зависимостей.

Имея это в виду, вы сможете легко находить циклы. Следующий псевдокод де- монстрирует алгоритм.

// Возвращаем True, если сеть содержит цикл. Boolean: ContainsCycle()

// Пытаемся отсортировать сеть топологически.

If (ExtendPartialOrdering() == null) Then Return True Return False

End ContainsCycle

В рассматриваемом алгоритме мы исходим из того, что сеть является полностью связной. Если это не так, алгоритм нужно применить к каждому связному компоненту.

## Раскрашивание карты

Цель подобного алгоритма — раскрашивание карты таким образом, чтобы на ней не было двух смежных участков одинакового цвета. Конечно, это можно сде- лать, используя разные цвета для каждого участка. Вопрос состоит в том, с помо- щью какого минимального количества цветов реально раскрасить конкретную кар- ту. И еще: какое минимальное количество цветов может потребоваться для раскра- ски *любой* карты?

Чтобы научиться раскрашивать карту с помощью сетевых алгоритмов, вам нуж- но привести проблему закрашивания ее участков к задаче раскраски узлов. Про- сто создайте для каждого участка по одному узлу и соедините узлы, представляю- щие смежные участки, ненаправленными звеньями.

В зависимости от карты, вам может потребоваться два, три или четыре цвета. Подобные типы карт и алгоритмы для их закрашивания описаны в следующих разделах.

#### Закрашивание двумя цветами

Некоторые карты, как та, что показана на рисунке 14.2, можно закрасить все- го двумя цветами.

**Рис. 14.2.** Карта, закрашенная всего двумя цветами

**ЗАМЕЧАНИЕ**

 Создать двухцветную карту довольно просто. Поместите карандаш на лист бу- маги и нарисуйте фигуру, которая возвращается в вашу исходную точку. Объ-

ект может иметь любую форму, нужно только избегать «наложения» линий одну на другую. Иными словами, линия может пересекать себя в любой точке, но не совмещаться с частью самой себя на отрезке какой-либо длины. Такая фигу- ра показана на рисунке 14.3.

**Рис. 14.3.** Если нарисовать замкнутую кривую, не отрывая карандаш от бумаги и не делая «двойных линий», результат можно будет закрасить двумя цветами

Неважно, как именно пересекает себя линия, полученный объект всегда мож- но будет закрасить двумя цветами. Если после этого поверх первой фигуры нарисовать еще одну, следуя тем же правилам, результат будет обладать ана- логичными свойствами.

В раскрашивании такого рода карт нет ничего сложного. Выберите любой уча- сток и назначьте ему один из двух цветов. Затем задайте всем его соседям другой цвет и рекурсивно обойдите их, раскрашивая смежные участки. Если вы столкне- тесь с тем, что соседний узел уже закрашен тем же цветом, каким заполнен и теку- щий, это будет означать следующее: карту нельзя сделать двухцветной.

Алгоритм показан на примере следующего псевдокода.

TwoColor()

// Создаем очередь узлов, которые уже закрашены. Queue(Of Node): colored

// Закрашиваем первый узел и добавляем его в список. Node: first\_node = <Any node>

first\_node.Color = color1 colored.Enqueue(first\_node)

// Обходим сеть, закрашивая узлы. While (colored contains nodes)

// Берем следующий узел из списка закрашенных. Node: node = colored.Dequeue()

// Вычисляем цвет соседних узлов. Color: neighbor\_color = color1

If (node.Color == color1) Then neighbor\_color = color2

// Закрашиваем соседние узлы. For Each link In node.Links

Node: neighbor = link.Nodes[1]

// Смотрим, не закрашены ли соседние узлы. If (neighbor.Color == node.Color) Then

<The map cannot be two-colored>

Else If (neighbor.Color == neighbor\_color) Then

Else

End If Next link

End While End TwoColor

// Соседний узел уже был правильно закрашен.

// Больше ничего не делаем.

// Соседний узел не был закрашен. Закрашиваем его.

neighbor.Color = neighbor\_color colored.Enqueue(neighbor)

В приведенном алгоритме мы исходим из того, что сеть является полностью связ- ной. Если это не так, алгоритм нужно применить к каждому связному компоненту.

#### Закрашивание тремя цветами

Как оказывается, определение того, является ли карта трехцветной, — крайне сложная задача. Фактически еще не изобретено такого алгоритма, который мог бы ее решить за полиномиальное время.

Одно из довольно очевидных решений заключается в применении к узлам каждого из трех цветов и поиске подходящей комбинации. Если сеть содержит *N* узлов, время поиска будет равно *O*(3*N*); это довольно медленно, если *N* большое. Для проверки комбинаций можно прибегнуть к алгоритму перебора, воспользовавшись одним из деревьев принятия решений, описанных в главе 12. Но скорость все равно будет очень низкой.

Вы можете попробовать улучшить поиск, упростив сеть. Если узел имеет мень- ше трех соседей, эти соседи нередко используют не более двух цветов — таким образом, оставшийся цвет следует задействовать в исходном узле. В данном слу- чае вы можете убрать из сети узел, у которого меньше трех соседей, раскрасить уменьшенную сеть и затем вернуть его обратно, назначив ему цвет, не примененный смежными узлами.

Удаление узла из сети уменьшает количество соседей для оставшихся узлов, поэтому вы можете попробовать сделать сеть еще меньше. Если вам повезет, сеть будет уменьшаться, пока в ней не останется всего один узел. Выбрав для него цвет, вы сможете постепенно возвращать удаленные узлы, раскрашивая каждый из них.

Ниже представлено пошаговое описание алгоритма, который использует этот подход.

1. Повторяем, пока сеть содержит узлы со степенью меньше 3, — удаляем узел со степенью меньше 3, запоминая его местоположение, чтобы он мог быть позже восстановлен.
2. Используем алгоритм обхода, чтобы раскрасить оставшуюся часть сети тре- мя цветами. Если решение найти не удается, это означает, что его нет и для исходной сети.
3. Восстанавливаем ранее удаленные узлы, начиная с тех, что были исключе- ны последними, и закрашиваем их цветами, которые не используются их соседями.

Если сеть не является полностью связной, вы можете применить этот алгоритм к каждому из ее связных компонентов.

#### Закрашивание четырьмя цветами

Согласно *теореме о четырех красках*, любую карту можно закрасить четырь- мя цветами. Впервые эта теорема была предложена Фрэнсисом Гутри в 1852 г. Она активно изучалась на протяжении 124 лет, пока в 1974 г. ее, наконец, не доказали Кеннет Аппел и Вольфганг Хакен. К сожалению, их доказательство включает ис- черпывающее исследование набора из 1936 специально подобранных карт, поэто- му оно не предлагает надежного метода для раскрашивания любой карты четырь- мя цветами.

**ЗАМЕЧАНИЕ**

Теорема о четырех красках исходит из того, что сеть является *планарной*. Это означает, что ее можно нарисовать на плоскости без пересечения звеньев. Зве- нья должны иметь вид прямых линий; они могут отклоняться и перемещаться по всей поверхности, но только не пересекаться.

Если сеть не планарная, нет никакой гарантии того, что ее получится закрасить четырьмя цветами. Например, вы можете создать 10 узлов с 90 звеньями, со- единяющими каждую из пар. Поскольку все узлы связаны между собой, для закрашивания сети понадобится 10 цветов. Однако сеть, созданная из обыч- ной карты, является планарной.

Вы можете воспользоваться приемами, похожими на те, что применялись в предыдущем разделе для закрашивания тремя цветами.

1. Повторяем, пока в сети есть узлы со степенью меньше 4, — удаляем узел со степенью меньше 4, запоминая его местоположение, чтобы он мог быть позже восстановлен.
2. Используем алгоритм обхода, чтобы раскрасить оставшуюся часть сети че- тырьмя цветами. Если решение найти не удается, это означает, что его нет и для исходной сети.
3. Восстанавливаем ранее удаленные узлы, начиная с тех, что были исключе- ны последними, и закрашиваем их цветами, которые не используются их со- седями.

Опять же, если сеть не является полностью связной, вы можете применить этот алгоритм к каждому из ее связных компонентов.

#### Закрашивание пятью цветами

В отличие от предыдущего случая, для которого не придумано простого и кон- структивного алгоритма, для закрашивания пятью цветами существует одно реше- ние, хотя и не самое легкое.

Рассматриваемый алгоритм, как и те, что были описаны в предыдущих двух разделах, постепенно упрощает сеть. Его отличие заключается в том, что он всег- да может привести сеть к состоянию, когда она будет содержать всего один узел.

После этого вы можете обратить изменения, восстанавливая исходную сеть и за- крашивая узлы, как делали это ранее.

Приведенный алгоритм использует упрощения двух типов.

Первый тип похож на подход, который применялся в предыдущих двух раз- делах. Если сеть содержит узел, у которого меньше пяти соседей, он удаляется из сети. При восстановлении ему присваивается цвет, который не используется его соседями. Назовем это правилом 1.

Второй тип упрощения используется в случае, если в сети нет узлов с менее чем пятью соседями. Можно утверждать (хотя это выходит за рамки книги) сле- дующее: такая сеть должна иметь по крайней мере один узел *K* с соседями *M* и *N*, при условиях, что:

* + у *K* есть ровно пять соседей;
  + *M* и *N* имеют максимум семь соседей;
  + *M* и *N* не являются соседями друг друга.

Чтобы упростить сеть, найдите такие узлы *K*, *N* и *M*; при этом *N* и *M* должны иметь один и тот же цвет. Это допустимо, поскольку мы знаем, что они не явля- ются смежными. Узел *K* имеет ровно пять соседей, а *M* и *N* закрашены одним цве- том, поэтому соседи узла *K* не могут использовать все пять доступных цветов, ведь один из них нужно оставить для *K*.

Упрощение заключается в удалении из сети узлов *K*, *N* и *M* и создании ново- го узла *M*/*N* для представления цвета, которым будут закрашены *M* и *N*. Назначь- те новому узлу тех же соседей, которые были у *M* и *N*. Назовем это правилом 2.

При восстановлении узлов *K*, *N* и *M*, которые были удалены с использованием правила 2, назначьте последним двум цвет, которым закрашивался узел *M*/*N*. За- тем выберите для узла *K* цвет, который не занят ни одним из его соседей.

Вы можете воспользоваться приемами, похожими на те, что применялись в предыдущем разделе для закрашивания тремя цветами.

1. Повторяем, пока в сети есть больше одного узла:
   * удаляем узел со степенью меньше 5, если такой имеется, и запоминаем его местоположение, чтобы он мог быть восстановлен позже;
   * если сеть не содержит ни одного узла со степенью меньше 5, находим узел *K* со степенью 5 и два его дочерних элемента *M* и *N*, как было описано ра- нее. Удаляем узлы *K*, *N* и *M*, после чего создаем новый узел *M*/*N* (см. пра- вило 1).
2. Когда в сети останется один узел, назначаем ему цвет.
3. Восстанавливаем ранее удаленные узлы, закрашивая их соответствующим образом.

Если сеть не является полностью связной, вы можете применить этот алгоритм к каждому из ее связных компонентов.

На рисунке 14.4 показан процесс упрощения небольшой демонстрационной сети. Если взглянуть на верхнюю его часть, можно заметить, что каждый узел име- ет пять соседей, поэтому для упрощения сети нельзя использовать правило 1.

Несмотря на это, в нашем распоряжении все еще есть правило 2. Существу- ет несколько узлов, которые потенциально могут играть роль *K*, *N* и *M*. В рассма- триваемом примере используются узлы *C*, *B* и *H*. Они удаляются из сети, а их ме- сто занимает новый узел, *B*/*H*, который имеет такие же дочерние элементы, как *B* и *H* до него.

После замены узлов *C*, *B* и *H* на *B*/*H* оказывается, что узлы *G*, *A* и *D* имеют мень- ше пяти соседей, поэтому их следует удалить (в данном примере мы предполага- ем, что они были исключены именно в таком порядке).



*L*

*G*

*H*

*C*

*B*

*D*

*A*

*K*

*E*

*I*

*F*

*J*

Правило 2

*L*

*G*

*B/H*

*A*

*D*

*K*

*I*

*F E*

*J*

Правило 1

*L*

*B/H*

*K*

*I*

*F E*

*J*

Правило 1



*J*

**Рис. 14.4.** Эту сеть можно упростить до единственного узла, применив единожды правило 2 и несколько раз правило 1

После удаления оказывается, что узлы *L*, *B*/*H*, *K*, *F*, *E* и *I* имеют меньше пяти соседей, поэтому они тоже удаляются.

На данном этапе в сети остается всего один узел *J*, поэтому алгоритм произволь- ным образом присваивает ему цвет и приступает к восстановлению сети.

Допустим, алгоритм назначает узлам цвета в таком порядке: красный, зеле- ный, синий, желтый и оранжевый. Если соседние узлы закрашены красным, зе- леным и оранжевым цветами, тогда сам узел получит первый неиспользованный цвет — в нашем случае синий.

Начиная с итоговой сети, показанной на рисунке 14.4, алгоритм выполняет следующие шаги.

1. Алгоритм закрашивает узел *J* красным цветом.
2. Алгоритм восстанавливает узел, который был удален последним (*I*). Сосед узла *I* — *J* — имеет красный цвет, поэтому алгоритм делает узел *I* зеленым.
3. Алгоритм восстанавливает узел, который был удален предпоследним (*E*). Соседи узла *E*, — *J* и *I*, — закрашены красным и зеленым, поэтому алгоритм делает узел *E* синим.
4. Алгоритм восстанавливает узел *F*. Его соседи, *J* и *E*, закрашены красным и си- ним, поэтому алгоритм делает узел *F* зеленым.
5. Алгоритм восстанавливает узел *K*. Его соседи, *J* и *F*, закрашены красным и зеленым, поэтому алгоритм делает узел *K* синим.
6. Алгоритм восстанавливает узел *B*/*H*. Его соседи, *K*, *F* и *I*, закрашены синим, зеленым и зеленым, поэтому алгоритм делает узел *B*/*H* красным.
7. Алгоритм восстанавливает узел *L*. Его соседи, *K*, *B*/*H* и *I*, закрашены синим, красным и зеленым, поэтому алгоритм делает узел *L* желтым (на данном эта- пе сеть выглядит подобно нижнему варианту на рисунке 14.4, только с за- крашенными узлами).
8. Алгоритм восстанавливает узел *D*. Его соседи, *B*/*H*, *E* и *I*, закрашены крас- ным, синим и зеленым, поэтому алгоритм делает узел *D* желтым.
9. Алгоритм восстанавливает узел *A*. Его соседи, *B*/*H*, *F*, *E* и *D*, закрашены крас- ным, зеленым, синим и желтым, поэтому алгоритм делает узел *A* оранжевым.
10. Алгоритм восстанавливает узел *G*. Его соседи, *L*, *B*/*H* и *K*, закрашены жел- тым, красным и синим, поэтому алгоритм делает узел *G* зеленым (на дан- ном этапе сеть выглядит подобно среднему варианту на рисунке 14.4, толь- ко с закрашенными узлами).
11. Теперь алгоритм отменяет действие правила 2. Он восстанавливает узлы *B* и *H*, закрашивая их тем же цветом, который был у *B*/*H* (то есть красным). В завершение он восстанавливает узел *C*; его соседи, *G*, *H*, *D*, *A* и *B*, закраше- ны зеленым, красным, желтым, оранжевым и красным, поэтому алгоритм де- лает его синим (на данном этапе сеть выглядит подобно верхнему варианту на рисунке 14.4, только с закрашенными узлами).

На рисунке 14.5 показана исходная сеть. Узлы разных цветов представлены различными фигурами.

Красный Зеленый



Синий  Желтый

Оранжевый

**Рис. 14.5.** Фигуры представляют узлы разных оттенков в этой пятицветной сети

#### Другие алгоритмы закрашивания карт

Рассмотренные выше алгоритмы не единственные возможные для закраски карт. Например, стратегия восхождения на вершину может циклически переби- рать узлы сети, выдавая каждому из них первый цвет, который не используется ни одним из его соседей. Этот алгоритм не всегда позволяет закрасить сеть мини- мальным количеством цветов, однако при этом является чрезвычайно простым и быстрым. Он работает даже для непланарных сетей (например, как в случае с се- тью, на рисунке 14.6) и в некоторых ситуациях дает возможность использовать для закрашивания четыре цвета.



*F*

*A*

*G*

*J*

*E*

*B*

*C*

*D*

*H I*

**Рис. 14.6.** Эта сеть не является планарной, но ее можно закрасить тремя цветами

Приложив определенные усилия, вы можете применить другие эвристические методики, описанные в главе 12, и попытаться найти с их помощью минимальное количество цветов, необходимых для закрашивания конкретной планарной или непланарной сети. Например, можно попробовать стратегии случайного присва- ивания или инкрементального улучшения, чтобы поменять местами цвета двух или более узлов.

## Максимальный поток

В *нагруженной сети* каждое звено имеет максимальную пропускную способ- ность, указывающую на то, поток какого объема способен через него пройти. Это может быть количество галлонов воды, которую может пропустить через себя труба за минуту, количество автомобилей, способных проехать по улице за час, или максимальный ток, который выдерживает провод.

При решении данной задачи цель заключается в присвоении звеньям потоков таким образом, чтобы максимизировать общий поток из исходного узла в конечный.

Рассмотрим, например, сети, показанные на рисунке 14.7. Цифры на звеньях обозначают их поток и пропускную способность. Так, звено между узлами *B* и *C* в первой сети имеет поток объемом 1 и пропускную способность 2.

3/3 1/2 3/3 2/2



*A*



*B*



*C*



*A*



*B*



*C*

1/2

2/2 1/3

2/2

1/2 2/3

1/2 3/3 2/2 3/3



*F*



*E*



*D*



*F*



*E*



*D*

**Рис. 14.7.** Цель задачи максимального потока заключается в максимизации потока из исходного узла в конечный

В сети, размещенной слева, общий поток от узла *A* к узлу *F* равен 4. Общий объем потока, исходящего из узла *A*, в этой сети составляет 1 единицу в направле- нии *A* ➔ *D* плюс 3 единицы в направлении *A* ➔ *B*, что в сумме дает 4. Аналогичная ситуация со входящим потоком, который ведет к конечному узлу *F*/3 единицы по звену *E* ➔ *F* плюс 1 единица по звену *C* ➔ *F* (если сеть не получает дополнитель- ные потоки и не теряет имеющиеся, общий поток, проникающий в конечный узел, будет равен общему потоку, исходящему из него).

Нельзя увеличить общий поток за счет увеличения потока отдельных звеньев. В этом примере уже используется максимальная пропускная способность звена *A* ➔ *B*, и ее нельзя больше расширить. То же самое касается звеньев *A* ➔ *D* и *E* ➔ *F* — дополнительный поток просто некуда было бы девать.

Однако вы можете улучшить данное решение, перенеся одну единицу потока из пути *B* ➔ *E* ➔ *F* в путь *B* ➔ *C* ➔ *F*. Это позволит звену *E* ➔ *F* задействовать не- используемую пропускную способность и добавить новую единицу потока к пути *A* ➔ *D* ➔ *E* ➔ *F*. Сеть, показанная в правой части рисунка 14.7, демонстрирует внесен- ные изменения, благодаря которым общий поток от узла *A* к узлу *F* увеличился до 5.

Алгоритм поиска максимального потока является довольно простым (по край- ней мере на высоком уровне). Однако принцип его работы не столь очевиден. По- мочь разобраться в нем может краткое знакомство с такими понятиями, как оста- точная пропускная способность, остаточная сеть и дополняющий путь.

Это может звучать устрашающе, но вам не стоит беспокоиться. Данные кон- цепции помогают понять алгоритм, однако для вычисления максимальных пото-

ков не нужно создавать множество новых сетей. Остаточная пропускная способ- ность, остаточные сети и дополняющие пути можно найти в рамках исходной сети, не прилагая особых усилий.

*Остаточная пропускная способность* звена — объем дополнительного потока, который можно добавить к этому звену. Например, звено *C* ➔ *F* в левой части ри- сунка 14.7 имеет остаточную пропускную способность в размере 2, поскольку все- го она может пропустить 3 единицы, а ее текущий поток равен 1.

У каждого звена есть виртуальное обратное звено, которое может не быть ча- стью сети. Например, для звена *A* ➔ *B* на рисунке 14.7 неявно определено обрат- ное звено *B* ➔ *A*. Обратные звенья имеют большое значение, поскольку они мо- гут обладать остаточной пропускной способностью и передавать через себя поток в противоположном направлении.

Остаточная пропускная способность обратного звена — это объем потока, про- ходящего в прямом направлении по соответствующему обычному звену. Напри- мер, звено *B* ➔ *E* в левой части рисунка 14.7 имеет поток в размере 2, поэтому оста- точная пропускная способность обратного звена *E* ➔ *B* тоже равна 2. Чтобы улуч- шить решение, показанное слева, алгоритм должен провести поток обратно по пути *E* ➔ *B*, освободив пропускной потенциал звена *E* ➔ *F*. Именно так в алгоритме ис- пользуется остаточная пропускная способность обратных звеньев.

*Остаточная сеть* — сеть, которая состоит из прямых и обратных звеньев, поме- ченных объемом их остаточной пропускной способности. Пример, который осно- ван на сети, показанной в левой части рисунка 14.7, представлен на рисунке 14.8. Обратные звенья обозначены пунктирными линиями.



0

1

*A B C*

3 1

1 1

2 0

1 2

1

0

*D E F*

1 3

**Рис. 14.8.** Остаточная сеть, демонстрирующая остаточную пропускную способность прямых и обратных звеньев сети

Например, звено *C* ➔ *F* в левой части рисунка 14.7 имеет пропускную способ- ность 3 и поток 1. Его остаточная пропускная способность равна 2, поскольку вы можете добавить к ней еще две единицы потока. Остаточная пропускная способ- ность его обратного звена равна 1, потому что из прямого звена можно убрать одну единицу потока. На рисунке 14.8 вы можете видеть, что звено *C* ➔ *F* помечено чис- лом 2 (это его остаточная пропускная способность). Обратное звено *F* ➔ *C* обозна- чено числом 1.

Чтобы улучшить это решение, нужно просто найти в рамках остаточной сети путь от начального звена к конечному, который проходит по прямым и обратным

звеньям с положительной остаточной пропускной способностью. Затем необходи- мо направить по сформированному пути дополнительный поток. В контексте ис- ходной сети данное действие будет эквивалентно добавлению потока к соответ- ствующему прямому звену (если звено остаточной сети обычное) или вычитанию из него этого потока (если звено обратное). Поскольку путь доходит до конечного узла, он увеличивает поток до него. Такой путь улучшает решение задачи, поэто- му он называется *дополняющим*.

Звенья, выделенные на рисунке 14.8 толстыми линиями, изображают допол- нительный путь в рамках остаточной сети для примера, представленного в левой части рисунке 14.7.

Чтобы определить, поток какого объема способен пропускать данный путь, про- следуйте по нему в обратном направлении от конечного к исходному узлу и най- дите прямое или обратное звено с наименьшей остаточной пропускной способно- стью. После этого обновите сетевые потоки, направив найденный объем по нашему пути. Преобразовав таким способом сеть, показанную в левой части рисунка 14.7, вы получите результат, изображенной в его правой части.

Это может показаться сложным, но после усвоения терминологии алгоритм не должен выглядеть таким уж запутанным. Приведенные ниже шаги описывают алгоритм Форда — Фалкерсона.

1. Повторяем, пока есть возможность находить дополняющий путь в рамках остаточной сети:
   1. находим дополняющий путь от исходного узла к конечному;
   2. проходим по дополняющему пути и определяем в нем наименьшую оста- точную пропускную способность;
   3. опять проходим по дополняющему пути и корректируем его, обновляя потоки для звеньев.

Помните, что вам не нужно создавать настоящую остаточную сеть. Вы може- те воспользоваться исходной сетью, сравнивая пропускную способность и потоки звеньев, чтобы вычислить их остаточную пропускную способность.

**ЗАМЕЧАНИЕ**

Вы можете добавить список обратных звеньев в каждый узел — так будет про- ще искать прямые звенья, которые к ним ведут (чтобы по ним можно было следовать в обратном направлении). В остальном структура сети не нужда- ется в изменениях.

Сетевые потоковые алгоритмы имеют несколько применений помимо, соб- ственно, вычисления потоков (входящих или текущих). Два из них описываются в следующих двух разделах: «Распределение рабочих мест» и «Поиск минималь- ных разрезов в потоке».

#### Распределение рабочих мест

Представьте, что в вашем распоряжении есть 100 служащих (каждый со сво- ей специализацией) и 100 рабочих мест, требующих определенного набора навы-

ков. *Задача о назначениях* заключается в распределении людей по должностям та- ким образом, чтобы количество выполненной ими работы было максимальным.

На первый взгляд это выглядит как сложная комбинаторная проблема. Вы мо- жете перепробовать все доступные варианты назначений, чтобы увидеть, какой из них дает лучший результат. Максимальное число комбинаций служащих состав- ляет 100! � 9,3  10157, поэтому для такого процесса потребуется некоторое время. Вы можете попытаться получить приблизительные результаты, воспользовавшись эвристическими методиками из главы 12, однако у этой проблемы есть лучшее ре- шение: нужно задействовать алгоритм максимального потока.

Создайте сеть должностных назначений, где для одного служащего и рабочего места отводится по узлу. Соедините сотрудника с каждым рабочим местом, кото- рое ему подходит. Создайте исходный узел, связанный со всеми служащими, а ра- бочие места «подключите» к конечному узлу. Назначьте всем звеньям пропуск- ную способность, равную 1.

На рисунке 14.9 показана сеть должностных назначений с пятью служащими (помечены буквами) и пятью рабочими местами (обозначены цифрами). Все зве- нья направлены вправо и наделены пропускной способностью 1. Стрелки и обо- значения пропускной способности были опущены, чтобы не усложнять рисунок.



*A*

1

*B*

2

*Сотрудники C*

3

*Рабочие места*

*D*

4

*E*

5

**Рис. 14.9.** Максимальный поток через сеть должностных назначений обеспечивает оптимальный результат

Теперь нужно найти максимальный поток от начального узла к конечному. Каждая единица потока проходит через служащего к рабочему месту, которое тот должен получить. Общий поток обозначает количество рабочих мест, что мож- но покрыть.

**ЗАМЕЧАНИЕ**

В *двудольной сети* узлы можно разделить на две группы, *A* и *B*, таким образом, чтобы каждое звено соединяло узел из группы *A* с узлом из группы *B*. Если удалить исходный и конечный узлы из сети, показанной на рисунке 14.9, по- лучится двудольная сеть.

*Двудольное паросочетание* — процесс сопоставления узлов из группы *A* с узла- ми из группы *B*. Методика, описанная в этом разделе, обеспечивает неплохое решение проблемы двудольных паросочетаний.

#### Минимальный разрез в потоке

Цель, которую преследует задача минимального разреза в потоке, заключается в отделении исходного узла от конечного путем исключения звеньев из сети; при этом пропускная способность удаляемых звеньев минимизируется.

Возьмем, например, сеть, показанную на рисунке 14.10. Попробуем найти наи- более подходящие для удаления звенья, чтобы отделить начальный узел *A* от ко- нечного — *O*. Мы могли бы удалить звенья *A* ➔ *B* и *A* ➔ *E* с общей пропускной спо- собностью 9. Но у нас есть лучший вариант — звенья *K* ➔ *O*, *N* ➔ *O* и *P* ➔ *O*, кото- рые в сумме обеспечивают пропускную способность размером 6 (попробуйте най- ти собственное решение и посмотрите, насколько оно хорошее).

4 4 0



*A*



*B*



*D*



*C*

5 2 2 2

2 0 2



*E*



*F*



*H*



*G*

2 0 2 2

2 2 2



*I*



*J*



*L*



*K*

2 0 2 2

0 2 2



*M*



*N*



*P*



*O*

**Рис. 14.10.** Попробуйте найти в этой сети наиболее подходящие для удаления звенья, чтобы отделить узел *A* от узла *O*

Полное удаление всех допустимых комбинаций звеньев выльется в огромную работу даже в рамках небольшой сети. Каждое звено либо исключается, либо оста- ется на месте, поэтому в сети с *N* звеньями количество возможных вариантов бу- дет равно 2*N*. Относительно маленькая сеть, показанная на рисунке 14.10, содержит 24 звена, стало быть, нам придется рассмотреть 224 � 16,8 млн комбинаций. В сети со 100 звеньями, которая в ряде сценариев (например, при моделировании транс- портной сети) тоже считается довольно небольшой, у нас было бы 2100 � 1,3  1030 вариантов. Если ваш компьютер способен рассматривать 1 млн комбинаций в се- кунду, на их полную обработку ушло бы примерно 4,0  1016 лет. Несомненно, по- иск можно облегчить, воспользовавшись некоторыми эвристическими методика- ми, но это чрезвычайно сложная задача.

К счастью, алгоритм максимального потока предлагает куда более простое ре- шение. Его высокоуровневое описание представлено в виде следующих шагов.

1. Вычисляем максимальный поток между исходным и конечным узлами.
2. Посещаем все доступные узлы, начиная с конечного. Используем только прямые и обратные звенья, у которых остаточная пропускная способность больше 0.
3. Помещаем все посещенные на шаге 2 узлы во множество *A*, а все оставшие- ся — во множество *B*.
4. Удаляем звенья, которые ведут от узлов во множестве *A* к узлам из множе- ства *B*.

Есть, правда, один нюанс, касающийся рассматриваемого алгоритма: причины, по которым данный подход работает, могут показаться неочевидными.

Для начала рассмотрим максимальный набор потоков при условии, что об- щий максимальный поток равен *F*, как и разрез, создаваемый алгоритмом. Этот разрез должен разделить начальные и конечные узлы. Без него от исходной к ко- нечной точке вел бы путь, через который можно пропустить дополнительный по- ток. В этом случае соответствующий дополняющий путь проходил бы по остаточ- ной сети, что исключает корректную работу алгоритма максимального потока, ко- торый выполняется на шаге 1.

Обратите внимание, что любой разрез, останавливающий поток от начального узла к конечному, должен иметь общий поток *F* через все свои звенья. Движение по разрезу может быть прямым и обратным, но в итоге к конечному узлу должно доходить *F* единиц потока, чтобы общий его объем был равен *F*.

Это означает, что звенья в разрезе, созданном с помощью алгоритма, должны иметь общую пропускную способность не меньше *F*. Остается только понять, по- чему данный показатель равен всего лишь *F*, но не больше. *F* — это общий объем потока, проходящего через разрез, однако некоторые его составляющие двигаются в разных направлениях, увеличивая суммарный поток звеньев разреза.

Предположим, что все так и есть. Тогда звено *L* направлено обратно от узлов во множестве *B* к узлам из множества *A*, а другое звено передает поток уже от *A* к *B*. Этот поток, двигающийся в противоположных направлениях, нивелирует сам себя, вследствие чего итоговый результат равен 0.

Тем не менее если звено *L* существует, оно имеет положительный поток от мно- жества *B* к множеству *A*. В этом случае соответствующее обратное звено будет на- делено положительной остаточной пропускной способностью. Но на шаге 2 алго- ритм проходит по всем прямым и обратным звеньям с положительной остаточной пропускной способностью, чтобы создать множество *A*. Поскольку звено *L* попа- дает в их число и ведет к узлу из данного множества, алгоритм должен был пройти по нему и добавить в эту же группу узел, находящийся на другом его конце.

Из этого следует, что звено с положительным потоком, ведущее от *B* к *A*, не мо- жет существовать.

Общий поток через разрез равен *F*. При этом обратного потока через разрез к узлам во множестве *A* быть не может. Следовательно, поток через разрез должен иметь объем *F*, равно как и общая пропускная способность, нивелированная в ре- зультате разреза.

Я предупреждал, что это будет выглядеть запутанно. Формальное объяс- нение, которое используют теоретики в области графов, является еще более сложным.

Теперь, поработав над задачей из рисунка 14.10, можно ознакомиться с ее ре- шением. Оптимальный разрез должен удалить звенья *E* ➔ *I*, *F* ➔ *J*, *F* ➔ *G*, *C* ➔*G* и *C* ➔ *D*, которые имеют общую пропускную способность 4. То, как будет выгля- деть сеть после процесса удаления, показано на рисунке 14.11.

4 4



*A*



*B*



*D*



*C*

5 2 2

2 2



*E*



*F*



*H*



*G*

2 2

2 2 2



*I*



*J*



*L*



*K*

2 0 2 2

0 2 2



*M*



*N*



*P*



*O*

**Рис. 14.11.** Демонстрация решения задачи минимального разреза в потоке для сети, представленной на рисунке 14.10

## Резюме

Некоторые сетевые алгоритмы довольно точно моделируют ситуации из реаль- ной жизни. Например, алгоритм кратчайшего пути может помочь вам найти опти- мальный маршрут проезда. Другие сетевые алгоритмы имеют не столь очевидное применение. Алгоритм максимального потока, например, может не только вычис- лить, какой наибольший поток способна пропустить через себя сеть, но и помочь распределить рабочие места между служащими.

Алгоритм редакционного расстояния, о котором речь пойдет далее, тоже исполь- зует сеть не совсем очевидным образом. С ее помощью он решает, насколько одна строка отличается от другой. Например, он может определить, что слова «персик» и «праздник» имеют больше сходства, чем «маслина» и «рассол».

Этот и другие алгоритмы для анализа и изменения строк рассматриваются в следующей главе.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

1. Реализуйте алгоритм топологической сортировки в программе, которую вы написали, решая упражнения для главы 13.
2. В некоторых программах можно одновременно выполнять несколько за- дач. Например, в случае с ремонтом кухни электрик и водопроводчик могут делать свою работу в одно и то же время. Как бы вы видоизмени- ли алгоритм топологической сортировки, чтобы добиться подобного па- раллелизма?
3. Как бы вы расширили алгоритм, созданный вами для упражнения 2, чтобы вычислить предполагаемое время завершения всех задач (исходя из того, что вам заранее известна продолжительность выполнения каждой из них).
4. Алгоритм топологической сортировки, описанный в этой главе, основывает- ся на том факте, если задачи могут быть полностью упорядоченными, одна из них не должна иметь предварительных условий (то есть ее исходящая степень равна 0, если использовать сетевую терминологию). Можете ли вы сказать то же самое об узлах с нулевой входящей степенью? Влияет ли это на время работы алгоритма?
5. Расширьте программу, которую вы использовали в упражнении 1, чтобы за- красить узлы сети двумя цветами.
6. Когда для упрощения сети, показанной на рисунке 14.4, применялось пра- вило 2, в нашем примере использовались узлы *C*, *B* и *H*. Перечислите все пары узлов, которые вы могли бы задействовать при условии, что *C* яв- ляется промежуточной точкой. Прибегая к формулировкам правила 2, какие узлы реально использовать в качестве *M* и *N*, если узел *C* играет роль *K*? Сколько вариантов применения этих пар вы можете придумать для упрощения сети?

7\*. Расширьте программу из упражнения 5, чтобы в результате исчерпывающе- го поиска найти минимальное количество цветов, которыми можно закра- сить планарную сеть. Подсказка: для начала воспользуйтесь алгоритмом за- краски двумя цветами, чтобы определить, достаточно ли двух цветов для за- краски сети. Если это не сработает, вам нужно будет проделать то же самое для трех и четырех цветов.

1. Используйте программу из упражнения 7 для закрашивания четырьмя цве- тами сети, показанной на рисунке 14.5.
2. Расширьте программу из упражнения 5, реализовав эвристический алго- ритм восхождения на вершину, описанный в подразделе «Другие алгорит- мы закрашивания карт» раздела «Раскрашивание карты» в текущей главе. Сколько цветов понадобится для закрашивания сетей, показанных на ри- сунках 14.5 и 14.6?
3. Нарисуйте для сети, изображенной на рисунке 14.12 (с исходным узлом *A* и конечным *I*), остаточную сеть. Найдите дополнительный путь и обновите сеть для оптимизации потоков. Можете ли вы после этого выполнить даль- нейшие улучшения?

4/4 1/2



*B*



*C*



*A*

2/4

3/4

1/2

1/1 2/2



*E*



*F*



*D*

1/3

2/2

3/5

1/4 3/3



*H*



*G*



*I*

**Рис. 14.12.** Остаточная сеть, с помощью которой следует найти дополняющий путь для этой сети

11\*\*. Расширьте программу из упражнения 9 для поиска максимального потока между исходным и конечным узлами в нагруженной сети.

1. Используйте программу, созданную вами в упражнении 11, чтобы найти оптимальное распределение рабочих мест для сети, показанной на рисун- ке 14.9. Каково максимальное количество должностей, для которых можно найти работников?
2. Чтобы определить устойчивость сети, можно вычислить количество разных путей между двумя узлами. Каким образом вы бы задействовали алгоритм максимального потока для подсчета путей между двумя узлами, которые не имеют общих звеньев? Как можно определить количество путей, у которых нет не только общих звеньев, но и узлов?
3. Сколько цветов нужно для закрашивания двудольной сети? Сколько оттен- ков необходимо, чтобы закрасить сеть распределения рабочих мест?

15\*\*. Расширьте программу, созданную вами в упражнении 12, чтобы найти раз- рез минимального потока между исходным и конечным узлами в нагружен- ной сети.

1. Используйте программу, созданную вами в упражнении 15, чтобы найти разрез минимального потока в сети, которая показана на рисунке 14.12. Ка- кие звенья были удалены и какова общая пропускная способность разреза?

# Глава 15 СТРОКОВЫЕ АЛГОРИТМЫ

Строковые операции часто применяются в ряде программ, благодаря чему они хорошо изучены. Множество программных библиотек имеет хорошие инструмен- ты для работы со строками. Поскольку эти операции настолько важны, доступные вам утилиты, скорее всего, используют наиболее оптимальные алгоритмы, стало быть вашему коду вряд ли удастся их превзойти.

Например, алгоритм Бойера — Мура, описанный в текущей главе, позво- ляет определять первое вхождение подстроки в другой строке. Это настолько распространенная операция, что в большинстве языков программирования для нее предусмотрены специальные инструменты. В них, вероятно, используется какая-нибудь разновидность алгоритма Бойера — Мура, поэтому вряд ли ваша собственная реализация окажется намного лучше. На самом деле многие би- блиотеки написаны на ассемблере или на каком-нибудь другом очень низком уровне, поэтому они могут обеспечить лучшую производительность, даже если вы задействуете тот же алгоритм в своем коде.

Если ваша программная библиотека содержит инструменты для выполнения таких задач, используйте их. Алгоритмы, рассматриваемые в этой главе, были вы- браны, потому что они представляют определенный интерес и являются важной частью базового алгоритмического образования. Кроме того, они служат примером практичных методик, которые можно применять и в других целях.

## Парные скобки

Некоторые строковые значения, такие как арифметические выражения, ино- гда содержат вложенные скобки. Чтобы обеспечить корректную вложенность, вы можете поместить одни парные скобки внутрь других; поместить внутрь пары все- го одну скобку нельзя. Например, ()(()(())) имеет корректную вложенность, а (() и (())) — нет.

Если обратиться к графическому представлению, то скобки выражения вло- жены корректно, если вы можете соединить линиями каждую скобку, которая находится слева, с каждой скобкой с противоположной стороны; при этом ли- нии не должны пересекаться и могут находиться либо сверху, либо снизу от вы- ражения. На рисунке 15.1 выражение ()(()(())) имеет корректную вложенность, а (() и (())) — нет.

### ( ) ( ( ) ( ( ) ) )





Корректно вложены

### ( ( )

...?

Некорректно вложены

### ( ( ) ) )



Некорректно вложены

?...

**Рис. 15.1.** Линии соединяют парные скобки

Алгоритмически довольно легко проверить, соответствуют ли скобки друг дру- гу, — достаточно использовать счетчик для отслеживания количества открытых скобок, у которых нет пары. Теперь следует инициализировать счетчик значени- ем 0 и сделать циклический перебор выражения. При обнаружении открывающей или закрывающей скобки счетчик, соответственно, увеличивается или уменьшает- ся на 1. Если счетчик в какой-либо момент становится отрицательным, это свиде- тельствует о некорректной вложенности. То же самое можно сказать, если по окон- чании проверки выражения счетчик не равен 0.

Алгоритм показан на примере следующего псевдокода.

Boolean: IsProperlyNested(String: expression) Integer: counter = 0

For Each ch In expression

If (ch == ‘(‘) Then counter = counter + 1 Else If (ch == ‘)’) Then

counter = counter - 1

If (counter < 0) Then Return False

End If

Next ch

If (counter == 0) Then Return True Else Return False

IsProperlyNested

Например, когда алгоритм сканирует выражение ()(()(())), значения счетчи- ка после считывания каждого символа выглядят так: 1, 0, 1, 2, 1, 2, 3, 2, 1, 0. Счет- чик ни разу не становится отрицательным и заканчивает работу на значении 0, по- этому выражение имеет корректную вложенность.

Некоторые выражения, помимо скобок, содержат какой-нибудь текст. Напри- мер, (8  3) + (20 / (7 – 3)) состоит из цифр, операторов, таких как  и +, и ско- бок. Чтобы узнать, вложены ли скобки корректно, вы можете проверять только их, игнорируя любые другие символы. Для этого подойдет предыдущий алгоритм IsProperlyNested.

#### Вычисление арифметических выражений

Арифметическое выражение, полностью заключенное в скобки, можно рекур- сивно определить как:

* + литеральное значение, такое как 4 или 1,75;
  + выражение *expr*, заключенное в скобки — (*expr*);
  + два выражения, разделенные оператором, как в *expr*1 + *expr*2 или *expr*1   *expr*2.

Например, в выражении 8  3 используется правило 3, где два операнда, 8 и 3 (они являются таковыми согласно правилу 1), разделены оператором .

Для создания рекурсивного алгоритма, который будет заниматься вычислени- ем арифметических выражений, можно воспользоваться рекурсивным определени- ем. Пошаговое высокоуровневое описание алгоритма представлено ниже.

1. Если выражение является литеральным значением, используем для его разбо- ра инструменты языка программирования, после чего возвращаем результат.
2. Если выражение имеет вид (*expr*), удаляем внешние скобки, рекурсивно вы- полняем алгоритм для вычисления *expr* и возвращаем результат.
3. Если выражение выглядит как *expr*1?*expr*2, где два операнда *expr*1 и *expr*2 разделены оператором ?, рекурсивно выполняем алгоритм для вычисления *expr*1 и *expr*2, объединяем полученные значения в соответствии с операто- ром ? и возвращаем результат.

В целом, это довольно очевидный подход. Наверное, самое сложное — опре- делить, с каким из трех случаев мы имеем дело, и если вам выпал третий вариант, разбить выражение на два операнда и оператор. Это можно реализовать с помо- щью счетчика, похожего на тот, что использовался в предыдущем разделе в алго- ритме IsProperlyNested.

Если счетчик равен 0 и вы находите оператор — это третий случай. Тогда опе- ранды расположены по обе стороны от оператора.

Если вы закончили сканировать выражение и не нашли оператор, а счетчик при этом равен 0 — вы имеете дело с одним из первых двух случаев. Если выраже- ние начинается открывающей скобкой — это второй случай, если нет — первый.

#### Синтаксические деревья

Алгоритм, описанный в предыдущем разделе, разбирает и затем вычисляет арифметические выражения. Но у вас может появиться необходимость в выполне- нии каких-то других действий с выражениями после их разбора. Представьте, что вам нужно многократно вычислить выражение с переменной (допустим, X), кото- рая принимает разные значения, — например, чтобы нарисовать график уравнения (X  X) – 7. Вы могли бы выполнить разбор и вычисления, раз за разом применяя предыдущий алгоритм и подставляя в него разные значения для X. К сожалению, разбор текста — относительно медленная операция.

Есть еще один вариант: разобрать выражение, но не вычислять его сразу. Та- ким образом вы можете вычислить его много раз, подставляя разные значения X, не выполняя одновременно разбор. Это можно сделать с помощью алгоритма, ко- торый очень похож на тот, что был описан в предыдущем разделе. Только вместо рекурсивных вызовов самого себя и объединения полученных результатов он ге- нерирует дерево объектов, которые представляют выражение.

Например, чтобы представить умножение, алгоритм создает узел с двумя до- черними элементами, которые обозначают операнды. Аналогичным образом пока- зывается операция сложения.

Для каждого типа узлов можно также создать отдельный класс. Классы долж- ны предоставлять метод Evaluate, который вычисляет и возвращает значение узла, вызывая при необходимости одноименный метод для своих дочерних узлов, если таковые имеются.

Построив подобное синтаксическое дерево, вы сможете циклически вызывать метод Evaluate из корневого узла для разных значений X.

На рисунке 15.2 показано синтаксическое дерево для выражения (X  X) – 7.



–



7

X

X

**Рис. 15.2.** Синтаксические деревья можно использовать для представления таких выражений, как (X  X) – 7

## Сопоставление с шаблоном

Алгоритмы, описанные в предыдущих разделах, являются полезными и эффек- тивными, но они привязаны к конкретной задаче, а именно к разбору и вычислению арифметических выражений. Данная операция очень распространена в програм- мировании, поэтому было бы полезно иметь более универсальный подход, с помо- щью которого можно разбирать и другие виды текстов.

Например, *регулярное выражение* — это строка, в виде которой программа может представлять шаблон для сопоставления в тексте. Программисты разработали не- сколько разных языков регулярных выражений. Чтобы не слишком усложнять этот раздел, мы будем использовать в нем язык, где применяются следующие символы.

* + Алфавитные символы, такие как *A* или *Q*, представляют соответствующие буквы.
  + Символ + представляет конкатенацию. Чтобы сделать выражения бо- лее простыми для чтения, такое обозначение часто опускается. Поэтому *ABC* — то же самое, что и *A* + *B* + *C*. Однако было бы удобно сделать его обязательным, чтобы программа смогла легко разбирать регулярные вы- ражения.
  + Символ \* означает, что предыдущее выражение может быть повторено сколь- ко угодно раз (в том числе и ноль).
  + Символ | означает, что текст должен соответствовать либо предыдущему, либо последующему выражению.
  + Скобки () определяют порядок выполнения задачи.

Например, в этом урезанном языке выражение *AB*\**A* соответствует строке, ко- торая начинается и заканчивается буквой *A* и содержит любое количество букв *B*. Под этот шаблон подошли бы строки *ABA*, *ABBBBA* и *AA*.

В более общем случае программа может искать первое вхождение шаблона в строку. Например, строка *AABBA* соответствует предыдущему шаблону *AB*\**A*, начиная со второй буквы.

Чтобы понять описанные здесь алгоритмы, предназначенные для сопоставле- ния регулярных выражений, сначала лучше познакомиться с детерминированны- ми и недетерминированными конечными автоматами (ДКА и НКА). Им посвяще- ны следующие два подраздела. В подразделе, который следует за ними, вы узнае- те, каким образом эти автоматы можно использовать для сопоставления с шабло- нами с помощью регулярных выражений.

#### Детерминированные конечные автоматы

*Детерминированный конечный автомат* (ДКА) — по сути, виртуальный ком- пьютер, который следит за выполняемыми им операциями с помощью набора ре- жимов (состояний). На каждом шаге он считывает некий ввод и в зависимости от этого ввода и своего текущего состояния переходит к новому режиму. Одно состо- яние, из которого автомат принимается за работу, является начальным. Одно или несколько состояний могут быть помечены как принимающие.

Если автомат заканчивает свои вычисления в принимающем режиме, он при- нимает ввод. С точки зрения обработки регулярных выражений это означает, что введенный текст соответствует шаблону.

Некоторым моделям автоматов удобно принимать ввод при переключении в любой принимающий режим.

ДКА можно представить в виде *диаграммы перехода состояний*, которая, в сущ- ности, является сетью, где режимы обозначены окружностями, а переходы к но- вым состояниям — звеньями. Каждое звено помечено вводом, который заставляет автомат переключиться в новый режим. Если автомат сталкивается с вводом, для которого не предусмотрено соответствующего звена, он останавливается в непри- нимающем состоянии.

Например, на рисунке 15.3 показаны переходы состояний для ДКА, который распознает шаблон *AB*\**A*. ДКА начинает с состояния 0. Если вводится символ *A*,

он переходит к состоянию 1. Сталкиваясь с каким-либо другим символом, он оста- навливается в непринимающем режиме.



*A*

*B*

*A*

0

1

2

**Рис. 15.3.** Сеть, представляющая переходы состояний для ДКА, который распознает шаблон AB\*A

Затем, если в состоянии 1 считывается символ *B*, ДКА проникает в цикл и воз- вращается к прежнему режиму. Если считывается символ *A*, ДКА переходит к со- стоянию 2.

Состояние 2 помечено двойной окружностью; это говорит о том, что оно явля- ется принимающим. В зависимости от способа использования ДКА может вернуть успешное соответствие, просто перейдя в данное состояние. Иногда автомату в этом состоянии необходимо завершить считывание ввода; в таких случаях, если входя- щая строка содержит больше символов, соответствие не обнаруживается.

Рассмотрим другой пример. Взгляните на диаграмму перехода состояний, по- казанную на рисунке 15.4. Она представляет автомат, ищущий строку, которая со- стоит из символов *AB* или *BA*, повторяющихся произвольное количество раз.

*A*



1 *B*

2

*A*

0

*B*

*A*

3

4

*B*

**Рис. 15.4.** Сеть, представляющая переходы состояний для ДКА, который распознает шаблон (AB)\*|(BA)\*

Чтобы реализовать ДКА на программном уровне, можно создать объект, кото- рый представляет каждое состояние на диаграмме перехода. При получении вхо- дящих данных текущий объект сменяется таким, который годится для подобно- го ввода.

ДКА часто реализуют в виде таблицы, демонстрирующей переходы состояний. Например, в таблице 15.1 показаны переходы состояний для диаграммы, представ- ленной на рисунке 15.3.

**Òàáëèöà 15.1.** Ïåðåõîäû ñîñòîÿíèé äëÿ AB\*A

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Состояние** | **0** | **1** | **1** | **2** |
| **При вводе** | *A* | *A* | *B* | — |
| **Новое состояние** | 1 | 2 | 1 | — |
| **Принимающее?** | Нет | Нет | Нет | Да |

**ЗАМЕЧАНИЕ**

ДКА используются не только для обработки регулярных выражений. С их по- мощью можно моделировать состояния любой системы, которую удобно опи- сывать в виде правил и диаграммы/таблицы переходов.

Например, система обработки заказов позволяет следить за их состоянием. Каждому состоянию можно назначить интуитивно понятное название: Разме- щен, Выполнен, Отправлен, Счет-фактура, Отменен, Оплачен, Возвращен. Состояние заказа будет изменяться по мере возникновения событий. Напри- мер, если заказ пребывает в состоянии «Размещен» и клиент решает его от- менить, этот заказ переходит в состояние «Отменен» и прекращает свое про- движение по системе.

#### Построение ДКА для регулярных выражений

Простые регулярные выражения можно довольно легко и интуитивно переве- сти в диаграммы и таблицы состояний. Но в случае со сложными регулярными вы- ражениями лучше вооружиться методическим подходом, который позволит деле- гировать программе выполнение этой работы.

Чтобы преобразовать регулярное выражение в таблицу перехода состояний ДКА, можно построить синтаксическое дерево, которое будет рекурсивно генери- ровать соответствующие переходы.

Листья синтаксического дерева обозначают входящие литеральные символы, такие как *A* и *B*. Диаграмма перехода состояний для считывания одиночных вхо- дящих символов представляет собой обычное начальное состояние, соединенное с принимающим конечным с помощью звена, помеченного необходимым симво- лом. На рисунке 15.5 изображена простая диаграмма перехода состояний для счи- тывания входящего символа *B*.

Внутренние узлы синтаксического дерева представляют операторы +, \* и |.

Чтобы реализовать оператор +, возьмем принимающее состояние диаграммы перехода левого поддерева и совместим его с начальным состоянием диаграммы

перехода правого поддерева. Таким образом, автомат должен по очереди выпол- нить действия обоих поддеревьев. Например, слева на рисунке 15.6 показаны ди- аграммы перехода для простых литеральных шаблонов *A* и *B*, а справа — для объ- единенного шаблона *A* + *B*.



0

B

1

**Рис. 15.5.** Диаграмма перехода, представляющая простое регулярное выражение B



0

*A*

1



0

*B*

1



0

*A*

1

*B*

2

**Рис. 15.6.** Диаграмма перехода, показанная справа, представляет регулярное выражение A + B

Чтобы реализовать оператор \*, объединим одиночное принимающее состояние подвыражения с его начальным состоянием. На рисунке 15.7 показаны диаграммы перехода для шаблонов A + B (слева) и (A + B)\* (справа).



0

*A*

1

*B*

2



0

*A*

1

*B*

**Рис. 15.7.** Диаграмма перехода, показанная справа, представляет регулярное выражение (A + B)\*

Наконец, чтобы реализовать оператор |, свяжем начальное и конечное состо- яния диаграмм перехода левого и правого подвыражений. В левой части рисун- ка 15.8 представлены диаграммы перехода для шаблонов A + B и B + A; в правой его части можно видеть объединенный шаблон (A + B) | (B + A).



0

*A*

1

*B*

2



0

*B*

1

*A*

2



0

*A*

*B*

1

2

*B*

*A*

3

**Рис. 15.8.** Диаграмма перехода, показанная справа, представляет регулярное выражение (A + B) | (B + A)

Этот подход работает в нашем случае, но в некоторых ситуациях у него могут проявиться существенные недостатки. Что случится с оператором |, если оба под- выражения начинаются с одних и тех же входящих переходов? Возьмем, напри- мер, подвыражения A + A и A + B. Если слепо следовать ранее описанной ме- тодике, получится диаграмма перехода, показанная в левой части рисунка 15.9. Она содержит два звена, помеченные как *A*, которые покидают состояние 0. Если ДКА в начальном состоянии столкнется с вводом символа *A*, какое звено он дол- жен будет выбрать?

A B



0

A

A

1

2

A

B

3



0

A

1

2

**Рис. 15.9.** Эти диаграммы перехода представляют регулярное выражение (A + A) | (A + B)

Одно из решений, представленное в правой части рисунка 15.9, заключается в небольшом изменении структуры диаграммы таким образом, чтобы диаграммы двух подвыражений имели общее первое состояние (состояние 1). И хотя подоб- ный подход работает, он требует определенной сноровки, а это не так-то легко ре- ализовать в программе. Найти аналогичное решение для более сложных подвыра- жений было бы еще труднее — по крайней мере, на программном уровне.

Чтобы решить эту проблему, можно прибегнуть к НКА вместо ДКА.

#### Недетерминированные конечные автоматы

Конечный автомат называют *детерминированным*, потому что его поведение полностью определяется текущим состоянием и вводом, который он принимает. Если ДКА, находясь в состоянии 0 и используя диаграмму перехода из правой ча- сти рисунка 15.8, прочитает символ *B*, он однозначно перейдет в состояние 2.

*Недетерминированные конечные автоматы* (НКА) похожи на ДКА, однако для одного и того же ввода они могут иметь несколько звеньев, исходящих из те- кущего состояния (см. рис. 15.9). Если такая ситуация возникает во время обра- ботки, НКА сам способен выбрать путь, чтобы в конечном итоге попасть в прини- мающее состояние. Все выглядит так, будто НКА советуется с гадалкой, которая заранее знает, какой ввод будет принят позже, и может решить, по каким звеньям нужно перемещаться.

Естественно, в реальности компьютер не сумеет сам определить, по какому пути он должен проследовать, чтобы в итоге найти принимающее состояние. Зато он способен перебрать все возможные пути. Для этого программа будет хранить список состояний, в которых она может оказаться. Сталкиваясь с вводом, она об- новляет каждое из них и в некоторых случаях генерирует дополнительные состоя- ния. На это можно посмотреть иначе: представьте, что НКА находится во всех этих состояниях одновременно. Тогда, если любое из его текущих состояний является принимающим, весь НКА начинает принимать ввод.

Вы можете внести еще одно изменение в переходы НКА, чтобы немного упро- стить их реализацию. Операции, представленные на рисунках 15.6–15.9, требуют совмещения состояний из разных подвыражений — и это может быть неудобно.

В качестве альтернативного варианта стоит попробовать ввести новый тип ну- левых переходов, которые происходят безо всякого ввода. Сталкиваясь с таким пе- реходом, НКА немедленно его выполняет.

На рисунке 15.10 показано, как совместить автоматы перехода состояний для подвыражений, чтобы произвести более сложные выражения. Символ  обозна- чает нулевой переход, а прямоугольник — потенциально сложную сеть состояний, представляющую подвыражение.

В первой части рисунка 15.10 показан набор состояний, описывающих не- кое подвыражение. Это может быть простой единичный переход, который со- поставляет единичный ввод (как на рис. 15.5), или же сложное множество со- стояний и переходов. С точки зрения остальных состояний единственной важ- ной чертой этой конструкции является наличие одного входящего и одного ис- ходящего состояний.

Во второй части рисунка 15.10 показано, как совместить два автомата, *M*1 и *M*2, с помощью оператора +. Исходящее состояние *M*1 соединено со входящим состояни- ем *M*2 с использованием нулевого перехода. Это позволяет избежать их совмещения.





















*M*1 *M*1+*M*2 *M*1\* *M*1|*M*2

**Рис. 15.10.** Использование НКА и нулевых переходов упрощает объединение подвыражений

В третьей части рисунка 15.10 показано, как добавить оператор \* к *M*1. Исхо- дящее состояние этого автомата соединено с его входящим состоянием с помо- щью нулевого перехода. Оператор \* позволяет указывать выражение, за которым он следует, сколько угодно раз (или не указывать вообще), поэтому посредством

еще одного нулевого перехода НКА может оказаться в принимающем состоянии без сопоставления с содержимым *M*1.

Последняя часть рисунка 15.10 демонстрирует процесс совмещения автоматов *M*1 и *M*2 с помощью оператора |. Итоговый автомат использует новое входящее со- стояние, связанное нулевым переходом со входящими состояниями *M*1 и *M*2. Исхо-

дящие состояния *M*1 и *M*2 соединены нулевым переходом с итоговым исходящим

состоянием нового совмещенного автомата.

Чтобы подытожить все вышесказанное, можете создать анализатор регуляр- ных выражений, выполнив следующие шаги.

1. Создайте синтаксическое дерево для регулярного выражения.
2. С помощью синтаксического дерева рекурсивно сгенерируйте состояния для НКА, представляющие выражение.
3. Запустите НКА в состоянии 0 и посимвольно обработайте с его помощью входящую строку.

## Поиск строк

В предыдущем разделе объяснялось, как с помощью ДКА и НКА искать ша- блоны в строках. Описанные там методики являются довольно гибкими, но в то же время относительно медленными. Для поиска сложного шаблона НКА иногда приходится отслеживать огромное количество состояний, так как он отдельно про- веряет каждый символ во входящей строке.

Если вы хотите найти во фрагменте текста определенную подстроку, а не шаб- лон, у вас есть более быстрые варианты. Самое очевидное решение — это цикли- чески перебрать все символы в тексте и проверить наличие искомой подстроки

в каждой из позиций. Этот прямолинейный подход показан на примере следующего псевдокода.

// Возвращаем позицию подстроки в тексте. Integer: FindTarget(String: text, String: target)

For i = 0 To <last index of string>

// Проверяем, начинается ли подстрока с позиции i. Boolean: found\_it = True

For j = 0 To <last index of target>

If (string[i + j] != target[j]) Then found\_it = False

Next j

// Проверяем, нашли ли мы искомую подстроку. If (found\_it) Then Return i

Next i

// На этом этапе можно сказать, что подстрока не найдена. Return -1

End FindTarget

В этом алгоритме переменная i изменяется, пока не достигнет длины текста. Для каждого ее значения переменная j проходит по всей длине подстроки. Если текст содержит *N* символов, а подстрока — *M*, тогда общее время работы алгорит- ма равно *O*(*N*  *M*). Это уже проще по сравнению с использованием НКА, но все равно не очень эффективно.

Алгоритм Бойера — Мура имеет другой подход, который значительно упрощает поиск строк. Вместо того чтобы перебирать искомую подстроку с самого начала, он начинает проверять ее с конца, двигаясь в обратном направлении.

Чтобы вам было проще понять рассматриваемый алгоритм, представьте, что ис- комая подстрока расположена под текстом в позиции, в которой может быть най- дено совпадение. Сравнение символов начинается с левого края подстроки. Если в текущей позиции она не совпадает с текстом, алгоритм перемещает ее вправо к следующей позиции, где он может найти соответствие.

Представьте, например, что вы хотите найти строку Roosevelt в тексте

A man a plan a canal Panama. Рассмотрите рисунок 15.11.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R | o | o | s | e | v | e | l | t |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A |  | m | a | n |  | a |  | p | | l | a | n |  | a |  | c | a | n | | a | l |  | P | a | n | a | m | a | |
|  | | | | | | | | |  | | | | | | | | | |  | | | | | | | | | |  |
|  | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R | o | o | s | e | v | e | l | t |

**Рис. 15.11.** Поиск подстроки Roosevelt в тексте A man a plan a canal Panama

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R | o | o | s | e | v | e | l | t |

требует всего трех сравнений

Сначала алгоритм выравнивает обе строки по левому краю и сравнивает по- следний символ подстроки (t) с соответствующим символом в тексте (p). Они не совпадают, поэтому искомая подстрока сдвигается вправо, чтобы можно было най- ти очередную позицию, где возможно совпадение. Следующий символ, p, не содер-

жится в подстроке, поэтому алгоритм сдвигает подстроку вправо на девять симво- лов, чтобы полностью поменять текущее местоположение.

На новой позиции последний символ подстроки, t, не совпадает с соответству- ющим символом в тексте, n. Совпадение опять не найдено, и подстрока сдвигает- ся вправо на девять символов, потому как она не содержит n.

В новой позиции последний символ подстроки, t, не совпадает с соответству- ющим символом в тексте, a. Опять не найдено совпадения, и опять в искомой под- строке не встречается символ a, поэтому она снова сдвигается на девять позиций вправо.

На данном этапе подстрока выходит за пределы текста, поэтому совпадение не- возможно. Алгоритм делает заключение: подстрока не содержится в тексте. Чтобы узнать то же самое с помощью ранее описанного алгоритма перебора, пришлось бы выполнить 37 сравнений — алгоритму Бойера — Мура понадобилось всего три.

Но не всегда все проходит так гладко. В качестве более сложного примера по- пробуем найти подстроку cadabra в тексте abba daba abadabracadabra. Рас- смотрим рисунок 15.12.

a

r

b

a

d

a

c

a

r

b

a

d

a

b

a

a

b

a

d

a

b

b

a

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| c | a | d | a | b | r | a |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| c | a | d | a | b | r | a |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| c | a | d | a | b | r | a |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| c | a | d | a | b | r | a |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| c | a | d | a | b | r | a |

**Рис. 15.12.** Поиск cadabra в тексте abba daba abadabracadabra требует

18 сравнений

Алгоритм начинает свою работу с двух строк, выровненных по левому краю, и сравнивает символ подстроки a с символом в тексте — a. Они совпадают, поэто- му алгоритм проверяет следующие символы — r и d. Совпадения нет, в связи с чем искомая подстрока сдвигается вправо. Однако в данном случае символ d встреча- ется в подстроке, поэтому он может оказаться частью совпадения. Подстрока сдви- гается до тех пор, пока ее последний символ d (закрашен серым цветом на рисун- ке 15.12) не поравняется с аналогичным символом в тексте.

На новой позиции последний символ подстроки a не совпадает с соответству- ющим символом в тексте, — пробелом. Подстрока не содержит пробелов, поэтому она сдвигается вправо на всю свою длину (семь символов).

На новой позиции последний символ подстроки a не совпадает с соответству- ющим символом в тексте r. В подстроке встречается символ r, поэтому она сдви- гается вправо, пока ее r (закрашенный серым цветом) не поравняется с r в тексте.

На очередной позиции последний символ подстроки a совпадает с соответ- ствующим символом в тексте a, поэтому алгоритм проверяет предыдущие симво- лы. Поскольку они тоже совпадают, алгоритм продолжает двигаться в обратном направлении, сравнивая содержимое подстроки с текстом. Совпало шесть симво- лов. Однако на первом символе подстроки c обнаружилось несовпадение с соот- ветствующим символом в тексте, b.

В искомой подстроке встречается символ b, но не на той позиции, в которой алгоритм находится в данный момент. Чтобы совместить его с подходящим сим- волом в тексте, алгоритму пришлось бы сдвигать подстроку влево. Однако этого не произойдет, поскольку все позиции слева уже были исключены. Алгоритм дви- гается в противоположном направлении, сдвигая подстроку вправо на семь сим- волов — туда, где еще возможно совпадение. На этой позиции все символы под- строки совпадают с аналогичными символами в тексте. Следовательно, алгоритм нашел соответствие.

Пошаговое высокоуровневое описание алгоритма Бойера — Мура представлено ниже.

1. Выравниваем искомую подстроку и текст по левому краю.
2. Повторяем, пока последний символ подстроки не выйдет за пределы текста:
   1. начиная с конца подстроки, сравниваем ее символы с соответствующими символами в тексте. Двигаемся в обратном направлении к началу;
   2. если все символы совпадают — соответствие найдено;
   3. допустим, символ x в тексте не совпадает с соответствующим символом в подстроке. Сдвигаем подстроку вправо, пока x не поравняется с таким же символом в той части подстроки, которая находится слева от текущей пози- ции. Если такого символа нет, сдвигаем подстроку вправо на всю ее длину.

Одним из наиболее трудоемких участков этого алгоритма является шаг 2c, ко- торый вычисляет, насколько далеко нужно сдвинуть подстроку вправо. Его мож- но ускорить, если заранее подсчитать количество несовпадающих символов в раз- ных позициях подстроки.

Представьте, например, что в позиции 3, где текст содержит символ *G*, обнару- живается несовпадение. Это заставит алгоритм сдвинуть текст вправо, чтобы его символ *G* поравнялся с первым символом *G* в подстроке, который находится сле- ва от позиции 3.

**ЗАМЕЧАНИЕ**

В некоторых разновидностях алгоритма Бойера — Мура используются более сложные правила для эффективного сдвига искомой подстроки. Представьте, к примеру, что алгоритм находится в следующем положении.

... what shall we draw today ... abracadabra

Он сканирует abracadabra в обратном направлении. Первые два символа, a и r, совпадают. Затем символ d в тексте не совпадает с символом b в под- строке. Предыдущая версия алгоритма сдвинула бы подстроку так, чтобы несовпавший символ текста d находился в следующей позиции:

... what shall we draw today ...

abracadabra

Но мы уже видели, что текст совпадает со следующими двумя символами, ra, поэтому знаем: в этом месте символы в тексте dra соответствуют символам в подстроке dab.

Вместо того чтобы делать сдвиг на позицию несовпавшего символа d, находя- щегося в тексте, мы можем выровнять совпавший суффикс (ra в нашем слу- чае) с более ранним вхождением этих символов в искомой подстроке. Проще говоря, подстрока сдвигается так, чтобы поместить предыдущее вхождение ra напротив текущего суффикса.

... what shall we draw today ...

abracadabra

Это позволяет алгоритму сдвинуть подстроку дальше, что ускорит выполнение поиска.

Больше о разновидностях алгоритма Бойера — Мура можно узнать по адресу

https://ru.wikipedia.org/wiki/Àëãîðèòì\_Áîéåðà\_—\_Ìóðà.

Алгоритм Бойера — Мура имеет необычную особенность: чем длиннее искомая подстрока, тем быстрее он работает. Это связано с тем, что при нахождении несо- впадающего символа он может сдвинуть подстроку на большее расстояние.

## Вычисление редакционного расстояния

*Редакционное расстояние* между двумя строками — это минимальное количе- ство изменений, необходимых для превращения одной строки в другую. Допусти- мые изменения можно определить несколькими способами. Мы будем исходить из того, что нам позволено только удалять и вставлять буквы. Есть еще один вид изменений, который здесь не рассматривается, — замена одной буквы на другую. Тот же результат можно получить путем удаления первой буквы и вставки второй.

Рассмотрим, например, слова *encourage* и *entourage*. Очевидно, что для превра- щения первого во второе достаточно удалить c и вставить t. Это два изменения, по- этому редакционное расстояние между данными словами равно 2.

Давайте возьмем еще два слова: *assent* и *descent*. Один из способов преобразо- вания первого во второе будет состоять из следующих шагов.

1. Удаляем *a*, чтобы получить *ssent*.
2. Убираем *s*, чтобы образовать *sent*.
3. Удаляем *s*, чтобы получить *ent*.
4. Добавляем *d*, чтобы сформировать *dent*.
5. Добавляем *e*, чтобы получить *deent*.
6. Вставляем *s*, чтобы образовать *desent*.
7. Добавляем *c*, чтобы получить *descent*.

Всего потребовалось семь шагов, поэтому редакционное расстояние не больше

1. Но как определить, является ли данный способ преобразования *assent* в *descent* самым эффективным? В случае с более длинными словами или строками (либо файлами, как вы вскоре увидите в этом разделе) проверить, что мы нашли опти- мальное решение, может оказаться довольно сложно.

Один из способов вычисления редакционного расстояния заключается в по- строении редакционного графа, представляющего все изменения, которые можно внести, чтобы превратить одно слово в другое. Начнем с создания массива узлов, похожего на тот, что изображен на рисунке 15.13.

Узлы в верхней части графа представляют буквы первого слова. Буквы второго слова находятся слева. Создадим звенья между узлами, ведущие вправо и вниз.

Добавим диагональные звенья, отсылающие к тем узлам, где совпадают соот- ветствующие буквы в обоих словах. Например, в слове *assent* буква *e* является чет- вертой, а в слове *descent* — второй, поэтому диагональное звено ведет к узлу снизу от *e* в *assent* и справа от первого *e* в *descent*.

Каждое звено представляет собой преобразование первого слова, которое дела- ет его более похожим на второе. Звено, ведущее вправо, обозначает удаление буквы из первого слова. Например, звено, отсылающее к букве a в верхней строке, описы- вает удаление *a* из *assent*, результатом которого будет *ssent*.

Звено, указывающее вниз, обозначает добавление буквы в слово. Например, звено, ведущее к букве *d* в первом столбце, иллюстрирует вставку этой буквы, в ре- зультате чего получится слово *dassent*.

Диагональное звено означает, что буква остается без изменений.

Любой путь в этом графе, идущий из левого верхнего в правый нижний угол, соответствует последовательности изменений для преобразования первого слова во второе. Например, изменения, описанные выше, представлены на рисунке 15.13 в виде звеньев, которые выделены толстыми линиями.

Теперь найти в редакционном графе наименее затратный путь будет достаточ- но легко. Присвоим горизонтальным и вертикальным звеньям затраты в размере 1, а дополнительным — 0. Теперь нам нужно просто вычислить самый короткий путь в сети.

Для этого можно использовать методики, которые были описаны в главе 13, но эта сеть обладает особой структурой, которая позволяет прибегнуть к более про- стому способу.

Для начала установим для узлов расстояния, равные номерам их столбцов. Расстояние от левого верхнего угла до узла в пятом столбце равно 5 — то есть, количеству звеньев, которое нужно до него пройти.

Аналогично расстояния до узлов в крайнем левом столбце равны номерам их строк. Чтобы получить узел в строке 7, необходимо пройти по семи звеньям.



a

s

s

e

n

t

d e s c e n

t

**Рис. 15.13.** Редакционный граф, описывающий возможные пути превращения *assent*

в *descent*

Теперь циклически переберем все строки и для каждой из них выполним пе- ребор соответствующих столбцов. Самый короткий путь к узлу на позиции (*r*, *c*) проходит через узел, размещенный сверху (*r* – 1, *c*) или слева (*r*, *c* – 1) от него; если же допускается движение по диагонали, промежуточный узел может находиться сверху слева, (*r* – 1, *c* – 1). Расстояния ко всем этим узлам уже были установлены, поэтому вы можете определить затраты всех возможных путей и определить узел, который находится на самом коротком расстоянии (*r*, *c*).

Выяснив, как искать редакционное расстояние между двумя словами или стро- ками, сделать то же самое для двух файлов. Алгоритм можно оставить без измене- ний, сравнивая файлы посимвольно. К сожалению, для этого потребуется слиш- ком большой редакционный граф. Например, если файлы содержат примерно по 40 000 символов (допустим, как эта глава), редакционный граф будет состоять из 40 000  40 000 = 1,6 млрд узлов. Его построение потребует большого объема па- мяти, а на его использование уйдет много времени.

В качестве альтернативного подхода алгоритм можно изменить таким образом, чтобы вместо символов он сравнивал строчки в файлах. Для файлов, состоящих из 700 строчек, редакционный граф будет иметь 700  700 = 49 000 узлов. Это все равно довольно много, но уже более приемлемо.

## Резюме

Многим программам приходится заниматься анализом и изменением строк. И хотя программные библиотеки содержат самые разные инструменты для этих задач, вам стоит понимать, как именно работают некоторые из таких алгоритмов.

Например, использовать готовое регулярное выражение намного проще, чем пи- сать свое собственное, однако методики применения ДКА и НКА для обработки команд могут пригодиться во многих других ситуациях. Известный алгоритм по- иска строк Бойера — Мура достоин как минимум краткого знакомства. Алгорит- мы редакционного расстояния позволяют определить, насколько близкими друг к другу являются слова, строки или даже файлы, и найти разницу между ними.

Мы не стали рассматривать здесь алгоритмы для шифрования и дешифрования строковых и других данных. Наиболее важным и интересным из них посвящена следующая глава.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

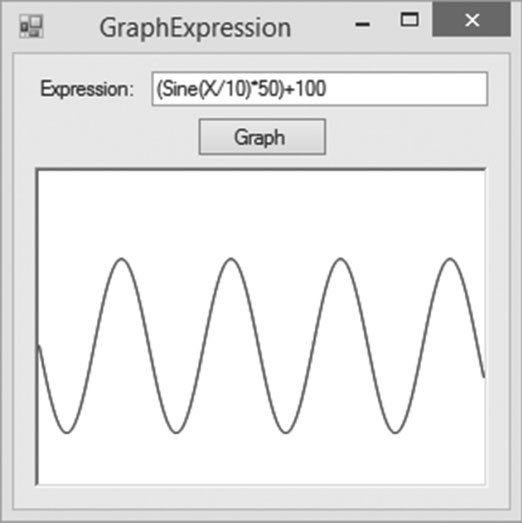
* 1. Напишите программу, содержит ли введенное пользователем выражение корректно вложенные скобки. Применять в выражении другие символы. Пример: (8  3) + (20  (7 – 3)).
  2. Напишите программу, которая разбирает и вычисляет арифметические вы- ражения, содержащие вещественные числа и операторы +, –, \* и /.
  3. Как бы вы изменили программу из упражнения 2, чтобы добавить поддержку унарного оператора отрицания? Пример: – (2 / 7).
  4. Как бы вы изменили программу из упражнения 2, чтобы добавить поддержку функций, таких как sine? Пример: 3 \* sine(45).
  5. Напишите программу, которая разбирает и вычисляет булевы выражения, такие как T&(-F|T), где *T* означает TRUE, *F* — FALSE, & — AND, | — OR, а NOT представлено знаком -.

6\*\*.Напишите программу, аналогичную той, что показана на рисунке 15.14. Она должна генерировать синтаксическое дерево для введенного пользователем выражения и затем выводить его на экран. В зависимости от способа про- рисовки графиков начальная точка в системе координат в вашем изображе- нии (0, 0), скорее всего, будет находиться в левом верхнем углу, а оси — на- правлены вправо и вниз. Координатная сетка *X* и *Y* нередко привязывается к пикселям, из-за чего итоговый график может оказаться довольно мелким. Если у вас нет опыта программирования графиков, можете не волноваться о масштабировании и подгонке результата к размерам окна.

1. Создайте таблицу переходов для диаграммы перехода состояний ДКА, по- казанной на рисунке 15.4.
2. Нарисуйте диаграмму перехода состояний ДКА, которая соответствует ре- гулярному выражению ((AB) | (BA))\*.
3. Создайте таблицу переходов для диаграммы перехода состояний, которую вы нарисовали в упражнении 8.

10\*. Напишите программу, которая позволяет выбирать переходы состояний ДКА, вводить строки и определять, принимает ли их автомат.

1. Как вы думаете, каким образом ДКА лучше принимать переходы состояний — из таблицы, аналогичной 15.1, или в виде объектов, которые представляют состояния? Почему?

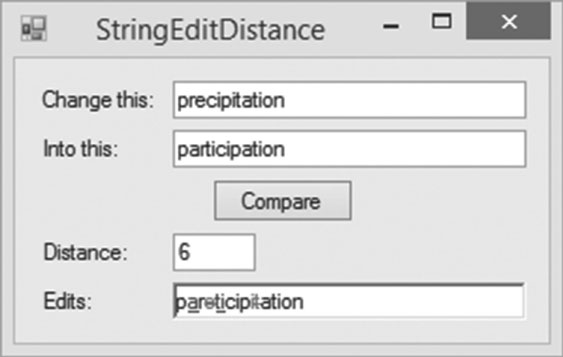


**Рис. 15.14.** Программа GraphExpression генерирует на основе выражения синтаксическое дерево и затем многократно его вычисляет, чтобы отобразить это выражение на экране

1. Как бы вы создали набор состояний для НКА, чтобы проверить вхождение шаблона в строку? Например, как бы определили вхождение в длинную строку шаблона ABA? Нарисуйте диаграмму перехода состояний, исполь- зуя блок для представления соответствующего автомата (как было сдела- но на рисунке 15.10).
2. Нарисуйте синтаксическое дерево для выражения (AB\*) | (BA\*). Нари- суйте сеть НКА, которая является результатом применения правил, описан- ных в этой главе, к вашему синтаксическому дереву.
3. Приведите диаграмму перехода состояний НКА, которую вы нарисовали в упражнении 13, к простой диаграмме перехода состояний ДКА.
4. Представьте, что вам нужно найти некую подстроку длиной *M* в тексте раз- мера *N*. Подберите вариант, при котором поиск методом перебора займет *O*(*N*  *M*) шагов.
5. Изучите редакционный граф, представленный на рисунке 15.13. Каким пра- вилом нужно руководствоваться, чтобы найти наименее затратный путь из левого верхнего в правый нижний угол? Каким будет реальное редакцион- ное расстояние?

17\*. Напишите программу, которая вычисляет редакционное расстояние.

18\*. Усовершенствуйте программу, написанную вами в упражнении 17, таким образом, чтобы она отображала изменения, необходимые для преобразова- ния одной строки в другую. Удаленные символы выводите зачеркнутыми, а вставленные — подчеркнутыми (рис. 15.15).



**Рис. 15.15.** Проходя путь по редакционному графу, можно показать, какие именно правки были нужны для превращения одной строки в другую

1. Является ли редакционное расстояние коммутативным? (То есть идентич- ны ли редакционные расстояния между парами слов 1, 2 и 2, 1?) Объясни- те свой ответ.

20\*. Измените программу, написанную вами в упражнении 17, чтобы она вычис- ляла редакционное расстояние между файлами, а не между двумя строками.

21\*. Измените программу, написанную вами в упражнении 18, чтобы она отобра- жала разницу между файлами, а не между двумя строками.

# Глава 16 КРИПТОГРАФИЯ

*Криптография* — это исследование методов безопасной передачи информа- ции, которая потенциально может быть перехвачена злоумышленниками. На ранних этапах развития криптография представляла собой обычную письмен- ность; она работала по той простой причине, что немногие тогда, на заре циви- лизации, умели читать. Позже в обиход вошли специальные алфавиты, извест- ные только отправителю и получателю. В одном из самых ранних образцов это- го вида криптографии использовались нестандартные иероглифы, высеченные в гробницах (Египет, 1900 г. до н. э.).

Другая форма криптографии, популярная в Древней Греции и Спарте, была основана на использовании деревянного стержня — скиталы. Вокруг него по спи- рали наматывалась полоска пергамента, на которой записывались слова. Когда пергамент разворачивали, буквы оказывались в неправильном положении. Что- бы прочитать сообщение, получатель должен был намотать пергамент на стержень того же диаметра.

Такой подход к криптографии иногда называют «безопасность через неясность» (*security through obscurity*), поскольку в нем предполагается, что злоумышленник не знает, в чем подвох. Но если вам известен секретный алфавит или то, что сооб- щение написано на скитале, вы сможете без труда его воспроизвести.

В более современных криптографических методиках исходят из того, что зло- умышленник знает все о способе шифрования сообщения, кроме небольшого, но крайне важного фрагмента информации, который называют *ключом*. Ключ исполь- зуется как отправителем, чтобы зашифровать данные, так и получателем, чтобы их расшифровать. Поскольку метод шифрования известен, злоумышленник тоже мо- жет расшифровать сообщение, получив ключ.

Такой подход к криптографии более надежен, чем безопасность через неясность, поскольку способ шифрования рано или поздно будет разгадан.

В этой главе описываются некоторые интересные и полезные криптографиче- ские методики. Сначала будут рассмотрены классические подходы. Они больше не считаются безопасными, но все еще представляют определенный интерес, демон- стрируя несколько практичных идей вроде частотного анализа.

Криптоанализ (исследование способов взлома и восстановления зашифрован- ных данных) зародился примерно в те же времена, что и сама криптография. В сле- дующих разделах описываются не только классические подходы к шифрованию, но и способы их взлома.

В дальнейших разделах уделяется внимание более безопасным методикам, таким как перестановочная сеть и шифрование с открытым ключом. Полное опи- сание самых последних криптографических методов, таких как AES (*Advanced Encryption Standard* — улучшенный стандарт шифрования) и *Blowfish*, выхо- дит за рамки тем, рассматриваемых в этой книге, однако общее представление о современных подходах к криптографии вы сможете получить в завершающих разделах.

## Терминология

Прежде чем приступить к изучению криптографии, вам следует усвоить не- сколько основных понятий.

Задача шифрования — сделать так, чтобы *отправитель* мог послать сообщение *получателю*, а третья сторона, которую обычно зовут *злоумышленником* или *пере- хватчиком*, не могла данное сообщение понять. Предполагается, что злоумышлен- ник в любом случае перехватит данные, поэтому помешать ему их прочитать мо- жет только шифрование.

Расшифрованное сообщение называют *открытым текстом*, а зашифрован- ное — *криптограммой*. Процесс превращения открытого текста в криптограмму именуется *шифрованием,* или *кодированием*. Восстановление открытого текста из криптограммы называется *дешифрованием,* или *декодированием*.

Формально *шифр* — пара алгоритмов для шифрования и дешифрования со- общений.

*Криптоанализ* — это исследование злоумышленником методов взлома шиф- рования.

Чтобы упростить работу с мелкими сообщениями, они обычно шифруются в верхнем регистре без пробелов или знаков препинания. Получается, что отпра- вителю и адресату нужно брать во внимание только необходимые символы — важ- ный фактор при ручном шифровании/дешифровании сообщений. Это также лиша- ет злоумышленника зацепок, которые могли бы у него появиться благодаря про- белам и знакам препинания.

Чтобы зашифрованные сообщения было легче читать, они обычно записыва- ются моноширинным шрифтом (для выравнивания текста) и разбиваются на ку- ски по пять символов каждый. Например, сообщение *This is a secret message* запи- сывается как THISI SASEC RETME SSAGE, а его зашифрованная версия будет выглядеть примерно так: TSRSH AESIS TASEM GICEE. Получателю может по- требоваться дополнительное время на выяснение того, куда нужно вставлять про- белы и знаки препинания.

Современные криптографические алгоритмы занимаются шифрованием и де- шифрованием байтовых потоков. Таким образом, в зависимости от сообщения они могут поддерживать буквы в верхнем и нижнем регистрах, пробелы, знаки препи- нания, символы *Unicode* и даже изображения. Эти алгоритмы достаточно хоро- ши для того, чтобы не дать злоумышленнику отличить пробелы и знаки препина- ния от других символов, благодаря чему он лишается дополнительной информа- ции о сообщении.

## Перестановочные шифры

В перестановочном шифре буквы открытого текста меняются местами каким-то определенным образом, результатом чего является криптограмма. Чтобы прочитать сообщение, получатель возвращает буквы на их исходные позиции.

Такие шифры частично полагаются на безопасность через неясность, если зло- умышленник не знает, какой именно метод перестановки используется. Например, метод скиталы, описанный в начале этой главы, выполняет перестановку, вызван- ную обертыванием пергамента вокруг стержня. Он полностью зависит от того, что злоумышленнику не должен быть известен способ шифрования сообщения.

Большинство из этих методик, помимо прочего, предоставляют ключ, который несет в себе определенную информацию о перестановке. Например, перестановоч- ный шифр по строкам/столбцам, описанный в следующем подразделе, использу- ет в качестве ключа количество столбцов. Однако такие ключи обычно допускают достаточно ограниченное число значений, поэтому их несложно подобрать и взло- мать тем самым шифрование, особенно если использовать компьютер.

С такими шифрами довольно легко работать с помощью карандаша и бума- ги — они могут играть роль занятных упражнений (если они кажутся вам слишком простыми, попробуйте проводить вычисления в голове).

#### Перестановка строк/столбцов

В перестановочном шифре по строкам/столбцам текстовое сообщение построч- но записывается в массив. Затем оттуда же считывается криптограмма, но уже по столбцам. Например, на рисунке 16.1 показано текстовое сообщение *THIS IS A SECRET MESSAGE*, построчно записанное в массив размерностью четыре строки на пять столбцов (обычно, если после сохранения сообщения остаются пустые ячей- ки, они заполняются буквами *X* или случайными символами).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T | H | I | S | I |
| S | A | S | E | C |
| R | E | T | M | E |
| S | S | A | G | E |

**Рис. 16.1.** В перестановочном шифре по строкам/столбцам открытый текст построчно записывается в массив, после чего оттуда по столбцам считывается криптограмма

Чтобы получить криптограмму, нужно прочитать каждый столбец сверху вниз. В нашем примере результат будет выглядеть так: TSRSH AESIS TASEM GICEE. В качестве ключа выступает количество столбцов, которые использовались при перестановке.

Для расшифровки криптограммы, в сущности, нужно выполнить операцию шифрования в обратном порядке. Создается массив, туда по вертикали записыва- ется криптограмма, а расшифрованное сообщение считывается построчно.

При реализации этого алгоритма текст не обязательно помещать в массив. Если количество столбцов равно num\_columns, вы можете просто считывать содержимое строки с открытым текстом, пропуская num\_columns между каждым символом. Данный подход представлен на примере следующего псевдокода.

String: ciphertext = «»

For col = 0 To num\_columns - 1 Integer: index = col

For row = 0 To num\_rows - 1

ciphertext = ciphertext + plaintext[index] index += num\_columns

Next row Next col

При программном дешифровании сообщения стоит принять во внимание сле- дующее: расшифровка сообщения, изначально записанного в массив с *R* строками и *C* столбцами, идентична его шифрованию в массиве с *C* строками и *R* столбцами.

В предыдущем примере сообщение вставлялось в массив размером 45. На ри- сунке 16.2 показана криптограмма *TSRSH AESIS TASEM GICEE*, построчно запи- санная в массив размером 54. Взглянув на нее, вы увидите, что открытый текст можно прочитать по столбцам.

|  |  |  |  |
| --- | --- | --- | --- |
| T | S | R | S |
| H | A | E | S |
| I | S | T | A |
| S | E | M | G |
| I | C | E | E |

**Рис. 16.2.** Расшифровывание в массиве *R**C* эквивалентно шифрованию в массиве *C**R*

Перестановка строк/столбцов — довольно простой алгоритм. Это занятное упражнение, но такую систему относительно легко взломать. В качестве секретного ключа выступает количество столбцов в массиве. Разбивая строку на отрезки раз- ной длины, можно получить несколько версий ключа. Например, предыдущая крип- тограмма содержит 20 символов. 20 делится на 1, 2, 4, 5, 10 и 20 — это все возмож- ные варианты количества столбцов. Массивы размером 120 и 201 делают крип- тограмму идентичной открытому тексту, поэтому для проверки остается всего два

варианта. Попробовав каждое из значений, вы увидите, что при четырех столбцах у вас получится тарабарщина, а вот при пяти символы сложатся в понятные слова.

Отправитель может попытаться усложнить задачу злоумышленнику, доба- вив в конец криптограммы какие-нибудь случайные символы (в частности, раз- мер массива нельзя будет определить лишь длиной сообщения). Например, если к предыдущей криптограмме добавить девять символов (чтобы получить стро- ку длиной 29), будет уже не так очевидно, что массив должен состоять из четырех или пяти столбцов.

Но даже в этом случае не составляет никакого труда написать программу, ко- торая перебирает все варианты количества столбцов между 2 и 1, которые не пре- вышают длину криптограммы. Наличие слов в соответствующем расшифрованном тексте будет означать, что ключ найден.

#### Перестановка столбцов

В перестановочном шифре по столбцам открытое сообщение построчно запи- сывается в массив, как это делалось в предыдущем случае. Затем столбцы меняют- ся местами и текст считывается по строкам.

На рисунке 16.3 показан открытый текст *THIS IS A SECRET MESSAGE*, по- строчно записанный в массив размером четыре строки на пять столбцов (слева). Столбцы меняются местами. Номера, указанные над массивом, представленным слева, обозначают порядок следования столбцов в перегруппированном массиве справа. Если считать сообщение из итогового массива по строкам, получится крип- тограмма *HTIIS ASSCE ERTEM SSAEG*.

###### 2 1 3 5 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T | H | I | S | I |
| S | A | S | E | C |
| R | E | T | M | E |
| S | S | A | G | E |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| H | T | I | I | S |
| A | S | S | C | E |
| E | R | T | E | M |
| S | S | A | E | G |

**Рис. 16.3.** В перестановочном шифре по столбцам запись открытого текста

и считывание криптограммы выполняется построчно; между этими двумя операциями происходит перестановка столбцов в массиве

В данном случае ключом является количество столбцов в массиве плюс порядок их следования. Для этого примера ключ будет выглядеть как 21354.

Более удобно было бы хранить ключ в виде слова, чья длина совпадает с ко- личеством столбцов и чьи буквы выстроены в алфавитном порядке, который со- ответствует порядку перестановок. В этом примере ключом могло бы быть слово

*CARTS*. Буква *A* в алфавите идет первой, поэтому ее значение — 1, буква *C* идет вто- рой, поэтому ее значение — 2, буква *R* идет третьей — 3 и т. д. Расставив алфавит- ные значения букв в порядке, в котором они выстроены в слове, мы получим ключ 21354, описывающий порядок следования столбцов. На практике сначала выби- рается слово, а потом уже в соответствии с ним переставляются столбцы. Не нуж- но первым делом выполнять перестановку, а потом подбирать подходящее слово.

Чтобы расшифровать сообщение, криптограмму нужно записать в массив, у ко- торого количество столбцов совпадает с длиной ключевого слова. Затем необходи- мо обратить в противоположную сторону соответствие, основанное на алфавитном порядке букв. В данном примере цифровой ключ 21354 означает, что столбцы пе- реставляются следующим образом:

* + столбец 1 передвигается в позицию 2;
  + столбец 2 передвигается в позицию 1;
  + столбец 3 передвигается в позицию 3;
  + столбец 4 передвигается в позицию 5;
  + столбец 5 передвигается в позицию 4.

Обратим соответствие в противоположную сторону:

* + столбец 2 передвигается в позицию 1;
  + столбец 1 передвигается в позицию 2;
  + столбец 3 передвигается в позицию 3;
  + столбец 5 передвигается в позицию 4;
  + столбец 4 передвигается в позицию 5.

Теперь мы можем поменять местами столбцы и построчно прочитать откры- тый текст.

Как и в случае с перестановочным шифром по строкам/столбцам, программе, которая выполняет перестановки, вовсе не обязательно записывать значения в мас- сив; ей просто нужно тщательно следить за тем, куда должен быть перемещен тот или иной символ. На самом деле, чтобы определить, на какой позиции символ дол- жен находиться в криптограмме, можно использовать обратное соответствие, рас- смотренное в предыдущих разделах.

Допустим, mapping — это массив целых чисел, описывающий перестановку столбцов. Например, если столбец 2 перемещается на позицию 1, mapping[2] = 1. Теперь представьте, что inverse\_mapping — это массив, описывающий обратное соответствие; то есть в нашем случае inverse\_mapping[1] = 2. Тогда следу- ющий псевдокод демонстрирует то, как программно закодировать открытый текст.

String: ciphertext = «»

For row = 0 to num\_rows - 1

// Считываем эту строку в измененном порядке. For col = 0 to num\_columns - 1

Integer: index = row \* num\_columns + inverse\_mapping[col] ciphertext = ciphertext + plaintext[index]

Next col Next row

Обратите внимание: этот псевдокод использует обратное соответствие для шиф- рования открытого текста. Чтобы найти символ, который привязан к определен- ному номеру столбца в криптограмме, он должен сначала узнать, из какого столб- ца данный символ был взят.

Для расшифровки криптограммы можно использовать прямое соответствие. Чтобы взломать перестановочный шифр по столбцам, злоумышленник мо-

жет записать сообщение в массив, узнав количество столбцов из длины ключево- го слова. Затем ему бы пришлось менять столбцы местами, угадывая корректный порядок. Если массив содержит *C* столбцов, количество возможных вариантов их размещения равно *C*!, что потребует перебора огромного числа комбинаций. На- пример, для 10 столбцов таких вариантов было бы 3 628 800. Это довольно боль- шое число, особенно если злоумышленник не использует компьютер, но расшиф- ровать сообщение можно инкрементально. Для начала следует попробовать найти пять первых столбцов в открытом тексте. Если удастся это сделать, в первой стро- ке будет содержаться пять символов из искомого сообщения, которые представ- ляют собой или целое слово, или как минимум его префикс. Другие строки мо- гут начинаться с отрывков, но за ними тоже будут следовать слова или префиксы. Существует всего лишь 10  9  8  7  6 = 30 240 возможных комбинаций, если взять 5 столбцов из 10; это уже намного меньше вариантов для проверки, но зада- ча по-прежнему не из легких.

#### Маршрутные шифры

В маршрутном шифре открытый текст записывается в массив (или упорядо- чивается каким-то другим способом) и затем считывается из него в порядке, ко- торый определяется конкретным маршрутом. Например, на рисунке 16.4 показа- но открытое сообщение, построчно записанное в массив. Криптограмма форми- руется путем перемещения по диагонали, начиная с левого края массива: *SRSSE ATATG HSMEI EESCI*.

T

H

I

S

I

S

A

S

E

C

R

E

T

M

E

S

S

A

G

E

**Рис. 16.4.** В маршрутном шифре открытый текст построчно записывается в массив и затем считывается оттуда в каком-то другом порядке, формируя криптограмму

Теоретически количество допустимых маршрутов через массив может быть огромным. Если в массиве содержится *N* элементов, число маршрутов через него будет равно *N*! В примере, показанном на рисунке 16.4, их будет 20! � 2,4  1018.

Тем не менее хороший маршрут должен быть достаточно простым, чтобы по- лучатель мог его запомнить. Диагональный маршрут, показанный на рисунке 16.4, годится для этой цели, но если он будет перескакивать по всему массиву в произ- вольном порядке, получателю придется его записать, что сделает ключ таким же длинным, как и само сообщение. (Позже в текущей главе вы увидите, что то же са- мое можно сказать и о ключе для схемы одноразовых блокнотов, которая ко все- му прочему меняет буквы сообщения таким образом, что злоумышленник не мо- жет получить дополнительную информацию — например, частоту использования тех или иных букв в сообщении.)

Некоторые маршруты оставляют большие фрагменты текста нетронутыми или сохраняют их в обратном порядке. Например, спираль, ведущая внутрь по часовой стрелке, начиная с левого верхнего угла, легко запоминается, но при этом первая строка сообщения попадает в криптограмму неизменной. Подобные маршруты дают злоумышленнику дополнительные сведения, что может облегчить их подбор.

Если отказаться от маршрутов, которые сложно запоминаются и содержат круп- ные фрагменты нетронутого открытого текста, количество оставшихся вариантов окажется значительно меньше теоретического максимума.

## Шифры подстановки

В шифрах подстановки буквы открытого текста заменяются другими. В следу- ющих разделах описываются четыре распространенных шифра этого типа.

#### Шифр Цезаря

Примерно 2100 лет назад Юлий Цезарь (100 г. до н. э. — 44 г. до н. э.) кодировал сообщения, которые он отправлял своим офицерам, с помощью простого шифра подстановки. В этой версии шифра он сдвигал каждый символ сообщения на три буквы в алфавите. Буква *A* превращалась в *D*, *B* — в *E* и т. д. Чтобы расшифровать сообщение, получатель вычитал 3 из каждой буквы: *Z* становилась *W*, *Y* — *V* и т. д.

Например, сообщение *This is a secret message* со сдвигом в три позиции выгля- дит как *WKLVL VDVHF UHWPH VVDJH*.

Племянник Юлия Цезаря, Август, использовал похожий шифр, только с оди- нарным сдвигом вместо тройного. В принципе вы можете сдвигать буквы в откры- том тексте на любое количество символов.

Злоумышленник способен раскодировать сообщение, зашифрованное этим ме- тодом, изучив частоту использования букв в криптограмме. В английском языке буква *E* встречается намного чаще других — приблизительно в 12,7 % случаев. Вто- рая по популярности буква, *T*, имеет частоту 9,1 %. Если злоумышленник подсчи- тает, сколько раз каждая буква встречается в криптограмме, наиболее используе- мой из них, вероятно, окажется зашифрованная буква *E*. Посчитав разницу меж- ду буквой *E* и закодированным символом, он получит сдвиг, который применял- ся при шифровании.

Такой метод взлома лучше всего подходит для длинных сообщений, поскольку в коротких частота распределения букв может оказаться нетипичной.

В таблице 16.1 показано количество вхождений разных букв в криптограмму

*WKLVL VDVHF UHWPH VVDJH*.

**Òàáëèöà 16.1.** ×àñòîòà èñïîëüçîâàíèÿ áóêâ â âûøåïðèâåäåííîé êðèïòîãðàììå

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Буква** | D | F | H | J | K | I | P | U | V | W |
| **Число**  **вхождений** | 2 | 1 | 4 | 1 | 1 | 2 | 1 | 1 | 5 | 2 |

Если предположить, что *V* — зашифрованная буква *E*, тогда размер сдвига дол- жен быть равен 17. Расшифровав сообщение с этим сдвигом, вы получите текст вида *FTUEU EMEQO DQFYQ EEMSQ*, который не содержит настоящих слов.

Если принять за *E* вторую по использованию букву в криптограмме, *H*, то сдвиг будет равен 3, что позволит расшифровать исходное сообщение.

#### Шифр Виженера

У подстановочного шифра Цезаря есть одна проблема — для него существу- ет всего 26 ключей. Злоумышленник может с легкостью перепробовать все 26 ва- риантов сдвига и найти тот, который позволяет получить членораздельный текст. Шифр Виженера устраняет подобный недостаток, используя разную длину сдви- га для разных букв в сообщении.

**ЗАМЕЧАНИЕ**

Шифр Виженера изначально был описан в 1553 г. Джованом Баттистой Бел- лазо, но в XIX в. его авторство начали приписывать Блезу Виженеру. С тех пор это название прочно вошло в обиход.

В шифре Виженера ключевое слово обозначает сдвиги для разных букв в сооб- щении. Каждый символ в ключевом слове указывает сдвиг исходя из своей пози- ции в алфавите. *A* означает нулевой сдвиг, *B* — сдвиг на одну позицию и т. д.

Чтобы расшифровать сообщение, открытый текст вставляется снизу от ключе- вого слова, которое повторяется, пока не достигнет длины сообщения. На рисун- ке 16.5 показан текст, записанный снизу от повторяющегося несколько раз клю- чевого слова ZEBRAS.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Z* | *E* | *B* | *R* | *A* | *S* | *Z* | *E* | *B* | *R* | *A* | *S* | *Z* | *E* | *B* | *R* | *A* | *S* | *Z* | *E* |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *T* | *H* | *I* | *S* | *I* | *S* | *A* | *S* | *E* | *C* | *R* | *E* | *T* | *M* | *E* | *S* | *S* | *A* | *G* | *E* |

**Рис. 16.5.** В шифре Виженера ключевое слово записывается сверху от открытого текста, повторяясь необходимое количество раз

Теперь с помощью соответствующих букв можно создать шифрограмму. На- пример, ключевая буква *Z* обозначает сдвиг 25, поэтому буква открытого текста *T* превращается в *S*.

Чтобы упростить процесс сдвига, можно использовать «таблицу умножения» по примеру той, которая показана на рисунке 16.6. Для кодирования буквы откры- того текста *T* с буквой ключа *Z* нужно найти строку *T* и столбец *Z*.

Буква ключа



Буква сообщения

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | A B B C C D D E   1. F 2. G 3. H H I I J 4. K 5. L 6. M 7. N N O O P P Q Q R R S S T 8. U 9. V 10. W 11. X 12. Y 13. Z 14. A | | C D E F G H I J K L M N O P Q R S T U V W X Y Z A  B | D E F G H I J K L M N O P Q R S T U V W X Y Z A B  C | E F G H I J K L M N O P Q R S T U V W X Y Z A B C  D | F G H I J K L M N O P Q R S T U V W X Y Z A B C D  E | G H I J K L M N O P Q R S T U V W X Y Z A B C D E  F | 1. I J 2. J K 3. K L 4. L M 5. M N M N O N O P O P Q P Q R 6. R S 7. S T 8. T U 9. U V 10. V W 11. W X 12. X Y 13. Y Z 14. Z A 15. A B 16. B C 17. C D 18. D E 19. E F 20. F G 21. G H 22. H I | | | K L M N O P Q R S T U V W X Y Z A B C D E F G H I  J | L M N M N O N O P O P Q P Q R   1. R S 2. S T 3. T U 4. U V 5. V W 6. W X 7. X Y 8. Y Z 9. Z A 10. A B 11. B C 12. C D 13. D E 14. E F 15. F G 16. G H 17. H I 18. I J 19. J K 20. K L 21. L M | | | O P Q R S T U V W X Y Z A B C D E F G H I J K L M  N | P Q R S T U V W X Y Z A B C D E F G H I J K L M N  O | Q R S T U V W X Y Z A B C D E F G H I J K L M N O  P | R S T U V W X Y Z A B C D E F G H I J K L M N O P  Q | S T U V W X Y Z A B C D E F G H I J K L M N O P Q  R | T U V W X Y Z A B C D E F G H I J K L M N O P Q R  S | 1. V 2. W 3. X 4. Y 5. Z 6. A   A B B C C D   1. E 2. F 3. G 4. H 5. I 6. J 7. K 8. L 9. M 10. N 11. O 12. P 13. Q 14. R 15. S 16. T 17. V | | W X Y Z A B C D E F G H I J K L M N O P Q R S T V  W | 1. Y Z 2. Z A 3. A B A B C B C D 4. D E 5. E F 6. F G 7. G H 8. H I 9. I J 10. J K 11. K L 12. L M 13. M N M N O N O P O P Q 14. Q R 15. R S 16. S T 17. T V 18. V W 19. W X 20. X Y 21. Y Z | | |
| B |
| C |
| D |
| E |
| F |
| G |
| H |
| I |
| J |
| K |
| L |
| M |
| N |
| O |
| P |
| Q |
| R |
| S |
| T |
| U |
| V |
| W |
| X |
| Y |
| Z |

**Рис. 16.6.** Приведенная таблица упрощает сдвиг букв в шифре Виженера

Чтобы расшифровать букву криптограммы, ее нужно искать в столбце буквы ключа. Расшифрованный символ берется из строки.

Простой частотный анализ, с помощью которого можно взломать шифр Цезаря, здесь не сработает, поскольку не все буквы имеют одинаковый сдвиг. Однако для взлома можно воспользоваться частотой вхождения букв в криптограмму.

Допустим, ключевое слово имеет длину *K*. В этом случае каждая *K*-я буква (на- пример, 1, *K* + 1, 2  *K* + 1 и т. д.) имеет один и тот же сдвиг. В открытом тексте эти буквы могут отличаться, но их относительные частоты совпадают.

Процесс взлома начинается с попытки угадать длину ключа и поисков букв с одинаковым сдвигом. Например, можно попробовать ключ длиной 2, проверив буквы под номерами 2, 4, 6, 8 и т. д. Если мы угадали с длиной, частота букв долж- на быть подобна той, что в обычном английском языке (или в языке, который вы используете). В частности, несколько букв, которые соотносятся с символами от- крытого текста (такими как *E*, *S* и *T*), должны встречаться намного чаще других, таких как *X* и *Q*.

Если длина ключа не равна 2, частоты букв должны распределяться достаточно равномерно — ни одна из них не должна встречаться намного чаще других. В этом случае придется пробовать другую длину и повторять все заново.

Если вы найдете длину ключа, при которой частота распределения похожа на ту, что существует в английском языке, вам нужно будет проверять отдельные ча- стоты, как это делалось в подстановочном шифре Цезаря. Буква, которая встреча- ется чаще других, скорее всего, представляет зашифрованную *E*.

Точно так же нужно проверить другие буквы с тем же сдвигом, чтобы опре- делить, насколько они смещены на самом деле. В сущности, на данном этапе происходит декодирование подстановочного шифра Цезаря для каждой бук- вы в ключе.

Результатом должен стать сдвиг для каждого символа в ключевом слове. Взлом этого шифра требует больше усилий, чем в предыдущем случае, но он все равно возможен.

#### Простая подстановка

В простом подстановочном шифре у каждой буквы есть фиксированная за- мена. Например, вместо *A* можно подставить *H*, вместо *B* — *J*, вместо *C* — *X* и т. д.

В качестве ключа здесь выступает соответствие между буквами открытого тек- ста и шифрограммы. Если в сообщении используются только буквы английского алфавита (*A*–*Z*), число возможных соответствий будет равно 4,0  1026.

В том случае, когда вы шифруете и расшифровываете сообщения вручную, вам необходимо записывать используемые соответствия.

Если в вашем распоряжении есть компьютер, можно попробовать воссоздать соответствие с помощью генератора псевдослучайных чисел. Отправитель вы- бирает число *K* и использует его для инициализации генератора, который затем придает случайный характер буквам *A*–*Z* и создает соответствие. Значение *K* становится ключом. Получатель выполняет те же шаги, инициализируя гене- ратор случайных чисел и восстанавливая соответствие, которое использова- лось отправителем.

Единственное число запомнить проще, чем соответствия для всех букв, но в большинстве генераторов случайных чисел количество возможных внутренних состояний куда меньше, чем 4,0  1026. Например, если число, выбранное вами для инициализации генератора, является целым и имеет знак, у ключа будет лишь око- ло 2 млрд потенциальных значений. Это все равно много, однако компьютер спосо- бен с легкостью их перебрать и найти число, которое дает членораздельный текст.

Чтобы немного упростить данный процесс, можно также использовать частот- ность. Если в криптограмме больше всего встречается буква *W*, то это, скорее все- го, зашифрованная *E*.

#### Схема одноразовых блокнотов

*Схема одноразовых блокнотов* — это некая разновидность шифра Виженера, в которой ключ имеет ту же длину, что и сообщение. Сдвиг присваивается каж- дой букве отдельно, поэтому для его поиска нельзя использовать частотный ана- лиз криптограммы.

Поскольку зашифрованные буквы могут быть сдвинуты как угодно, то и соот- ветствовать им по всей вероятности могут любые символы в открытом тексте. Сле- довательно, злоумышленник не имеет возможности извлечь какую-либо информа- цию из криптограммы (если не считать длины сообщения; хотя вы можете скрыть и ее, добавив в текст дополнительные буквы).

Если подходить к процессу вручную, отправитель и получатель должны иметь по одному блокноту с одинаковым набором случайных букв. С помощью этих букв отправитель шифрует сообщение, вычеркивая их после того, как они были использо- ваны, и больше не прибегая к ним снова. Те же буквы применяются для расшифров- ки сообщения получателем; после использования они точно так же вычеркиваются.

Поскольку каждая буква, в сущности, имеет свой собственный сдвиг, шифр нельзя будет взломать до тех пор, пока у злоумышленника не окажется одна из ко- пий одноразового блокнота.

Недостаток подобной схемы заключается в том, что отправитель и получатель должны обладать идентичными копиями блокнота, а безопасная передача этих ко- пий может оказаться такой же сложной, как и отправка самого сообщения. Когда-то блокнотами обменивались при помощи курьеров. Если курьера перехватывали, блокнот выбрасывался и вместо него высылался новый.

**ЗАМЕЧАНИЕ**

 В программной реализации схемы одноразовых блокнотов для шифрования каждого символа вместо сдвига можно использовать побитовый оператор XOR

(исключающее ИЛИ). Если в качестве байтов в блокноте взять случайные зна- чения от 0 до 255, зашифрованный результат тоже будет иметь вид случайных чисел в этом диапазоне.

## Блочные шифры

В блочном шифре сообщение разбивается на блоки, которые затем отдельно шифруются и объединяются в криптограмму.

Во многих блочных шифрах кодирование выполняется путем многократной трансформации данных. Трансформация должна быть обратимой, чтобы поз- же криптограмму можно было расшифровать. Разбиение текста на одинаковые блоки позволяет предусмотреть изменения фрагментов определенной длины.

Полезное свойство блочных шифров заключается в том, что они позволяют криптографическому программному обеспечению работать с относительно не- большими фрагментами текста. Представьте, например, что вам нужно зашифро- вать очень длинное сообщение размером, скажем, в несколько гигабайтов. Если ис- пользовать шифр с перестановкой по столбцам, программе придется переходить в разные участки сообщения в памяти. Это может привести к сбрасыванию дан- ных в файл подкачки, что существенно замедлит работу.

Блочный шифр ведет себя иначе. Он может анализировать сообщение по частям, которые легко помещаются в памяти. Даже если программе придется использовать файл подкачки, каждый фрагмент сообщения будет загружаться в память всего один раз.

В подразделах, представленных ниже, описываются наиболее распространен- ные виды блочных шифров.

#### Подстановочно-перестановочные сети

Шифр подстановочно-перестановочной сети совершает несколько череду- ющихся раундов, состоящих из подстановочной и перестановочной стадий. Пред- ставьте, что эти стадии выполняются автоматами, размещенными внутри блоков двух видов: подстановочных (*S*-блоки) и перестановочных (*P*-блоки).

*S*-блок берет небольшой фрагмент текста и объединяет его с частью ключа, запу- тывая тем самым конечный результат. Чтобы «замести следы» как можно сильнее, измените один-единственный бит ключа — в идеале это должно привести к транс- формации примерно половины итоговых битов. Например, если *S*-блок имеет дело с 1 байтом, он способен применить операцию XOR, чтобы совместить первый бит ключа с битами 1, 3, 4 и 7, взятыми из текста. В разных шаблонах могут соединять- ся разные биты ключа и сообщения. Вы можете варьировать *S-*блоки для разных частей текстового фрагмента.

*P*-блок меняет местами биты во всем фрагменте и отправляет их в разные *S*-блоки. Например, бит 1 из первого *S*-блока может отправиться к биту 7 из сле- дующего раунда в третьем *S*-блоке.

На рисунке 16.7 показан трехраундовый шифр подстановочно-перестановочной сети. *S*-блоки *S*1–*S*4 совмещают ключ с фрагментами сообщения (стоит отметить, что в каждом раунде могут использоваться разные части ключа). Все *P*-блоки выпол-

няют одну и ту же перестановку, отправляя результат работы *S*-блоков в *S*-блоки следующего раунда.

Чтобы расшифровать сообщение, нужно выполнить те же шаги, только в об- ратном порядке. Криптограмма пропускается через перевернутые *S*-блоки, после чего результаты поступают в перевернутый *P*-блок; все это повторяется необходи- мое количество раундов.

Один из недостатков такого подхода заключается в том, что *S*- и *P*-блоки долж- ны уметь работать в обратном направлении, чтобы сообщения можно было рас- шифровывать. Кроме того, для шифрования и расшифровки нужен разный код, что усложняет процесс написания, отладки и сопровождения программ.

**ЗАМЕЧАНИЕ**

AES (Advanced Encryption Standard — улучшенный стандарт шифрования), ве- роятно, наиболее распространенный на сегодня метод шифрования, тоже основан на подстановочно-перестановочной сети. Он использует фрагменты размером 128 бит и 128-, 192- или 256-битные ключи, в зависимости от же- лаемого уровня безопасности.

Чтобы осознать, какое количество ключей из этого можно извлечь, достаточ- но принять во внимание, что 2128 � 3,4  1038 и 2256 � 1,2  1077. Если компьютер злоумышленника способен проверить 1 млрд ключей за секунду (что доволь- но маловероятно для обычного персонального компьютера, учитывая слож- ность шагов, необходимых для шифрования сообщения), на перебор всех 128- и 256-битных ключей у него уйдет, соответственно, 1,1  1022 и 3,7  1060 лет.

Метод AES использует разные количества раундов в зависимости от размера ключа: 10 раундов для 128-битных ключей, 12 раундов для 192-битных и 14 ра- ундов для 256-битных. Чем больше раундов и длиннее ключ, тем сильнее за- путывается сообщение, что замедляет атаку методом перебора.

Больше о методе AES можно узнать по адресу https://ru.wikipedia.org/wiki/ Advanced\_ Encryption\_Standard.



*S*3

*S*2

*S*1

*P*

*S*3

*S*2

*S*1

*P*

*S*3

*S*2

*S*1

Открытый текст

Криптограмма

*S*4

*S*4

*S*4

**Рис. 16.7.** В шифре подстановочно-перестановочной сети стадии подстановки и перестановки чередуются

#### Шифр Фейстеля

В шифре Фейстеля, названном в честь криптографа Хорста Фейстеля, сообще- ние разбивается пополам на левую и правую части — *L*0 и *R*0. К правой части при- меняется функция, и полученный результат совмещается с левой частью с помо-

щью операции XOR. Затем эти два фрагмента меняются местами и процесс повто- ряется определенное количество раундов.

Высокоуровневое описание такого алгоритма показано в виде следующих шагов.

1. Разбиваем открытый текст пополам на два блока — *L*0 и *R*0.
2. Повторяем:
   1. присваиваем *Li*+1 = *Ri*;
   2. присваиваем *Ri*+1 = *Li XOR F*(*Ri*, *Ki*).

*Ki* — это подключ, который используется в раунде *i* (набор значений, сгенери- рованных с помощью ключа сообщения). Например, проще всего было бы разбить ключ на части и использовать их по порядку, повторяя при необходимости (это похоже на шифр Виженера, который задействует каждую букву ключа для шиф- рования одного символа из открытого текста и затем, если понадобится, повторя- ет данные буквы).

После выполнения нужного количества раундов криптограмма будет иметь вид *Li* + 1 + *Ri* + 1.

Чтобы расшифровать сообщение, нужно получить итоговые значения *Li* + 1 и *Ri* + 1, разделив криптограмму пополам. Если взглянуть на предыдущие шаги, мож- но увидеть, что *Ri* — это *Li* + 1. Поскольку мы уже знаем, чему равно *Li* + 1, значение *Ri* нам тоже известно.

Чтобы восстановить *Li*, заменим в уравнении, использованном на шаге 2b, *Li* + 1

на *Ri*. Получим следующее:

*Ri* + 1 = *Li XOR F*(*Ri*, *Ki*) = *Li XOR F*(*Li* + 1, *Ki*).

На данном этапе нам уже известно значение *Li* + 1, поэтому мы можем вычислить *F*(*Li* + 1, *Ki*). Если совместить это с *Ri* + 1, условия *F*(*Li* + 1, *Ki*) нивелируются. В резуль- тате остается только восстановленное значение *Li*.

Алгоритм расшифровывания описан в виде следующих шагов.

1. Разделяем криптограмму пополам, на *Li* + 1 и *Ri* + 1.
2. Повторяем:
   1. присваиваем *Ri* = *Li* + 1;
   2. присваиваем *Li* = *Ri* + 1 *XOR F*(*Li* + 1, *Ki*).

Одним из преимуществ шифров Фейстеля является то, что расшифровывание не требует инвертировать функцию *F*. Это означает, функция *F* может быть любой, даже такой, которую сложно обратить в противоположную сторону.

Еще одно преимущество шифров Фейстеля заключается в том, что процессы шифрования и расшифровывания практически совпадают. Отличается только по- рядок использования подключей. Это означает, что для шифрования и расшифро- вывания нужен всего лишь один фрагмент кода.

**ЗАМЕЧАНИЕ**

 Стандарт шифрования данных (*Data Encryption Standard*, или DES), который до недавних пор был одним из наиболее популярных методов шифрования,

основан на шифре Фейстеля. Он больше не считается достаточно безопасным для высокозащищенных приложений, в основном из-за относительно корот- кого 56-битного ключа.

Разновидность этого метода под названием Triple DES просто применяет алгоритм DES по три раза к каждому блоку. Triple DES считается безопас- ным, хотя в высокозащищенных приложениях вместо него теперь исполь- зуют AES.

Больше о методе DES можно узнать по адресу https://ru.wikipedia.org/wiki/DES.

## Шифрование с открытым ключом и RSA

В шифровании с открытым ключом используется два отдельных ключа — от- крытый и закрытый. Открытый ключ обычно публикуется, поэтому его может узнать кто угодно (включая злоумышленника). Закрытый ключ известен только получателю.

Отправитель использует открытый ключ для шифрования сообщения и пере- дает результат получателю. Однако расшифровать его можно только с помощью закрытого ключа.

В других типах шифрования для кодирования и декодирования сообщения используется один и тот же ключ, поэтому их называют *симметричными*.

Одним из самых известных алгоритмов шифрования с открытым ключом яв- ляется RSA, названный в честь тех, кто впервые его описал: Рональда Ривеста, Ади Шамира и Леонарда Адлемана.

**ОСТОРОЖНО: МАТЕМАТИКА**

Алгоритм RSA является крайне интересным, и определенное понимание прин- ципов его работы может пригодиться при прохождении собеседования. К сожа- лению, он, кроме всего прочего, еще и очень «математический». Если вам не по душе данная наука, можете сразу переходить к подразделу «Практические со- ображения» раздела «Шифрование с открытым ключом и RSA» в текущей главе.

Чтобы сгенерировать открытый и закрытый ключи для данного алгоритма, нужно выполнить следующие шаги.

1. Выбираем два больших простых числа, *p* и *q*.
2. Вычисляем *n* = *p*  *q*. Это будет модуль открытого ключа.
3. Вычисляем (*n*), где  — функция Эйлера (подробнее об этом чуть позже).
4. Выбираем целое число *e* в диапазоне 1 *e*  (*n*); *e* и (*n*) должны быть вза- имно простыми (то есть у них не должно быть общих делителей). Это будет экспонента открытого ключа.
5. Вычисляем значение *d*, обратное к числу *e* по модулю (*n*). Другими слова- ми, *e*  *d*  1 mod (*n*) (об этом тоже поговорим чуть ниже). Значение d яв- ляется закрытым ключом.
6. Открытый ключ состоит из значений *n* и *e*. Чтобы зашифровать цифровое сообщение *M*, отправитель использует формулу *C* = *M*e mod *n*.

**ЗАМЕЧАНИЕ**

 В главе 2 описывается несколько методик, необходимых для реализации алгоритма RSA. В частности, там объясняется, как провести вероятностную

проверку, чтобы определить, является ли число простым. Чтобы найти большое простое число, выберите случайное большое значение и проверьте, простое ли оно. Повторяйте до тех пор, пока не получите нужный вам результат.

В главе 2 также рассказывалось об эффективном возведении в степень и о том, как с помощью алгоритма Евклида быстро определить, имеют ли два числа общий делитель.

Стойкость алгоритма RSA основывается на том факте, что разложение боль- ших чисел на множители является сложной задачей. Если злоумышленнику удаст- ся разложить на множители открытый модуль *n*, он сумеет восстановить простые числа *p* и *q*. Имея *p* и *q*, а также открытую экспоненту *e*, он сможет подобрать за- крытый ключ и взломать шифр.

Вот почему простые числа *p* и *q* должны быть большими — так злоумышлен- нику будет труднее разложить *n* на множители.

**ЗАМЕЧАНИЕ**

 Интересный факт: хотя задача разложения на множители считается сложной и многие люди потратили на ее изучение огромное количество времени, нет

никакой гарантии того, что однажды кто-нибудь придумает быстрый способ разложения на множители больших чисел.

#### Функция Эйлера

В шагах 1 и 2 алгоритма генерирования ключа непосредственно используются методики, описанные в главе 2.

На шаге 3 необходимо вычислить функцию Эйлера, (*n*) (ее еще называют фи- функцией). Она возвращает количество натуральных чисел, меньших определен- ного числа *n* и не имеющих с ним общего делителя. Например, (12) равно 4, по- скольку существует четыре числа меньших, чем 12, и выступающих взаимно про- стыми с ним: 1, 5, 7 и 11.

Простое число является взаимно простым по отношению к любому целому по- ложительному значению, которое меньше этого числа, поэтому (*p*) = *p* – 1, если *p* простое.

Оказывается, если p и q являются взаимно простыми, то (*p*  *q*) = (*p*)  (*p*). Это возможно, если они являются простыми по отдельности, поэтому в шаге 3 име- ем: (*n*) = (*p*  *q*) = (*p*)  (*q*) = (*p* – 1)  (*q* – 1). Это легко вычислить.

Представим, например, что *p* = 3 и *q* = 5. Тогда (15) = (3)  (5) =

= (3 – 1)  (5 – 1) = 2  4 = 8. Действительно, всего существует восемь по-

ложительных целых чисел, которые меньше 15 и взаимно простые с 15: 1, 2,

4, 7, 8, 11, 13 и 14.

#### Обратные величины

Шаг 4 в алгоритме генерирования ключа довольно прост. Достаточно выбрать случайные числа в диапазоне между 1 и (*e*) и проверить с помощью алгоритма Ев- клида, являются ли они взаимно простыми по отношению к *e*.

В шаге 5 необходимо обнаружить значение *d*, обратное к числу e по модулю

(*n*). Другими словами, нужно найти такое *e*, при котором *e*  *d*  1 mod (*n*).

Один из простых способов получить обратную величину заключается в вычис- лении (1  *d*) mod (*n*), (2  *d*) mod (*n*), (3  *d*) mod (*n*) и т. д., пока не будет най- дено значение, дающее результат 1.

Чтобы сделать поиск значения *e* более эффективным, можно воспользоваться расширенным алгоритмом Евклида. Подробности ищите по адресу http:// ru.wikipedia. org/wiki/Àëãîðèòì\_Åâêëèäà.

#### Пример использования RSA

Для начала рассмотрим образец подбора открытого и закрытого ключей.

1. Выберем два больших простых числа *p* и *q*.
2. Пусть *p* = 17 и *q* = 29. В реальных условиях следует выбирать значительно большие значения (например, 128-битные числа, записанные в двоичном виде), порядка 11038.
3. Вычислим *n* = *p*  *q*. Это будет модуль открытого ключа.
4. Модуль открытого ключа *n* равен *p*  *q* = 493.
5. Вычислим (*n*), где  — функция Эйлера.

6. Значение (*n*) = (*p* – 1)  (*q* – 1) = 16  28 = 448.

1. Выберем целое число *e* в диапазоне 1 *e*  (*n*); *e* и (*n*) должны быть взаимно простыми.
2. В этом примере нужно выбрать такое *e*, чтобы оно находилось в диапазоне 1  *e*  448 и было взаимно простым по отношению к 448. Разложение на простые сомножители числа 448 дает 26  7, поэтому *e* не может делиться на 2 и 7. Пусть *e* = 3  5  11 = 165.
3. Вычислим значение *d*, обратное к числу *e* по модулю (*n*). Другими слова- ми, *e*  *d*  1 mod (*n*).
4. В этом примере нужно найти значение, обратное к 165 mod 448. То есть по- лучить такое *d*, при котором *d*  165  1 mod 448. В нашем случае 429  165 

 1 mod 448, поэтому обратная величина будет равна 429. (Демонстрационная программа MultiplicativeInverse, которую можно загрузить среди других при- меров к этой главе, вычисляет обратные величины, аналогичные вышепри- веденной. Она проверяет все значения, пока не находит такое, при котором 429 является корректным результатом.)

Итак, открытая экспонента *e* = 165, открытый модуль *n* = 493, а закрытый ключ

*d* = 429.

Теперь представьте, что вам нужно зашифровать значение 321 (назовем его *C*). *C* должно быть равно *Me* mod *n* = 321165 mod 493. Программа ExponentiateMod, яв- ляющаяся частью решения к упражнению 11 из главы 2 и доступная для загрузки на веб-сайте книги, быстро вычисляет большие экспоненты. Зашифрованное зна- чение, которое она выдает, равно 321165 mod 493 = 359.

Чтобы расшифровать значение 359, получателю необходимо вычислить *Cd* mod *n*. В нашем случае это будет 359429 mod 493. Программа ExponentiateMod из главы 2 дает результат 359429 mod 493 = 321, поэтому расшифрованное значение, как и ожи- далось, равно 321.

#### Практические соображения

На создание надежных закрытых ключей и вычисление больших экспонент может уйти некоторое время, даже если использовать быстрое возведение в сте- пень по модулю. Как вы помните, *p* и *q* являются очень большими числами, поэто- му применение криптографии с закрытым ключом для шифрования длинных со- общений (с предварительной разбивкой на мелкие блоки, которые легко предста- вить в виде чисел) может оказаться довольно медленным.

Для экономии времени в некоторых криптографических системах применя- ется шифрование с открытым ключом. Это позволяет отправителю и получателю обмениваться закрытыми ключами и использовать симметричное шифрование.

**ЗАМЕЧАНИЕ**

Популярная программа под названием Pretty Good Privacy (PGP) использует шифрование с открытым ключом как минимум для части своих вычислений. Чтобы сделать криптограмму максимально запутанной, сохранив при этом уме- ренную длину сообщения и приемлемую скорость, PGP обрабатывает данные в несколько этапов, включая хеширование, сжатие, а также шифрование с от- крытым и закрытым ключами.

Больше о PGP можно узнать по адресу https://ru.wikipedia.org/wiki/PGP.

## Другие области применения криптографии

Алгоритмы, описанные в этой главе, нацелены на шифрование и расшифро- вывание сообщений, однако криптографию можно использовать и в других целях.

Например, криптограграфическая функция хеширования принимает в каче- стве ввода блок данных, такой как файл, и возвращает хеш, с помощью которого эти данные можно идентифицировать. После файл и его хеш можно сделать пу- блично доступными.

Получатель, который хочет использовать этот файл, должен выполнить ту же функцию и проверить, совпадает ли его хеш с публичным. Если кто-то подделал файл, совпадения быть не должно; таким образом, адресат будет знать, что данные дошли к нему в измененном виде.

Хорошая функция хеширования должна:

* + легко вычисляться;
  + максимально усложнять злоумышленнику задачу создания файла с уста- новленным значением хеша (чтобы нельзя было заменить оригинальный файл поддельным);
  + максимально усложнять изменение файла без преобразования хеша;
  + максимально усложнять поиск двух файлов с одним и тем же хешем.

Одна из возможностей применения криптографического хеширования заклю- чается в проверке паролей. Когда вы создаете пароль, система сохраняет его хеш. Но она не хранит сам пароль, поэтому злоумышленник не может его заполучить.

Затем при входе в систему вы снова вводите свой пароль. Для него создается хеш, который сравнивается с уже сохраненным значением.

*Цифровая подпись* — это криптографический инструмент, чем-то напомина- ющий криптографическое хеширование. Если вы хотите доказать, что какой-то конкретный документ написан вами, его нужно подписать. Позже можно будет проверить подлинность вашей подписи. Если кто-то другой изменит документ, он не сможет подписать его от вашего имени.

Обычно система цифровых подписей состоит из трех частей:

* + - алгоритм генерирования закрытых и открытых ключей;
    - алгоритм, который подписывает документ с помощью закрытого ключа;
    - алгоритм, проверяющий подлинность вашей подписи с помощью откры- того ключа.

Цифровая подпись в некотором смысле является противоположной по отноше- нию к системе шифрования с закрытым ключом. Сообщение может быть зашиф- ровано с помощью открытого ключа любым числом отправителей, но расшифро- вать его сумеет только один из получателей, обладающий закрытым ключом. В си- стеме цифровых подписей сообщение подписывается отправителем с помощью за- крытого ключа, а открытый ключ, позволяющий проверить его подлинность, до- ступен любому количеству получателей.

## Резюме

В этой главе было рассмотрено несколько криптографических алгоритмов. Про- стейшие из них, такие как перестановочные и подстановочные шифры, не являют- ся безопасными с точки зрения криптографии, но могут представлять собой инте- ресные упражнения. Любой, кто изучает алгоритмы, должен иметь о них какое-то представление, особенно это касается шифров Цезаря и Виженера.

Далее мы познакомились с современными алгоритмами и узнали, какие из них все еще находятся в употреблении. AES (использует подстановочно-перестановочные сети) и RSA (основан на шифровании с открытым ключом) являются сегодня наи- более распространенными. Стандарт DES больше не считается достаточно безопас- ным, однако в нем применяется шифр Фейстеля, который сам по себе может быть интересным; к тому же на его основе легко создавать безопасные методы шифрова- ния, такие как Triple DES.

Эта глава охватывает всего лишь незначительную часть изученных криптогра- фических алгоритмов. Больше информации можно найти в Интернете или кни- ге, посвященной криптографии. Свои поиски следует начать с веб-сайтов https:// ru.wikipedia.org/wiki/Êðèïòîãðàôèÿ и [http://mathworld.wolfram.com/Cryptography.html.](http://mathworld.wolfram.com/Cryptography.html) Если вы пред- почитаете печатный вариант, настоятельно рекомендую вам второе издание кни- ги Брюса Шнайера «Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си» (Триумф, 2002). В нем описано огромное количество алгорит- мов для шифрования и расшифровывания, цифровых подписей, аутентификации, защищенной системы выборов и цифровой валюты.

Эти алгоритмы исходят из предположения, что при наличии ключа можно до- вольно легко выполнить определенные вычисления и что без него этого сделать нельзя. Например, в стандарте RSA получателю не составляет труда расшифровы- вать сообщения, однако злоумышленнику нужно разложить на множители комби- нацию из двух больших простых чисел. Эта операция считается крайне сложной, благодаря чему шифрование RSA нельзя взломать.

В теории алгоритмов выделяют два чрезвычайно важных класса проблем: *P* и *NP*. К первому классу относят проблемы, которые относительно просто решить, — например, умножение двух чисел или поиск бинарного дерева для фрагмента ин- формации. Во второй класс входят проблемы повышенной сложности, такие как упаковка в контейнеры, задача о ранце и задача коммивояжера, описанные в гла- ве 12.

Следующая глава посвящена классам задач *P* и *NP*. В ней рассматриваются не- которые интересные вопросы, касающиеся этих классов, а также проблемы, кото- рые остаются нерешенными.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

1. Напишите программу, которая шифрует и расшифровывает сообщения с по- мощью перестановочного шифра по строкам/столбцам.
2. Напишите программу, которая шифрует и расшифровывает сообщения с по- мощью перестановочного шифра по столбцам.
3. Перестановочный шифр по столбцам использует относительный алфавит- ный порядок для букв своих ключей, чтобы определить соответствие столб- цов. Что произойдет, если в ключе некоторые буквы будут дублироваться, как в *PIZZA* или *BOOKWORM*? Как решить эту проблему? Имеет ли такой подход какие-либо преимущества?
4. Перестановочный шифр по столбцам меняет местами столбцы в сообщении, сохраненном в массиве. Удастся ли улучшить безопасность, если перестав- лять не только столбцы, но и строки?
5. Напишите программу, которая подобна шифру перестановки столбцов, что меняет местами не только столбцы, но и строки.
6. Напишите программу, которая использует подстановочный шифр Цезаря для шифрования и расшифровывания сообщений. Как будет выглядеть текст Nothing but gibberish, если его зашифровать со сдвигом 13?

7\*. Напишите программу, что выводит частоту, с которой буквы встречаются в сообщении. Отсортируйте результат по количеству вхождений и выведи- те для каждой буквы сдвиг относительно *E*.

1. Затем используйте написанный вами код и программу, созданную в упраж- нении 6, для расшифровки сообщения *KYVIV NRJRK ZDVNY VETRV JRIJL SJKZK LKZFE NRJKY VJKRK VFWKY VRIK*. Какой сдвиг применялся при шифровании?
2. Напишите программу, которая использует шифр Виженера для шифрования и расшифровывания сообщений. Расшифруйте с ее помощью криптограм- му *VDOKR RVVZK OTUII MNUUV RGFQK TOGNX VHOPG RPEVW VZYYO WKMOC ZMBR*, применив ключ *VIGENERE*.
3. Почему нельзя начать заново, когда заканчиваются все буквы в одноразовом блокноте?
4. Представьте, что вы обмениваетесь сообщениями с другим человеком с по- мощью большого одноразового блокнота и одно из полученных вами сооб- щений после расшифровывания оказалось бессмысленным набором симво- лов. С чем это может быть связано и как следует поступить в таком случае?
5. Представьте, что при использовании одноразового блокнота вы отправляете сообщения вместе с индексом первой буквы, с помощью которого они были зашифрованы. Поставит ли это под угрозу ваш шифр?
6. Напишите программу, которая использует псевдослучайный одноразовый блокнот. При запуске она должна генерировать блокнот с помощью генера- тора псевдослучайных чисел (не путать с генератором случайных чисел). Программа должна следить за тем, какие символы используются при шиф- ровании и расшифровывании сообщений.
7. Объясните, почему генератор случайных чисел, безопасный с точки зрения криптографии, является эквивалентом метода шифрования, который невоз- можно взломать. Другими словами, как бы вы использовали такой генера- тор для создания абсолютно надежной системы шифрования (и наоборот)?
8. Длина и время передачи сообщений иногда предоставляют злоумышленнику дополнительную информацию. Например, он может заметить, что вы всег- да отправляете длинное сообщение перед важным событием, таким как ви- зит высокопоставленного лица или покупка крупного объема акций. Каким образом реально скрыть подобные сведения?
9. Представьте, что вы используете шифр RSA со значениями *p* = 107, *q* = 211 и *e* = 4,199. Чему в этом случае будут равны *n*, (*n*) и *d*? Как будут выглядеть после шифрования значения 1,337 и 19,905? (При желании воспользуйтесь программой ExponentiateMod из главы 2. Вы также можете написать про- грамму, которая находит обратные величины для модулей, или обратиться к MultiplicativeInverse, включенной в архив загрузок к этой главе.)

# Глава 17

**ТЕОРИЯ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ**

При решении задач производительность алгоритма всегда имеет большое зна- чение. Он не будет достаточно эффективным, если работает очень долго или тре- бует слишком много памяти либо других ресурсов компьютера.

*Теория вычислительной сложности* — тесно связанный с этим раздел информа- тики, изучающий сложность вычислительных задач. Данная теория ориентирова- на не на конкретные алгоритмы, а на проблемы, которые решаются с их помощью.

Например, алгоритм сортировки слиянием, описанный в главе 6, способен от- сортировать список из *N* чисел за время *O*(*N*  log (*N*)). Теория вычислительной сложности рассматривает процесс сортировки в целом, а не какой-то определен- ный алгоритм. Оказывается, *любой* алгоритм, который сортирует с помощью срав- нения, в худшем случае потратит как минимум *N*  log (*N*) времени.

**СОРТИРОВКА ЗА ВРЕМЯ N**  **LOG (N)**

Чтобы понять, почему любой алгоритм, который использует сравнение для сортировки списка, в худшем случае тратит как минимум *N*  log(*N*) време- ни, представьте, что у вас есть массив из *N* редких элементов. Ввиду своей уникальности они могут быть упорядочены *N*! разными способами. Если по- смотреть на это с другой стороны, алгоритму может понадобиться перебрать *N*! вариантов, чтобы расставить элементы в правильном порядке. Получается, что он должен быть пройти по всем *N*! путям выполнения, чтобы выдать мак- симально возможное количество результатов.

Единственный способ, который можно использовать для перехода по разным путям выполнения, — это сравнение двух значений. Следовательно, потенци- альные пути выполнения можно представить в виде бинарного дерева, в ко- тором каждый узел и лист иллюстрируют, соответственно, операцию сравне- ния и итоговую расстановку элементов в массиве.

Существует *N*! способов расставить элементы в массиве, поэтому дерево выполнения должно иметь *N*! листовых узлов. Поскольку это бинарное дере- во, его высота равна log2(*N*!). Таким образом, log2(*N*!) = log2(*N*) + log2(*N* – 1) +

+ log2(*N* – 2) + ... + log2(2). Половина из этих условий (то есть *N*  2) равна как

минимум log2(*N*  2), поэтому log2(*N*!)  *N*  2  log2(*N*  2). Указанное значение имеет порядок *N*  log (*N*).

Теория вычислительной сложности — обширная и сложная тема, для рас- крытия которой здесь просто не хватит места. Тем не менее любой программист, изучающий алгоритмы, должен иметь о ней хотя бы общее представление, что

в первую очередь касается классов *P* и *NP*. В этой главе вы познакомитесь с теорией вычислительной сложности и узнаете, что представляют эти важные виды проблем.

## Обозначения

Одна из первых тем, рассмотренных в этой книге, была посвящена обозначе- ниям «O» большое и «o» малое. В первой главе «O» большое описывалось интуи- тивно понятным образом, как улучшение худшего варианта производительности при росте масштаба проблемы.

В ряде случаев такое определение является достаточно хорошим и полезным, но в теории вычислительной сложности оно приобретает более формальный ха- рактер. Если время работы алгоритма равно *f*(*N*), тогда производительность «O» большого выражается как *g*(*N*) *if f*(*N*) < *g*(*N*)  *k*, где *k* — некая константа, а *N* — до- статочно большое число. Другими словами, функция *g*(*N*) является верхним пре- делом функции времени работы *f*(*N*).

При обсуждении сложности алгоритмов могут пригодиться еще два обозначе- ния, похожих на «O» большое. *«Омега» большое* записывается как (*g*(*N*)) и озна- чает, что функция времени работы ограничена снизу функцией *g*(*N*). Например, как уже было сказано выше, *N*  log(*N*) является нижней границей для алгоритмов сортировки, основанных на сравнении, поэтому такие алгоритмы можно предста- вить как (*N*  log (*N*)).

*«Тета» большое* записывается как (*g*(*N*)) и означает, что функция времени ра- боты ограничена функцией *g*(*N*) как снизу, так и сверху. Например, время, за кото- рое алгоритм выполнит сортировку слияния, не может превышать *O*(*N*  log(*N*)), а время работы любой сортировки, основанной на сравнении, ограничено снизу ве- личиной (*N*  log (*N*)); таким образом, производительность сортировки слияни- ем равна (*N*  log (*N*)).

Итак, «O» большое обозначает верхнюю границу, «Омега» большое () — ниж- нюю, а «Тета» большое () — обе.

Стоит отметить, что в некоторых алгоритмах верхняя и нижняя границы мо- гут быть другими. Например, быстрая сортировка, как и любой аналогичный алгоритм, основанный на сравнении, имеет в качестве нижней границы (*N*  log(*N*)). В лучшем и предполагаемом случаях производительность быстрой сор- тировки будет равна (*N*  log(*N*)). Однако в худшем случае она опускается до *O*(*N*2). У данного алгоритма другие границы, поэтому нет такой функции, кото- рая могла бы описать его в виде «Тета» большого. Хотя в реальности указанный алгоритм часто показывает лучшую скорость, чем методики наподобие сортиров- ки слиянием, строго ограниченные функцией (*N*  log(*N*)), поэтому он все еще является популярным.

## Классы сложности

Алгоритмические задачи иногда разбивают на классы алгоритмов по време- ни выполнения или требованиям к ресурсам на каком-то определенном либо аб- страктном компьютере.

Двумя наиболее распространенными видами абстрактных вычислительных ма- шин являются детерминированные и недетерминированные автоматы.

Действия детерминированного компьютера полностью определяются конеч- ным набором внутренних состояний (переменными программы и ее кодом) и вво- дом. Другими словами, если передать такому компьютеру конкретный набор дан- ных, результат будет полностью предсказуемым (говоря более формальным язы- ком, «компьютером» в этом определении является *машина Тьюринга*).

**МАШИНЫ ТЬЮРИНГА**

Понятие машины Тьюринга было придумано Аланом Тьюрингом в 1936 г. (хотя сам он называл ее «а-машиной»). Идея состояла в создании абстрактного ав- томата настолько простого, чтобы теоремы о том, что он способен или не спо- собен вычислить, можно было бы доказать.

Машина Тьюринга — простой конечный автомат, основанный на наборе вну- тренних состояний, которые определяют его реакцию на конкретный ввод. Это очень похоже на автоматы ДКА и НКА, описанные в главе 15. Разница заклю- чается в том, что ввод машины Тьюринга состоит из нулей и единиц на полу- бесконечной ленте, доступной для чтения и записи. Когда машина считывает с ленты 0 или 1, ее состояние определяет следующее:

* + должна ли она записать 0 или 1 в текущую позицию на ленте;
  + должна ли двигаться считывающая/записывающая головка вдоль ленты и если должна, то в каком направлении;
  + новое состояние, в которое она должна войти.

Несмотря на свою простоту, машина Тьюринга позволяет достаточно неплохо моделировать настоящие компьютеры, хотя создание такой машины для си- муляции сложной программы может оказаться нетривиальной задачей.

Существует несколько вариантов машины Тьюринга. Некоторые из них ис- пользуют ленту без начала и конца. В других применяется несколько лент и считывающих/записывающих головок. Встречаются и недетерминированные машины, способные находиться сразу в нескольких состояниях одновремен- но. Бывают и такие, которые допускают нулевые переходы, что позволяет им менять состояния без чтения каких-либо данных.

Один из интересных выводов, который можно сделать при изучении машин Тьюринга, заключается в том, что все их разновидности имеют одинаковую степень, другими словами, могут выполнять одни и те же вычисления.

Больше информации о машинах Тьюринга можно найти по адресу https:// ru.wikipedia.org/wiki/Ìàøèíà\_Òüþðèíãà.

Для сравнения: недетерминированный компьютер может находиться в не- скольких состояниях одновременно. Это похоже на принцип работы автоматов НКА, описанных в главе 15. Поскольку недетерминированная машина способна использовать для перехода к принимающему состоянию любое количество пу- тей, ей достаточно принять ввод во всех состояниях, в которых она может нахо- диться, и установить тот, что является рабочим путем выполнения. В сущности (если опустить некоторые подробности), это означает, что такая машина способ- на угадать правильное решение и затем удостовериться в том, что оно действи- тельно правильное.

Заметьте, что недетерминированному компьютеру не нужно удостоверяться в отсутствии результата. Если решение есть, он может его угадать и проверить. Если решения нет, компьютеру не обязательно это доказывать.

Например, чтобы определить простые множители для целого числа, детерми- нированному компьютеру пришлось бы каким-то образом найти все его множите- ли — возможно, перебрав каждый из них вплоть до квадратного корня числа или воспользовавшись решетом Эратосфена (больше информации об упомянутых ме- тодиках можно найти в главе 2). Это заняло бы очень много времени.

Недетерминированные компьютеры ведут себя иначе. Они могут разложить числа на простые множители и проверить, что результат получился корректным, выполнив произведение данных множителей и сравнив его с исходным числом. Это делается очень быстро.

Усвоив значение понятий «детерминированный» и «недетерминированный» в данном контексте, вы сможете относительно легко разобраться с большинством классов сложности. Ниже перечислены наиболее важные из этих классов, которые являются детерминированными.

* + DTIME(*f*(*N*)) — задачи, которые детерминированный компьютер может ре- шить за время *f*(*N*). Время работы алгоритмов для решения таких задач со- ставляет *O*(*f*(*N*)), где *f*(*N*) — некая функция. Например, DTIME(*N*  log (*N*)) включает в себя класс задач, которые можно решить за время *O*(*N*  log (*N*)), например сортировку путем сравнения.
  + *P* — задачи, с которыми детерминированный компьютер способен справить- ся за полиномиальное время. Время работы алгоритмов для решения таких задач составляет O(*NP*), причем степень *P* может быть сколь угодно боль- шой, даже *O*(*N*1000).
  + EXPTIME (EXP) — задачи, которые детерминированный компьютер может решить за экспоненциальное время. Время работы алгоритмов для решения таких задач составляет *O*(2*f*(*N*)), где *f*(*N*) — некая полиномиальная функция.

В следующем списке приводятся наиболее важные недетерминированные клас- сы сложности.

* + NTIME(*f*(*N*)) — задачи, которые недетерминированный компьютер может решить за время *f*(*N*). Время работы алгоритмов для решения таких задач составляет *O*(*f*(*N*)), где *f*(*N*) — некая функция. Например, NTIME(*N*2) вклю- чает задачи, с которыми алгоритм может справиться, если будет угадывать проверку ответа за время *O*(*N*2).
  + NP — задачи, которые недетерминированный компьютер способен решить за полиномиальное время. На поиск решения и проверку его корректности уходит время объемом *O*(*NP*), где *P* — некая степень.
  + NEXPTIME (NEXP) — задачи, которые недетерминированный компьютер может решить за экспоненциальное время. На поиск решения и проверку его корректности уходит время объемом *O*(2*f*(*N*)), где *f*(*N*) — некая поли- номиальная функция.

Аналогичным образом задачи можно разделить на классы в зависимости от объ- ема свободного пространства, которое требуется для их решения. Они имеют до- вольно предсказуемые названия: DSPACE(*f*(*N*)), PSPACE (полиномиальное про-

странство), EXPSPACE (экспоненциальное пространство), NPSPACE (недетерми- нированное полиномиальное пространство) и NEXPSPACE (недетерминирован- ное экспоненциальное пространство).

Отношения между некоторыми из этих классов расцениваются как довольно тривиальные. Например, *P* � *NP* (символ � означает «является подмножеством», поэтому данное выражение можно прочитать как «*P* является подмножеством *NP*»). Чтобы понять, почему это действительно так, представьте, что задача имеет класс *P*. Следовательно, существует детерминированный алгоритм, способный найти ее решение за полиномиальное время. В данном случае вы можете решить эту зада- чу и с помощью недетерминированного компьютера, используя тот же алгоритм. Если алгоритм срабатывает (то есть если найденное им решение корректно), это очевидно доказывает корректность результата, поэтому недетерминированный алгоритм тоже подходит.

Бывают и не такие очевидные отношения. Например, PSPACE = NSPACE и EXPSPACE = NEXSPACE.

Наиболее глубокий вопрос в теории сложности вычислений звучит так: явля- ется ли *P* равным *NP*? Некоторые задачи, допустим, сортировка, определенно на- ходятся во множестве *P*. Другие же (например, задачи о ранце и коммивояжере, описанные в главе 12) совершенно точно имеют класс *NP*. Вопрос в том, входят ли проблемы класса *NP* во множество *P*?

Многие люди на протяжении долгого времени пытались определить, являют- ся ли два указанных множества одинаковыми. Никому еще не удалось изобрести детерминированный алгоритм, который бы решал задачи о ранце или коммивоя- жере за полиномиальное время, однако это не означает, что такого алгоритма не может быть в принципе.

Среди способов сравнения сложности двух алгоритмов можно выделить та- кой: сведение одного из них к другому. Об этом пойдет речь в следующем разделе.

## Сведение

Чтобы свести одну задачу к другой, нужно придумать такое решение первой проблемы, которое позволяет решить вторую. Если это можно сделать за опреде- ленный промежуток времени, максимальное время работы двух алгоритмов будет, как и при сведении.

Например, вам уже известно, что разложение на простые множители имеет сложность *NP*, а сортировка принадлежит к классу задач *P*. Представьте, что у вас есть алгоритм, который сводит разложение к сортировке, и на это уходит всего лишь полиномиальное время. В таком случае вы могли бы выполнить разложение на множители за то же полиномиальное время, решив задачу сортировки. Если бы мы знали, как свести операцию разложения на множители к сортировке, она не была бы настолько сложной. Но это еще никому не удалось.

Сведение за полиномиальное время особенно важно, поскольку оно позволя- ет превращать одни задачи класса *NP* в другие. На самом деле существуют такие задачи, к которым можно свести любую проблему в рамках *NP*. Их называют *NP- полными*.

Первой известной *NP*-полной проблемой была задача выполнимости (сокра- щенно *SAT*). В ней вам дается булево выражение с переменными, которые могут быть равны TRUE или FALSE, например (A AND B) OR (B AND NOT C). Необхо- димо определить, можно ли присвоить этим переменным такие значения, которые бы сводили все выражение к результату TRUE.

Теорема Кука—Левина (или просто теорема Кука) доказывает, что задача *SAT* является *NP*-полной. Она по большей части состоит из теоретических нюансов (см. https://ru.wikipedia.org/wiki/Òåîðåìà\_Êóêà\_—\_Ëåâèíà), но ее основная идея понятна.

Чтобы продемонстрировать принадлежность *SAT* к *NP*-полным задачам, необ- ходимо сделать две вещи: показать, что она входит в класс *NP*, и доказать, что к ней можно свести любую другую задачу этого класса.

Задача выполнимости принадлежит к *NP*, поскольку вы можете подобрать зна- чения переменных и затем проверить, делают ли они выражение истинным.

Доказать, что любую другую задачу класса *NP* можно свести к *SAT*, оказыва- ется куда сложнее. Допустим, проблема входит в *NP*. В этом случае у нас должна быть возможность создать недетерминированную машину Тьюринга с внутренни- ми состояниями, которые позволяют ей решить нашу задачу. Доказательство осно- вывается на построении такого булевого выражения, которое будет сигнализиро- вать о передаче ввода машине Тьюринга, корректном срабатывании состояний и об остановке машины в принимающем режиме.

Булево выражение содержит три вида переменных, обозначенных как Tijk,

Hik и Qqk, где *i*, *j*, *k* и *q* могут принимать разные значения. Их описание приводит-

ся в следующем списке.

* + Переменная Tijk равна true, если во время вычисления на шаге *k* ячейка ленты *i* содержит символ *j*.
  + Переменная Hik равна true, если во время вычисления на шаге *k* считыва- ющая/записывающая головка находится над ячейкой ленты *i*.
  + Переменная Qqk равна true, если во время вычисления на шаге *k* машина на- ходится в состоянии *q*.

Выражение также должно содержать некоторые условия, описывающие прин- цип работы машины Тьюринга. Представим, например, что на ленте могут хра- ниться только нули и единицы. В этом случае выражение (T001 AND NOT T011) OR

(NOT T001 AND T011) будет означать, что при вычислении на шаге 1 ячейка 0 содер-

жит либо 0, либо 1 (но не оба значения сразу).

Благодаря другим частям выражения гарантируется, что на любом этапе вы- числения считывающая/записывающая головка находится в какой-то одной пози- ции, что машина начинает свою работу в состоянии 0, что исходной позицией го- ловки является ячейка 0 и т. д.

Полное булево выражение является эквивалентом оригинальной машины Тью- ринга для задачи класса *NP*. Другими словами, если присвоить переменным Tijk зна- чения, представляющие наборы входящих данных, истинность или ложность выраже-

ния будет свидетельствовать о том, принимает ли машина Тьюринга указанные данные.

Это сводит изначальную проблему к задаче определения выполнимости буле- вого выражения, потому *SAT* является *NP*-полной.

Найдя одну *NP*-полную задачу (такую как *SAT*), вы можете свести к ней дру- гую проблему и доказать тем самым, что она тоже является *NP*-полной.

Если задачу *A* можно свести к задаче *B* за полиномиальное время, выразить ее стоит как *A* *p B*.

Примеры сведения одних задач к другим приводятся в следующих разделах.

#### 3SAT

*3SAT* — это задача выполнимости булевых выражений в *3-конъюнктивной нор- мальной форме* (3КНФ). Это означает, что булево выражение состоит из набора опе- рандов, объединенных операторами AND или NOT, где каждый операнд представля- ет собой ровно три переменные, которые соединены операторами OR или NOT. На- пример, все три выражения, приведенные ниже, принадлежат к 3КНФ.

* + (A OR B OR NOT C) AND (C OR NOT A OR B).
  + (A OR C OR C) AND (A OR B OR B).
  + (NOT A OR NOT B OR NOT C).

3*SAT* определенно имеет класс *NP*, поскольку, как и в случае с *SAT*, вы може- те присвоить переменным значения TRUE и FALSE, а затем проверить, является ли выражение истинным.

Приложив определенные усилия, любое булево выражение сводится к 3КНФ за полиномиальное время. Это означает, что *SAT* преобразуется к 3*SAT*. Следователь- но, задача 3*SAT*, как и *SAT*, должна быть *NP*-полной.

#### Паросочетание в двудольном графе

*Двудольным* называют граф, узлы которого разделены на два множества таким образом, что ни одно из его звеньев не соединяет узлы в одном и том же множе- стве (рис. 17.1).



**Рис. 17.1.** В двудольном графе узлы разделены на два множества и звенья могут связывать только узлы из разных множеств

*Паросочетание в двудольном графе* — это множество звеньев, у каждого из ко- торых своя уникальная конечная точка.

Задача о паросочетаниях в двудольном графе звучит так: существует ли такое паросочетание, которое содержит как минимум *k* звеньев (где *k* — некое число)?

В подразделе «Распределение рабочих мест» раздела «Максимальный поток» из главы 14 рассказывалось, как выполнить должностные назначения с помощью задачи максимального потока. Распределение рабочих мест — обычное сопостав- ление служащих с должностями в двудольном графе, стало быть алгоритм, кото- рый для этого используется, подходит и в данном случае.

Добавим исходный узел и соединим его со всеми узлами в одном из множеств. Создадим конечный узел и свяжем его с узлами из другого множества. Теперь ал- горитм максимального потока найдет наибольшее паросочетание в двудольном гра- фе. После этого можно сравнить максимальный поток со значением *k*.

## NP-сложность

Задача является *NP-полной*, если она входит в класс *NP* и к ней можно свести любую другую задачу этого класса за полиномиальное время. *NP-сложная* задача от- личается от *NP*-полной только тем, что она не обязательно должна иметь класс *NP*.

Обратите внимание: все *NP*-полные задачи считаются *NP*-сложными и входят в *NP*.

Если задача является *NP*-сложной, это в некотором роде означает, что она как минимум не уступает в сложности любой задаче класса *NP*, поскольку все такие задачи могут быть сведены к ней.

Для демонстрации того, что проблема определяется как *NP*-полная, можно по- казать, что она преобразуется в *NP*-сложную задачу за полиномиальное время. Ана- логичным образом доказывается и то, что она *NP*-сложная, если свести ее за то же полиномиальное время к *NP*-сложной задаче.

## Задачи обнаружения, сообщения и оптимизации

Многие задачи можно разделить на три вида: обнаружение, сообщение и опти- мизация. В задаче обнаружения проверяется само наличие решения заявленного качества. В задаче сообщения это решение нужно найти, а в задаче оптимизации — улучшить.

Например, задача о сумме подмножеств состоит из трех соответствующих под- задач, относящихся к одному и тому же набору чисел.

* + Обнаружение: существует ли такое подмножество, сумма чисел которого равна *k*?
  + Сообщение: найдите такое подмножество, сумма чисел которого равна *k*

(если такое существует).

* + Оптимизация: найдите такое подмножество, сумма чисел которого находит- ся как можно ближе к значению *k*.

Одна из разновидностей этой задачи требует найти подмножество, сумма зна- чений которого равна 0.

На первый взгляд некоторые из этих задач могут показаться проще, чем осталь- ные. Например, задача обнаружения всего лишь требует доказать, что элементы какого-то подмножества в сумме дают 0. Поскольку находить само подмножество не нужно, как в случае с задачей сообщения, вам может показаться, что процесс обнаружения является более легким. На самом же деле, воспользовавшись сведе- нием, вы можете убедиться в том, что все три вида задач имеют одинаковую слож- ность (по крайней мере с точки зрения теории вычислительной сложности). Для этого необходимо выполнить четыре операции:

* + Обнаружение *p* Сообщение;
  + Сообщение *p* Оптимизация;
  + Сообщение *p* Обнаружение;
  + Оптимизация *p* Сообщение.

Операции сведения являются транзитивными, поэтому первые две из них озна- чают, что Обнаружение *p* Сообщение *p* Оптимизация, а из двух последних следу- ет, что Оптимизация *p* Сообщение *p* Обнаружение.

#### Обнаружение p Сообщение

Сведение вида *Обнаружение* *p* Сообщение является достаточно очевидным. Имея алгоритм для создания сообщений о подмножествах, вы можете использо- вать его для нахождения этих подмножеств. С его помощью можно определить

подмножество, сумма членов которого равна некоему значению *k*. Если поиск дал хотя бы один результат, мы можем ответить на вопрос, который ставится в задаче обнаружения: «Да, такое подмножество существует».

В качестве конкретного примера возьмем ReportSum — алгоритм, который сооб- щает о результатах в задаче о сумме подмножеств (то есть ReportSum(*k*) возвращает подмножество, сумма членов которого равна *k*, если такое подмножество существу- ет). В таком случае DetectSum(*k*) может просто вызывать алгоритм ReportSum(*k*) и возвращать true, если тот отзывает подмножество.

#### Обнаружение p Оптимизация

Сведение вида *Обнаружение* *p* Оптимизация тоже выглядит довольно очевид- ным. Представим, что у нас есть алгоритм, занимающийся поиском оптимального решения. Если возвращаемый им результат находится в рамках значения *k*, которое

было получено при решении задачи сообщения, тогда задача сообщения может отозвать результат работы оптимизационного алгоритма. Если это не так, нужно вернуть false.

В качестве конкретного примера представим, что OptimizeSum(*k*) возвращает подмножество, сумма членов которого должна быть как можно ближе к значению *k*. Тогда ReportSum(*k*) может задействовать алгоритм OptimizeSum(*k*) и проверить, отзывает ли тот нужный результат. Если ответ положительный, ReportSum(*k*) воз- вращает соответствующее подмножество. В противном случае он не возвращает результат, сигнализируя о том, что такого подмножества не существует.

#### Сообщение p Обнаружение

Сведение вида *Сообщение* *p* Обнаружение выглядит менее очевидным по срав- нению с предыдущими случаями. Для начала мы воспользуемся алгоритмом обна- ружения, чтобы проверить, существует ли решение. Если решения нет, алгоритму

сообщения можно ничего не делать.

Если решение возможно, упростим задачу: чтобы убедиться в том, что реше- ние по-прежнему существует, воспользуемся алгоритмом обнаружения. Если ре- шения нет, откажемся от текущего упрощения и попробуем другое. Облегчив за- дачу всеми возможными способами и не найдя ни одного подходящего, мы можем сказать, что оставшийся вариант является решением, и именно его должен вернуть алгоритм сообщения.

В качестве конкретного примера представим, что при обнаружении подмноже- ства, члены которого в сумме дают *k*, DetectSum(*k*) возвращает true. В следующем псевдокоде показано, как на основе DetectSum построить алгоритм ReportSum.

1. Применяем DetectSum(*k*) ко всему множеству, проверяя, существует ли решение. Если решения нет, алгоритм ReportSum сигнализирует об этом и завершает свою работу.
2. Для каждого значения *Vi* внутри множества:
   1. удаляем *Vi* из множества. Вызываем DetectSum(*k*) для оставшихся эле- ментов и проверяем, существует ли подмножество, сумма членов которо- го равна *k*;
   2. если DetectSum(*k*) возвращает FALSE, вставляем отозванное *Vi* во мно- жество и возвращаемся в начало цикла (шаг 2);
   3. если DetectSum(*k*) возвращает TRUE, оставляем *Vi* вне множества и воз- вращаемся в начало цикла (шаг 2).

По окончании цикла, инициированного на шаге 2, оставшиеся внутри множе- ства значения будут составлять подмножество, которое в сумме дает *k*.

#### Оптимизация p Сообщение

На последнем этапе, демонстрирующем, что все три вида задач имеют одинако- вый уровень сложности, нужно доказать утверждение Оптимизация *p* Сообщение. Допустим, у нас есть алгоритм сообщения Report(*k*). Тогда оптимизационный ал- горитм может вызывать Report(*k*), Report(*k* + 1), Report(*k* + 2) и т. д., пока ре- шение не будет найдено (можно также попробовать решения вида Report(*k* – 1), если задача это позволяет).

В качестве конкретного примера представим, что ReportSum(*k*) возвращает подмножество членов, сумма которых равна *k* (если такое существует). Тогда ал- горитм OptimizeSum(*k*) можно описать в виде следующих шагов.

1. For *i* = 0 To *N*, где *N* — количество элементов внутри множества:
   1. если ReportSum(*k* + *i*) возвращает подмножество, тот же результат дол- жен вернуть и алгоритм OptimizeSum;
   2. если ReportSum(*k* – *i*) возвращает подмножество, тот же результат дол- жен вернуть и алгоритм OptimizeSum;
   3. возвращаемся к шагу 1.

Данные операции сведения показывают, что Обнаружение *p* Сообщение *p* Оптимизация и Оптимизация *p* Сообщение *p* Обнаружение. Следовательно, все эти задачи имеют одинаковую сложность.

## NP-полные задачи

Всего было найдено более 3000 *NP*-полных задач, поэтому ниже перечислены только некоторые из них. Это сделано для того, чтобы вы имели представление о том, проблемы какого рода являются *NP*-полными.

Как вам уже известно, у *NP*-полных задач нет известного решения, нахожде- ние которого занимает полиномиальное время, поэтому все они считаются край- не сложными. Какие-то задачи можно решить только в небольшом масштабе.

Поскольку все эти задачи являются *NP*-полными, каждую из них всегда мож- но свести к другой.

* + **Задача о картинной галерее.** Зная план комнат и коридоров в картинной галерее, нужно найти минимальное количество охранников, необходимых для присмотра за картинами.
  + **Задача об упаковке в контейнеры.** Имея набор объектов и контейнеров, не- обходимо упаковать эти объекты так, чтобы количество использованных кон- тейнеров было минимальным.
  + **Задача о коммивояжере с устранением узких мест.** Нужно найти такой гамильтонов путь по взвешенной сети, в котором минимизируется наиболее затратное звено.
  + **Задача о китайском почтальоне (о проверке маршрута)**. Необходимо най- ти кратчайший цикл, состоящий из каждого звена сети.
  + **Хроматическое число (или раскраска вершин графа)**. Нужно найти наи- меньшее количество цветов, необходимых для раскраски узлов графа (граф не обязательно должен быть планарным).
  + **Задача о клике.** Нужно найти наибольшую клику в графе. (*Кликой* называет- ся множество, в котором все узлы соединены друг с другом. Другими словами, у каждой пары узлов внутри множества есть ребро, которое их соединяет).
  + **Задача о покрытии кликами.** Нужно разделить граф на *k* множеств, каждое из которых является кликой.
  + **Задача об остовном дереве с ограниченной степенью.** Нужно найти в рам- ках графа остовное дерево со степенью, не больше заданной.
  + **Задача о доминантном множестве.** Нужно найти в рамках графа такое мно- жество *S*, в котором один из узлов является смежным по отношению ко всем остальным.
  + **Задача о множестве вершин, разрезающих циклы.** Нужно найти в рамках графа такое минимальное множество вершин *S*, после удаления которого в графе не осталось бы циклов.
  + **Гамильтоново заключение.** Нужно найти минимальное количество ребер, при добавлении которых граф становится гамильтоновым (таким, что со- держит гамильтонов путь).
  + **Гамильтонов цикл.** Нужно определить, существует ли такой путь, который проходит через все узлы графа ровно один раз и возвращается в начальную точку.
  + **Гамильтонов путь.** Нужно определить, существует ли такой путь, который проходит через все узлы графа ровно один раз.
  + **Задача планирования заданий.** Имея *N* заданий разного объема и *M* иден- тичных машин, нужно распланировать работу таким образом, чтобы мини- мизировать время, за которое машины выполнят все задания.
  + **Задача о ранце.** Нужно найти множество объектов с максимальным общим весом, способных поместиться в ранец заданного объема.
  + **Задача о самом длинном пути.** Нужно найти в сети самый длинный путь, который не проходит через один и тот же узел дважды.
  + **Задача о максимальном независимом множестве.** Нужно найти в рамках графа наибольшее множество, в котором ни один из узлов не связан с другим.
  + **Задача о связном доминирующем множестве.** Необходимо найти остовное дерево с максимальным количеством листьев в графе.
  + **Задача об остовном дереве с минимальной степенью.** Нужно найти в рам- ках графа остовное дерево с минимальной степенью.
  + **Задача о минимальном *k*-разрезе.** Требуется найти множество ребер с ми- нимальным весом, без которых граф окажется разделенным на *k* частей.
  + **Задача разделения.** Имея множество целых чисел, нужно разделить их на два подмножества с одним и тем же суммарным значением (в некоторых раз- новидностях этой задачи используется больше двух подмножеств).
  + **Задача выполнимости (*SAT*)**. Имеется булево выражение с переменными. Нужно подобрать для этих переменных такие значения true и false, что- бы выражение было истинным (больше подробностей можно найти в разде- ле «Сведение» текущей главы).
  + **Задача о кратчайшем пути**. Нужно найти кратчайший путь между двумя за- данными узлами сети (не обязательно планарной).
  + **Задача о сумме подмножеств.** Необходимо найти внутри множества целых чисел подмножество с заданным суммарным значением.
  + **Задача тройного разделения.** Нужно разделить множество целых чисел на три части, каждая из которых имеет одно и то же суммарное значение.
  + **Задача выполнимости булевых выражений в 3-конъюнктивной нормальной форме (3*SAT*)**. Имеется булево выражение в конъюнктивной нормальной

форме. Нужно присвоить его переменным такие значения true или false, чтобы оно оказалось истинным (больше подробностей ищите в разделе

«3SAT» текущей главы).

* + **Задача о коммивояжере.** Имеется список городов и расстояния между ними. Нужно найти кратчайший маршрут, проходящий через все города и возвра- щающийся в начальную точку.
  + **Задача о безразмерном ранце.** То же самое, что и обычная задача о ранце, только любой элемент можно выбрать больше одного раза.
  + **Задача маршрутизации транспортных средств.** Имеется список клиентов, находящихся в определенных местах, и парк автомобилей. Нужно найти наи- более оптимальные маршруты, позволяющие посетить всех клиентов. (У этой задачи есть разновидности. Например, маршрут может предусматривать до- ставку с возвратом или без; иногда товар необходимо доставлять в поряд- ке «последним погрузил — следующим доставил»; автомобили могут иметь разную вместимость и т. д.)
  + **Задача о вершинном покрытии.** Нужно найти такое минимальное множе- ство вершин, чтобы каждое ребро графа проходило хотя бы через одну из них.

## Резюме

Эта глава предоставляет краткое введение в теорию сложности вычислений. В ней приведено объяснение классов сложности и дано описание наиболее важ- ных из них, включая *P* и *NP*. Вам не обязательно знать все тонкости каждого тако- го класса, но в понятиях *P* и *NP* вы определенно должны ориентироваться. Также не помешает познакомиться, наверное, с самым глубоким вопросом в сфере ком- пьютерных наук: «Является ли *P* равным *NP*?»

За этим следовали разделы, которые были посвящены операциям сведения, выполняемым за полиномиальное время, с их помощью реально показать, что одна задача является не менее сложной, чем другая. Сведения подобного рода потребуются при изучении теории сложности вычислений, но их также мож- но применять и в более общем случае — для использования имеющегося реше- ния в новых задачах. В этой главе не описывается никаких определенных ал- горитмов, которые можно было бы реализовать на компьютере, однако опера- ции сведения демонстрируют, как посредством одного и того же алгоритма ре- шить две разные проблемы.

Задачи, описанные в текущей главе, позволят распознать самую сложную проб- лему и дать вам понимание того, что у нее может просто не быть идеального реше- ния. Сталкиваясь с разновидностями задач о гамильтоновом пути, коммивояжере или ранце, вы не будете удивлены, что решить их программным способом можно только в очень небольшом масштабе.

Методики, которые могут пригодиться при решении некоторых из этих край- не сложных проблем, приводятся в главе 12. С помощью ветвления и ограничения получится добиваться результатов даже там, где нельзя применять метод просто- го перебора. Эвристика позволяет находить приблизительные решения и в случа- ях даже для более масштабных задач.

Еще одним подходом, помогающим справляться со сложными проблемами, является параллелизм. Разделяя работу между несколькими центральными про- цессорами или компьютерами, вы можете решать задачи, которые не по силам одной вычислительной машине. Некоторые алгоритмы, являющиеся эффективными благодаря этой методике, описаны в следующей главе.

#### Упражнения

Звездочкой отмечены задачи повышенной сложности.

1. Если любому алгоритму, который упорядочивает элементы с помощью срав- нения, требуется для выполнения работы максимум *O*(*N*  log (*N*)) времени, почему алгоритмы сортировки подсчетом и блочной сортировки, описанные в главе 6, работают быстрее?
2. Существует задача, в которой нужно определить, является ли граф двудоль- ным. Сведите ее к задаче закраски карты за полиномиальное время. Как бы вы охарактеризовали класс сложности, к которому принадлежит задача опре- деления двудольного графа?
3. Задача о тройных циклах звучит так: содержит ли назначенный граф хотя бы один цикл длиною три узла? Сведите ее к другой задаче за полиномиаль- ное время. Как бы вы охарактеризовали класс сложности, к которому при- надлежит задача о тройных циклах?
4. Задача о цикле нечетной длины звучит так: содержит ли граф циклы с не- четным количеством вершин? Сведите ее к другой задаче за полиномиальное время. Как бы вы охарактеризовали класс сложности, к которому она при- надлежит? Как она соотносится с задачей о тройных циклах?
5. Задача о гамильтоновом пути (ЗГП) звучит так: существует ли в сети путь, который проходит через все ее узлы ровно один раз? Покажите, что эта за- дача принадлежит к классу *NP*.
6. Задача о гамильтоновом цикле (ЗГЦ) звучит так: существует ли в сети путь, который проходит через все ее узлы ровно один раз и возвращается в началь- ную точку? Покажите, что эта задача принадлежит к классу *NP*.

7\*\*. Сведите ЗГП к ЗГЦ за полиномиальное время. 8\*\*. Сведите ЗГЦ к ЗГП за полиномиальное время.

1. Задача закраски сети звучит так: возможно ли закрасить узлы сети *k* цвета- ми, где *k* — некое число? Покажите, что эта задача принадлежит к классу *NP*.
2. Задача о нулевой сумме подмножеств звучит так: существует ли внутри мно- жества чисел такое подмножество, сумма членов которого равна нулю? По- кажите, что эта задача принадлежит к классу *NP*.

11\*. Представьте, что у вас есть множество объектов с весом *Wi* и значениями *Vi*, а также ранец, способный выдержать вес *W*. На основе этого можно сформу- лировать три разновидности задачи о ранце.

* + Обнаружение. Существует ли подмножество объектов, которые влезают в ранец и имеют суммарное значение не меньше *k*?
  + Сообщение. Найдите подмножество объектов, которые влезают в ранец и имеют суммарное значение не меньше *k* (если такое существует).
  + Оптимизация. Найдите подмножество объектов, которые влезают в ранец и имеют как можно большее суммарное значение.

1. Сведите задачу сообщения к обнаружению.

13\*. Сведите задачу оптимизации к обнаружению (см. упражнение 11).

14\*\*. Представьте, что у вас есть множество объектов со значениями *Vi*. На осно- ве этого можно сформулировать два вида задачи разделения.

* + Обнаружение. Возможно ли разделить объекты на два подмножества *A*

и *B*, имеющие одно и то же суммарное значение?

* + Сообщение. Найдите способ разделения объектов на два подмножества *A*

и *B* так, чтобы они имели одно и то же суммарное значение.

15. Сведите задачу сообщения к обнаружению.

# Глава 18 РАСПРЕДЕЛЕННЫЕ АЛГОРИТМЫ

В статье, опубликованной в 1965 г., Гордон Мур отметил, что с момента изобре- тения микропроцессоров (1958) количество транзисторов в их кристаллах удваи- валось каждые два года. Исходя из данного наблюдения он предположил, что та- кая тенденция сохранится на протяжении как минимум следующих 10 лет. Пред- сказание, которое ныне известно как *закон Мура*, оказалось на удивление точным и не теряло своей актуальности на протяжении последних 50 лет (хотя уже видно, что этому скоро придет конец).

Размер объектов, которые можно разместить внутри микросхемы, постепенно приближается к пределу текущих технических возможностей. Даже если миниа- тюризация транзисторов продолжится (это вполне возможно, ведь производите- ли электроники известны своей смекалкой), на каком-то этапе все равно будет до- стигнут квантовый уровень, где законы физики приобретают непредсказуемый ха- рактер и делают современные технологии бессильными. Некоторые из эффектов, проявляющихся на данном уровне, могут быть использованы в квантовых вычис- лениях для создания новых потрясающих компьютеров, однако, по всей видимо- сти, закон Мура не вечен.

Самый эффективный способ повышения вычислительной мощности без уве- личения количества транзисторов на кристалле заключается в применении сразу нескольких процессоров. Большинство компьютеров, которые продаются в наши дни, содержат свыше одного центрального процессорного устройства (ЦПУ). Ча- сто они имеют несколько ядер, представляя собой многоядерный процессор на об- щей микросхеме. Умные операционные системы могут пользоваться преимущества- ми дополнительных ядер, а качественный компилятор способен распознать участ- ки программы, которые реально выполнять параллельно на нескольких ядрах. Но чтобы извлечь максимальную выгоду от многопроцессорных систем, вам необхо- димо понимать, как пишутся параллельные алгоритмы.

В этой главе рассмотрены проблемы, возникающие при использовании не- скольких процессоров для решения одной задачи. В ней описываются разные мо- дели параллельной обработки и объясняется принцип работы отдельных алгорит- мов и методик, которые позволяют ускорить решение проблем параллелизации.

## Виды параллелизма

Существует несколько моделей параллелизма, и каждая из них основана на от- дельном наборе допущений, в первую очередь количестве доступных процессоров и том, как они между собой соединены. Самой распространенной моделью сегод- ня являются *распределенные вычисления*, но и другие виды параллельной обработ-

ки представляют определенный интерес; некоторые из них, начиная с систоличе- ских массивов, будут так или иначе затронуты в этой главе. Вам вряд ли удастся применить на практике большой систолический массив, но понимание принципа его работы даст некоторое представление о других алгоритмах, которыми вы смо- жете воспользоваться в распределенных системах.

#### Систолические массивы

*Систолическим* называют массив модулей обработки данных (МОД или просто

*ячеек*). Он может быть одно-, двух- или даже многомерным.

Все смежные ячейки в массиве связаны друг с другом — прямой связи между остальными ячейками быть не может.

Все ячейки синхронно реализуют одну и ту же программу. Такая модель на- зывается *параллелизмом данных*, поскольку процессоры выполняют один код, но с разными наборами информации (выбор термина «систолический массив» связан с тем, что данные проходят через процессоры за равные промежутки времени — по аналогии с бьющимся сердцем, прокачивающим кровь по телу).

Систолические массивы могут быть крайне эффективными, однако область их применения очень узкая, к тому же их построение стоит недешево. Алгоритмы, ко- торые для них написаны, часто отталкиваются от того, что количество ячеек мас- сива зависит от количества входящих данных. Например, алгоритм умножения ма- триц размерностью *N**N* подразумевает, что в его наличии есть массив ячеек того же размера. Это предположение ограничивает масштаб доступных для решения за- дач объемом массива, который реально построить.

Вам может никогда не представиться случая поработать с систолическими мас- сивами, однако их алгоритмы достаточно любопытны, и чтобы вы понимали прин- цип их работы, мы рассмотрим один из них в данном разделе.

Предположим, у вас есть последовательность из *N* чисел, которые нужно отсо- ртировать в одномерном систолическом массиве, содержащем *N* ячеек. Ниже пред- ставлено пошаговое описание того, как каждая из ячеек обрабатывает свои данные.

1. Чтобы ввести первую половину последовательности, повторяем *N* раз:
   1. каждая ячейка должна передать свое текущее значение вправо;
   2. если это нечетный шаг, помещаем новое число в первую ячейку. Если шаг четный, не делаем этого.
2. Чтобы ввести вторую половину последовательности, повторяем *N* раз:
   1. если ячейка содержит два значения, она должна их сравнить и передать меньшее из них влево, а большее — вправо;
   2. если первая ячейка содержит одно значение, она должна передать его вправо;
   3. если последняя ячейка содержит одно значение, она должна передать его влево;
   4. если это нечетный шаг, помещаем новое число в первую ячейку. Если шаг четный — не делаем этого.
3. Чтобы вывести отсортированный список, повторяем *N* раз:
   1. если ячейка содержит два значения, она должна их сравнить и передать меньшее из них влево, а большее — вправо;
   2. если ячейка содержит одно значение, она должна передать его влево.

Алгоритм, показанный на рисунке 18.1, сортирует значения 3, 4, 1 и 2 в массиве с четырьмя ячейками. В первой строке ячейки изображены пустыми, а числа для сортировки представлены слева.

3, 4, 1, 2

**Такт 4** 3, 4

1

2

Такт 5 3

4

1

2

Такт 6 3

4

1

2

Такт 7

3

1

4

2

Такт 8 1

3

2

4

Такт 9 1

2

3

4

**Такт 10** 1, 2

4

3

**Такт 11** 1, 2

3

4

**Такт 12** 1, 2, 3

4

4

**Такт 13** 1, 2, 3

4

**Такт 14** 1, 2, 3, 4

**Рис. 18.1.** Систолический массив с четырьмя ячейками может отсортировать четыре числа за 14 тактов

Четыре начальных систолических такта (на рисунке они обозначены как так- ты, чтобы вы не перепутали их с шагами алгоритма) помещают в массив первые два значения (2 и 1). Все это происходит на шаге 1 алгоритма.

Самое интересное начинается на такте под номером 5, когда алгоритм перехо- дит к шагу 2. На данном такте значение 4 помещается в первую ячейку. В этот мо- мент третья ячейка содержит значения 1 и 2; она их сравнивает и перемещает мень- шее (1) влево, а большее (2) — вправо.

На такте 6 вторая ячейка сравнивает значения 4 и 1. Первое отправляется вле- во, а второе — вправо. Вместе с этим алгоритм помещает последнее значение, 3, в первую ячейку.

На такте 7 первая ячейка сравнивает значения 3 и 1; 3 перемещается вправо, а 1 — влево. В то же время третья ячейка сравнивает 4 и 2; 2 перемещается влево, а 4 — вправо.

На такте 8 вторая ячейка сравнивает 3 и 2; 3 отправляется вправо, а 2 — влево.

Последняя ячейка передает влево значение 4.

На такте 9 начинается шаг 3 алгоритма. Первая ячейка выводит значение 2.

Третья ячейка сравнивает 3 и 4; 3 идет влево, а 4 — вправо.

В тактах с 10 по 14 ячейки содержат не более одного значения. Они перемеща- ют свое содержимое влево и в итоге добавляют их к имеющемуся выводу.

Вам может показаться, что на сортировку всего лишь четырех элементов тра- тится слишком много шагов, но при работе с более крупными списками данный алгоритм способен сэкономить время. Если количество элементов *N*, алгоритму требуется *N* тактов, чтобы добавить в массив половину из них (шаг 1), потом еще *N* тактов, чтобы вставить вторую половину (шаг 2), и завершающие *N* тактов для вывода последних отсортированных значений.

Общее число тактов равно *O*(3  *N*) = *O*(*N*). Это меньше, чем у непараллельных алгоритмов, сортирующих *N* значений с помощью сравнения — *O*(*N*  log (*N*)). По- скольку числа могут распределяться по *N*/2 ячейкам, массив способен выполнять одновременно до (*N*/2)2 сравнений.

Этот алгоритм имеет несколько интересных особенностей. Прежде всего на такте 7 в массив добавляется последнее значение, а на такте 8 выводится первое отсортированное число. Поскольку первое готовое значение выносится сразу по- сле ввода последнего, все выглядит так, как будто на упорядочивание элементов вообще не тратится времени. В связи с этим данный алгоритм называют *мгновен- ной сортировкой*.

Еще одно интересное его свойство заключается в том, что на любом этапе только половина ячеек содержит данные. При желании в свободные ячейки можно поме- стить значения из второй последовательности чисел, благодаря чему массив будет сортировать одновременно две последовательности.

#### Распределенные вычисления

*Распределенные вычисления* — это когда несколько компьютеров работают вме- сте по сети, чтобы выполнить некую задачу. У каждого из них своя оперативная память, хотя жесткие диски могут быть общими.

По сравнению с взаимодействием между ЦПУ в рамках одной вычислитель- ной машины, сети являются относительно медленными, поэтому распределенные

алгоритмы должны пытаться свести к минимуму передачу информации между ком- пьютерами. Обычно вычислительные машины получают данные, работают какое-то время над задачей и отправляют обратно готовое решение.

Существует два вида распределенных сред: *кластеры* и так называемые *гри- ды*. Первые представляют собой набор тесно связанных компьютеров, обычно соединенных с помощью интранета или сети специального назначения, кото- рая имеет ограниченный доступ ко внешним сетям. На практике к кластеру ча- сто относятся как к гигантскому компьютеру с необычными внутренними ком- муникациями.

В грид-вычислениях компьютеры связаны между собой намного слабее. Они могут взаимодействовать через открытую сеть, состоять из разного оборудования и работать под управлением разных операционных систем.

Взаимодействие между компьютерами в грид-среде может быть довольно мед- ленным и ненадежным. Поскольку связь между ними является слабой, любой из них может быть выключен до завершения доверенных ему вычислений. В связи с этим система в случае необходимости должна иметь возможность переназначать подзадачи другим компьютерам.

Несмотря на такие недостатки, как относительно медленное взаимодействие и ненадежность отдельных устройств, грид-вычисления сделали возможным соз- дание виртуального суперкомпьютера, который потенциально способен выделить для решения задачи огромный объем вычислительных ресурсов. Ниже перечисле- ны некоторые публичные грид-проекты.

* + MilkyWay@home

<http://milkyway.cs.rpi.edu/milkyway>

Этот проект занимается созданием очень точной модели галактики Млеч- ного Пути для использования в исследованиях в сфере астроинформатики и компьютерных наук. В нем принимает участие примерно 38 000 компью- теров общей мощностью 1,6 терафлопсов.

* + BOINC (*Berkeley Open Infrastructure for Network Computing* — открытая про- граммная платформа университета Беркли для сетевых вычислений)

[http://boinc.berkeley.edu](http://boinc.berkeley.edu/)

Эта платформа с открытым исходным кодом используется во многих от- дельных проектах, посвященных исследованиям в сфере астрофизики, ма- тематики, медицины, химии, биологии и других областях. В ее состав входит приблизительно 600 000 компьютеров общей мощностью 9,2 петафлопсов.

* + Folding@home

[http://folding.stanford.edu](http://folding.stanford.edu/)

Этот проект, моделирующий сворачивание белка, призван расширить наше понимание таких недугов, как болезнь Альцгеймера, коровье бешенство (ГЭКРС), СПИД, болезнь Хантингтона, болезнь Паркинсона и раковые за- болевания. Количество компьютеров в проекте достигает почти 200 000, а их суммарная мощность равна примерно 12 петафлопсов.

**ФЛОПСЫ**

Скорость компьютеров, которые используются для ресурсоемких математиче- ских вычислений, часто измеряют в операциях с плавающей запятой в секун- ду (*floating-point operations per second* или *flops*). Один терафлопс (тфлоп) равен 1012 (то есть миллиарду) флопсов. Один петафлопс (пфлопс) равен 1015 флопсов, или тысяче терафлопсов. Для сравнения: диапазон мощности обыч- ных настольных компьютеров колеблется от 0,25 до 10 гигафлопсов.

**УЧАСТИЕ В ГРИД-ВЫЧИСЛЕНИЯХ**

Если вас заинтересовали эти проекты, посетите их веб-страницы и загрузите, программное обеспечение, которое позволит делиться процессорными цик- лами вашего компьютера, когда тот простаивает без работы.

Процессы в распределенных компьютерах могут выполнять разные задачи — это пример распараллеливания работы. Сравните это с параллелизмом данных, при котором акцент делается на распределение информации между несколькими процессорами.

#### Многопроцессорные вычисления

Большинство современных компьютеров имеет больше одного процессора. Ино- гда процессоры размещаются на отдельных платах, но чаще всего речь идет о не- скольких ядрах на одном кристалле.

ЦПУ, входящие в состав одного компьютера, могут взаимодействовать между собой намного быстрее, чем отдельные компьютеры в сети, поэтому они не страда- ют некоторыми коммуникационными проблемами, характерными для распределен- ных систем. Например, чтобы производительность не упиралась в скорость сети, распределенные системы должны передавать внутри себя как можно меньше дан- ных. ЦПУ в рамках одного компьютера способны общаться между собой на очень высокой скорости, поэтому они могут обмениваться большими объемами инфор- мации без существенного влияния на производительность.

Кроме того, разные ЦПУ в одном и том же компьютере нередко имеют общие дисковый накопитель и память.

Возможность обмениваться большими объемами данных и иметь доступ к од- ним и тем же дискам и памяти — удобно, но это также может привести к таким проблемам, как состояние гонки или взаимная блокировка. От подобных явле- ний не застрахованы и распределенные системы, однако отмеченные неприятно- сти характерны больше для многопроцессорных компьютеров, поскольку несколь- ким ЦПУ очень легко оказаться в ситуации соперничества за одни и те же ресурсы.

#### Состояние гонки

*Состояние гонки* означает, что два процесса пытаются выполнить запись внутрь ресурса примерно в один и тот же момент времени. Процесс, который записывает вторым, побеждает.

Чтобы понять, как это может случиться, представьте, что два процесса исполь- зуют эвристические методики для решения задачи гамильтонова пути (см. главу 17)

и затем с помощью следующего псевдокода обновляют общие переменные, которые содержат лучший путь, найденный на этот момент, и его общую длину.

// Эвристический анализ.

...

// Сохраняем лучшее решение.

If (test\_length < BestLength) Then

// Сохраняем новое решение.

...

// Сохраняем новую общую длину. BestLength = test\_length

End If

Псевдокод начинается с применения эвристических методик для поиска нуж- ного решения. Затем он сравнивает итоговую длину лучшего найденного маршру- та со значением, находящимся в общей переменной BestLength. Если новое ре- шение лучше предыдущего, псевдокод сохраняет его вместе с его итоговой длиной.

К сожалению, невозможно предсказать, в какой момент несколько процессов могут обратиться к общей памяти. Допустим у нас есть два процесса, которые вы- полняются в порядке, представленном на примере следующего псевдокода.

|  |  |
| --- | --- |
| // Эвристический анализ.  ...  // Сохраняем лучшее решение. If (test\_length < BestLength) Then  // Сохраняем новое решение.  ...  // Сохраняем новую общую длину.  BestLength = test\_length End If | // Эвристический анализ.  ...  // Сохраняем лучшее решение. If (test\_length < BestLength) Then  // Сохраняем новое решение.  ...  // Сохраняем новую общую длину.  BestLength = test\_length End If |

Действия, выполняемые процессом *A* (слева) и процессом *B* (справа), показа- ны в хронологическом порядке.

Сначала процесс *A* проводит эвристический анализ, потом это делает процесс *B*.

Затем процесс *A* выполняет проверку If, чтобы узнать, нашел ли он улучшен- ное решение. Допустим, в нашем примере исходное лучшее решение имеет марш- рут длиной 100, а процесс *A* обнаруживает путь, общая длина которого равна 70 и, следовательно, входит в блок If Then.

Затем проверку If делает процесс *B*. Допустим, он обнаруживает маршрут об- щей длиной 90 и тоже входит в блок If Then.

Процесс *A* сохраняет свое решение.

После этого процесс *B* делает то же самое. Кроме того, он присваивает общей переменной BestLength длину нового маршрута — 90.

Теперь процесс *A* назначает переменной BestLength длину найденного им маршрута — 70.

На данном этапе в качестве лучшего используется решение, полученное про- цессом *B*; из двух найденных вариантов оно является наихудшим. Переменная BestLength хранит значение 70, которое является длиной маршрута, полученно- го процессом *A* (а не того, который на самом деле был сохранен).

Состояние гонки можно предотвратить с помощью *мьютекса* (*mutex*, или *mutual exclusion*, — взаимное исключение). Такая методика позволяет гарантировать, что определенная операция выполняется одновременно только одним процессом. В кон- тексте общих переменных это означает, что считывать и записывать их может лишь один процесс в любой отдельно взятый момент времени.

**РЕАЛИЗАЦИЯ МЬЮТЕКСОВ**

Некоторые компьютеры могут предоставлять аппаратные ресурсы для более эффективной реализации мьютексов, но в остальных случаях это приходится делать в программном режиме.

Псевдокод, показанный ниже, демонстрирует добавление мьютекса в предыду- щий алгоритм, что позволяет избежать состояния гонки.

// Эвристический анализ.

...

// Захватываем мьютекс.

...

// Сохраняем лучшее решение.

If (test\_length < BestLength) Then

// Сохраняем новое решение.

...

// Сохраняем новую общую длину. BestLength = test\_length

End If

// Освобождаем мьютекс.

...

Данная версия кода, как и предыдущая, начинается с эвристического анализа. Это делается без использования общей памяти, поэтому здесь не может возникнуть состояние гонки.

Когда процесс готов обновить общее решение, он сначала захватывает мью- текс. В разных языках программирования это может выглядеть по-разному. На- пример, в языках семейства .NET (таких как C# и Visual Basic) процесс создает

объект Mutex и вызывает из него метод WaitOne, чтобы попробовать завладеть мьютексом.

Если другой процесс попытается захватить мьютекс в этот момент, он будет заблокирован; ему придется подождать, пока мьютекс не будет освобожден первым процессом.

Захватив мьютекс, процесс начинает работу с общей памятью. Поскольку этот захват эксклюзивен, никто другой не может изменить общую память, пока с ней работает первый процесс.

Закончив проверку и обновление общего решения, он освобождает мьютекс, чтобы любой другой процесс, ожидающий своей очереди, мог продолжить работу.

В следующем коде показаны описанные выше события, но при условии исполь- зования процессами *A* и *B* мьютексов.

// Процесс B захватывает мьютекс, разблокируется и продолжает работу.

// Сохраняем лучшее решение. If (test\_length < BestLength) Then

// Сохраняем новое решение.

...

// Сохраняем новую общую длину.

BestLength = test\_length End If

// Освобождаем мьютекс.

...

// Процесс B пытается захватить мьютекс, но процесс A уже им владеет; процесс B блокируется.

// Эвристический анализ.

...

// Сохраняем новое решение.

...

// Сохраняем новую общую длину.

BestLength = test\_length End If

// Освобождаем мьютекс.

...

// Эвристический анализ.

...

// Захватываем мьютекс.

...

// Сохраняем лучшее решение. If (test\_length < BestLength) Then

Теперь два процесса не мешают друг другу пользоваться общей памятью, по- этому состояния гонки не возникает.

Стоит отметить, что в данном сценарии процесс *B* остается заблокированным, пока ждет мьютекс. Чтобы не тратить большое количество времени на ожидание, процесс не должен запрашивать мьютексы слишком часто.

В рассматриваемом примере, где выполняется эвристический анализ гамиль- тонова пути, процессу не нужно сравнивать каждый найденный им маршрут с об- щим лучшим решением. Вместо этого он может начать сравнение только тогда, ког- да находит более оптимальный вариант рядом с имеющимся у *него самого*.

При захвате мьютекса процесс также может обновить свое локальное реше- ние, чтобы у него был более короткий путь для сравнения. Представим, напри- мер, что процесс *A* находит новый лучший маршрут длиной 90. Он захватыва- ет мьютекс и обнаруживает, что длина общего лучшего решения (обнаружен- ного процессом *B*) равна 80. На данном этапе процесс *A* должен использовать это значение в качестве длины своего локального маршрута. Ему не обязатель- но знать оптимальное решение; достаточно уже того, что его будут интересо- вать маршруты длиной меньше 80.

При использовании мьютексов можно допустить следующие ошибки:

* + захватить мьютекс и не освободить его;
  + освободить мьютекс, который никогда не был захвачен;
  + удерживать мьютекс длительное время;
  + использовать ресурс без предварительного захвата мьютекса.

Даже если все делать правильно, у вас так или иначе могут возникнуть следу- ющие проблемы.

* + **Перестановка приоритетов.** Высокоприоритетный процесс блокируется в ожидании низкоприоритетного, владеющего мьютексом. В данном случае было бы неплохо забрать мьютекс у второго процесса и отдать его первому. Это означает, что низкоприоритетный процесс должен уметь каким-то обра- зом откатывать любые не завершенные им изменения и затем снова захваты- вать мьютекс. В качестве альтернативного решения можно заставить процес- сы отдавать мьютекс как можно быстрее, чтобы высокоприоритетный про- цесс не был заблокирован слишком долго.
  + **Голодание.** Процесс не может получить ресурсы, которые ему нужны для за- вершения работы. Иногда это происходит в ситуациях, когда операционная система пытается решить проблему перестановки приоритетов. Если высо- коприоритетная задача загружает центральный процессор, у процесса с низ- ким приоритетом может никогда не появиться шанса быть выполненным, из-за чего он не завершится.
  + **Взаимная блокировка.** Два процесса блокируются в ожидании друг друга. Более подробно взаимные блокировки освещены в следующем разделе.

#### Взаимная блокировка

При *взаимной блокировке* два процесса блокируют друг друга, пока каждый ждет освобождение мьютекса, захваченного другим.

Представьте, например, что процессам *A* и *B* нужны два ресурса, которые кон- тролируются мьютексом 1 и мьютексом 2. Предположим, процесс *A* захватил пер- вый мьютекс, а процесс *B* — второй. Теперь процесс *A* блокируется в ожидании мью- текса 2, а процесс *B* ждет мьютекс 1. Оба они заблокированы, поэтому ни один из них не способен освободить удерживаемый им мьютекс и дать возможность про- должить работу другому процессу.

Есть один способ, предотвращающий взаимные блокировки: нужно дого- вориться о том, что каждый процесс будет захватывать мьютексы в порядке их номеров (при условии, что они пронумерованы). Вернемся к предыдущему при- меру. Оба процесса попытаются захватить мьютекс 1. У одного из них это полу- чится, а другой будет заблокирован. Более успешный процесс сможет потом за- хватить мьютекс 2. Закончив работу, он освободит оба мьютекса, после чего их захватит другой процесс.

Эта проблема усугубляется в сложных средах, таких как операционные систе- мы, где за общие ресурсы соперничают десятки и сотни процессов и где нет четко- го порядка запрашивания мьютексов.

Частный случай взаимного блокирования, проблема обедающих философов, будет описан в этой главе позже.

#### Квантовые вычисления

Квантовые компьютеры управляют данными с помощью таких квантовых эф- фектов, как *запутанность* (частицы остаются в одинаковом состоянии, даже если их разделить) и *суперпозиция* (частица пребывает в нескольких состояниях одно- временно).

В настоящее время квантовые вычисления находятся в зачаточном состоянии. Очень немногие лаборатории могут позволить себе создание и использование даже небольшого квантового компьютера всего с несколькими кубитами (квантовыми битами — основной единицей информации в квантовых вычислениях). Пока что та- кую машину удается применять лишь для разложения на множители чисел 15 и 21 с использованием алгоритма Шора. Учитывая столь скромные результаты, вам, ве- роятно, стоит повременить с добавлением в свои программы квантовых алгоритмов.

Тем не менее все передовые технологии начинались с таких крошечных про- тотипов и существует вероятность того, что квантовые компьютеры когда-нибудь станут обычным явлением. Тогда производители смогут создавать по-настоящему недетерминированные и вероятностные вычислительные машины, способные точ- но решать задачи класса *NP*.

Например, алгоритм Шора может разложить число на множители за время *O*((log *N*)3), где *N* — размер входящего числа. Это намного быстрее самого опти- мального алгоритма, известного сегодня, — решета числового поля, которому для выполнения работы требуется экспоненциальное время (хотя все равно медлен- нее, чем методики полиномиального времени).

Квантовые вычисления — довольно запутанная тема, поэтому в данной книге мы не будем рассматривать их подробно. К счастью, в ближайшие не- сколько лет вам не придется писать собственные алгоритмы для квантовых компьютеров.

**ЗАМЕЧАНИЕ**

Больше информации о квантовых компьютерах и алгоритме Шора можно найти на страницах https://ru.wikipedia.org/wiki/Êâàíòîâûé\_êîìïüþòåð и https://ru.wikipedia.org/ wiki/Àëãîðèòì\_Øîðà.

## Распределенные алгоритмы

Некоторые виды параллелизма, описанные в предыдущих разделах, являются довольно редкими. Очень немногие домашние и промышленные компьютеры со- держат систолические массивы (хотя создание микросхемы для выполнения мгно- венной сортировки может быть оправданным). Прежде чем квантовые компьюте- ры появятся на прилавках магазинов, могут пройти десятилетия (если это вообще когда-нибудь случится).

Тем не менее, распределенные вычисления уже сейчас широко распростране- ны. В крупных грид-проектах для решения сложных проблем используется огром- ная вычислительная мощь десятков или даже сотен тысяч компьютеров. Менее масштабные сетевые кластеры позволяют работать вместе десяткам машин. Даже настольные и переносные системы в большинстве своем имеют несколько ядер.

Они могут полагаться на быстрое взаимодействие между ядрами на одном кри- сталле или учитывать медленное, ненадежное сетевое соединение, но во всех этих случаях используются распределенные алгоритмы.

В следующих двух разделах рассматриваются общие проблемы, характерные для распределенных методик: отладка и установка задач, которые хорошо распа- раллеливаются.

Далее описываются одни из самых интересных классических распределенных алгоритмов. Некоторые из них больше похожи на IQ-тесты или загадки, чем на практические методы, но они тоже являются полезными по нескольким причинам.

Во-первых, они подчеркивают некоторые проблемы, которые могут испыты- вать распределенные системы, и показывают, под каким углом нужно смотреть на задачу, чтобы суметь обнаружить в ней потенциальные проблемные места.

Во-вторых, эти алгоритмы на самом деле применяются в некоторых реальных ситуациях. Для многих приложений сбой какого-то одного набора процессов не име- ет значения. Если грид-процесс не возвращает результат, вы можете просто перело- жить задачу на другой компьютер и продолжать работу. Но если группа процессо- ров управляет системой жизнеобеспечения пациента, крупным пассажирским лай- нером или космическим кораблем стоимостью в миллиарды долларов, никогда не помешает лишний раз удостовериться в том, что процессы пришли к правильному решению, даже и в том случае, когда один из них вернул некорректный результат.

#### Отладка распределенных алгоритмов

Поскольку в сложных ЦПУ события могут происходить в произвольном по- рядке, отладка распределенных алгоритмов может оказаться крайне сложной. Вер- немся к примеру с гамильтоновым путем, описанному ранее. Состояние гонки воз- никает только в том случае, если события в процессах *A* и *B* происходят в строго

определенной последовательности. Если они не станут обновлять общее лучшее решение слишком часто, шанс столкнуться с попыткой одновременного обновле- ния будет небольшим. Эти два процесса могут работать очень долго, прежде чем что-то пойдет не так.

Но даже если возникнет проблема, вы можете ее просто не заметить. Обнару- жить ее реально только в тот момент, когда процесс *B* посчитает, что его решение лучше текущего сохраненного. Но прежде чем вы об этом узнаете, другой процесс может найти более оптимальный выход и перезаписать некорректный результат.

Некоторые отладчики позволяют изучать переменные, которые используются одновременно несколькими процессами; это дает возможность отслеживать проб- лемы в распределенных системах. К сожалению, останавливая процессы для ис- следования их переменных, вы сбиваете временные рамки, которые нередко явля- ются причиной ошибки.

Еще один подход заключается в том, чтобы заставить процессы записывать ин- формацию о своих действиях в файл или терминал для возможности дальнейшего ее изучения. Если запись выполняется слишком часто, имеет смысл использовать отдельные файлы, чтобы процессы не соперничали друг с другом за доступ к ре- сурсу. В этом случае они также должны добавлять временные метки, с помощью которых можно было бы понять, в каком порядке создавались записи.

Но даже если у вас есть хороший журнал событий, перед возникновением проб- лемы каждый процесс может прорабатывать часы и даже дни, выполняя миллио- ны шагов.

Вероятно, лучшим решением задачи отладки распределенных алгоритмов будет написание кода, который изначально не содержит ошибок. Тщательно обдумывай- те наиболее важные участки вашего приложения, где несколько процессов могут помешать друг другу, и используйте мьютексы, чтобы избежать неприятностей.

При написании программы ее нужно тестировать максимально тщательно. Стоит добавить код для регулярной проверки всех общих переменных, чтобы быть уверенным в корректности их значений. Если вы протестировали приложение и считаете, что оно работает надежно, можете закомментировать код для проверки значений и записи сведений в журнал — это повысит производительность.

#### Чрезвычайно параллельные алгоритмы

*Чрезвычайно параллельными* называют алгоритмы, которые естественным об- разом разбиваются на самостоятельные задачи, хорошо подходящие для решения отдельными процессами. Это не требует интенсивного взаимодействия между про- цессами; объединение результатов в идеале должно выполняться довольно легко.

Ниже представлены примеры чрезвычайно параллельных задач.

* + В компьютерной графике применяется методика, в которой луч трассирует- ся из точки обзора на сцену, чтобы увидеть, какие объекты он задевает. Луч способен проходить через прозрачные предметы и отражаться от зеркаль- ных. Данная задача является чрезвычайно параллельной, поскольку вычис- ления, необходимые для каждого луча, выполняются независимо, что позво- ляет легко разделить их между процессорами. Имея 10 процессоров, вы мо-

жете генерировать на каждом из них одну десятую часть изображения. Каж- дый процессор должен знать размеры сцены, но его не должны интересовать вычисления, выполняемые его «коллегами». Он записывает свои результа- ты в определенный участок изображения, поэтому здесь можно обойтись без мьютексов для управления доступом к общей памяти.

* + **Фракталы.** Многие фракталы, такие как множество Мандельброта, требу- ют выполнения длинных цепочек вычислений для каждого пикселя в ито- говом изображении. Как и в случае с трассировкой лучей, эти вычисления являются полностью независимыми, поэтому их легко распределять между доступными процессорами.
  + **Поиск методом полного перебора.** Если пространство поиска легко раз- бить на части, вы можете доверить каждую такую часть отдельному процес- су. Представьте, что вам нужно получить точное решение задачи о ранце и вы хотите смоделировать проблему в виде дерева принятия решений, описанно- го в главе 12. Будем исходить из того, что у вас есть восьмиядерный процес- сор. Каждая ветвь дерева имеет два ответвления, представляющие добавление элемента в ранец и изъятие его оттуда. В этом случае третий уровень дерева будет иметь восемь узлов, на нем вы можете назначать каждому процессору отдельное поддерево, которое будет возвращать лучшее найденное решение.
  + **Случайный поиск.** Если вам нужно производить поиск по пространству ре- шений случайным образом, вы можете сделать так, что каждый процессор будет искать отдельно, обновляя общее лучшее решение. В том случае ког- да пространство большое, вероятность того, что несколько процессоров од- новременно станут проверять одно и то же решение, является минимальной, поэтому на дублирование попыток не будет уходить много времени.
  + **Поиск по неиндексированной базе данных.** Если вам нужно выполнить по- иск по большой базе данных, у которой нет индексов, вы можете разделить ее на части и назначить их отдельным процессам. Представьте, например, что у вас есть библиотека из 100 000 фотопортретов и вы хотите найти тот из них, который больше всего похож на только что сделанную фотографию. Можно поделить библиотеку на 10 разделов с 10 000 фотографий в каждом и распределить их между 10 процессами.
  + **Обработка файлов.** Представьте, что вам нужно выполнить медленную опе- рацию с большим количеством файлов. Допустим, у вас есть база данных из 100 000 изображений и вы хотите создать для них эскизы, сделать рельеф- ные версии или выполнить с ними какую-то другую графическую опера- цию. Вы можете распределить их между группами процессоров и обраба- тывать отдельно.

**ОСТЕРЕГАЙТЕСЬ КОНКУРИРУЮЩИХ ПРОЦЕССОВ**

В примерах с неиндексированной базой данных и обработкой фотографий ис- пользуется большое количество файлов. Если вы хотите, чтобы этим занима- лось несколько процессоров, вам нужно знать, как долго выполняются опе- рации чтения и записи. Чтение и запись файлов на жестком диске происхо- дит значительно медленнее, чем обработка данных в памяти. Если выполня- емая вами операция относительно быстрая, процессы могут потратить мно- го времени, конкурируя за диск и ожидая своей очереди. В худшем случае это

затянется настолько, что производительность приложения будет в большей степени зависеть от скорости доступа к диску, чем от скорости самой обра- ботки (это называется *привязкой к диску*).

Конкуренции часто можно избежать, записывая файлы на разные диски или запуская процессы на разных компьютерах, на каждом из которых находится своя часть базы данных.

Иногда задачу можно полностью распараллелить естественным образом между тем количеством процессоров, которое у вас имеется; иногда это можно сделать только частично. Возможно, вам не удастся разделить все приложение между груп- пами процессоров, но вы можете попытаться сэкономить время, выделяя отдельные участки задачи.

В следующем разделе рассказывается, как выполнять сортировку слиянием на многопроцессорной системе. В разделах, которые идут дальше, описываются клас- сические алгоритмы для распределенных вычислений. Некоторые из них являют- ся не очень распространенными и могут показаться запутанными, однако они под- черкивают отдельные низкоуровневые проблемы, которые могут возникнуть в рас- пределенных системах.

#### Сортировка слиянием

Алгоритм сортировки слиянием, о котором рассказывалось в главе 6, является естественно рекурсивным. Его пошаговое высокоуровневое описание представлено ниже.

1. Разделяем значения на два подсписка одинакового размера.
2. Рекурсивно вызываем сортировку слиянием, чтобы упорядочить оба под- списка.
3. Объединяем два отсортированных подсписка в один итоговый список. Ниже объясняется, как распределить сортировку слиянием между *N* процес-

сорами, где *N* — относительно небольшое постоянное число.

1. Разделяем значения на *N* подсписков одинакового размера.
2. Запускаем *N* процессов для сортировки *N* подсписков.
3. Объединяем *N* отсортированных подсписков в один итоговый список.

Стоит отметить, что для упорядочивания отдельных подсписков процессорам не обязательно использовать сортировку слиянием.

#### Задача обедающих философов

*Задача обедающих философов* формулируется следующим образом. За столом сидят *N* философов, перед каждым находится тарелка спагетти. Между любыми двумя соседними философами лежит вилка. Философы предпочитают есть спагетти двумя руками, поэтому каждому из них нужно две вилки. Цель философа — от- ведать спагетти, отложить обе вилки на какое-то время, чтобы подумать, и затем снова приступить к приему пищи. Они повторяют этот процесс до тех пор, пока не постигнут все тайны Вселенной. Чтобы усложнить задачу, философы не должны

разговаривать друг с другом (по-видимому, они слишком погружены в свои раз- мышления).

Ниже пошагово описан один из алгоритмов, которым могли бы руководство- ваться философы.

1. Повторять до бесконечности:
   1. поразмышлять, пока не освободится вилка слева; взять ее;
   2. поразмышлять, пока не освободится вилка справа; взять ее;
   3. поесть досыта;
   4. отложить левую вилку;
   5. отложить правую вилку;
   6. поразмышлять, пока не вернется чувство голода.

К сожалению, данный алгоритм может привести к взаимной блокировке. Пред- ставим, что философы похожи друг на друга и что все они начинают трапезу од- новременно. Изначально каждый философ находит у себя слева свободную вил- ку и пытается ее взять. На этом этапе все вилки уже были разобраны теми, кто на- ходился справа от них. Следовательно, философы входят в ступор, ожидая, когда освободится вилка справа от них.

У этой проблемы есть несколько решений.

#### Привнесение эффекта случайности

Избавиться от взаимной блокировки можно следующим образом: если филосо- фу больше 10 мин не удается получить правую вилку, он откладывает левую вил- ку и ждет 10 мин. Однако нередко это приводит к *активной взаимной блокировке*, когда процессы блокируются не навсегда, но все равно не могут завершить рабо- ту из-за способа, с помощью которого они пытаются получить доступ к ресурсам. В данном примере все философы поднимут левую вилку, подождут 10 мин, поло- жат ее обратно, опять подождут 10 мин и начнут все сначала.

Впрочем, иногда простой эффект случайности может сдвинуть ситуацию с мерт- вой точки. Если философ берет вилку и ждет больше 10 мин, он может положить ее обратно. Даже если синхронизировать эти действия, вы все равно сумеете полу- чить активную взаимную блокировку.

Вместо 10 мин философ может ждать произвольное количество времени — на- пример, в диапазоне от 5 до 15 мин. Рано или поздно действия рассинхронизиру- ются настолько, что кто-то все-таки сумеет поесть.

Это решение может потребовать некоторого времени, в зависимости от ситуа- ции. Например, если множество процессов соперничает за большое количество общих ресурсов, рассинхронизация должна быть довольно существенной, прежде чем хотя бы один из процессов получит все необходимые ему ресурсы.

**ЗАМЕЧАНИЕ**

Не помешает также убедиться в том, что генераторы псевдослучайных чи- сел, которые используются для философов, не синхронизированы между собой, чтобы они не выбирали одно и то же «случайное» количество времени

для ожидания. Для инициализации генераторов можно использовать, напри- мер, их идентификаторы.

#### Иерархия ресурсов

В этом варианте решения ресурсы упорядочиваются в виде иерархии, и когда философ пытается их получить, он учитывает их уровень. Например, вы можете пронумеровать вилки от 1 до *N*, и тогда философы в первую очередь будут искать вилки с наименьшим номером. Если все философы берутся за вилку одновремен- но, большинство из них выбирает ту, которая лежит слева (предполагается, что но- мера вилок увеличиваются слева направо или против часовой стрелки).

Однако у последнего философа слева будет вилка *N*, а справа вилка 1, поэтому он потянется за правой вилкой. И здесь возможно два варианта развития событий в зависимости от того, удалось ему получить вилку или нет.

Если последний философ смог добраться до вилки 1, он возьмет и вилку *N*. Тем временем философ, сидящий слева от него, уже успел взять вилку *N* – 1 и теперь пы- тается получить вилку *N*. У одного из них окажется две вилки и он, наконец, поест.

Последний философ может не застать вилку 1, если его сосед справа взял ее первым. В этом случае философ, сидящий слева, берет вилку *N* – 1, лежащую по левую руку от него. Поскольку последний философ все еще ждет вилку 1, его со- сед слева теперь может беспрепятственно взять вилку *N* и поесть.

Если хотя бы один философ ест, это означает, что синхронизированный выбор времени, приведший к активной взаимной блокировке, нарушен. Как только насту- пает рассинхронизация, философы окончательно избавляются от перспектив беско- нечного блокирования, хотя время от времени им придется ждать свободной вилки.

#### Официант

Еще одно решение проблемы активной взаимной блокировки заключается в добавлении «официанта» (так называемого процесса-судьи). Прежде чем взять вилку, философ должен спросить разрешение у человека, который знает, где ка- кая вилка лежит, и способен предотвратить блокирование. Если философ просит вилку и это может застопорить процесс, официант просит его подождать, пока не освободится другая вилка.

#### Чанди/Мисра

В 1984 г. К. М. Чанди и Дж. Мисра из Техасского университета в Остине пред- ложили еще одно решение, которое позволяет любому числу процессов соперни- чать за любое количество ресурсов, хотя при этом необходимо, чтобы философы могли разговаривать друг с другом.

Любая вилка может быть чистой или грязной. Изначально все они считаются грязными. Исходя из этого, алгоритм состоит из следующих шагов.

1. В самом начале выдаем вилку соседнему философу с наименьшим иденти- фикатором (если вилки и философы пронумерованы так, как было описано в подразделе «Иерархия ресурсов» раздела «Распределенные алгоритмы» в текущей главе, вилку слева не смогут получить только философы 1 и *N*).
2. Когда философу нужна вилка, он просит ее у своего соседа.
3. Когда у философа попросят вилку, он оставит ее у себя, если она чистая. Если вилка грязная, он почистит ее и отдаст тому, кто ее попросил.
4. После начала приема пищи вилки философа становятся грязными. Если в этот момент их кто-то попросит, он почистит и отдаст их только после за- вершения трапезы.

Представим, что философы и вилки пронумерованы от 1 до *N* таким образом, что философ *K* имеет слева от себя вилку *K*. Изначально каждый философ име- ет одну вилку. Исключения составляют философ *N*, оставшийся с пустыми рука- ми, и философ 1, который взял вилки 1 и *N*. На этом этапе активная взаимная бло- кировка, которая могла возникнуть в синхронизированном решении, исключает- ся благодаря асимметрии.

После этого наличие чистых и грязных вилок, в сущности, заставляет филосо- фов делиться друг с другом. Если вы воспользовались вилкой, она становится гряз- ной, как следствие — ваш сосед при желании может взять ее у вас.

#### Задача двух генералов

В этой задаче у двух генералов есть армии, которые дислоцируются рядом со вражеским городом по разные стороны от него. Если оба генерала одновременно нападут на город, они победят. В том же случае, когда в нападении будет участвовать только один из них, победу одержит противник.

Теперь представим, что единственный способ, каким генералы могут общаться друг с другом, — отправка связного через вражеский город; при этом связной может быть захвачен. Нужно сделать так, чтобы генералы смогли договориться и пойти в атаку в одно и то же время.

Очевидное решение выглядит так: генерал *A* отправляет связного с сообщением о том, что армия *A* будет атаковать на рассвете. К сожалению, генерал *A* не может знать, прошло ли сообщение. Если он пойдет в атаку, а генерал *B* нет, то армия *A* будет разбита. Поэтому генерал *A* сильно заинтересован в том, чтобы атаковать только после того, как станет известно об успешном рейде связного.

Чтобы генерал *A* узнал о получении сообщения, генерал *B* может отправить ему подтверждение. Если генерал *A* его получит, он будет в курсе, что армии со- гласовали свои действия и атаковать можно в соответствии с планом. Но откуда генерал *B* узнает, что его подтверждение дошло до адресата? Если генерал *A* его не получил, генерал *B* не будет располагать сведениями о том, остается ли договорен- ность в силе и безопасно ли идти в атаку.

Очевидно, решение заключается в том, чтобы генерал *A* отправил подтвержде- ние о том, что он получил подтверждение генерала *B*.

Вы уже, вероятно, поняли, в чем кроется проблема. Неважно, сколько раз ге- нералы подтвердят свои планы, ведь невозможно узнать наверняка, что последнее сообщение дошло благополучно, поэтому нельзя сказать, достигнуто ли соглашение между армиями.

Один из способов разрешить данную дилемму — отправить достаточное количество копий одного и того же сообщения, чтобы гарантировать высокую

вероятность доставки хотя бы одного из них. Представим, например, что шан- сы захвата связного равны один к двум. Если один генерал пошлет *N* сообще- ний содержания «атакуем на рассвете», вероятность того, что все они будут перехвачены, составит 1/2*N*. Стопроцентной гарантии добиться нельзя, однако главнокомандующие армиями могут свести шансы на несогласованность до приемлемого минимума.

Но как они узнают вероятность того, что связной будет захвачен? Они могут догадаться об этом, обмениваясь сообщениями друг с другом. Сначала генерал *A* отправляет 10 сообщений с текстом «Это 1 сообщение из 10. Атакуем на рассвете». После какого-то небольшого промежутка времени генерал *B* получает некоторые из этих сообщений. Их количество (и тот факт, что всего их должно было быть 10) позволяет ему вычислить вероятность их успешного прохождения (а об атаке на рассвете он узнает из содержания посланий).

На основе этих данных генерал *B* решает, сколько подтверждений ему нужно отправить, чтобы с какой-то приемлемой долей уверенности полагаться на про- хождение хотя бы одного из них.

Такой подход будет работать, если сообщения будут доходить до генерала *B*, но как быть, если все они окажутся перехваченными? В данном случае генерал *A* не получит никаких подтверждений и не будет знать о том, дошли ли хоть какие-то из его посланий до адресата.

Чтобы решить эту проблему, генералу *A* нужно немного подождать. Если под- тверждений не будет, он отправит новый набор сообщений с текстом «Это 1 сооб- щение из 20. Атакуем на рассвете». Если их получение опять не будет подтвержде- но, он отправит еще один набор из 30 посланий. И так будет продолжаться до тех пор, пока в какой-то момент он не получит подтверждение.

Рано или поздно сообщения дойдут, генерал *B* выполнит вычисления и пошлет в ответ нужное количество подтверждений, а генерал *A* их получит.

#### Задача византийских генералов

В этой задаче византийские генералы должны согласовать план дей- ствий. К сожалению, некоторые из них могут оказаться предателями, рас- пространяющими противоречивые сведения среди своих коллег. Имеются следующие цели:

* + генералы, верные присяге, должны договориться о слаженных действиях;
  + если у них это получится, предатели не должны склонить их к какому-то другому решению.

Подходя к задаче более формально, каждому генералу можно присвоить зна- чение *Vi*; тогда те из них, кто остался верным присяге, должны узнать значения друг друга. Их цель можно сформулировать так: узнать значения *Vi* у других вер-

ных генералов.

Трудности возникают из-за того, что предатели передают другим генералам противоречивые сведения. Другими словами, один и тот же человек может отпра- вить генералу *A* одно значение, генералу *B* — второе, а вдобавок навести подозрения

на генерала *B* (например, сообщить кому-нибудь, что тот сказал ему нечто такое, чего последний никогда не говорил).

Данную проблему будет легче решать, если свести ее к связанной с нею *зада- че о генерале и лейтенантах*, в которой командующий генерал отдает приказ всем своим лейтенантам, но при этом любой из них (включая самого генерала) может оказаться предателем (рис. 18.2). Верные лейтенанты должны:

* + договориться о слаженных действиях;
  + выполнить приказ генерала, если тот не является предателем.

Стоит отметить, что эту задачу невозможно решить, если лейтенантов всего два и один из них предатель.

Предатель



Он сказать «наступать» Он сказал «отступать»



Он сказать «наступать» Он сказал «отступать»

Предатель

**Рис. 18.2.** Верный лейтенант не может распознать предателя, будь тот лейтенантом или генералом

В ситуации, представленной слева, предателем является генерал. Он раздает противоречивые приказы своим лейтенантам, которые честно передают их друг другу.

В правой части рисунка генерал остается верным присяге и приказывает обо- им лейтенантам отступать, но второй лейтенант искажает его приказы.

В обоих случаях первый лейтенант видит один и тот же результат — приказ к отступлению, пришедший от генерала, и приказ атаковать, полученный от дру- гого лейтенанта. Он не знает, где правда, а где ложь.

Если лейтенантов хотя бы трое (в общей сложности четыре человека) и толь- ко один из них предатель, задача будет иметь простое решение.

1. Генерал отдает приказы лейтенантам.
2. Каждый лейтенант рассказывает сослуживцам о том, какой приказ был по- лучен от генерала.
3. Каждый лейтенант действует согласно приказу, о котором он узнал от боль- шинства людей (включая генерала).

Чтобы понять, как это работает, взгляните на рисунок 18.3. Если генерал явля- ется предателем, как в левой части рисунка, он может отдавать лейтенантам про- тиворечивые приказы. В данном случае все лейтенанты остаются верными, честно сообщая о полученных ими сведениях. Это означает, что они, получив одну и ту же информацию, придут к единому выводу о том, какой из приказов имеет большин- ство. В левой части рисунка 18.3 все три лейтенанта видят два приказа атаковать и один приказ к отступлению, поэтому решают идти в атаку. Они выработали об- щее решение, которое совпадает с распоряжением верного генерала.

Если предателем оказывается лейтенант (см. правую часть рисунка 18.3), гене- рал отдает всем подчиненным один и тот же приказ. Предатель может попытать- ся внести неясность, распространяя противоречивые или ложные приказы, но два других лейтенанта имеют одинаковые сведения (поскольку генерал остался верен) и честно рапортуют о них друг другу. В зависимости от того, что сообщил измен- ник, другие два лейтенанта могут получить разные наборы приказов, но так как верных среди них достаточно, не стоит бояться, что большинство у каждого лей- тенанта предпочтет неверный приказ.

**ЗАМЕЧАНИЕ**

 Решение, основанное на приказе, который набрал большинство, подходит для задачи о генерале и лейтенантах, только если количество предателей равно T,

а количество лейтенантов не меньше 3  T.

Нападение

Нападение

Отступление Нападение



Нападение



Отступление

**Рис. 18.3.** Три лейтенанта всегда могут выработать общее решение, независимо от того, кто является предателем

Если вы поняли, как решить задачу о генерале и лейтенантах, то можете свести к ней задачу византийских генералов. Если исходить из того, что каждый генерал имеет значение *Vi*, для передачи корректных данных тем из них, кто является вер-

ным, нужно выполнить следующие шаги.

1. Для каждого генерала *Gi*:
   1. выполняем алгоритм генерала и лейтенантов с *Gi* в качестве командующего генерала. Остальные генералы ведут себя как лейтенанты, а *Gi* играет роль отданного приказа;
   2. все некомандующие генералы должны выбирать значение *Vi* для *Gi* боль- шинством голосов.

После выполнения всех этапов шага 1 каждый генерал будет знать значения, хранящиеся у его верных коллег. Они могут иметь разные сведения о том, какие значения находятся у предателей, но в контексте данной задачи это несущественно.

#### Согласование

В *проблеме согласования* некоторое количество процессов должны прийти к какому-то общему значению, даже если отдельные из них не могут этого сделать (случай очень похож на задачу византийских генералов, в которой генералы долж- ны согласовать план действий, несмотря на потенциальное наличие предателей). Существуют определенные правила.

* + **Результат.** Каждый корректный процесс рано или поздно выбирает значение.
  + **Корректность.** Если один корректный процесс останавливается на значе- нии *V*, его в итоге выбирают все остальные.
  + **Надежность.** Если корректный процесс выбирает значение *V*, значит это зна- чение должно было быть предложено неким корректным процессом.
  + **Согласие.** Все корректные процессы в конце концов должны прийти к одно- му и тому же значению.

Алгоритм «царь этапа» (*phase king*) решает проблему согласования, если отка- зывают не более *F* процессов и если общее количество процессов равно как мини- мум 4  *F* + 1. Например, чтобы справиться с одним сбоем, алгоритму потребует- ся не менее пяти процессов.

Представим, что у нас есть *N* процессов, *F* из которых отказывают. Изначаль- но каждый процесс пытается выбрать то значение, которое, по его мнению, долж- но быть итоговым. Пусть для процесса *P*i это будет *V*i.

Чтобы выдержать максимум F сбоев, алгоритм выполняет серию из *F* + 1 эта- пов. На каждом этапе один из процессов становится царем. Какой именно, может зависеть от его идентификатора или другого произвольного значения — главное, чтобы у всех этапов были разные цари.

Каждый из *F* + 1 этапов состоит из двух стадий. На первой стадии все процес- сы поочередно сообщают остальным свои текущие варианты итогового значения.

Процессы анализируют полученные варианты (плюс свой собственный) и на- ходят тот, который встречается чаще всего. Если такого нет, используется некое заранее подготовленное значение. Пусть в процессе *Pi* основная часть набирает ва-

риант *Mi*.

Во второй стадии текущий царь этапа, *Pk*, транслирует значение, набравшее у него большинство, всем остальным процессам, которые должны использовать его в качестве дополнительного фактора. Каждый процесс (включая царя этапа) прове-

ряет значение *Mi*, которое набрала основная часть. Если оно встречается чаще, чем *N*/2 + *F* раз, процесс обновляет свой вариант, присваивая *Vi* = *Mi*. В противном слу- чае процесс присваивает *Vi* значение, переданное царем этапа в качестве дополни- тельного фактора.

Чтобы понять, как это работает, возьмем для примера пять процессов, один из которых может оказаться некорректным, но на самом деле все пять рабо- тают без сбоев. Пусть царем этапа *i* будет процесс *Pi*, и пусть изначальные ва-

рианты процессов будут выглядеть так: *атака*, *отступление*, *отступление*,

*атака* и *атака*.

* + **Этап 1, стадия 1.** Процессы честно транслируют свои значения всем осталь- ным, поэтому каждый из них думает, что у варианта *атака* три голоса, а у отступления — только два.
  + **Этап 1, стадия 2.** Царь этапа транслирует значение, набравшее у него боль- ше всего голосов (атака), остальным процессам. Каждый процесс сравнивает количество раз, которое он встречал данное значение (во всех случаях это 3), с *N*/2 + *F* (которое равно 5/2 + 1 = 3,5). Поскольку значение, набравшее боль- шую часть голосов, встречается не чаще, чем 3,5 раза, все процессы исполь- зуют в качестве собственного варианта дополнительный фактор — *атака*.
  + **Этап 2, стадия 1**. Процессы опять честно транслируют свои значения друг другу. Теперь все они выбирают *атаку*.
  + **Этап 2, стадия 2**. Царь этапа транслирует значение, набравшее у него больше всего голосов (*атака*), остальным процессам. Теперь каждый процесс встре- чает это значение пять раз. Все они используют его в качестве своего вари- анта, поскольку 5 больше, чем 3,5.

Поскольку этот пример может справиться только с одним сбоем, он заканчива- ется всего лишь после второго этапа. Каждый процесс голосует за значение *атака*, которое действительно встречается чаще.

Рассмотрим более сложный пример. Пусть у нас будет пять процессов, как и прежде, но первый из них не просто отказывает, а является предателем (как в за- даче с византийскими генералами). Представим, что изначальные варианты вы- глядят так: *предатель*, *атака*, *атака*, *отступление*, *атака* (у предателя нет исход- ного варианта — он просто хочет запутать остальных).

* + **Этап 1, стадия 1**. На этом этапе царем является процесс-предатель *P*1. Про- цессы транслируют свои значения друг другу. Предатель соглашается с их вариантами, какими бы они ни были. Таким образом, процессы получают

следующие значения:

* *P*1 — <Предателю все равно>;
* *P*2 — *Атака*, *атака*, *атака*, *отступление*, *атака*;
* *P*3 — *Атака*, *атака*, *атака*, *отступление*, *атака*;
* *P*4 — *Отступление*, *атака*, *атака*, *отступление*, *атака*;
* *P*5 — *Атака*, *атака*, *атака*, *отступление*, *атака*.

Далее представлены варианты, набравшие большинство, и количество отдан- ных за них голосов в каждом процессе: <*предатель*>, *атака*  4, *атака*  4, *атака*  3 и *атака*  4.

* + - **Этап 1, стадия 2**. Царь этапа (*предатель*) раздает другим процессам про- тиворечивые дополнительные факторы. *P*2 и *P*3 он передает значение *атака*,

а *P*4 и *P*5 — *отступление*. Процессы *P*2, *P*3 и *P*5 считают, что все четыре раза большинство набирает *атака*, поэтому они принимают его в качестве ново- го варианта. У процесса *P*4 это значение набирает большинство только три раза, что не дотягивает до отметки 3,5, необходимой для полной уверенно- сти. В связи с этим *P*4 использует дополнительный фактор, *отступление*. Новые варианты процессов распределены так: <*предатель*>, *атака*, *атака*, *отступление*, *атака*.

* + **Этап 2, стадия 2**. На этом этапе царем становится корректный процесс *P*2. Процессы транслируют свои значения друг другу. В отчаянной попытке сбить всех с толку, предатель говорит остальным процессам, что они должны *от- ступать*. В итоге будут получены следующие значения:
* *P*1 — <Предателю все равно>;
* *P*2 — *Отступление*, *атака*, *атака*, *отступление*, *атака*;
* *P*3 — *Отступление*, *атака*, *атака*, *отступление*, *атака*;
* *P*4 — *Отступление*, *атака*, *атака*, *отступление*, *атака*;
* *P*5 — *Отступление*, *атака*, *атака*, *отступление*, *атака*.

Варианты, набравшие большинство, а также количество отданных за них го- лосов в каждом процессе выглядят так: <*предатель*>, *атака*  3, *атака*  3, *атака*  3 и *атака*  3.

* + **Этап 2, стадия 2**. Большинство у царя этапа набрало значение *атака* (встре- чалось три раза), поэтому *P*2 транслирует его всем остальным процессам в ка- честве дополнительного фактора. Все корректные процессы (включая царя этапа) видят значение *атака* меньше 3,5 раз, поэтому они соглашаются на еще один фактор (тоже *атака*).

В этот момент все процессы имеют в качестве текущего варианта значение

*атака*.

Данный алгоритм работает благодаря тому, что он проходит *F* + 1 этапов. Если отказов не больше *F*, он будет иметь как минимум один этап с честным царем.

Предположим, что на этом этапе корректный процесс *Pi* не встречает значение, которое было набрано большинством, больше *N*/2 + *F* раз. В таком случае он ис- пользует дополнительный фактор, полученный от царя.

Это означает, что все корректные процессы *P*i, которые не встречают значение больше *N*/2 + *F* раз, останавливаются в итоге на одном и том же варианте. Но что, если кто-то из них увидит значение больше *N*/2 + *F* раз? Поскольку количество не- корректных процессов не может превышать *F*, мы получим больше *N*/2 коррект- ных вхождений. То есть у значения будет настоящее большинство, вдобавок оно будет одинаковым для всех процессов, которые встречают его больше *N*/2 + *F* раз. В таком случае мы имеем значение, набирающее подавляющее число голосов, по- этому оно получит большинство и у царя текущего этапа (даже если он не видел его больше *N*/2 + *F* раз).

Из этого следует, что после правления честного царя этапа все корректные про- цессы проголосуют за одно и то же значение.

После этого уже не важно, что попытается предпринять некорректный царь эта- па. Корректные процессы (их насчитывается *N* – *F*) согласовали единое значение. Поскольку *F* < *N*/4, количество корректных процессов равно *N* – *F* > *N* – (*N*/4) =

= 3/4  *N* = *N*/2 + *N*/4. Значение будет равно *N*/2 + *N*/4 > *N*/2 + *F*, потому что *N*/4 > *F*. Но если корректный процесс встречает какое-то значение чаще, чем со- гласованный вариант, он использует именно его. Это означает, что все корректные процессы сохраняют собственные значения, независимо от попыток некорректно- го царя этапа их запутать.

#### Выбор лидера

Иногда набору процессов требуется некий центральный лидер для координации действий. Если у него происходит сбой или связь с ним теряется, процессы должны каким-то образом выбрать нового лидера.

*Алгоритм хулигана* использует для выбора лидера идентификаторы процессов.

Побеждает тот, у кого идентификатор больше.

Вопреки такому простому описанию, полная версия этого алгоритма являет- ся более сложной, чем можно было бы подумать. Необходимо предусмотреть не- стандартные ситуации, которые могут возникнуть при разнообразных неполад- ках в сети. Представьте, например, что один процесс объявляет себя лидером, а за- тем то же самое делает другой процесс, но с меньшим идентификатором. Лидером должен быть первый процесс, но очевидно, что второй просто не получил соответ- ствующее сообщение.

Полный алгоритм хулигана описывается в виде следующих шагов.

1. Если процесс *P* считает, что с текущим лидером произошел сбой (посколь- ку тот не отвечал дольше положенного времени), он транслирует всем про- цессам с большим идентификатором сообщение: «Вы живы?».
2. Если за отведенный период процесс *P* не получил ответ вида «Я жив» ни от одного процесса с более высоким идентификатором, он сам становится ли- дером, транслируя всем сообщение «Я лидер».
3. Если процесс *P* все-таки получает сообщение «Я жив» от одного из процес- сов с большим идентификатором, он начинает ждать, когда тот пришлет со- общение «Я лидер». Если процесс *P* не получает его за отведенный период, он считает, что у предполагаемого лидера произошел сбой, и предпринима- ет новые выборы с шага 1.
4. Если процесс *P* получает сообщение «Вы живы?» от процесса с меньшим идентификатором, он отвечает «Я жив» и начинает новые выборы с шага 1.
5. Если процесс *P* получает от процесса с меньшим идентификатором сообще- ние «Я лидер», он начинает новые выборы с шага 1.

На шаге 5, когда процесс с меньшим идентификатором объявляет себя лиде- ром, процесс, имеющий более высокий идентификатор, в сущности, говорит «Нет, ты не лидер», заставляет его уступить и перебирает командование на себя.

#### Снимок

Представьте, что у вас есть набор распределенных процессов, и вы хотите сде- лать снимок состояния всей системы, описав действия каждого из процессов в за- данный момент.

На самом деле довольно сложно решить, когда именно нужно создать снимок. Допустим, процесс *A* отправляет процессу *B* сообщение, которое все еще находит- ся в пути. Когда следует копировать состояние системы — до отправки сообщения, во время его передачи или после того, как оно дойдет до адресата?

Если выбрать первый вариант, процесс *A* может не запомнить свое состояние в этот момент и ничего не получится (разве что заставить все процессы запоминать свои предыдущие состояния, что может быть довольно обременительно).

Если состояния процессов сохранить, пока сообщение находится в пути, ре- зультат может оказаться несогласованным. Представьте, например, что вам нуж- но восстановить состояние системы путем сброса всех процессов до их сохранен- ного состояния. У вас не получится восстановить всю систему, потому что про- цесс *B* получил сообщение после создания снимка, а в восстановленной версии он его не получит.

Обратимся к конкретному примеру. Пусть процессы *A* и *B* хранят баланс бан- ковских счетов, принадлежащих клиентам *A* и *B*. Представим, что клиент *A* хочет перевести клиенту *B* 100 $. Процесс *A* вычитает деньги и передает процессу *B* со- общение, в котором просит положить на счет клиента *B* 100 $. И пока сообщение находится в пути, создается снимок системы. Если позже этот снимок будет вос- становлен, окажется, что клиент *A* уже послал 100 $, а клиент *B* их еще не получил. Таким образом, 100 $ теряются. Такой подход абсолютно не годится для управле- ния банковскими счетами. При сбое в сети вместе с сообщениями теряются и день- ги. Чтобы оба процесса знали о денежном переводе, необходимо использовать бо- лее безопасный согласованный протокол.

Итак, чтобы сделать хороший снимок системы, необходимо сохранять не только состояния всех процессов, но и любые сообщения, которые они передают друг другу.

Ниже пошагово описан алгоритм создания снимка, разработанный К. М. Чанди и Лесли Лэмпортом.

1. Любой процесс (назовем его *наблюдателем*) может инициировать создание снимка. Для этого он:
   1. сохраняет собственное состояние;
   2. отправляет сообщение о снимке остальным процессам. Сообщение содер- жит адрес наблюдателя и маркер, которым помечен снимок.
2. Если процесс впервые получает маркер определенного снимка, он:
   1. отправляет наблюдателю свое сохраненное состояние;
   2. прикрепляет маркер этого снимка ко всем последующим сообщениям, ко- торые будет отправлять другим процессам.
3. Допустим, процесс *B* получает маркер снимка и вслед за этим ему при- ходит сообщение от процесса *A*, к которому не прикреплено никакого

маркера. Это означает, что сообщение было в пути. Его послали перед тем, как процесс *A* получил маркер снимка, поэтому оно не было учтено в сохра- ненном состоянии этого процесса. Чтобы не потерять это сообщение, про- цесс *B* отправляет его копию наблюдателю.

Как только сообщения перестают циркулировать по системе, наблюдатель по- лучает запись о состоянии каждого процесса и о каждом сообщении, которое было в пути в момент создания снимка.

#### Синхронизация часов

Точная синхронизация часов может оказаться непростой задачей ввиду того, что время передачи сообщений в общей сети может варьироваться. Все становит- ся намного проще, если процессы минуют сеть и взаимодействуют напрямую. На- пример, если соединить проводом два компьютера, которые стоят в одной комна- те, их часы можно будет синхронизировать, измерив длину провода и подсчитав, сколько времени уходит на передачу сигнала по нему.

Такой подход работает, но он является довольно громоздким и не годится для синхронизации компьютеров, находящихся далеко друг от друга. К счастью, непло- хих результатов можно добиться и с использованием сети. Для этого нужно, чтобы за какой-то определенный короткий период время передачи сообщений в сети не варьировалось слишком сильно.

Представьте, что вам нужно синхронизировать часы процесса *B* с часами, ко- торые используются в процессе *A*. Время в процессе *A* будем считать эталонным.

Ниже пошагово изложен обмен сообщениями между процессами.

1. Процесс *A* отправляет процессу *B* сообщение, содержащее *TA*1 (текущее вре- мя согласно процессу *A*).
2. Процесс *B* получает сообщение и шлет процессу *A* ответ, в котором указаны

*TA*1 и *TB*1 (текущее время согласно процессу *B*).

1. Процесс *A* получает ответ и отправляет процессу *B* новое сообщение, содер- жащее *TA*1, *TB*1 и *TA*2 (новое текущее время согласно процессу *A*).

Теперь процесс *B* может выполнить некоторые вычисления, чтобы синхрони- зировать свои часы с процессом *A*.

Допустим, между часами возникло отклонение *E*, тогда *TB* = *TA* + *E* в любой заданный момент времени. Также введем в наше уравнение *D* — задержку, необ- ходимую для передачи сообщения от одного процесса к другому.

Когда процесс *B* записывает время *TB*1, в исходное сообщение добавляется за- держка *D*, которая нужна для его транспортировки из *A* в *B*, поэтому:

*TB*1 = (*TA*1 + *E*) + *D*.

Аналогичным образом в ответ процесса *A* записывается та же задержка *D*, ко- торая требуется для прохождения из *B* в *A*:

*TA*2 = (*TB*1 – *E*) + *D*.

Если вычесть второе уравнение из первого, получится:

*TB*1 – *TA*2 = (*TA*1 + *E* + *D*) – (*TB*1 – *E* + *D*) =

= *TA*1 – *TB*1 + 2  *E*.

Решение этого уравнения для погрешности E дает следующий результат:

*E* = (2  *TB*1 – *TA*2 – *TA*1) / 2.

Теперь, имея поправку по *E*, процесс *B* может откорректировать свои часы со- ответствующим образом.

Этот алгоритм исходит из того, что во время передачи сообщений туда и обрат- но задержка остается примерно одинаковой. Также предполагается, что доставка сообщений из *A* в *B* и из *B* в *A* занимает одно и то же время.

## Резюме

В этой главе освещались проблемы, связанные с параллельной обработкой. Были рассмотрены разные модели параллельных вычислений и описано несколь- ко алгоритмов, предназначенных для работы в распределенных системах. Некото- рые из них, такие как мгновенная сортировка, систолический массив или решение задачи обедающих философов, могут вам никогда не понадобиться, но все они под- черкивают те или иные проблемы, присущие распределенным системам. В число этих проблем входят состояние гонки, активная и пассивная взаимные блокиров- ки, согласованность и синхронизация.

Распределенные среды варьируются от настольных компьютеров и ноутбуков до многоядерных и масштабных грид-проектов, в которых для решения одной- единственной задачи задействуются сотни и тысячи вычислительных машин. Даже если закон Мура продержится еще одно-два десятилетия, в наши дни доступно столько недоиспользованной процессорной мощи, что было бы странно не вклю- чить ее в распределенные вычисления. Чтобы получить максимальную выгоду от современных вычислительных сред, которые становятся все более и более парал- лельными, вы должны иметь представление об этих проблемах и о методиках, ко- торые позволяют их решать.

## Упражнения

Звездочкой отмечены задачи повышенной сложности.

1. Создайте диаграмму, похожую на ту, что представлена на рисунке 18.1. Она должна описывать процесс одновременного упорядочивания последователь- ностей чисел 3, 5, 4, 1 и 7, 9, 6, 8 алгоритмом мгновенной сортировки. Обо- значьте одну из последовательностей полужирным начертанием или дру- гим цветом, чтобы во время работы алгоритма было легче отличить два спи- ска. Сколько дополнительных тактов понадобится для двух списков вме- сто одного?
2. Во многих системах процессы могут безопасно считывать общий участок памяти, а для безопасной записи им достаточно использовать мьютексы (предполагается, что такие системы поддерживают атомарное чтение, пото- му что они не позволяют прерывать процесс считывания). Как поведет себя

пример с гамильтоновым путем, если процесс прочитает общую итоговую длину маршрута внутри выражения If и затем захватит мьютекс в качестве первой операции в блоке If Then?

3\*. Рассмотрим решение Чанди/Мисра для задачи об обедающих философах. Представим, что философы синхронизированы и что все разом пытаются приступить к поеданию пищи. Пусть после еды они надолго уходят в раз- мышления, чтобы ни одному из них не нужно было возвращаться к обеду до того, как все остальные закончат с трапезой.

В каком порядке едят философы? Другими словами, кто ест первым, вто- рым, третьим и т. д. Подсказка: попробуйте пошагово изобразить этот про- цесс в виде рисунков — это может помочь?

1. Вернемся к задаче двух генералов. Что произойдет, если какое-то из первых сообщений дойдет до адресата, но генерал *B* окажется на редкость неудач- ливым и все его подтверждения будут перехвачены?
2. Пусть в задаче двух генералов *PAB* будет вероятностью того, что связной бу- дет схвачен по пути от *A* к *B*. Пусть вероятность того, что его схватят по до- роге от *B* к *A* будет равна *PBA*. В исходном алгоритме предполагается, что *PAB* = *PBA*, но представьте, что это не так. Каким образом генералы смогут вычислить вероятность обоих событий?
3. Рассмотрим задачу о генерале и лейтенантах с тремя участниками (рис. 18.2). Ее можно попытаться решить путем введения следующего правила: любой лейтенант, услышавший противоречащие друг другу приказы, должен под- чиниться приказу генерала. Почему это не сработает?
4. Рассмотрим задачу о генерале и лейтенантах с тремя участниками (рис. 18.2). Ее можно попытаться решить путем введения следующего правила: любой лейтенант, услышавший противоречащие друг другу приказы, должен от- ступить. Почему это не сработает?
5. Могут ли верные лейтенанты вычислить предателя в задаче о генерале и лей- тенантах с четырьмя участниками (рис. 18.3)? Если нет, то сколько лейте- нантов для этого потребуется?
6. Опять обратимся к задаче о генерале и лейтенантах с четырьмя участниками (рис. 18.3). Придумайте ситуацию, в которой лейтенанты могут вычислить предателя. Какие действия они должны предпринять? Естественно, смыш- леный предатель никогда не позволит этому случиться.
7. Как бы вы изменили задачу обедающих философов, чтобы при ее решении можно было использовать алгоритм выбора лидера? Как именно это могло бы помочь?
8. Может ли алгоритм хулигана или подобный ему помочь в решении задачи обедающих философов?
9. Сформулируйте задачу прожорливых философов. Она должна быть похожа на задачу обедающих философов, только ее участники всегда должны быть голодными. Закончив есть, философ кладет свои вилки на стол. Если прибо-

ры сразу же не будут взяты кем-то другим, он хватает их и продолжает обед. К каким проблемам это приведет? Какого рода алгоритм мог бы их решить?

1. Представьте, что в алгоритме синхронизации часов отправка сообщения от *A* к *B* и от *B* к *A* занимает разное время. К какому отклонению в итоговом значении для процесса *B* может привести эта разница?
2. В алгоритме синхронизации часов предполагается, что при обмене сообще- ниями время их передачи остается примерно одинаковым. К какому откло- нению в итоговом значении для процесса *B* может привести изменение ско- рости сети в процессе работы алгоритма?
3. Представьте, что на протяжении работы алгоритма скорость сети сильно колеблется. Как бы вы улучшили алгоритм синхронизации часов, используя ответ к упражнению 14?

# Глава 19 ГОЛОВОЛОМКИ, ВСТРЕЧАЮЩИЕСЯ

**НА СОБЕСЕДОВАНИЯХ**

Во время собеседования работодатели часто задают вопросы, которые требу- ют от соискателя навыков решения задач определенного рода. Каждая глава этой книги содержит упражнения, из которых могли бы получиться отличные вопросы для собеседования — по крайней мере, если кандидат на должность хорошо ориен- тируется в алгоритмах. Многие из этих вопросов могут оказаться довольно слож- ными для тех, кто давно не читал соответствующей литературы.

В последнее время определенные виды головоломок становятся популярными на собеседованиях в таких компаниях, как Microsoft и Google. Предполагается, что они должны помочь оценить креативность соискателя и его способность критиче- ски мыслить. К сожалению, в подобных головоломках используется множество до- пущений, которые могут оказаться неверными. Большинство реальных ситуаций, даже в программировании, нельзя выразить в виде задачек с весами и стеклянны- ми шариками. В них обычно нет остроумных подвохов или невероятных зацепок, которые кажутся до смешного очевидными, когда о них рассказывает кто-то дру- гой, но о существовании которых практически невозможно догадаться на 10-ми- нутном собеседовании.

Действительно, поиск оптимального решения для реальных проблем часто тре- бует креативности, но многие подобные головоломки не имеют с ней ничего обще- го. Вместо этого они просто показывают, сколь долго вы искали в Интернете ту са- мую задачку (или ее аналог), которую вам задали на собеседовании.

Рассмотрим следующие вопросы.

* 1. Почему крышки канализационных люков делают круглыми?
  2. С какой стороны у автомобиля находится крышка топливного бака?
  3. Что означает словосочетание *dead beef*?
  4. Каким должно быть следующее число в последовательности 17, 21, 5, 19, 20, 9, 15?

Подумайте немного (но только немного) над этими вопросами. Вот ответы с краткими комментариями.

1. Для того чтобы их нельзя было поддеть, поставить ребром и уронить в от- верстие. Это разумно (хотя подошли бы и другие формы, особенно если от-

верстие относительно небольшое, а крышка достаточно толстая), но вопрос сформулирован так, что вам нужно идти от решения к проблеме, а не наобо- рот. Как часто этим приходится заниматься в реальной жизни?

1. Он находится с противоположной стороны от выхлопной трубы (разве что крышка бака или выхлопная труба находятся по центру, но тогда вопрос не имеет смысла). Этот вопрос тоже требует от вас сформулировать задачу на основе готового решения (как предотвратить попадание бензина на раска- ленную выхлопную трубу).
2. Когда-то, во времена мейнфреймов и программирования на ассемблере, разработчики могли поместить в свой код шестнадцатеричное значение 0 x DEADBEEF, которое позволяет найти определенное место в программе. Этот вопрос не помогает оценить креативность или умственные способно- сти соискателя, он лишь дает понять, программировал ли тот когда-либо на ассемблере и сталкивался ли с подобным приемом. Было бы намного проще спросить у кандидата на должность, какой опыт программирования на ас- семблере он имеет (для справки, я учился разработке на ассемблере, но ни- когда не сталкивался с этим трюком).
3. Ответ — 14. Если пронумеровать буквы (например, *A*=1, *B*=2, *C*=3 и т. д.), последовательность в вопросе превратится в *QUESTIO*. Если вы об этом до- гадаетесь, вам не составит труда понять, что последней буквой должна быть *N* (чтобы получилось слово *QUESTION* — «вопрос»), номер которой 14. По- добная задачка заставляет соискателя думать о числах, хотя на самом деле речь здесь идет о буквах и кодировках. Это, скорее всего, бесполезный на- вык — разве что вы нанимаете криптографа (если вам действительно нужен криптограф, кандидата лучше спрашивать о преобразовании Лапласа и ги- перболических функциях).

В журнале *Journal of Applied Psychology* (журнал о прикладной психологии) была опубликована статья под названием «Почему канализационные люки круглые? Ла- бораторные исследования реакций на собеседования с головоломками», в которой ставится под сомнение полезность подобного рода вопросов. В ней говорится, что они не демонстрируют высокой эффективности в оценивании способностей логи- чески мыслить. Кроме того, кандидаты на должность считают такие вопросы не- справедливыми и бессмысленными, что может снизить их стремление к сотрудни- честву или привести к отказу от предложенной работы.

Означает ли это, что подобные вопросы бесполезны на собеседованиях? Безу- словно, если использовать их неправильно.

В следующих двух разделах объясняется, как их следует задавать и как на них отвечать.

## Как задавать вопросы с подвохом

В предыдущем разделе приводились примеры плохих головоломок. Они осно- вываются на знании определенных мелочей или в лучшем случае заставляют идти от разгадки к загадке. Для того чтобы мыслить в обратном направлении, действи- тельно нужна креативность, но вы можете быть новатором и без этого навыка.

Прежде всего эти задачки говорят о том, насколько скрупулезно кандидат на должность прочесывает Интернет в поисках головоломок, которые могут попасть- ся на собеседовании. Конечно, похвально, что человек тщательно подготовился, но это ничего не говорит о его сообразительности или умении решать проблемы.

Чтобы прояснить ситуацию, вам нужно задать соискателю вопрос, с которым он прежде не сталкивался. При этом, вопрос не должен быть чрезмерно сложным, чтобы человек не запаниковал. Он не должен основываться на подвохе или какой- то мелочи, иначе все, что вы узнаете — это читал ли кандидат определенный вы- пуск какого-то малоизвестного журнала.

К сожалению, это исключает использование множества головоломок. Те, что соответствуют критериям, требуют выполнить вычисления, дать оценку либо про- делать какую-то другую более или менее тривиальную операцию, которая дает кан- дидату простор для выбора разных подходов.

Например, один из вопросов, популярных на собеседованиях, формулирует- ся примерно так: «Сколько бейсбольных мячиков помещается внутри школьно- го автобуса?». Вряд ли кандидат заранее запомнил какое-то определенное число, поэтому ему придется дать свою приблизительную оценку. Хороший ответ будет содержать перечень допущений, на которых основана оценка, а также некоторые вычисления (предположим, что школьный автобус имеет длину 11 м и салон вы- сотой 2 м, размер бейсбольного мячика составляет 7,5 см и т. д.). Не так уж важно, точны ли допущения, — главное, чтобы процесс был осмысленным.

Этот вопрос показывает, способен ли кандидат выполнить предварительные расчеты, что является ценным качеством в программировании.

Еще одна вычислительная головоломка звучит следующим образом: «Сколь- ко мне лет, если я втрое старше моего брата и через два года мне будет в два раза больше лет, чем ему?» (ответ ищите в упражнении 6). Этот вопрос по большей части основывается на получении набора уравнений из задачи тождества. Безу- словно, данный навык может оказаться полезным, но задачи тождества приходят- ся по душе далеко не всем и большинство реальных проблем все равно не фор- мулируются в таком виде.

Пример головоломки с часами: «Сколько раз пересекаются часовая и минутная стрелки между полуднем и полночью?» (ответ ищите в упражнении 7). Подобно- го рода задачки решаются путем вписывания в таблицу определенных значений. Такой подход не дает кандидату проявить свою креативность, но может продемон- стрировать его организованность.

Еще один способ узнать о кандидате что-нибудь интересное с помощью голо- воломок — обсудить их постфактум. Например, вы можете попросить кандидата решить довольно простую задачу, с которой он, по вашему мнению, должен спра- виться. После этого предложите вместе обсудить, почему его решение работает, как он к нему пришел, какие еще подходы стоит попробовать (даже если они не сработают) и т. д.

Как вариант, вы можете дать кандидату очень сложную головоломку. Пусть он подумает над ней хорошенько, осознает все ее аспекты. Затем обсудите реше- ние. Так вы сможете поговорить о разных потенциальных способах получения результата.

**РЕШЕНИЕ НЕКОРРЕКТНОЙ ЗАДАЧИ**

Если дать кандидату нерешаемую задачу и небольшое количество времени, это не поможет ни ему, ни вам. Но как быть, если он не сумел ответить на во- прос, который, по вашему мнению, не представлял особой сложности? Вы мо- жете потратить остаток собеседования, расспрашивая кандидата о том, поче- му у него не получилось найти решение, и указывая на то, что задача выглядит просто, если посмотреть на нее определенным образом, — другими словами, издеваться над бедным парнем, раздувая собственное эго.

Но куда более продуктивно приуменьшить важность вопроса и продолжать со- беседование дальше. Вы можете сказать: «Все нормально. Почти никто не ре- шает данную задачу. На самом деле это лишь проверка того, как вы ведете себя в сложных ситуациях». А затем вернуться к нормальному ходу собеседования.

Наверное, вместо головоломки лучше описать ситуацию, которая напоминает реальную бизнес-проблему из вашего опыта. Например, вы можете сказать: «Да- вайте спроектируем базу данных, которая будет хранить планы на отпуск для при- шельцев с других планет». Эта задача является достаточно масштабной, тем самым позволяет кандидату продемонстрировать креативность и навыки проектирования баз данных, однако то, как нелепо она сформулирована, должно успокоить его. Если хотите, поработайте над задачей вместе — так вы поймете, как кандидат взаимо- действует с другими людьми. Или же предложите разные неожиданные повороты и попросите его рассказать, что может пойти не так в разных ситуациях, оценивая, насколько творчески он реагирует на непредвиденные обстоятельства. Подобного рода интерактивные испытания сложнее контролировать, и разные люди способны выдать совершенно разные решения, которые часто не так-то просто оценить. Но из этого можно извлечь куда больше информации, чем из банальной головоломки.

Вопросы с подвохом могут быть интересными и веселыми, однако следует пом- нить: они не самый лучший способ оценить характеристики, которые интересуют вас в соискателях на ту или иную должность.

## Как отвечать на вопросы с подвохом

В предыдущем разделе приводились доводы в пользу того, что вопросы с под- вохом на самом деле не помогают оценить качества, которые нужны работодателю от претендента на рабочее место. Они проверяют не креативность и способность к критическому мышлению, а умение запоминать мелкие детали и бороздить про- сторы Интернета.

Впрочем, тот факт, что головоломки не оценивают нужные качества, вовсе не означает, что они не попадутся вам на собеседовании. Некоторые интервьюе- ры проверяют с их помощью то, как вы справляетесь с давлением, реагируете на необоснованные требования и подходите к решению нереальных задач. Подобные головоломки могут не иметь ничего общего с творческим мышлением, но они спо- собны составить ваш психологический портрет.

Так как же следует реагировать на вопросы с подвохом? Первое и самое глав- ное: не паникуйте. Неважно, ожидают ли от вас решения задачи или просто хотят посмотреть на вашу реакцию. Паника в этом деле не поможет, она практически све- дет к нулю шансы на правильный ответ и выставит вас в плохом свете.

Вместо этого сосредоточьтесь на проблеме. Когда вы начнете над ней работать, у вас не останется времени, чтобы паниковать.

Многие головоломки на технических собеседованиях связаны с программиро- ванием. Вас могут попросить изменить порядок следования символов в строке на противоположный, отсортировать объекты каким-то необычным способом, ско- пировать структуру данных или выполнить какие-то другие прямолинейные, но сбивающие с толку задания. В таких ситуациях следует вспомнить известные вам алгоритмические методики. Приведем те, на которые стоит обратить внимание.

* + **Разделяй и властвуй.** Можно ли разбить задачу на части, которые легче ре- шать по отдельности?
  + **Рандомизация.** Возможно ли избежать худших сценариев с помощью ран- домизации?
  + **Вероятность.** Приходит ли вам на ум такой вероятностный метод, который способен угадать решение или решить задачу с определенной вероятностью?
  + **Адаптивные методики.** Можете ли вы придумать такой подход, который на- целен на отдельные участки задачи? Действительно ли задача состоит все- го из нескольких аспектов, а основная ее часть призвана лишь отвлечь вас от решения?
  + **Структура задачи.** Является ли структура задачи естественно рекурсивной, иерархической или похожей на сеть? Можете ли вы воспользоваться древо- видными или сетевыми алгоритмами для поиска данных?
  + **Деревья принятия решений.** Сумеете ли вы применить к задаче метод поис- ка по дереву принятия решений (во многих случаях это возможно, но рабо- тает слишком медленно)? Вы можете сказать: «Что ж, в такой ситуации не- плохо бы попытаться проверить все допустимые сочетания данных, но это займет целую вечность. Пожалуй, здесь лучше прибегнуть к методу “разде- ляй и властвуй”».

Если вам никак не удается продвинуться в решении задачи, можете восполь- зоваться следующими советами общего характера.

* + Убедитесь в том, что поняли задачу. Если что-то в ней вам кажется неясным, попросите уточнить.
  + Переформулируйте задачу, чтобы убедиться, что ее понимаете. Если вы сде- лали неудачное допущение, вас могут поправить.
  + Сравните текущую задачу с теми, которые уже встречались в прошлом.
  + Разбейте задачу на небольшие части. Если задача крупная, подбирайте раз- мер частей так, чтобы их можно было решить по отдельности.
  + Сосредоточьтесь на деталях. Иногда легче работать с мелкими нюансами.
  + Сосредоточьтесь на общей картине. Иногда детали приобретают смысл, толь- ко если рассматривать их как единое целое.
  + Составьте перечень известных вам фактов.
  + Назовите перечень фактов, которые вам хотелось бы узнать. Перечислите способы, с помощью которых можно это выяснить.
  + Составьте таблицу значений. Попробуйте пополнить ее новыми данными.
  + Делайте предположения и сразу же их проверяйте. Вы можете решить зада- чу, формулируя догадки и корректируя их для получения нужного вам ре- зультата.
  + Мыслите нестандартно. Если задача касается чисел, думайте о буквах, фигу- рах и других нечисловых значениях. Если речь идет о буквах, обратите вни- мание на числа.
  + Мозговой штурм. Расскажите вслух о шагах, которые собираетесь предпри- нять. Это может быть подходящим моментом для того, чтобы показать ин- тервьюеру, в каких методиках вы разбираетесь. «Бинарное последовательное деление, скорее всего, не сработает… Задача является естественно рекурсив- ной, но это приведет к бесконечному выполнению операций…». Опять-таки, вас могут поправить, но вы уместно продемонстрируете рекрутеру приемы, которыми владеете.
  + Нарисуйте изображение, если в этом есть какой-то смысл. Иногда может быть полезно взглянуть на задачу в графическом виде, а не в текстовом.
  + Если выбранный вами подход не дает сдвинуться с мертвой точки, попро- буйте другой. Интервьюеру неинтересно наблюдать за тем, как вы стойко идете по очевидно ложному пути.
  + Идите до конца или сдавайтесь. Если у вас есть время и интервьюер ясно дал понять, что вы можете решать предложенную вам головоломку сколь угодно долго, — дерзайте. Если же чувствуете, что времени не хватает, лучше спро- сить, стоит ли продолжать.

Неудача при решении головоломки вовсе не означает, что вы провалили собе- седование. Если старались, испробовали все приемы, которые приходят вам на ум, и все равно зашли в тупик, лучше спросить, стоит ли вам остановиться. Это может звучать примерно так: «Рекурсивный выглядит многообещающе, но мне кажется, что я что-то упускаю. Хотите, чтобы я продолжал?». Если интервьюер не хочет, чтобы вы останавливались, он сможет вам об этом сказать.

Даже если вам не удалось решить задачу, интервьюер наверняка составит о вас какое-то мнение. Если вы разговариваете во время работы, он сможет понять, ка- кие приемы вам известны, и каков ход вашей мысли. У вас есть шанс также пока- зать, что за предположения и действия помогают вам раскрыть проблему, прежде чем браться за решение, проявив при этом все свое упорство.

**ОДИН НЕСЕРЬЕЗНЫЙ ОТВЕТ**

Вы можете позволить себе один несерьезный ответ, но на этом шутки долж- ны закончиться. Например, на собеседованиях часто задают оценочный вопрос, такой как «Сколько бейсбольных мячиков помещается внутри школь- ного автобуса?» или «Сколько парикмахеров проживает в Тампе, Флорида?».

Подобные вопросы часто располагают к остроумным ответам. Например, если вас спросят: «Сколько вы бы запросили за очистку всех дымоходов в Детрой- те?», вы можете ответить: «Столько, сколько выдержит рыночная экономика» или «30 $ за дымоход». Вам позволительно даже сделать паузу, чтобы пода- вить смешок, но потом вам следует всерьез приняться за подсчет. Если шут- ливый ответ предполагался изначально, вас в этот момент остановят, но, ско- рее всего, интервьюер хочет увидеть, как вы справитесь с вычислениями, пол- ными неизвестных значений.

Если у вас нет ни малейшего представления о какой-то величине, оставьте ее в виде переменной. Выработав некое уравнение, вы сможете подставить в него какие-нибудь значения, просто чтобы увидеть, как оно себя поведет, и поду- мать над тем, выглядит ли оно корректным. В примере с дымоходами у вас мо- жет получиться уравнение наподобие следующего:

сумма = ставка  время  население  процент,

где сумма — итоговая сумма, которую вы запросите; ставка — почасовая опла- та (скажем, 20 $ в час); время — как долго чистится дымоход (например, 1 ч); население — население Детройта (скажем, 1 млн); процент — доля населе- ния, проживающего в домах с дымоходами (например, 25 %).

Последнее значение требует некоторых дополнительных расчетов. Вы може- те попробовать оценить количество людей в каждой семье и количество лю- дей, проживающих в частных домах, а затем прикинуть, как это соотносится с числом жильцов квартир, не оснащенных дымоходами.

Закончив с этим, вы сможете подставить в уравнение свои оценочные значе- ния и посмотреть, какая сумма у вас получится. Если взять приведенные выше цифры, результат будет следующим:

сумма = 20  1  1 000 000  0.25 = 5 000 000 $.

Не так уж важно, оказался ли ваш ответ правильным (почти наверняка он да- лек от реальности). Важно то, каким образом вы его получили.

Чего никогда не следует делать — так это придираться к задаваемым вопро- сам, чтобы продемонстрировать, насколько глупыми они являются. Просматривая веб-сайты с головоломками для собеседований, я наткнулся на статью, автор ко- торой отпускает едкие комментарии в сторону вопроса «Как бы вы устроили про- цесс копирования файла?». В статье кандидат дотошно расспрашивает о всяких подробностях: о каком типе файлов идет речь, нужно ли копировать права досту- па, должен ли файл быть зашифрован или помечен для резервного копирования и т. д. Так продолжалось до тех пор, пока вымышленный интервьюер не раздоса- довался и не заявил: «Слушай, просто скопируй чертов файл».

Суть статьи заключалась в том, что это был дурацкий вопрос, потому что никто не пишет свои собственные операции копирования файлов. В большинстве случа- ев это действительно так, хотя я работал над проектами, где процесс копирования был довольно мудреным из-за проблем с блокированием файлов. На самом деле наиболее узким местом в работе подобных проектов было ежедневное многократ- ное копирование десятков тысяч файлов между множеством компьютеров, кото- рые проводили с ними различные операции. Малейшая ошибка в процессе копи- рования означала потерю данных или накапливание сотен тысяч необработанных файлов. Даже если вопрос кажется вам бессмысленным, вы не можете быть в этом уверены, пока не узнаете всей подоплеки.

Попытки блеснуть интеллектом за счет уничижения поставленного вопроса не помогут вам получить работу. Предъявляя претензии к задаче, в лучшем слу- чае вам удастся продемонстрировать нетерпеливость и отсутствие заинтересо- ванности. В худшем — вызвать антипатию у интервьюера, который может поду- мать, что вы не способны ответить на поставленный вопрос и что проблемы ра- ботодателя вас не интересуют.

Куда лучше будет спросить, с чем именно связан данный вопрос, чтобы понять точку зрения интервьюера и ответить надлежащим образом.

## Резюме

Иногда головоломки на собеседованиях используются для оценки креативно- сти и способности критически мыслить. Они не очень хорошо для этого подходят, хотя с их помощью можно узнать, как кандидат на должность ведет себя в стрес- совых ситуациях.

Если вы интервьюер, избегайте вопросов, которые основываются на мелких деталях, требуют мыслить в обратном направлении от ответа к задаче или явля- ются настолько сложными, что разъяснить их можно только с изрядной долей ве- зения. Больше пользы принесут головоломки, требующие выполнить предвари- тельные вычисления.

В любом случае лучше задавать вопросы, которые имеют отношение к буду- щей работе кандидата. Вы также можете использовать упражнения, подобные тем, что приводятся в этой или какой-то другой книге, посвященной алгоритмам либо программированию в целом. Будьте осторожны, чтобы не выбрать слиш- ком сложные задачи. Только тот, кто изучал алгоритмы углубленно или относи- тельно недавно, сможет припомнить все тонкости поворота сбалансированного дерева или объяснить, почему Оптимизация p Сообщение (или даже просто по-

нять, что это означает).

Задавая вопросы и обсуждая способы решения проблем, вы, скорее всего, узна- ете о кандидате больше, чем предложив ему одну головоломку, для решения кото- рой у него может не хватить опыта.

Если вы сами являетесь соискателем, попытайтесь не паниковать и не обижать- ся, сталкиваясь с хитроумными задачками. Убедитесь в том, что поняли условие, и постарайтесь выдать лучший ответ, на который способны. Помните, что нерешен- ная задача вовсе не означает провал собеседования.

В Интернете можно найти огромное множество головоломок, если ввести в стро- ку поиска «задачи на собеседовании». Ознакомьтесь с некоторыми из них, чтобы получить представление о том, о чем вас могут спросить и как следует подходить к ответу. Даже если они не будут заданы, вы все равно не потеряете время впустую, ведь подобные задачи часто оказываются интересными и веселыми.

Не забудьте уделить внимание другим навыкам, которые могут пригодить- ся на собеседовании. Освежите в памяти алгоритмы, проектирование баз данных, архитектуру, управление проектами и другие связанные с этим темы. Последний, но далеко не маловажный совет: не забудьте обзавестись парочкой хороших книг, посвященных более общей подготовке к собеседованиям.

Ниже представлены ссылки на некоторые веб-сайты с особо интересными го- ловоломками, которые применялись в таких компаниях, как Microsoft и Google, и другой информацией о подобных задачах.

* + Насколько эффективными являются вопросы-головоломки на собеседо- ваниях: [http://resourceszone.com/blog/how-effective-are-those-puzzle-interview-questions.](http://resourceszone.com/blog/how-effective-are-those-puzzle-interview-questions)
  + 10 головоломок, встречающихся на собеседованиях в Google: [http://www.](http://www/) mytechinterviews.com/10-google-interview-questions.
  + 10 знаменитых головоломок, встречающихся при приеме на работу в Microsoft:

[http://www.mytechinterviews.com/10-famous-microsoft-interview-puzzles.](http://www.mytechinterviews.com/10-famous-microsoft-interview-puzzles)

* + Как успешно пройти собеседование в Google: <http://online.wsj.com/article/SB10001> 424052970204552304577112522982505222.html.
  + Технические собеседования: [http://www.techinterview.org.](http://www.techinterview.org/)
  + Группа в Facebook, посвященная головоломкам на собеседованиях:

[https://www.facebook.com/interviewpuzzles.](http://www.facebook.com/interviewpuzzles)

* + Головоломки для собеседований от Хайдона Ванга (стоит отметить, что на этом веб-сайте приводятся решения для многих из них): [http://www-cs-students.](http://www-cs-students/) stanford.edu/~hdwang/puzzle.html.
  + 5 лучших головоломок при приеме на работу в Miscrosoft: [http://dailybrainteaser.](http://dailybrainteaser/) blogspot.com/2010/08/top-5-microsoft-interview-questions.html.
  + Собеседования в компанию A2Z: головоломки (ответы приводятся без объ- яснений): [http://www.a2zinterviews.com/Puzzles/logical-puzzles.](http://www.a2zinterviews.com/Puzzles/logical-puzzles)
  + Вопросы, подходящие для собеседований: [http://www.knoowgle.com/Puzzles.](http://www.knoowgle.com/Puzzles)
  + Головоломки, подобранные редакцией веб-сайта CoolInterview: [http://www.](http://www/) coolinterview.com/type.asp?iType=619.
  + CareerCup: [http://www.careercup.com.](http://www.careercup.com/)
  + Олимпиады по математике (этот веб-сайт поддерживает организацию Math Olympiads, которая проводит математические состязания для учащихся 4–8 классов. Многие из задач похожи на вопросы, встречающиеся на собе- седованиях; плюс они довольно забавные): [http://www.moems.org.](http://www.moems.org/)

Подобным головоломкам также посвящено множество изданий. Одна из таких книг, задачи в которой будут в основном интересны программистам, называется

«Карьера программиста. Как устроиться на работу в Google, Microsoft или другую ведущую IT-компанию» (автор Гейл Лакман Макдауэлл, CareerCup, 2011). Мно- гие приведенные в ней головоломки помогают проверить то, насколько хорошо вы разбираетесь в важных структурах данных и методиках программирования, хотя попадаются и такие хитроумные вопросы (например, задача о черепахе и зайце), которые требуют знания определенных приемов.

## Упражнения

Ниже представлены краткие примеры отдельных видов головоломок, которые часто встречаются на собеседованиях.

1. У мужчины в ящике комода находится 10 коричневых и 10 черных носков. Проснувшись рано утром, он хочет найти носки, не включая свет в спальне, чтобы не разбудить свою жену. Сколько носков он должен взять с собой в го- стиную (где можно включить свет), чтобы гарантированно получить хотя бы одну совпавшую пару?
2. У вас есть 10 черных и 10 белых шариков, а также две миски. Вы можете раз- делить шарики между мисками так, как вам угодно. Затем вам завязывают глаза и меняют миски местами. Вы должны залезть рукой в миску и выбрать оттуда шарик. Каким образом следовало распределить содержимое мисок, чтобы максимально увеличить вероятность выбора белого шарика?
3. Если составить круг из случайно расставленных шариков (четырех красных и восьми синих), каковы шансы на то, что у вас не получится ни одной пары шариков одинакового цвета, прилегающих друг к другу?
4. Какая структура данных лучше всего подходит для того, чтобы изменить по- рядок следования записей в списке клиентов без использования дополни- тельной памяти?
5. Если я втрое старше своего брата, а через два года стану старше него в два раза, каков сейчас мой возраст?
6. Сколько раз пересекаются часовая и минутная стрелки между полуднем и полуночью?
7. В некоторых странах родители особенно ценят мальчиков, поэтому каждая пара рожает детей до тех пор, пока у них не появится наследник, и на этом останавливается. Если предположить, что рождение на свет мальчиков и де- вочек равновероятно, какую долю будут составлять девочки от общего на- селения?
8. Вы наняли консультанта, который требует оплату своих услуг золотом. Работа займет от одного до семи дней (вы не знаете заранее). Если консуль- тант проработает полную неделю, он получит золотой слиток, если меньше, вы будете давать ему 1/7 слитка за каждый день. На какое минимальное ко- личество частей нужно разделить слиток, чтобы консультант всегда мог по- лучить оплату, независимо от того, сколько он проработал?
9. У вас есть восемь золотых яиц, но вам известно, что одно из них всего лишь позолоченное, поэтому оно легче остальных. Как бы вы нашли это яйцо с по- мощью весов с двумя чашами, если позволено сделать только два взвеши- вания?
10. У вас есть пять баночек без этикеток, в каждой из которых находится от 10 до 20 таблеток. В четырех баночках содержатся 1-граммовые таблетки, а в пятой — 0,9-граммовые (плацебо). Как с помощью цифровых весов (не чашечных, а таких, что показывают вес в граммах) найти баночку с плацебо, используя всего одно взвешивание?

# Приложение А

**СОБРАНИЕ АЛГОРИТМИЧЕСКИХ ПОНЯТИЙ**

## Глава 1. Основы алгоритмизации

**Понимание алгоритмов.** Для того чтобы понимать алгоритмы, вам нужно усво- ить несколько связанных с ними аспектов.

* + **Поведение.** Всегда ли алгоритм находит лучшее решение?
  + **Скорость.** Как сильно колеблется скорость в зависимости от количества входных данных?
  + **Требования к памяти.** Являются ли они приемлемыми? Насколько исполь- зование памяти зависит от количества входных данных?
  + **Основные методики.** Можете ли вы их применить в своих задачах?

**Алгоритмы и структуры данных.** Структура данных хранит информацию, раз- мещенную в каком-то порядке. Алгоритм выдает результат. Нужно, чтобы ваш ал- горитм создавал и использовал структуру данных.

*Псевдокод* — это текст, который выглядит как исходный код, но не соответствует какому-то конкретному языку программирования. Перед выполнением его следует перевести в реальный программный код.

**Алгоритмические цели.** Чтобы приносить какую-то пользу, алгоритм должен быть корректным, эффективным и удобным в сопровождении.

«*О*» *большое* — это обозначение, которое описывает, как количество входящих данных коррелирует с требованиями к памяти или времени выполнения по мере масштабирования задачи. Оно игнорирует постоянные множители, беря во вни- мание только функцию для описания производительности с самым высоким по- казателем роста.

**Функции времени выполнения.** Некоторые распространенные функции вре- мени выполнения в порядке увеличения скорости роста: 1 (константа), log(*N*), sqrt(*N*), *N*, *N*  log(*N*), *N*2, 2*N* и *N*!

## Глава 2. Численные алгоритмы

**Рандомизация.** Программа может рандомизировать коллекцию объектов. Затем в качестве случайного объекта она может выбрать первый элемент рандомизиро- ванной коллекции. Например, чтобы получить из колоды пять случайных карт,

программа может выбрать первых пять из них, предварительно перетасовав колоду. Вы можете использовать разные диапазоны случайных величин (как в случае с под- брасыванием монеты для выбора числа от 0 до 7). Для генерирования приемлемой выборки случайные величины можно брать из неправильных источников.

**Равноправие.** Псевдослучайный процесс является равноправным, если все ге- нерируемые им результаты возникают с одинаковой вероятностью.

**Неправильные источники.** Псевдослучайный процесс считается неправиль- ным, если он неравноправен. В качестве примера можно привести шестигранную игральную кость, в которой в половине случаев выпадает 1.

**Вероятностный алгоритм.** Алгоритм, выдающий результат с заданной достовер- ностью, является вероятностным. Например, тест простого числа Ферма определяет непростые числа как минимум в 50 % случаев. Если многократно повторить тест, можно с высокой долей уверенности сказать, является ли число простым.

**Предварительно вычисленные значения.** При быстром возведении в степень используются предварительно вычисленные значения, чтобы ускорить получение экспонент. Этот же подход применяется в решете Эратосфена, чтобы быстро ис- ключать величины из списка потенциальных простых чисел. Многие программы можно оптимизировать, предварительно вычисляя значения (на лету или заблаго- временно) и сохраняя их для дальнейшего использования.

**Точность моделирования.** Методы прямоугольников и трапеций показывают, что качественное моделирование задачи может давать лучшие результаты, не требуя при этом большого объема дополнительной работы.

**Адаптивные методики.** Многие алгоритмы можно улучшить, если прилагать больше усилий для решения сложных частей задачи, а на более простые участки тратить меньше времени.

**Метод Монте-Карло.** Некоторые алгоритмы для получения приблизительного результата используют псевдослучайные значения. Такой подход часто уступает в точности детерминированным методикам, но во многих случаях его проще при- менить.

## Глава 3. Связные списки

В основе связных списков лежат объекты под названием «ячейки», каждый из которых содержит фрагмент данных и ссылку на следующую ячейку в списке.

В двунаправленных связных списках ячейки ссылаются как на следующий, так и на предыдущий элементы списка.

**Ограничитель.** В связных списках ограничителем называют ячейку, которая не содержит полезных данных и используется для обозначения начала или конца списка.

**Операции со связными списками.** В связных списках легко хранить коллекции объектов. При необходимости список может увеличиваться, чтобы вместить боль- ше данных. Он позволяет легко добавлять и удалять объекты внутри себя. Однако связные списки не очень эффективны при сортировке или поиске элементов.

## Глава 4. Массивы

*Массив* — это непрерывный участок памяти, который хранит элементы.

**Упаковка массива.** Вам разрешено располагать элементы внутри участка па- мяти, привязывая индексы массива к физическим адресам. Например, чтобы сэ- кономить место, можно сформировать треугольный массив, наладив контакт меж- ду индексами треугольного и одномерного массивов. Без особых усилий создают- ся ненулевые нижние границы, которые связываются с обычным массивом, где от- счет индексов начинается с нуля.

**Массивы с разрывом.** С помощью связных структур данных можно создавать массивы или матрицы с разрывом, которые эффективно используют память.

## Глава 5. Стеки и очереди

*Стек* — это структура данных, которая предоставляет доступ к элементам в по- рядке «последним пришел — первым ушел» (*last-in-first-out* — LIFO). Стек можно реализовать на основе связного списка или массива, хотя если массив заполнится, его придется расширять.

*Очередь* — это структура данных, которая предоставляет доступ к элементам в порядке «первым пришел — первым ушел» (*first-in-first-out* — FIFO). Очередь можно реализовать на основе связного списка или кольцевого массива, хотя во вто- ром случае, если массив заполнится, его придется расширять.

В очередях с приоритетом элементы извлекаются в порядке своей важности.

*Дек* — это очередь, которая позволяет добавлять и удалять элементы с обоих концов.

**Применение.** Разные алгоритмы при обработке элементов часто помещают их в стеки и очереди.

## Глава 6. Сортировка

**Вставка.** Алгоритм берет элемент и вставляет его в подходящую позицию в некой структуре данных. В качестве примера можно привести сортировку вставкой.

**Выбор.** Алгоритм анализирует пока еще не обработанные элементы, находит среди них лучший на данный момент и добавляет его в некую структуру данных. В качестве примера можно привести сортировку выбором.

**Сужение диапазона рассматриваемых элементов.** Алгоритм отслеживает ин- тересующий его диапазон, сужая его со временем, чтобы уменьшить количество элементов, которые нужно анализировать. В качестве примера можно привести сортировку пузырьком.

*Куча* — это дерево, в котором каждая вершина имеет значение, как минимум не уступающее по величине значениям своих дочерних элементов. Куча может использоваться в очереди с приоритетом.

**Сортировка полного дерева.** Алгоритм может хранить полное дерево в массиве.

**Разделяй и властвуй.** Алгоритм разбивает задачу на более мелкие части и ре- шает их отдельно, как правило, рекурсивно.

**Рандомизация.** Поскольку элементы изначально находятся в определенном порядке, быстрая сортировка может применять рандомизацию для уменьшения вероятности получить наихудший сценарий (это бесполезно в случае, когда оди- наковых элементов большое количество).

**Распараллеливание.** Сортировку слиянием, быструю и блочную сортировку можно распараллелить.

**Внешнее упорядочивание.** Ввиду того, как сортировка слиянием продвигается по памяти, она хорошо подходит для выполнения внешнего упорядочивания.

**Подсчет.** Если элементы имеют ограниченный диапазон, вы можете их по- считать, не прибегая к упорядочиванию. Этот подход используется в сортировке подсчетом.

**Разделение.** Чтобы упростить задачу, блочная сортировка разделяет элементы на блоки.

**Выбор подходящего алгоритма.** Алгоритмы сортировки — это тот случай, когда для задачи важно выбрать подходящее решение. От вашего выбора зависит, насколько быстро будет решена проблема — за секунды, минуты или годы.

## Глава 7. Поиск

Линейный (исчерпывающий) поиск просто циклически перебирает все воз- можные элементы, пытаясь найти подходящий. Время его работы составляет *O*(*N*). Если список отсортирован, алгоритм может завершиться преждевременно, то есть как только дойдет до позиции, в которой находится искомый элемент.

Алгоритм бинарного поиска на каждом своем шаге уменьшает интересующую его область вдвое, благодаря чему время его работы равно *O*(log (*N*)). Этот метод подходит для многих задач.

Алгоритм интерполяционного поиска угадывает место нахождения искомого элемента с помощью интерполяции. Время его выполнения равно *O*(log(log (*N*))).

## Глава 8. Хеш-таблицы

**Необходимые условия для хеширования.** Для процесса хеширования необхо- димо наличие структуры данных, где можно хранить значения, функцию хеширова- ния для привязки значений к структуре данных и политику разрешения коллизий.

**Привязка.** Хеш-таблицы используют функции хеширования для привязки значений из одной проблемной области (имена, идентификаторы работников) к другой (индексы массива). Хорошая функция хеширования привязывает любые входящие значения к случайному распределению исходящих данных.

*Связывание* — это использование связных списков на случай переполнения ди- апазона блоков.

**Упорядоченное связывание.** Сортировка связных списков улучшает произво- дительность многих алгоритмов.

*Открытая адресация* — это привязка данных напрямую к записям массива.

**Маркирование элементов как удаленных.** Во многих структурах данных уда- ление элементов является непростой задачей. Иногда элемент можно не вырезать, а просто пометить его удаленным, чтобы занимаемое им место было доступно для дальнейшего использования.

**Кластеризация.** В некоторых алгоритмах, таких как линейное и квадратичное пробирование, вероятность сохранения элемента в разных позициях может быть неравномерной. Это снижает их эффективность.

**Рандомизация.** Иногда рандомизация данных позволяет избежать негативных последствий. Например, для борьбы с кластеризацией в двойном хешировании используется две хеш-функции.

## Глава 9. Рекурсия

**Свойства рекурсии.** При использовании рекурсивных алгоритмов, помимо тре- бований ко времени выполнения и памяти, необходимо учитывать максимальную глубину рекурсии и объем занимаемого стека.

**Рекурсивный подход.** Рекурсивный алгоритм вызывает сам себя, чтобы сделать задачу более мелкой. На этапе, когда задача уменьшилась до нужных размеров, он решает ее без рекурсии и возвращается в исходную точку вызова.

**Рекурсивные определения.** Некоторые последовательности (например, факто- риальная функция и числа Фибоначчи) имеют естественно рекурсивные определе- ния. Для них легко подобрать рекурсивный алгоритм, хотя это не всегда оказывается настолько же эффективным, как нерекурсивный подход.

*Самоподобный фрактал* представляет собой кривую-инициатор, части которой рекурсивно заменяются кривыми-генераторами, что масштабируются, поворачи- ваются и преобразуются соответствующим образом.

*Алгоритмы с возвратом* — рекурсивный подход, который ищет частичные ре- шения. Если частичное решение не может быть расширено до полного, оно отвер- гается, после чего алгоритм возвращается к предыдущему подходящему варианту и использует его как отправную точку для дальнейшего поиска.

*Выборка* — неупорядоченное подмножество объектов. Количество вариантов выбора *k* элементов без дубликатов при условии, что общее число элементов рав- но *n*, описывается следующим уравнением:

(*n*) = *k*!( *n*!

.

1. !

*k n* –

Количество вариантов выбора *k* элементов из множества *n* элементов с дубли- катами описывается так:

(*n* + *k* – 1).

*k*

*Перестановка* — это упорядоченное подмножество элементов, взятое из множе- ства. Количество перестановок с дублированием *k* элементов, позаимствованных из множества размером *n* элементов, равно *nk*. Количество перестановок без дублика- тов составляет *n*  (*n* – 1)  (*n* – 2)  ...  (*n* – *k* + 1). В отдельном случае, когда *k* = *n* (именно под этим обычно и понимают перестановку), число перестановок равно *n*!.

**Удаление хвостовой рекурсии.** Хвостовую рекурсию можно заменить циклом, который перед окончанием сбрасывает параметры.

**Хранение промежуточных значений.** Если во время вычислений вам прихо- дится многократно рассчитывать одни и те же значения (как в случае с числами Фибоначчи), вы можете сэкономить время, сохраняя их в таблице поиска. Так они будут вычисляться всего один раз.

**Удаление общей рекурсии.** От рекурсии можно избавиться более универ- сальным образом, имитируя рекурсивные вызовы метода. Перед выполнением рекурсивного вызова переменные следует поместить в стек; по завершении их можно будет оттуда забрать.

## Глава 10. Деревья

Деревья используются во многих алгоритмах, поэтому вам следует запомнить их основные свойства.

**Логарифмический рост.** Если дерево правильно сбалансировано, его высота растет как логарифм количества его вершин.

**Большое число листовых вершин.** Если бинарное дерево правильно сбаланси- ровано, примерно половина его вершин являются листовыми.

**Индуктивное доказательство.** Доказательство путем индукции является по- лезной методикой доказывания теорем.

**Ветви и ссылки на объекты.** Ссылки и указатели позволяют связывать верши- ны с дочерними элементами. С их помощью также можно создавать связи между деревьями и строить сети.

**Обход.** Прямой, симметричный и обратный обход, а также обход в ширину позволяют обрабатывать вершины дерева в разном порядке.

**Упорядоченные деревья.** В упорядоченном дереве вершины сгруппированы определенным образом, поэтому во время симметричного обхода они обрабатыва- ются в порядке сортировки.

**Треды.** Это специальные ветви или ссылки, которые позволяют посещать вершины дерева или сети в необычном порядке. Структура данных может иметь столько тредов, сколько вы посчитаете нужным, хотя их обслуживание требует дополнительных усилий.

**Деревья знаний.** В игре «Животные» дерево знаний используется для пред- ставления вопросов и ответов на них (результатов).

Выражения могут быть представлены и вычислены в виде деревьев.

Квадро- и октодеревья делят на сегменты двух- или трехмерное пространство, ускоряя тем самым нахождение объектов.

## Глава 11. Сбалансированные деревья

**Амортизационные операции.** Иногда во время рутинных операций имеет смысл провести некоторую дополнительную работу, чтобы избежать выполнения куда более ресурсоемких действий в дальнейшем. Например, это может быть баланси- рование дерева при добавлении или удалении значений.

В АВЛ-деревьях для балансировки применяется операция поворота, благодаря чему разница в высоте двух поддеревьев в любой вершине не превышает 1.

**2–3-деревья.** Каждая внутренняя вершина имеет либо два, либо три дочерних объекта. Если их три, то при добавлении новой вершины она будет разделена на две с двумя дочерними объектами. Если их две, то при удалении одной из них остав- шаяся вершина обменивается значениями с вершиной того же уровня или объеди- няется с содержимым своего родителя.

***B*-деревья.** Каждая вершина содержит от *K* до 2  *K* значений, где *K* — порядок дерева. Вершина, хранящая *M* значений, имеет *M* + 1 дочерних объектов. *B*-деревья являются общим случаем 2–3-деревьев и используют похожую модель разделения и слияния вершин.

**Иерархически организованные *B*-деревья.** Перемещаясь вниз по дереву для добавления значения, алгоритм разделяет любые полные вершины, встречающие- ся на его пути, чтобы в случае необходимости иметь нужное пространство. Это еще один пример амортизационной операции, когда алгоритм выполняет какую-то до- полнительную работу, чтобы упростить свои дальнейшие действия.

## Глава 12. Деревья принятия решений

С помощью деревьев принятия решений можно смоделировать множество за- дач. Каждая ветвь дерева представляет собой один выбор.

*Деревья игры* можно определить как частный случай дерева принятия решений. Минимальная стратегия (минимакс) позволяет минимизировать позицию оппо- нента на игровой доске. Среди эвристических методик, применяемых в деревьях игры, можно выделить предварительное вычисление первого хода и ответных хо- дов, поиск родительских объектов и нумерацию позиций на доске (номера могут меняться со временем).

**Типы задач.** Многие сложные задачи имеют одну из двух форм: оптимизаци- онную и такую, что требует строгого ответа «да» или «нет». Деревья принятия ре- шений помогают моделировать оптимизационную форму. Если у них не получа- ется найти решение в виде ответа «да» или «нет», это не дает оснований утверж- дать, что решения не существует.

Примеры.

* + Чтобы определить упорядочивание элементов, ветвь дерева на уровне *K* пред- ставляет выбор одного из оставшихся элементов для *K*-й позиции в наборе. Дерево содержит *N*! листовых вершин.
  + Чтобы выбрать подмножество из коллекции элементов, ветвь дерева на уров- не определяет, находится ли *K*-й элемент внутри множества. Каждая вершина имеет две ветви. Дерево содержит 2*N* листовых вершин.
  + Чтобы выбрать подмножество размером *M* элементов из коллекции, в которую входит *N* элементов, ветвь дерева на уровне выбирает один элемент из остав- шихся *N* – *K*. Дерево содержит *N*  (*N* – 1)  ...  *K* листовых вершин.

Метод ветви и границы может значительно опережать поиск полным перебо- ром при нахождении оптимального решения.

**Эвристика.** При решении масштабных задач все деревья достигают огромных размеров, из-за чего нахождение точного результата становится невозможным. В связи с этим приходится обращаться к эвристическим методикам. К любому де- реву принятия решений можно применить такие виды эвристики, как случайный поиск, усовершенствование путей, имитация отжига и т. д. Алгоритм восхожде- ния на вершину (в том числе и с упорядочиванием) может выдать приемлемый ре- зультат чрезвычайно быстро, хотя для его использования необходимо определить, чем именно является «восхождение на вершину» в контексте конкретной задачи.

## Глава 13. Основные сетевые алгоритмы

Чтобы освежить в памяти сетевую терминологию, просмотрите первый раздел главы 13 и ознакомьтесь с таблицей 13.1.

В некоторых представлениях сети используется отдельный класс Link. В дру- гих информацию о звеньях записывают в узлы, из которых эти звенья выходят.

**Обход сети.** Алгоритмы обхода сети в глубину и ширину работают аналогично алгоритмам обхода деревьев, за исключением того, что в классе узлов появляется новое свойство, по которому можно узнать, посещался ли узел прежде. Без этого свойства алгоритм обхода способен войти в бесконечный цикл.

**Связность.** Вы можете определить, является ли сеть связной, начиная с задан- ного узла. Для этого ее нужно обойти, используя узел в качестве исходной точки, и затем установить, был ли посещен каждый узел в сети.

*Остовным* называют дерево, которое охватывает все узлы сети. Минимальное остовное дерево имеет наименьшие затраты из возможных. Обход остовного дерева, описанный в главе 13, является хорошим примером жадного алгоритма.

Алгоритм с установкой меток всегда добавляет к промежуточному значению элементы, которые затем войдут в итоговый результат. Поиск кратчайшего пути с установкой меток является обходом сети в ширину.

Алгоритм с установкой меток прикрепляет к промежуточному значению эле- менты, которые затем могут быть заменены другими элементами. Поиск кратчай- шего пути с коррекцией меток является обходом сети в глубину.

Поиск кратчайшего пути между всеми парами вершин находит маршрут меж- ду любыми двумя узлами сети. Он выполняется за полиномиальное время, одна- ко полином *O*(*N*3) достаточно велик для того, чтобы данный алгоритм был слиш- ком медленным для объемных сетей. Ему также требуется память объемом *O*(*N*2), что делает его довольно ресурсоемким в больших масштабах.

## Глава 14. Дополнительные сетевые алгоритмы

Топологическая сортировка расширяет частичное упорядочивание до полно- го, позволяя вам выполнять задачи в подходящем порядке. С ее помощью можно также определить, содержит ли сеть циклы.

Если карта является двухцветной, для нее легко найти способ раскрашивания двумя цветами. Задача определения трехцветной карты считается крайне слож- ной. Любую карту можно закрасить максимум четырьмя цветами, но сделать это не всегда просто. Жадный алгоритм во многих случаях способен закрасить карту без использования слишком большого количества лишних цветов.

Для поиска максимального потока можно задействовать (виртуальную) оста- точную сеть. Дополняющий путь показывает, как улучшить потоки. С помощью максимальных потоков можно распределять должности между работниками, на- ходить минимальный разрез потока и устанавливать количество несвязных путей между исходной и конечной точками.

## Глава 15. Строковые алгоритмы

Сопоставление скобок обеспечивает анализ математических, логических и дру- гих выражений. С его помощью можно также создавать синтаксические деревья, которые позволяют многократно вычислять выражения и генерировать структу- ры данных.

Регулярные выражения дают возможность программе определять, содержит ли текст подстроку, соответствующую шаблону. Алгоритм для работы с регулярными выражениями использует при обработке строки ДКА и НКА. Эти автоматы могут быть полезны и в других ситуациях — например, когда нужно управлять виртуаль- ным компьютером посредством набора простых правил.

Алгоритм Бойера — Мура позволяет программе искать в тексте заданную под- строку, не проверяя каждый символ этого текста. Принцип, лежащий в его основе, можно применять в разных сценариях: простая проверка (анализ конца целевой строки) иногда способна исключить из области поиска большие участки текста.

Алгоритм редакционного расстояния оценивает степень схожести двух строк или файлов. Узнать с его помощью, насколько похожи два объекта, можно в любой ситуации, в которой позволено определять типы доступных изменений.

## Глава 16. Криптография

Для большинства программистов данные алгоритмы представляют разве что академический интерес или развлечение. Тем не менее они демонстрируют не- сколько полезных методик.

**Перестановочные шифры.** Некоторые из этих алгоритмов представляют текст в виде массива, хотя, если вам нужно просто вычислить местонахождение тех или иных участков текста, массив использовать вовсе не обязательно.

**Одноразовые блокноты.** Обеспечивают доказуемо стойкое шифрование. Их недостаток заключается в том, что копии одного и того же блокнота необходимо безопасным образом передать как отправителю, так и получателю.

**Блочные шифры.** Разделяют сообщение на части, каждая из которых шифрует- ся отдельно. Большинство подобных алгоритмов многократно выполняют с блока- ми текста относительно простые операции, повышая тем самым стойкость к взлому.

**Алгоритмы шифрования с открытым ключом.** По сути, публикуют частичное преобразование, которое применяется отправителем. Завершает трансформацию получатель, когда восстанавливает исходное сообщение.

Среди других криптографических алгоритмов можно выделить хеширование и цифровую подпись.

## Глава 17. Теория вычислительной сложности

**Классы сложности.** Задачи (не алгоритмы) разделяют по классам сложности в зависимости от того, насколько трудно их решить.

**P и NP.** *P* — это множество задач, которые детерминированный компьютер мо- жет решить за полиномиальное время. Задачи класса *NP* тоже решаются за полино- миальное время, но уже недетерминированным компьютером. Вероятно, наиболее острый вопрос в информатике звучит так: «Равны ли классы *P* и *NP*?».

**DTIME и NTIME.** Задача входит во множество DTIME(*f*(*N*)), если детерми- нированный компьютер может решить ее за время *O*(*f*(*N*)). Если ее можно решить за тот же промежуток на недетерминированном компьютере, она является частью NTIME(*f*(*N*)).

**EXPTIME и NEXPTIME.** Задача входит во множество EXPTIME, если детер- минированный компьютер может решить ее за экспоненциальное время. Задачи класса NEXPTIME решаются за тот же промежуток, но уже недетерминирован- ным компьютером.

**Сведение.** С помощью сведений, занимающих полиномиальное время, мож- но показать, что одна задача является как минимум такой же сложной, как другая (в обычном программировании подобный подход используется для того, чтобы не изобретать совершенно новое решение).

**NP-полные задачи.** Задача является *NP*-полной, если она входит в *NP* и если к ней можно свести любые другие задачи этого класса.

**Определение, сообщение и оптимизация.** С помощью сведения можно пока- зать, что эти виды задач одинаково сложны — по крайней мере, с точки зрения те- ории вычислительной сложности.

## Глава 18. Распределенные алгоритмы

Существует множество видов параллелизма, включая систолические массивы, сетевые компьютеры, распределенные и квантовые вычисления, а также многопро- цессорную/многоядерную обработку в рамках одного компьютера. В настоящее

время наиболее распространенными являются распределенные вычисления на се- тевых и многопроцессорных устройствах.

Отладка параллельных алгоритмов может оказаться крайне сложной, посколь- ку проблемы с синхронизацией мешают воспроизвести некорректное поведение программы.

*Чрезвычайно параллельными* называют алгоритмы, которые можно естествен- ным образом подробить на части и сделать общими для нескольких процессов с минимальным взаимодействием.

*Сортировка слиянием* имеет естественно распределенную реализацию.

*Задача обедающих философов* посвящена проблеме взаимных блокировок, в том числе и активных.

*Задача двух генералов* демонстрирует проблемы небезопасного взаимодействия.

*Задача византийских генералов*, включая ее разновидности, имеет дело с процес- сами, в которых может произойти самый худший из возможных сбоев — это когда выдаются неправильные результаты вместо их полного отсутствия.

В проблеме согласования речь идет о ситуации, в которой несколько процессов должны согласовать общий результат.

*Задача выбора лидера* посвящена определению главного процесса, который за- нимается координацией и помогает решать многие другие распределенные про- блемы.

Снимок копирует состояние распределенной системы, включая состояния всех ее внутренних процессов и любые сообщения, которые в этот момент находятся в пути.

*Задача синхронизации часов* — это попытка синхронизировать два процесса при наличии коммуникационных задержек.

## Глава 19. Головоломки, встречающиеся на собеседованиях

Задачи-головоломки, которые используют на собеседованиях, далеко не всег- да помогают понять ход мыслей кандидата на должность и проверить его способ- ности к решению проблем. Часто они сводятся к мелким деталям, хитрым трюкам или необходимости мыслить в обратном направлении — от решения к самой задаче. Кандидаты, которые знают, в чем подвох, могут решить их, не демонстрируя особой креативности. Те же, кто не знаком с тонкостями подобных задач, столкнутся, ско- рее всего, с большими трудностями, и творческое мышление им здесь не поможет.

Задачи, связанные с программированием, на порядок выше головоломок с шари- ками или колодами карт. Еще лучше, если они не основаны на остроумных приемах.

Слишком сложная головоломка может заставить кандидата нервничать на про- тяжении всего собеседования (или оставшегося дня), и тогда вам вряд ли удастся узнать многое о том, как он ведет себя в обычных условиях.

# Приложение Б РЕШЕНИЯ К УПРАЖНЕНИЯМ

## Глава 1. Основы алгоритмизации

1. В новой версии алгоритма внешний цикл по-прежнему имеет *O*(*N*) ите- раций. Когда его счетчик равен i, внутренний цикл выполняется *O*(*N* – *i*) раз. Если сложить количество итераций внутреннего цикла, получится *N* + (*N* – 1) + (*N* – 2) +...+ 1 = *N*  (*N* – 1) / 2 = (*N*2 – *N*) / 2. Время работы но- вой версии все так же будет равно *O*(*N*2), хотя по скорости она, вероятно, бу- дет опережать оригинал.
2. Рассмотрите таблицу Б.1. Значение Бесконечность говорит о том, что про- грамма может выполнить алгоритм для задачи любого разумного размера. Демонстрационное приложение, доступное для загрузки на веб-сайте этой книги, генерирует следующие значения.

**Òàáëèöà Á.1.** Çàäà÷è ìàêñèìàëüíûõ ðàçìåðîâ, êîòîðûå âûïîëíÿþòñÿ çà ðàçíîå âðåìÿ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Время** | **LOG2(*n*)** | **SQRT(*n*)** | ***n*** | ***n*2** | **2*n*** | ***n*!** |
| Секунда | Бесконечность | 11012 | 1106 | 1000 | 20 | 10 |
| Минута | Бесконечность | 41015 | 6107 | 7746 | 26 | 12 |
| Час | Бесконечность | 11019 | 4109 | 60 000 | 32 | 13 |
| День | Бесконечность | 71021 | 91010 | 293 939 | 36 | 14 |
| Неделя | Бесконечность | 41011 | 61011 | 777 689 | 39 | 15 |
| Год | Бесконечность | 11027 | 31013 | 5 617 615 | 45 | 17 |

1. Вопрос в том, для какого *N* справедливо неравенство 1500  *N* > 30  *N*2? Ре- шение относительно *N* дает 50 < *N*, поэтому первый алгоритм будет медлен- нее, если *N* > 50. Второй алгоритм подход при *N* > 50, а первый — при *N*  50.
2. Вопрос в том, для какого *N* справедливо неравенство *N*3 / 75 – *N*2 / 4 + *N* +

+ 10 > *N*/2 + 8? Вы можете решить эту задачу любым удобным для вас спо- собом — например, с помощью алгебры или метода Ньютона (см. главу 2). Двумя положительными решениями данного уравнения являются *N* < 4,92 и *N* > 15,77. Это означает, что первый алгоритм следует использовать, если 5  *N*  15. Демонстрационное приложение Ch01Ex04, доступное для загрузки на веб-сайте данной книги, представляет оба уравнения в графическом виде и находит точки их пересечения с помощью метода Ньютона.

1. Имея *N* букв, вы можете сделать первой каждую из них. Это *N* вариантов. Как только сделаете свой выбор, для второй буквы останется *N* – 1 вариантов. Общее количество комбинаций будет равно *N*  (*N* – 1). Помните, что каж- дая пара учитывается по два раза (*AB* и *BA*), поэтому суммарное число не- упорядоченных пар составит *N* (*N* – 1) / 2. В представлении «О» большого это будет *O*(*N*2).
2. Если грани куба имеют длину *N*, площадь каждой из них равна *N*2. Куб име- ет шесть граней, поэтому площадь всей его поверхности составляет 6  *N*2. Если алгоритм генерирует одно значение для каждой единицы измерения площади, время его работы будет равно *O*(*N*2).
3. Если подойти к задаче менее формально, можно заметить, что поверхность куба зависит от величины *N*2. Не прибегая к вычислениям, легко прийти к вы- воду, что алгоритм будет работать на протяжении времени *O*(*N*2).
4. Если грани куба имеют длину *N*, такая же длина будет и у всех его двенад- цати ребер. Следовательно, общая длина ребер равна 12  *N*. Однако каждая кубическая единица, находящаяся в углу, является частью сразу трех граней, поэтому в выражении 12  *N* она учитывается целых три раза. У куба есть восемь углов, поэтому чтобы подсчет был правильным, нужно вычесть 2  8 из 12  *N*, — так каждая из кубических единиц, размещенных в углу, будет учтена только один раз. Реальное количество элементарных кубов составит 12  *N* – 16, значит время работы алгоритма будет *O*(*N*).
5. Если подойти к задаче менее формально, можно заметить, что общая длина граней куба зависит от *N*. Следовательно, без выполнения всех вычислений можно заключить: время работы алгоритма равно *O*(*N*).
6. В таблице Б.2 показано количество элементарных кубов для некоторых зна- чений *N*.

*Òàáë. Á.2.* Ýëåìåíòàðíûå êóáû äëÿ ðàçíûõ çíà÷åíèé *N*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***N*** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Количество кубов** | 1 | 4 | 10 | 20 | 35 | 56 | 84 | 120 |

1. По тому, как фигуры увеличиваются в длину, ширину и высоту (см. рис. 1.5), можно догадаться, что количество кубов каким-то образом зависит от *N*3. Если предположить, что количество кубов равно *A*  *N*3 + *B*  *N*2 + *C*  *N* +*D*, где *A*, *B*, *C* и *D* — некие константы, становится возможным подставить зна- чения из таблицы Б.2 и вычислить выражение для *A*, *B*, *C* и *D*. Сделав это, вы обнаружите, что количество кубов равно (*N*3 + 3  *N*2 + 2  *N*)  6, поэтому время выполнения алгоритма составляет *O*(*N*3).
2. Если подойти к задаче менее формально, можно заметить, что суммарный объем фигур зависит от *N*3. Даже если не прибегать к вычислениям, напра- шивается вывод о том, что алгоритм будет работать на протяжении време- ни *O*(*N*3).
3. Может ли быть алгоритм без структуры данных? Да. Алгоритм — всего лишь набор инструкций, поэтому наличие структуры данных для него не являет-

ся обязательным. В качестве примера можно привести численные алгорит- мы, описанные в главе 2.

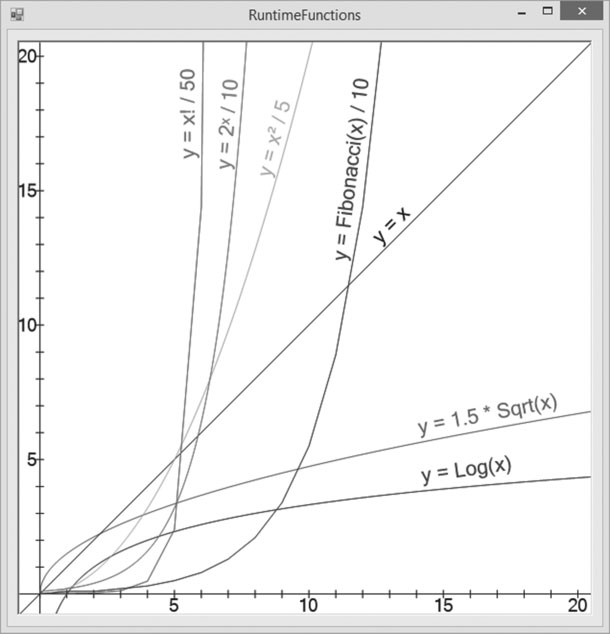
Возможно ли существование структуры данных без алгоритма? Нет. Алго- ритмы нужны как для построения структуры данных, так и для получения определенного результата с ее использованием. В структуре данных, кото- рая не используется ни в каких вычислениях, нет особого смысла.

1. Первый алгоритм просто красит доски забора из конца в конец. Количество досок равно *N*, поэтому время работы алгоритма составляет *O*(*N*).

Второй алгоритм рекурсивно делит забор на части, но в итоге закрашива- ет все *N* досок. На рекурсивное разделение забора и его покраску уходит, соответственно, *O*(log (*N*)) и *O*(*N*) шагов. Общее количество шагов равно *N* + log (*N*), поэтому время работы у второго алгоритма будет таким же, как и у первого — *O*(*N*).

Теоретическая продолжительность обоих алгоритмов одинакова, но на прак- тике, если отойти от представления «О» большого, второй из них работает чуть дольше. Кроме того, он отличается сложностью и запутанностью, ста- ло быть, лучше использовать первый алгоритм.

1. На рисунке Б.1 показана функция Фибоначчи вместе с другими функциями времени работы алгоритма. Внимательно приглядевшись, можно заметить, что по крутости подъема вверх Fibonacci(*x*)  10 превосходит x2  5 и не- много уступает 2*x*  10. По форме своей кривой она очень близка к функции 2*x*  10, поэтому вы будете правы, если предположите, что она является экс- поненциальной.



**Рис. Б.1.** Функция Фибоначчи растет быстрее, чем *x*2, но медленнее, чем 2*x*

Как известно, числа Фибоначчи можно вычислить непосредственно с помо- щью следующей формулы.

где:

*N*

*Fibonacci(n)* ✓5 ,



 1  ✓5 .

2

Таким образом, функция Фибоначчи является экспоненциальной с осно- вой . По скорости возрастания она уступает 2*N*, потому что значение  рав- но 1,618, но ввиду своей экспоненциальности превосходит полиномиаль- ные функции.

## Глава 2. Численные алгоритмы

* 1. Просто привяжите числа 1, 2 и 3 к орлу, а 4, 5 и 6 — к решке.
  2. В данном случае шансы на то, что орел выпадет два раза подряд, равны 3  4  3  4 = 9  16. Вероятность получить подряд две решки составляет 1  4  1  4 = 1  16. Поскольку эти события не зависят друг от друга, вы мо- жете суммировать их вероятности. Следовательно, шанс того, что вам при- дется повторить попытку, равен 9  16 + 1  16 = 10  16 = 0,625, или 62,5 %.
  3. Если монета правильная, вероятность того, что орел выпадет два раза под- ряд, составляет 1  2  1  2 = 1  4. Аналогично вычисляются шансы для двой- ной решки: 1  2  1  2 = 1  4. Поскольку эти события не зависят друг от дру- га, вы можете суммировать их вероятности. Стало быть, шанс того, что вам придется повторить попытку, равен 1  4 + 1  4 = 1  2, или 50 %.
  4. Здесь можно воспользоваться подходом, похожим на тот, в котором с по- мощью неправильной монеты генерировались правильные подбрасывания.

Бросаем неправильную кость 6 раз.

Если выпадают все 6 возможных значений, возвращаем первое из них. В противном случае повторяем все сначала.

От того, насколько неправильной является кость, зависит и количество по- пыток которые потребуются для выпадения всех шести значений. Напри- мер, если кость правильная (идеальный вариант), вероятность такого исхо- да будет составлять 6!  66 = 720  46 656 � 0,015, поэтому шансы на то, что шесть бросков дадут шесть разных результатов, равны примерно 1,5 %. Возь- мем еще один пример: если в половине случае встречается число 1, а каждое из оставшихся пяти значений попадается через девять бросков на десятый, вероятность получения всех шести результатов за шесть бросков составля- ет 0,5  0,15  6! = 0,0036, или 0,36 %. Поэтому кость, скорее всего, придется бросать очень долго.

* 1. Для рандомизации массива можно использовать тот же алгоритм, останав- ливаясь после выбора позиций для первых *M* элементов:

String: PickM(String: array[], Integer: M) Integer: max\_i = <Upper bound of array> For i = 0 To M - 1

// Выбираем элемент для позиции i в массиве.

Integer: j = <pseudo-random number between i and max\_i

inclusive>

Next i

<Swap the values of array[i] and array[j]>

<Return array[0] through array[M - 1]> End PickM

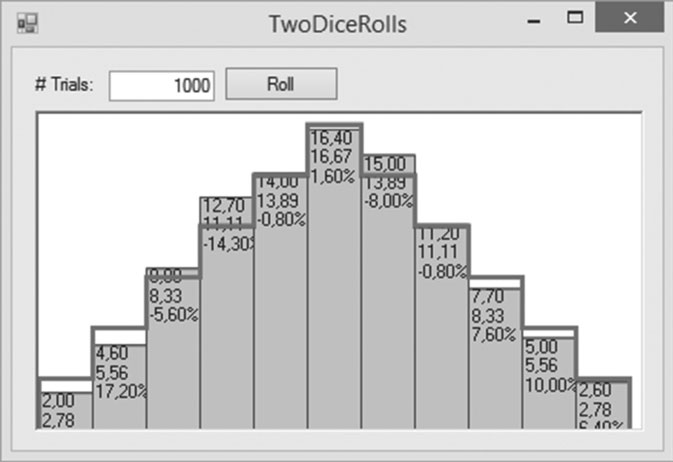
Время работы этого алгоритма равно *O*(*M*). *M*  *N*, поэтому *O*(*M*)  *O*(*N*). На практике *M* часто оказывается значительно меньше *N*, поэтому данный алгоритм может по скорости значительно превысить рандомизацию всего массива.

Чтобы раздать пять книг, нужно выбрать пять имен, которые следует раз- местить в первых пяти позициях массива, и затем остановиться. Вам потре- буется всего пять шагов, поэтому такой подход будет очень быстрым. И не- важно, сколько всего имен находится в массиве, — главное, чтобы их было не меньше пяти.

* 1. Просто создайте массив из 52 карт, рандомизируйте его и затем обращайтесь с картами так, как обычно — раздавайте их по одной за раз до тех пор, пока их количество у каждого игрока не достигнет пяти.

На самом деле не имеет значения, раздаете ли вы по одной карте или вруча- ете игроку сразу все пять. Если колода рандомизирована, у всех игроков бу- дет случайный расклад.

* 1. На рисунке Б.2 показана программа, написанная на C#. Ее можно загрузить на веб-сайте этой книги. Числа, указанные для каждого значения, — отно- сительная доля бросков, с помощью которых оно было получено, ожидае- мое процентное отношение для данного значения и относительная разница.



**Рис. Б.2.** Относительно небольшое количество попыток иногда приводит к существенной разнице между наблюдаемым и ожидаемым числом бросков

Полученные результаты не полностью совпадают с ожидаемыми значения- ми, пока не выполнить достаточно большое количество попыток. Програм- ма часто выдает 5%-ное отклонение, если число попыток меньше 10 000.

* 1. Если *A*1 < *B*1 и *A*1 mod *B*1 = *A*1, тогда *A*2 = *B*1 и *B*2 = *A*1 mod *B*1 =*A*1. Другими сло- вами, при первом прохождении цикла While значения *A* и *B* меняются ме- стами. После этого алгоритм продолжает работу как обычно.
  2. LCD(*A*, *B*) = *A*  *B* / GCD (*A*, *B*). Пусть *g* = GCD(*A*, *B*), тогда *A* = *g*  *m* и *B* = *g*  *n*, где *m* и *n* — некоторые целые числа. Из этого следует, что *A*  *B*  GCD(*A*, *B*) = *g*  *m*  *g*  *n*  *g* = *g*  *m*  *n*. У *m* и *n* нет общих множите- лей, следовательно, это наименьшее общее кратное.
  3. В следующем псевдокоде показан алгоритм, который используется в демон- страционном приложении FastExponentiation, доступном для загрузки на веб-сайте данной книги.

// Выполняем возведение в степень.

Integer: Exponentiate(Integer: value, Integer: exponent)

// Создаем список степеней и соответствующих значений. List Of Integer: powers

List Of Integer: value\_to\_powers

// Начинаем со степени 1 и значения^1. Integer: last\_power = 1

Integer: last\_value = value powers.Add(last\_power) value\_to\_powers.Add(last\_value)

// Вычисляем другие степени, пока не дойдем до той, что больше экспоненты.

While (last\_power < exponent) last\_power = last\_power \* 2

last\_value = last\_value \* last\_value powers.Add(last\_power) value\_to\_powers.Add(last\_value)

End While

// Объединяем значения, чтобы получить нужную степень. Integer: result = 1

// Получаем индекс самой большой степени, не превышающей экспоненту.

For power\_index = powers.Count - 1 To 0 Step - 1

// Проверяем, не превышает ли данная степень экспоненту.

If (powers[power\_index] <= exponent)

// Если нет, используем ее.

exponent = exponent - powers[power\_index] result = result \* value\_to\_powers[power\_index]

End If Next power\_index

// Возвращаем результат. Return result

End Exponentiate

* 1. Вам придется изменить строки, выделенные в вышеприведенном коде жир- ным шрифтом. Если модуль равен *m*, нужно заменить строку

last\_value = last\_value \* last\_value

на

last\_value = (last\_value \* last\_value) Modulus m

Также нужно поменять строчку

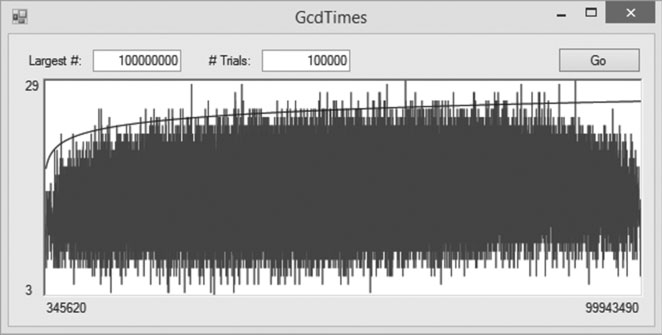
result = result \* value\_to\_powers[power\_index]

на

result = (result \* value\_to\_powers[power\_index]) Modulus m

Данный алгоритм демонстрируется на примере приложения ExponentiateMod, доступного для загрузки на веб-сайте этой книги.

* 1. На рисунке Б.3 представлено демонстрационное приложение GcdTimes, ко- торое можно загрузить на веб-сайте этой книги. Серыми линиями изображен график зависимости количества шагов от значений. Логарифм значений вы- водится в виде кривой, размещенной вверху. Глядя на этот график, сложно сказать, действительно ли количество шагов коррелирует с логарифмом, но растет оно определенно очень медленно.



**Рис. Б.3.** Демонстрационное приложение GcdTimes выводит график зависимости количества шагов в алгоритме НОД от размера значений

* 1. Вы уже знаете, что значение next\_prime × 2 было убрано, поскольку оно кратно 2. Если next\_prime > 3, тогда next\_prime × 3 тоже вычеркива- ется, поскольку число 3 уже рассматривалось. На самом деле проверку прош- ли все простые числа p меньше next\_prime, поэтому удаленным можно счи- тать и значение next\_prime × p. Первое простое число, которое не мень- ше next\_prime, — само next\_prime, поэтому первый множитель, кото- рый еще не был вычеркнут, равен next\_prime × next\_prime. Это озна- чает, что вы можете внести в цикл следующие изменения.

// Вычеркиваем множители этого простого числа.

For i = next\_prime \* next\_prime To max\_number Step next\_prime Then is\_composite[i] = true

Next i

* 1. Следующий псевдокод демонстрирует алгоритм для вывода чисел Кармайкла и их простых множителей.

// Генерируем числа Кармайкла. GenerateCarmichaelNumbers(Integer: max\_number)

Boolean: is\_composite[]

<Make is\_composite a sieve of Eratosthenes for numbers 2 through max\_number>

// Ищем числа Кармайкла. For i = 2 To max\_number

// Игнорируем простые числа. If (is\_composite[i]) Then

// Проверяем, является ли i числом Кармайкла. If (IsCarmichael(i)) Then

<Output i and its prime factors>

Next i

End If

End If

End GenerateCarmichaelNumbers

// Возвращаем true, если number — число Кармайкла. Boolean: IsCarmichael(Integer: number)

// Проверяем всех возможных свидетелей. For i = 1 to number  1

// Проверяем только те числа, для которых GCD(number, 1) = 1. If (GCD(number, i) == 1) Then

<Use fast exponentiation to calculate i ^ (number-1) mod number>

Integer: result = Exponentiate(i, number - 1, number)

Next i

End If

// Если мы нашли свидетеля Ферма, это не число Кармайкла.

If (result != 1) Then Return false

// Они все обманщики!

// Это число Кармайкла. Return true

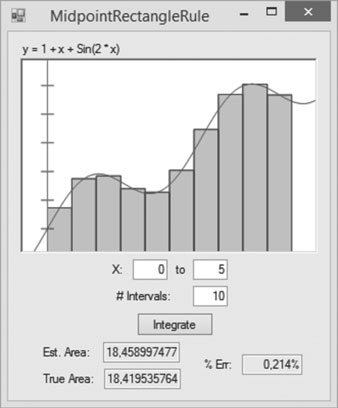
End IsCarmichael

Чтобы ознакомиться с реализацией данного алгоритма на языке C#, мо- жете загрузить на веб-сайте этой книги демонстрационное приложение CarmichaelNumbers.

* 1. Предположим, что вы используете значение функции, равноудаленное от верхней и нижней точек прямоугольника. При возрастании функции левая часть прямоугольника становится слишком низкой, а правая — очень высо- кой, поэтому отклонения в обеих частях постепенно уравновешиваются (по крайней мере, до какого-то предела). Аналогичный процесс происходит при убывании функции, что существенно нивелирует отклонение, при этом не увеличивая количество прямоугольников.

Однако этот метод не сработает (и может даже навредить), если локальный минимум или максимум кривой находятся рядом с центром прямоугольни- ка. В таких случаях отклонения левой и правой части кривой будут сумми- роваться, приводя к еще большей общей погрешности.

Приложение MidpointRectangleRule, которое демонстрирует этот подход, показано на рисунке Б.4; его можно загрузить на веб-сайте данной книги. Если сравнить результат с тем, что представлен на рисунке 2.2, нетрудно за- метить, что использование центральной точки прямоугольника снизило об- щее отклонение с примерно –6,5 % до 0,2 % — практически в 30 раз. И для этого не пришлось менять количество прямоугольников.



**Рис. Б.4.** Демонстрационное приложение MidpointRectangleRule снижает погрешность, используя для вычисления высоты прямоугольников их центральные точки

* 1. Да. Подобная программа будет похожа на разновидность приложения AdaptiveGridIntegration, в котором вместо решетки используются псевдо- случайные точки. Если сделать все правильно, это будет значительно эффек- тивнее, чем адаптивная интеграция Монте-Карло, поскольку больше точек будет приходиться на интересующие нас области и меньше на участки, ко- торые располагаются либо полностью внутри фигуры, либо за ее пределами.
  2. Следующий псевдокод демонстрирует алгоритм для выполнения интеграции Монте-Карло в трехмерном пространстве.

Float: EstimateVolume(Boolean: PointIsInShape(,,), Integer: num\_trials, Float: xmin, Float: xmax, Float: ymin, Float: ymax,

Float: zmin, Float: zmax) Integer: num\_hits = 0 For i = 1 To num\_trials

Float: x = <pseudorandom number between xmin and xmax>

Float: y = <pseudorandom number between ymin and ymax>

Float: z = <pseudorandom number between zmin and zmax>

If (PointIsInShape(x, y, z)) Then num\_hits = num\_hits + 1

Next i

zmin)

Float: total\_volume = (xmax - xmin) \* (ymax - ymin) \* (zmax —

Float: hit\_fraction = num\_hits / num\_trials \* Return total\_volume \* hit\_fraction

End EstimateVolume

* 1. Чтобы найти точки пересечения функций *y* = *f*(*x*) и *y* = *g*(*x*), можно восполь- зоваться методом Ньютона, который позволяет получить корни уравнения *y* = *f*(*x*) – *g*(*x*). Эти корни и будут значениями абсциссы, при которых пере- секаются *f*(*x*) и *g*(*x*).

## Глава 3. Связные списки

1. С учетом того, что в программе имеется переменная bottom, указывающая на последний элемент связного списка, процесс добавления элемента в конец списка можно изобразить в виде следующего псевдокода.

Cell: AddAtEnd(Cell: bottom, Cell: new\_cell) bottom.Next = new\_cell

new\_cell.Next = null

// Возвращаем новую последнюю ячейку. Return new\_cell

End AddAtEnd

Данный алгоритм возвращает новую последнюю ячейку, поэтому перемен- ную, которая на нее указывает, легко обновить внутри вызывающего кода. Вы также могли бы передать указатель bottom по ссылке, чтобы его преоб- разовал сам алгоритм.

Использование указателя bottom позволяет не изменять алгоритмы для поиска элементов или добавления их в начало списка.

Операция удаления происходит аналогичным образом, разве что выреза- емый элемент располагается в конце списка; в этом случае придется допол- нительно обновлять указатель bottom. Поскольку элемент, который нужно изъять, определяется указателем на значение, находящееся перед ним, изме- нения будут минимальными. Следующий код демонстрирует модифициро- ванный алгоритм для удаления последнего элемента списка.

Cell: DeleteAfter(Cell: after\_me, Cell: bottom)

// Если удаляемая ячейка является последней, обновляем bottom. If (after\_me.Next.Next == null) Then bottom = after\_me

// Удаляем заданную ячейку. after\_me.Next = after\_me.Next.Next

// Возвращаем указатель на последнюю ячейку. Return bottom

End DeleteAfter

1. На примере следующего кода представлен алгоритм для нахождения наи- большего элемента в несортированном однонаправленном списке с ячейками, содержащими целые числа.

Cell: FindLargestCell(Cell: top)

// Если список пустой, возвращаем null. If (top.Next == null) Return null

// Переходим к первой ячейке с данными. top = top.Next

// Сохраняем ячейку и ее значение. Cell: best\_cell = top

Integer: best\_value = best\_cell.Value

// Переходим к следующей ячейке. top = top.Next

// Проверяем другие ячейки. While (top != null)

// Смотрим, является ли значение ячейки большим.

If (top.Value > best\_value) Then best\_cell = top best\_value = top.Value

End If

// Переходим к следующей ячейке. top = top.Next

End While

Return best\_cell End FindLargestCell

1. В следующем псевдокоде показан алгоритм для добавления элемента в на- чало двунаправленного списка.

AddAtBeginning(Cell: top, Cell: new\_cell)

// Обновляем ссылки Next. new\_cell.Next = top.Next top.Next = new\_cell

// Обновляем ссылки Prev. new\_cell.Next.Prev = new\_cell new\_cell.Prev = top

End AddAtBeginning

1. В следующем псевдокоде показан алгоритм для добавления элемента в конец двунаправленного списка.

AddAtEnd(Cell: bottom, Cell: new\_cell)

// Обновляем ссылки Prev. new\_cell.Prev = bottom.Prev bottom.Prev = new\_cell

// Обновляем ссылки Next. new\_cell.Prev.Next = new\_cell new\_cell.Next = bottom

End AddAtEnd

1. Алгоритм InsertCell принимает в качестве параметра ячейку, после ко- торой должен быть вставлен новый элемент. Алгоритмам AddAtBeginning

и AddAtEnd достаточно в InsertCell занять подходящую ячейку. Новый алгоритм показан на примере следующего кода.

AddAtBeginning(Cell: top, Cell: new\_cell)

// Вставляем после верхнего ограничителя. InsertCell(top, new\_cell)

End AddAtBeginning

AddAtEnd(Cell: bottom, Cell: new\_cell)

// Вставляем после заданной ячейки, но перед нижним ограничителем.

InsertCell(bottom.Prev, new\_cell) End AddAtEnd

1. Следующий псевдокод демонстрирует алгоритм, удаляющий определенную ячейку из двунаправленного списка.

DeleteCell(Cell: target\_cell)

// Обновляем ссылку Prev следующей ячейки. target\_cell.Next.Prev = target\_cell.Prev

// Обновляем ссылку Next предыдущей ячейки. target\_cell.Prev.Next = target\_cell.Next

End DeleteCell

Графическое представление этого процесса показано на рисунке Б.5.

целевая ячейка

...

...

...

...

целевая ячейка

...

...

...

...

**Рис. Б.5.** Для удаления элемента из двунаправленного списка нужно изменить ссылки следующей и предыдущей ячеек, чтобы «обойти вокруг» заданного элемента

1. Если имя, которое вам нужно найти, располагается ближе к концу алфавита, чем к его началу, оно должно начинаться с *N* (или другой буквы, следующей за ней). Следовательно, вы можете выполнить поиск по списку в обратном направлении, начиная с нижнего ограничителя. Это не повлияет на теорети- ческое время работы *O*(*N*), но на практике может ускорить поиск примерно в два раза, если имена распределены достаточно равномерно.
2. Следующий псевдокод демонстрирует алгоритм вставки элемента в отсор- тированный двунаправленный список.

// Вставляем элемент в отсортированный двунаправленный список.

InsertCell(Cell: top, Cell: new\_cell)

// Находим ячейку, перед которой нужно вставить новый элемент.

While (top.Next.Value < new\_cell.Value) top = top.Next

End While

// Обновляем ссылки Next. new\_cell.Next = top.Next top.Next = new\_cell

// Обновляем ссылки Prev. new\_cell.Next.Prev = new\_cell new\_cell.Prev = top

End InsertCell

Этот алгоритм похож на тот, который использовался для однонаправленного списка, если не считать двух строчек, которые обновляют ссылки Prev.

1. Следующий псевдокод определяет, является ли связный список отсорти- рованным.

Boolean: IsSorted(Cell: sentinel)

// Если список содержит 0 или 1 элемент, он отсортирован.

If (sentinel.Next == null) Then Return true

If (sentinel.Next.Next == null) Then Return true

// Сравниваем другие элементы. sentinel = sentinel.Next; While (sentinel.Next != null)

// Сравниваем этот элемент со следующим.

If (sentinel.Value > sentinel.Next.Value) Then Return false

// Переходим к следующему элементу. sentinel = sentinel.Next

End While

// Если мы добрались сюда, список отсортирован. Return true

End IsSorted

1. В процессе сортировки методом вставки из исходного списка берется пер- вый элемент и затем в растущем отсортированном списке ищется место, куда его нужно расположить. Это место может находиться ближе к началу или концу списка, в зависимости от значения элемента. Алгоритму не обя- зательно перебирать весь список, разве что новый объект больше всех тех, что уже отсортированы.

В отличие от вышеприведенного метода, сортировка методом выбора иссле- дует весь исходный список, чтобы найти самый большой элемент. В связи с этим она никогда не может завершить поиск преждевременно.

1. Одно из решений демонстрирует приложение PlanetList, доступное для за- грузки на веб-сайте этой книги.
2. Программа BreakLoopTortoiseAndHare, доступная для загрузки на веб- сайте данной книги, демонстрирует решение на языке C#.

## Глава 4. Массивы

1. Следующий алгоритм вычисляет дисперсию выборки массива.

Double: FindSampleVariance(Integer: array[])

// Находим среднее.

Integer: total = 0

For i = 0 To array.Length - 1 total = total + array[i]

Next i

Double: average = total / array.Length

// Находим дисперсию выборки. Double: sum\_of\_squares = 0 For i = 0 To array.Length - 1

sum\_of\_squares = sum\_of\_squares +

(array[i] - average) \* (array[i] - average)

Next i

Return sum\_of\_squares / array.Length End FindSampleVariance

1. В следующем коде алгоритм, описанный выше, используется для расчета стандартного отклонения выборки.

Integer: FindSampleStandardDeviation(Integer: array[])

// Находим дисперсию выборки.

Double: variance = FindSampleVariance(array)

// Возвращаем стандартное отклонение. Return Sqrt(variance)

End FindSampleStandardDeviation

1. Поскольку массив отсортирован, медианой является элемент в центре мас- сива. Здесь есть два момента, на которые следует обратить внимание. Во- первых, с массивами, имеющими четную и нечетную длину, необходимо об- ращаться по-разному. Во-вторых, нужно быть осторожным при вычислении индекса центрального элемента, помня, что отсчет начинается с 0.

Double: FindMedian(Integer: array[]) If (array.Length Mod 2 == 0) Then

// Массив имеет четную длину.

// Возвращаем среднее значение двух центральных элементов.

Integer: middle = array.Length / 2

Return (array[middle - 1] + array[middle]) / 2

Else

End If

// Массив имеет нечетную длину.

// Возвращаем центральный элемент. Integer: middle = (array.Length - 1)/ 2 Return array[middle]

End FindMedian

1. Следующий псевдокод удаляет элемент из линейного массива.

RemoveItem(Integer: array[], Integer: index)

// Сдвигаем элементы на 1 позицию влево, чтобы перекрыть местонахождение элемента. For i = index + 1 To array.Length - 1

Array[i - 1] = Array[i]

Next i

// Изменяем размер, убирая последнюю лишнюю запись.

<Resize the array to delete 1 item from the end> End RemoveItem

1. Все, что вам нужно изменить в начальном классе треугольного массива, — это метод, который вычисляет индекс в одномерном массиве с помощью строки и столбца. Исходный метод показан в виде следующего псевдокода.

Integer: FindIndex(Integer: r, Integer: c) Return ((r - 1) \* (r - 1) + (r - 1)) / 2 + c

End FindIndex

Ниже приведен псевдокод, демонстрирующий новую версию, в которой стро- ка и столбец меняются местами.

Integer: FindIndex(Integer: r, Integer: c) Return ((c - 1) \* (c - 1) + (c - 1)) / 2 + r

End FindIndex

1. Отношение между строкой и столбцом для непустых записей в массиве раз- мером *N**N* выглядит как строка + столбец < N. Вы можете переделать уравнение для привязки строки и столбца к индексу в одномерном масси- ве, но проще будет выполнить привязку к новым строке и столбцу, кото- рые в исходном треугольном размещении элементов заполняют собой ниж- нюю левую часть. Для этого можно заменить r на N - 1 - r, как показано в нижеприведенном псевдокоде.

Integer: FindIndex(Integer: r, Integer: c) r = N - 1 - r

Return ((r – 1) \* (r - 1) + (r - 1)) / 2 + c End FindIndex

Это изменение, по сути, переворачивает массив вверх ногами так, что ма- ленькие номера строк привязываются к нижней части массива, а большие но- мера — к верхней его части. Представим, например, что *N* = 5. Тогда запись [0, 4] находится в верхнем правом углу. Эта позиция недопустима в обыч- ном нижнем левом треугольном массиве, поэтому содержимое строки меня- ется на *N* – 1 – 0 = 4. Позиция [4, 4] допускается, поскольку она находится в нижнем правом углу.

1. Следующий псевдокод заполняет массив значениями ll\_value и ur\_ value. Чтобы добиться желаемого результата, вы можете присвоить им 1 и 0.

FillArrayLLtoUR(Integer: values[,], Integer: ll\_value, Integer: ur\_value)

For row = 0 To <Upper bound for dimension 1>

For col = 0 To <Upper bound for dimension 2>

If (row >= col) Then

values[row, col] = ur\_value

Else

End If Next col

Next row End FillArrayLLtoUR

values[row, col] = ll\_value

1. Следующий псевдокод заполняет массив значениями ul\_value и lr\_value. Чтобы добиться желаемого результата, вы можете присвоить им 1 и 0.

FillArrayULtoLR(Integer: values[,], Integer: ul\_value, Integer: lr\_value)

Integer: max\_col = <Upper bound for dimension 2> For row = 0 To <Upper\_bound for dimension 1>

For col = 0 To max\_col

If (row > max\_col - col) Then values[row, col] = ul\_value

Else

End If Next col

Next row End FillArrayULtoLR

values[row, col] = lr\_value

1. Одно из решений заключается в том, чтобы присвоить каждой записи в мас- сиве минимальное значение, выбранное из индексов ее строки и столбца, а также из расстояния от нее к правому нижнему углу массива.

FillArrayWithDistances(Integer: values[,]) Integer: max\_row = values.GetUpperBound(0) Integer: max\_col = values.GetUpperBound(1)

For row = 0 To max\_row

For col = 0 To max\_col

values[row, col] =

Minimum(row, col, max\_row - row, max\_col - col)

Next col Next row

End FillArrayWithDistances

1. Ключевой момент здесь — это привязка [строка, столбец, высота] к индексам массива. Чтобы ее выполнить, программе необходимо знать, сколько ячеек содержится в полных четырехгранных и треугольных группах. Как вы зна- ете из главы 4, количество ячеек в полном треугольном массиве составля- ет (*N*2 + *N*)  2, поэтому искомое значение можно получить с помощью сле- дующего псевдокода.

Integer: NumCellsForTriangleRows(Integer: rows) Return (rows \* rows + rows) / 2

End NumCellsForTriangleRows

Получить число ячеек в полном четырехгранном массиве будет сложнее. Вы можете воспользоваться подходом, описанным в данной главе (см. таблицу 4.1 и соседние разделы), если нарисуете схему массива и посчитаете ячейки. Если

отталкиваться от того, что искомое значение зависит от количества строк, воз- веденного в куб, можно прийти к такому решению: (*N*3 + 3  *N*2 + 2  *N*) / 6. Эта формула используется в следующем псевдокоде.

Integer: NumCellsForTetrahedralRows(Integer: rows)

Return (rows \* rows \* rows + 3 \* rows \* rows + 2 \* rows) / 6 End NumCellsForTetrahedralRows

С помощью этих двух подходов можно написать метод для привязки [строка, столбец, высота] к индексам массива.

Integer: RowColumnHeightToIndex(Integer: row, Integer: col, Integer: hgt)

Return

NumCellsForTetrahedralRows(row) + NumCellsForTriangleRows(col) + hgt;

End RowColumnHeightToIndex

Приведенный код возвращает количество записей, находящихся перед задан- ной ячейкой. Для вычисления этого значения он суммирует записи согласно полным четырехгранным и треугольным группам, размещенным перед ука- занным элементом, и добавляет к полученному результату отдельные записи, которые стоят перед данным элементом в рамках его треугольной группы.

1. В массиве с разрывом пропущенные записи не занимают никакого места, поэтому этот вопрос не касается упорядочивания структуры данных для экономии дискового пространства. Все, что вам нужно сделать, — проверить при доступе к записи, является ли индекс строки больше индекса столбца.
2. Чтобы сложить два треугольных массива, достаточно просто суммировать их соответствующие элементы. Единственная хитрость здесь заключается в том, чтобы учитывать только те записи, у которых индекс строки превышает индекс столбца. Эту задачу решает следующий псевдокод.

AddTriangularArrays(Integer: array1[,], Integer: array2[,], Integer: result[,])

For row = 0 To <Upper bound for dimension 1> For col = 0 To row

Result[row, col] = array1[row, col] +

array2[row, col]

Next col

Next row End AddTriangularArrays

1. Этот псевдокод для умножения двух матриц был показан в основном тексте данной главы.

MultiplyArrays(Integer: array1[], Integer: array2[], Integer: result[]) For i = 0 To <Upper bound for dimension 1>

For j = 0 To <Upper bound for dimension 2>

// Вычисляем результат [i, j]. result[i, j] = 0

For k = 0 To <Upper bound for dimension 2>

array2[k, j]

Next j

Next i

End MultiplyArrays

Next k

result[i, j] = result[i, j] + array1[i, k] \*

Теперь рассмотрим внутренний цикл For k. Если i < k, значит запись array1[i, k] равна 0. Аналогично, если k < i, то array2[k, j] = 0. Если хотя бы одна из этих записей содержит 0, их произведение тоже бу- дет равно 0.

В следующем коде показано, каким образом можно отредактировать внутрен- нюю операцию присваивания, чтобы значение записи изменялось, только если умножаемые элементы присутствуют в обоих массивах.

If (i >= k) And (k >= j) Then

result[i, j] = result[i, j] + array1[i, k] \* array2[k, j] End If

Это можно сделать немного проще, если обратить внимание на индексы k у тех записей, которые имеются и в первом, и во втором массивах. Данные индексы существуют, если k <= i и k >= j. Вы можете использовать эти границы для индексов k во внутреннем цикле For, как показано ниже.

For k = j To i

total += this[i, k] \* other[k, j]; Next k

1. В следующем коде показан метод CopyEntries, копирующий элементы в связный список ArrayEntry, начиная с from\_entry и заканчивая по- следним элементом списка, которым в данный момент является to\_entry.

// Копируем записи в заданный список, начиная с from\_entry и заканчивая to\_entry.

CopyEntries(ArrayEntry: from\_entry, ArrayEntry: to\_entry) While (from\_entry != null)

to\_entry.NextEntry = new ArrayEntry to\_entry = to\_entry.NextEntry

to\_entry.ColumnNumber = from\_entry.ColumnNumber to\_entry.Value = from\_entry.Value to\_entry.NextEntry = null

// Переходим к следующей записи. from\_entry = from\_entry.NextEntry

End While End CopyEntries

В список to\_entry добавляются новые объекты типа ArrayEntry, пока элемент to\_entry не окажется пустым.

Метод AddEntries, показанный ниже, копирует записи из списков from\_ entry1 и from\_entry2 в итоговый список to\_entry.

// Слагаем элементы в списках from\_entry1 и from\_entry2 и сохраняем результат в итоговом списке после to\_entry.

AddEntries(ArrayEntry: from\_entry1, ArrayEntry: from\_entry2,

ArrayEntry: to\_entry)

// Повторяем до тех пор, пока в обоих списках не закончатся элементы.

While (from\_entry1 != null) And (from\_entry2 != null)

// Создаем новую итоговую запись. to\_entry.NextEntry = new ArrayEntry to\_entry = to\_entry.NextEntry to\_entry.NextEntry = null

// Ищем запись с меньшим индексом столбца. If (from\_entry1.ColumnNumber < from\_entry2. ColumnNumber) Then

// Копируем запись from\_entry1. to\_entry.ColumnNumber = from\_entry1.

ColumnNumber

to\_entry.Value = from\_entry1.Value from\_entry1 = from\_entry1.NextEntry

Else If (from\_entry2.ColumnNumber < from\_entry1. ColumnNumber)

Then

Else

End If End While

// Копируем запись from\_entry2. to\_entry.ColumnNumber = from\_entry2.

ColumnNumber

to\_entry.Value = from\_entry2.Value from\_entry2 = from\_entry2.NextEntry

// Номера столбцов одинаковые. Слагаем обе записи.

to\_entry.ColumnNumber = from\_entry1.

ColumnNumber

to\_entry.Value = from\_entry1.Value + from\_ entry2.Value

from\_entry1 = from\_entry1.NextEntry from\_entry2 = from\_entry2.NextEntry

// Добавляем содержимое списка, в котором еще остаются записи. if (from\_entry1 != null) CopyEntries(from\_entry1, to\_entry) if (from\_entry2 != null) CopyEntries(from\_entry2, to\_entry)

End AddEntries

Приведенный код перебирает оба списка from и отбирает из каждого запи- си с меньшим номером столбца. Если номера столбцов текущих элементов в обоих списках совпадают, программа создает новую запись и сохраняет туда сумму значений этих элементов.

Следующий код демонстрирует сложение двух матриц с помощью методов

CopyEntries и AddEntries, которые используются внутри метода Add.

// Слагаем два массива с разрывом, представляющие матрицы. SparseArray: Add(SparseArray: array1, SparseArray: array2)

SparseArray: result = new SparseArray

// Переменные для перебора всех массивов. ArrayRow: array1\_row = array1.TopSentinel.NextRow ArrayRow: array2\_row = array2.TopSentinel.NextRow ArrayRow: result\_row = result.TopSentinel

While (array1\_row != null) And (array2\_row != null)

// Создаем новую итоговую строку. result\_row.NextRow = new ArrayRow result\_row = result\_row.NextRow result\_row.RowSentinel = new ArrayEntry result\_row.NextRow = null

// Проверяем, какая из входящих строк имеет меньший индекс. If (array1\_row.RowNumber < array2\_row.RowNumber) Then

// array1\_row идет первой. Копируем ее значения в итоговый массив.

result\_row.RowNumber = array1\_row.RowNumber CopyEntries(array1\_row.RowSentinel.NextEntry,

result\_row.RowSentinel) array1\_row = array1\_row.NextRow

Else If (array2\_row.RowNumber < array1\_row.RowNumber) Then

// array2\_row идет первой. Копируем ее значения в итоговый массив.

result\_row.RowNumber = array2\_row.RowNumber CopyEntries(array2\_row.RowSentinel.NextEntry,

result\_row.RowSentinel) array2\_row = array2\_row.NextRow

Else

End If End While

// Номера строк совпадают. Слагаем их значения. result\_row.RowNumber = array1\_row.RowNumber AddEntries(

array1\_row.RowSentinel.NextEntry, array2\_row.RowSentinel.NextEntry, result\_row.RowSentinel)

array1\_row = array1\_row.NextRow array2\_row = array2\_row.NextRow

// Добавляем оставшиеся строки. If (array1\_row != null) Then

// Создаем новую итоговую строку. result\_row.NextRow = new ArrayRow result\_row = result\_row.NextRow result\_row.RowNumber = array1\_row.RowNumber result\_row.RowSentinel = new ArrayEntry result\_row.NextRow = null

CopyEntries(array1\_row.RowSentinel.NextEntry, result\_row.RowSentinel)

End If

If (array2\_row != null) Then

// Создаем новую итоговую строку. result\_row.NextRow = new ArrayRow result\_row = result\_row.NextRow result\_row.RowNumber = array2\_row.RowNumber result\_row.RowSentinel = new ArrayEntry result\_row.NextRow = null

CopyEntries(array2\_row.RowSentinel.NextEntry, result\_row.RowSentinel)

End If

return result End Add

Этот метод перебирает два массива from. Если текущая строка одного спи- ска имеет меньший номер, чем текущая строка другого, вызывается метод CopyEntries, который копирует записи этой строки в список to.

Если номера текущих строк в обоих списках совпадают, вызывается метод

AddEntries, который объединяет эти строки в итоговом массиве.

Когда один список from станет пустым, метод CopyEntries скопирует в итоговый массив оставшиеся элементы другого списка.

1. Для умножения двух матриц следует умножить строки первой из них на столбцы второй. Чтобы сделать это эффективно, нужно иметь возможность последовательно перебирать записи в столбцах второго массива. Массивы с пропуском, описанные в тексте данной главы, позволяют перемещаться по строкам, но не по столбцам.

Чтобы вам легче было перебирать содержимое столбцов, вы можете объеди- нить их в связный список, где в каждом столбце хранится связный список за- писей — по аналогии с тем, как создавались связные списки строк.

Однако вместо создания совершенно нового класса вы можете использовать уже имеющийся класс SparseArray. Поменяв местами строки и столбцы, вы получите аналогичный массив, позволяющий перебирать записи в столб- цах. Конечно, здесь можно запутаться, ведь класс будет работать со строка- ми, как со столбцами, и наоборот. Высокоуровневый алгоритм умножения двух матриц с пропуском показан в виде следующего псевдокода.

Multiply(SparseArray: array1, SparseArray: array2, SparseArray: result)

// Меняем местами строки и столбцы в массиве array2. SparseArray: new\_array2

For Each entry [i, j] in array2 new\_array2[j, i] = array2[i, j]

Next [i, j]

// Умножаем.

For Each row number r in array1

For Each «row» number c in array2 // В реальности это столбцы.

Integer: total = 0

For Each <k that appears in both array1’s row

and array2’s column> total = total +

<The row’s k value> \* <the column’s k value>

Next r

Next c

Next k

result[r, c] = total

End Multiply

## Глава 5. Стеки и очереди

1. Когда один из стеков полный, NextIndex1>NextIndex2. В этот момент за- полняются оба стека, NextIndex1 становится индексом верхнего элемента второго стека, а NextIndex2 — первого.
2. Просто переместите каждый элемент исходного стека в новый. Алгоритм по- казан в виде следующего псевдокода.

Stack: ReverseStack(Stack: values) Stack: new\_stack

While (<values is not empty>) new\_stack.Push(values.Pop())

End While

Return new\_stack End ReverseStack

1. Сортировка вставкой для стека демонстрируется в программе StackInsertionsort, которую можно загрузить на веб-сайте этой книги.
2. На самом деле алгоритму не обязательно возвращать все неотсортирован- ные элементы в исходный стек, поскольку он все равно будет брать их отту- да и вставлять согласно порядку сортировки. Вместо этого вы можете вер- нуть в начальный стек отсортированные записи и затем уже вставлять туда остальные элементы. Это немного ускорит алгоритм, но время его работы так или иначе будет равно *O*(*N*2).
3. Тот факт, что алгоритм сортировки вставкой работает, говорит о том, что ва- гоны поезда можно отсортировать с использованием всего двух колей — на- копительной и выходной. Накопительная колея может служить в качестве второго стека, выводя на выходную те вагоны, которые вы в данный момент сортируете (или наоборот). Это потребует больше шагов по сравнению с си- туацией, в которой накопительная колея не одна. Имеет смысл увеличить количество таких колей, поскольку перемещение железнодорожных вагонов происходит куда медленнее, чем перетягивание элементов из стека в стек.
4. Сортировка выбором для стека демонстрируется в программе StackSelectionsort, которую можно загрузить на веб-сайте этой книги.
5. Внеся небольшие изменения, вы сможете применить алгоритм сортировки выбором для упорядочивания железнодорожных вагонов. Версия, описан- ная в главе 5, хранит самый крупный элемент в отдельной переменной. Во время сортировки вагонов нельзя просто отложить некоторые из них в сто- рону. Однако его можно отправить на выходящий путь, а остальные сохра- нить на накопительной колее. Найдя вагон с наибольшим номером, вы без труда вернете его обратно на накопительную колею, переместив на выход- ной путь новый вагон.

Естественно, в случае с реальными поездами вам не обязательно ограничи- ваться передним вагоном. Прежде чем начинать какие-то перемещения, вы можете пройтись вдоль неупорядоченного состава и посмотреть, какой вагон имеет самый большой номер. Когда такой найдется, его можно будет выка- тить на выходной путь, а остальные вагоны переместить на накопительную колею. Это избавит вас от необходимости выводить на выходной путь не те вагоны и сократит длинный процесс упорядочивания.

1. Демонстрационное приложение InsertionsortPriorityQueue, доступное для загрузки на веб-сайте этой книги, реализует очередь с приоритетом, исполь- зуя связный список.
2. Демонстрационное приложение LinkedListDeque, доступное для загрузки на веб-сайте этой книги, реализует дек с помощью двунаправленного списка.
3. Работу многоголовой очереди демонстрирует приложение MultiHeadedQueue, доступное для загрузки на веб-сайте этой книги.

Среднее время ожидания сильно зависит от количества кассиров. Если их будет хотя бы на одного меньше от оптимального числа, клиентов в очереди заметно возрастет, а среднее время, которое они в ней проводят, увеличится в разы. Если добавить еще хотя бы одного кассира — очередь может почти полностью исчезнуть, а время ожидания сократится до нескольких секунд (некоторые торговые сети хорошо усвоили этот урок; как только количество ожидающих клиентов начинает превышать пару человек, открываются до- полнительные кассы и туда перебрасываются работники из других отделов, что быстро рассеивает очередь).

1. Это делает демонстрационное приложение QueueInsertionsort.
2. Это делает демонстрационное приложение QueueSelectionsort.

## Глава 6. Сортировка

Все демонстрационные приложения выводят максимум 1000 элементов, ко- торые они сортируют. Если элементов оказывается больше, все они по-прежнему обрабатываются, но в итоговом списке отображается только первая 1000.

* 1. Алгоритм сортировки вставкой реализует демонстрационное упражнение Insertionsort.
  2. Когда алгоритм начинает с индекса 0, он переходит к 0-му элементу в 0-й по- зиции, при этом ничего не меняя. Если заставить цикл For начинать с ин- декса 1 вместо 0, алгоритм, в сущности, будет относиться к первому элемен- ту как к уже отсортированному. Это разумно, поскольку группа из одного элемента сама по себе является упорядоченной.

Запуск цикла с индекса 1 не повлияет на время работы алгоритма.

* 1. Алгоритм сортировки выбором реализован в демонстрационном приложе- нии Selectionsort.
  2. Внешний цикл For мог бы остановиться перед последним элементом в мас- сиве; на последней итерации цикла элемент все равно никуда не двигается, так как алгоритм помещает его в позицию *N* – 1 с индексом *N* – 1. Изменен- ная инструкция For показана на примере следующего псевдокода.

For i = 0 To <длина значений> - 2

Это не повлияет на время работы алгоритма.

* 1. Пузырьковая сортировка осуществлена в демонстрационном приложении Bubblesort.
  2. Данные улучшения реализованы в демонстрационном приложении ImprovedBubblesort.
  3. Реализация очереди с приоритетом, подразумевающим применение кучи, де- монстрируется в приложении PriorityQueue (оно работает напрямую с мас- сивами значений и приоритетов; попрактикуйтесь, попробовав вынести код кучи в отдельный класс).
  4. На добавление элемента в кучу и удаление его оттуда потребуется время *O*(log (*N*)) — при условии, что в куче находится *N* элементов. Подробнее об этом написано при рассмотрении времени работы алгоритма древовидной сортировки.
  5. Алгоритм древовидной сортировки реализован в демонстрационном при- ложении Heapsort.
  6. Если в полном дереве степени *d* узел имеет индекс *p*, его дочерние узлы на- ходятся на позициях *d*  *p* + 1, *d*  *p* + 2 и *d*  *p* + 3, а индекс его родительско- го узла равен (*p* – 1)  *d*.
  7. Быстрая сортировка со стеками реализована в демонстрационном приложе- нии QuicksortStack*.*
  8. Быстрая сортировка с очередями реализована в демонстрационном прило- жении QuicksortQueue. С точки зрения алгоритма, реализующего быструю сортировку, стеки и очереди обеспечивают одинаковую производительность. Любые различия будут вытекать из их реализации.
  9. Алгоритм быстрой сортировки на месте осуществлен в демонстрационном приложении Quicksort.
  10. Вместо того чтобы на каждом шагу разделять элементы на две половины, рас- пределите их на три группы. В первой и последней группах будут значения, которые, соответственно, меньше и больше разделяющего элемента, а в сред- нюю группу попадут значения, которые его дублируют. Затем рекурсивно от- сортируйте первую и последнюю группы, а вторую оставьте без изменений.
  11. Алгоритм сортировки подсчетом реализован в демонстрационном прило- жении Countingsort.
  12. Выделите массив counts с индексами от 0 до 10 000. На каждой итерации перед увеличением счетчика вычтите из текущего элемента наименьшее значение в массиве (100 000). Затем, записывая подсчет обратно в исходный массив, снова прибавьте к элементам 100 000 (как вариант, можете исполь- зовать массив с ненулевыми нижними границами, описанный в главе 4).
  13. В этом случае блочная сортировка превращается в сортировку подсчетом. Алгоритму нужно упорядочить каждый блок, но все элементы любого отдель- но взятого блока будут иметь одинаковые значения. Это не проблема, если блоки имеют не слишком большой размер, однако у сортировки подсчетом все равно есть небольшое преимущество, поскольку ей достаточно просто посчитать элементы в каждом блоке.
  14. Алгоритм блочной сортировки реализован в демонстрационном приложе- нии Bucketsort.
  15. Ниже по пунктам перечисляются алгоритмы, которые хорошо подходят для заданных условий.
      + **10 значений с плавающей запятой.** Здесь подойдет любой алгоритм за ис- ключением сортировки подсчетом. Наиболее простыми и, вероятно, про- изводительными решениями в данном случае будут алгоритмы сортиров- ки вставкой, выбором и пузырьком.
      + **1000 целых чисел.** В этой ситуации подойдут сортировка с кучей и сор- тировка слиянием. Быстрая сортировка будет наиболее производитель- ным вариантом, если среди значений не встречается слишком много дубликатов и если они не отсортированы изначально; плюс следует из- бегать выбора разделяющих элементов случайным способом. Для успеш- ного применения сортировки подсчетом нужно, чтобы диапазон значе- ний был ограниченным.
      + **1000 имен.** Здесь подойдут сортировка с кучей, быстрая сортировка и сор- тировка слиянием. Быстрая сортировка будет наиболее производительным вариантом, если среди значений не встречается слишком много дубликатов и если они не отсортированы изначально; плюс следует избегать выбора разделяющих элементов случайным способом. Сортировка подсчетом не сработает. Применение блочной сортировки может оказаться затрудни- тельным (префиксное дерево, описанное в главе 10, похоже на блочную сортировку и тоже подходит для этой задачи).
      + **100 000 целых чисел со значениями от 0 до 1000.** Здесь очень хорошо по- дойдет сортировка подсчетом. Блочная сортировка тоже будет неплохим вариантом, хотя и не таким удачным. Можно также использовать сорти- ровку с кучей, быструю сортировку и сортировку слиянием, но это будет медленнее.
      + **100 000 целых чисел со значениями от 0 до 1 млрд.** Сортировка подсчетом здесь не очень подходит, потому что для хранения результатов ей пришлось бы выделить массив с 1 млрд записей. Хорошим решением для этой задачи будет блочная сортировка. Можно также использовать сортировку с ку- чей, быструю сортировку и сортировку слиянием, но это будет медленнее.
      + **100 000 имен.** Сортировка подсчетом не работает со строками. Использо- вание блочной сортировки может оказаться проблематичным (хотя, опять же, префиксное дерево, описанное в главе 10, будет работать). Хорошо с этой задачей справятся сортировка с кучей и сортировка слиянием, а са- мым производительным вариантом будет быстрая сортировка.
      + **1 млн значений с плавающей запятой.** Сортировка подсчетом не работа- ет со строками. Здесь хорошо подойдет блочная сортировка. Можно так- же использовать сортировку с кучей, быструю сортировку и сортировку слиянием, но это будет медленнее.
      + **1 млн имен.** Представленная задача плохо совместима с алгоритмами, опи- санными в текущей главе. Сортировка подсчетом не работает со строками. Использование блочной сортировки для строковых значений может ока- заться проблематичным, но выполнимым заданием. Сортировка с кучей, быстрая сортировка и сортировка слиянием тоже сработают, но медлен- но. С этой задачей также могут справиться деревья, описанные в главе 10.
      + **1 млн целых чисел с равномерным распределением.** Сортировка подсче- том может сработать, если ограничить диапазон значений. В противном

случае лучшим вариантом, вероятно, будет блочная сортировка. Можно также использовать сортировку с кучей, быструю сортировку и сортиров- ку слиянием, но это будет медленнее.

* + - **1 млн целых чисел с неравномерным распределением.** Сортировка под- счетом может сработать, если ограничить диапазон значений. С блочной сортировкой могут возникнуть проблемы из-за неравномерности распре- деления. Сортировка с кучей, быстрая сортировка и сортировка слияни- ем тоже сработают, но медленно.

## Глава 7. Поиск

1. Алгоритм линейного поиска реализован в демонстрационном приложении LinearSearch.
2. Алгоритм рекурсивного линейного поиска представлен в демонстрационном приложении RecursiveLinearSearch. Если массив содержит *N* элементов, дан- ный метод может потребовать *N* уровней рекурсии. Рекурсия большой глуби- ны нередко оказывается недопустимой для некоторых языков программиро- вания, поэтому нерекурсивный подход, скорее всего, будет более безопасным.
3. Алгоритм линейного поиска по связным спискам реализован в демонстра- ционном приложении LinearLinkedListSearch*.*
4. Алгоритм бинарного поиска представлен в демонстрационном приложении BinarySearch.
5. Алгоритм рекурсивного бинарного поиска реализован в демонстрационном приложении RecursiveBinarySearch. Для этого метода требуется стек больше- го объема, чем для нерекурсивной версии. Рекурсия большой глубины может привести к проблемам, но это актуально лишь для действительно огромных массивов, которые вряд ли встретятся на практике. В таком случае лучшим алгоритмом будет тот, который вам кажется наименее запутанным (лично я выбрал бы нерекурсивную версию).
6. Алгоритм интерполяционного поиска реализован в демонстрационном при- ложении InterpolationSearch.
7. Алгоритм рекурсивного интерполяционного поиска представлен в демон- страционном приложении RecursiveInterpolationSearch. Как и в случае с би- нарным поиском, для данного метода необходимо наличие стека большего объема по сравнению с нерекурсивной версией. Рекурсия большой глуби- ны может привести к проблемам, но это актуально лишь для чрезвычайно больших массивов, которые вряд ли встретятся на практике. В таком случае лучшим алгоритмом будет тот, который вам кажется наименее запутанным (лично я выбрал бы нерекурсивную версию).
8. Вычисления, подобные интерполяционному поиску, используются в блочной сортировке при выборе блока для каждого элемента.
9. Вы можете просто перебирать массив в обратном направлении, пока не най- дете первый элемент, не совпадающий с целевым значением. В худшем слу- чае это займет *O*(*N*) времени. Например, если программа выполняет бинар-

ный поиск по массиву, состоящий исключительно из копий целевого эле- мента, алгоритм найдет данный элемент на полпути и затем вернется в на- чало, что займет *N* / 2 = *O*(*N*) шагов.

В качестве более быстрого, но и сложного подхода можно выполнить бинар- ный или интерполяционный поиск, начиная с позиции, в которой был об- наружен первый целевой элемент, и затем найти следующее меньшее значе- ние. Это не повлияет на время работы исходного алгоритма: *O*(log (*N*)) для бинарного и *O*(log(log (*N*))) для интерполяционного поиска.

## Глава 8. Хеш-таблицы

1. Хеш-таблица с прямым связыванием реализована в демонстрационном при- ложении Chaining.
2. Хеш-таблица с прямым связыванием и отсортированными связными списка- ми представлена в демонстрационном приложении SortedChaining. В ходе одного тестирования средняя длина пробной последовательности програм- мы Chaining была равна 9,46 позициям, а аналогичный показатель програм- мы SortedChaining составил 5,55 позиций. При этом хеш-таблицы обеих про- грамм использовали 10 блоков и хранили 100 элементов.
3. На рисунке Б.6 показана средняя длина пробной последовательности для про- грамм Chaining и SortedChaining. По мере роста две кривые становятся ли- нейными; это говорит о том, что при фиксированном количестве блоков вре- мя работы обоих алгоритмов равно *O*(1). Однако приложение SortedChaining имеет лучшую производительность, поэтому во время его работы использу- ются более мелкие константы.

25.00



Средняя длина пробной последовательности

20.00

15.00

10.00

5.00

0.00

50 100 150 200 250

№ элемента

Связывание Связывание

с отсортированными

связными списками

**Рис. Б.6.** Связывание с отсортированными связными списками обеспечивает меньшую среднюю длину пробной последовательности, чем простое связывание

1. Хеш-таблица, использующая открытую адресацию с линейным пробирова- нием, реализована в демонстрационном приложении LinearProbing.
2. Хеш-таблица, задействующая открытую адресацию с квадратичным проби- рованием, представлена в демонстрационном приложении QuadraticProbing.
3. Хеш-таблица, использующая открытую адресацию с псевдослучай- ным пробированием, реализована в демонстрационном приложении PseudoRandomProbing.
4. Хеш-таблица, обращающаяся к открытой адресации с двойным хешировани- ем, реализована в демонстрационном приложении DoubleHashing.
5. Пробные последовательности, которые используются этими алгоритмами, пропускают значения, если их шаг по индексу делит таблицу размером *N* поровну. Представьте, например, что таблица имеет размер 10, а значение привязано к позиции 1 с шагом по индексу 2. Пробная последовательность посещает позиции 1, 3, 5, 7, 9 и затем повторяется. Этого не произойдет, если шаг по индексу не сможет разделить *N* поровну. Таким образом, в качестве *N* следует взять простое число.
6. Хеш-таблица, использующая открытую адресацию с упорядоченным квад- ратичным хешированием, реализована в демонстрационном приложении OrderedQuadraticHashing.
7. Хеш-таблица, прибегающая к открытой адресации с упорядоченным двойным хешированием, приведена в демонстрационном приложении OrderedDoubleHashing.
8. На рисунке Б.7 показана средняя длина пробной последовательности для разных алгоритмов открытой адресации. Все алгоритмы, не использующие упорядочивание, имеют похожую производительность. В целом самым мед- ленным является линейное пробирование, но остальные опережают его не больше чем на одну пробу. Некоторое преимущество имеет двойное хеши- рование.

9.00



8.00

Средняя длина пробной последовательности

7.00

6.00

5.00

4.00

3.00

2.00

1.00

0.00

50 60 70 80 90

Процент заполнения

Линейное пробирование Квадратичное пробирование Псевдослучайное Двойное хеширование Упорядоченное квадратичное Упорядоченное двойное

**Рис. Б.7.** Двойное хеширование имеет меньшую среднюю длину пробной последовательности, но квадратичное и псевдослучайное пробирование демонстрируют сравнимую производительность

По приведенному графику сложно сказать, но если добавить эти же значения в таблицы, этого будет достаточно, чтобы изменить расклад производитель- ности данных алгоритмов.

Алгоритмы упорядоченного квадратичного пробирования и упорядоченного двойного хеширования обеспечивают почти одинаковую среднюю длину пробной последовательности. Их показатели значительно меньше, чем у дру- гих алгоритмов, хотя добавление элементов в упорядоченные хеш-таблицы занимает больше времени.

## Глава 9. Рекурсия

1. Факториальный алгоритм реализован в демонстрационном приложении Factorial.
2. Алгоритм Фибоначчи представлен в демонстрационном приложении FibonacciNumbers.
3. Алгоритм ханойской башни реализован в демонстрационном приложении TowerOfHanoi.
4. Графическая реализация алгоритма ханойской башни представлена в демон- страционном приложении GraphicalTowerOfHanoi. Подсказки приведены ниже.
   * Создайте класс Disk для представления дисков. Оснастите его свойствами, описывающими его размер, позицию, а также список точек, обозначающих позиции, которые он должен посетить. Добавьте в него метод Draw для рисования диска и Move для перемещения диска на определенное рас- стояние по направлению к следующей точке в списке.
   * Создайте стеки, обозначающие колышки. Изначально объекты Disk долж- ны находиться в стеке первого колышка, обозначая тем самым исходную башню.
   * Для представления движений создайте класс Move. Он должен записывать номер исходного и конечного колышков, между которыми нужно переме- стить диск. Снабдите его методом MakeMovePoints, который принимает верхний диск из начального колышка в объекте Move, формирует точки передвижения объекта Disk и передвигает его в нужный стек.
   * Когда пользователь нажимает кнопку, нужно решить задачу ханойской башни, сформировав список объектов Move, представляющих собой реше- ние. Затем следует запустить таймер, который задействуют объекты Move из списка, чтобы создать точки движения для дисков, а потом перемещает и рисует эти диски с помощью методов Move и Draw из объектов Disk.
5. Черчением снежинок Коха занимается демонстрационное приложение KochSnowflake.
6. Приложение, с помощью которого пользователь может указывать углы в ге- нераторе, называется AngleSnowflake.
7. Черчение кривых Гильберта реализовано в демонстрационном приложении Hilbert. Подсказка: если ширина всей кривой должна быть равна with, уста- новите dx = width / (2depth+1 - 1).
8. В следующем псевдокоде показаны методы, которые чертят части кривой Серпинского вниз, влево и вверх.

// Чертим вниз в правой части. SierpDown(Integer: depth, Float: dx, Float: dy)

If (depth > 0) Then

depth = depth - 1 SierpDown(depth, gr, dx, dy) DrawRelative(gr, -dx, dy) SierpLeft(depth, gr, dx, dy) DrawRelative(gr, 0, 2 \* dy) SierpRight(depth, gr, dx, dy) DrawRelative(gr, dx, dy) SierpDown(depth, gr, dx, dy)

End If End SierpDown

// Чертим влево в нижней части. SierpLeft(Integer: depth, Float: dx, Float: dy)

If (depth > 0) Then

depth = depth - 1 SierpLeft(depth, gr, dx, dy) DrawRelative(gr, -dx, -dy) SierpUp(depth, gr, dx, dy) DrawRelative(gr, -2 \* dx, 0) SierpDown(depth, gr, dx, dy) DrawRelative(gr, -dx, dy) SierpLeft(depth, gr, dx, dy)

End If End SierpLeft

// Чертим вверх в левой части. SierpUp(Integer: depth, Float: dx, Float: dy)

If (depth > 0) Then

depth = depth - 1 SierpUp(depth, gr, dx, dy) DrawRelative(gr, dx, -dy) SierpRight(depth, gr, dx, dy) DrawRelative(gr, 0, -2 \* dy) SierpLeft(depth, gr, dx, dy) DrawRelative(gr, -dx, -dy) SierpUp(depth, gr, dx, dy)

End If End SierpUp

1. Черчение кривых Серпинского реализовано в демонстрационном прило- жении Sierpinski. Подсказка: если ширина всей кривой должна быть равна with, установите dx = width / (2depth + 2 - 2).
2. Следующий псевдокод чертит салфетку Серпинского. Подразумевается, что тип данных Point имеет свойства x и y.

// Чертим салфетку. SierpinskiGasket(Integer: depth,

Point: point1, Point: point2, Point: point3)

// Если глубина равна 0, заполняем оставшийся треугольник. If (depth == 0) Then

Point: points[] = { point1, point2, point3 } FillPolygon(points)

Else

// Находим точки слева, справа и снизу от треугольника.

Point: lpoint = new Point( (point1.X + point2.X) / 2, (point1.Y + point2.Y) / 2)

Point: bpoint = new Point( (point2.X + point3.X) / 2, (point2.Y + point3.Y) / 2)

Point: rpoint = new Point( (point3.X + point1.X) / 2, (point3.Y + point1.Y) / 2)

// Чертим треугольники по углам. SierpinskiGasket(depth - 1, gr, point1, lpoint, rpoint) SierpinskiGasket(depth - 1, gr, lpoint, point2, bpoint) SierpinskiGasket(depth - 1, gr, rpoint, bpoint, point3)

End If

End SierpinskiGasket

1. Следующий псевдокод чертит ковер Серпинского. Подразумевается, что тип данных Rectangle имеет свойства X и Y, Width и Height.

// Чертим ковер.

SierpinskiCarpet(Integer: depth, Rectangle: rect)

// Если глубина равна 0, заполняем оставшийся четырехугольник. If (depth == 0) Then

FillRectangle(rect)

Else

// Заполняем 8 внешних четырехугольников. Float: width = rect.Width / 3

Float: height = rect.Height / 3 For row = 0 To 2

For col = 0 To 2

// Пропускаем центральный четырехугольник. If ((row != 1) || (col != 1)) Then

SierpinskiCarpet(depth - 1, New Rectangle(

rect.X + col \* width, rect.Y + row \* height, width, height))

End If Next col

Next row

End If

End SierpinskiCarpet

1. Решение задачи о восьми ферзях представлено в демонстрационном прило- жении EightQueens.
2. Демонстрационное приложение EightQueens2 следит за тем, сколько раз было атаковано каждое поле, чтобы быстрее решить, является ли позиция для нового ферзя корректной. За один пробный запуск это снизило количество проверяемых позиций почти с 1,5 млн до 26 000 и со- кратило общее время выполнения с 2,13 до 0,07 с. Чем быстрее и эффек- тивнее вы сможете отбросить лишние варианты, тем производительнее будет ваша программа.
3. Демонстрационное приложение EightQueens3 ищет только следующий ряд для новой позиции ферзя. За один пробный запуск это снизило количество проверяемых позиций с почти 26 000 до 113 и практически свело к нулю общее время выполнения, которое ранее составляло 0,07 с. Данная программа ограничивает число возможных позиций ферзя еще сильнее, чем предыдущая версия, что позволяет ей выполнять меньше работы.
4. Демонстрационное приложение KnightsTour использует для решения задачи о ходе коня только обратное отслеживание. Наименьшая квадратная доска, на которой возможно решить эту задачу, имеет размер 55.
5. Эвристический алгоритм Варнсдорфа реализован в демонстрационном при- ложении KnightsTour2.
6. Если взять коллекцию выборок и сгенерировать для каждой из них разме- щение, получатся перестановки исходного множества. Рассмотрим, напри- мер, множество {*A*, *B*, *C*}. Выборки из двух элементов в нем выглядят так:

{*A*, *B*}, {*A*, *C*} и {*B*, *C*}. Если добавить повторное размещение этих выборок,

{*B*, *A*}, {*C*, *A*} и {*C*, *B*}, результатом будут перестановки исходного множества:

{*A*, *B*}, {*A*, *C*}, {*B*, *A*}, {*B*, *C*}, {*C*, *A*} и {*C*, *B*}.

1. Такие алгоритмы SelectKofNwithDuplicates и SelectKofNwithoutDuplicates, реализованы в демонстрационном приложении SelectKofN.
2. Такие алгоритмы, вы как PermuteKofNwithDuplicates и PermuteKofNwithoutDuplicates, рассмотрены в демонстрационном приложении Permutations.
3. Нерекурсивное вычисление факториала демонстрируется в приложении NonrecursiveFactorial.
4. Рекурсивный алгоритм Фибоначчи с сохраняемыми значениями реализован в демонстрационном приложении FastFibonacci.
5. Нерекурсивный алгоритм Фибоначчи разбирается в демонстрационном при- ложении NonrecursiveFibonacci.
6. Нерекурсивный алгоритм, который рассчитывает числа Фибоначчи до не- обходимого числа без использования глобального массива, приведен в де- монстрационном приложении NonrecursiveFibonacci2.
7. Нерекурсивный алгоритм, занимающийся построением кривой Гильберта, реализован в демонстрационном приложении NonrecursiveHilbert.

## Глава 10. Деревья

1. Нет. Количество вершин *N* в идеальном бинарном дереве с высотой *H* рав- но *N* = 2*H* + 1 = 1. Значение 2*H* + 1 кратно 2, поэтому оно всегда четное. Следо- вательно, 2*H* + 1 – 1 всегда является нечетным.
2. Полное и завершенное, но неидеальное дерево показано на рисунке Б.8.



**Рис. Б.8.** Не все полные завершенные деревья являются идеальными

1. База: если *N* = 1, дерево является корневой вершиной без ветвей, поэтому

*B* = 0. Таким образом, *B* = *N* – 1.

Шаг: рассмотрим бинарное дерево с N вершинами и предположим, что свой- ство для таких деревьев равно True. Добавляя вершину, вы должны пред- усмотреть ветвь, которая свяжет ее с деревом. Изначально количество вет- вей равно *N* – 1, поэтому новое дерево будет иметь *N* + 1 вершин и (*N* – 1) +

+ 1 = (*N* + 1) – 1 ветвей. Данное выражение подходит для свойства, предна- значенного для деревьев с *N* + 1 вершинами, следовательно, свойство оста- ется истинным.

Это доказывает, что *B* = *N* – 1.

1. Каждая вершина в бинарном дереве (за исключением корневой) крепится к родительскому элементу с помощью ветви. Количество таких вершин равно *N* – 1 (это касается всех деревьев, не только бинарных).
2. База: если *H* = 0, дерево является корневой вершиной без ветвей. В таком случае существует только одна листовая вершина, поэтому выражения *L* = 1 и *L* = 2*H* = 20 = 1 истинны.

Шаг: предположим, что свойство истинно для идеального бинарного дере- ва высотой *H*. В идеальном бинарном дереве высотой *H* + 1 содержится кор- невая вершина, соединенная с двумя идеальными бинарными поддеревья- ми высотой *H*. Поскольку мы исходим из того, что свойство истинно для де- ревьев высотой *H*, общее количество листовых вершин в каждом поддереве равно 2*H*. Добавление новой корневой вершины сверху от двух поддеревьев не приведет к появлению в дереве высотой *H* + 1 никаких новых листьев, поэтому общее количество листовых вершин составляет (2  2*H*) = 2*H* + 1. Та- ким образом, свойство остается истинным для идеальных бинарных дере- вьев высотой *H* + 1.

Это доказывает, что *L* = 2*H*.

1. База: если *H* = 1, дерево является корневой вершиной без ветвей. Этой вер- шине не хватает двух веток. Следовательно, *M* = 2 = 1 + 1, поэтому свойство *M* = *N* + 1 истинно для *N* = 1.

Шаг: предположим, что свойство истинно для бинарных деревьев с числом вершин *N*. Рассмотрим такое дерево. Для добавления новой вершины ее нуж- но соединить с родительским элементом с помощью ветви, которая заме- нит недостающую ветвь (их количество уменьшается на 1). У новой верши- ны тоже будет две недостающие ветви. Если добавить их к уже имеющимся

(*N* + 1), получится *M* = (*N* + 1) – 1 + 2 = (*N* + 1) + 1. Данное выражение под- ходит для свойства, предназначенного для деревьев с *N* + 1 вершинами, сле- довательно, свойство остается истинным.

Это доказывает, что *M* = *N* + 1.

1. Прямой обход дерева, изображенного на рисунке 10.24, выглядит так: *E*, *B*, *A*, *D*, *C*, *F*, *I*, *G*, *H*, *J*.
2. Симметричный обход для дерева, представленного на рисунке 10.24, выгля- дит так: *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*.
3. Обратный обход для дерева, изображенного на рисунке 10.24, выглядит так:

*A*, *C*, *D*, *B*, *H*, *G*, *J*, *I*, *F*, *E*.

1. Обход в глубину для дерева, приведенного на рисунке 10.24, выглядит так:

*E*, *B*, *F*, *A*, *D*, *I*, *C*, *G*, *J*, *H*.

1. Разные типы обходов для дерева, изображенного на рисунке 10.24, реализо- ваны в демонстрационном приложении BinaryTraversals.
2. Если в алгоритме обхода в глубину, описанном в разделе «Обход в глубину», применить очередь вместо стека, результат будет аналогичным обходу в об- ратном направлении, только перевернутым. То же самое можно сделать ре- курсивно, используя прямой обход, но при этом правый дочерний элемент каждой вершины должен посещаться перед левым.
3. Демонстрационное приложение TextDisplay отображает дерево, показанное на рисунке 10.25, в текстовом виде.
4. Демонстрационное приложение DrawTree воссоздает дерево, подобное тому, что изображено на рисунке 10.26.
5. Демонстрационное приложение DrawTree2 отображает дерево, подобное тому, что показано на рисунке 10.27.
6. Приведенный ниже псевдокод иллюстрирует алгоритм для выполнения об- ратного симметричного обхода в связном сортированном дереве. Отличия от прямого обхода выделены полужирным начертанием.

ReverseInorderWithThreads(BinaryNode: root)

// Начинаем с корня. BinaryNode: node = root

// Запоминаем, как мы добрались до вершины: через ветвь или тред. Делаем вид, что это была ветвь,

и, следовательно, сворачиваем дальше направо. Boolean: via\_branch = True

// Повторяем, пока не выполним обход. While (node != null)

// Если мы попали сюда через ветвь, спускаемся вниз и идем направо как можно дальше.

If (via\_branch) Then

While (node.RightChild != null) node = node.RightChild

End While

End If

// Обрабатываем эту вершину.

<Process node>

// Находим для обработки следующий узел. If (node.LeftChild == null) Then

// Используем тред. node = node.LeftThread via\_branch = False

Else

End If End While

// Используем левую ветвь. node = node.LeftChild via\_branch = True

End ReverseInorderWithThreads

1. Созданием связного упорядоченного дерева и отображением его обходов за- нимается демонстрационное приложение ThreadedTree.
2. Созданием связного упорядоченного дерева и отображением его обходов по примеру того, как показано на рисунке 10.28, занимается демонстрационное приложение ThreadedTree.
3. Все внутренние вершины дерева знаний, которое используется в игре «Жи- вотные», хранят вопросы и ведут к дочерним вершинам, поэтому они имеют степень 2. Степень всех листовых вершин равна 0. Степень 1 не встречается, поэтому дерево является полным.

Дерево растет несимметрично, в зависимости от порядка добавления живот- ных и того, какие вопросы используются для описания их отличий. Таким образом, дерево нельзя назвать ни полным, ни идеальным.

1. Вершины, представляющие вопросы, являются внутренними и имеют по два дочерних элемента. Вершины, изображающие животных, считаются листо- выми и не содержат дочерние элементы. Чтобы их отличить, можете прове- рить, обладает ли свойство вершины YesChild значением null.
2. Игра «Животные» реализована в демонстрационном приложении AnimalGame.
3. Деревья выражений показаны на рисунке Б.9.

(15/3) + (24/6)



+



/

15 3 24 6

8  12 – 14  32



–





8 12 14 32

1/2 + 1/4 + 1/20



/

 

1 2 / /

1 4 1 20

**Рис. Б.9.** Деревья в правой части рисунка представляют выражения, изображенные слева

1. Расчетом необходимых выражений занимается демонстрационное прило- жение Expressions.
2. Деревья выражений показаны на рисунке Б.10.



✓

* ((36  2)/(9  32))

/

 

36

2 9

32

5! (5–3)!  3!



/





!

!

5

–

3

5

3

sine(45°)2



Squared

Sine

45

**Рис. Б.10** Деревья выражений

1. Расчетом необходимых выражений занимается демонстрационное прило- жение Expressions2.
2. Деревья квадрантов рассматриваются в приложении Quadtree.
3. Префиксное дерево для заданных строк показано на рисунке Б.11.



*N*

*T*

40 *E*

*A P*

60

*P*

*L*

*E*

*B*

*A*

*T*

50

20 10

*E*

*AR*

30

**Рис. Б.11.** Префиксное дерево, представляющее строки *APPLE*, *APP*, *BEAR*, *ANT*, *BAT* и *APE*

1. Построение префиксного дерева и поиск по нему реализованы в демонстра- ционном приложении Trie.

## Глава 11. Сбалансированные деревья

* 1. Случай «правый — левый» изображен на рисунке Б.12.
  2. АВЛ-дерево, при добавлении к нему значений от 1 до 8 в порядке следова- ния чисел, показано на рисунке Б.13.
  3. Процесс удаления вершины 33 и повторная балансировка дерева показаны на рисунке Б.14. Сначала вам нужно заменить вершину 33 на ту, что нахо- дится правее всего по левую сторону от нее, — в данном случае это вершина

17. Дерево оказывается разбалансированным в вершине 12, поскольку вы- сота его левого и правого поддеревьев равна соответственно 2 и 0. Разбалан- сировку вызывает высокое поддерево-внук, которое состоит из вершины 8, поэтому мы имеем дело со случаем «левый — правый». Чтобы сбалансиро- вать дерево, нужно выполнить поворот влево, переместив вершину 8 на уро- вень выше, а вершину 5 — на уровень ниже. Затем следует сделать поворот вправо, продвинув вершину 8 вверх на еще один уровень и понизив на уро- вень вершину 12.



*B*

*C*

*A*

*B*1

*A*1

*A*2 *C*2



*B*

*A*

*C*

*B*1 *A*1

*A*2 *C*2



*A*

*B*

*C*

*B*1 *A*1 *A*2 *C*2

**Рис. Б.12.** Для сбалансирования АВЛ-дерева в случае «правый—левый», нужно последовательно выполнить повороты вправо и влево

1 1 2



2 1 3

2 3

1 3 2 4

4 1 5

3 3



2 5



3

2 5

1 1 6

1 4 6

7

2 5

1 4 7

6 8

**Рис. Б.13.** АВЛ-дерево остается сбалансированным, даже если добавить в него упорядоченные значения



33

12

47

5 17 35 65

8 57 71

17

12 47

5 35 65

8 57 71



17

12

47

8 35 65

5 57 71



17

8

47

5 12 35 65

57 71

**Рис. Б.14.** Для сбалансирования верхнего дерева, нужно значение 33 заменить на 17 и затем выполнить повороты влево-вправо

* 1. Процесс добавления в дерево значения 24 показан в левой части рисунка Б.15.



30

10 20 37

5

13

23

22

24



30

10 20 37

5

13

22 23

24

**Рис. Б.15.** При добавлении в дерево, представленное слева, значения 24 листовая вершина со значениями 22 и 23 разделяется

* 1. Процесс удаления из дерева значения 20 показан в левой части рисунка Б.16. Сначала значение 20 заменяется на 13. Из-за этого листовая вершина, содер- жавшая число 13, остается пустой. Балансировка проводится путем заим- ствования значения у вершины того же уровня.



30

10 20 37

5

13

22 23



30

10 13 37

5

22 23



30

10 22 37

5

13 23

**Рис. Б.16.** У вас может не получиться заимствовать значение у вершины того же уровня, если у вершины 2–3, содержимое которой удаляется, нет значений

* 1. Процесс добавления в *B*-дерево значения 56 показан в верхней части рисун- ка Б.17. Чтобы вставить новое значение, следует разделить блок, содержа- щий числа 52, 54, 55 и 58. Нужно добавить к этим значениям 56, создать два новых блока и переместить среднее значение (55) вверх к родительской вер- шине. Там для него не найдется свободного места, поэтому вам опять нуж- но будет выполнить разделение. Родительская вершина содержит значения 21, 35, 49, 55 и 60 (в том числе и новое). Числа 21 и 35 помещаются в новые блоки, а среднее значение (49) передвигается вверх к родительской верши- не, которая является корнем дерева. Таким образом, дерево становится на один уровень выше.



56

21 35 49 55

60

2 5 7 16 26 27 30 33 42 44 47 48 52 54

56 58

48

47

44

42

60

49

35

21

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 7 | 16 |

|  |  |  |  |
| --- | --- | --- | --- |
| 26 | 27 | 30 | 33 |

|  |  |  |  |
| --- | --- | --- | --- |
| 52 | 54 | 55 | 58 |

|  |  |  |  |
| --- | --- | --- | --- |
| 62 | 67 | 70 | 73 |

|  |  |  |  |
| --- | --- | --- | --- |
| 62 | 67 | 70 | 73 |

|  |  |  |  |
| --- | --- | --- | --- |
| 49 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 21 | 35 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 55 | 60 |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 7 | 16 | 26 | 27 | 30 | 33 | 42 | 44 | 47 | 48 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 52 | 54 |  |  | 56 | 58 |  |  | 62 | 67 | 70 | 73 |

**Рис. Б.17.** Иногда разделение блоков доходит до корня *B*-дерева, в результате чего оно становится выше

* 1. Чтобы удалить из дерева, показанного в нижней части рисунка Б.17, значе- ние 49, достаточно просто заменить его числом, которое находится правее всего по левую сторону от него (в данном случае это 48). Вершина, в кото- рой изначально содержалось число 48, все еще имеет три значения, поэтому ее не нужно балансировать. Результат показан на рисунке Б.18.



49



|  |  |  |  |
| --- | --- | --- | --- |
| 21 | 35 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 55 | 60 |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 7 | 16 | 26 | 27 | 30 | 33 | 42 | 44 | 47 | 48 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 52 | 54 |  |  | 56 | 58 |  |  | 62 | 67 | 70 | 73 |

|  |  |  |  |
| --- | --- | --- | --- |
| 48 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 21 | 35 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 55 | 60 |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 7 | 16 | 26 | 27 | 30 | 33 | 42 | 44 | 47 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 52 | 54 |  |  | 56 | 58 |  |  | 62 | 67 | 70 | 73 |

**Рис. Б.18.** Иногда при удалении значения повторная балансировка не требуется

* 1. На рисунке Б.19 показано *B*-дерево, растущее по мере добавления значений в диапазоне от 1 до 11. После добавления числа 11 корневая вершина имеет четыре дочерних элемента, поэтому дерево на данном этапе будет хранить 11 значений.



|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 |  |  |  |

5

8



|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 | 6 | 7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 6 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 6 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 8 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 8 | 9 | 10 |

9 10 11

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 6 | 9 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 8 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | 11 |  |  |

**Рис. Б.19.** В результате добавления 11 значений в *B*-дерево второго порядка корневой узел будет содержать четыре дочерних элемента

* 1. *B*-дерево порядка *K* заняло бы 1024  (2  *K*) + 8  (2  *K*+1) = 2048  *K* +



+ 16  *K* + 8 = 2064  *K* + 8 байт. Чтобы поместиться в четыре блока

по 2 Кб, оно не должно превышать 4  2  1024 = 8192 байт, поэтому 2064  *K* + 8  8192. Если решить это неравенство для K, получится, что *K*  (8192 – 8)  2064, или *K*  3,97. *K* должно быть целым, поэтому его сле- дует округлить до 3.

*B*+дерево порядка *K* заняло бы 100  (2  *K*) + 8  (2  *K* + 1) = 200  *K* + 16   *K* + 8 = 216  *K* + 8 байт. Чтобы поместиться в четыре блока по 2 Кб, оно не должно превышать 4  2  1024 = 8192 байт, поэтому 216  *K* + 8  8192. Если решить это неравенство для *K*, получится, что *K*  (8192 – 8) / 216, или *K*  37,9. K должно быть целым, поэтому его следует округлить до 37.

При наличии 10 000 элементов каждое дерево может иметь высоту не более log(K + 1)(10 000). Для *B*-дерева это значение равно log4(10 000) � 6,6, поэто- му оно может достигать семи уровней. Для *B*+дерева данное значение равно log38(10 000) � 2,5, поэтому количество его уровней не может превышать 3.

## Глава 12. Деревья принятия решений

1. Это делает демонстрационное приложение CountTicTacToeBoards. Оно на- ходит следующие результаты:
   * *X* выигрывает 131 184 раза;
   * *O* выигрывает 77 904 раза;
   * 46 080 раз игра завершается вничью;
   * общее количество возможных вариантов игры составляет 255 168.

Если ходы делаются случайным образом, больше побед одерживает *X*, одна- ко стратегия, которой придерживаются опытные игроки, подразумевает све- дение матчей к ничьей, что делает подобный исход наиболее частым.

1. Это делает демонстрационное приложение CountPrefilledBoards. Количе- ство возможных партий для каждого начального поля показано на рисунке Б.20. Например, игру, которую запустили в левом верхнем углу с *X*, можно провести 27 732 способами.

|  |  |  |
| --- | --- | --- |
| 27,732 | 29,592 | 27,732 |
| 29,592 | 25,872 | 29,592 |
| 27,732 | 29,592 | 27,732 |

**Рис. Б.20.** Эти числа показывают, сколько возможных вариантов игры начинается с установки *X* в соответствующем поле

Поскольку доска для игры в крестики-нолики симметрична, вам не нужно подсчитывать партии для каждой начальной позиции. Все угловые поля при- водят к одному и тому же количеству возможных партий; то же самое касает- ся средних полей. Чтобы получить все значения, достаточно посчитать пар- тии для одного углового, среднего и центрального полей.

1. Это делает демонстрационное приложение TicTacToe.
2. Это выполняет демонстрационное приложение PartitionProblem.
3. Это делает демонстрационное приложение PartitionProblem.
4. На рисунке Б.21 показаны две диаграммы. В обоих случаях график лога- рифмов посещенных вершин выглядит как почти полностью прямая линия, поэтому количество вершин, рассмотренных каждым алгоритмом, является экспоненциальной функций числа весов *N*. Другими словами, если *N* — ко- личество весов, число посещенных вершин равно *CN*. При использовании ме- тода ветвей и границ это значение окажется меньшим, чем при полном пере- боре, но оно все равно будет экспоненциальным.
5. Это делает демонстрационное приложение PartitionProblem.
6. Это выполняет демонстрационное приложение PartitionProblem.
7. Будут найдены группы {9, 6, 7, 7, 6} и {7, 7, 7, 5, 5}. Их общие веса составля- ют 35 и 31, поэтому разница равна 4.
8. Это делает демонстрационное приложение PartitionProblem.

80,000,000

30.00

70,000,000

Кол. посещенных узлов

60,000,000

50,000,000

25.00

20.00

Кол. посещенных узлов

40,000,000 15.00

30,000,000

20,000,000

10,000,000

10.00

5.00

5 7 9 11 13 15 17 19 21 23 25

Кол. разделенных весов

0.00

5 7 9 11 13 15 17 19 21 23 25

Кол. разделенных весов

**Рис. Б.21***.* Поскольку график зависимости логарифма посещенных вершин от количества весов является линейным, число посещенных вершин растет экспоненциально

по отношению к общему числу вершин

1. Будут найдены группы {5, 6, 7, 7, 7} и {5, 6, 7, 7, 9}. Их общие веса составля- ют 32 и 34, поэтому разница равна 2.
2. Будут обнаружены группы {7, 9, 7, 5, 5} и {6, 7, 7, 7, 6}. Общие веса в обоих случаях составляют 33, поэтому разница равна 0.
3. Это делает демонстрационное приложение PartitionProblem.

## Глава 13. Основные сетевые алгоритмы

1. Это делает демонстрационное приложение NetworkMaker. Выберите пункт меню Add Node и щелкните по поверхности рисования, чтобы создать но- вые узлы. При использовании любого из инструментов для добавления зве- ньев начальный узел выбирается щелчком левой клавиши мыши, а конеч- ный — правой.
2. Это выполняет демонстрационное приложение NetworkMaker.
3. Это делает демонстрационное приложение NetworkMaker.
4. Этот алгоритм не подходит для направленных сетей, поскольку в нем предпо- лагается, что если есть путь от узла *A* к узлу *B*, то должен существовать путь от узла *B* к узлу *A*. Представим, например, что в сети три узла, соединенные последовательно: *A* ➔ *B* ➔ *C*. Если алгоритм начнет в точке *A*, он доберется до всех трех узлов, но если первой выбрать точку *B*, тогда он сможет найти только лишь узлы *B* и *C*. Из-за этого он ошибочно решит, что в сети имеет- ся всего два связных компонента: {*B*, *C*} и {*A*}.
5. Это делает демонстрационное приложение NetworkMaker.
6. Если сеть состоит из *N* узлов, любое остовное дерево будет содержать ровно *N* – 1 звеньев. Если все звенья имеют одинаковые затраты *C*, общие затраты любого остовного дерева будут равны *C*  (*N* – 1).
7. Это делает демонстрационное приложение NetworkMaker.
8. Это выполняет демонстрационное приложение NetworkMaker.
9. Нет, дерево кратчайшего пути не обязательно должно быть минимальным остовным деревом. На рисунке Б.22 показан обратный пример. Слева изо- бражена исходная сеть, в центре находится дерево кратчайшего пути с кор- нем в узле *A*, а справа можно увидеть минимальное остовное дерево с тем же корневым узлом.



*A*

3

5

*B*

4

*C*



*A*

3

5

*B*

4

*C*



*A*

3

5

*B*

4

*C*

**Рис. Б.22.** Дерево кратчайшего пути всегда является остовным деревом, но не обязательно минимальным

1. Это делает демонстрационное приложение NetworkMaker.
2. Это выполняет демонстрационное приложение NetworkMaker.
3. Это делает демонстрационное приложение NetworkMaker.
4. Это выполняет демонстрационное приложение NetworkMaker.
5. Если сеть содержит цикл с общим отрицательным весом, алгоритм поиска кратчайшего пути с коррекцией меток будет следовать по нему бесконечно, уменьшая расстояние до содержащихся в нем узлов.

Этого не может произойти с алгоритмом установки меток, поскольку в нем расстояние до каждого узла назначается всего один раз и больше никогда не меняется.

1. Расстояние до узла, который помечен алгоритмом поиска кратчайшего пути с установкой меток, никогда не меняется, поэтому алгоритм можно преры- вать сразу после маркировки целевой точки. Если сеть большая, а началь- ная и конечная точки находятся близко друг к другу, это, скорее всего, сэ- кономит время.
2. Когда алгоритм поиска кратчайшего пути с коррекцией меток маркирует узел, это вовсе не означает, что он позже не изменит расстояние до него, поэтому вы не можете сразу же сказать, является ли данный путь к узлу самым коротким (как в случае с установкой меток).

Однако алгоритм не способен оптимизировать расстояние, если оно коро- че тех, что предоставляются списком потенциальных звеньев. Это означа- ет, что вы можете периодически проверять звенья в списке. Если ни одно из них не обеспечивает более короткий путь, значит вы нашли итоговый крат- чайший маршрут.

Это достаточно сложное и медленное изменение вряд ли ускорит алгоритм, разве что сеть окажется очень большой, а начальный и конечный узлы рас- полагаются совсем близко друг к другу. Вероятно, вам лучше использовать алгоритм с установкой меток.

1. В качестве корня для дерева кратчайшего пути можно выбрать не началь- ный узел, а конечный и следовать при этом по звеньям в обратном направ- лении. Такое дерево будет демонстрировать кратчайший путь от любого узла сети к финальной точке, а не от исходной точки к любому другому узлу. Если при его построении покинуть текущий кратчайший путь, оно уже будет содержать оптимальный маршрут, берущий начало в той точке, где вы находитесь.
2. Это делает демонстрационное приложение NetworkMaker.
3. Это осуществляет демонстрационное приложение NetworkMaker.
4. При построении массива на каждые 100 узлов будет уходить 1 сек. Это 16,67 мин для 1000 узлов и 11,57 дня для 10 000 узлов.
5. На рисунке Б.23 изображены массивы Distance и Via для сети, представ- ленной на рисунке 13.15. Начальный путь от узла *A* к узлу *B* выглядит как *A* ➔ *C* и имеет затраты в размере 18. Затраты итогового кратчайшего пути, *A* ➔ *B* ➔ *C*, равны 16.

*A B C D*

*A B C D*

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 10 | 18 |  |
| 10 | 0 | 6 | 11 |
| 18 | 6 | 0 | 20 |
|  | 11 | 20 | 0 |

*A B C D*

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *C* |  |
| *A* | *B* | *C* | *D* |
| *A* | *B* | *C* | *D* |
|  | *B* | *C* | *D* |

*A B C D A B C D A B C D A B C D*

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 10 | 18 |  |
| 10 | 0 | 6 | 11 |
| 18 | 6 | 0 | 20 |
|  | 11 | 20 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 10 | 16 21 | |
| 10 | 0 | 6 | 11 |
| 16  21 | 6 | 0 | 17 |
| 11 | 17 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 10 | 16 | 21 |
| 10 | 0 | 6 | 11 |
| 16 | 6 | 0 | 17 |
| 21 | 11 | 17 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 10 | 16 | 21 |
| 10 | 0 | 6 | 11 |
| 16 | 6 | 0 | 17 |
| 21 | 11 | 17 | 0 |

Оптимизация с помощью узла A

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *C* |  |
| *A* | *B* | *C* | *D* |
| *A* | *B* | *C* | *D* |
|  | *B* | *C* | *D* |

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *B B* | |
| *A* | *B* | *C* | *D* |
| *B*  *B* | *B* | *C* | *B* |
| *B* | *B* | *D* |

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *B* | *B* |
| *A* | *B* | *C* | *D* |
| *B* | *B* | *C* | *B* |
| *B* | *B* | *B* | *D* |

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *B* | *B* |
| *A* | *B* | *C* | *D* |
| *B* | *B* | *C* | *B* |
| *B* | *B* | *B* | *D* |

Оптимизация с помощью узла B

Оптимизация с помощью узла C

Оптимизация с помощью узла D

**Рис. Б.23.** Массивы и позволяют находить кратчайший путь между всеми парами вершин

Distance Via

## Глава 14. Дополнительные сетевые алгоритмы

1. Это делает демонстрационное приложение NetworkMaker.
2. Когда алгоритм вставляет узел в итоговый список, он обновляет счетчик NumBeforeMe из соседнего узла. Если этот счетчик становится равным 0, ал- горитм добавляет соседний узел в список ready. На данном этапе он также мог бы включить его в список узлов, готовых к работе. Например, при добав- лении в итоговый список узла Гипсокартон становятся готовыми узлы Про- водка и Трубопровод, поэтому они могут оказаться в списке вместе. Резуль- татом будет перечень заданий, которые завершаются в одно и то же время.
3. Алгоритм должен следить за временем с момента запуска первой задачи. Присваивая счетчику узла NumAfterMe значение 0, он должен сделать время, которое уже успело пройти, начальным. После этого он может вычислить ожидаемое время завершения работы этого узла. Когда возникает необходи- мость убрать элемент из списка готовых узлов, нужно выбрать узел с самым ранним временем завершения.
4. Да. Помимо минимум одного узла с нулевой исходящей степенью, пред- ставляющего задачу без предварительных требований, у нас должен быть по крайней мере один узел с нулевой входящей степенью, который обозначает задачу, не являющуюся предварительным требованием ни для одного друго- го узла. Вы можете добавить его в конец списка полностью упорядоченных задач. Это позволит удалить узлы из сети и вставить их в начало и конец упорядоченного списка.
5. Это делает демонстрационное приложение NetworkMaker.
6. Для узлов *M* и *N* можно использовать следующие пары: *B*/*H*, *B*/*D*, *G*/*D*, *G*/*A*, *H*/*A*, *H*/*B*, *D*/*B*, *D*/*G*, *A*/*G* и *A*/*H*. У всех них есть обратный эквивалент (на- пример, для *B*/*H* таковым является *H*/*B*), поэтому на самом деле существу- ет только пять разных вариантов.
7. Это делает демонстрационное приложение NetworkMaker.
8. Это выполняет демонстрационное приложение NetworkMaker. Запусти- те его, чтобы увидеть, какой из вариантов закраски карты четырьмя цвета- ми будет найден.
9. Это делает демонстрационное приложение NetworkMaker. Когда я его про- верял, оно использовало пять цветов для сети, показанной на рисунке 14.5, и три цвета для сети, изображенной на рисунке 14.6. Запустите его, чтобы увидеть, какие цвета оно выбирает.
10. В левой части рисунка Б.24 показана остаточная версия сети, представлен- ной на рисунке 14.12, где дополняющий путь выделен толстыми линиями. Если использовать этот путь для обновления сети, получится новый поток, изображенный справа. Это лучшее решение из возможных, поскольку общий поток равен 7 и больше из исходного узла *A* получить нельзя.

4/4 2/2



*A*

0

1

*B*

1

1

*C*

2 2

3 1

1 1

*D*

0

1

*E*

0

2

*F*

1 2

2 0

3 2

*G*

3

1

*H*

0

3

*I*



*F*



*B*



*E*



*C*



*A*

3/4



*D*

2/4

2/2

1/1 2/2

2/3



*G*

1/2

2/4 3/3



*H*

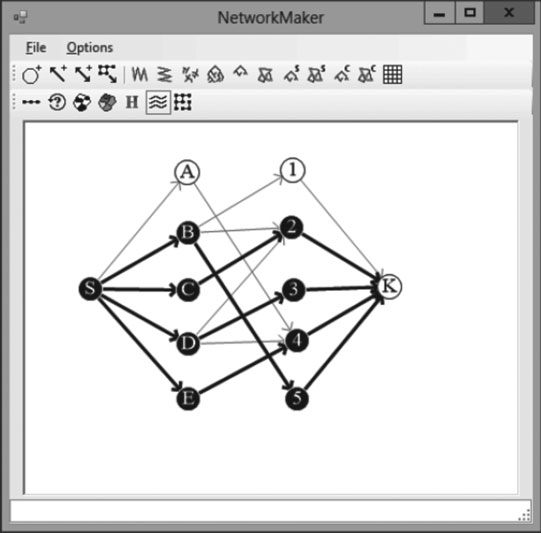
4/5

**Рис. Б.24.** Дополнительный путь в остаточной сети слева позволяет получить обновленную сеть, показанную справа



*I*

1. Это делает демонстрационное приложение NetworkMaker. Загрузите сеть и выберите инструмент Calculate Maximal Flows. Затем щелкните левой и правой клавишами мыши, чтобы выделить исходный и конечный узлы. Выведите потоки и емкость звеньев с помощью пункта Show Links Capacities в меню Options.
2. На рисунке Б.25 показан пример приложения NetworkMaker, которое выво- дит максимальные потоки для сети, представленной на рисунке 14.9. Работ- ников можно назначить всего на четыре должности. Загрузите или создайте сеть распределения рабочих мест и выберите инструмент Calculate Maximal Flows. После этого щелкните левой и правой клавишами мыши, чтобы выделить исходный и конечный узлы.



**Рис. Б.25.** Пример сети, в которой работников можно назначить максимум на четыре должности

1. Чтобы узнать, сколько путей не имеет общих звеньев, достаточно присвоить каждому звену емкость 1 и найти максимальный поток между узлами. Ито- говый поток и будет означать количество искомых путей.

Чтобы узнать, сколько путей не имеет общих звеньев и узлов, замените каж- дый узел двумя новыми — входящим и исходящим. Свяжите их звеном про- пускной способностью 1. Подсоедините звено, которое было в составе заме- ненного узла, к новому входящему узлу. Звенья, которые выходили из заме- ненного узла, теперь должны выходить из исходящего узла. Сделав все это, найдите максимальный поток. Поскольку каждый узел представлен двумя новыми узлами (одним входящим и одним исходящим), соединенными зве- ном емкостью 1, через любую такую пару может проходить только один путь.

1. Для закрашивания двудольной сети достаточно всего лишь двух цветов. Просто закрасьте первый набор узлов одним цветом, а второй другим. По- скольку узлы в одном и том же наборе не связаны между собой, двуцветное закрашивание сработает.

Для закрашивания сети распределения рабочих мест хватит трех цветов. Один цвет назначьте узлам, которые представляют работников, другой — узлам, отведенным для должностей, и еще один используйте для входяще- го и исходящего узлов.

1. Это делает демонстрационное приложение NetworkMaker. Загрузите сеть и выберите инструмент Minimal Flow Cut. Затем щелкните левой и правой кла- вишами мыши, чтобы выбрать исходный и конечный узлы.
2. Существует два решения. Первое заключается в удалении звеньев *B* ➔ *C*, *E* ➔ *F* и *H* ➔ *I*, а во втором нужно удалить звенья *C* ➔ *F*, *E* ➔ *F* и *H* ➔ *I*. В обо- их случаях общая пропуская способность уменьшается на 7. Приложение NetworkMaker находит первое решение.

## Глава 15. Строковые алгоритмы

* 1. Это делает демонстрационное приложение ParenthesisMatching.
  2. Это выполняет демонстрационное приложение EvaluateExpression.
  3. Приложению придется рассматривать четвертый рекурсивный случай, в ко- тором выражение имеет вид -*expr*.
  4. Приложению придется рассматривать другой рекурсивный случай, в кото- ром выражение имеет вид sine(*expr*).
  5. Это делает демонстрационное приложение EvaluateBoolean.
  6. Это выполняет демонстрационное приложение GraphExpression.
  7. В таблице Б.3 представлены переходы состояний для ДКА, который соот- ветствует регулярному выражению (AB)\* | (BA)\*.

**Òàáëèöà Á.3.** Ïåðåõîäû ñîñòîÿíèé äëÿ âûðàæåíèÿ (AB)\* | (BA)\*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Состояние** | 0 | 0 | 1 | 2 | 3 | 4 |
| **На входе** | *A* | *B* | *B* | *A* | *A* | *B* |
| **Новое состояние** | 1 | 3 | 2 | 1 | 4 | 3 |
| **Принимающее** | Нет | Нет | Нет | Да | Нет | Да |

* 1. Диаграмма перехода состояний показана на рисунке Б.26.



*A*

*A*

1 *B*

2

0

*B*

*B*

*A*

3 *A*

4

*B*

**Рис. Б.26.** Эта диаграмма перехода состояний для ДКА соответствует регулярному выражению ((AB)|(BA))\*

* 1. В таблице Б.4 представлены переходы состояний для диаграммы, показан- ной на рисунке Б.26.

**Òàáëèöà Á.4.** Ïåðåõîäû ñîñòîÿíèé äëÿ âûðàæåíèÿ ((*AB*)|(*BA*))\*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Состояние** | 0 | 0 | 1 | 2 | 2 | 3 | 4 | 4 |
| **На входе** | *A* | *B* | *B* | *A* | *B* | *A* | *A* | *B* |
| **Новое состояние** | 1 | 3 | 2 | 1 | 3 | 4 | 1 | 3 |
| **Принимающее** | Нет | Нет | Нет | Да | Да | Нет | Да | Да |

* 1. Это делает демонстрационное приложение DFAMatch.
  2. Таблицу легко записать, но в ней непросто найти нужные переходы. Пред- ставьте, например, что ДКА пребывает в состоянии 17 и сталкивается с вво- дом H. Программе придется искать в таблице соответствующую запись, пре- жде чем перейти к новому состоянию. Если таблица большая, на это может потребоваться какое-то время. Чтобы ускорить процесс, переходы следует хранить в дереве, хеш-таблице или другой структуре данных. Это сделает поиск более простым, но потребует больше памяти и усложнит программу.

Теперь представьте, что в качестве состояний используются объекты (по ана- логии с тем, как точки в сети обозначаются в виде объектов узлов). Каждый подобный объект может содержать список, предоставляющий новые состо- яния для разного ввода. В таком случае переходы состояний будут относи- тельно быстрыми и простыми. Если какие-то объекты способны считывать множество разных поступающих данных (то есть на диаграмме состояний из них выходит много звеньев), вам придется тратить время на поиск новых состояний. В связи с этим имеет смысл использовать дерево, хеш-таблицу или другую структуру данных, что может ускорить поиск — по крайней мере, вам не нужно будет перебирать всю таблицу переходов.

* 1. Как всегда, нужно создать НКА для распознавания шаблона, затем добавить новое начальное состояние и нулевой переход, который соединяет его с обыч- ными состояниями. Еще по одному переходу понадобится, чтобы при вводе любого символа новое исходное состояние и принимающее состояние шабло- на связывались сами с собой. Диаграмма переходов показана на рисунке Б.27.

Любой символ





**Рис. Б.27.** Диаграмма переходов НКА, ищущая шаблон по всей строке

* 1. Решение показано на рисунке Б.28.



|

+

+

*A* \* *B* \*

*B*

*A*







*A B*

   

*B*

*A*





 2 

**Рис. Б.28.** Синтаксическое дерево *(слева)* и соответствующая сеть НКА *(справа)*

для выражения (AB\*) | (BA\*)

* 1. Решение показано на рисунке Б.29.

*B A*



0

*A*

*B*

1

2

**Рис. Б.29.** Эта диаграмма переходов ДКА соответствует тем же значениям, что и диаграмма для НКА, показанная в правой части рисунка Б.28

* 1. Пусть искомая подстрока состоит из последовательности символов *A*, а текст содержит повторяющиеся цепочки из *M* – 1 символов *A*, за каждой из кото- рых следует *B*. Например, если *M* равно 4, целевая строка будет выглядеть как *AAAA*, а текст — иметь вид повторяющихся произвольное количество раз символов *AAAB*. В этом случае подстроки не может быть в тексте, поскольку в нем ни разу не встречается *M* символов *A* подряд. Пока внешний цикл из- меняет переменную i, внутренний цикл должен перебирать *M*, *M* – 1, *M* – 2… значений (пока не останется один-единственный символ), чтобы понять, что совпадение не найдено. Затем последовательность повторяется. Внутренний цикл в среднем встречается примерно *M*/2 раз, из-за чего общее время вы- полнения равно *O*(*N*  *M*/2) = *O*(*N*  *M*).
  2. Путь с наименьшими затратами использует максимально возможное количе- ство диагональных звеньев. На четыре диагональных звена приходится пять обычных, поэтому редакционное расстояние будет равно 5.
  3. Это делает демонстрационное приложение StringEditDistance.
  4. Это делает демонстрационное приложение StringEditDistance.
  5. Да, редакционное расстояние является коммутативным. Для преобразования первого слова во второе необходимо выполнить обратные операции. То есть вместо удаления символ нужно добавить, а вместо добавления — удалить.
  6. Это делает демонстрационное приложение FileEditDistance.
  7. Это делает демонстрационное приложение FileEditDistance.

## Глава 16. Шифрование

1. Это делает демонстрационное приложение RowColumnTransposition.
2. Это делает демонстрационное приложение SwapColumns.
3. Если ключ содержит дублирующиеся символы, узнать, какой из соответству- ющих столбцов должен идти первым, нельзя.

Одно из решений заключается в использовании только первого вхождения каждой буквы; таким образом, *PIZZA* превращается в *PIZA*, а *BOOKWORM* в *BOKWRM*. Это также позволяет маскировать количество столбцов в шиф- рующем массиве, что усложняет задачу для злоумышленника, который не знает, какой метод шифрования был применен.

1. Меняя местами как столбцы, так и строки, вы значительно увеличиваете ко- личество возможных вариантов их размещения. Например, если сообщение, состоящее из 100 символов, записать в массив размером 1010, получится 10!  10! � 1,31013 вариантов.

Если злоумышленнику удастся определить порядок следования столбцов, то во всех строках, хотя они и не упорядочены, будет находиться членораз- дельный текст. Распознавая отдельные слова, злоумышленник сумеет вос- становить порядок размещения столбцов и затем сможет попытаться пере- ставить строки так, чтобы получить все сообщение целиком.

Если каждая строка начинается с нового слова, восстановить исходную рас- становку строк может оказаться непросто, но если допускается перенос сло- ва на следующую строку, это даст злоумышленнику дополнительные зацеп- ки. Представьте, например, что первая строка заканчивается на GREA, а вто- рая и третья начинаются соответственно с NT и TLY. В таком случае третья строка, скорее всего, должна следовать за первой.

Это пример того, как, казалось бы, совершенно оправданная попытка совме- стить два метода шифрования оказывается практически бесполезной. И хотя перестановка строк в сочетании с перестановкой столбцов значительно уве- личивает количество возможных комбинаций, в целом данная операция мало влияет на число вариантов, которые должны интересовать злоумышленника.

1. Это делает демонстрационное приложение SwapRowsAndColumns.
2. Это делает демонстрационное приложение CaesarSubstitution. Зашифро- ванная версия сообщения *Nothingbut gibberish* со сдвигом 13 будет выглядеть как ABGUV ATOHG TVOOR EVFU.
3. Демонстрационное приложение LetterFrequencies отображает относитель- ную частоту вхождения букв в сообщение. Для трех наиболее используемых букв в криптограмме оно находит следующие значения.

**Òàáëèöà Á.5.** Îòíîñèòåëüíàÿ ÷àñòîòà âõîæäåíèÿ áóêâ â ñîîáùåíèå

|  |  |  |
| --- | --- | --- |
| **Буква** | **Частота, %** | **Сдвиг** |
| *K* | 18,5 | 6 |
| *V* | 14,8 | 17 |
| *R* | 13,0 | 13 |

Если попробовать расшифровать сообщение с помощью демонстрацион- ной программы CaesarSubstitution и сдвига 6, получится бессвязный текст. Но если указать сдвиг 17, получится следующий результат: *THERE WASAT IMEWH ENCAE SARSU BSTIT UTION WASTH ESTAT EOFTH EART* (были

времена, когда подстановка Цезаря считалась передовым методом шиф- рования).

1. Это делает демонстрационное приложение VigenereCipher. Расшифрованное сообщение будет выглядеть как *AVIGE NEREC IPHER ISMOR ECOMP LICAT EDTHA NCAES ARSUB STITU TION* (шифр Виженера превосходит по слож- ности подстановку Цезаря).
2. Если повторно использовать одноразовый блокнот, получится, в сущно- сти, шифр Виженера, в котором весь блокнот играет роль ключевого слова.

Это может сработать на некоторое время, но рано или поздно злоумышлен- ник узнает, что вы используете один и тот же блокнот, и шифр может быть взломан.

1. Это может случиться, если сообщение было утеряно или пришло не в том порядке, в результате чего вы используете в своем блокноте не те буквы. Расшифровать сообщение удастся, если подбирать разные начальные по- зиции, пока не найдется та, которая выдает членораздельный текст. Коли- чество букв в блокноте, которые необходимо пропустить, говорит о том, насколько длинным было пропущенное вами сообщение. Вы можете либо игнорировать факт пропуска, либо связаться с отправителем (зашифро- ванным образом, естественно) и попросить его повторно выслать пропу- щенный текст.
2. Несущественно. Это даст злоумышленнику информацию о длине и частоте отправляемых вами сообщений, но он и так мог это узнать, производя заме- ры перехваченного текста (все сообщения перехватываются).
3. Это делает демонстрационное приложение OneTimePad.
4. Если у вас есть криптографически безопасный генератор случайных чисел, вы можете сгенерировать с его помощью одноразовые блокноты и использо- вать их для шифрования. Такие блокноты нельзя будет взломать, поскольку каждая буква сообщения может принимать в криптограмме какой угодно вид.

Не исключен такой алгоритм действий. Представьте, что у вас есть метод шифрования, устойчивый ко взлому. Вы можете сгенерировать с его помо- щью последовательность случайных букв, зашифровав сообщение, состоя- щее из одних лишь букв *A*. Затем, используя полученные буквы, реально сге- нерировать случайные числовые значения для безопасного генератора слу- чайных чисел.

1. Каждый день в одно и то же время отправляйте сообщение одной длины. Если вам нечего сказать, напишите об этом в сообщении, а недостающую дли- ну компенсируйте за счет случайных символов. Если шифрование является безопасным, злоумышленник не сможет отличить важную информацию от набора бессмысленных букв.
2. В этой ситуации значения будут следующими:

*n* = 22 577;

(*n*) = 106  210 = 22 260;

*d* = 18 899;

*C* = 13374 199 mod 22 577 = 13 400;

*M* = 19 90518 899 mod 22 577 = 12 345.

## Глава 17. Теория вычислительной сложности

* 1. Алгоритмы сортировки подсчетом и блочной сортировки не используют сравнение. Вместо этого они ищут, в каком месте отсортированного списка должно находиться каждое значение, используя математические вычисления.
  2. Эту задачу можно свести к закрашиванию карты двумя цветами. Двухцвет- ный граф является также двудольным. Попробуйте закрасить его двумя цветами, чтобы решить задачу обнаружения двудольного графа. Если вам это удастся, граф действительно двудольный. При этом вы также получите два множества узлов. Поместите узлы первого цвета в одно множество, а узлы второго цвета — в другое.

Алгоритм закрашивания двумя цветами, описанный в главе 14, выполняется за полиномиальное время (на самом деле это довольно быстро), поэтому он, как и алгоритм обнаружения двудольного графа, принадлежит к классу *P*.

* 1. Задачу о тройных циклах можно свести к обнаружению двудольного графа. Если граф содержит цикл длиной 3, он недвудольный. Для решения подобной задачи проверьте, является ли граф двудольным, используя соответствующий алгоритм. Если ответ положительный, это означает, что граф не содержит ни одного цикла длиной 3.

Стоит отметить, что обратное утверждение не всегда является справедливым. Чтобы граф не был двудольным, в его состав должен входить любой цикл нечетной длины.

Как показано в упражнении 2, задача обнаружения двудольного графа принадлежит к классу P, поэтому тройной цикл имеет такой же класс слож- ности.

* 1. Задачу о цикле нечетной длины можно свести к обнаружению двудольного графа. Если граф содержит цикл нечетной длины, он недвудольный. Для решения такой задачи проверьте, является ли граф двудольным, используя соответствующий алгоритм. Если ответ положительный, это означает, что граф не содержит ни одного цикла с нечетной длиной.

Стоит отметить, что в данном случае обратное утверждение тоже является справедливым. Если в состав графа не входят циклы нечетной длины, он является двудольным.

Как показано в упражнении 2, задача обнаружения двудольного графа при- надлежит к классу *P*, поэтому цикл нечетной длины имеет такой же класс сложности.

* 1. Недетерминированный алгоритм для решения ЗГП (задачи о гамильтоновом пути) заключается в следующем: необходимо угадать порядок, в котором можно посетить узлы, и затем проверить (за полиномиальное время), явля- ется ли данный маршрут гамильтоновым путем. Для этого нужно опреде- лить, что все узлы в таком маршруте посещаются ровно один раз и что пары смежных узлов соединены звеньями.
  2. Вы можете воспользоваться алгоритмом, похожим на тот, что описан в реше- нии упражнения 5: необходимо угадать порядок, в котором можно посетить узлы, и затем проверить (за полиномиальное время), является ли данный маршрут гамильтоновым циклом. Единственное отличие состоит в том, что в этой задаче нужно также удостовериться, что первый и последний узлы совпадают.
  3. Задачи, описанные в предыдущих двух упражнениях, являются тесно свя- занными, но их решения нельзя считать полностью взаимозаменяемыми,

поскольку в сети с гамильтоновым путем может не быть гамильтонова цикла. Возьмем, например, сеть с двумя узлами и единственным звеном *A* ➔ *B*. Маршрут *A* ➔ *B* представляет собой гамильтонов путь, но не гамиль- тонов цикл.

Чтобы свести ЗГП к ЗГЦ, вам нужно придумать, как с помощью алгоритма ЗГЦ решать задачи о гамильтоновом пути. Другими словами, если сеть содер- жит гамильтонов путь, вы должны обнаружить его, используя алгоритм ЗГЦ.

Для начала обратите внимание на то, что сеть с гамильтоновым циклом обя- зательно содержит гамильтонов путь. Чтобы получить этот путь, просто уберите из цикла последнее звено.

Теперь представим, что в сети нет гамильтонова цикла, но есть гамильтонов путь. Как превратить путь в цикл? Нужно всего лишь соединить звеном ис- ходный узел с конечным. Сеть не может изначально содержать такое звено, иначе в ее состав входил бы гамильтонов цикл.

Чтобы обнаружить гамильтонов цикл, добавьте новое звено *LAB* между узла- ми *A* и *B*. Теперь, если цикл существует, по тому же маршруту в исходной

сети будет находиться гамильтонов путь. Предположим, что цикл проходит через узлы *N*1, *N*2, ..., *A*, *B*, *Nk*, *Nk*+1, ... *N*1. Тогда исходная сеть будет содержать гамильтонов путь *B*, *Nk*, *Nk*+1, ... *N*1, *N*2, ..., *A*.

Полная версия алгоритма для решения ЗГП выглядит так.

* + 1. Используем алгоритм ЗГЦ, чтобы узнать, содержит ли исходная сеть га- мильтонов цикл. Если ответ положительный, в ее состав также есть га- мильтонов путь.
    2. Проверяем каждую пару узлов *A* и *B*. Если между ними нет звена:
       1. добавляем звено *LAB* между *A* и *B*;
       2. проверяем с помощью алгоритма ЗГЦ, содержит ли сеть гамильтонов цикл. Если ответ положительный, в ее состав также входит гамильто- нов путь;
       3. удаляем звено *LAB* и продолжаем проверять другие пары узлов.

Общая идея показана на рисунке Б.30. Исходная сеть изображена слева. Она определенно не содержит гамильтонов цикл, поскольку узел *L* имеет всего одно звено, и путь, который через него проходит, не может пройти дальше.



*A*

*B*

*C*

*D*

*E*

*F*

*G*

*H*

*I*

*K*

*L*

*J*



*A*

*B*

*C*

*D*

*E*

*F*

*G*

*H*

*I*

*K*

*L*

*J*

**Рис. Б.30.** Сеть, изображенная слева, содержит гамильтонов путь, но не гамильтонов цикл

В процессе своей работы алгоритм сведения рано или поздно попытается добавить звено между узлами *I* и *L*. Алгоритм ЗГЦ находит гамильтонов путь, показанный в правой части рисунка Б.30. Его можно получить путем удаления звена между узлами *I* и *L* из начальной сети.

Стоит отметить, что этот алгоритм может оказаться неэффективным для нахождения гамильтоновых путей. Если сеть содержит *N* узлов, вам, воз- можно, придется повторить второй шаг *N*2 раз. Это все еще вписывается в рамки полиномиального времени, поэтому данное сведение удовлетворяет условию упражнения.

Алгоритм ЗГЦ является *NP*-полным, поэтому у него нет известных решений, которые были бы достаточно быстрыми. Это означает, что его выполнение *N*2 раз и в самом деле будет очень медленным.

* 1. Чтобы свести ЗГЦ к ЗГП, вам нужно придумать, как с помощью алгоритма ЗГП решать задачи о гамильтоновом цикле. Другими словами, если сеть со- держит гамильтонов цикл, вы должны обнаружить его, используя алгоритм ЗГП.

К сожалению, расширить гамильтонов путь до цикла можно только в том случае, если в этом пути есть звено, идущее от последнего узла к первому. Алгоритм ЗГП способен найти путь, в котором такого звена нет, поэтому он не гарантирует формирование цикла.

Допустим, сеть содержит гамильтонов цикл со звеном *LAB*, соединяющим узлы *A* и *B*. Теперь представьте, что вы соединяете узел *A*’ с узлом *A* и узел *B*’ с узлом *B*. В таком случае новая сеть будет содержать нециклический гамиль- тонов путь, ведущий от *A*’ к *B*’.

Справедливо и обратное утверждение: любой гамильтонов путь должен на- чинаться с узла *A*’ и заканчиваться узлом *B*’ (или наоборот, если сеть не- направленная). Предположим, что алгоритм ЗГП обнаруживает путь, про-

ходящий по узлам *A*’, *A*, *N*1, *N*2, ..., *B*, *B*’. Тогда путь *A*, *N*1, *N*2, ..., *B*, *A* является гамильтоновым циклом.

Полная версия алгоритма для решения ЗГЦ выглядит так. Для каждого звена *LAB*, соединяющего узлы *A* и *B*:

1. соединяем новый узел *A*’ с узлом *A* и новый узел *B*’ с узлом *B*;
2. проверяем с помощью алгоритма ЗГП, содержит ли сеть гамильтонов путь. Если ответ положительный, в ее состав также входит гамильтонов цикл;
3. удаляем новые узлы *A*’ и *B*’ вместе со звеном, которое их соединяет. Повторяем то же самое с другим звеном.

Общая идея показана на рисунке Б.31. Вам, вероятно, без особых сложностей удастся найти гамильтонов цикл, но представьте, что это не так. Как видно в правой части рисунка, алгоритм сведения проверяет звено *LQR*, соединяющее

узлы *Q* и *R*. Он добавил узел *Q*’, связанный с узлом *Q*, и узел *R*’, сведенный

с узлом *R*. Алгоритм ЗГП находит в измененной сети, что показана справа, путь, выделенный толстыми линиями. Удалив узлы *Q*’ и *R*’ вместе с их зве- ньями и добавив к пути звено *LQR*, мы получим гамильтонов цикл.



*A*

*B*

*C*

*D*

*E*

*G*

*I*

*J*

*F*

*K*

*H*

*M*

*O*

*L*

*P Q*

*N*

*R*



*A*

*B*

*C*

*D*

*E*

*G*

*I*

*J*

*F*

*K*

*H*

*M*

*O*

*L*

*P Q*

*N*

*R*

*Q'*

*R'*

**Рис. Б.31.** Гамильтонов путь, выделенный в сети справа толстыми линиями, соответствует гамильтонову циклу, представленному слева

* 1. Недетерминированный алгоритм выглядит так: выбираем цвет наугад и сле- дим за тем, чтобы у каждого узла он отличался от цветов его соседей.
  2. Недетерминированный алгоритм выглядит так: выбираем наугад подмноже- ство, добавляем к нему числа и следим за тем, чтобы их общая сумма была равна 0.
  3. Один из способов сведения задачи сообщения к алгоритму обнаружения имеет вид следующего псевдокода.
     1. Если DetectKnapsack(k) возвращает False, ни одно из подмножеств с общим значением не меньше *k* не влезет в ранец. Таким образом, алгоритм сообщения может вернуть эту информацию и остановиться.
     2. Если DetectKnapsack(k) возвращает True, выполняем для каждого объекта множества *Oi*:
        1. удаляем объект *Oi* из множества;
        2. проверяем с помощью алгоритма DetectKnapsack(k), по-прежнему ли существует решение со значением k;
        3. если решение со значением k все еще возможно, оставляем объект *Oi*

удаленным и продолжаем цикл с шага 2;

* + - 1. если решение со значением k больше не является возможным, возвра- щаем объект *Oi* во множество и продолжаем цикл с шага 2.

По завершении цикла мы получим решение, состоящее из элементов, которые остались внутри множества.

* 1. Один из способов сведения задачи оптимизации к алгоритму обнаружения имеет вид следующего псевдокода.
     1. Допустим, Kall — это суммарное значение всех объектов во множе- стве. Используем алгоритм обнаружения DetectKnapsack(Kall),

DetectKnapsack(Kall – 1), DetectKnapsack(Kall – 2) и т. д., пока DetectKnapsack(Kall – m) не вернет true. Когда это произой- дет, Kall – m будет максимальным значением, которое можно вместить в ранец. Пусть Kmax = Kall – m.

* + 1. Найдем ReportKnapsack(Kmax), воспользовавшись алгоритмом сведе- ния, описанным в решении к упражнению 11. Полученный результат бу- дет решением задачи оптимизации.
  1. Один из способов сведения задачи сообщения к алгоритму обнаружения имеет вид следующего псевдокода.
     1. Если DetectKnapsack() возвращает false, разделение на множества невозможно. Таким образом, алгоритм сообщения может вернуть эту ин- формацию и остановиться.
     2. Если DetectKnapsack() возвращает true:
        1. пусть *Omax* входит в подмножество *A* и является при этом объектом с наибольшим значением. (Если разделение возможно, *Omax* может на- ходиться в подмножестве *A*. Если после разделения объект Omax по-

пал в подмножество *B*, нужно просто поменять местами все элементы двух подмножеств);

* + - 1. выполняем для всех остальных объектов *Oi*:
         * удаляем *Oi* из множества и добавляем его значение *Vi* к значению объекта *Omax*;
         * проверяем с помощью алгоритма DetectKnapsack, по-прежнему ли возможно выполнить разделение. Если ответ положительный, разде- лить исходное множество можно при нахождении объекта *Oi* в под- множестве *A*. Оставляем множество и значение *Vmax* как есть и про- должаем цикл с шага 2*b*;
         * если алгоритм DetectPartition указывает на то, что разделе- ние больше не возможно, объект *Oi* должен быть помещен в под- множество *B*. В этом случае из *Vmax* нужно вычесть значение *Vi*, до- бавленное ранее, и вернуть *Oi* во множество, восстановив его до со- стояния, которое оно имело на шаге 2*b*-II. Чтобы представить про-

цесс помещения объекта *Oi* в подмножество *B*, удалим его из мно- жества и вычтем его значение *Vi* из *Vmax*. Теперь цикл можно про- должить с шага 2*b*.

По завершении цикла, начатого на шаге 2*b*, множество должно содержать только объект *Omax*, значение которого, *Vmax*, равно 0. Шаги, предпринятые для достижения данного результата, показывают, какие элементы принадлежат

к подмножеству *A*, а какие должны входить в подмножество *B*.

## Глава 18. Распределенные алгоритмы

1. На рисунке Б.32 показан алгоритм мгновенной сортировки, упорядочива- ющий два списка.

3, 4, 1, 2

7, 9, 6, 8

**Такт 4** 3, 5 6

7,9

**Такт 5** 3 5

7,9

**Такт 6** 3 9

7

4 8 1

6 4 8

1

5 6 4

1 8

**Такт 7**

7

3

1

9

6

5

4

8

**Такт 8**

1

7

6

3

4

9 5

8

**Такт 9** 1

6

**Такт 10** 1, 3

6

**Такт 11** 1, 3

6,7

7 4 9

3 8 5

8 5

7 4 9

9

4 8 5

**Такт 12**

1, 3, 4

6, 7

8

5

9

**Такт 13**

1, 3, 4

6, 7, 8

5

9

**Такт 14**

1, 3, 4, 5

6, 7, 8

9

**Такт 15**

1, 3, 4, 5

6, 7, 8, 9

**Рис. Б.32.** Систолический массив с четырьмя ячейками способен отсортировать два списка, содержащие по четыре числа, за 15 тактов

Сортировка двух числовых списков занимает на один такт больше, чем сор- тировка одного списка.

1. Допустим, длина общего лучшего маршрута равна 100 и процесс *A* имеет но- вый маршрут с суммарной длиной 80. Он считывает общее значение и входит в свой блок If Then. Теперь процесс *B* сравнивает общее значение с дли- ной нового маршрута (70) и тоже включает его в свой блок If Then, пер- вым захватывая мьютекс. Процесс *B* сохраняет свое решение вместе с сум- марной длиной нового маршрута (70) и освобождает мьютекс. После этого процесс *A*, захватив мьютекс, сохраняет свое решение с маршрутом, суммар- ная длина которого 80.

В таком случае решение, сохраненное процессом *A*, уступает решению, со- храненному процессом *B*. Оно совпадает с длиной маршрута, сохраненного процессом *A*, поэтому данный подход выигрывает у начального примера с со- стоянием гонки, но лучшее решение по-прежнему перезаписывается. Про- цесс *B* также хранит локальную копию длины маршрута, который он счита- ет оптимальным (70), поэтому он не сообщит о ней, если подобный марш- рут будет найден.

Процесс *A* может сохранить себя, если снова проверит общую длину после за- хвата мьютекса. Этого дополнительного шага можно избежать, захватив мью- текс раньше, но на самом деле двойная проверка значения может иметь смысл.

Захват мьютекса обычно является относительно медленной операцией, по крайней мере если сравнивать его с проверкой участка памяти. Если значе- ние процесса *A* уступает значению, которое уже находится в общей пере- менной, процесс *A* может быстро об этом узнать, не прибегая к использова- нию мьютексов.

1. Сценарий с четырьмя философами показан на рисунке Б.33. Зачеркнутые вилки являются грязными. В левой части изображения показано исходное состояние, в котором философ 4 остался без вилок, философ 1 имеет две вил- ки, а остальные держат по одной вилке в левой руке.

*P*1 *P*1 *P*1



*P*2

*P*2

*P*2

*P*4

*P*4

*P*4

**Рис. Б.33.** Согласно решению Чанди/Мисра с использованием синхронизации, философ 1 ест первым, а философ *N* — вторым

Когда все философы одновременно пытаются приступить к обеду, философ 1 уже держит в руках две вилки, поэтому он начинает есть первым. Осталь- ные (за исключением философа 4) уже держат в левой руке грязную вилку и ждут, когда освободится правая. Философ *N* остался без вилок, поэтому он просит обе — правую и левую. Поскольку все вилки грязные, тот, у кого они есть, чистит и кладет их на стол (кроме философа 1, который все еще ест). Результат показан в средней части рисунка Б.33.

Теперь в данном примере два философа имеют справа чистую вилку; левой вилки у них нет, поэтому они ее просят (в более масштабном варианте зада- чи здесь оказалось бы большинство философов). Их соседи держат чистые вилки и, следовательно, отказываются их отдавать. В итоге все ждут.

Когда философ 1 заканчивает есть, он чистит свои вилки и отдает их сосе- дям, философам 2 и 4. Результат показан в правой части рисунка Б.33. Те- перь у философа 4 есть две вилки, поэтому он приступает к обеду вторым.

Закончив прием пищи, философ 4 отдает правую вилку философу 3, который сразу же начинает есть.

Когда философ 3 закончит с едой, он отдаст правую вилку философу 2, ко- торый приступит к обеду последним.

В целом философы едят в таком порядке: 1, *N*, *N* – 1, *N* – 2, ..., 2.

1. В этом случае генерал *A* никогда не получит подтверждения. Следовательно, он решит, что количество отправленных им посланий было недостаточным, и отправит еще один набор сообщений, на этот раз больший. Некоторые из них дойдут до генерала *B*, и тот ответит новым набором подтверждений (воз- можно, увеличив их число). Рано или поздно одна из попыток увенчается успехом, и генерал *A* получит подтверждение.
2. Генерал *A*, как и прежде, шлет набор сообщений. Если генерал *B* получит хотя бы какие-то из них, он вычислит *PAB* и вернет в ответ 10 подтверждений примерно такого содержания: «*PAB* = <вычисленное значение>. Это 1 под-

тверждение из 10. Я согласен атаковать на рассвете».

Если генерал *A* получит хотя бы одно подтверждение, он вычислит *PBA*. Из содержания подтверждений он узнает, сколько равно *PAB*. На основе этого он решит, какое число сообщений ему следует отправить, чтобы добиться

желаемого уровня определенности. Сообщения будут выглядеть так: «*PBA* =

<вычисленное значение>. Подтверждение получено».

Если после первого набора сообщений генерал *A* не получит ни одного под- тверждения (когда он их отправлял, ему еще не было известно значение *PAB*, поэтому он не знал, сколько всего нужно послать сообщений), он будет

считать, что до генерала *B* не дошли новые сведения. Следовательно, он от-

правляет еще один набор, уже побольше.

Аналогично, если генерал *B* не получит ответа на свои подтверждения (на момент отправки он не знал, чему равно *PBA*, поэтому не понимал, сколько подтверждений нужно послать), то будет считать, что генерал *A* ничего не

получил. Следовательно, он отправляет еще один более крупный набор.

Рано или поздно генерал *A* пошлет достаточное количество сообщений, чтобы генерал *B* получил хотя бы пару из них и вычислил *PAB*. В какой-то момент число отправленных генералом *B* подтверждений тоже станет достаточным;

некоторые из них дойдут до генерала *A*, и тот сможет вычислить *PBA*. Наконец, однажды генерал *A* ответит столькими посланиями, что некоторые из них

достигнут генерала *B*, и тот получит значение *PBA*. После этого оба генерала будут знать вероятность прохождения сообщений в обе стороны.

1. Это решение будет работать в ситуации, показанной в правой части рисунка 18.2.

Но, согласно левой части этого рисунка, каждый лейтенант принимает соб- ственное решение. В некотором смысле это не так уж и важно, поскольку предателем является генерал, которому, если следовать условию, никто не обязан подчиняться. Однако в условии говорится, что два верных лейтенанта должны принять общее решение, чего они в данном случае не делают.

1. Это сработает в обеих ситуациях, показанных на рисунке 18.2. И там, и там лейтенанты, получив противоречивые инструкции, отступают.

Но давайте представим, что генерал остался верным (как в правой части рисунка 18.2). Он приказывает атаковать, а лейтенант-предатель, изобра- женный справа, врет об этом. В таком случае другой лейтенант получает противоречивые сведения и, вопреки приказу генерала, отступает.

1. Нет, у верных лейтенантов не хватает информации, чтобы вычислить пре- дателя. Рассмотрим в качестве примера два случая. Пусть в первом из них

предателем является генерал, который приказывает первому лейтенанту ата- ковать, а второму — отступать. Обменявшись информацией, все лейтенанты, кроме первого, будут считать, что им следует отступить.

Во втором примере предателем является первый лейтенант. Генерал при- казывает всем отступить, но предатель говорит остальным лейтенантам, что им нужно атаковать.

В обоих случаях лейтенанты получают одни и те же сведения, поэтому они не могут сказать, кто является предателем — генерал или первый лейтенант. Причем это не зависит от количества лейтенантов, следовательно, предателя невозможно обнаружить.

Кроме того, предатель может маскироваться под верного военнослужащего и не передавать противоречивых приказов. Это тоже исключит его обнару- жение, хотя так он все равно не принесет никакого вреда.

1. Представьте, что генерал приказывает двоим лейтенантам атаковать, а еще двоим — отступать. Обменявшись приказами, лейтенанты узнают о двух при- казах об атаке и еще двух — об отступлении. Один лейтенант-предатель не мог привести к такой комбинации, поэтому предателем должен быть генерал.

В этом случае лейтенанты не обязаны подчиняться предателю, но им все равно нужно выработать общее решение. Для этого они могут воспользовать- ся заранее определенным правилом, которое гласит: «Если командующий генерал окажется предателем, отступайте».

1. В задаче с обедающими философами предполагается, что разговоры за столом запрещены. Если разрешить философам общаться друг с другом, они могут выбрать лидера, который будет играть роль официанта.
2. Алгоритм хулигана позволил бы отбирать вилки у философов с меньшим идентификатором. Подобный план помог бы, если бы философы не ели слиш- ком часто. Но если они заняты обедом существенную часть времени, это может привести к активным взаимным блокировкам, которые значительно затрудняют прием пищи философам с маленькими идентификаторами.
3. Если философы хотят есть слишком часто, они, скорее всего, будут тратить много времени в ожидании вилок. Официант, который разрешает одновре- менный прием пищи поочередно философам с четными и нечетными но- мерами, будет куда более эффективным, чем тот, что позволяет приступать к обеду всем сразу.
4. Допустим, DAB и DBA — это задержки при отправке сообщения из *A* в *B* и наоборот. Рассмотрим следующие уравнения, с помощью которых алгоритм вычисляет погрешность часов:

*TB*1 = (*TA*1 + *E*) + *DAB TA*2 = (*TB*1 + *E*) + *DBA.*

Теперь, если вычесть второе уравнение из первого, получится следующее:

*TB*1 – *TA*2 = (*TA*1 + *E* + *DAB*) – (*TB*1 + *E* + *DBA*) =

= *TA*1 – *TB*1 + *DAB* – *DBA* + 2  *E.*

Когда мы выполняли предыдущий анализ, в котором *DAB* было равно *DBA*, эти два выражения сокращались. Если оставить их в уравнении и выполнить вычисление для *E*, получится следующее:

*E* = ( 2  *TB*1 – *TA*2 – *TA*1) / 2 + (*DBA* – *DAB*) / 2.

Погрешность в *E* между *DBA* и *DAB* равна половине разницы между ними. В худшем случае, когда одно из этих значений близко к 0, а другое немно- гим меньше общей задержке в обе стороны (*D*), погрешность будет *D* / 2.

После использования алгоритма часов получается, что *TB* = *TA* ± *D* / 2.

1. На самом деле нет никакой разницы между изменением скорости сети и вре- мени, которое уходит на передачу сообщения из *A* в *B* или из *B* в *A*. И в том и в другом случае важна только разница между значениями *DBA* и *DAB* из от-

вета к упражнению 13. В этом ответе было показано, что после работы алго-

ритма часов имеет место уравнение *TB* = *TA* ± *D* / 2, где *D* — общая задержка.

1. Из ответа к упражнению 14 становится понятно, что погрешность по ново- му времени процесса *B*’ определяется общей задержкой передачи сообщения в обе стороны (*D*). Если скорость сети сильно колеблется, значение *D* ино- гда будет маленьким, а иногда большим. Чтобы получить лучший результат, выполните алгоритм несколько раз и скорректируйте время в соответствии с попыткой, в которой задержка была самой маленькой.

## Глава 19. Головоломки, встречающиеся на собеседованиях

1. Если он возьмет в гостиную три носка, как минимум два из них будут чер- ными или коричневыми (все три из них не могут оказаться разного цвета).

В этой задаче нет ничего сложного, если вы сталкивались с чем-то подобным ранее, поэтому она не позволяет проверить способность мыслить логически. В любом случае она легко решается, если записать на бумаге некоторые воз- можные комбинации. В целом этот вопрос не такой уж и плохой, просто ин- тервьюер мало что может из него узнать.

1. Если положить все шарики в одну миску или распределить по разным мискам половину шариков каждого цвета, получится, что вероятность выбора белого шарика равна 50 %. Это будет нижняя граница возможной вероятности. Того же результата можно добиться, если положить все белые и черные шарики в две разные миски согласно их цвету.

Однако лучше было бы положить один белый шарик в одну миску, а все остальные — в другую. В половине случаев вам повезет — и вы выберите ми- ску с единственным белым шариком. В остальных случаях вам достанется другая миска, где на выбор белого шарика у вас будет девять шансов из 19. Общие шансы на успех следующие:

0,5 + 0,5  (9 / 19) � 74 %.

Эта задача хотя и затрагивает вероятностные вычисления, но по большей части основывается на хитроумном приеме. Было бы проще, быстрее и, скорее всего, надежнее спросить у кандидата, что он знает о вероятности.

1. Поначалу кажется, что данная задача связана с вероятностью или вычисле- ниями, но на самом деле это вопрос с подвохом. Красных шариков не хватит, чтобы поместить их между каждой парой синих, — это невозможно сделать.

Здесь все просто, если вы знаете, в чем загвоздка. Если нет, на поиск ответа можно потратить уйму времени. В этом случае интервьюер мало что узнает о кандидате.

1. Эта задача действительно основана на вычислениях. Вероятность можно найти, если поделить количество подходящих комбинаций на их общее ко- личество.

Для начала следует отметить, что вас не должны интересовать комбинации, в которых меняются местами одинаковые шарики (неважно, красные или синие).

Дальше, чтобы упростить процесс, предположим, что вы выпрямили круг в линию, в результате чего у вас получилось 12 позиций, в которых может находиться шарик. Теперь поместите синий шарик в первую позицию. Раз- мещение других 11 шариков в оставшихся 11 позициях равноценно расста- новке исходных 12 шариков в виде круга. В такой ситуации проще работать, поскольку вам не нужно беспокоиться о том, чтобы первый шарик прилегал к последнему. Сейчас между первым и последним шариками красного цвета находится синий шарик.

Теперь задачу можно свести к размещению 11 оставшихся шариков по 11 име- ющимся позициям. Если сразу разместить четыре красных шарика, остальные свободные позиции будут заполнены синими шариками. Если подойти к за- даче с этой стороны, количество возможных вариантов будет следующим.

11!

11

4

=

11!

=

30 916 800

=

= 330.

4! (11 – 4) ! 4! x 7! 2 x 435 040

Чтобы узнать, сколько всего позиций вам может подойти, начните с размеще- ния четырех красных шариков подряд. Чтобы они не оказались смежными, поместите между каждой из пар по одному синему шарику (для этого пона- добится три синих шарика из оставшихся семи).

На этом этапе у вас останется четыре синих шарика, которые можно разме- стить где угодно. Если представить красные шарики в виде перегородок, то синие должны находиться между ними (причем неважно, в каком порядке, потому что все они одинаковые).

Эту ситуацию можно представить следующим образом: у вас есть восемь яче- ек, куда следует поместить либо синий шарик, либо перегородку. Теперь вы можете поставить перегородки и заполнить оставшиеся позиции синими ша- риками. Количество способов, которыми это можно сделать, вычисляется так:

8! 8!

8

4

= =

40 320

=

= 70 .

4! (8 – 4) ! 4! x 4! 24 x 24

Количество подходящих расстановок равно 70, а всего имеется 330 возмож- ных вариантов, поэтому шансы на то, что шарики, размещенные случайным образом, окажутся в подходящих позициях, равны 70 / 330 = 7 / 33 � 21 %.

Подобное упражнение довольно сложное, хотя и интересное. Подобные за- дачи достаточно часто используются в курсах обучения, посвященных веро- ятности и статистике, поэтому кандидат, который изучал эти науки, скорее всего, будет иметь представление о том, как их решать. В любом случае, та- кие задачи требуют очень непростых вычислений, и для неподготовленного человека они будут крайне сложными.

На самом деле единственное, что интервьюер сможет узнать о кандидате благодаря данному вопросу, — это сталкивался ли тот с подобными упраж- нениями ранее и имеет ли он соответствующее образование.

1. Наиболее очевидным вариантом является использование связного списка. То же самое можно сделать и внутри массива с помощью цикла For. Вы можете получить дополнительные баллы, если упомянете оба эти подхода и заметите, что выбор лучшего из них зависит от того, каким образом список хранился изначально с расчетом на выполнение других операций.

Это действительно хороший вопрос! Возможно, вам никогда не придет- ся обращать список клиентов подобным образом, однако своим ответом вы продемонстрируете знание по меньшей мере одной структуры данных. Упоминание связного списка и массива даст понять, что вам знакомы как минимум две структуры и что вы способны не останавливаться на каком- то одном решении.

1. Подобная задача имеет несколько хороших решений. Возьмем, например, алгебраический подход. Пусть *M* — это мой текущий возраст, а *B* — возраст брата:

*M* = 3  *B;*

(*M* + 2) = 2  (*B* + 2).

Включив первое уравнение во второе, получим:

((3  *B*) + 2) = 2  (*B* + 2).

Переставив элементы, решим уравнения для *B*:

3  *B* + 2 = 2  *B* + 4;

3  *B* – 2  *B* = 4 – 2 = 2.

Если вставить это значение в первое уравнение, получится *M* = 3  2 = 6, по- этому мне сейчас шесть лет.

Второй подход заключается в создании таблицы, подобной той, что показа- на ниже. Для начала в качестве своего возраста возьмем 30 лет.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Возраст брата** | **Мой возраст** | **Возраст брата + 2** | **Мой возраст + 2** | **2**  **(возраст**  **брата + 2)** |
| 10 | 30 | 12 | 32 | 24 |

Когда заполните первую строку, сравните два последних столбца. В этом случае мой возраст + 2 оказывается слишком большим, поэтому вам следу- ет повторить попытку, понизив возрастную планку.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Возраст брата** | **Мой возраст** | **Возраст брата + 2** | **Мой возраст + 2** | **2**  **(возраст**  **брата + 2)** |
| 10 | 30 | 12 | 32 | 24 |
| 5 | 15 | 7 | 17 | 14 |

Во второй строке разница между последними двумя столбцами сократилась, но по-прежнему остается слишком большой. Пробуем еще больше умень- шить возраст.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Возраст брата** | **Мой возраст** | **Возраст брата + 2** | **Мой возраст + 2** | **2**  **(возраст**  **брата + 2)** |
| 10 | 30 | 12 | 32 | 24 |
| 5 | 15 | 7 | 17 | 14 |
| 2 | 6 | 4 | 8 | 8 |

В данном примере решение находится в третьей строке (если вы переборщите и значение в предпоследнем столбце окажется слишком маленьким, попро- буйте поднять возрастную планку).

Эта задача в некотором смысле годится для использования на собеседовании. Она позволяет проверить, какой подход кандидат считает более удобным — алгебраический или табличный. Однако мне сложно сказать, насколько по- лезной может оказаться такая информация. Многие задачи подобного рода являются более сложными, и решать их с помощью таблиц будет не очень уместно, но сам факт использования табличного метода вовсе не означает, что кандидат не сумеет применить знания алгебры, если это потребуется.

Если вы даете кандидату такую задачу, лучше всего будет его спросить, по- чему он выбрал тот или иной подход и какие еще методы решения могли бы подойти.

1. Это упражнение некоторым образом относится к программированию, по- скольку в нем используется «ошибка заборного столба». Представьте, что вы сооружаете изгородь с поперечными досками между столбами. Столбов будет на один больше, чем досок, потому что они должны размещаться на каждом конце изгороди. Ошибка (ее еще называют «ошибка на единицу») возникает в тот момент, когда программист случайно учитывает какой-то объект на единицу больше или меньше необходимого.

В контексте данного упражнения минутная стрелка обгоняет часовую чуть позже 1:05, еще немного позже 2:10 и т. д., пока не сделает это после 10:50. В следующий раз стрелки встретятся в полночь. Это означает, что пересече- ние стрелок будет происходить между 1:00 и 2:00, 2:00 и 3:00… между 10:00 и 11:00 (наверное, вы уже догадались, где здесь закралась ошибка заборного столба).

Если считать время строго между 12:00 утра и 12:00 вечера, получится 10 пересечений. Но если учитывать полдень и полночь, результат будет 12.

То, как кандидат подойдет к решению, может показать, насколько он мето- дичен, но этим польза от рассматриваемой задачи может и ограничиться. По крайней мере дотошному кандидату вовсе не обязательно знать, в чем здесь подвох, чтобы дать хотя бы какой-то ответ, пусть даже неправильный.

1. Этот вопрос использовался на собеседованиях при приеме на работу в ком- панию Microsoft. Ниже описано три разных взгляда на эту проблему.
   * Первый ответ, что приходит на ум, — это «50 %, потому что мальчики и девочки появляются на свет с одинаковой вероятностью, следовательно, в любой стране девочек среди детей будет половина». Такой подход демон- стрирует умение мыслить широко, но он не выглядит очевидным, поэтому интервьюер может попросить вас привести обоснование (на самом деле пояснение уже содержится в ответе, но дальнейшее углубление в эту тему может только сделать интервьюера более раздражительным).
   * Еще один, чуть более очевидный подход заключается в том, чтобы отдельно рассматривать первого ребенка в семье, второго, третьего и т. д. Каждая пара рожает как минимум один раз. Половина новорожденных первенцев будет состоять из мальчиков, а вторая половина — из девочек.

Пары, у которых родилась девочка, захотят второго ребенка. Мальчиков и девочек среди тех, кто родится вторым, будет поровну.

Пары с двумя девочками попробуют родить третий раз. И опять среди родившихся детей будет половина мальчиков и половина девочек.

На данном этапе уже должно быть очевидно, что равномерное распреде- ление полов присуще каждому поколению, поэтому все население в целом должно делиться поровну на мужчин и женщин.

* + К задаче можно подойти еще с одной стороны. Обратите внимание на первенцев: если у пары рождается мальчик (что происходит в 50 % случа- ев), она на этом останавливается. Но с той же вероятностью у них может родиться девочка. В среднем после рождения первенца предполагаемое количество мальчиков и девочек равно 0,5.

Если первой на свет появляется девочка, пара пытается завести второго ребенка. И опять шансы на то, что это будет девочка или мальчик, раз- делены поровну. Предполагаемый прирост в семье в этом случае равен изначальному шансу на получение второго ребенка — это 0,5 шанса ро- дить мальчика или девочку. Следовательно, в среднем семья пополняется 0,5  0,5 = 0,25 мальчиками и 0,5  0,5 = 0,25 девочками, что ведет к общему приросту двух полов, равному 0,75.

Надеюсь, вы уже уловили закономерность.

В целом, чтобы получить *N*-го ребенка, пара должна родить *N* – 1 девочек подряд. Вероятность этого равна 1/2 *N* – 1.

Если допустить, что у пары действительно родилось подряд *N* – 1 де- вочек, шансы на то, что их следующий ребенок будет иметь определен- ный пол, равны 50 %. Таким образом, пополнение в семье за счет *N*-го ребенка в среднем увеличивает количество мальчиков и девочек на 0,5  1/2*N* – 1 = 1/2*N*.

Если сложить предполагаемое количество детей вплоть до ребенка *N*, по- лучится следующее.

# Мальчики: 1/2 + 1/22 + 1/23 + ... + 1/2*N*.

# Девочки: 1/2 + 1/22 + 1/23 + ... + 1/2*N*.

Эти значения совпадают. Каждая пара, как и население в целом, имеет одинаковое предполагаемое количество мальчиков и девочек.

Если допустить, что *N* стремится к бесконечности, эти уравнения будут свидетельствовать о том, что у каждой пары в среднем должно родиться по одному мальчику и по одной девочке.

Это интересная задача, но ее довольно легко решить, если вам приходилось с ней сталкиваться. Если вы видите ее впервые, она может оказаться ко- варной. В реальности шансы на рождение мальчиков и девочек не являют- ся строго равными и даже зависят от возраста родителей. Больше об этом можно узнать в статье журнала Psychology Today под названием «Why Are Older Parents More Likely to Have Daughters» (почему у более возрастных родителей чаще рождаются дочери), которую можно найти по адресу https:// [www.psychologytoday.com/blog/the-scientific-fundamentalist/201104/why-are-older-parents-more-](http://www.psychologytoday.com/blog/the-scientific-fundamentalist/201104/why-are-older-parents-more-) likely-have-daughters.

1. Не стоит раздражать интервьюера, указывая на то, что эта ситуация нереали- стична. Задача, по сути, требует от вас разбить слиток на части, из которых можно сложить 1/7, 2/7, 3/7, 4/7, 5/7, 6/7 или весь слиток целиком.

Очевидно, что слиток можно разрезать на семь частей и выдавать по одной за каждый день работы.

Оптимального результата можно достичь несколькими способами. Во- первых, вы можете прикинуть, как будут оплачиваться услуги консультанта после каждого дня, который потенциально может стать последним.

Если работа заканчивается после первого же дня, вы должны заплатить консультанту 1/7 часть слитка, поэтому, очевидно, вам нужен кусок таких размеров.

Если консультант освобождается после двух дней, он должен получить 2/7 слитка. Вы могли бы выдать ему еще одну 1/7 часть, но в будущем это лишило бы вас пространства для маневра. Будет лучше заплатить с помощью 2/7 слитка, а 1/7 оставить себе.

Поскольку у вас есть 1/7 и 2/7 слитка, вы можете выдать их консультанту после третьего дня работы. Если бы днем ранее вы заплатили двумя частями по 1/7, у вас бы не было такой возможности. Вместо двух вам бы потребова- лось бы три куска размером 1/7 слитка.

Если отпилить от целого слитка части размером 1/7 и 2/7, у вас останется еще 4/7. Вы можете дать консультанту этот кусок, если он завершит работу через четыре дня, а остальные оставить себе.

Если на работу уйдет пять дней, вы сможете заплатить двумя кусками раз- мером 4/7 и 1/7.

После шестого дня консультант сможет получить 4/7 плюс 2/7.

И наконец, если работа займет всю неделю, вы сможете расплатиться всеми частями слитка.

Если в задаче фигурируют магические числа, которые имеют степень двой- ки или меньше на единицу под той же степенью, вам следует подумать

o двоичной системе счисления. В этом примере три куска, отпиленные от слитка, представляют 1, 2 и 4 седьмых его части. Числа с 1 по 7, которые

в двоичном виде выглядят как 001, 010, 011, 100, 101, 110 и 111, показывают, какие из кусков следует выдавать консультанту после каждого дня работы. Цифра 1 означает, что вы должны отдать соответствующий кусок, а 0 говорит о том, что вам следует оставить его у себя.

Например, 6 — это 110 в двоичной системе. То есть на шестой день консуль- танту нужно выдать куски размером 4/7 (для первой единицы) и 2/7 (для второй единицы), а 1/7 часть оставить (последний нуль). Общая оплата в этот день будет равна 4/7 + 2/7 = 6/7.

Рассматриваемая задача покажется легкой для тех, кто видел ее прежде, но кандидат, который с ней не сталкивался, может посчитать ее запутанной. Однако решение можно выработать в любом случае, если перебрать все ва- рианты, начиная с первого дня. Помочь в этом смогут знания о магических числах.

1. Это одна из целого семейства задач с чашами весов. Когда в упражнении фигурирует инструмент, такой как весы, который разделяет объекты на мно- жества, есть смысл подумать об использовании метода последовательного деления. Если на каждом шагу объекты можно делить на две группы, одна из которых впоследствии исключается из расчетов, для нахождения результата, что бы вы ни искали, подойдет двоичное последовательное деление.

Это не совсем относится к данному упражнению. Если поместить по четыре яйца на каждую чашу весов, за одно взвешивание можно будет исключить из расчетов половину яиц. Затем можно поместить на чаши оставшиеся яйца, по два с каждой стороны, и снова исключить половину из них. Все взвешивания будут потрачены, но у вас все еще останется два яйца.

Ключ к решению этой задачи лежит в понимании того, что весы определяют не только те два множества яиц, которые размещают на обеих чашах. Есть также третье множество, которое не попало ни в одну из чаш. Вместо того чтобы делить яйца поровну и исключать половину из них, вы можете раз- делить их на три части и избавиться сразу от двух из них.

Представьте, что у вас есть три яйца. Вы можете положить по одному на чаши весов, а третье убрать. Если два яйца весят одинаково, позолоченным является третье. Если одно из яиц весит меньше, вы будете знать, что оно позолоченное.

Это объясняет, как сделать итоговый выбор с помощью второго взвешивания. Но вам все еще нужно придумать, как свести число оставшихся яиц до трех (или меньше), взвесив их в первый раз.

Весы позволяют исключить две из трех групп. После первого взвешивания группа, которая вас все еще интересует, должна состоять всего лишь из трех яиц. Это означает, что группы, которые взвешиваются в первый раз, должны содержать по три яйца каждая.

Итоговое решение выглядит так.

* 1. Помещаем по три яйца на левую и правую чаши, а оставшиеся два от- кладываем.
  2. Если чаши весят одинаково:
     1. позолоченным является одно из двух яиц, которые не участвовали в первом взвешивании. Помещаем их на каждую из чаш;
     2. яйцо, которое весит меньше, будет позолоченным.
  3. Если чаши имеют разный вес:
     1. убираем содержимое более тяжелой чаши и яйца, которые не взве- шивались в первый раз;
     2. помещаем одно из оставшихся трех яиц на левую чашу и еще одно — на правую;
     3. если чаши весят одинаково, позолоченным является то, которое не взвешивалось;
     4. если чаши имеют разный вес, позолоченное будет на той из них, что весит меньше.

Стоит отметить, что если чаши весов находятся на одном уровне при пер- вом взвешивании, количество вариантов сразу сводится до двух яиц. При втором взвешивании поиск можно проводить по трем яйцам, поэтому чис- ло яиц, оставшихся после первого этапа, можно увеличить до трех. Такой подход годится даже в том случае, если изначально у вас имеется девять яиц вместо восьми.

Задача представляет определенный интерес, но для ее решения нужно знать, в чем подвох: с помощью весов объекты можно делить на три группы: те, что помещаются на левую чашу, те, что помещаются на правую чашу, и те, что не участвуют во взвешивании. В этом нет ничего сложного, если вы уже встре- чали подобные головоломки.

1. Пронумеруйте баночки от 1 до 5. Затем поместите на весы одну таблетку из баночки 1, две таблетки из баночки 2 и т. д. Если все таблетки весят одина- ково, их суммарный вес будет равен 1 + 2 + 3 + 4 + 5 = 15. Вычтите из это- го значения тот вес, который у вас получился, и поделите результат на 0,1 (разницу между настоящей таблеткой и плацебо). Так вы узнаете, сколь- ко всего таблеток плацебо находится на весах, и номер баночки, из которой они были взяты.

# ГЛОССАРИЙ

* 1. **вершина**. Вершина в 2–3-дереве, которая имеет два дочерних объекта.
  2. **вершина**. Вершина в 2–3-дереве, которая имеет три дочерних объекта.

***B*-дерево**. Сбалансированное дерево, в котором внутренние вершины (блоки) могут содержать несколько значений и соответствующих ветвей.

***B*+-дерево**. Похоже на *B*-дерево, за исключением того, что его вершины хра- нят только ключи и указатели на остальную часть данных каждой записи, а не сами данные.

**FIFO**. *См.* очередь.

**LIFO**. *См.* стек.

**NP**. Класс задач, которые могут быть решены недетерминированным компью- тером за полиномиальное время.

**NP-полный**. Задача является *NP*-полной, если к ней за полиномиальное время можно свести любую задачу класса *NP*.

**P**. Класс задач, которые могут быть решены детерминированным компьюте- ром за полиномиальное время.

**АВЛ-дерево**. Упорядоченное бинарное дерево, в котором разница в высоте двух поддеревьев в любой вершине не превышает 1.

**Адаптивная квадратура.** Подход, при котором программа определяет области, где метод аппроксимации может привести к большим отклонениям, и затем совер- шенствует свой метод в этих областях.

**Активная взаимная блокировка**. Ситуация, похожая на обычную взаимную блокировку, в которой распределенные процессы не блокируются, но все равно не могут выполнить свою работу. Причиной является способ, с помощью которого они пытаются получить доступ к ресурсам.

**Алгоритм кролика и черепахи**. Алгоритм для обнаружения и удаления циклов в связных списках (см. подраздел «Черепаха и кролик» раздела «Связные списки с циклами» в главе 3).

**Алгоритм нахождения циклов Флойда**. *См.* алгоритм кролика и черепахи.

**Алгоритм с возвратом**. Рекурсивный алгоритм, который ищет частичные ре- шения. Если частичное решение не может быть расширено до полного, оно отвер- гается, после чего алгоритм возвращается к предыдущему подходящему варианту и использует его как отправную точку для дальнейшего поиска.

**Алгоритм Форда** — **Фалкерсона**. Алгоритм для расчета максимального по- тока в сети.

**Алгоритм Евклида**. Алгоритм для быстрого нахождения наибольшего обще- го делителя.

**Алгоритм**. Рецепт выполнения определенной задачи.

**Асимптотическая производительность**. Предел производительности алгоритма, возникающий, когда масштаб задачи становится очень большим.

**Ассоциативный массив**. *См.* хеш-таблица.

**Бинарное дерево**. Дерево со степенью 2.

**Бинарный поиск**. Стратегия, при которой область поиска последовательно делится на две равные части, после чего выбирается та из них, в которой находится искомый элемент.

**Блок**. Структура данных для хранения элементов, привязанных к ней самой. В контексте хеш-таблицы в качестве блока может выступать связный список, эле- менты которого связаны между собой. В *B*-дереве блоком является внутренняя вершина, которая может хранить несколько значений и соответствующих ветвей.

**Блочный шифр**. Шифр, в котором открытый текст разбивается на блоки оди- наковой длины, которые отдельно шифруются и объединяются в единую крипто- грамму.

**Вероятностный алгоритм**. Алгоритм, который выдает правильный результат с определенной вероятностью.

**Вершина (узел)**. Объект дерева или сети, который хранит данные. Вершины соединяются ветвями (в деревьях) или звеньями/ребрами (в сетях и графах).

**Вершины одного уровня**. Две вершины дерева, которые имеют одного родителя.

**Вес**. То же самое, что затраты в сети.

**Ветвь**. Соединяет родительскую и дочернюю вершины дерева.

**Взаимная блокировка**. Ситуация, в которой каждый из двух распределенных процессов ждет, когда другой освободит мьютекс, в результате чего они блокируют друг друга.

**Взаимно простые числа**. Два целых числа являются взаимно простыми, если их наибольший общий делитель равен 1.

**Внешняя вершина**. Листовая вершина дерева.

**Внешняя сортировка**. Упорядочивание данных, которые не помещаются в па- мяти. Данные могут быть отсортированы внутри файлов на диске или ленточном накопителе.

**Внутренняя вершина**. Вершина дерева, которая имеет как минимум один до- черний объект.

**Восходящее *B*-дерево**. *B*-дерево, которое выполняет деление на блоки путем рекурсивных вызовов, продвигаясь по направлению к корню.

**Восхождение на вершину**. Эвристическая стратегия, выполняющая на каж- дом шагу такие действия, которые приближают алгоритм к лучшему решению из возможных. Это чем-то напоминает туриста, который пытается ночью отыскать вершину холма, постоянно двигаясь вверх.

**Восьмеричные деревья**. Древовидная структура данных, которая используется для поиска объектов в трехмерном пространстве.

**Вторичная кластеризация**. Эффект, при котором значения, привязываемые к одному и тому же адресу в хеш-таблице с открытой адресацией и квадратичным пробированием, выстраиваются за той же пробной последовательностью, увели- чивая ее длину.

**Входящая степень**. Количество звеньев направленной сети, входящих в узел.

**Выборка**. Неупорядоченное подмножество объектов, взятых из некоего мно- жества.

**Высота**. Длина самого длинного пути вниз от вершины до листового узла, если речь идет о вершине в дереве. Если имеется в виду само дерево — это то же самое, что высота корневой вершины.

**Генератор истинно случайных чисел**. Генератор, в котором для создания по- настоящему непредсказуемых чисел используется источник истинно случайных величин, такой как радиоактивный распад или атмосферный шум.

**Генератор псевдослучайных чисел (ГПСЧ)**. Генератор, который с помощью вычислений выдает числа, кажущиеся случайными, но на самом деле являющиеся предсказуемыми.

**Генератор**. Одна из версий кривой самоподобного фрактала, которая заменяет всех или некоторых инициаторов на каждом уровне рекурсии. Инициатор опреде- ляет исходную форму фрактала.

**ГИСЧ**. *См.* генератор истинно случайных чисел.

**Глубина**. Уровень вершины дерева.

**ГПСЧ**. *См.* генератор псевдослучайных чисел.

**Граф**. Сеть.

**Грид-вычисления**. Распределенные вычисления, использующие слабо связан- ные компьютеры, которые могут взаимодействовать через публичную сеть. Грид может включать разные виды компьютеров, работающих под управлением раз- личных операционных систем.

**Двудольная сеть**. Сеть, в которой узлы можно разделить на две группы, *A* и *B*, так, чтобы каждое звено соединяло узел из группы *A* с узлом из группы *B*.

**Двудольное паросочетание**. Процесс сопоставления узлов двудольной сети, принадлежащих к разным группам.

**Дек**. Очередь, позволяющая добавлять и удалять элементы на обоих концах.

**Декодирование**. В контексте криптографии это процесс преобразования за- шифрованного сообщения в открытый текст.

**Дерево квадрантов**. Древовидная структура данных, которая помогает искать объекты в двумерном пространстве.

**Дерево кратчайшего пути**. Остовное дерево, которое обеспечивает самый ко- роткий путь из корня к любому другому узлу сети.

**Дерево принятия решений**. Дерево, позволяющее моделировать задачи в виде последовательности решений, которые ведут к получению результата.

**Детерминированный автомат с конечным числом состояний**. *См.* детермини- рованный конечный автомат.

**Детерминированный конечный автомат (ДКА)**. Виртуальный компьютер, ко- торый следит за своими действиями с помощью набора состояний. В зависимости от своего текущего состояния, а также от ввода, который он считывает на каждом шаге, компьютер переходит к новому состоянию. Одно из состояний является ис- ходным, в нем автомат начинает свою работу. Одно или несколько состояний мо- гут быть помечены как принимающие.

**Диаграмма перехода состояний**. Сеть, представляющая переходы состояний автоматов ДКА и НКА.

**Дополняющий путь**. Путь, пролегающий через остаточную сеть, который улуч- шает решение с максимальным потоком.

**Доступный узел**. Узел *B* доступен из узла *A*, если в сети существует путь от узла *A* к узлу *B*.

**Дочерний объект**. Дочерняя вершина дерева, соединенная со своим родителем.

Обычно изображается снизу от родительского объекта.

**Емкость (пропускная способность)**. В контексте сети это максимальный объ- ем чего-либо, который может пройти через узел или звено. Таковым является мак- симальный объем тока, который можно пропустить по проводу в электросети, или наибольшее количество автомобилей, которые способны проехать по улице за какую-то единицу времени.

**Завершенное дерево**. Дерево, в котором каждый уровень является абсолютно полным, за исключением, возможно, нижнего, в котором все вершины сдвигаются влево настолько, насколько это возможно.

**Задача о византийских генералах**. Задача, в которой группа генералов, среди которых могут быть предатели, должна прийти к единому решению относительно какого-то действия. Каждый верный генерал обладает частью информации и должен получить сведения от других верных генералов.

**Задача о восьми ферзях**. Необходимо расставить на шахматной доске восемь ферзей так, чтобы ни один из них не мог атаковать других.

**Задача о генерале и лейтенантах**. Задача, в которой генерал отдает приказ своим лейтенантам, при этом любой участник (включая самого генерала) может оказаться предателем. Цель верных лейтенантов — договориться об общих действи- ях. Они должны совпадать с приказом генерала, если тот не предатель.

**Задача о двух генералах**. В этой задаче у двух генералов есть армии, дисло- цирующиеся рядом с вражеским городом, по разные стороны от него. С помощью связных, которые могут быть перехвачены противником, генералы должны ско- ординировать атаку.

**Задача о ходе коня**. Конь должен ровно один раз посетить каждую клетку на шахматной доске. Ход считается завершенным, если окончательная позиция нахо- дится в одном шаге от начальной позиции. Незавершенный ход считается открытым.

**Задача обедающих философов**. За столом сидят *N* философов, между кажды- ми двумя из них лежит вилка. Чтобы приступить к обеду, философ должен взять вилки, лежащие рядом по обе стороны от него, и при этом не переговариваться со своими соседями.

**Задача оптимизации**. Задача, в которой нужно найти наилучшее решение опре- деленной проблемы. Задачи оптимизации часто имеют приблизительные решения.

**Закон Мура**. Тенденция, описанная Гордоном Э. Муром в 1965 г., согласно которой количество транзисторов в интегральных схемах удваивается примерно каждые два года.

**Затраты**. Звено в сети может иметь сопутствующие затраты. В отдельных слу- чаях это относится и к узлам.

**Злоумышленник**. В контексте криптографии это человек, который пытается пе- рехватить и расшифровать сообщение, посланное отправителем в адрес получателя.

**Идеальное дерево**. Полное дерево, все листовые вершины которого находятся на одном уровне.

**Иерархически организованное *B*-дерево**. *B*-дерево, которое при движении вниз по дереву в поисках места для нового значения выполняет разделение бло- ков (везде, где это возможно).

**Имитация отжига**. Метод эвристического улучшения результата, начина- ющийся с внесения крупных изменений, которые постепенно становятся все мень- ше и меньше. Каждое изменение делается в попытке улучшить решение.

**Инициатор**. Кривая, определяющая базовую форму для одного типа фракта- лов. На каждом уровне рекурсии все или часть инициаторов заменяются кривой- генератором.

**Интеграция Монте-Карло**. Численный метод интеграции, при которой про- грамма, генерируя множество псевдослучайных точек, а затем, определяя, какие из них находятся внутри целевой области, оценивает искомую площадь фигуры.

**Исходящая степень**. Количество звеньев направленной сети, выходящих из узла.

**Исчерпывающий поиск**. Перебор всех возможных элементов для нахождения искомого. Перебор всех возможных решений для нахождения лучшего из них.

**Квадратура**. См. *численное интегрирование*.

**Квантовый компьютер**. Компьютер, использующий для работы с данными та- кие квантовые эффекты, как запутанность и суперпозиция.

**Кластерные вычисления**. Распределенные вычисления, в которых используется набор из тесно связанных между собой компьютеров, часто находящихся в одной внутренней сети или в сети специального назначения.

**Ключ**. С точки зрения криптографии это фрагмент информации, который позволяет получателю расшифровывать закодированное сообщение. В современ- ной криптографии изначально подразумевается, что злоумышленнику известен метод шифрования, поэтому расшифровать сообщение может любой, кто владеет ключом.

**Коллизия**. В хеш-таблицах возникновение коллизий вызвано привязкой зна- чений к позициям, которые уже заняты.

**Кольцевой массив**. Массив для хранения очереди, в которой последний эле- мент должен идти сразу перед первым.

**Кольцевой связный список**. Связный список, в котором последняя ссылка ука- зывает обратно на первый элемент.

**Корень**. Корнями уравнения y = f(x) являются значения *x*, для которых

f(x) = 0.

**Корневая вершина**. Уникальная вершина, находящаяся в самом верху дерева и не имеющая родителя.

**Кривая Гильберта**. Фрактальная кривая, заполняющая пространство. Начи- нается с кривой инициатора, части которой рекурсивно заменяются на генератор, масштабированный, повернутый и преобразованный должным образом.

**Кривая Коха**. Самоподобный фрактал, начинающийся с кривой инициатора, части которой рекурсивно заменяются кривой генератора, масштабированной, по- вернутой и преобразованной должным образом.

**Кривая Серпинского**. Фрактал, заполняющий пространство; начинается с кри- вой инициатора, части которой рекурсивно заменяются на генератор, масштабиро- ванный, повернутый и преобразованный должным образом.

**Криптоанализ**. Исследование методик, используемых злоумышленниками для взлома систем шифрования.

**Криптограмма**. В контексте криптографии это сообщение, зашифрованное с це- лью безопасной передачи.

**Криптография**. Исследование методик шифрования и расшифровывания, с по- мощью которых отправитель может безопасно передавать информацию получателю, исключая возможность восстановления исходного сообщения злоумышленником.

**Куча**. Завершенное бинарное дерево, значение каждой вершины которого по крайней мере не меньше значений всех ее дочерних объектов.

**Линейное пробирование**. Методика построения хеш-таблиц с политикой разрешения коллизий, которая предусматривает добавление числовой констан- ты (обычно 1) к каждой занятой позиции с целью создания пробной последова- тельности.

**Линейный конгруэнтный генератор**. Генератор псевдослучайных чисел, кото- рый в своей работе использует рекуррентное уравнение.

**Линейный массив**. Одномерный массив.

**Линейный поиск**. Поиск значения в линейном массиве или другой линейной структуре данных.

**Листовая вершина**. Вершина дерева, которая не имеет дочерних объектов.

**Массив с пропуском**. Структура данных для построения массивов, в которой находится очень мало записей без значений по умолчанию.

**Массив**. Непрерывный участок памяти, доступ к которому можно осущест- влять с помощью индексов — по одному на измерение.

**Матрица смежности**. Матрица, представляющая связи между узлами в сети.

**Машина Тьюринга**. Гипотетический компьютер, который работает с символа- ми на входящей ленте в соответствии с простой таблицей правил.

**Метод ветвей и границ**. Древовидный алгоритм поиска, который спускается по ветви и затем решает, возможно ли улучшить текущее решение до такой степе- ни, чтобы оно стало лучшим, найденным на этот момент. Если этого сделать нель- зя, алгоритм прекращает работу с данной частью дерева.

**Метод Ньютона**. Способ нахождения корней функции.

**Метод Ньютона** — **Рафсона**. *См.* метод Ньютона.

**Минимальное остовное дерево**. Основное дерево, которое имеет наименьшие возможные затраты в сети.

**Множественная рекурсия**. Возникает, когда метод вызывает себя больше одно- го раза.

**Моделирование по методу Монте-Карло**. Вероятностная методика, при кото- рой программа выбирает псевдослучайные значения и определяет, какой процент из них удовлетворяет неким критериям. Это позволяет оценить общее количество значений, подпадающих под эти условия.

**Мьютекс**. Метод, гарантирующий, что в любой отдельно взятый момент вре- мени операция выполняется только одним процессом (название происходит от вы- ражения *mutual exclusion* — взаимное исключение).

**Нагруженная сеть**. Сеть, узлы которой имеют максимальную емкость.

**Наибольший общий делитель (НОД)**. Наибольшее целое число, которое де- лит без остатка два других целых числа.

**Наименьший общий предок**. *См.* первый общий предок.

**Направленный**. Звено сети является направленным, если по нему можно прой- ти только в одном направлении. Направленной называется сеть, которая состоит исключительно из направленных звеньев.

**Натуральное число**. Целое число, которое больше 0.

**Недетерминированный конечный автомат (НКА)**. То же самое, что ДКА, толь- ко для одного и того же ввода из состояния может исходить множество ссылок. Другими словами, НКА пребывает одновременно в каждом состоянии, возмож- ном для его ввода.

**Ненаправленный**. Звено сети является ненаправленным, если по нему можно пройти в любом направлении. Ненаправленной называется сеть, которая состоит исключительно из ненаправленных звеньев.

**Непрямая рекурсия**. Возникает, когда метод вызывает себя не напрямую, а че- рез промежуточный метод.

**Обманщик Ферма**. Если *p* не является простым, а значение *n* в диапазоне 1  *n*  *p* удовлетворяет уравнению *np* – 1 mod *p* = 1, тогда *n* называют обманщиком Ферма, поскольку оно делает ложное предположение о простоте *p*.

**Обнаружение циклов**. Процесс определения наличия циклов в сети.

**Обозначение «О» большое**. Время выполнения алгоритма будет равно *O*(*g*(*N*)) («О» большое), если функция *g*(*N*) является его верхним пределом.

**Обозначение «Омега» большое**. Время выполнение алгоритма равно (*g*(*N*)) («Омега» большое), если функция *g*(*N*) является его нижним пределом.

**Обозначение «Тета» большое**. Время выполнение алгоритма равно (*g*(*N*)) («Тета» большое), если функция *g*(*N*) является одновременно его нижним и верх- ним пределом.

**Обратный обход**. Обход дерева или сети, при котором сначала посещается ле- вый дочерний объект вершины, потом правый дочерний объект и затем сама вер- шина.

**Обход в глубину**. Обход, при котором некоторые вершины дерева или сети, расположенные далеко от начальной точки, посещаются раньше тех, что находят- ся к ней ближе всего.

**Обход в ширину**. Обход дерева или сети, при котором в первую очередь посе- щаются все дочерние элементы узла. Это позволяет посетить вершины, находящи- еся недалеко от корня, и только затем идти дальше.

**Обход**. Посещение всех узлов дерева или сети в определенном порядке и вы- полнение с ними каких-то действий.

**Ограничитель**. Ячейка, которая является частью связного списка, находится на одном из его концов, но не содержит никаких ценных данных. Она использует- ся как заполнитель, чтобы алгоритмы могли ссылаться на ячейку перед первой по- зицией или после последней.

**Одиночная рекурсия**. Рекурсия, которая возникает, когда метод вызывает себя ровно один раз.

**Остаточная пропускная способность**. Дополнительный поток, который мож- но добавить к звену в нагруженной сети.

**Остаточная сеть**. Сеть, прямые и обратные звеня которой помечены величи- ной их остаточной пропускной способности.

**Остовное дерево**. Дерево, содержащее узлы и звенья, которые соединяют каж- дый узел сети.

**Открытая адресация**. Метод построения хеш-таблиц, в котором ключи привя- зываются к записям массива. В разных версиях открытой адресации используются разные функции хеширования и политики разрешения коллизий.

**Открытый текст**. С точки зрения криптографии это сообщение, которое нуж- но отправить безопасным образом.

**Отправитель**. С точки зрения криптографии это человек, который пытается от- править сообщение получателю безопасным образом.

**Отсортированное дерево**. Дерево, вершины которого упорядочены таким об- разом, что при определенном обходе они обрабатываются в отсортированном по- рядке. Обычно речь идет о симметричном обходе.

**Очередь**. Структура данных, в которой добавление и удаление элементов про- исходит в порядке «первым зашел — первым вышел».

**Параллелизм данных**. Разновидность параллельных вычислений, в которых акцент делается на распределении данных между процессорами, выполняющими одинаковые или похожие программы.

**Параллелизм задач**. Вид параллельных вычислений, основной акцент в которых делается на распределении задач между процессорами.

**Первичная кластеризация**. Эффект, при котором значения, привязываемые к кластеру записей в хеш-таблице с открытой адресацией, в итоге расширяют кла- стер, формируя из занятых записей длинные блоки. Это увеличивает среднюю длину пробной последовательности.

**Первый общий предок**. Узел, являющийся предком обеих вершин и находя- щийся наиболее близко к ним.

**Перестановка**. Упорядоченное подмножество элементов, взятых из множества.

**Перестановочный шифр**. Шифр, в котором буквы открытого текста перестав- ляются определенным образом, формируя криптограмму.

**Перехватчик информации**. В контексте криптографии это человек, который пытается перехватить и расшифровать сообщение, посланное отправителем в адрес получателя.

**Петля**. В контексте сети это цикл.

**Планарный**. Сеть является планарной, если ее можно нарисовать на плоскости без пересечения звеньев.

**Поддерево**. Одна из вершин дерева вместе со всеми ее потомками.

**Подстановочный шифр Цезаря**. Подстановочный шифр, который сдвигает каждую букву сообщения на какое-то определенное расстояние.

**Подстановочный шифр**. Шифр, в котором буквы открытого текста заменяются другими буквами. В качестве примера можно привести подстановочные шифры Цезаря и Виженера.

**Полиномиальное время выполнения**. Время выполнения алгоритма является полиномиальным, если оно включает в себя многочлен *N*. Примерами полиноми- ального времени могут служить *O*(*N*), *O*(*N*2), *O*(*N*6) и даже *O*(*N*4000).

**Политика разрешения коллизий**. Определяет процесс добавления в хеш- таблицу нового значения в случае возникновения коллизии.

**Полное дерево**. Дерево, в котором количество дочерних объектов каждой вер- шины колеблется от нуля до величины степени самого дерева.

**Получатель**. С точки зрения криптографии это человек, который пытается по- лучить сообщение, посланное отправителем.

**Порядок**. В *B*-дереве порядка K внутренние вершины хранят от *K* до 2  *K*

значений и от *K* + 1 до 2  *K* + 1 ветвей.

**Потомок**. Дочерний объект вершины дерева, дочерний объект ее дочернего объекта и т. д.

**Правило трапеции**. Метод численного интегрирования, в котором для аппрок- симации площади под кривой используются трапеции.

**Предок**. Родительский объект вершины дерева, родительский объект ее роди- тельского объекта и т. д. вплоть до корневой вершины.

**Префиксное дерево**. Дерево, в котором вершины представляют буквы строк, а путь от корня к вершине определяет префикс, общий для всех строк снизу от нее.

**Проблема согласования**. Проблема, возникающая в распределенных вычисле- ниях, когда несколько процессов должны согласовать единое значение, даже если с некоторыми из них произошел сбой.

**Пробная последовательность**. Последовательность позиций, в которые хеш- таблица с открытой адресацией пытается записать значение.

**Простое число**. Натуральное число больше 1, которое кратно только 1 и себе самому.

**Процент заполнения**. Доля структуры данных (например, хеш-таблицы), ко- торая является заполненной. Хеш-таблицы с высокой долей заполнения могут де- монстрировать снижение производительности.

**Прямая рекурсия**. Рекурсия, которая возникает, когда метод вызывает себя напрямую.

**Прямой обход**. Обход дерева или сети, при котором сначала посещается сама вершина, потом ее левый дочерний объект и затем правый дочерний объект.

**Путь**. Последовательность чередующихся узлов и звеньев сети, которая ведет от одного узла к другому. Если любая точка сети соединяется со смежным узлом всего одним звеном, вы можете указать путь, перечислив все содержащиеся в нем узлы или звенья.

**Равноправие**. Генератор псевдослучайных чисел является равноправным, если он выдает все возможные варианты вывода с одинаковой вероятностью. Неравно- правные генераторы называют неправильными.

**Разделение корня**. Когда ряд разделений вершин проходит каскадом вверх по сбалансированному дереву, приводя в конечном итоге к разделению корня.

**Распределение рабочих мест**. Имеется *N* человек с определенными навыка- ми и *M* рабочих мест с конкретными требованиями к квалификации. Нужно раз- делить людей по должностям таким образом, чтобы максимизировать объем вы- полняемой работы.

**Распределенные вычисления**. Несколько компьютеров работает вместе по сети для выполнения какой-то задачи.

**Расшифровывание**. В контексте криптографии это процесс преобразования открытого сообщения в криптограмму.

**Регулярное выражение**. Шаблон для сопоставления символов в строке.

**Редакционное расстояние**. Количество изменений, необходимых для преоб- разования одной строки в другую.

**Рекурсия**. Процесс, который происходит, когда метод вызывается напрямую или опосредованно.

**Родительская вершина**. Вершина дерева, которая имеет дочерние вершины, со- единенные с ней с помощью ветвей. Любая вершина дерева, кроме корневой, име- ет ровно одного родителя.

**Салфетка**. Разновидность самоподобных фракталов. Начинается с геометриче- ской фигуры, такой как квадрат или треугольник, затем делится на такие же фигу- ры, только меньшего размера, и рекурсивно заполняет некоторые из них (но не все).

**Самоподобный фрактал**. Кривая, части которой напоминают ее саму.

**Сбалансированное дерево**. Дерево, которое при необходимости переставляет свои вершины, чтобы не становиться слишком высоким или тонким. Это позволя- ет алгоритмам, которые его обходят, иметь время выполнения *O*(log(*N*)).

**Свидетель Ферма**. Если p и n являются натуральными числами в диапазоне 1  *n*  *p*, и при этом *np* – 1 mod *p*  1, тогда n называют свидетелем Ферма, поскольку он доказывает, что число p непростое.

**Связанный**. Узлы *A* и *B* являются связанными в ненаправленной сети, если узел *B* доступен из узла *A*. Ненаправленная сеть является связной, если каждый ее узел доступен из любого другого узла.

**Связное дерево**. Дерево, содержащее один или несколько тредов.

**Связный компонент**. Набор взаимосвязанных узлов сети.

**Связный список**. Список, ячейки которого соединены одной или нескольки- ми ссылками.

**Сильно связанный**. Направленная сеть является сильно связанной, если каж- дый ее узел доступен из любого другого узла.

**Симметрично связное дерево**. Дерево, содержащее треды, которые следуют по симметричному обходу в прямом и обратном направлениях.

**Симметричное шифрование**. Метод шифрования, в котором для кодирования и декодирования сообщений используется один и тот же ключ. Этот ключ должен быть как у отправителя, так и у получателя.

**Симметричный обход**. Обход дерева или сети, при котором сначала посеща- ется левый дочерний объект вершины, потом сама вершина и затем правый дочер- ний объект.

**Систолический массив**. Массив, состоящий из модулей обработки данных, ко- торые называются ячейками и используют параллелизм данных для обеспечения параллельной обработки.

**Слабо связанный**. Направленная сеть является слабо связанной, если каждый ее узел доступен из любого другого узла при условии, что направленные звенья можно заменить на ненаправленные.

**Слияние вершин**. Процесс разделения вершины на две во время добавления нового значения в сбалансированное дерево.

**Словарь**. Хеш-таблицу иногда называют словарем, поскольку она связывает ключи со значениями.

**Случайный поиск**. Эвристическая методика нахождения результата путем по- иска по дереву принятия решений.

**Смежный**. Если два узла сети соединены звеном, они являются смежными.

**Сортировка деревом**. Алгоритм сортировки, который сначала генерирует от- сортированное дерево и затем возвращает упорядоченные элементы с помощью симметричного обхода.

**Составное число**. Натуральное число больше 1, которое не является простым.

**Список FIFO**. См. *очередь*.

**Список LIFO**. См. *стек*.

**Ссылка**. Указатель, ведущий из одной ячейки связного списка к другой или, в случае с деревом или сетью, от одного узла к другому.

**Стек магазинного типа**. См. *стек*.

**Стек**. Структура данных, добавление и удаление элементов в которой проис- ходит в порядке «последним зашел — первым вышел».

**Степень**. В контексте вершины дерева это количество дочерних объектов, ко- торые она имеет. В контексте дерева это максимальная степень любых его вершин. В контексте узла в сети или графе это количество звеньев, исходящих из данно- го узла.

**Структура данных**. Способ упорядочивания данных, упрощающий решение конкретных задач.

**Теорема о четырех красках**. Теорема, утверждающая, что любую карту мож- но закрасить максимум четырьмя цветами.

**Теория вычислительной сложности**. Раздел информатики, связанный со слож- ностью вычислительных задач, Ориентирован не на конкретные алгоритмы, а на проблемы, которые решаются с их помощью.

**Топологическая сортировка**. Процесс расширения частичного упорядочива- ния сети до полного.

**Тред**. Последовательность ссылок, прокладывающих путь по таким структу- рам данных, как дерево или сеть.

**Треугольный массив**. Двумерный массив, в котором элементы, находящиеся над диагональю (это когда индекс столбца больше индекса строки), имеют некие значения по умолчанию, такие как 0, null или «пусто».

**Упорядоченное дерево**. Дерево, в котором порядок следования дочерних объ- ектов каждой вершины имеет значение.

**Уровень**. Уровень вершины дерева — это расстояние от нее до корня.

**Условие гонки**. Ситуация, при которой два процесса пытаются выполнить за- пись внутрь ресурса примерно в один и тот же момент времени. Процесс, который

записывает вторым, побеждает. Эта проблема характерна для распределенных вы- числений, в частности когда один компьютер имеет несколько ЦПУ.

**Факториал**. Факториал числа n (записывается как *n*! и произносится как «эн факториал») равен *n*  (*n* – 1)  (*n* – 2)  ...  1.

**Флопс**. Количество операций с плавающей запятой в секунду (floating-point operations per second, или FLOPS). Скорость выполнения вычислений иногда из- меряют в мегафлопсах, гигафлопсах, терафлопсах (1 трлн флопсов) или петафлоп- сах (1 тыс терафлопсов).

**Формула прямоугольников**. Метод численного интегрирования, в котором для аппроксимации площади под кривой используются прямоугольники.

**Формула Симпсона**. Численный метод интеграции, который аппроксимирует площадь снизу от кривой с помощью полиномов второй степени.

**Формулы Ньютона** — **Котеса**. Численный метод интеграции, который ис- пользует полиномы для аппроксимации кривой, чтобы найти площадь снизу от нее.

**Ханойская башня**. Головоломка, в которой нужно переместить стопку дисков из одного колышка на другой. Диски нужно перемещать по одному; диск больше- го размера нельзя помещать над диском меньшего диаметра.

**Хеширование**. Процесс привязки ключа к определенному месту в хеш-таблице.

**Хеш-таблица**. Структура данных и алгоритмы, которые привязывают к опре- деленному ее месту сами данные.

**Цикл**. Путь в сети, который возвращается в свою начальную точку.

**Частичное упорядочивание**. Множество зависимостей, которое определяет по- рядок следования некоторых, но не обязательно всех содержащихся в нем объектов.

**Числа Фибоначчи**. Числа Фибоначчи определяются через функции Fibonacci(0)=

= 0, Fibonacci(1) = 1, and Fibonacci(*n*) = Fibonacci(*n* – 1) + Fibonacci(*n* – 2), где *n* > 1.

**Числовая квадратура**. См. *численное интегрирование*.

**Численное интегрирование**. Процесс численной аппроксимации площади под кривой, когда ее нельзя рассчитать точно с помощью первообразной функции и функционального исчисления.

**Чрезвычайно параллельный**. Алгоритм, который можно естественным обра- зом разбить на параллельные части, требующие минимального взаимодействия.

**Шаг по индексу**. В хеш-таблице с открытой адресацией и линейным пробиро- ванием это значение, которое добавляется в каждую позицию пробной последова- тельности.

**Шифр Виженера**. Подстановочный шифр, в котором каждая буква сообще- ния сдвигается на некое количество позиций, определяемое соответствующей бук- вой ключа.

**Шифр с одноразовыми блокнотами**. Шифр, при котором каждая буква в со- общении объединяется с соответствующей буквой в блокноте; содержимое блок- нота генерируется случайно или путем сдвигов. Это похоже на шифр Виженера, в котором длина ключа совпадает с длиной самого сообщения.

**Шифр**. Пара алгоритмов, которые используются для шифрования и расшиф- ровывания сообщений.

**Шифрование с открытым ключом**. Метод шифрования, в котором используется два ключа — закрытый и открытый. Открытый ключ используется отправителем, чтобы зашифровать сообщение, а закрытый — получателем, чтобы его расшифро- вать.

**Эвристика**. Алгоритм, результат работы которого часто оказывается хорошим, но не обязательно наилучшим.

**Ячейка**. Объект, из экземпляров которого состоит связный список. Ячейка содержит данные и ссылку на следующую ячейку в списке.

# АЛФАВИТНЫЙ УКАЗАТЕЛЬ

А

Автомат:

детерминированный конечный (ДКА) 349–351

детерминированный с конечным числом состояний 525, *см. также* автомат детерминированный конечный

недетерминированный конечный (НКА) 354, 355, 528

Адельсон-Вельский, Георгий Максимо- вич 257

Адлеман, Леонард 380

Адресация открытая 171, 172, 446, 529

Аксессор 105

Алгоритм 24, 25, 522

Базовый 181

Бойера – Мура 356–359, 450

Вероятностный 60, 443, 523

Графический 187

Дополнительный сетевой 324

Евклида 53, 54, 380, 382, 522,

Нахождения циклов Флойда 92, 522

Основной сетевой 296

Распределенный 402, 413, 414

С возвратом 194–196, 522

Специализированный 240

Строковый 345

Форда — Фалкерсона 338, 522 Флойда — Уоршелла 315

Ход коня 198–200

Черепахи и кролики 92–94, 522, *см. также* алгоритм нахождения циклов Флойда

Численный 45, 442

Чрезвычайно параллельный 414,

452

Асимптотическая сложность (производительность) 29, 33, 34, 36, 37,

40, 523

Ассоциативный массив 169, 523

Б

Башня ханойская 184–187, 534 Безопасность через неясность 365 Беллазо, Джован Баттиста 373 Блок 170, 523

Блокировка:

активная взаимная 417, 522

взаимная 411, 412, 523

В

Вдавливание в стек 120 Ведро 266

Величина 28

Обратная 381

Вершина (узел) 215, 523

Внешняя 523

Внутренняя 216, 523

Дочерняя 215

Корневая 215, 527 *см. также*

объект дочерний Одного уровня 523

Родительская 215

Терминальная (листовая) 215, 216

Ветвь 215, 218, 523

Виженер, Блез 373 Визуализация:

дерева квадрантов 244

списка 85

функций 38, 39

Возведение в степень 54–56, 242

Вставка 160, 444

Элементов 100

Ячеек 77

Выбор разделяющего элемента 151 Выражение:

арифметическое 347

булево 392, 393

математическое 241

регулярное 240, 348, 351, 531

Высота 218, 524

Вершины 216

Дерева 216 Выталкивание из стека 120 Вычисление:

квантовое 412

кластерное 526

многопроцессорное 407

распределенное 405, 531

редакционного состояния 359

Г

Генератор 187, 524

Криптостойкий 48, 524

Линейный конгруэнтный 46, 48, 49,

527

Неправильный 48 Псевдослучайных чисел (ГПСЧ) 45–47, 187, 524

Равноправный 48 Генерирование:

величин 46

неравномерных распределений 52

случайных величин 45–50

Голодание 411 Граф

Двудольный 393

Грид 406

Грид-вычисление 406, 407, 524

Д

Дек 133, 444

Дерево 215

2–3-дерево 261–266, 448

B-деревo 266–270, 448, 522

Восходящее В-дерево 270, 523 Иерархически организованное B-дерево 270, 526

B+-дерево 270–272, 522

АВЛ-дерево 257–260, 522

Бинарное 34, 142, 143, 215, 218, 523

Восьмеричное 248, 523

Завершенное 217, 218, 223, 525

Игры 448

Идеальное 217, 218, 253 Иерархически организованное В-дерево 270, 448

Квадрантов (квадродерево) 243–

248, 524

Кратчайшего пути 524

Минимальное остовное 308, 528

Неупорядоченное 216

Остовное 307, 449 , 495, 529

Отсортированное 529

Полное 36, 217, 218, 530

Префиксное 248–252, 531

Принятия решений 274–294,

525

Сбалансированное 257, 532

Связное 235–240, 532

Симметрично связное 235, 532

Синтаксическое 347, 348, 351, 363,

450, 501

Упорядоченное 216, 218, 229, 533 Декодирование 524, *см. также* расшифровка

Делитель наибольший общий 52, 528

Дешифрование (декодирование) 366 Диаграмма перехода состояний 349, *см. также* автомат детерминированный конечный Добавление:

вершин 230, 231

значений 258–260, 262–264, 267

ячеек 76, 77

Е

Емкость (пропускная способность) 525

З

Задача:

NP-полная 394, 397

NP-сложная 394

Византийских генералов 420–423,

452

Выбора лидера 452 Выполнимости булевых формул 293, 294

Двух генералов 419, 452

Дерева принятия решений 290–294 Коммивояжера 293

О восьми ферзях 196, 197, 294, 525 О генерале и лейтенантах 421–423, 525

О назначениях 339

О ранце 292, 293

Об упаковке в контейнеры 291 Обедающих философов 416, 417,

452, 526

Обнаружения 394–396

Оптимизации 282, 394–396, 526

Раскроя 291, 292

Синхронизации часов 452

Сообщения 394–396

Закон Мура 402, 429, 526

Злоумышленник (перехватчик) 366,

526

И

Игра:

Животные 240, 241

Крестики-нолики 275–278, 279

Инициатор 187, 526

Интеграция Монте-Карло 45, 66, 67,

69, 526

Интегрирование численное 60, 61, 534 Использование:

RSA 380, 382

быстрой сортировки 154

ограничителей 75

связных деревьев 238–240

стеков 124–129

хеш-таблиц 88

К

Квадратура 60, 526

Квадратура адаптивная 63–66, 522

Квадратура числовая 534,

*см. также* интегрирование численное, квадратура

Класс сложности 451

Кластер 174, 406

Кластеризация 446

Вторичная 176

Первичная 173, 176, 530

Ключ 365, 526

Коллизия 169, 527

Компонент связный 297, 532

Компьютер квантовый 412, 413, 526

Копирование 82

Корень 218, 527

Коэффициент заполнения хеш- таблицы 169

Кривая

Коха 187, 188, 527

Гильберта 189, 527

Серпинского 190–193, 527

Криптоанализ 366

Криптография 21, 365–384, 527

Куча 142, 527

Двойная 143–147

Л

Ландис, ЕвгенийМихайлович 257 Линия 215, *см. также* ссылка Лист 218

М

Малая теорема Ферма 59 Маркировка ячеек 87

Массив 96–119, 444, 528

Ассоциативный 523, *см. также*

хеш-таблица

Высокой размерности 102–105

Двумерный 101, 102

Одномерный (линейный) 98 Равноправно рандомизированный 51

С пропуском 527

С разрывом 108–110

Систолический 403, 532

Треугольный 105–108, 533

Матрица 115–117

С разрывом 117

Смежности 105, 528

Машина Тьюринга 389

Метод 24, 25

Ветвей и границ 284, 285, 528

Монте-Карло 67, 443

Ньютона –Рафсона 67, 68, 528, *см. также* метод Ньютона Ньютона 67, 528, *см. также* метод Ньютона – Рафсона

Полного перебора 282–284

Сборки мусора 79, 115

Минимакс 275–278, 448

Мур, Гордон Э. 402, 526

Мьютекс 409–411, 529

Н

Нахождение:

минимальной, максимальной и средней величин 98–100

наибольшего общего делителя 52 нулей 67–69

простых множителей 56–58

простых элементов 58, 59

строки и столбца 110, 111

элементов 98

ячеек 74, 75

О

Обольшое 442, 529

Обманщик Ферма 59, 71, 528

Обработка файлов 415

Обход 529

В глубину 303, 304

В обратном порядке 227, 228

В ширину 228, 304

Дерева 223

Сети 302

Симметричный 226, 227, 532

Объект дочерний 525,

*см. также* вершина дочерняя Ограничитель 75, 76, 529

Омега большое 388, 529

Операнд 242, 393

Отправитель 366, 529

Очередь 129–134, 130, 444, 530,

*см. также* список FIFO Массивов 130–133

С приоритетом 133

Связных списков 130

Специализированная 133

П

Параллелизм 402

Данных 530

Задач 530

Паросочетание двудольное 339, 394,

524

Передвижение:

по спискам 73, 74

элемента 148

Перестановка 447, 530

Приоритетов 411

Столбцов 369–371

Строк/столбцов 367–369

Перехватчик информации 366, 530 Петля 296, *см. также* цикл Повторная трассировка списка 89, 90

Поддерево 216, 218, 530

Подпись цифровая 384, 451

Подстановка простая 375

Поиск 163–168, 309–315

Бинарный 164, 165, 523

Вершин 231

Интерполяционный 165, 166

Кратчайшего пути 310–315

Линейный (исчерпывающий) 98,

163, 164, 526, 527

Методом полного перебора 415 По неиндексированной базе данных 415

По деревьям игры 274–281

Произвольного пути 309

Случайный 415, 533

Циклов 324, 327, 328

Получатель 366, 530

Получение значения 111

Последовательность пробная 171 Построение:

деревьев 220–223

ДКА для регулярных выражений 351–354

завершенных деревьев 223

связных деревьев 236–238

Поток 85, 86, 336

Максимальный 336

Потомок 215, 218, 530

Предок 215, 218, 528, 531

Первый (наименьший) общий 216*,* 218, 528, 530

Привязка к диску 416 Пробирование:

квадратичное 174–176, 446, 480

линейное 173, 174, 180, 446, 480,

527

псевдослучайное 176, 480 Проверка:

на простоту 59

связности 305–307

Процент заполнения 480, 531

Псевдокод 25–27

Путь 531

Дополняющий 338, 525

Р

Равноправие 48, 49, 443, 531 Разделение:

вершины 262

корня 263, 531

Размещение 201

Без повторений 205

С повторениями 204 , 205

Разрез минимальный 340–342

Рандомизация 436, 442, 445, 446

Данных 45–52

Массивов 50–52, 151

Раскрашивание карты 328

Двумя цветами 328–330

Пятью цветами 331–335

Тремя цветами 330, 331

Четырьмя цветами 331 Распределение рабочих мест 338–340, 531

Расшифровка 531, *см. также* декодирование Расшифровывание 531 Ребро 531, *см. также* звено Реверсирование:

массива 124

списка 90–92

Рекурсия 181–212, 532

Множественная 528

Одиночная 529

Прямая 531

Решето Эратосфена 58, 71, 390,

443

Ривест, Рональд 380

Родитель 218

С

Салфетка 193, 194, 532

Серпинского 193

Свидетель Ферма 59, 532 Свойство:

алгоритма 28, 29

бинарного дерева 219

класса 243

Связывание 445

Прямое 169, 170

Упорядоченное 446

Сеть:

двудольная 339

нагруженная 336, 528

остаточная 337, 529

планарная 331

подстановочно-перестановочная 377

сильно связная 297

слабо связная 297

Скобки парные 345–349

Слияние вершин 265, 533

Словарь 169, 533, *см. также* хеш- таблица

Снимок 427, 428

Сообщение симметричное 380 Сортировка:

блочная (корзинная) 159, 160

быстрая 148, 149, 152, 154, 160:

вагонов поезда 124–126

вставкой 82, 127, 128, 136–138,

160

выбором 128, 129, 138, 139, 160

деревом 533

методом выбора 84

пирамидальная 142, 147, 148, 160

подсчетом 157, 158, 160

пузырьковая 139–142, 160

слиянием 155–157, 160, 416, 452

топологическая 18, 324, 326, 533

внешняя 156

со стеками 151

устойчивая 156

Состояние гонки 407, 409, 413, 429

Сочетание 200, 201, 533

Без повторений 204

С повторениями 202, 203

С циклами 201, 202 Список:

двунаправленный связный 79, 80

кольцевой связный 86, 87, 527

многопотоковый связный 85

однонаправленный связный 73–78

связный 72, 81, 443, 532

с циклами 86–94

сортированный 80, 81, 128

FIFO 444, 533, *см. также* очередь

LIFO 119, 533, *см. также* стек Способность остаточная пропускная 337

Стек 119, 120, 183, 444, 533, *см. также*

LIFO, список LIFO Вызова 183

Двойной 123, 124

Магазинного типа 120, 121, 533

Массивов 121–123

Связного списка 120

Степень 34, 218, 298, 533

Вершины 215, 218

Входящая 298, 524

Дерева 215, 266, 267

Исходящая 298, 526

Узла 296, 297

Стивенс, Род 16

Схема одноразовых блокнотов 375, 376

Ссылка 215, 533, *см. также* линия

Структура 25

Данных 442, 533

Т

Текст открытый 366 Теорема Кука – Левина 392

Теорема о четырех красках 331, 533

Теория вычислительной сложности 387

Терафлопс 407, *см. также* флопс Тета большое 388, 529

**У**

Удаление:

вершин 232–235

значения 113–115, 260, 264, 268

рекурсии 206–212

элементов 101, 172

ячеек 78, 79

Узел-сестра 215

Доступный 525

Дочерний 216

Упорядочивание частичное 325

Уровень (глубина) вершины 216, 218,

534

Условие гонки 534

Установка значения 111

Ф

Факториал 181–183, 415, 534

Флойд, Роберт 92

Флопс 534, *см. также* терафлопс Формула прямоугольников 61, 534

Ньютона – Котеса 61, 63, 534

Симпсона 63, 534

Трапеций 61, 62

Функция Эйлера 381, 382

Х

Ханойская башня 126, 127, 534

Хеширование 169, 534

Двойное 176, 180

Упорядоченное 176, 179

Хеш-таблица 101, 168–171, 173, 174,

178, 534, *см. также* ассоциативный массив, словарь

Хип 183 Хранение:

полных бинарных деревьев в массивах 142, 143

промежуточных значений 208, 447

Ц

Цикл 296, 534

В двунаправленных связных списках 94

Рабочий 33–38

Флойда 92

Ч

Число:

взаимно простое 381, 523

натуральное 528

начальное 46

простое 56, 531

составное 56, 533

Фибоначчи 183, 184, 456, 534

Ш

Шамир, Ади 380

Шифр 366, 535

Блочный 376

Виженера 373, 534

Маршрутный 371

Подстановочный 372–376, 530

Перестановочный 367–371, 530 С одноразовыми блокнотами 535 Фейстеля 378

Цезаря 372

Шифрование (кодирование) 366

С открытым ключом 380, 535

Э

Эвристика 38, 279, 449, 535

Дерева игры 279–281 Дерева принятия решений 285–290

Эффект случайности 417

Я

Ячейка 74–79, 535

*Âñå ïðàâà çàùèùåíû. Êíèãà èëè ëþáàÿ åå ÷àñòü íå ìîæåò áûòü ñêîïèðîâàíà, âîñïðîèçâåäåíà*

*â ýëåêòðîííîé èëè ìåõàíè÷åñêîé ôîðìå, â âèäå ôîòîêîïèè, çàïèñè â ïàìÿòü ÝÂÌ, ðåïðîäóêöèè èëè êàêèì-ëèáî èíûì ñïîñîáîì, à òàêæå èñïîëüçîâàíà â ëþáîé èíôîðìàöèîííîé ñèñòåìå áåç ïîëó÷åíèÿ ðàçðåøåíèÿ îò èçäàòåëÿ. Êîïèðîâàíèå, âîñïðîèçâåäåíèå è èíîå èñïîëüçîâàíèå êíèãè èëè åå ÷àñòè áåç ñîãëàñèÿ èçäàòåëÿ ÿâëÿåòñÿ íåçàêîííûì è âëå÷åò óãîëîâíóþ, àäìèíèñòðàòèâíóþ è ãðàæäàíñêóþ îòâåòñòâåííîñòü.*

Ïðîèçâîäñòâåííî-ïðàêòè÷åñêîå èçäàíèå ÌÈÐÎÂÎÉ ÊÎÌÏÜÞÒÅÐÍÛÉ ÁÅÑÒÑÅËËÅÐ

**Ðîä Ñòèâåíñ ÀËÃÎÐÈÒÌÛ**

**Òåîðèÿ è ïðàêòè÷åñêîå ïðèìåíåíèå**

(орыс тiлiнде)

Äèðåêòîð ðåäàêöèè *Å. Êàïü¸â* Îòâåòñòâåííûé ðåäàêòîð *Â. Îáðó÷åâ* Ëèòåðàòóðíûé ðåäàêòîð *Ñ. Ïëîòíèêîâ* Õóäîæåñòâåííûé ðåäàêòîð *Å. Ìèøèíà*

ООО «Издательство «Э»

123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86.

8ндіруші: «Э» АК,Б Баспасы, 123308, Мдскеу, Ресей, Зорге ктшесі, 1 ей.

Тел. 8 (495) 411-68-86.

Тауар белгісі: «Э»

К,азавстан Республикасында дистрибьютор ждне тнім бойынша арыз-талаптарды вабылдаушыны9 ткілі «РДЦ-Алматы» ЖШС, Алматы в., Домбровский ктш., 3«а», литер Б, офис 1.

Тел.: 8 (727) 251-59-89/90/91/92, факс: 8 (727) 251-58-12 вн. 107.

8німні9 жарамдылыв мерзімі шектелмеген.

Сертификация туралы авпарат сайтта 8ндіруші «Э»

**Оптовая торговля книгами Издательства «Э»:** 142700, Московская обл., Ленинский р-н, г. Видное, Белокаменное ш., д. 1, многоканальный тел.: 411-50-74.

**По вопросам приобретения книг Издательства «Э» зарубежными оптовыми покупателями обращаться в отдел зарубежных продаж**

*International Sales: International wholesale customers should contact Foreign Sales Department for their orders.*

**По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении**, *обращаться по тел.: +7 (495) 411-68-59, доб. 2115/2117/2118; 411-68-99, доб. 2762/1234.*

**Оптовая торговля бумажно-беловыми**

**и канцелярскими товарами для школы и офиса**: 142702, Московская обл., Ленинский р-н, г. Видное-2,

Белокаменное ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).

Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить на сайте Издательства «Э»

8ндірген мемлекет: Ресей Сертификация варастырылмабан

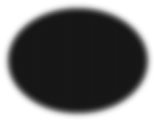
Ïîäïèñàíî â ïå÷àòü 16.11.2015.

Ôîðìàò 70x1001/ . Ïå÷àòü îôñåòíàÿ. Óñë. ïå÷. ë. 44,07.

16

Òèðàæ ýêç. Çàêàç

ИНТЕРНЕТ-МАГАЗИН



И Н Т Е Р Н Е Т - М А Г А З И Н

И Н Т Е Р Н Е Т - М А Г А З И Н

ИНТЕРНЕТ-МАГАЗИН





Практичное и подробное руководство по освоению алгоритмов

**Алгоритмы. Теория и практическое применение**

Книга является практически ориентированным руководством

по изучению алгоритмов. В книге проводится их анализ, разбираются типы структур данных и методы работы с массивами, связными списками, деревьями, сетями; описываются основные алгоритмы поиска и сортировки и сетевые алгоритмы; отдельно рассмотрены алгоритмы шифрования информации.

* **Большой объем обучающеrо материала с подробным практическим применением**
* **Упражнения в конце каждой rлавы для закрепления материала**
* **Все алrоритмические термины в отдельном разделе**
* **Разбор задач, встречающихся при приеме разработчиков на работу**

Род *Стивенс начинал свою карьеру как математик, но , окунувшись в захватываю­ щий мир алгоритмов во время пребывания в Массачусетском технологическом институт, еувлекся программированием, чем и занимается по сей день. Он отме­ чен наградами* за *свою преподавательскую работу и регулярно выступает на конференциях. На его счету* 26 *книг, переведенных на десятки языков.*

##### В книге последовательно и доходчиво рассмотрены обязательные

для понимания прикладные вопросы - от работы с сортировкой, списками и деревьями до парамельных и криптографических алгоритмов. Книга пригодится специалистам начального и среднего уровня. Студенты найдут там все необходимое для решения типичных алгоритмов, специалисты - понятное изложение сложных вещей. Псевдокод вместо использования какого -либо языка программирования и тщательно подобранные задачи дают желающим возможность подробно ознакомиться с любым заинтересовавшим их разделом.

Дмитрий Белявский, Технический центр Интернет

**Дополнительные материалы можно скачать по адресу:** [**http://eksmo.ru/upload/Algorithms\_examples.zip**](http://eksmo.ru/upload/Algorithms_examples.zip)

ISBN978-5-699-81729-О

1111 11 1 111 11 111 111 11 11 11111 111 

9 >

78 56 9 9 817290