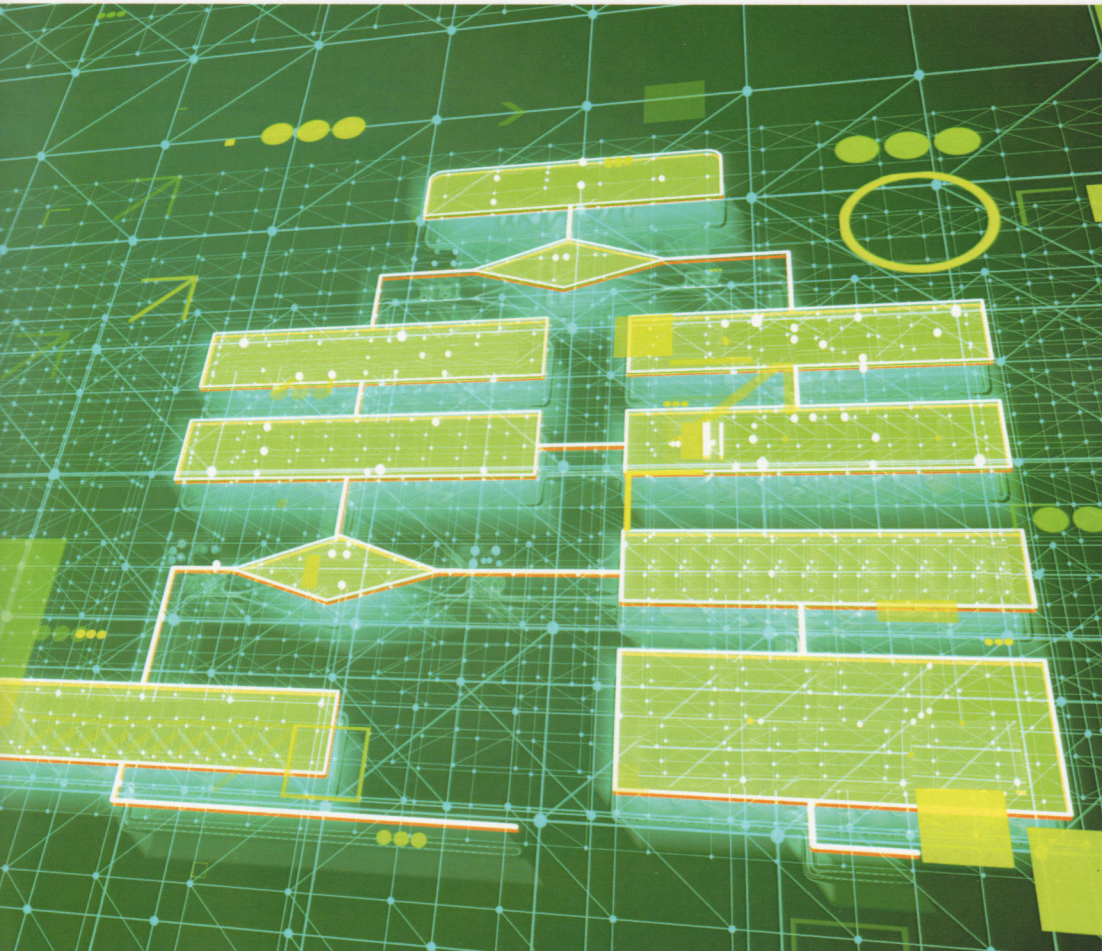
Ахо • Хопкрофт • Ульман

РАЗРАБОТКА и АНАЛИЗ компьютерных алгоритмов



РАЗРАБОТКА и АНАЛИЗ компьютерных алгоритмов

THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS

Alfred V. Aho
Bell Laboratories
John E. Hopkroft
Cornell University
Jeffrey D. Ullman
Princeton University

Pearson

РАЗРАБОТКА и АНАЛИЗ компьютерных алгоритмов

Альфред В. Ахо
Bell Laboratories

Джон Э. Хопкрофт
Корнеллский университет

Джеффри Д. Ульман
Принстонский университет

Київ
Комп'ютерне видавництво
"ДІАЛЕКТИКА"
2021

УДК 004.421
А95

Перевод с английского и редакция докт. физ.-мат. наук *Д.А. Клюшина*,
при участии *И.В. Василенко*

Ахо, А.В., **Хопкрофт**, Дж.Э., **Ульман**, Дж. Д.

А95 Разработка и анализ компьютерных алгоритмов/Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман; пер. с англ. Д.А. Клюшина. — Киев. : "Диалектика", 2021. — 544 с. : ил. — Парал. тит. англ.

ISBN 978-617-7874-04-0 (укр.)
ISBN 978-0-201-00029-0 (англ.)

В этой монографии, ставшей классикой, излагаются результаты теоретических и прикладных исследований по построению и анализу эффективных вычислительных алгоритмов. Рассмотрены задачи поиска, сортировки массивов, умножения целых чисел, умножения матриц, алгоритмы на графах, а также основы теории сложности. Книга предназначены для специалистов по компьютерным наукам и программистов, а также будет полезна студентам и аспирантам, специализирующимся в области информатики.

УДК 004.421

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Copyright © 2021 by Dialektika Computer Publishing.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc, Copyright © 1974.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

ISBN 978-617-7874-04-0 (укр.)
ISBN 978-0-201-00029-0 (англ.)

© "Диалектика", перевод, 2021
© 1974 Addison-Wesley Publishing Company, Inc.

Оглавление

Telegram https://t.me/it_boooks

Предисловие 11

Глава 1. Модели вычислений 15

Глава 2. Разработка эффективных алгоритмов 61

Глава 3. Сортировка и порядковые статистики 97

Глава 4. Структуры данных для работы с множествами 131

Глава 5. Алгоритмы на графах 199

Глава 6. Умножение матриц и связанные с ним операции 259

Глава 7. Быстрое преобразование Фурье и его применения 291

Глава 8. Арифметические операции над целыми числами и полиномами 319

Глава 9. Алгоритмы сопоставления с образцом 365

Глава 10. *NP*-полные задачи 419

Глава 11. Некоторые доказуемо трудноразрешимые задачи 469

Глава 12. Нижние оценки числа арифметических операций 495

Список литературы 523

Предметный указатель 537

Содержание

Предисловие 11
Содержание книги 11
Как пользоваться книгой 12
Благодарности 13
От издательства 14

Глава 1. Модели вычислений 15
1.1. Алгоритмы и их сложность 15
1.2. Машины с произвольным доступом к памяти 18
1.3. Вычислительная сложность RAM-программ 25
1.4. Модель машины, хранящей программу в памяти 28
1.5. Модификации модели RAM 33
I. Линейные программы 34
II. Битовые вычисления 37
III. Операции с двоичными векторами 39
IV. Деревья решений 39
1.6. Простейшая модель вычислений: машина Тьюринга 40
1.7. Связь между машинами Тьюринга и компьютерами RAM 47
1.8. Псевдокод высокого уровня, написанный на Алголе 50
Упражнения 50
Проблема для исследования 59
Библиографические замечания 59

Глава 2. Разработка эффективных алгоритмов 61
2.1. Структуры данных: списки, очереди и стеки 61
2.2. Представления множеств 66
2.3. Графы 67
2.4. Деревья 69
2.5. Рекурсия 73
2.6. Метод “Разделяй и властвуй” 78
2.7. Балансировка 84
2.8. Динамическое программирование 86
2.9. Заключение 88
Упражнения 89
Библиографические замечания 95

Глава 3. Сортировка и порядковые статистики 97
3.1. Задача сортировки 98
3.2. Поразрядная сортировка 99
3.3. Сортировка с помощью сравнений 108
3.4. Пирамидальная сортировка с помощью $O(n \log n)$ сравнений 110
3.5. Быстрая сортировка с ожидаемой временной сложностью $O(n \log n)$ 115
3.6. Порядковые статистики 120
3.7. Ожидаемое время для порядковых статистик 123
Упражнения 126
Проблема для исследования 130
Библиографические замечания 130

Глава 4. Структуры данных для работы с множествами 131
4.1. Базовые операции над множествами 131
4.2. Хеширование 135
4.3. Бинарный поиск 137
4.4. Деревья бинарного поиска 139
4.5. Оптимальные деревья бинарного поиска 143
4.6. Простой алгоритм объединения непересекающихся множеств 148
4.7. Древовидные структуры для задачи UNION-FIND 153
4.8. Приложение и обобщения алгоритма UNION-FIND 164
4.9. Схемы сбалансированных деревьев 170
4.10. Словари и очереди с приоритетами 173
4.11. Сливаемые кучи 178
4.12. Сцепляемые очереди 181
4.13. Разбиение 183
4.14. Резюме 190
Упражнения 191
Проблема для исследования 197
Библиографические замечания 197

Глава 5. Алгоритмы на графах 199
5.1. Остовное дерево минимальной стоимости 199
5.2. Метод поиска в глубину 204
5.3. Двусвязность 208
5.4. Метод поиска в глубину в ориентированном графе 216
5.5. Сильная связность 218
5.6. Задачи поиска путей 225
5.7. Алгоритм транзитивного замыкания 229
5.8. Алгоритм поиска кратчайшего пути 231
5.9. Задачи о путях и умножение матриц 232
5.10. Задачи с одним источником 238

5.11. Доминаторы в ориентированных ациклических графах: сочетание понятий 241
I. Прямые ребра 245
II. Поперечные ребра 248
Упражнения 250
Проблемы для исследования 256
Библиографические замечания 257

Глава 6. Умножение матриц и связанные с ним операции 259

6.1. Основные понятия 259
6.2. Алгоритм Штрассена для умножения матриц 264
6.3. Обращение матриц 266
6.4. LUP-разложение матриц 268
6.5. Приложения LUP-разложения матриц 277
6.6. Умножение булевых матриц 280
Упражнения 285
Проблемы для исследования 289
Библиографические замечания 289

Глава 7. Быстрое преобразование Фурье и его применения 291

7.1. Дискретное преобразование Фурье и обратное к нему 292
7.2. Алгоритм быстрого преобразования Фурье 297
7.3. Быстрое преобразование Фурье с битовыми операциями 305
7.4. Произведение полиномов 310
7.5. Алгоритм Шёнхаге–Штрассена для умножения целых чисел 311
Упражнения 315
Проблема для исследования 317
Библиографические замечания 318

Глава 8. Арифметические операции над целыми числами и полиномами 319

8.1. Аналогии между целыми числами и полиномами 320
8.2. Умножение и деление целых чисел 321
8.3. Умножение и деление полиномов 329
8.4. Модульная арифметика 332
8.5. Модульная арифметика полиномов и вычисление их значений 336
8.6. Применение китайской теоремы об остатках 338
8.7. Китайская теорема об остатках и интерполяция полиномов 342
8.8. Наибольший общий делитель и алгоритм Евклида 345
8.9. Асимптотически быстрый алгоритм нахождения наибольшего общего делителя полиномов 348
8.10. Наибольший общий делитель целых чисел 354

8.11. Еще раз о китайской теореме об остатках 356
8.12. Разреженные полиномы 357
Упражнения 359
Проблемы для исследования 362
Библиографические замечания 362

Глава 9. Алгоритмы сопоставления с образцом 365

9.1. Конечные автоматы и регулярные выражения 365
9.2. Распознавание образцов, задаваемых регулярными выражениями 375
9.3. Распознавание подстрок 379
9.4. Двусторонний детерминированный магазинный автомат 386
9.5. Позиционные деревья и идентификаторы подстрок 399
Упражнения 413
Проблемы для исследования 417
Библиографические замечания 418

Глава 10. *NP*-полные задачи 419

10.1. Недетерминированные машины Тьюринга 420
10.2. Классы $\mathcal{P}$ и $\mathcal{NP}$ 430
10.3. Языки и задачи 432
10.4. *NP*-полнота и задача выполнимости 436
10.5. Еще несколько *NP*-полных задач 445
10.6. Задачи с полиномиально ограниченной памятью 457
Упражнения 463
Проблемы для исследования 467
Библиографические замечания 468

Глава 11. Некоторые доказуемо трудноразрешимые задачи 469

11.1. Иерархии по сложности 469
11.2. Иерархия детерминированных машин Тьюринга по пространственной сложности 470
11.3. Задача, требующая экспоненциального объема времени и памяти 474
11.4. Неэлементарная задача 485
Упражнения 490
Проблемы для исследования 493
Библиографические замечания 493

Глава 12. Нижние оценки числа арифметических операций 495

12.1. Поля 495
12.2. Еще раз о неветвящихся программах 497

12.3. Матричная формулировка задач 499
12.4. Нижняя граница для количества умножений, ориентированная на строки 500
12.5. Нижняя граница для количества умножений, ориентированная на столбцы 503
12.6. Граница для количества умножений, ориентированная на строки и столбцы 508
12.7. Предобусловливание 511
Упражнения 514
Проблемы для исследования 521
Библиографические замечания 522

Список литературы 523

Предметный указатель 537

Предисловие

Изучение алгоритмов лежит в основе компьютерных наук. В последние годы[1] в этой области были достигнуты значительные успехи. Их диапазон простирается от разработки более быстродействующих алгоритмов, таких, как быстрое преобразование Фурье, до поразительного открытия, которое заключается в том, что для решения некоторых естественных задач не существует эффективных алгоритмов. Достигнутые результаты стимулировали огромный интерес к исследованию алгоритмов, и область их разработки и анализа стала стремительно расширяться. Цель данной книги — изложить основные результаты, полученные в этой области исследований, чтобы систематизировать принципы и концепции разработки алгоритмов.

Содержание книги

Анализ эффективности алгоритма выполняется на определенном вычислительном оборудовании. Книга начинается с определения моделей компьютера, достаточно простых для анализа, но в то же время точно отражающих характерные черты реальных вычислительных машин. К этим моделям относятся регистровые машины с произвольным доступом к памяти, машины с хранением программ в памяти с произвольной выборкой, а также некоторые их разновидности. Для доказательства экспоненциальных нижних оценок эффективности алгоритмов в главах 10 и 11 вводится машина Тьюринга. Поскольку общая тенденция в разработке программ состоит в отходе от использования машинных языков, для описания алгоритмов используется язык высокого уровня, называемый псевдо-Алголом (Pidgin ALGOL). Сложность программы, написанной на псевдо-Алголе, зависит от модели машины.

В главе 2 описаны основные структуры данных и приемы программирования, часто применяемые в эффективных алгоритмах — списки, стеки, очереди, деревья и графы. Приведено подробное объяснение рекурсии, метода "разделяй и властвуй" и принципов динамического программирования, а также приводятся примеры их использования.

В главах 3–9 указаны разнообразные области, в которых могут применяться фундаментальные методы из главы 2. В этих главах основное внимание уделяется построению асимптотически наиболее эффективных алгоритмов. С асимптотической точки зрения некоторые из рассматриваемых алгоритмов эффективны лишь для входных данных, длина которых намного больше, чем встречающиеся на практике. В частности, это утверждение справедливо для некоторых алгоритмов умножения матриц из главы 6, метода умножения целых чисел Шёнхаге–Штрассена

[1] Первое издание книги вышло в 1974 г. — *Примеч. ред.*

из главы 7 и отдельных алгоритмов, предназначенных для работы с полиномами и целыми числами из главы 8.

С другой стороны, большинство алгоритмов сортировки из главы 3, поиска из главы 4, алгоритмов на графах из главы 5, быстрое преобразование Фурье из главы 7 и алгоритмы сравнения строк из главы 9 используются широко, поскольку размеры входных данных, для которых эти алгоритмы являются эффективными, достаточно небольшие и реалистичные для прикладных задач.

В главе 10–12 обсуждаются нижние границы сложности вычислений. Как для разработки программ, так и для понимания природы вычислений представляет интерес истинная вычислительная сложность задачи. В главе 10 изучается важный класс, состоящий из *NP*-полных задач. Все задачи из этого класса являются эквивалентными по вычислительной сложности, т.е. если одна из них имеет эффективное решение (с полиномиально ограниченным временем), то и все *NP*-полные задачи имеют эффективные решения. Поскольку этот класс содержит много практически важных и хорошо изученных задач, таких, как задача целочисленного программирования или задача о коммивояжере, то можно предположить, что ни одну из задач этого класса невозможно решить эффективно. Следовательно, если разработчику программы известно, что задача, для которой он пытается найти эффективный алгоритм, является *NP*-полной, то у него есть все основания попытаться применить эвристические подходы. Вопрос, имеют ли *NP*-полные задачи эффективные решения, остается открытым, хотя существует огромное число эмпирических свидетельств обратного.

В главе 11 описаны конкретные задачи, для которых можно доказать, что эффективных алгоритмов их решения действительно не существует. Материал глав 10 и 11 основан на использовании понятия машины Тьюринга, введенного в разделах 1.6 и 1.7.

В последней главе мы исследуем понятие сложности вычислений в разрезе линейной независимости в векторных пространствах. В этой главе исследуются методы доказательства нижних оценок для гораздо более простых задач, чем рассмотренные в главах 10 и 11.

Как пользоваться книгой

Эта книга задумана как вводный курс по разработке и анализу алгоритмов. Мы уделили особое внимание идеям и простоте понимания, а не реализации или программистским приемам. Во многих местах вместо длинных нудных доказательств мы приводим лишь неформальные интуитивные объяснения. Книга является самодостаточной и не требует от читателей специальной подготовки ни по математике, ни по языкам программирования. Однако желательно, чтобы читатели хорошо понимали математические понятия, а также владели языками программиро-

вания высокого уровня, такими как Фортран или Алгол. Для полного понимания глав 6–8 и 12 необходимы некоторые знания линейной алгебры.

Эта книга использовалась при чтении специальных курсов по разработке алгоритмов на младших и старших курсах. За один семестр изучался материал, охватывающий бо́льшую часть глав 1–5, 9 и 10, а также кратко излагались остальные главы. На младших курсах основной упор делался на материал глав 1–5, но разделы 1.6, 1.7, 4.13, 5.11 и теорема 4.5 обычно не изучались. Книгу можно также использовать для более глубоких спецкурсов с акцентом на теорию алгоритмов. Основой для этого могли бы служить главы 6–12.

В конце каждой главы приведены упражнения, предоставляющие преподавателю широкий выбор домашних заданий. Упражнения ранжированы по трудности. Упражнения без звездочек подходят для младших курсов, упражнения с одной звездочкой — для средних, а с двумя звездочками — для старших курсов. В библиографических замечаниях в конце каждой главы мы старались указать литературный источник для каждого алгоритма и результата, содержащихся в тексте и упражнениях.

Благодарности

Книга основана на конспектах лекций, читавшихся авторами в Корнеллском и Принстонском университетах, а также в Стивенсонском технологическом институте. Авторы благодарны многим людям, которые критически прочитали различные части рукописи и внесли многочисленные полезные предложения. В частности, мы хотели бы поблагодарить Келлога Бута, Стэна Брауна, Стива Чена, Алена Сайфера, Арча Дейвиса, Майка Фишера, Хейнию Гаевску, Майка Гэри, Юдая Гупту, Майка Харрисона, Матта Хекта, Гарри Ханта, Дейва Джонсона, Марка Каплана, Дона Джонсона, Стива Джонсона, Брайана Кернигана, Дона Кнута, Ричарда Лэднера, Аниту Ла-Саль, Дуга Мак-Илроя, Альберта Мейера, Кристоса Пападимитриу, Билла Плогера, Джона Сэвиджа, Ховарда Зигеля, Кена Стейглица, Лэрри Стокмейера, Тома Жимански и Теодора Ена.

Мы выражаем особую благодарность Джеме Карневейл, Полине Камерон, Ханне Кressе, Эдит Персер и Руфи Сузуки за быструю и качественную перепечатку рукописи.

Авторы также благодарны фирме Bell Laboratories и Корнеллскому университету, Принстонскому университету и Калифорнийскому университету в Беркли за предоставленные возможности для подготовки рукописи.

Июнь 1974

Альфред Ахо
Джон Хопкрофт
Джефри Ульман

От издательства

Вы, читатель этой книги, — ее главный критик и комментатор. Мы ценим ваше мнение и хотим знать, что мы сделали правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо или зайти на наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или оставляя сообщение, не забудьте указать название книги и ее авторов, а также ваш электронный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию следующих книг.

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`
WWW: `http://www.dialektika.com`

ГЛАВА 1

Модели вычислений

Как найти эффективный алгоритм для решения конкретной задачи? Как сравнить найденный алгоритм с другими алгоритмами, решающими ту же задачу? Как оценить его качество? Эти вопросы интересуют и программистов, и теоретиков в области компьютерных наук. В этой книге мы изучим различные подходы к ответам на эти и подобные вопросы.

В этой главе рассматриваются несколько моделей компьютера — машина с произвольным доступом к памяти, машина, хранящая программы в памяти с произвольным доступом, и машина Тьюринга. Эти модели сравниваются по их способности отражать сложность алгоритмов, и на их основе создаются более специализированные модели вычислений: линейные последовательности арифметических операций, битовые вычисления, вычисления с векторами битов и деревья решений. В последнем разделе этой главы приводится описание псевдокода Алгол, на котором написаны все рассматриваемые в книге алгоритмы.

1.1. Алгоритмы и их сложность

Существует множество критериев для оценки алгоритмов. Чаще всего нас будет интересовать скорость увеличения времени и объема памяти при увеличении входных данных, необходимых для решения задачи. Нам хотелось бы связать с каждой конкретной задачей некоторое число, называемое ее *размером*, которое выражало бы меру количества входных данных. Например, размером задачи умножения матриц может быть наибольший размер перемножаемых матриц. Размером задачи о графах может быть число ребер графа.

Количество времени, затрачиваемое алгоритмом в зависимости от размера задачи, называется *временной сложностью* этого алгоритма. Поведение этой сложности в пределе при увеличении размера задачи до бесконечности называется *асимптотической временной сложностью*. Аналогично можно определить *пространственную сложность* и *асимптотическую пространственную сложность алгоритма*.

Именно асимптотическая сложность алгоритма в конечном итоге определяет размер задач, которые можно решить этим алгоритмом. Если алгоритм обрабатывает входы размера n за время cn^2, где c — некоторая константа, то говорят,

Telegram: https://t.me/it_boooks

что временная сложность этого алгоритма имеет порядок $O(n^2)$. Точнее, говорят, что функция $g(n)$ имеет порядок $O(f(n))$, если существует такая константа c, что $g(n) \leq cf(n)$ для всех, кроме конечного (возможно, пустого) множества, неотрицательных значений n.

На первый взгляд, огромный рост скорости вычислений, характерный для нынешнего поколения цифровых компьютеров, должен был уменьшить важность эффективных алгоритмов. Однако на самом деле все наоборот. Поскольку компьютеры работают все быстрее и мы можем решать все более крупные задачи, именно сложность алгоритма определяет то увеличение размера задач, которого можно достичь с увеличением скорости компьютеров.

Допустим, у нас есть пять алгоритмов A_1–A_5 со следующими временными сложностями.

Алгоритм	Временная сложность
A_1	n
A_2	$n \log n$[1]
A_3	n^2
A_4	n^3
A_5	2^n

Здесь временная сложность — это количество единиц времени, требуемого для обработки входных данных размера n. Если единицей времени является миллисекунда, то алгоритм A_1 может обработать за одну секунду входные данные размером 1000, в то время как алгоритм A_5 — входные данные размером не более 9. На рис. 1.1 приведены размеры задач, которые можно решить за одну секунду, одну минуту и один час каждым из этих пяти алгоритмов.

Алгоритм	Временная сложность	Максимальный размер задачи		
		1 с	1 мин	1 час
A_1	n	1000	$6 \cdot 104$	$3{,}6 \cdot 106$
A_2	$n \log n$	140	4893	$2{,}0 \cdot 105$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Рис. 1.1. Границы размеров задач в зависимости от роста сложности

Допустим, что следующее поколение компьютеров будет работать в 10 раз быстрее современного. На рис. 1.2 показано, как возрастут размеры задач, которые мы сможем решить благодаря такому увеличению быстродействия. Как видим, для ал-

[1] Если не указано иное, то все логарифмы в этой книге являются двоичными.

горитма A_5 десятикратное увеличение быстродействия увеличивает максимальный размер задачи, которую он может решить только на три единицы, тогда как для алгоритма A_3 максимальный размер задачи увеличивается более чем в три раза.

Алгоритм	Временная сложность	Максимальный размер задачи до ускорения	Максимальный размер задачи после ускорения
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	Приблизительно $10s_2$ при больших s_2
A_3	n^2	s_3	$3{,}16s_3$
A_4	n^3	s_4	$2{,}15s_4$
A_5	2^n	s_5	$s_5 + 3{,}3$

Рис. 1.2. Эффект десятикратного ускорения

Вместо эффекта ускорения компьютеров рассмотрим теперь эффект применения более эффективного алгоритма. Вернемся к рис. 1.1. Если в качестве основы для сравнения взять 1 минуту, то, заменяя алгоритм A_4 алгоритмом A_3, можно решить задачу, которая в шесть раз больше исходной, а заменяя A_4 на A_2, можно решить задачу, которая в 125 раз больше исходной. Эти результаты намного сильнее, чем двукратное улучшение, достигаемое за счет десятикратного ускорения компьютеров. Если в качестве основы для сравнения взять 1 час, то разница станет еще больше. Отсюда следует, что асимптотическая сложность алгоритма служит важной мерой качества алгоритма, причем ее важность при увеличении скорости вычислений возрастает.

Несмотря на то что наше внимание сосредоточено на порядке роста эффективности, необходимо понимать, что большой порядок роста сложности алгоритма может иметь меньшую константу пропорциональности, чем малый порядок роста сложности другого алгоритма. В таком случае для небольших задач, а возможно, и для всех задач, которые нас интересуют, может оказаться предпочтительнее алгоритм с быстро растущей сложностью. Например, предположим, что временные сложности алгоритмов A_1, A_2, A_3, A_4 и A_5 в действительности равны соответственно $1000n$, $100n \log n$, $10n^2$, n^5 и 2^n. Тогда алгоритм A_5 будет наилучшим для задач размером $2 \le n \le 9$, A_3 — для задач размером $10 \le n \le 58$, A_2 — для $59 \le n \le 1024$, а A_1 — для $n > 1024$.

Прежде чем продолжать обсуждение алгоритмов и их временной сложности, необходимо точно определить модель компьютера для реализации алгоритмов, а также дать определение элементарного шага вычисления. К сожалению, не существует такой вычислительной модели, которая была бы оптимальной во всех ситуациях. Одна из основных трудностей порождается размером машинных слов.

Например, если допустить, что машинное слово может хранить целое число произвольной длины, то всю задачу можно закодировать одним числом в виде одного машинного слова. Если же допустить, что машинное слово имеет конечную длину, то возникают трудности, связанные с простым хранением произвольно больших чисел, не говоря уже о других проблемах, которые часто удается избежать при решении задач небольшого размера. Для каждой задачи необходимо выбрать подходящую модель, которая точно отразит действительное время вычисления на реальном компьютере.

В следующих разделах мы обсудим несколько основных моделей компьютеров, наиболее важными из которых являются машина с произвольным доступом к памяти, машина, хранящая программы в памяти с произвольным доступом, и машина Тьюринга. Эти три модели эквивалентны с точки зрения вычислительных возможностей, но не с точки зрения быстродействия.

Возможно, наиболее важным стимулом к изучению моделей вычисления было желание раскрыть естественную вычислительную сложность различных задач. Нам хотелось бы получить нижние оценки времени вычисления. Чтобы показать невозможность существования алгоритма, выполняющего данное задание менее чем за определенное время, требуется точное и часто очень формальное определение алгоритма. Одним из примеров такого определения служат машины Тьюринга, описанные в разделе 1.6.

Для описания алгоритмов желательно использовать концепции, более естественные и понятные, чем программа для машины с произвольным доступом к памяти, машины, хранящей программы в памяти с произвольным доступом, и машины Тьюринга. Для этой цели мы будем представлять программы псевдокодом на языке Алгол. Именно этот язык будет использоваться во всей книге для описания алгоритмов. Однако, чтобы понимать вычислительную сложность алгоритма, представленного псевдокодом на Алголе, мы должны согласовать его с более формальными моделями. Это будет сделано в последнем разделе настоящей главы.

1.2. Машины с произвольным доступом к памяти

Машина с произвольным доступом к памяти (Random Access Machine — RAM) моделирует компьютер с одним сумматором, в котором команды программы не могут изменяться самостоятельно

Компьютер RAM состоит из входной ленты, предназначенной только для чтения, выходной ленты, предназначенной только для записи, и памяти (рис. 1.3). Входная лента — это последовательность ячеек, каждая из которых может содержать целое число (возможно, отрицательное). Каждый раз, когда с входной ленты считывается символ, ее головка сдвигается на одну ячейку вправо. Выходные

данные заносятся на ленту, состоящую из ячеек (пустых в начальном состоянии) и предназначенную только для записи. Каждый раз, когда в ячейку, на которой установлена головка выходной ленты, записывается символ, ее головка сдвигается на одну ячейку вправо. После записи выходной символ больше нельзя изменить.

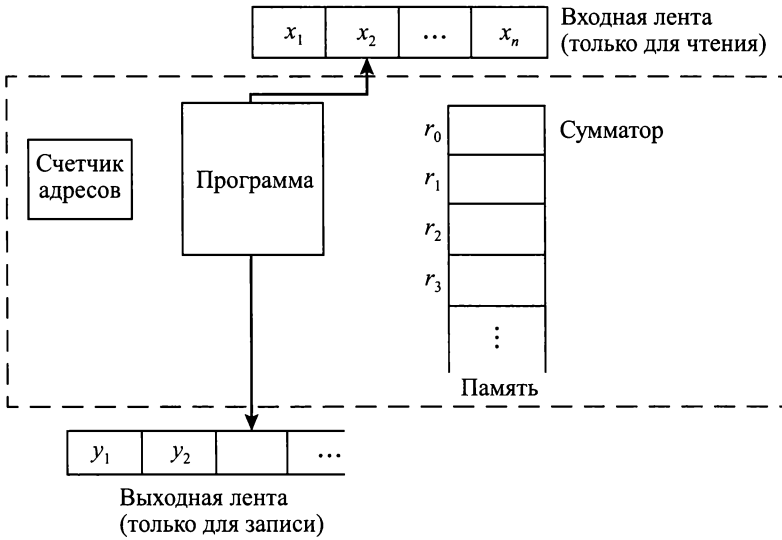


Рис. 1.3. Машина с произвольным доступом к памяти

Память состоит из последовательности регистров r_0, r_1, ..., r_i, ..., каждый из которых может хранить произвольное целое число. Количество регистров, которые можно использовать, не ограничено. Такая идеализация допустима в следующих случаях.

1. Размер задачи достаточно мал, чтобы она поместилась в основную память компьютера.
2. Целые числа, используемые в вычислении, достаточно малы, чтобы их можно было поместить в одном машинном слове.

Программа для компьютера RAM (или RAM-программа) не хранится в памяти. Поэтому мы предполагаем, что программа не может изменяться самостоятельно. По сути, программа является последовательностью (возможно) помеченных команд. Точная природа команд, допустимых в программе, не очень важна, поскольку они напоминают обычные команды в реальных компьютерах. Мы предполагаем, что имеются арифметические команды, команды ввода-вывода, косвенная адресация (например, для индексации массивов) и команды разветвления.

Все вычисления производятся в первом регистре r_0, называемом *сумматором*, который, как и всякий другой регистр памяти, может хранить произвольное целое число. Пример набора команд для компьютера RAM показан в таблице на рис. 1.4. Каждая команда состоит из двух частей — *кода операции* и *адреса*.

	Код операции	Адрес
1.	LOAD	операнд
2.	STORE	операнд
3.	ADD	операнд
4.	SUB	операнд
5.	MCLT	операнд
6.	DIV	операнд
7.	READ	операнд
8.	WRITE	операнд
9.	JUMP	метка
10.	JGTZ	метка
11.	JZERO	метка
12.	HALT	

Рис. 1.4. Таблица команд RAM

В принципе в этот набор можно было бы включить любые другие команды, встречающиеся в реальных компьютерах, например логические или символьные операции, никак не влияя на порядок сложности задач. Читатель может мысленно дополнить этот набор команд по своему усмотрению.

Операнд может иметь один из следующих типов.

1. $= i$, означает само целое число i.
2. Неотрицательное целое число i — содержимое регистра i.
3. $*i$ — косвенная адресация, т.е. значение операнда представляет собой содержимое регистра j, где j — целое число, находящееся в регистре i; если $j < 0$, машина останавливается.

Эти команды хорошо знакомы всем, кто программировал на языке ассемблера. Значение программы P можно определить с помощью двух объектов: отображения c из множества неотрицательных целых чисел в множество целых чисел и счетчика адресов, определяющего очередную выполняемую команду. Функция c — это *отображение памяти*, т.е. $c(i)$ — это целое число, содержащееся в регистре i (*содержимое* регистра i).

В начальный момент времени $c(i) = 0$ для всех $i \geq 0$, счетчик адресов установлен на первую команду в программе P, а выходная лента пуста. После выполнения k-й

команды программы P счетчик адресов автоматически переходит к команде $k + 1$ (т.е. к очередной команде), если k-я команда не была командой JUMP, HALT, JGTZ или JZERO.

Чтобы описать действие команды, определим *значение* $v(a)$ *операнда* a:

$$v(=i) = i,$$
$$v(i) = c(i),$$
$$v(*i) = c(c(i)).$$

Действие каждой команды из рис. 1.4 описано на рис. 1.5. Команды, не указанные в таблице (такие, как STORE = i), можно считать эквивалентными команде HALT. Аналогично деление на нуль останавливает машину.

	Код операции	Адрес
1.	LOAD a	$c(0) \leftarrow v(a)$
2.	STORE i	$c(i) \leftarrow c(0)$
	STORE $*i$	$c(c(i)) \leftarrow c(0)$
3.	ADD a	$c(0) \leftarrow c(0) + v(a)$
4.	SUB a	$c(0) \leftarrow c(0) - v(a)$
5.	MULT a	$c(0) \leftarrow c(0) \cdot v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0) / v(a) \rfloor$ †
7.	READ i	$c(i) \leftarrow$ текущий входной символ
	READ $*i$	$c(c(i)) \leftarrow$ текущий входной символ. В обоих случаях головка входной ленты перемещается на одну ячейку вправо
8.	WRITE a	Значение $v(a)$ записывается в ячейку выходной ленты, на которую установлена головка в текущий момент. Затем головка ленты перемещается на одну ячейку вправо
9.	JUMP b	Счетчик адресов устанавливается на команду с меткой b
10.	JGTZ b	Счетчик адресов устанавливается на команду с меткой b, если $c(0) > 0$, в противном случае счетчик адресов устанавливается на следующую команду
11.	JZERO b	Счетчик адресов устанавливается на команду с меткой b, если $c(0) = 0$, в противном случае счетчик адресов устанавливается на следующую команду
12.	HALT	Выполнение программы прекращается

† Во всей книге $\lceil x \rceil$ (округление числа x с избытком) означает наименьшее целое число, большее или равное x, а $\lfloor x \rfloor$ (округление числа x с недостатком) означает наибольшее целое число, меньшее или равное x.

Рис. 1.5. Описание команд компьютера RAM. Операнд a — это $=i$, i или $*i$

При выполнении любой из первых восьми команд счетчик адресов увеличивается на единицу. Следовательно, команды программы выполняются одна за другой до тех пор, пока не встретится команда JUMP или HALT, или JGTZ, когда содержимое сумматора больше нуля, или JZERO, когда содержимое сумматора равно нулю.

Вообще говоря, программа RAM определяет отображение из множества входных лент в множество выходных лент. Поскольку на некоторых входных лентах программа может не останавливаться, это отображение является частичным (т.е. для некоторых входных данных оно может быть не определено). Это отображение можно интерпретировать по-разному. Важными являются две интерпретации — в виде функции и в виде языка.

Допустим, программа P всегда считывает с входной ленты n целых чисел и записывает на выходную ленту не более одного целого числа. Пусть $x_1, x_2, ..., x_n$ — целые числа, находящиеся в n первых ячейках входной ленты, и программа P записывает число y в первую ячейку выходной ленты, а затем через некоторое время останавливается. Тогда говорят, что программа P вычисляет функцию $f(x_1, x_2, ..., x_n) = y$. Легко показать, что модель RAM, как и любая другая разумная модель компьютера, может вычислять не что иное, как *частично рекурсивные функции*. Иначе говоря, для произвольной частично рекурсивной функции f можно написать RAM-программу, вычисляющую f, и для произвольной RAM-программы можно указать эквивалентную ей частично рекурсивную функцию. (Частично рекурсивным функциям посвящены монографии Дэвиса [1958] и Роджерса [1967].)

Другой способ интерпретировать RAM-программу — это посмотреть на нее как на средство восприятия языка. *Алфавит* — это конечное множество символов, а *язык* — множество строк, состоящих из символов этого алфавита. Символы алфавита можно представить целыми числами $1, 2, ..., k$ при некотором k. Компьютер RAM воспринимает язык в следующем смысле. Пусть на входной ленте находится строка $s = a_1 a_2 ... a_n$, причем символ a_1 расположен в первой ячейке, a_2 — во второй и т.д., а в $(n + 1)$-й ячейке записан символ 0, который используется в качестве признака конца входной строки.

Входная строка s *воспринимается* RAM-программой P, если P прочитывает все ее символы и признак конца, записывает единицу в первую ячейку выходной ленты и останавливается. Язык, *воспринимаемый программой* P, — это множество всех воспринимаемых ею строк. Для входных строк, не принадлежащих языку, воспринимаемому программой P, она может напечатать на выходной ленте символ, отличный от единицы, и остановиться или же не остановиться вообще. Легко показать, что язык воспринимается некоторым компьютером RAM тогда и только тогда, когда он *рекурсивно перечислим*. Язык допускается компьютером RAM, останавливающимся на всех входных данных, тогда и только тогда, когда он *рекурсивен* (о рекурсивных и рекурсивно перечислимых языках см. Хопкрофт, Ульман [1969]).

Рассмотрим две программы для компьютера RAM. Первая определяет функцию, вторая воспринимает язык.

Пример 1.1. Пусть функция $f(n)$ определена следующим образом:

$$f(n)=\begin{cases} n^n \text{ для всех целых } n \geq 1, \\ 0 \text{ в противном случае.} \end{cases}$$

Программа, представленная псевдокодом на Алголе, которая вычисляет $f(n)$, умножая число n на себя $(n-1)$ раз, приведена на рис. 1.6.[2] Соответствующая программа для компьютера RAM приведена на рис. 1.7. Переменные `r1`, `r2` и `r3` хранятся в регистрах 1, 2 и 3 соответственно. Мы специально не пытались оптимизировать программы, чтобы подчеркнуть соответствие рис. 1.6 и 1.7. □

```
begin
     read r1;
     if r1 ≤ 0 then write 0
     else
          begin
               r2 ← r1;
               r3 ← r1 - 1;
               while r3 > 0 do
                    begin
                         r2 ← r2 * r1;
                         r3 ← r3 - 1
                    end;
                write r2
          end
end
```

Рис. 1.6. Программа вычисления n^n, представленная псевдокодом на Алголе

Пример 1.2. Рассмотрим RAM-программу, которая воспринимает язык на основе входного алфавита $\{1, 2\}$, состоящий из всех строк с одинаковым числом вхождений символов 1 и 2. Эта программа считывает каждый входной символ в регистр 1, а в регистре 2 хранит разность d между количеством символов 1 и 2, поступивших до текущего момента. Встретив признак конца, равный 0, программа сравнивает d с нулем и в случае совпадения выводит на печать символ 1 и останавливается. Символы 0, 1 и 2 считаются единственно возможными входными символами.

[2] Описание псевдо-Алгола см. в разделе 1.8.

	Программа RAM		Соответствующие инструкции псевдокода на Алголе
	READ	1	**read** *r*1
	LOAD	1	
	JGTZ	pos	**if** *r*1 ≤ 0 **then write** 0
	WRITE	=0	
	JUMP	endif	
pos:	LOAD	1	
	STORE	2	*r*2 ← *r*1
	LOAD	1	
	SUB	=1	*r*3 ← *r*1 - 1
	STORE	3	
while:	LOAD	3	
	JGTZ	continue	**while** *r*3 > 0 **do**
	JUMP	endwhile	
continue:	LOAD	2	
	MULT	1	*r*2 ← *r*2 * *r*1
	STORE	2	
	LOAD	3	
	SUB	=1	*r*3 ← *r*3 - 1
	STORE	3	
	JUMP	while	
endwhile:	WRITE	2	**write** *r*2
endif:	HALT		

Рис. 1.7. RAM-программа для вычисления $n!$

Основные детали алгоритма приведены в программе на рис. 1.8. Эквивалентная программа для компьютера RAM показана на рис. 1.9; число x хранится в регистре 1, число d — в регистре 2. □

```
begin
    d ← 0;
    read x;
    while x ≠ 0 do
        begin
            if x ≠ 1 then d ← d - 1 else d ← d + 1;
            read x
        end;
     if d = 0 then write 1
end
```

Рис. 1.8. Распознавание строк с одинаковыми числами вхождений символов 1 и 2

	Программа RAM		Соответствующие инструкции псевдокода на Алголе
	LOAD	=0	d ← 0
	STORE	2	
	READ	1	**read** *x*
while:	LOAD	1	
	JZERO	endwhile	**while** *x* ≠ 0 **do**
	LOAD	1	
	SUB	=1	**if** *x* ≠ 0
	JZERO	one	
	LOAD	2	
	SUB	=1	**then** *d* ← *d* - 1
	STORE	2	
	JUMP	endif	
one:	LOAD	2	
	ADD	=1	**else** *d* ← *d* + 1
	STORE	2	
endif:	READ	1	**read** *x*
	JUMP	while	
endwhile:	LOAD	2	
	JZERO	output	
	HALT		**if** *d* = 0 **then write** 1
output:	WRITE	=1	
	HALT		

Рис. 1.9. RAM-программа, соответствующая алгоритму, приведенному на рис. 1.8

1.3. Вычислительная сложность RAM-программ

Двумя важными мерами сложности алгоритма являются временная и пространственная, рассматриваемые как функции, зависящие от размера входных данных. Если при данном размере в качестве меры сложности выбирается наибольшая (по всем входным данным заданного размера), то она называется *сложностью в худшем случае*. Если в качестве меры сложности выбирается сложность, усредненная по всем входным данным заданного размера, то она называется *средней сложностью*. Обычно среднюю сложность вычислить труднее, чем сложность в худшем случае. Необходимо также сделать некоторое предположение о распределении входных данных, а реалистичные допущения часто трудно поддаются математической формализации. Мы будем уделять основное внимание худшему случаю, поскольку он проще и имеет универсальный характер. Однако следует помнить, что алгоритм с наименьшей сложностью в худшем случае не обязательно имеет наименьшую сложность в среднем.

Временная сложность в худшем случае (или просто временная сложность) RAM-программы — это функция $f(n)$, равная наибольшей сумме времени, затраченного на каждую выполненную команду (по всем входным данным размера n). *Средняя временная сложность* — это среднее тех же сумм по всем входным данным размера n. Такие же понятия определяются для объема памяти, только вместо "времени, затраченного на каждую выполненную команду" необходимо подставить "объем памяти всех регистров, к которым было выполнено обращение".

Чтобы точно определить временную и пространственную сложности, необходимо указать время, затрачиваемое на выполнение каждой RAMкоманды, и объем памяти, используемый каждым регистром. Мы рассмотрим две *функции стоимости* для наших программ. При использовании *равномерной функции стоимости* на выполнение каждой RAM-команды затрачивается одна единица времени и каждый регистр занимает одну единицу памяти. Если не указано иное, то сложность RAM-программы будет измеряться в соответствии с равномерной функцией стоимости.

Второе определение, иногда более реалистичное, учитывает ограниченность размера реального слова памяти и называется *логарифмической функцией стоимости*. Пусть $l(i)$ — *логарифмическая функция*, определенная на целых числах:

$$l(i) = \begin{cases} \lfloor \log|i| \rfloor + 1, & i \neq 0, \\ 1, & i = 0. \end{cases}$$

Таблица, приведенная на рис. 1.10, содержит логарифмические затраты $T(a)$ для всех трех возможных видов операнда a. На рис. 1.11 приведены временные затраты для выполнения каждой команды.

Операнд a	Затраты $T(a)$
$=i$	$l(i)$
i	$l(i) + l(c(i))$
$*i$	$l(i) + l(c(i)) + l(c(c(i)))$

Рис. 1.10. Логарифмические затраты, связанные с операндом

Затраты учитывают, что для представления целого числа n в регистре требуется $\lfloor \log n \rfloor + 1$ битов. Напомним, что регистры могут содержать произвольно большие целые числа.

Логарифмическая функция стоимости основана на грубом допущении, что затраты на выполнение команды пропорциональны длине ее операндов. Например, рассмотрим стоимость команды ADD $*i$. Сначала мы должны определить затраты, связанные с декодированием операнда, представленного адресом. Проверка целого числа i занимает время $l(i)$. Затем, чтобы прочитать содержимое $c(i)$ регистра i и найти

Инструкция	Затраты
1. LOAD a	$T(a)$
2. STORE i	$l(c(0)) + l(i)$
STORE *i	$l(c(0)) + l(i) + l(c(i))$
3. ADD a	$l(c(0)) + T(a)$
4. SUB a	$l(c(0)) + T(a)$
5. MULT a	$l(c(0)) + T(a)$
6. DIV a	$l(c(0)) + T(a)$
7. READ i	l(вход) $+ l(i) + l(i)$
READ *i	l(вход) $+ l(i) + l(c(i))$
8. WRITE a	$T(a)$
9. JUMP b	1
10. JGTZ b	$l(c(0))$
11. JZERO b	$l(c(0))$
12. HALT	1

Рис. 1.11. Логарифмические затраты, связанные с выполнением RAM-инструкций, где $T(a)$ — затраты, связанные с операндом a, b — метка

регистр $c(i)$, необходимо $l(c(i))$ единиц времени. Наконец, считывание содержимого регистра $c(i)$ требует $l(c(c(i)))$ единиц времени. Поскольку команда ADD *i прибавляет целое число $c(c(i))$ к целому числу $c(0)$ в сумматоре, то ясно, что стоимостью выполнения команды ADD *i разумно считать $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$.

Логарифмической пространственной сложностью RAM-программы будем называть сумму по всем регистрам, включая сумматор, величин $l(x_i)$, где x_i —наибольшее по абсолютной величине целое число, хранившееся в i-м регистре за все время вычислений.

Естественно, конкретная программа может иметь совершенно разную временную сложность в зависимости от используемой функции стоимости — равномерной или логарифмической. Если разумно предположить, что каждое число, встречающееся в задаче, можно хранить в виде одного машинного слова, то целесообразно использовать равномерную функцию стоимости. В иной ситуации для реалистического анализа сложности более подходящим может оказаться логарифмическая функция стоимости.

Оценим временную и пространственную сложности RAM-программы, вычисляющей n^n из примера 1.1. Основное влияние на временную сложность оказывает цикл с командой MULT. Когда эта команда выполняется i-й раз, сумматор содержит число n^i, а регистр 2 содержит число n. Всего команда MULT выполняется $n - 1$ раз. При равномерной функции стоимости на выполнение каждой команды MULT потребуется одна единица времени, поэтому на выполнение всех команд MULT будет

затрачено $O(n)$ времени. При логарифмической функции стоимости на выполнение i-й команды MULT потребуется $l(n^i) + l(n) \approx (i + 1)\log n$, так что время выполнения всех команд MULT равно

$$\sum_{i=1}^{n-1}(i+1)\log n,$$

т.е. $O(n2\log n)$.

Пространственная сложность определяется числами, которые хранились в регистрах от 0 до 3. При равномерной функции стоимости пространственная сложность составляет $O(1)$, а при логарифмической — $O(n \log n)$, поскольку наибольшее целое число, хранившееся в этих регистрах равно, n^n, а $l(n^n) \approx n \log n$. Таким образом, временная и пространственная сложности программы из примера 1.1 имеют следующий порядок:

	Равномерная функция	**Логарифмическая функция**
Временная сложность	$O(n)$	$O(n2\log n)$
Пространственная сложность	$O(1)$	$O(n \log n)$

Для этой программы использование равномерной функции стоимости является целесообразным, только если число n^n можно представить с помощью одного машинного слова. Если число n^n превышает размеры одного машинного слова, то даже логарифмическая временная сложность в определенном смысле нереалистична, поскольку она предполагает, что два целых числа, i и j, перемножаются за время $O(l(i)+l(j))$, а возможно ли это — неизвестно.

Для программы из примера 1.2 в предположении, что n — это длина входного слова, временная и пространственная сложность имеют следующий порядок:

	Равномерная функция	**Логарифмическая функция**
Временная сложность	$O(n)$	$O(n \log n)$
Пространственная сложность	$O(1)$	$O(\log n)$

Если число n невозможно записать в одном машинном слове, то для этой программы вполне реалистичной является логарифмическая функция стоимости.

1.4. Модель машины, хранящей программу в памяти

Поскольку RAM-программа не хранится в памяти RAM, она не может изменяться самостоятельно. Рассмотрим другую модель компьютера, называемую *машиной, хранящей программу в памяти с произвольным доступом* (Random Access

Stored Program — RASP), которая отличается от компьютера RAM лишь тем, что его программа хранится в памяти и может изменяться самостоятельно.

Набор команд для компьютера RASP совпадает с соответствующим набором команд для компьютера RAM за исключением команды косвенной адресации, которая не нужна больше. Далее будет показано, что компьютер RASP может моделировать косвенную адресацию с помощью изменения команд в процессе выполнения программы.

Общая структура компьютера RASP аналогична структуре компьютера RAM, за исключением того, что RASP-программа хранится в регистрах памяти. Каждая RASP-команда занимает два последовательных регистра памяти. Первый регистр содержит код операции, второй — адрес. Если адрес имеет вид $=i$, то первый регистр будет содержать (в закодированном виде) информацию о том, что операнд является литералом, а второй регистр будет содержать i. Для кодирования команд используются целые числа. На рис. 1.12 представлен один из возможных способов кодирования. Например, команда `LOAD =32` должна храниться в виде числа 2 в одном регистре и в виде числа 32 в следующем регистре.

Команда	Код	Команда	Код
`LOAD` i	1	`DIV` i	10
`LOAD` $=i$	2	`DIV` $=i$	11
`STORE` i	3	`READ` i	12
`ADD` i	4	`WRITE` i	13
`ADD` $=i$	5	`WRITE` $=i$	14
`SUB` i	6	`JUMP` i	15
`SUB` $=i$	7	`JGTZ` i	16
`MULT` i	8	`JZERO` i	17
`MULT` $=i$	9	`HALT`	18

Рис. 1.12. Кодирование команд RASP

Так же как для компьютера RAM, состояние компьютера RASP можно представить с помощью следующих инструментов.

1. Отображение памяти c, где $c(i)$, $i \geq 0$, — содержимое регистра i.
2. Счетчик адресов, указывающий на первый из двух последовательных регистров памяти, из которых надлежит взять текущую команду.

Вначале счетчик адресов устанавливается на некоторый специальный регистр. В исходном состоянии содержимое регистров памяти, как правило, не состоит из одних нулей, поскольку в самом начале в память уже введена программа. Однако мы требуем, чтобы в исходном состоянии все регистры, за исключением некоторых, содержали 0, и чтобы сумматор также содержал 0. После выполнения ка-

ждой команды счетчик адресов всегда увеличивается на 2, за исключением команд JUMP i, JGTZ i (при положительном сумматоре) и JZERO i (при нулевом сумматоре), при выполнении которых он устанавливается на регистр i. Действие каждой команды RASP в точности совпадает с действием команды RAM.

Временную сложность RASP-программы можно определить, по существу, тем же способом, что и для RAM-программы. Можно использовать либо равномерную, либо логарифмическую функцию стоимости. Правда, в последнем случае необходимо учесть затраты не только на вычисление операнда, но и на выбор самой команды. Затраты, связанные с выбором команды, равны l(LC), где LC означает содержимое счетчика адресов. Например, затраты на выполнение команды ADD =i, хранящейся в регистрах j и $j + 1$, равны $l(j) + l(c(0)) + l(i)$.[3] Затраты на выполнение команды ADD i, хранящейся в регистрах j и $j + 1$, равны $l(j) + l(c(0)) + l(i) + l(c(i))$.

Интересно, каково различие в сложности между RAM-программой и соответствующей RASP-программой. Ответ очевиден. Независимо от функции стоимости — равномерной или логарифмической — любое отображение вход-выход, выполняемое за время $T(n)$ одной моделью, другая модель может выполнить за время $kT(n)$, где k — некоторая константа. Аналогично объемы памяти, используемые этими моделями, при любой из двух рассматриваемых функций стоимости отличаются один от другого только на постоянный множитель.

Сформулируем эти соотношения в виде двух теорем, которые доказываются построением алгоритмов, моделирующих машину RAM с помощью машины RASP, и наоборот.

Теорема 1.1. Как при равномерной, так и при логарифмической функции стоимости для любой RAM-программы с временной сложностью $T(n)$ существует такая константа k, что найдется эквивалентная RASP-программа, временная сложность которой не превосходит $kT(n)$.

Доказательство. Покажем, как моделировать RAM-программу P с помощью RASP-программы. Регистр 1 в машине RASP будет служить для временного запоминания содержимого сумматора RAM. На основе программы P мы создадим RASP-программу P_S, которая будет занимать следующие $r - 1$ регистров RASP. Константа r определяется RAM-программой P. Содержимое регистра RAM с номером i, $i \geq 1$, будет храниться в регистре машины RASP с номером $r + i$, так что все адреса в RASP-программе будут на r больше соответствующих адресов в RAM-программе.

---

[3] Можно было бы также учесть затраты, связанные с чтением регистра $j + 1$, но они не могут сильно отличаться $l(j)$. В этой главе нас интересует не величина постоянных множителей, а лишь порядок роста функций. Следовательно, $l(j) + l(j + 1)$ "почти" равно $l(j)$, иначе говоря, отличается от него не более чем в три раза.

Каждая RAM-команда в программе P, не содержащая косвенной адресации, прямо кодируется в такую же RASP-команду (с соответствующим увеличением адресов). Каждая RAM-команда в программе P, содержащая косвенную адресацию, переводится в последовательность из шести RASP-команд, которые моделируют косвенную адресацию путем изменения команд.

Рассмотрим пример моделирования косвенной адресации. Для моделирования RAM-команды `SUB` $*i$, где i — положительное целое число, построим последовательность RASP-команд, которые выполняют следующие действия.

1. Временно запоминают содержимое сумматора в регистре 1.
2. Загружают содержимое регистра $r + i$ в сумматор (RASP-регистр с номером $r + i$ соответствует RAM-регистру с номером i).
3. Прибавляют r к сумматору.
4. Запоминают число, вычисленное на шаге 3 в адресном поле команды SUB.
5. Восстанавливают сумматор из временного регистра 1.
6. Используют команду `SUB`, созданную в п. 4, для выполнения вычитания.

Например, применяя кодирование команд RASP, приведенное на рис. 1.12, и предполагая, что последовательность RASP-команд начинается в регистре 100, можно смоделировать команду `SUB` $*i$ с помощью последовательности, показанной на рис. 1.13. Сдвиг r можно будет определить, когда станет известно количество RASP-команд в программе P_S.

Регистр	Содержимое		Значение
100	3	`STORE`	1
101	1		
102	1	`LOAD`	$r + i$
103	$r + i$		
104	5	`ADD`	$= r$
105	r		
106	3	`STORE`	111
107	111		
108	1	`LOAD`	1
109	1		
110	6	`SUB`	b, где b — содержимое i-го регистра RAM
111	–		

Рис. 1.13. Моделирование команды `SUB` $*i$ с помощью компьютера RASP

Легко видеть, что для моделирования каждой RAM-команды требуется не более шести RASP-команд, так что при равномерной функции стоимости временная сложность получаемой RASP-программы не превосходит $6T(n)$. (Заметим, что эта оценка не зависит от того, каким способом определен размер входных данных.)

При использовании логарифмической функции стоимости каждая команда I из RAM-программы P моделируется последовательностью S, состоящей либо из одной, либо из шести RASP-команд в программе P_S. Можно показать, что существует константа k, зависящая от программы P, такая, что суммарные затраты на выполнение команд в программе S не более чем в k раз превосходят затраты на выполнение команды I.

Например, затраты на выполнение команды `SUB` $*i$ для RAM-компьютера равны

$$M = l(c(0)) + l(i) + l(c(i)) + l(c(c(i))).$$

Последовательность S, моделирующая эту команду RAM, показана на рис. 1.14. Здесь $c(0)$, $c(i)$ и $c(c(i))$ относятся к содержимому регистров RAM. Поскольку программа P_S занимает в компьютере RASP регистры от 2 до r, то $j \le r - 11$. Кроме того, $l(x+y) \le l(x) + l(y)$, так что затраты на выполнение программы S определенно не превышают

$$2l(1) + 4M + 11l(r) < (6 + 11l(r))M.$$

Следовательно, константа $k = 6 + 11l(r)$ такова, что если программа P имеет временную сложность $T(n)$, то временная сложность программы P_S не превосходит $kT(n)$. □

RASP-регистр	Команда		Затраты
j	`STORE`	1	$l(j) + l(1) + l(c(0))$
$j+2$	`LOAD`	$r+i$	$l(j+2) + l(r+i) + l(c(i))$
$j+4$	`ADD`	$=r$	$l(j+4) + l(c(i)) + l(r)$
$j+6$	`STORE`	$j+11$	$l(j+6) + l(j+11) + l(c(i)+r)$
$j+8$	`LOAD`	1	$l(j+8) + l(1) + l(c(0))$
$j+10$	`SUB`	–	$l(j+10) + l(c(i)+r) + l(c(0)) + l(c(c(i)))$

Рис. 1.14. Кодирование команд RASP

Теорема 1.2. Как при равномерной, так и при логарифмической функции стоимости на выполнение команд любой RASP-программы с временной сложностью $T(n)$ существует такая константа k, что найдется эквивалентная RAM-программа, временная сложность которой не превосходит $kT(n)$.

Доказательство. RAM-программа, которую мы построим для моделирования заданной RASP-программы, будет использовать косвенную адресацию для декодирования и моделирования RASP-команд, хранящихся в памяти RAM. Некоторые регистры RAM будут иметь специальное назначение:

регистр 1 — для косвенной адресации;

регистр 2 — для счетчика адресов RASP;

регистр 3 — для хранения содержимого сумматора RASP.

При $i \geq 0$ RASP-регистр с номером i будет храниться в RAM-регистре с номером $i + 3$.

RAM-программа начинает работу с RASP-программы конечной длины, расположенной в ее памяти, начиная с регистра 4 и далее. Регистр 2 (счетчик адресов) содержит число 4, регистры 1 и 3 — число 0. RAMпрограмма состоит из цикла моделирования, начинающегося со считывания RASP-команды (с помощью RAM-команды `LOAD *2`) , декодирования ее и разветвления на один из 18 наборов команд, каждый из которых предназначен для обработки одного типа RASP-команды. На неправильном коде операции RAM, как и RASP, программа остановится.

Операции декодирования и разветвления строятся естественным образом; моделью может служить пример 1.2 (хотя символ, декодируемый там, считывался из входных данных, а в данном случае — из памяти). В качестве примера приведем те RAM-команды, которые моделируют RASP-команду с кодом 6, т. е. `SUB` *i*. Эта программа, изображенная на рис. 1.15, вызывается, если $c(c(2)) = 6$, т.е. когда счетчик адресов указывает на регистр, содержащий число 6 — код команды `SUB`.

Дальнейшие детали построения необходимой RAM-программы мы опустим. В качестве упражнения читатели могут доказать, что при равномерной и логарифмической функциях стоимости временная сложность RAM-программы не более чем в постоянное число раз превосходит временную сложность исходной RASP-программы. □

Из теорем 1.1 и 1.2 следует, что с точки зрения временной сложности (а также пространственной — оставим доказательство этого факта в качестве упражнения) модели RAM и RASP эквивалентны с точностью до постоянного множителя, т.е. порядки их сложностей одинаковы для одного и того же алгоритма. Обычно из этих двух моделей мы будем выбирать модель RAM, поскольку она проще.

1.5. Модификации модели RAM

Модели PAM и RASP сложнее, чем это часто бывает необходимо. Поэтому мы введем ряд упрощенных моделей, которые наследуют одни свойства RAM и игнорируют другие. В качестве обоснования этого решения отметим, что суммарные

```
LOAD    2
ADD    =1      Приращение счетчика адресов на 1, чтобы он указывал
STORE   2       на регистр, содержащий операнд i инструкции SUB i
LOAD   *2
ADD    =3      Записываем число i в накопитель, добавляем 3
STORE   1       и сохраняем результат в 1 регистр
LOAD    3
SUB    *1      Извлекаем содержимое накопителя RASP из регистра 3.
STORE   3       Вычитаем содержимое регистра i + 3 и записываем результат
                назад в регистр 3
LOAD    2
ADD    =1      Увеличиваем счетчик адресов на 1 снова, чтобы он указывал0
STORE   2       на следующую инструкцию RASP
JUMP    a       Возвращаемся в начало цикла моделирования (с именем "a")
```

Рис. 1.15. Моделирование команды SUB *i* командами RAM

затраты игнорируемых команд не превосходят некоторой фиксированной доли затрат любого эффективного алгоритма для задач, к которым применяется данная модель.

I. Линейные программы

Первая наша модель — линейная программа. Для многих задач разумно ограничиться рассмотрением класса RAM-программ, в которых команды разветвления используются только для того, чтобы повторить последовательность команд, причем число повторений пропорционально размеру входных данных n. В этом случае можно развернуть программу для каждого размера данных n, копируя повторяющиеся команды столько, сколько необходимо. В результате получается последовательность линейных программ (т.е. программ без циклов), вообще говоря, возрастающей длины — по одной программе для каждого значения n.

Пример 1.3. Рассмотрим умножение двух целочисленных матриц размером $n \times n$. Естественно ожидать, что в RAM-программе число выполнений цикла не будет зависеть от фактических значений элементов матрицы. Следовательно, в качестве полезного упрощения можно считать, что допускаются только такие циклы, у которых проверка конца зависит лишь от n, т.е. от размера задачи. Например, обычный алгоритм умножения матриц содержит циклы, которые следует выполнить точно n раз, при этом от команд разветвления требуется только сравнение параметра цикла с числом n. □

Развертывание циклов в линейную последовательность команд позволяет обходиться без разветвления. Оправданием служит то, что во многих задачах сложность команд, управляющих циклом, не превышает постоянную долю сложности

работы всей RAM-программы. Аналогично часто можно допускать, что операторы ввода составляют лишь постоянную долю сложности работы программы, и мы устраняем их, допуская, что перед началом выполнения программы в памяти находится конечное множество входных данных, требуемых при данном n. Действие косвенной адресации можно определить для фиксированного n, предположив, что регистры, используемые для нее, содержат величины, зависящие только от n и не зависящие от значений входных переменных. Таким образом, будем считать, что наши линейные программы не имеют команд косвенной адресации.

Кроме того, поскольку каждая линейная программа может обращаться только к конечному числу регистров памяти, удобно присвоить этим регистрам имена. По этой причине при ссылке на регистры мы будем употреблять *символьные адреса* (символы или строки), а не целые числа.

Устранив потребность в командах `READ`, `JUMP`, `JGTZ` и `JZERO`, мы остаемся с командами `LOAD`, `STORE`, `WRITE`, `HALT` и арифметическими операциями из системы команд RAM. Нам не нужна команда `HALT`, поскольку на остановку указывает конец программы. Без команды `WRITE` тоже можно обойтись, назначив в качестве *выходных переменных* определенные символьные адреса; результатом работы программы будет значение, принимаемое этими переменными в конце ее работы.

Наконец, можно объединить команды `LOAD` и `STORE` с арифметическими операциями, заменяя последовательности вида

```
LOAD   a
ADD    b
STORE  c
```

на $c \leftarrow a + b$. Весь набор команд линейной программы выглядит так:

$$x \leftarrow y + z,$$
$$x \leftarrow y - z,$$
$$z \leftarrow y * z,$$
$$z \leftarrow y / z,$$
$$x \leftarrow i.$$

где x, y и z — символьные адреса (или *переменные*), а i — константа. Легко видеть, что любую последовательность команд `LOAD`, `STORE` и арифметических операций в сумматоре можно заменить последовательностью, составленной из пяти приведенных выше команд.

С линейной программой связаны два специальных набора переменных — *входных* и *выходных*. Функцией, вычисляемой конкретной линейной программой, называется множество значений выходных переменных, выраженных через значения ее входных переменных в определенном порядке.

Пример 1.4. Рассмотрим вычисление полинома

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

Входными переменными служат коэффициенты a_0, a_1, ..., a_n и неопределенная переменная x. Выходной переменной будет p. По правилу Горнера полином $p(x)$ вычисляется так:

1. $a_1 x + a_0$ для $n = 1$,
2. $(a_2 x + a_1) x + a_0$ для $n = 2$,
3. $((a_3 x + a_2) x + a_1) x + a_0$ для $n = 3$.

Линейные программы, соответствующие этим выражениям, приведены на рис. 1.16. Правило Горнера для произвольного n теперь становится понятным. Для каждого n у нас есть линейная программа из $2n$ шагов, вычисляющая полином n-й степени. В главе 12 мы покажем, что для вычисления произвольного полинома n-й степени по его коэффициентам требуется n умножений и n сложений. Таким образом, если в качестве модели брать линейные программы, то правило Горнера является оптимальным. □

$n = 1$	$n = 2$	$n = 3$
$t \leftarrow a_1 * x$	$t \leftarrow a_2 * x$	$t \leftarrow a_3 * x$
$p \leftarrow t + a_0$	$t \leftarrow t + a_1$	$t \leftarrow t + a_2$
	$t \leftarrow t * x$	$t \leftarrow t * x$
	$p \leftarrow t + a_0$	$t \leftarrow t + a_1$
		$t \leftarrow t * x$
		$p \leftarrow t + a_0$

Рис. 1.16. Линейные программы, соответствующие правилу Горнера

Если в качестве модели вычислений используется линейная программа, то временн*ая* сложность последовательности программ равна числу шагов n-й программы как функции от n. Например, правило Горнера порождает последовательность с временной сложностью $2n$. Заметим, что измерение временн*ой* сложности есть не что иное, как подсчет количества арифметических операций. Пространственная сложность последовательности программ равна числу переменных, задействованных в программе, снова выраженному как функция от n. Пространственная сложность программы из примера 1.4 равна $n + 4$.

Определение. Если в качестве модели вычислений используются линейные программы, говорят, что задача имеет временную или пространственную сложность $O_A(f(n))$, если найдется последовательность программ для ее решения с временной или пространственной сложностью не более чем $cf(n)$ для некоторой константы c. (Выражение $O_A(f(n))$ означает "порядка $f(n)$ шагов линейной программы". Индекс A снизу обозначает "арифметический" — это основная характеристика линейных программ.) Таким образом, вычисление полинома имеет временную, а также и пространственную сложность порядка $O_A(n)$.

II. Битовые вычисления

Очевидно, что модель линейных программ основана на равномерной функции затрат. Как мы уже отмечали, эта функция приемлема, если все вычисляемые величины имеют умеренный размер. Существует простая модификация модели линейных программ, которая соответствует логарифмической функции стоимости. Эта модель, называемая *битовым вычислением*, по существу, является той же линейной программой, за исключением того, что в ней выполняются следующие условия.

1. Все переменные принимают значения 0 или 1, т. е. это биты.
2. Вместо арифметических[4] операций используются логические (операция **and** (**и**) обозначается через $\wedge$, **or** (**или**) — через $\vee$, **exclusive or** (**исключающее или**) — через $\oplus$, **not** (**не**) — через $\neg$).

Для битовой модели арифметические операции над целыми числами i и j занимают по меньшей мере $l(i) + l(j)$ шагов, что соответствует логарифмической функции стоимости операндов. В самом деле, умножение и деление i на j с помощью наилучших известных алгоритмов требуют более чем $l(i) + l(j)$ шагов.

При битовых вычислениях порядок величин обозначается через O_B. Битовая модель полезна, когда речь идет об основных операциях, таких как арифметические, которые являются элементарными в других моделях. Например, для модели линейных программ умножение двух n-разрядных двоичных целых чисел можно осуществить за $O_A(1)$ шагов, тогда как для битовой модели наилучший известный результат — $O_B(n \log n \log \log n)$ шагов.

Другое применение битовой модели — логические схемы. Линейные программы с двоичными входным данными и операциями взаимно-однозначно соответствуют логическим схемам для вычисления набора булевых функций. Количество шагов в такой программе равно числу логических элементов в схеме.

[4]Таким образом, набор команд для наших RAM-компьютеров должен состоять из этих операций.

Пример 1.5. На рис. 1.17, *а*, показана программа для сложения двух 2-битовых чисел $[a_1a_0]$ и $[b_1b_0]$. Выходными являются переменные c_2, c_1 и c_0, такие, что $[a_1a_0] + [b_1b_0] = [c_2c_1c_0]$. Линейная программа, приведенная на рис. 1.17, *а*, выполняет следующие вычисления:

$$c_0 = a_0 \oplus b_0,$$
$$c_1 = ((a_0 \wedge b_0) \oplus a_1) \oplus b_1,$$
$$c_2 = ((a_0 \wedge b_0) \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1).$$

Соответствующая логическая схема показана на рис. 1.17, *б*. В качестве упражнения читатели могут доказать, что сложение двух 2-битовых чисел можно выполнить за $O_B(n)$ шагов. □

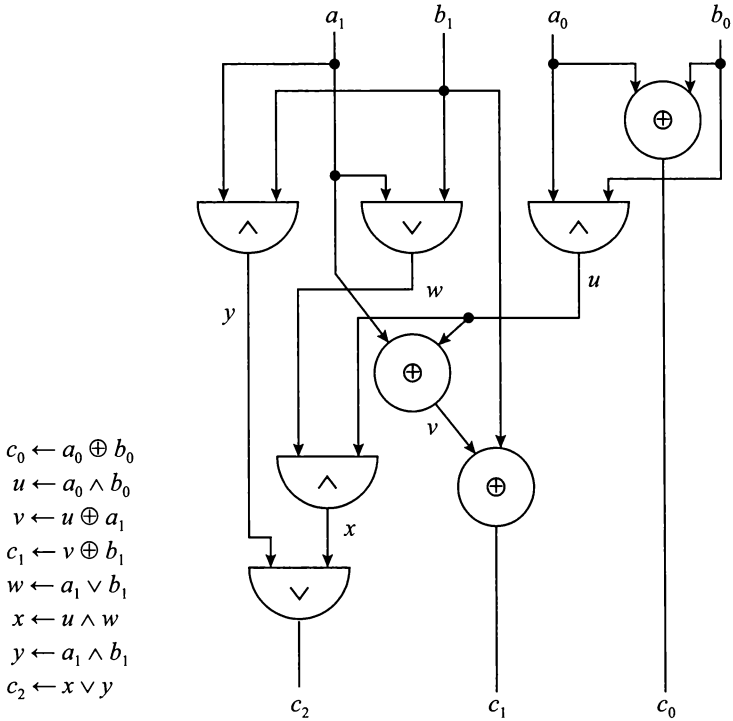


Рис. 1.17. Программа для сложения битов (*а*) и эквивалентная логическая схема (*б*)

III. Операции с двоичными векторами

Можно было бы не ограничивать значения переменных символами 0 и 1, а разрешить переменным принимать в качестве значения любой вектор битов. Фактически векторы битов фиксированной длины очевидным образом соответствуют целым числам, так что здесь не мы не выходим за пределы модели RAM, т.е., когда это удобно, просто разрешаются регистры неограниченного размера.

Однако, как мы увидим, в тех немногих алгоритмах, где применяется модель с векторами, битов длина векторов будет значительно больше числа битов, требуемых для представления размера задачи. Величина большинства целых чисел, фигурирующих в таком алгоритме, будет того же порядка, что и размер задачи. Например, решая задачу о выборе пути в графе со 100 узлами, можно было бы представить факт наличия или отсутствия пути из данного узла v в каждый из узлов с помощью вектора битов длины 100, а именно: в i-й позиции вектора для узла v стоит единица тогда и только тогда, когда существует путь из v в v_i. В этой же задаче можно также использовать целые числа для счета и индексации, например, и их размер, вероятно, совпадал бы с размером числа 100. Таким образом, для целых чисел требовалось бы 7 битов, тогда как для векторов — 100 битов.

Хотя это сравнение говорит не в пользу вычислений с векторами битов, большинство компьютеров выполняют логические операции с векторами битов, составляющими полное машинное слово, в рамках одной команды. Поэтому векторы битов длины 100 можно было бы обрабатывать за три или четыре шага (вместо одного для чисел). Тем не менее на результаты о временной и пространственной сложности алгоритмов в рамках модели с векторами битов мы должны смотреть критически, поскольку размер задачи, при котором модель становится нереалистичной, в этом случае намного меньше, чем в случае моделей RAM и линейных программ. Порядок величин в рамках модели с векторами битов мы будем обозначать через O_{BV}.

IV. Деревья решений

Мы рассмотрели три модификации модели RAM, игнорирующие команды разветвления и учитывающие только арифметические вычисления. В некоторых задачах в качестве основной меры сложности удобно выбирать количество выполняемых команд разветвления. В случае сортировки, например, выходные данные идентичны входным, за исключением порядка.

Обычно программу, состоящую из разветвлений, представляют в виде бинарного дерева, называемого *деревом решений*. Каждый внутренний узел представляет шаг решения. Проверка, представленная корнем, выполняется первой, а затем управление передается одному из его дочерних узлов в зависимости от результата проверки. В общем случае управление переходит от узла к одному из его дочер-

них узлов, причем выбор в каждом случае зависит от исхода проверки в этом узле до тех пор, пока не будет достигнут лист. Нужный результат находится на достигнутом листе.

Пример 1.6. На рис. 1.18 изображено дерево решений для программы, сортирующей три числа: a, b и c. Проверки представлены в виде овалов, содержащих операции сравнения в узлах; управление переходит влево, если ответ — "да", и вправо, если "нет". □

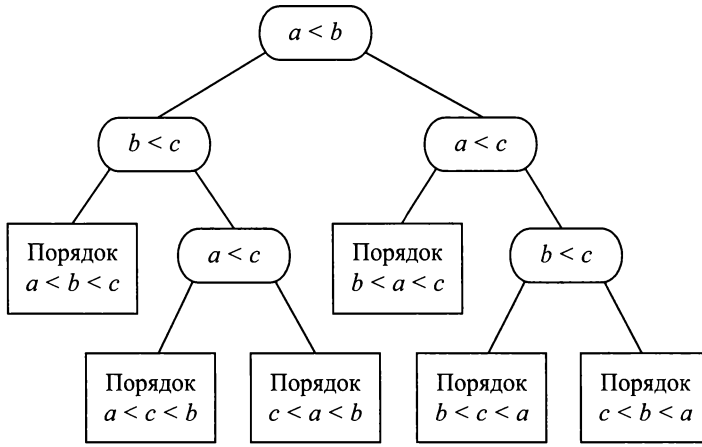


Рис. 1.18. Дерево решений

Временная сложность дерева решений равна высоте этого дерева как функции размера задачи. Обычно мы стремимся измерить наибольшее число сравнений, которые приходится делать, чтобы найти нужный путь от корня к листу. Порядок величин при использовании модели деревьев решений (сравнений) мы будем обозначать через O_C. Заметим, что общее число узлов в дереве может значительно превосходить его высоту. Например, дерево решений для сортировки n чисел должно содержать по крайней мере $n!$ листьев, хотя его высота может быть равной $n \log n$.

1.6. Простейшая модель вычислений: машина Тьюринга

Для доказательства того, что для вычисления конкретной функции требуется определенный минимальный объем времени, необходима модель, такая же универсальная, как рассмотренные ранее, но более простая. Система команд должна

быть максимально ограниченной, хотя эта модель должна быть в состоянии не только вычислить все то, что может вычислить RAM-компьютер, но и сделать это почти так же быстро. Под словом "почти" будем подразумевать полиномиальную связанность.

Определение. Говорят, что неотрицательные функции $f_1(n)$ и $f_2(n)$ *полиномиально связаны*, если найдутся такие полиномы $p_1(x)$ и $p_2(x)$, что для всех n справедливы неравенства $f_1(n) \le p_1(f_2(n))$ и $f_2(n) \le p_2(f_1(n))$.

Пример 1.7. Две функции $f_1(n) = 2n2$ и $f_2(n) = n5$ полиномиально связаны: можно положить $p_1(x) = 2x$, так как $2n2 \le 2n5$, и $p_2(x) = x3$, так как $n5 \le (2n2)3$. Однако функции $n2$ и 2^n не являются полиномиально связанными, так как нет такого полинома $p(x)$, что $p(n2) \ge 2^n$ для всех n. □

В настоящее время единственным классом функций, для которых мы можем применить такие общие вычислительные модели, как машины Тьюринга, чтобы получить нижние оценки вычислительной сложности, являются быстро растущие функции. Например, в главе 11 будет показано, что некоторые задачи требуют экспоненциального времени и объема памяти. (Функция $f(n)$ называется *экспоненциальной*, если существуют такие константы $c_1 > 0$, $k_1 > 1$, $c_2 > 0$ и $k_2 > 1$, что выполняется неравенство $c_1 k_1^n \le f(n) \le c_2 k_2^n$ для всех, кроме конечного количества, значений n.) Относительно полиномиальной связанности все экспоненциальные функции, по существу, одинаковы, поскольку любая функция, полиномиально связанная с экспоненциальной, сама является экспоненциальной.

Таким образом, у нас есть стимул использовать простую модель, для которой временная сложность задач полиномиально связана с их сложностью на RAM-компьютерах. В частности, модель, которую мы будем применять, а именно многоленточную машину Тьюринга, может потребовать $(f[n]4)$ шагов,[5] чтобы сделать то, что RAM-компьютер при логарифмической функции стоимости делает за $f(n)$ шагов, но не больше. Итак, временные сложности программ на RAM-компьютерах и машинах Тьюринга, как мы увидим, полиномиально связаны.

Определение. *Многоленточная машина Тьюринга* (МТ) изображена на рис. 1.19. Она состоит из k *лент*, бесконечных справа. Каждая лента разбита на *ячейки*, каждая из которых содержит один из конечного числа *символов на ленте*. Одна ячейка на каждой ленте обозревается *головкой этой ленты*; головка может считывать с ленты и записывать на нее. Работа машины Тьюринга определяется

[5]На самом деле верна более низкая оценка $O([f(n)\log f(n)\ \log\log f(n)]2)$, но мы интересуемся оценками с точностью до полиномов и нас вполне устроит результат с четвертой степенью (см. раздел 7.5).

простой программой, называемой *конечным управляющим устройством*. Оно всегда находится в одном из конечного числа *состояний*, которое можно интерпретировать как номер текущей команды в программе.

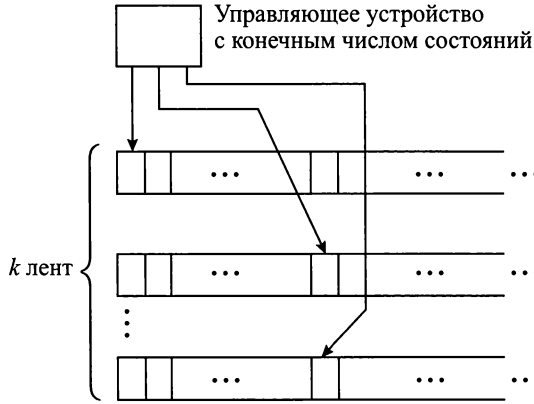


Рис. 1.19. Многоленточная машина Тьюринга

Один шаг вычисления на машине Тьюринга состоит в следующем. В соответствии с текущим состоянием управляющего устройства и символами на лентах, которые *обозреваются* (т.е. находятся под) каждой из головок, машина Тьюринга может выполнить некоторые или все из следующих операций.

1. Изменить состояние управляющего устройства.
2. Напечатать новые символы на лентах вместо старых в некоторых или во всех ячейках под головками.
3. Передвинуть некоторые или все головки независимо друг от друга на одну ячейку влево (L) или вправо (R) либо оставить на месте (S).

Формально k-ленточная машина Тьюринга задается семеркой

$$(Q, T, I, \delta, \mathrm{b}, q_0, q_\mathrm{f}),$$

где

1. Q — множество *состояний*;
2. T — множество *символов на лентах*;
3. I — множество *входных символов*, $I \subseteq T$;
4. b — *пустой символ* , $\mathrm{b} \in T - I$;
5. q_0 — *начальное состояние*;

6. q_f — *заключительное* (или *допускающее*) *состояние*;
7. δ — *функция переходов*, отображающая некоторое подмножество множества $Q \times T^k$ в $Q \times (T \times \{L, R, S\})^k$, т.е. по произвольному $(k+1)$-кортежу, содержащему состояние и k символов на лентах, она выдает новое состояние и k пар, каждая из которых состоит из нового символа на ленте и направления сдвига головки. Пусть $\delta(q, a_1, a_2, ..., a_k) = (q', (a_1', d_1), (a_2', d_2), ..., (a_k', d_k))$ и машина Тьюринга находится в состоянии q, а ее головка на i-й ленте обозревает символ a_i, где $1 \le i \le k$. Тогда за один шаг эта машина Тьюринга переходит в состояние q', заменяет a_i на a_i' и сдвигает головку на i-й ленте в направлении d_i, где $1 \le i \le k$.

Можно создать машину Тьюринга для распознавания языка. Символы на лентах такой машины включают алфавит рассматриваемого языка (его буквы играют роль *входных символов*), пустой символ, обозначаемый b, и, возможно, другие символы. Вначале на первой ленте записано слово из входных символов по одному на ячейку, начиная с самой левой. Все ячейки справа от ячеек, содержащих входное слово, пустые. Все остальные ленты совершенно пустые. Строка из входных символов *допускается* тогда и только тогда, когда машина Тьюринга, начав работу в выделенном начальном состоянии (головки при этом находились на левых концах своих лент), сделает последовательность шагов, которые в конце концов приведут ее в допускающее состояние. *Языком*, *допускаемым* данной машиной Тьюринга, называется множество всех слов из входных символов, допускаемых в указанном смысле.

Пример 1.8. Двухленточная машина Тьюринга на рис. 1.20 распознает палиндромы[6] в алфавите {0, 1} следующим образом.

1. Первая ячейка на ленте 2 отмечается специальным символом X, и вход копируется с ленты 1, где он записан вначале (см. рис. 1.20, *а*), на ленту 2 (см. рис. 1.20, *б*).
2. Головка на ленте 2 сдвигается на символ X (см. рис. 1.20, *в*).
3. Повторяется следующая процедура: головка на ленте 2 сдвигается на одну позицию вправо, а на ленте 1 — на одну позицию влево, и соответствующие символы сравниваются. Если все символы совпадают, то входная строка является палиндромом и машина Тьюринга доходит до допускающего состояния q_5. В противном случае в некоторый момент очередной шаг машины Тьюринга будет не определен, и она остановится, не допустив входного слова.

---

[6]Палиндромом называется слово, которое одинаково читается как в прямом, так и обратном направлениях, например, 0100010.

Функция переходов соответствующей машины Тьюринга приведена на рис. 1.21. □

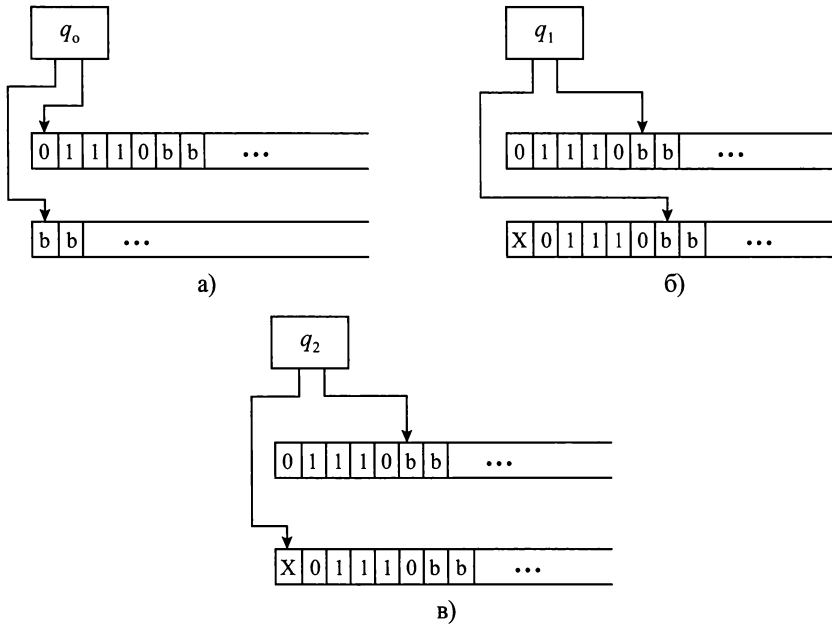


Рис. 1.20. Машина Тьюринга, обрабатывающая строку 01110

Работу машины Тьюринга формально можно описать с помощью мгновенных описаний. *Мгновенным описанием* (МО) k-ленточной машины Тьюринга M называется k-кортеж $(\alpha_1, \alpha_2, ..., \alpha_k)$, где α_i для каждого i представляет собой строку вида xqy, причем xy — это строка на i-й ленте машины M (пустые символы, стоящие справа от его правого конца, опускаются), а q — текущее состояние машины. Головка на i-й ленте обозревает символ, стоящий справа от q.

Если мгновенное описание D_1 переходит в мгновенное описание D_2 за один шаг машины Тьюринга M, то пишут $D_1 \vdash_M D_2$ (знак $\vdash$ читается "переходит в"). Если $D_1 \vdash_M D_2 \vdash_M ... \vdash_M D_n$, то для некоторого $n \geq 2$ и пишут $D_1 \vdash_M^+ D_n$. Если либо $D = D'$, либо $D \vdash_M^+ D'$, то $D \vdash_M^* D'$.

Данная k-ленточная машина Тьюринга $M(Q, T, I, \delta, b, q_0, q_f)$ *допускает* строку $a_1 a_2 ... a_n$, где a_i — элементы из I, если $(q_0 a_1 a_2 ... a_n, q_0, q_0, ..., q_0) \vdash_M^* (\alpha_1, \alpha_2, ..., \alpha_k)$ для некоторых α_i, содержащих q_f.

Текущее состояние	Символ		Новый символ, перемещение головки		Новое состояние	Комментарии
	Лента 1	Лента 2	Лента 1	Лента 2		
q_0	0	b	0, S	X, R	q_1	Если вход не пустой, то на ленте 2
	1	b	1, S	X, R	q_2	печатается символ X и головка сдви-
	b	b	b, S	b, S	q_3	гается вправо; машина переходит в состояние q_1. В противном случае машина переходит в состояние q_5
q_1	0	b	0, R	0, R	q_1	Машина остается в состоянии q_1
	1	b	1, R	1, R	q_1	и переписывает ленту 1 на ленту 2,
	b	b	b, S	b, L	q_2	пока на ленте 1 не встретится символ b. Затем она переходит в состояние q_2
q_2	b	0	b, S	0, L	q_2	Головка на ленте 1 остается на ме-
	b	1	b, S	1, L	q_2	сте, а на ленте 2 перемещается влево
	b	X	b, L	X, R	q_3	до символа X. Затем машина переходит в состояние q_3.
q_3	0	0	0, S	0, R	q_4	Состояния q_3 и q_4 чередуются.
	1	1	1, S	1, R	q_4	В состоянии q_3 машина сравнивает символы на обеих лентах, головка на ленте 2 сдвигается вправо и машина переходит в состояние q_4.
q_4	0	0	0, L	0, S	q_3	Из состояния q_4 машина переходит
	0	1	0, L	1, S	q_3	в состояние q_5 и допускает входное
	1	0	1, L	0, S	q_3	слово, если головка достигла симво-
	1	1	1, L	1, S	q_3	ла b на ленте 2. В противном случае
	0	b	0, S	b, S	q_3	головка на ленте 1 сдвигается влево
	1	b	1, S	b, S	q_3	и машина возвращается в состояние q_3. Чередование состояний q_3 и q_4 не позволяет головке на ленте 1 уходить за ее левый конец.
q_5						Входное слово допускается.

Рис. 1.21. Функция переходов Тьюринга для распознавания палиндромов

Пример 1.9. На рис. 1.22 приведена последовательность мгновенных описаний машины Тьюринга, изображенной на рис. 1.21, которой подано на вход слово 010. Поскольку q_5 — заключительное состояние, эта машина Тьюринга допускает строку 010. □

$$
\begin{aligned}
(q_0010, q_0) &\vdash (q_1010, \mathrm{X}q_1) \\
&\vdash (0q_110, \mathrm{X}0q_1) \\
&\vdash (01q_10, \mathrm{X}01q_1) \\
&\vdash (010q_1, \mathrm{X}010q_1) \\
&\vdash (010q_2, \mathrm{X}01q_20) \\
&\vdash (010q_2, \mathrm{X}0q_210) \\
&\vdash (010q_2, \mathrm{X}q_2010) \\
&\vdash (010q_2, q_2\mathrm{X}010) \\
&\vdash (01q_30, \mathrm{X}q_3010) \\
&\vdash (01q_40, \mathrm{X}0q_410) \\
&\vdash (0q_310, \mathrm{X}0q_310) \\
&\vdash (0q_410, \mathrm{X}01q_40) \\
&\vdash (q_3010, \mathrm{X}01q_30) \\
&\vdash (q_4010, \mathrm{X}010q_4) \\
&\vdash (q_5010, \mathrm{X}010q_5)
\end{aligned}
$$

Рис. 1.22. Последовательность мгновенных описаний машины Тьюринга

Помимо естественной интерпретации машины Тьюринга как устройства, распознающей какой-то язык, ее можно рассматривать как устройство, которое вычисляет некоторую функцию f. Аргументы этой функции кодируются на входной ленте в виде строки x со специальным маркером #, отделяющим их друг от друга. Если машина Тьюринга останавливается, когда на ее выходной ленте записано целое число y (значение функции), то полагают $f(x) = y$. Таким образом, процесс вычисления мало отличается от процесса распознавания языка.

Временная сложность $T(n)$ машины Тьюринга M равна наибольшему числу шагов, сделанных ею при обработке входа длины n (для всех входных данных длины n). Если на каком-нибудь входе длины n машина Тьюринга не останавливается, то для этого n значение $T(n)$ не определено. *Пространственная сложность* $S(n)$ машины Тьюринга равна наибольшему расстоянию от левого конца ленты, которое должна пройти головка при обработке входа длины n. Если головка на ка-

кой-то ленте неопределенно долго движется вправо, то функция $S(n)$ не определена. Порядок величин в отношении машины Тьюринга мы будем обозначать через O_{TM}

Пример 1.10. Временная сложность машины Тьюринга, изображенной на рис. 1.21, равна $T(n) = 4n + 3$, а ее пространственная сложность равна $S(n) = n + 2$. Это можно проверить, исследовав случай, когда вход на самом деле является палиндромом. □

1.7. Связь между машинами Тьюринга и компьютерами RAM

Основное применение машины Тьюринга (МТ) находят в установлении нижних оценок объемов времени и памяти, необходимых для решения конкретной задачи. В большинстве случаев мы можем установить нижние оценки только с точностью до полиномиальной связанности. Для более точных оценок потребуется рассматривать более специфические детали конкретных моделей. К счастью, вычисления на компьютерах RAM и RASP полиномиально связаны с вычислениями на машинах Тьюринга.

Рассмотрим связь между машинами RAM и МТ. Очевидно, что машина RAM может моделировать работу k-ленточной машины МТ, помещая по одной ячейке МТ в регистр. В частности, i-ю ячейку j-й ленты можно хранить в регистре $ki + j + c$, где c — константа, которая должна обеспечить машине RAM некоторое оперативное пространство. В него входят k регистров для запоминания положений k головок МТ. Машина RAM может считывать клетки машины МТ, используя косвенную адресацию с помощью регистров, содержащих положения головок на лентах.

Предположим, что заданная машина МТ имеет временную сложность $T(n) \geq n$. Тогда машина RAM может прочитать ее вход, запомнить его в регистрах, представляющих первую ленту, и смоделировать эту МТ за время $O(T(n))$ при равномерной функции стоимости или за время $O(T(n)\log T(n))$ при логарифмической функции стоимости. В любом случае время работы машины RAM ограничено сверху полиномом от времени работы машины Тьюринга, поскольку любая функция типа $O(T(n)\log n)$, естественно, имеет порядок $O(T2(n))$.

Обратное утверждение верно только при логарифмической функции стоимости для машины RAM. При равномерной функции стоимости n-шаговая программа для машины RAM без входа может вычислять числа порядка 22^n, что требует порядка 2^n ячеек машины Тьюринга только для запоминания и считывания. Следовательно, при равномерной функции стоимости, очевидно, полиномиальной связи между машинами RAM и МТ не существует (упражнение 1.19).

Хотя при анализе алгоритмов мы предпочитаем использовать равномерную функцию стоимости в силу ее простоты, пытаясь установить нижние границы

временной сложности, мы вынуждены отказаться от нее. Машина RAM с равномерной функцией стоимости вполне разумна, если рост чисел по порядку не превосходит размер задачи. Но, как мы уже говорили, при использовании машины RAM в качестве модели размер чисел "заметается под ковер", и вряд ли можно получить полезные нижние оценки. В то же время для логарифмической функции стоимости верна следующая теорема.

Теорема 1.3. Пусть L — язык, допускаемый некоторой машиной RAM за время $T(n)$ при логарифмической функции стоимости. Если в RAM-программе нет умножений и делений, то на многоленточных машинах Тьюринга распознавание языка L имеет временную сложность не более $O(T2(n))$.

Доказательство. Представим все не содержащие нуля регистры рассматриваемой машины RAM, как показано на рис. 1.23. На ленте помещена последовательность пар чисел (i_j, c_j), записанных в двоичной форме без нулей в начале строки и разделенных маркерами. Для каждого j число c_j представляет собой содержимое регистра i_j машины RAM. Содержимое сумматора RAM хранится в двоичном виде на второй ленте, а третья лента играет роль рабочей памяти. Еще две ленты служат для записи входа и выхода RAM. Каждый шаг программы RAM представлен конечным числом состояний МТ. Мы не будем моделировать все RAM-команды, а рассмотрим только команды `ADD` *20 и `STORE` 30, которые иллюстрируют общую идею. Для команды `ADD` *20 можно создать машину Тьюринга, которая будет работать следующим образом.

1. Ищем на ленте 1 место, соответствующее регистру 20 в машине RAM, т.е. последовательность ##10100#. Если оно найдено, то на ленту 3 помещается следующее за ним число, которое будет содержимым регистра 20. Если такое место не найдено, то машина останавливается. Содержимое регистра 20 равно нулю, поэтому косвенную адресацию произвести нельзя.
2. Ищем на ленте 1 место, где хранится регистр RAM, номер которого записан на ленте 3. Если оно найдено, то содержимое этого регистра записывается на ленту 3. Если нет, туда помещается нуль.
3. Число, записанное на ленту 3 на шаге 2, прибавляется к содержимому сумматора, которое хранится на ленте 2.

#	#	i_1	#	c_1	#	#	i_2	#	c_2	#	#	• • •	i_k	#	c_k	#	#	b	• • •

Рис. 1.23. Представление машины RAM с помощью машины Тьюринга

Для моделирования команды STORE 30 можно так построить МТ, чтобы она работала следующим образом.

1. Ищем место расположения регистра 30 в RAM, т.е. ##11110#.
2. Если оно найдено, то на ленту 3 записывается все, что расположено справа от ##11110#, кроме числа, стоящего непосредственно справа (т.е. старого содержимого регистра 30). Затем содержимое сумматора (на ленте 2) записывается непосредственно справа от ##11110#, а за ним помещается строка, скопированная на ленту 3.
3. Если на ленте 1 не нашлось места, соответствующего регистру 30, то, дойдя до самого левого пустого символа, машина печатает 11110#, затем содержимое сумматора и, наконец, ##.

Немного поразмыслив, нетрудно понять, что можно описать машину Тьюринга, которая правильно смоделирует работу машины RAM. Мы должны показать, что вычисление на машине RAM с логарифмической функцией стоимости k потребует не более чем $O(k2)$ шагов на машине Тьюринга. Сначала заметим, что регистр может появиться на ленте 1, только если его текущее значение когда-то раньше было помещено в этот регистр. Стоимость записи c_j в регистр i_j равен $l(c_j) + l(i_j)$, что с точностью до константы равно длине нашего представления $\#\#i_j\#c_j\#\#$. Отсюда заключаем, что длина непустой части ленты 1 имеет порядок $O(k)$. □

Моделирование любой команды, отличной от STORE, имеет порядок длины ленты 1, т.е. $O(k)$, поскольку основное время уходит на поиск на ленте. Аналогично время моделирования команды STORE не больше суммы объемов времени на ленте 1 и ее копирования, т.е. $O(k)$. Следовательно, одну команду RAM (кроме умножения и деления) можно смоделировать за не более чем за $O(k)$ шагов машины МТ. Поскольку одна команда RAM занимает при логарифмической функции стоимости по крайней мере одну единицу времени, общее время, затрачиваемое соответствующей МТ, имеет порядок $O(k2)$, что и требовалось доказать.

Если в RAM-программе используются команды умножения и деления, то можно написать подпрограммы для машины Тьюринга, выполняющие эти операции с помощью сложений и вычитаний. Мы предоставляем читателю доказать, что логарифмические функции стоимости этих подпрограмм не больше квадрата логарифмических функций стоимости соответствующих команд. Таким образом, нетрудно доказать следующую теорему.

Теорема 1.4. Машины RAM и RASP с логарифмической функцией стоимости и многоленточные машины Тьюринга полиномиально связаны.

Доказательство. Примените теоремы 1.1–1.3 и проанализируйте подпрограммы для умножения и деления. □

Аналогичный результат справедлив и для пространственной сложности, хотя он представляется менее интересным.

1.8. Псевдокод высокого уровня, написанный на Алголе

Хотя наши основные меры сложности определяются в терминах сложности операций для RAM, RASP и машин Тьюринга, мы, вообще говоря, не хотели бы описывать алгоритмы в терминах настолько простых машин. Впрочем, в этом нет необходимости. Для наглядного описания алгоритмов мы воспользуемся псевдокодом высокого уровня, написанным на Алголе.

Псевдокод на Алголе можно перевести непосредственно в программу для машин RAM или RASP. Фактически эту задачу должен был бы выполнить специальный компилятор. Нас, однако, не будут интересовать детали трансляции псевдокода на Алголе в конкретные программы для машин RAM или RASP. Нас интересуют только объемы времени и памяти, необходимые для выполнения команд, соответствующих операторам псевдокода.

Псевдокод, написанный на Алголе, отличается от кода традиционных языков программирования тем, что в нем разрешается использовать любой тип математических инструкций, если только их значения понятны, а перевод в команды RAM или RASP очевиден. Кроме того, этот язык не имеет фиксированного набора типов данных. Переменные могут представлять целые числа, строки и массивы. Дополнительные типы данных — множества, графы, списки, очереди и т.п. — можно вводить по мере необходимости. Формальные описания типов данных по возможности избегаются. Тип данных какой-то переменной и ее область видимости[7] должны быть ясны либо по ее названию, либо по контексту.

В псевдокоде на Алголе применяются общепринятые программные и математические конструкции, такие как выражения, условия, операторы и процедуры. Ниже приведено неформальное описание некоторых из таких конструкций. Никаких попыток дать точное определение мы предпринимать не будем, поскольку это выходит далеко за рамки тематики данной книги. Заметим, что нетрудно написать программы, смысл которых зависит от деталей, не рассмотренных здесь, но лучше избегать этого.

---

[7] *Область видимости переменной* — это окружение, в котором она имеет смысл. Например, областью видимости индекса суммирования является выражение, стоящее под знаком суммы, а вне его он не имеет смысла.

Псевдокод на Алголе представляется операторами одного из следующих типов.

1. Переменная ← выражение
2. **if** условие **then** оператор **else** оператор[8]
3.
   а) **while** условие **do** оператор
   б) **repeat** оператор **until** условие
4. **for** переменная ← исходное значение **step** размер шага[9] **until** конечное значение **do** оператор
5. Метка: оператор
6. **goto** метка
7. **begin**
   оператор;
   оператор;
   ...
   оператор;
   оператор
   end
8.
   а) **procedure** имя (список параметров): оператор
   б) **return** выражение
   в) имя процедуры (аргументы)
9.
   а) **read** переменная
   б) **write** выражение
10. **comment** комментарий
11. Любой другой произвольный оператор

[8]Часть "**else** оператор" не обязательна. Однако такой вариант обычно приводит к неоднозначности типа "висячий оператор **else**". Мы выбираем традиционный путь и предполагаем, что **else** связывается с ближайшим еще не связанным оператором **then**.

[9]Если размер шага равен 1, то часть "**step** размер шага" не обязательна.

Рассмотрим каждый из этих операторов.

1. Оператор присваивания

переменная ← выражение

означает, что необходимо вычислить выражение справа от стрелки и его значение присвоить переменной, стоящей слева от стрелки. Временная сложность оператора присваивания определяется временем, затрачиваемым на вычисление значения выражения и присваивание этого значения переменной.

Если значение выражения не принадлежит к данным элементарного типа, таким как целые числа, то в некоторых случаях можно уменьшить время с помощью указателей. Например, присваивание $A \leftarrow B$, где A и B — матрицы размера $n \times n$, обычно потребовало бы $O(n2)$ шагов. Однако если матрица B больше нигде не встречается, то путем простого переименования массива можно сделать это время фиксированным, не зависящим от n.

2. В операторе **if**

if условие **then** оператор **else** оператор

условием, следующим за **if**, может быть любое выражение, принимающее значение **true** или **false**. Если это условие имеет значение **true**, то необходимо выполнить оператор, стоящий за **then**. В противном случае необходимо выполнить оператор, стоящий за **else** (если **else** есть). Стоимость выполнения оператора **if** равен сумме стоимостей вычисления значения и его проверки, а также стоимости выполнения оператора, стоящего сразу за **then**, или оператора, стоящего за **else**, в зависимости от того, какой из них выполнялся на самом деле.

3. Назначение оператора **while**

while условие **do** оператор

и оператора **repeat**

repeat оператор **until** условие

состоит в организации цикла. В операторе **while** вычисляется значение условия, идущего после **while**. Если оно истинно (принимает значение **true**), то выполняется оператор, стоящий после **do**. Этот процесс повторяется до тех пор, пока условие не станет ложным. Если вначале это условие было истинным, то, чтобы выполнение оператора **while** закончилось, в ходе выполнения оператора это условие должно принять значение **false**. Для вычисления стоимости выполнения оператора **while** стоимости всех проверок условия и всех выполненных операторов суммируются.

Оператор **repeat** интерпретируется аналогично, но только теперь оператор, стоящий за **repeat**, выполняется перед проверкой условия.

4. В операторе **for**

for переменная ← исходное значение **step** размер шага
until заключительное значение **do** оператор

"исходное значение", "размер шага" и "заключительное значение" являются выражениями. Если размер шага положителен, то переменная, называемая *индексом*, полагается равной значению выражения для исходного значения. Если оно больше заключительного значения, то выполнение оператора заканчивается. В противном случае выполняется оператор, стоящий после **do**, значение переменной увеличивается на величину шага и сравнивается с заключительным значением. Процесс повторяется до тех пор, пока значение переменной не станет больше заключительного значения. Вариант, в котором размер шага отрицателен, интерпретируется аналогично, за исключением того, что выполнение завершается, когда значение переменной становится меньше заключительного значения. Стоимость выполнения оператора **for** должна быть очевидной из анализа оператора **while**.

В приведенном описании совершенно игнорируется такая деталь, как момент вычисления выражений для исходного значения, размера шага и заключительного значения. Выполнение оператора, стоящего после ключевого слова **do**, иногда может изменять размер шага. В таком случае вычисление размера шага каждый раз, когда переменная возрастает, может привести к результату, отличному от того, который получился бы, если бы размер шага вычислялся раз и навсегда. Точно так же вычисление размера шага может влиять на заключительное значение, а изменение размера шага — на проверку условия завершения. Мы избегаем этих трудностей, отказываясь от программ, в которых подобные явления сделали бы их смысл неясным.

5. Любой оператор можно переделать в помеченный оператор, написав перед ним метку, за которой стоит двоеточие. Главное назначение метки — создание цели для оператора **goto**. Меткам не приписывается никакого веса.

6. Оператор **goto**

goto метка

указывает, что дальше выполняется оператор с данной меткой. Этот оператор не может стоять внутри блока типа 7, если сам оператор **goto** не находится в том же блоке. Стоимость выполнения оператора **goto** равна 1. Операторы **goto** следует применять редко, так как, вообще говоря, они затрудняют понимание программы. Основное применение оператора **goto** — это выход из операторов **while**.

7. Последовательность операторов, отделенных друг от друга точками с запятыми и заключенных между ключевыми словами **begin** и **end**, образует оператор, который называется *блоком*. Поскольку блок является оператором, его можно

применять всюду, где предусмотрено применение оператора. Обычно программа считается блоком. Стоимость выполнения блока равна сумме стоимостей выполнения операторов, составляющих блок.

8. *Процедуры*. В псевдокоде на Алголе можно определять и впоследствии вызывать процедуры. Процедуры определяются *оператором определения процедур*

procedure имя (список параметров): оператор

Список параметров — это последовательность фиктивных переменных, называемых *формальными параметрами*. Например, следующий оператор определяет процедуру-функцию MIN:

procedure MIN (x, y):
if $x > y$ **then return** y **else return** x

Аргументы x и y являются формальными параметрами.

Процедуры используются одним из двух способов. Один способ — в качестве *функции*. После того как процедура-функция описана, к ней можно обратиться в некотором выражении, указав ее имя с требуемыми аргументами. В этом случае последним оператором, выполняемым в данной процедуре, должен быть оператор **return** (п. 8, *б*). Этот оператор приводит к вычислению выражения, следующего за ключевым словом **return**, и окончанию выполнения процедуры. Значением функции будет значение этого выражения. Например, оператор

$$A \leftarrow \texttt{MIN}(2 + 3, 7)$$

приводит к тому, что переменная A получает значение 5. Выражения 2 + 3 и 7 называются *фактическими параметрами* этого обращения к данной процедуре.

Второй способ применения процедуры состоит в вызове ее с помощью оператора вызова процедуры (п. 8, *в*). По сути, этот оператор — имя процедуры, за которым идет список фактических параметров. Оператор вызова процедуры может изменить и (обычно изменяет) данные (значения переменных, массивов и т.д.) вызываемой программы. В определении процедуры, вызываемой таким способом, оператор **return** не нужен. Завершение выполнения последнего оператора процедуры завершает и выполнение оператора ее вызова. Например, следующий оператор определяет процедуру INTERCHANGE:

```
procedure INTERCHANGE(x, y):
begin
        t ← x;
        x ← y;
        y ← t
end
```

Для обращения к этой процедуре можно было бы написать оператор вызова процедуры, такой, как

INTERCHANGE($A[i], A[j]$)

Обмен информацией между процедурами можно осуществлять двумя способами. Во-первых, с помощью глобальных переменных. Мы предполагаем, что глобальные переменные неявно описаны в некоторой универсальной области, а в этой области есть подобласть, в которой определены процедуры.

Во-вторых, связь между процедурами можно осуществлять с помощью параметров. В Алголе 60 используется *вызов по значению* и *вызов по имени*. При вызове по значению формальные параметры процедуры трактуются как локальные переменные, которым в качестве начальных значений придаются значения фактических параметров. При вызове по имени формальные параметры служат указателями места в программе, куда подставляются фактические параметры вместо каждого вхождения соответствующих формальных параметров. Для простоты мы отходим от Алгола 60 и используем *вызов по ссылке*. При вызове по ссылке параметры обрабатываются с помощью указателей на фактические параметры. Если фактический параметр является выражением (возможно, постоянной), то соответствующий формальный параметр трактуется как локальная переменная, которой в качестве начального значения присвоено значение этого выражения. Таким образом, стоимость вычисления функции или вызова процедуры на машинах RAM и RASP равна сумме стоимостей выполнения операторов в определении соответствующей процедуры. Стоимость выполнения процедуры, вызывающей другие процедуры (возможно, себя), обсуждается в главе 2.

9. Смысл операторов **`read`** и **`write`** очевиден. Стоимость выполнения оператора **`read`** равен 1. Стоимость выполнения оператора **`write`** на единицу больше стоимости вычисления значения выражения, стоящего за ключевым словом **`write`**.

10. Оператор **`comment`** позволяет вставлять замечания, облегчающие понимание программы, и имеет нулевой вес.

11. В дополнение к операторам традиционных языков программирования мы включили набор вспомогательных операторов, благодаря которым алгоритм можно понять легче, чем эквивалентную последовательность стандартных операторов языка программирования. Такие операторы используются в ситуациях, когда детали реализации либо несущественны, либо очевидны или когда желательно дать описание на языке еще более высокого уровня. Приведем примеры обычно используемых вспомогательных операторов:

а) Пусть a будет наименьшим элементом множества S

б) Пометить элемент a как "старый"[10]

в) **`without loss of generality`**[11] (**`wlg`**) считаем, что ... **`otherwise`** ... **`in`** оператор

Например, инструкция

`wlg` считаем, что $a \leq b$ **`otherwise`** поменять местами c и d **`in`** оператор

означает, что если $a \leq b$, то стоящий дальше оператор следует выполнять так, как он записан, а если $a > b$, то продублировать этот оператор, предварительно поменяв в нем местами c и d.

Реализация таких операторов в терминах обычных языков программирования или команд RAM не вызывает затруднений, но очень утомительна. Приписывание стоимости выполнения операторам такого типа зависит от контекста, в котором оказывается данный оператор. Дальнейшие примеры операторов такого рода не раз встретятся нам в этой книге в программах, представленных псевдокодом, который написан на Алголе.

Поскольку переменные обычно не будут описываться, условимся об областях их видимости. В одной программе или процедуре мы не будем использовать одинаковые имена для двух разных переменных. Таким образом, областью видимости переменной обычно можно считать всю процедуру или программу, в которую она входит.[12] Одно важное исключение возникает в случае, когда несколько процедур действуют на общей базе данных. В этом случае предполагается, что переменные общей базы данных являются глобальными, а переменные, используемые процедурой для временного запоминания в ходе работы с базой данных, — локальными по отношению к этой процедуре. Всякий раз, когда может возникнуть недоразумение по поводу области видимости переменной, мы будем давать явное описание.

Упражнения

1.1. Докажите, что $g(n)$ имеет порядок $O(f(n))$, если (a) $f(n) \geq \varepsilon$ для некоторого $\varepsilon > 0$ и для почти всех (т.е. для всех, кроме конечного числа) n и $(б)$ суще-

[10]Под этим мы подразумеваем, что имеется массив `STATUS`, такой, что `STATUS`[a] равно 1, если a — "старый" элемент, и 0, если a — "новый" элемент.

[11]Не теряя общего смысла

[12]Это утверждение имеет несколько несущественных исключений. Например, в процедуру могут входить два невложенных оператора **`for`**, оба с индексом i. Строго говоря, область видимости индекса оператора **`for`** — это сам оператор **`for`**, потому эти индексы i являются разными переменными.

ствуют такие постоянные $c_1 > 0$ и $c_2 > 0$, что $g(n) \le c_1 f(n) + c_2$ для почти всех $n \ge 0$.

1.2. Будем писать $f(n) \le g(n)$, если существует такая положительная постоянная c, что $f(n) \le cg(n)$ для всех n. Покажите, что $f_1 \le g_1$ и $f_2 \le g_2$ влечет $f_1 + f_2 \le g_1 + g_2$. Какие еще свойства сохраняет отношение $\le$?

1.3. Напишите программы для машин RAM, RASP, представленные псевдокодом на Алголе, для решения следующих задач:

а) Вычислите $n!$ по заданному n.

б) Прочитайте n положительных чисел, за которыми следует признак конца 0, а затем напечатайте их в порядке неубывания.

в) Допустите все входы вида $1^n 2^{n^2} 0$

1.4. Проанализируйте временную и пространственную сложности программ из упражнения 1.3 при (*а*) равномерной и (*б*) логарифмической функции стоимости. Введите меру "размера" входа.

***1.5.** Напишите RAM-программу для вычисления n^n с временной сложностью $O(\log n)$ при равномерной функции стоимости. Докажите, что ваша программа правильная.

***1.6.** Покажите, что для любой RAM-программы с временной сложностью $T(n)$ при логарифмической функции стоимости существует эквивалентная RAM-программа с временной сложностью $O(T2(n))$, в которой нет команд `MULT` и `DIV`. Указание: смоделируйте `MULT` и `DIV` подпрограммами, в которых регистры с четными номерами используются для промежуточной памяти. В случае команды `MULT` покажите, что если i необходимо умножить на j, то можно сосчитать каждое из $l(j)$ частичных произведений и сложить их за $O(l(j))$ шагов, а каждый шаг занимает время $O(l(i))$.

***1.7.** Что случится с вычислительной мощностью машин RAM или RASP, если из системы команд убрать команды `MULT` вместе с `DIV`? Как это отразится на стоимости вычислений?

****1.8.** Покажите, что любой язык, допускаемый машиной RAM, может допустить RAM без косвенной адресации. (*Подсказка*: покажите, что всю ленту машины Тьюринга можно целиком закодировать одним целым числом. Таким образом, машину Тьюринга можно смоделировать в конечном числе регистров RAM.)

1.9. Покажите, что при (*а*) равномерной и (*б*) логарифмической функции стоимости машины RAM и RASP эквивалентны в смысле равенства их пространственной сложности с точностью до постоянного множителя.

1.10. Найдите линейную программу, которая вычисляет определитель матрицы 3×3 по ее девяти элементам в качестве входных данных.

1.11. Напишите последовательность битовых операций для вычисления произведения двух двухразрядных двоичных целых чисел.

1.12. Покажите, что множество функций, вычислимых любой n-строчной линейной программой с двоичными входным данными и булевыми операциями, можно реализовать в виде комбинационной логической схемы из n булевых элементов.

1.13. Покажите, что любую булеву функцию можно вычислить с помощью линейной программы.

***1.14.** Пусть граф с n узлами представлен множеством двоичных векторов v, где j-я компонента вектора v_i равна единице тогда и только тогда, когда узлы i и j соединены ребром. Найдите алгоритм сложности $O_{\text{в}}V(n)$, который создает вектор p_i, у которого на j-м месте стоит единица тогда и только тогда, когда есть путь из узла i в узел j. Можно использовать поразрядные битовые логические операции на двоичных векторах, арифметические операции на целочисленных переменных, команды, которые преобразуют отдельные компоненты данных векторов в 0 или 1, и команду, которая присваивает значение j целочисленной переменной a, если самая левая единица в векторе v находится в разряде j, и полагает $a = 0$, если вектор v состоит из одних нулей.

***1.15.** Создайте машину Тьюринга, которая по двум двоичным целым числам на лентах 1 и 2 печатает их сумму на ленте 3. Можете считать, что левые концы лент отмечены специальным символом #.

1.16. Приведите последовательность конфигураций (мгновенных описаний) ТМ для рис. 1.21, если входом является (*а*) 0010, (*б*) 01110.

***1.17.** Постройте машину Тьюринга, которая решает следующие задачи:

а) Печатает 0^{n^2} на ленте 2, начиная работу с 0^n на ленте 1.

б) Распознает входы вида $0^n10^{n^2}$.

1.18. Укажите множество состояний и функцию переходов машины Тьюринга, моделирующей RAM-команду LOAD 3, как в доказательстве теоремы 1.3.

1.19. Постройте RAM-программу, вычисляющую 22^n по данному n за $O(n)$ шагов. Чему равны (*а*) равномерная и (*б*) логарифмическая стоимость выполнения вашей программы?

***1.20.** Определим $g(m, n)$ равенствами $g(0, n) = n$ и $g(m, n)$ $2^{g(m-1,\, n)}$ при $m > 0$. Напишите RAM-программу, вычисляющую $g(n, n)$ по n. Как соотносятся

равномерная и логарифмическая функции стоимости выполнения вашей программы?

1.21. Выполните процедуру `INTERCHANGE` из раздела 1.8 с фактическими параметрами i и $A[i]$, используя сначала вызов по имени, а затем вызов по ссылке. Будут ли результаты одинаковыми?

Проблема для исследования

Можно ли улучшить верхнюю оценку $O(T2(n))$ времени моделирования машины RAM машиной Тьюринга, как в теореме 1.3?

Библиографические замечания

PAM и RASP получили формальную трактовку в работах Sheperdson and Sturgis [1963], Elgot and Robinson [1964], а также Hartmanis [1971]. В изложении большей части результатов о машинах RAM и RASP в этой главе мы следуем работе Cook and Reckhow [1973].

Идея машины Тьюринга принадлежит Turing [1936]. Более подробное изложение этого понятия можно найти в работе Minsky [1967] и Hopcroft and Ullman [1969]; это же касается и ответа к упражнению 1.8. Изучение временной сложности машин Тьюринга было начато в работе Hartmanis and Stearns [1965], а пространственной сложности — в работах Hartmanis, Lewis and Stearns [1965] и Lewis, Stearns, Hartmanis [1965]. Начиная с работы Blum [1967], делались попытки трактовать вычислительную сложность с гораздо более абстрактной точки зрения. Обзоры можно найти в работах Hartmanis, Hopcroft [1971] и Borodin [1973a].

Rabin [1972] предложил одно интересное обобщение понятия дерева решений.

ГЛАВА 2

Разработка эффективных алгоритмов

Цель этой главы имеет двоякий характер. Во-первых, мы вводим в рассмотрение фундаментальные структуры данных, которые полезны при разработке эффективных алгоритмов для широких классов задач. Во-вторых, описываем методы программирования, такие как рекурсия и динамическое программирование, которые используются во многих эффективных алгоритмах.

В главе l описаны основные модели вычислений. Несмотря на то что нашей первичной моделью является RAM, как правило, мы не будем описывать алгоритмы в терминах такого устройства. По этой причине мы ввели в рассмотрение псевдо-Алгол (см. раздел 1.8). Но даже этот язык оказывается слишком примитивным, если не использовать структуры данных, более сложные, чем массивы. В начале главы рассматриваются такие элементарные структуры данных, как списки и стеки, которые часто используются в эффективных алгоритмах. Мы продемонстрируем, как с помощью этих структур можно представлять множества, графы и деревья. Наше изложение по необходимости будет лаконичным, и читателям, не знакомым с методами обработки списков, следует обратиться к одному из первоисточников, указанных в конце главы, или уделить особое внимание упражнениям.

Мы включили в главу раздел о рекурсии. Один из важных аспектов рекурсии — концептуальное упрощение алгоритмов. Несмотря на то что примеры, приведенные в этой главе, слишком просты, чтобы считать их полноценным обоснованием этого утверждения, в последующих главах рекурсия позволит избежать громоздких подробностей без потери точности изложения сложных алгоритмов. Рекурсия сама по себе не всегда гарантирует эффективность алгоритмов. Однако в сочетании с другими методами, такими как балансировка, "разделяй и властвуй" и алгебраические упрощения, она, как мы увидим, часто позволяет создать эффективные и элегантные алгоритмы.

2.1. Структуры данных: списки, очереди и стеки

Мы предполагаем, что читатели знакомы с элементарными понятиями математики такими, как множества и отношения, а также основными типами данных, такими как целые числа, строки и массивы. В этом разделе мы приводим краткий обзор основных операций над списками.

С математической точки зрения *список* — это конечная последовательность элементов, извлеченных из некоего множества. В описаниях алгоритмов часто используются списки, в которые добавляются и из которых удаляются элементы. В частности, мы можем пожелать добавить или удалить элемент где-то в середине списка. По этой причине нам необходимы структуры данных, позволяющие реализовать списки, в которых можно удалять и добавлять новые элементы без ограничений.

Рассмотрим список

Элемент 1, Элемент 2, Элемент 3, Элемент 4. (2.1)

Простейшей его реализацией является односвязная структура, изображенная на рис. 2.1. Каждый элемент в этой структуре состоит из двух ячеек памяти. Первая ячейка содержит сам элемент,[1] а вторая — указатель на следующий элемент. Эту структуру можно реализовать в виде двух массивов, которые на рис. 2.2 названы NAME и NEXT.[2] Если ITEM — это индекс в рассматриваемом массиве, то NAME[ITEM] — элемент, хранящийся в массиве, а NEXT[ITEM] — индекс следующего элемента в списке, если такой элемент существует. Если ITEM — индекс последнего элемента в списке, то NEXT[ITEM] = 0.

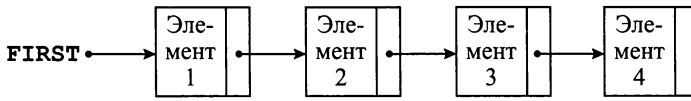


Рис. 2.1. Связанный список

На рис. 2.2 выражение NEXT[0] означает постоянный указатель на первый элемент списка. Заметим, что порядок элементов в массиве NAME не совпадает с порядком элементов в списке. Однако рис. 2.2 правильно изображает список, показанный на рис. 2.1, поскольку массив NEXT располагает элементы в том же порядке, в котором они следуют в списке (2.1).

Следующая процедура вставляет элемент в список. Предполагается, что FREE — это индекс свободной ячейки в массивах NAME и NEXT, а POSITION — индекс элемента в списке, после которого следует вставить ITEM.

---

[1] Если этот элемент сам является сложной структурой, то первая ячейка может содержать указатель на него.

[2] Существует альтернативная (и эквивалентная) точка зрения, в соответствии с которой считается, что каждый элемент хранится в отдельной ячейке. Каждая ячейка обладает адресом, представляющим собой номер первого (возможно, единственного) регистра памяти в группе регистров, зарезервированных для данного элемента. Ячейка состоит из одного или нескольких полей. В данном случае ячейка состоит из полей NAME и NEXT, а выражения NAME[ITEM] и NEXT[ITEM] — это ссылки на эти поля в ячейке с адресом ITEM.

```
procedure INSERT(ITEM, FREE, POSITION):
begin
      NAME[FREE] ← ITEM
      NEXT[FREE] ← NEXT[POSITION]
      NEXT[POSITION] ← FREE
end
```

Трансляция этой процедуры в команды RAM приведет к тому, что время выполнения процедуры `INSERT` не будет зависеть от размера списка.

Пример 2.1. Допустим, мы хотим вставить в список (2.1) новый элемент после второго элемента и в результате получить список

Элемент 1, Элемент 2, Новый элемент, Элемент 3, Элемент 4.

Если пятая ячейка в каждом массиве на рис. 2.2 свободна, то можно вставить новый элемент после Элемента 2 (позиция 3), вызвав процедуру `INSERT`(Новый элемент, 5, 3). В результате выполнения трех инструкций в процедуре `INSERT` получим: `NAME[5]`= Новый элемент, `NEXT[5]=4` и `NEXT[3]=5`. В итоге будут созданы массивы, показанные на рис. 2.3.

	NAME	NEXT
0	—	1
1	Элемент 1	3
2	Элемент 4	0
3	Элемент 2	4
4	Элемент 3	2

Рис. 2.2. Представление списка из четырех элементов

	NAME	NEXT
0	—	1
1	Элемент 1	3
2	Элемент 4	0
3	Элемент 2	5
4	Элемент 5	2
5	Новый элемент	4

Рис. 2.3. Список со вставленным новым элементом

Для того чтобы удалить элемент, следующий за элементом в ячейке I, можно выполнить присваивание `NEXT`[I] = `NEXT`(`NEXT`[I]). При желании индекс удаленного элемента можно записать в список свободных ячеек памяти.

Часто в одном и том же массиве хранятся несколько списков. Как правило, один из этих списков состоит из свободных ячеек. Назовем его *свободным списком*. Для

добавления элемента к списку A можно изменить процедуру INSERT, так чтобы свободная ячейка возникала с помощью удаления первой ячейки в свободном списке. При удалении элемента из списка A соответствующая ячейка возвращается в свободный список для повторного использования.

Этот способ организации памяти не единственно возможный, но он описан здесь для того, чтобы показать, что операции добавления и удаления элементов списка можно выполнить за конечное число шагов, если местоположение элемента, который мы хотим добавить или удалить, определено заранее.

Существуют еще две базовые операции над списками — конкатенация двух списков, т.е. объединение их в один список, и обратная к ней операция разделения списка, после которой исходный список разделяется на два списка, один из которых начинается с элемента, следующего после заданного. Конкатенацию можно выполнить за ограниченное число шагов, добавив в представление списка дополнительный указатель. Этот указатель содержит индекс последнего элемента списка и позволяет не просматривать весь список в поиска его последнего элемента. Разделение списка можно выполнить за ограниченное время, если известен индекс элемента, непосредственно предшествующего месту разделения.

Списки можно обходить в обоих направлениях, если добавить еще один массив с именем PREVIOUS. Значение PREVIOUS[I] — это адрес ячейки, в которой находится элемент списка, который стоит непосредственно перед элементом из ячейки I. Список такого рода называется *двусвязным*. Для вставки и удаления элемента двусвязного списка не обязательно знать адрес предыдущего элемента.

Часто работа со списками ограничивается очень небольшим числом операций. Например, иногда элементы добавляются или удаляются только в конце списка. Иначе говоря, элементы вставляются и удаляются по принципу "последний вошел — первый вышел". В этом случае список называют *стеком* или *магазином*.

Часто стек реализуется в виде отдельного массива. Например, список

Элемент 1, Элемент 2, Элемент 3

можно было бы хранить в массиве NAME, как показано на рис. 2.4. Переменная TOP является указателем на последний элемент, добавленный в стек. Чтобы затолкнуть (PUSH) новый элемент в стек, значение TOP увеличивают на единицу, а затем помещают новый элемент в ячейку NAME[TOP]. (Поскольку массив NAME имеет конечную длину l, перед вставкой нового элемента следует проверить условие TOP $< l - 1$.) Чтобы вытолкнуть (POP) элемент из вершины стека, надо просто уменьшить значение TOP на единицу. Заметим, что не обязательно физически стирать элемент, удаляемый из стека. Чтобы узнать, пуст ли стек, достаточно проверить условие TOP < 0. Очевидно, что время выполнения операций PUSH и POP, а также проверка пустоты стека не зависят от количества элементов в стеке.

Другой специальный вид списка — *очередь*, т.е. список, в который элементы всегда добавляются с одного конца (FRONT), а удаляются с другого (REAR). Как

и стек, очередь можно реализовать в виде массива (рис. 2.5), где продемонстрирована очередь, содержащая список из элементов P, Q, R, S, T. Два указателя обозначают ячейки, соответствующие голове (front) и хвосту (rear) очереди в текущий момент. Чтобы вставить (ENQUEUE) новый элемент в очередь, как и в случае стека, выполняют операцию FRONT = FRONT + 1 и помещают новый элемент в ячейку NAME[FRONT]. Чтобы удалить (DEQUEUE) элемент из очереди, выполняют операцию REAR = REAR + 1. Заметим, что этот способ доступа к элементам основан на принципе "первый вошел — первый вышел".



Рис. 2.4. Реализация стека

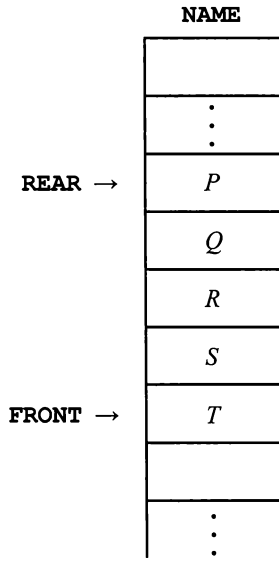


Рис. 2.5. Реализация очереди в виде массива

Поскольку массив NAME имеет конечную длину, скажем, l, то указатели FRONT и REAR рано или поздно достигнут его конца. Если длина списка, представленного этой очередью, никогда не превосходит l, то можно интерпретировать выражение NAME[0] как элемент, следующий за элементом NAME[l − 1].

Элементы, организованные в виде списка, сами могут быть сложными структурами. Например, при работе со списком массивов на самом деле добавляются и удаляются не массивы, а указатели на них, поскольку каждое добавление или удаление массива потребовало бы времени, пропорционального его размеру. Таким образом, сложную структуру можно добавить или удалить за фиксированное время, не зависящее от ее размера.

2.2. Представления множеств

Обычно списки применяются для представления множеств. В этом случае объем памяти, необходимый для представления множества, пропорционален числу его элементов. Количество времени, требуемое для выполнения операции над множествами, зависит от ее природы. Например, предположим, что A и B — два множества. Операция $A \cap B$ требует времени, по крайней мере, пропорционального сумме размеров этих множеств, поскольку списки, представляющие множества A и B, надо просмотреть хотя бы один раз.[3]

Аналогично для выполнения операции $A \cup B$ требуется количество времени, пропорциональное сумме размеров множеств, поскольку надо найти элементы, входящие в оба множества, и удалить один экземпляр каждого такого элемента. Если же множества A и B не пересекаются, найти $A \cup B$ можно за время, не зависящее от размера A и B, выполнив конкатенацию списков, представляющих множества A и B. Задача объединения двух непересекающихся множеств усложняется, если необходимо быстро определить, входит ли заданный элемент в заданное множество. Этот вопрос подробно обсуждается в разделах 4.6 и 4.7.

Альтернативный способ представления множества — вектор битов. Пусть U — универсальное множество (т.е. множество всех множеств), состоящее из n элементов. Упорядочим его линейно. Подмножество $S \subseteq U$ представляется в виде вектора v_S из n битов, в котором i-й бит равен единице тогда и только тогда, когда i-й элемент множества U принадлежит множеству S. Будем называть v_S *характеристическим вектором* множества S.

Преимущество представления множества в виде вектора битов заключается в том, что оно позволяет определять принадлежность i-го элемента множества U заданному множеству за время, не зависящее от размера заданного множества. Более того, основные операции над множествами, такие как объединение и пересечение, можно осуществить как операции $\vee$ и $\wedge$ над векторами битов.

Если мы не хотим считать операции над векторами битов элементарными (выполняемыми за единицу времени), то можно с таким же успехом вместо характеристического вектора определить массив A, для которого $A[i] = 1$ тогда и только тогда, когда i-й элемент множества U принадлежит S. При таком представлении также нетрудно определить, принадлежит ли данный элемент данному множеству. Недостаток этого представления заключается в том, что для выполнения операций объединения и пересечения требуется количество времени, пропорциональное $||U||$,[4] а не размерам рассматриваемых множеств. Аналогично объем памяти, необходимой для хранения множества S, пропорционален $||U||$, а не $||S||$.

---

[3] Если оба списка упорядочены, то найти их пересечение можно с помощью линейного алгоритма.

[4] Здесь $||X||$ обозначает количество элементов (размер, или мощность) множества X.

2.3. Графы

Введем математическое понятие графа и рассмотрим структуры данных, которые обычно применяются для его представления.

Определение. *Граф* $G = (V, E)$ состоит из конечного непустого множества *вершин* V и множества *ребер* E. Если ребра представлены в виде упорядоченных пар вершин (v, w), то граф называется *ориентированным*. Вершина v называется *началом*, а вершина w — *концом* ребра (v, w). Если ребра представляют собой неупорядоченные пары (множества) различных вершин, также обозначаемые (v, w), то граф называют *неориентированным*.[5]

Если в ориентированном графе $G = (V, E)$ пара (v, w) принадлежит множеству ребер E, то вершина w называется *смежной* с вершиной v. Говорят, что ребро (v, w) *ведет из* v *в* w. Количество вершин, смежных с вершиной v, называется *полустепенью исхода* вершины v.

Если в неориентированном графе $G = (V, E)$ пара (v, w) принадлежит множеству ребер E, то считается, что $(v, w) = (w, v)$, так что (w, v) — то же самое ребро. Вершина w называется *смежной* с вершиной v, если (v, w) (следовательно, и (w, v)) принадлежит множеству E. *Степень* вершины — это количество вершин, смежных с ней.

Путем в ориентированном или неориентированном графе называют последовательность ребер вида $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$. Говорят, что этот путь *ведет из* v_1 *в* v_n, а его *длина* равна $n - 1$. Часто такой путь представляют в виде последовательности $v_1, v_2, ..., v_n$ вершин, лежащих на нем. В вырожденном случае одна вершина обозначает путь нулевой длины, ведущий из этой вершины в нее же. Путь называется *простым*, если все ребра и все вершины на этом пути, кроме, возможно, первой и последней, отличаются одна от другой. *Цикл* — это простой путь длины не менее единицы, который начинается и заканчивается в одной и той же вершине. Заметим, что в неориентированном графе длина цикла должна быть не менее трех.

Существует несколько представлений графа $G = (V, E)$. Один из них — *матрица смежности*, т.е. матрица A размером $||V|| \times ||V||$, состоящая из нулей и единиц, в которой $A[i, j] = 1$ тогда и только тогда, когда существует ребро, ведущее из вершины i в вершину j. Представление в виде матрицы смежности удобно для тех алгоритмов на графах, в которых часто требуется знать, существует ли в графе заданное ребро, поскольку время, необходимое для проверки наличия ребра, фиксировано и не зависит от $||V||$ и $||E||$. Основной недостаток матрицы смежности заключается в том, что она занимает память объема $||V||^2$ даже тогда, когда граф содержит только $O(||V||)$ ребер. Даже простая инициализация матрицы смежности требует времени порядка $O(||V||^2)$, что препятствует созданию алгоритмов сложности $O(||V||)$ для работы

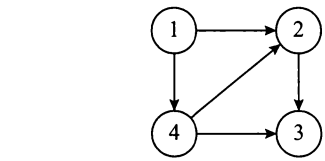
[5] Заметим, что в ориентированном графе может существовать ребро (a, a), а в неориентированном — нет.

с графами, содержащими $O(\|V\|)$ ребер. Несмотря на то что существуют методы для преодоления этой трудности (см. упражнение 2.12), почти неизбежно возникают другие проблемы, которые приводят к тому, что алгоритмы сложности $O(\|V\|)$, основанные на работе с матрицей смежности, являются редкостью.

Интересной альтернативой является представление строк и/или столбцов матрицы смежности в виде векторов битов. Такое представление может значительно повысить эффективность алгоритмов на графах.

Еще одно возможное представление графа основано на списках. *Списком смежности* для вершины v называется список всех вершин w, смежных с вершиной v. Граф можно представить с помощью $\|V\|$ списков смежности, по одному для каждой вершины.

Пример 2.2. На рис. 2.6, *а*, изображен ориентированный граф, содержащий четыре вершины, а на рис. 2.6, *б*, — его матрица смежности. На рис. 2.6, *в*, показаны четыре списка смежности, по одному для каждой вершины. Например, в графе есть ребра, ведущие из вершины 1 в вершины 2 и 4, поэтому список смежности для вершины 1 содержит элементы 2 и 4, связанные так, как показано на рис. 2.1.



а)

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

б)



в)



г)

Рис. 2.6. Ориентированный граф и его представления: (*а*) ориентированный граф; (*б*) матрица смежности; (*в*) списки смежности; (*г*) табличное представление списков смежности

Табличное представление списков смежности приведено на рис. 2.6, *г*. Каждая из первых четырех ячеек в массиве NEXT содержит указатель на первую вершину списка смежности, причем указатель NEXT[*i*] ссылается на первую вершину списка смежности для вершины i. Заметим, что NEXT[3]=0, поскольку список смежности для вершины 3 пуст. Остальные элементы массива NEXT представляют ребра графа. Массив HEAD содержит вершины из списков смежности. Таким образом, список смежности вершины 1 начинается в ячейке 5, поскольку NEXT[1]=5, HEAD[5]=2; это показывает, что существует ребро (1, 2). Равенства NEXT[5]=6 и HEAD[6]=4 означают, что существует ребро (1, 4), а равенство NEXT[6]=0 означает, что больше нет ребер, начинающихся в вершине 1. □

Заметим, что представление графа в виде списков смежности требует объема памяти порядка $||V|| + ||E||$. Представлением с помощью списков смежности часто пользуются, когда $||E|| \ll ||V||^2$.

Если граф является неориентированным, то каждое ребро (v, w) представляется дважды: один раз в списке смежности для вершины v и один раз в списке смежности для вершины w. В этом случае можно добавить новый массив с именем LINK, чтобы согласовать оба экземпляра неориентированного ребра. Таким образом, если i — ячейка, соответствующая вершине w в списке смежности для вершины v, то LINK[i] — ячейка, соответствующая вершине v в списке смежности для вершины w.

Чтобы эффективно удалять ребра из неориентированного графа, списки смежности можно связать дважды (как описано в разделе 2.1). Это бывает необходимо потому, что даже если всегда удалять ребро (v, w), стоящее первым в списке смежности вершины v, то все равно может оказаться, что ребро, ведущее в обратном направлении, стоит в середине списка смежности вершины w. Чтобы быстро удалить ребро (v, w) из списка смежности для вершины w, надо уметь быстро находить ячейку, содержащую предыдущее ребро в этом списке смежности.

2.4. Деревья

Теперь введем очень важный вид ориентированных графов — деревья — и рассмотрим структуры данных, подходящие для их представления.

Определение. Ориентированный граф без циклов называется *ориентированным ациклическим графом*. *Ориентированное дерево* (иногда его называют *корневым деревом*) — это ориентированный ациклический граф, удовлетворяющий следующим условиям.

1. Существует только одна вершина, называемая *корнем*, в которую не входит ни одно ребро.

2. В каждую вершину, кроме корня, входит только одно ребро.
3. Из корня в каждую вершину ведет путь (который, как легко показать, единственный).

Ориентированный граф, состоящий из нескольких деревьев, называется *лесом*. Леса и деревья представляют собой частные случаи ориентированных ациклических графов, которые встречаются настолько часто, что для описания их свойств разработана специальная терминология.

Определение. Пусть $F = (V, E)$ — граф, являющийся лесом. Если (v, w) принадлежит множеству E, то v называется *отцом* вершины w, а w — *сыном* вершины v. Если существует путь из v в w, то v называется *предком* вершины w, а w — *потомком* вершины v. Более того, если $v \neq w$, то v называется *прямым предком* вершины w, а w — *прямым потомком* вершины v. Вершина без прямых потомков называется *листом*. Вершина v и ее потомки в совокупности образуют *поддерево* леса F, и вершина v называется *корнем* этого поддерева.

Глубина вершины v в дереве — это длина пути из корня в вершину v. *Высота вершины* v в дереве — это длина самого длинного пути из вершины v в какой-нибудь лист. *Высотой дерева* называется высота его корня. *Уровень вершины* v в дереве равен разности высоты дерева и глубины вершины v. Например, на рис. 2.7, *а*, вершина 3 имеет глубину 2, высоту 0 и уровень 1.

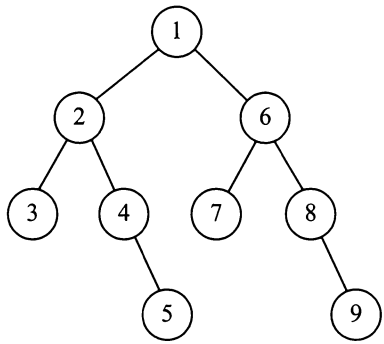
Упорядоченным деревом называется дерево, в котором множество сыновей каждой вершины упорядочено. При изображении упорядоченного дерева будем считать, что множество сыновей каждой вершины упорядочено слева направо. *Бинарным деревом* называется такое упорядоченное дерево, что

1) каждый сын произвольной вершины идентифицируется либо как *левый сын*, либо как *правый*;
2) каждая вершина имеет не более одного левого сына и не более одного правого.

Поддерево T_l (если оно существует), корнем которого является левый сын вершины v, называется *левым поддеревом* вершины v. Аналогично поддерево T_r (если оно существует), корнем которого является правый сын вершины v, называется *правым поддеревом* вершины v. Все вершины в поддереве T_l расположены левее всех вершин в поддереве T_r.

Бинарное дерево обычно представляют в виде двух массивов `LEFTSON` и `RIGHTSON`. Пусть вершины бинарного дерева занумерованы целыми числами от 1 до n. В этом случае условие `LEFTSON[`i`]` $= j$ выполняется тогда и только тогда, когда вершина с номером j является левым сыном вершины с номером i.

Если у вершины i нет левого сына, то LEFTSON[i] = 0. Условия для элемента RIGHTSON[i] определяются аналогично.



	LEFTSON	RIGHTSON
1	2	6
2	3	4
3	0	0
4	0	5
5	0	0
6	7	8
7	0	0
8	0	9
9	0	0

а) б)

Рис. 2.7. Бинарное дерево и его представление

Пример 2.3. Бинарное дерево и его представление изображены на рис. 2.7, *а* и *б*. □

Определение. Бинарное дерево называется *полным*, если для некоторого целого числа k каждая вершина глубины, меньшей k, имеет как левого, так и правого сына, и каждая вершина глубины k является листом. Полное бинарное дерево высоты k имеет ровно $2^{k+1} - 1$ вершин.

Полное бинарное дерево высоты k часто представляют с помощью одного массива. В первой ячейке этого массива находится корень. Левый сын вершины в ячейке i расположен в ячейке $2i$, а его правый сын — в ячейке $2i + 1$. Отец вершины, находящейся в положении $i > 1$, расположен в ячейке $\lfloor i / 2 \rfloor$.

Многие алгоритмы, использующие деревья, часто *обходят* дерево (посещают каждую его вершину) в определенном порядке. Известно несколько систематических способов обхода деревьев Мы рассмотрим три широко распространенных способа обхода: прямой, обратный и симметричный.

Определение. Пусть T — дерево с корнем r и сыновьями $v_1, v_2, ..., v_k$, $k \geq 0$. При $k = 0$ это дерево состоит из единственной вершины r.

Прямой обход дерева T определяется следующей рекурсией:

1) посетить корень r;
2) выполнить прямой обход поддеревьев с корнями $v_1, v_2, ..., v_k$ в указанной последовательности.

Обратный обход дерева T определяется следующей рекурсией:

1) выполнить обратный обход поддеревьев с корнями $v_1, v_2, ..., v_k$ в указанной последовательности;
2) посетить корень r.

Симметричный обход бинарного дерева определяется следующей рекурсией:

1) выполнить симметричный обход левого поддерева корня (если оно существует);
2) посетить корень;
3) выполнить симметричный обход правого поддерева корня (если оно существует).

Пример 2.4. На рис. 2.8 изображено бинарное дерево, вершины которого пронумерованы в соответствии с прямым (рис. 2.8, *а*), обратным (рис. 2.8, *б*) и симметричным обходом (рис. 2.8, *в*). □

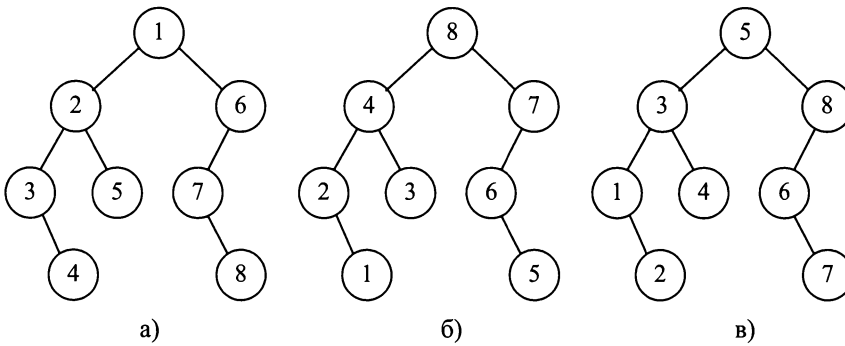


Рис. 2.8. Обход вершин дерева: (*а*) прямой, (*б*) обратный, (*в*) симметричный

Если при обходе дерева его вершинам были присвоены номера, то на вершины удобно ссылаться по этим номерам. Тогда v будет обозначать вершину, которой был присвоен номер v. Если все вершины занумерованы в порядке обхода, то рассмотренные нумерации обладают рядом интересных свойств.

При нумерации в прямом обходе все вершины поддерева с корнем r имеют номера, не меньшие r. Точнее, если D — множество потомков вершины r, то v будет номером некоторой вершины из множества D тогда и только тогда, когда $r \le v < r + ||D_r||$. Поставив в соответствие каждой вершине v ее номер в прямом обходе и количество ее потомков, легко определить, является ли вершина w потомком вершины v. Присвоив вершинам номера в прямом обходе и вычислив количество потомков каждой вершины, на вопрос, является ли вершина w потомком вершины v, можно ответить за фиксированное время, не зависящее от размера дерева. Номера, соответствующие обратному обходу, обладают аналогичным свойством.

При нумерации вершин бинарного дерева в симметричном обходе номера вершин в левом поддереве с корнем v меньше v, а в правом поддереве — больше v. Таким образом, чтобы найти вершину с номером w, надо сравнить w с номером корня r. Если $w = r$, то искомая вершина найдена. Если $w < r$, следует повторить этот процесс для левого поддерева, если $w > r$ — для правого поддерева. В конце концов, вершина с номером w будет найдена. Такие свойства обхода нам понадобятся в следующих главах.

Дадим теперь еще одно, последнее определение, касающееся деревьев.

Определение. *Неориентированным деревом* называется неориентированный ациклический *связный* граф (т.е. неориентированный ациклический граф, любые две вершины которого соединены путем). *Корневое* неориентированное дерево — это неориентированное дерево, в котором одна вершина выделена в качестве корня.

Ориентированное дерево можно превратить в корневое неориентированное, просто сделав все его ребра неориентированными. Мы будем использовать одни и те же термины и обозначения как для неориентированных, так и ориентированных корневых деревьев. Основное математическое различие между ними заключается в том, что все пути в ориентированном дереве ведут от предков к потомкам, а в корневом неориентированном дереве пути могут вести в обоих направлениях.

2.5. Рекурсия

Процедура, которая прямо или косвенно вызывает сама себя, называется *рекурсивной*. Применение рекурсии часто позволяет создавать более четкие и лаконичные описания алгоритмов. В настоящем разделе приводится пример рекурсивного алгоритма и кратко описана реализация рекурсии на машинах RAM.

Рассмотрим определение симметричного обхода бинарного дерева, данное в разделе 2.4. При создании алгоритма, который присваивает вершинам номера в соответствии с симметричным обходом, хорошо было бы отразить в нем опреде-

ление симметричного обхода. Один из таких алгоритмов приведен ниже. Заметим, что он рекурсивно обращается к себе для нумерации поддерева.

Алгоритм 2.1. Нумерация вершин бинарного дерева в соответствии с порядком симметричного обхода.

Вход. Бинарное дерево, представленное массивами LEFTSON и RIGHTSON.

Выход. Массив NUMBER, такой, что NUMBER[*i*] — номер вершины *i* в порядке симметричного обхода.

Метод. Кроме массивов LEFTSON, RIGHTSON и NUMBER, алгоритм использует глобальную переменную COUNT, значением которой является номер очередной вершины в соответствии с симметричным порядком обхода. Начальное значение переменной COUNT равно единице. Параметр VERTEX вначале равен корню. Процедура, изображенная на рис. 2.9, применяется рекурсивно.

```
    procedure INORDER(VERTEX);
    begin
1.      if LEFTSON[VERTEX] ≠ 0 then
            INORDER(LEFTSON[VERTEX]);
2.      NUMBER[VERTEX] ← COUNT;
3.      COUNT ← COUNT + 1;
4.      if RIGHTSON[VERTEX] ≠ 0 then
            INORDER(RIGHTSON[VERTEX]);
    end
```

Рис. 2.9. Рекурсивная процедура симметричного обхода

Сам алгоритм имеет следующий вид.

```
    begin
        COUNT ← 1;
        INORDER(ROOT)
    end  □
```

Рекурсия имеет несколько преимуществ, главное из которых — программы становятся понятнее. Если бы приведенный выше алгоритм не был записан в рекурсивном виде, пришлось бы создать явный механизм для обхода дерева. Двигаться вниз по дереву нетрудно, но, чтобы обеспечить возможность вернуться к предку, необходимо помнить предков в стеке, а инструкции для работы со стеком усложнили бы алгоритм. Его нерекурсивная версия выглядела бы примерно следующим образом.

Алгоритм 2.2. Нерекурсивная версия алгоритма 2.1.

Вход. Как у алгоритма 2.1.

Выход. Как у алгоритма 2.1.

Метод. При прохождении дерева в стеке запоминаются все вершины, которые еще не были пронумерованы и которые лежат на пути из корня в текущую вершину. При переходе из вершины v к ее левому сыну вершина v запоминается в стеке. После обнаружения левого поддерева вершины v она нумеруется и выталкивается из стека. Затем нумеруется правое поддерево вершины v.

При переходе из вершины v к ее правому сыну она не заталкивается в стек, поскольку после нумерации правого поддерева мы планируем возвращаться в вершину v. Вместо этого мы хотим вернуться к тому предку вершины v, который еще не был пронумерован (т.е. к ближайшему предку w вершины v, такому, что v лежит в левом поддереве вершины w). Этот алгоритм приведен на рис. 2.10. □

```
        begin
            COUNT ← 1;
            VERTEX ← ROOT;
            STACK ← пуст;
left:       while LEFTSON[VERTEX] ≠ 0 do
                begin
                    затолкнуть VERTEX в STACK
                    VERTEX ← LEFTSON[VERTEX]
                end;
center:     NUMBER[VERTEX] ← COUNT;
            COUNT ← COUNT + 1;
            if RIGHTSON[VERTEX] ≠ 0 then
                begin
                    VERTEX ← RIGHTSON[VERTEX];
                    goto left;
                end;
            if STACK не пуст then
                begin
                    VERTEX ← элемент вершины стека;
                    вытолкнуть STACK;
                    goto center;
                end;
          end
```

Рис. 2.10. Нерекурсивный алгоритм симметричного обхода

Корректность рекурсивного варианта нетрудно доказать индукцией по числу вершин в бинарном дереве. Корректность нерекурсивного варианта также можно доказать, но в этом случае предположение индукции не столь интуитивно понятно и возникают дополнительные трудности, связанные со стеком и правильным об-

ходом бинарного дерева. С другой стороны, платой за рекурсию может оказаться увеличение временной и пространственной сложностей.

Возникает естественный вопрос: как перевести рекурсивные алгоритмы в инструкции RAM? Согласно теореме 1.2, достаточно рассмотреть построение RASP-программы, так как такие программы можно моделировать с помощью машин RAM с замедлением не более чем в постоянное число раз. Рассмотрим довольно простой способ реализации рекурсии. Он пригоден для всех программ, о которых идет речь в этой книге, но не является универсальным.

В основе реализации рекурсивной процедуры лежит стек, где хранятся данные, участвующие во всех вызовах процедуры, при которых она еще не завершила свою работу. Иными словами, в стеке находятся все локальные данные. Стек разделен на фреймы, представляющие собой блоки последовательных ячеек (регистров). Каждый вызов процедуры использует фрейм стека, длина которого зависит от вызываемой процедуры.

Допустим, сейчас выполняется процедура A. Стек выглядит, как показано на рис. 2.11. Если процедура A вызывает процедуру B, происходит следующее.

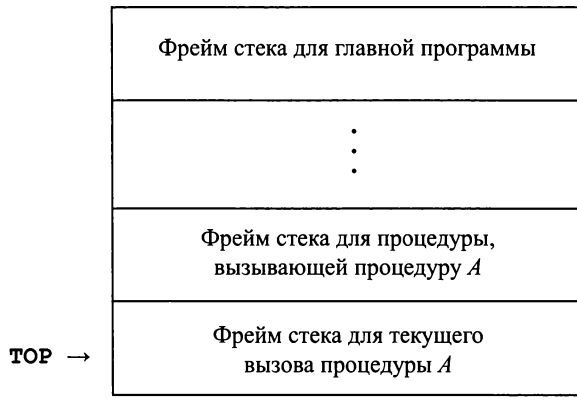


Рис. 2.11. Стек для рекурсивного вызова процедур

1. В вершину стека помещается фрейм стека нужного размера. В него входят следующие параметры, в порядке, известном процедуре B.
   а) Указатели фактических параметров текущего вызова процедуры.[6]
   б) Пустое место для локальных переменных, используемых в процедуры B.

[6]Если фактическим параметром является выражение, то его значение вычисляется во фрейме стека процедуры A, а указатель помещается во фрейм стека процедуры B. Если фактическим параметром является структура (например, массив), то достаточно передать указатель на ее первое слово.

в) Адрес инструкции RASP в процедуре A, которую следует выполнить после того, как данный вызов процедуры B закончит работу (*адрес возврата*).[7] Если B — функция, возвращающая некоторое значение, то во фрейм стека процедуры B также помещается указатель ячейки во фрейме стека для процедуры A, в которую следует поместить это значение (*адрес значения*).

2. Управление переходит к первой инструкции процедуры B. Адрес значения любого параметра или локального идентификатора, принадлежащего процедуре B, определяется с помощью индексации во фрейме стека процедуры B.
3. Когда процедура B заканчивает работу, управление передается процедуре A с помощью следующей последовательности шагов.
   а) Из вершины стека извлекается адрес возврата.
   б) Если B — функция, то значение, заданное выражением из оператора `return`, запоминается в ячейке, заданной адресом значения в стеке.
   в) Фрейм стека процедуры B выталкивается из стека (в вершине стека помещается фрейм процедуры A).
   г) Выполнение процедуры A возобновляется с ячейки, указанной в адресе возврата.

Пример 2.5. Рассмотрим процедуру `INORDER` из алгоритма 2.1. Когда она вызывает себя с фактическим параметром `LEFTSON[VERTEX]`, в ее стеке запоминается адрес нового значения параметра `VERTEX` вместе с адресом возврата, указывающим, что выполнение программы продолжается со строки 2. В результате во всех инструкциях процедуры, в которых встречается переменная `VERTEX`, она заменяется на `LEFTSON[VERTEX]`.

Описанную выше реализацию в определенном смысле моделирует нерекурсивный алгоритм 2.2. Однако там окончание выполнения вызова процедуры `INORDER` с фактическим параметром `RIGHTSON[VERTEX]` завершает выполнение самой вызывающей процедуры. По этой причине нам не обязательно хранить адрес возврата и переменную `VERTEX` в стеке, если фактическим параметром является `RIGHTSON[VERTEX]`. □

Время, требуемое для вызова процедуры, пропорционально времени, которое затрачивается на вычисление значений фактических параметров и запоминание указателей их значений в стеке. Время возврата, конечно, не превосходит этого времени.

При подсчете времени, затрачиваемого несколькими рекурсивными процедурами, обычно легче всего оценить затраты, связанные с вызовом процедуры, кото-

[7] Мы используем модель RAS именно из-за этих переходов к адресам возврата, которые в модели RAM доставляют неудобства (впрочем, вполне преодолимые).

рая выполняет этот вызов. Тогда разность между временем, затрачиваемым на вызов каждой процедуры, и временем, затрачиваемым теми процедурами, которые она вызывает, можно оценить сверху как функцию от размера входных данных. Суммируя эти оценки по всем вызовам процедур, получаем верхнюю границу общего затраченного времени.

Для подсчета времени работы рекурсивного алгоритма применяются рекуррентные уравнения. С i-й процедурой связывается функция $T_i(n)$, обозначающая время выполнения i-й процедуры как функцию некоторого параметра n рассматриваемых входных данных. Обычно рекуррентное уравнение для $T_i(n)$ можно записать относительно времени выполнения процедур, вызываемых процедурой i. Затем решается полученная система рекуррентных уравнений. Часто в алгоритме используется только одна процедура, и $T(n)$ зависит лишь от значений $T(m)$ для конечного числа аргументов m, меньших n. В следующем разделе мы изучим решения некоторых часто встречающихся систем рекуррентных уравнений.

Напомним, что здесь, как в других разделах, весь анализ сложности основан на равномерной функции стоимости. Если брать логарифмическую функцию стоимости, то на анализе временной сложности может сказаться длина стека, используемого для реализации процедур с рекурсией.

2.6. Метод "Разделяй и властвуй"

Задачи часто разделяют на части, находят их решения и затем из них получают решение всей задачи. Этот подход, особенно в его рекурсивном варианте, часто приводит к эффективному решению задач, подзадачи которых представляют собой их меньшие версии. Проиллюстрируем этот метод на двух примерах и проанализируем получающиеся рекуррентные уравнения.

Рассмотрим задачу о нахождении наибольшего и наименьшего элементов множества S, содержащего n элементов. Для простоты будем считать, что n — это степень двойки. Очевидный путь поиска наибольшего и наименьшего элементов состоит в том, чтобы искать их по отдельности. Например, следующая процедура находит наибольший элемент множества S, выполнив n 1 сравнений его элементов.

```
begin
    MAX ← произвольный элемент из S;
    for все другие элементы x из S do
        if x > MAX then MAX ← x
end
```

Аналогично можно найти наименьший из остальных $n - 1$ элементов, произведя $n - 2$ сравнений. В итоге для нахождения наибольшего и наименьшего элементов при $n \geq 2$ потребуется $2n - 3$ сравнений.

Применяя метод "разделяй и властвуй", мы могли бы разбить множество S на два подмножества — S_1 и S_2, каждое из которых содержит $n/2$ элементов. Тогда

описанный выше алгоритм нашел бы наибольший и наименьший элементы в каждой из двух половин с помощью рекурсии. Наибольший и наименьший элементы множества S можно было бы определить, произведя еще два сравнения — наибольших и наименьших элементов в S_1 и S_2 соответственно. Сформулируем этот алгоритм более точно.

Алгоритм 2.3. Нахождение наибольшего и наименьшего элементов множества

Вход. Множество S из n элементов, где n — степень двойки и $n \geq 2$.
Выход. Наибольший и наименьший элементы множества S.
Метод. К множеству S применяется рекурсивная процедура MAXMIN. Она имеет один аргумент,[8] представляющий собой множество S, такое, что $||S|| = 2^k$ для некоторого $k \geq 1$, и возвращает пару (a, b), где a — наибольший и b — наименьший элементы множества S. Процедура MAXMIN приведена на рис. 2.12. □

Заметим, что сравнения элементов множества S происходят только на шаге 3, где сравниваются два элемента множества S, и на шаге 7, где сравниваются max1 с max2 и min1 с min2. Пусть $T(n)$ — количество сравнений элементов множества S, которые надо выполнить в процедуре MAXMIN, чтобы найти наибольший и наименьший элементы множества, состоящего из n элементов. Ясно, что $T(2) = 1$. Если $n > 2$, то $T(n)$ равно сумме общего количества сравнений, выполненных в двух вызовах процедуры MAXMIN (строки 5 и 6), работающей на множествах размера $n/2$, и двух сравнений в строке 7.

```
    procedure MAXMIN(S):

1.  if ||S|| = 2 then
        begin
2.          пусть S ={a, b};
3.          return (MAX(a, b), MIN(a, b))
        end
    else
        begin
4.          разбить S на два равных подмножества S1 и S2;
5.          (max1, min1) ← MAXMIN(S1);
6.          (max2, min2) ← MAXMIN(S2);
7.          return (MAX(max1, max2), MIN(min1 , min2))
        end
```

Рис. 2.12. Процедура для нахождения MAX и MIN

[8]Поскольку здесь подсчитываются только сравнения, способ перебора аргументов не имеет значения. Однако если множество S представлено массивом, можно организовать эффективный вызов процедуры MAXMIN, установив указатели на первый и последний элементы подмножества S, состоящего из последовательных элементов этого массива.

Таким образом, при $n = 2$,

$$T(n) = \begin{cases} 1, & \text{при } n = 2; \\ 2T\dfrac{n}{2} + 2, & \text{при } n > 2. \end{cases} \qquad (2.2)$$

Решением рекуррентных уравнений (2.2) является функция $T(n) = \dfrac{3}{2}n - 2$. Нетрудно убедиться, что эта функция удовлетворяет уравнению (2.2) при $n = 2$, и если она удовлетворяет (2.2) при $n = m$, где $m \geq 2$, то

$$T(2m) = 2\left(\frac{3m}{2} - 2\right) + 2 = \frac{3}{2}(2m) - 2,$$

т.е. она является решением уравнения (2.2) при $n = 2m$. Таким образом, индукцией по n доказано, что $T(n) = \dfrac{3}{2}n - 2$ является решением уравнения (2.2), если n — степень двойки.

Можно показать, что для одновременного нахождения наибольшего и наименьшего элементов множества, состоящего из n элементов, необходимо выполнить не менее $\dfrac{3}{2}n - 2$ сравнений его элементов. Следовательно, алгоритм 2.3 оптимален в смысле количества сравнений элементов множества S, если n — степень двойки.

В предыдущем примере метод "разделяй и властвуй" позволил уменьшить количество сравнений лишь в фиксированное число раз. В следующем примере с помощью этого метода мы уменьшим порядок роста сложности алгоритма.

Рассмотрим умножение двух n-разрядных двоичных чисел. Традиционный метод требует выполнить $O(n^2)$ битовых операций. В методе, изложенном ниже, достаточно выполнить порядка $n^{\log 3}$, т.е. примерно $n^{1,59}$ битовых операций.[9]

Пусть x и y — два n-разрядных двоичных числа. Снова предположим для простоты, что n — степень двойки. Разобьем x и y на две равные части, как показано на рис. 2.13. Если рассматривать каждую из этих частей как $(n/2)$-разрядное число, то можно представить произведение чисел x и y в следующем виде:

$$\begin{aligned} xy &= (a2^{n/2} + b)(c2^{n/2} + d) = \\ &= ac2^n + (ad + bc)2^{n/2} + bd. \end{aligned} \qquad (2.3)$$

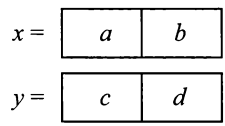


Рис. 2.13. Разбиение битовых строк

[9]Напомним, что по умолчанию все логарифмы в книге являются двоичными.

Равенство (2.3) позволяет вычислить произведение x и y с помощью четырех умножений ($n/2$)-разрядных чисел и нескольких сложений и сдвигов (умножений на степень двойки). Произведение z чисел x и y также можно вычислить по следующей программе:

```
begin
    u ← (a + b) + (c + d);
    v ← a * c;
    w ← b * d;
    z ← v * 2^n + (u - v - w) * 2^(n/2) + w
end
```
(2.4)

На время забудем тот факт, что из-за переноса разряда числа $a + b$ и $c + d$ могут иметь $n/2 + 1$ разрядов, и предположим, что они состоят лишь из $n/2$ разрядов. Описанная выше схема для умножения двух n-разрядных чисел требует только трех умножений ($n/2$)-разрядных чисел и нескольких сложений и сдвигов. Для вычисления произведений чисел u, v и w эту программу можно применять рекурсивно. Сложения и сдвиги занимают $O_B(n)$ времени. Следовательно, временная сложность умножения двух n-разрядных чисел ограничена сверху функцией

$$T(n)=\begin{cases}k & \text{при } n=1,\\ 3T(n/2)+kn & \text{при } n>1,\end{cases} \tag{2.5}$$

где k — константа, соответствующая операциям сложения и сдвига в выражениях, входящих в программу (2.4). Решение рекуррентных уравнений (2.5) ограничено сверху функцией

$$3kn^{\log 3} \approx 3kn^{1,59}.$$

На самом деле можно показать, что в формуле (2.5)

$$T(n) = 3kn^{\log 3} - 2kn.$$

Доказательство проведем индукцией по n, где n — степень двойки. База рекурсии, т.е. случай $n = 1$, тривиальна. Если функция $T(n) = 3kn^{\log 3} - 2kn$ удовлетворяет равенству (2.5) при $n = m$, то по предположению индукции

$$\begin{aligned} T(2m) &= 3T(m) + 2km = \\ &= 3\left(3km^{\log 3} - 2km\right) + 2km = \\ &= 3k(2m)^{\log 3} - 2k(2m). \end{aligned}$$

Отсюда следует, что $T(n) \le 3kn^{\log 3}$. Заметим, что попытка использовать в индукции $3kn^{\log 3}$ вместо $3kn^{\log 3} - 2kn$ не проходит.

Для завершения алгоритма умножения мы должны учесть, что числа $a + b$ и $c + d$, вообще говоря, имеют $n/2 + 1$ разрядов, и поэтому произведение $(a + b) \times (c + d)$ нельзя вычислить непосредственным рекурсивным применением нашего алгоритма к задаче размера $n/2$. Вместо этого нужно записать $a + b$ в виде $a_1 2^{n/2} + b_1$, где a_1 — ведущий бит суммы $a + b$, а b_1 — остальные биты. Аналогично запишем $c + d$ в виде $c_1 2^{n/2} + d_1$. Тогда произведение $(a + b)(c + d)$ можно представить в виде

$$a_1 c_1 2^n + (a_1 d_1 + b_1 c_1) 2^{n/2} + b_1 d_1. \tag{2.6}$$

Слагаемое $b_1 d_1$ вычисляется с помощью рекурсивного применения нашего алгоритма умножения к задаче размера $n/2$. Остальные умножения в выражении (2.6) можно выполнить за время порядка $O_B(n)$, поскольку они содержат в качестве одного из аргументов либо единственный бит a_1 или c_1, либо степень двойки.

Пример 2.6. Этот асимптотически быстрый алгоритм умножения целых чисел можно применять не только к двоичным, но и к десятичным числам. Проиллюстрируем это следующими вычислениями:

$$\begin{array}{lll} x = 3141 & a = 31 & c = 59 \\ y = 5927 & b = \underline{41} & d = \underline{27} \\ & a + b = 72 & c + d = 86 \end{array}$$

$$\begin{aligned} u &= (a+b)(c+d) = 72 \cdot 86 = 6192, \\ v &= ac = 31 \cdot 59 = 1829, \\ w &= bd = 41 \cdot 27 = 1107, \\ xy &= 18\ 290\ 000 + (6192 - 1829 - 1107) \cdot 100 + 1107 = \\ &= 18\ 616\ 707. \quad \square \end{aligned}$$

Заметим, что в алгоритме, основанном на программе (2.4), одно умножение было заменено тремя сложениями и вычитанием (по сравнению с (2.3)). Асимптотическая эффективность, такого алгоритма интуитивно объясняется тем, что умножение выполнить труднее, чем сложение, и для достаточно больших n любое фиксированное количество n-разрядных сложений требует меньше времени, чем n-разрядное умножение, независимо от того, какой из (известных) алгоритмов используется. На первый взгляд, уменьшение числа $(n/2)$-разрядных произведений с четырех до трех может уменьшить общее время в лучшем случае на 25%. Однако соотношения (2.4) применяются рекурсивно для вычисления $(n/2)$-разрядных, $(n/4)$-разрядных и т.д. произведений. Эти 25-процентные уменьшения накапливаются и дают в результате асимптотическое снижение временно*й* сложности.

Временн*ая* сложность процедуры определяется количеством и размером подзадач и в меньшей степени операциями, необходимыми для разбиения данной за-

дачи на подзадачи. Поскольку рекуррентные уравнения вида (2.2) и (2.5) часто возникают при анализе рекурсивных алгоритмов типа "разделяй и властвуй", рассмотрим решение таких уравнений в общем виде.

Теорема 2.1. Пусть a, b и c — неотрицательные константы. Решение рекуррентных уравнений

$$T(n)=\begin{cases} b & \text{при } n=1, \\ aT(n/c)+bn & \text{при } n>1, \end{cases}$$

где n — степень c, имеет вид

$$T(n)=\begin{cases} O(n), & \text{если } a<c, \\ O(n\log n), & \text{если } a=c, \\ O\left(n^{\log_c a}\right), & \text{если } a>c. \end{cases}$$

Доказательство. Если n — степень числа c, то

$$T(n)=bn\sum_{i=0}^{\log_c n} r^i, \text{ где } r=a/c.$$

Если $a < c$, то ряд $\sum_{i=0}^{\infty} r^i$ сходится и $T(n)$ имеет порядок $O(n)$. Если $a = c$, то каждый член суммы равен единице, а всего в ней $O(\log n)$ членов. Поэтому $T(n)=O(n\log n)$. Наконец, если $a > c$, то

$$bn\sum_{i=0}^{\log_c n} r^i = bn\frac{r^{1+\log_c n}-1}{r-1}.$$

Это выражение имеет порядок $O\left(a^{\log_c n}\right)$, что эквивалентно $O\left(n^{\log_c a}\right)$. □

Из теоремы 2.1 вытекает, что разбиение задачи размера n (за линейное время) на две подзадачи размера $n/2$ позволяет создать алгоритм, имеющий сложность $O(n\log n)$. Если бы подзадач было 3, 4 или 8, то получился бы алгоритм сложности порядка $n^{\log 3}$, n^2 и n^3 соответственно. С другой стороны, разбиение задачи на четыре подзадачи размера $n/4$ приводит к алгоритму, имеющему сложность порядка $O(n\log n)$, а разбиение на 9 и 16 подзадач приводит к алгоритму, имеющему порядок сложности $n^{\log 3}$ и n^2 соответственно. Следовательно, асимптотически более быстрый алгоритм умножения целых чисел можно было бы получить, если бы удалось так разбить исходные целые числа на четыре части, чтобы суметь выразить исходное умножение через восемь или меньшее количество умножений чисел меньшей длины. Другой тип рекуррентных соотношений возникает в слу-

чае, когда сложность разбиения задачи не пропорциональна ее размеру. Некоторые типы рекуррентных соотношений описаны в упражнениях.

Если n не является степенью числа c, то обычно можно вложить задачу размера n в задачу размера n', где n' — наименьшая степень числа c, большая или равная n. Таким образом, асимптотические порядки роста, указанные в теореме 2.1, сохраняются для любого n. На практике часто можно разработать рекурсивные алгоритмы, разбивающие задачи произвольного размера на c равных частей, где c велико, насколько возможно. Эти алгоритмы, как правило, эффективнее (на постоянный множитель) тех, которые получаются путем представления размера входа в виде ближайшей сверху степени числа c.

2.7. Балансировка

В обоих наших примерах применения метода "разделяй и властвуй" задача разбивалась на подзадачи одинаковых размеров. Это не было случайностью. Поддержание равновесия — основной принцип разработки хорошего алгоритма. Для иллюстрации этого принципа рассмотрим пример сортировки и сравним эффекты от разбиения задачи на подзадачи одинаковых и неодинаковых размеров. Из этого примера не следует, что метод "разделяй и властвуй" — единственный, в котором полезна балансировка. В главе 4 приводится несколько примеров, в которых балансировки размеров поддеревьев или весов двух операций приводят к эффективным алгоритмам.

Рассмотрим задачу упорядочения целых чисел в порядке неубывания. По-видимому, проще всего было бы найти наименьший элемент, просмотрев всю последовательность и поменяв местами наименьший элемент с первым. Этот процесс применяется к оставшимся $n - 1$ элементам, и в результате наименьший элемент оказывается на втором месте. Применение этого процесса к остальным $n - 2$, $n - 3$, ..., 2 элементам упорядочивает всю последовательность в порядке неубывания.

Этот алгоритм приводит к рекуррентным уравнениям

$$T(n) = \begin{cases} 0, & \text{если } n = 1, \\ T(n-1) + n - 1, & \text{если } n > 1, \end{cases} \tag{2.7}$$

которые задают количество сравнений упорядочиваемых элементов. Решением уравнений (2.7) является функция $T(n) = n(n-1)/2$, имеющая порядок $O(n^2)$.

Этот алгоритм можно считать рекурсивным применением метода "разделяй и властвуй" с разбиением задачи на неравные части, но при больших n он не эффективен. Для разработки асимптотически эффективного алгоритма сортировки необходимо обеспечить балансировку. Вместо того чтобы разбивать задачу размера n на две подзадачи, одна из которых имеет размер 1, а другая — $n - 1$, необхо-

димо разбить ее на две подзадачи с размерами примерно по $n/2$. Эту задачу решает *сортировка слиянием*.

Рассмотрим последовательность целых чисел $x_1, x_2, ..., x_n$. Снова предположим для простоты, что n — степень двойки. Один из способов упорядочить эту последовательность — разделить ее на две подпоследовательности, $x_1, x_2, ..., x_{n/2}$ и $x_{n/2+1}, ..., x_n$, упорядочить каждую из них, а затем выполнить их слияние. Под слиянием здесь понимается объединение двух уже упорядоченных последовательностей в одну упорядоченную последовательность.

Алгоритм 2.4. Сортировка слиянием

Вход. Последовательность чисел $x_1, x_2, ..., x_n$, где n — степень двойки.

Выход. Последовательность $y_1, y_2, ..., y_n$, являющаяся перестановкой входной последовательности и удовлетворяющая неравенствам $y_1 \le y_2 \le ... \le y_n$.

Метод. Применим процедуру MERGE(S, T), входом которой служат две упорядоченные последовательности S и T, а выходом — последовательность элементов из S и T, упорядоченных в порядке неубывания. Поскольку последовательности S и T упорядочены, процедура MERGE требует не больше сравнений, чем сумма длин S и T без единицы. Эта повторяющаяся процедура выбирает больший из наибольших элементов, остающихся в последовательностях S и T, а затем удаляет этот элемент. Если наибольшие элементы в обеих последовательностях совпадают, то предпочтение отдается последовательности S.

Кроме того, в этом алгоритме применяется процедура SORT(i, j) (рис. 2.14), сортирующая подпоследовательность $x_1, x_{i+1}, ..., x_j$ в предположении, что она имеет длину 2^k при некотором $k \ge 0$.

Для сортировки последовательности $x_1, x_2, ..., x_n$ выполняется вызов процедуры SORT(1, n). □

```
procedure SORT(i, j):
if i = j then return x_i
else
    begin
        m ← (i + j - 1)/2;
        return MERGESORT(SORT(i, m), SORT(m + 1, j))
    end
```

Рис. 2.14. Сортировка слиянием

Подсчет количества сравнений в алгоритме 2.4 приводит к рекуррентным уравнениям

$$T(n)=\begin{cases}0, & \text{если } n=1,\\ T(n-1)+n-1, & \text{если } n>1,\end{cases}$$

решением которых по теореме 2.1 является функция $T(n)=O(n\log n)$. При больших n сбалансированность размеров подзадач обеспечивает значительное преимущество. Аналогичный анализ показывает, что общее время работы процедуры `SORT`, помимо операций сравнения, также имеет порядок $O(n\log n)$.

2.8. Динамическое программирование

Рекурсия полезна, если разбиение задачи на подзадачи не требует значительных усилий, а суммарный размер подзадач невелик. Из теоремы 2.1 следует, что если сумма размеров подзадач равна an для некоторой константы $a>1$, то рекурсивный алгоритм, скорее всего, имеет полиномиальную временную сложность. Однако если очевидное разбиение задачи размера n сводит ее к n задачам размера $n-1$, то рекурсивный алгоритм, скорее всего, имеет экспоненциальную сложность. В этом случае более эффективные алгоритмы часто можно получить с помощью табличного метода, который называется *динамическим программированием*.

По существу, динамическое программирование вычисляет решение всех подзадач. Вычисление идет от меньших подзадач к большим, и ответы запоминаются в таблице. Преимущество этого метода состоит в том, что решения подзадач сохраняются и никогда не вычисляются заново. Этот метод легко понять на простом примере.

Рассмотрим вычисление произведения n матриц

$$M=M_1\times M_2\times\ldots\times M_n,$$

где M_i — матрица с r_{i-1} строками и r_i столбцами. Порядок перемножения этих матриц может существенно повлиять на общее количество операций, необходимых для вычисления матрицы M, независимо от алгоритма умножения матриц.

Пример 2.7. Предположим, что умножение матрицы размерностью $p\times q$ на матрицу размерностью $q\times r$ требует pqr операций, как в обычном алгоритме. Рассмотрим произведение

$$M=\underset{[10\times 20]}{M_1}\times\underset{[20\times 50]}{M_2}\times\underset{[50\times 1]}{M_3}\times\underset{[1\times 100]}{M_4},$$

где размеры каждой матрицы M_i указаны в скобках. Если вычислять матрицу M в порядке

$$M_1\times(M_2\times(M_3\times M_4)),$$

то потребуется 125000 операций, тогда как для вычисления M в порядке

$$(M_1 \times (M_2 \times M_3)) \times M_4,$$

понадобятся лишь 2200 операций. □

Процесс перебора всех вариантов порядка вычисления рассматриваемого произведения n матриц с целью минимизировать количество операций имеет экспоненциальную сложность (см. упражнение 2.31), что при больших n непрактично. Однако динамическое программирование приводит к алгоритму, имеющему сложность $O(n^3)$. Пусть m_{ij} — минимальная сложность вычисления $M_i \times M_{i+1} \times ... \times M_j$ при $1 \le i \le j \le n$. Очевидно, что

$$m_{ij} = \begin{cases} 0, & \text{если } i = j, \\ \underset{i \le k \le j}{\mathrm{MIN}}\left(m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j\right), & \text{если } j > i, \end{cases} \tag{2.9}$$

где m_{ik} — минимальная сложность вычисления произведения матриц $M' = M_i \times \times M_{i+1} \times ... \times M_k$, второй член, $m_{k+1,j}$, представляет собой минимальную сложность вычисления матрицы

$$M'' = M_{k+1} \times M_{k+2} \times ... \times M_j,$$

а третье слагаемое — это сложность перемножения матриц M' и M''. Обратите внимание на то, что M' — матрица размерностью $r_{i-1} \times r_k$, а M'' — матрица размерностью $r_k \times r_j$. Формула (2.9) утверждает, что m_{ij}, $(j > i)$ — наименьшая из сумм этих трех членов по всем возможным значениям k, лежащим между i и $j - 1$.

При динамическом программировании величины m_{ij} вычисляются в порядке возрастания разностей нижних индексов. Вычисления начинаются с m_{ii} для всех i, затем $m_{i,i+1}$ для всех i, затем $m_{i,i+2}$ и т.д. Таким образом, когда мы приступим к вычислению m_{ij}, члены m_{ik} и $m_{k+1,j}$ в формуле (2.9) будут уже вычислены. Это следует из того, что при $i \le k < j$ разность $j - i$ должна быть больше $k - i$, а также и $j - (k + 1)$. Соответствующий алгоритм приведен ниже.

Алгоритм 2.5. Алгоритм динамического программирования для вычисления порядка, минимизирующего сложность перемножения n матриц $M = M_1 \times M_2 \times ... \times M_n$.

Вход. $r_0, r_1, ..., r_n$, где r_{i-1} и r_i — размерности матрицы M_i.

Выход. Минимальная сложность умножения матриц M_i при условии, что для умножения матрицы размерностью $p \times q$ на матрицу размерностью $q \times r$ требуется pqr операций.

Метод. Алгоритм, приведенный на рис. 2.15. □

```
    begin
1.      for i ← 1 until n do m_ii ← 0:
2.      for l ← 1 until n - 1 do
3.          for i ← 1 until n - l do
                begin
4.                  j ← i + l;
5.                  m_ij ← MIN_{i≤k<j} (m_ik + m_{k+1,j} + r_{i-1} * r_k * r_j)
                end;
6.      write m_1n
    end
```

Рис. 2.15. Алгоритм динамического программирования для вычисления порядка при перемножении матриц

Пример 2.8. Если применить этот алгоритм к цепочке из четырех матриц (2.8), где r_0, ..., r_4 равны соответственно 10, 20, 50, 1, 100, то в результате вычислений получатся значения m_{ij}, приведенные на рис. 2.16. Таким образом, минимальное количество операций, необходимых для вычисления этого произведения, равно 2200. Порядок, в котором можно выполнить эти умножения, легко определить, приписав каждой ячейке таблицы то значение k, на котором достигается минимум в формуле (2.9). □

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10{,}000$	$m_{23} = 1000$	$m_{34} = 5000$	
$m_{13} = 1200$	$m_{24} = 3000$		
$m_{14} = 2200$			

Рис. 2.16. Сложность вычисления произведений $M_i \times M_{i+1} \times ... \times M_j$

2.9. Заключение

В этой главе рассмотрен ряд основных методов, которые используются при разработке эффективных алгоритмов. Мы увидели, как структуры высокого уровня, такие как списки, очереди и стеки, избавляют разработчиков алгоритмов от утомительной работы, например, связанной с указателями, позволяя сосредоточиться на общей структуре самого алгоритма. Кроме того, было показано, как мощные методы рекурсии и динамического программирования часто приводят к элегантным и естественным алгоритмам. Мы также ознакомились с некоторыми общими методами, такими как "разделяй и властвуй" и балансировка.

Конечно, эти методы — не единственно доступные, но являются одними из важнейших. Далее в этой книге мы рассмотрим и другие методы. Они будут относиться к различным задачам — от выбора подходящего представления до определения разумного порядка выполнения операций. Возможно, важнейшим качеством

хорошего разработчика алгоритмов должно быть постоянное чувство неудовлетворенности. Разработчик должен исследовать задачу с разных точек зрения, пока не убедится, что создал алгоритм, наиболее подходящий для его целей.

Упражнения

2.1. Выберите реализацию для дважды связанного списка. Напишите алгоритм на псевдо-Алголе для вставки и удаления элемента. Убедитесь в работоспособности своей программы на примерах, когда требуется удалить первый и (или) последний элементы и когда список пуст.

2.2. Напишите алгоритм для изменения порядка следования элементов в списке на противоположный. Докажите, что он работает правильно.

2.3. Напишите алгоритмы, реализующие операции `PUSH`, `POP`, `ENQUEUE` и `DEQUEUE`, упомянутые в разделе 2.1. Не забывайте проверять, достиг ли указатель конца массива, зарезервированного для стека или очереди.

2.4. Напишите условия для проверки пустоты очереди. Допустим, массив `NAME` из раздела 2.1 имеет размер k. Сколько элементов можно хранить в соответствующей очереди? Создайте рисунки, иллюстрирующие эту очередь и типичные положения указателей `FRONT` и `REAR` в случае, когда очередь (*а*) пустая, (*б*) содержит один элемент и (*в*) полная.

2.5. Напишите алгоритм для удаления первого ребра (v, w) из списка смежности для вершины v в неориентированном графе. Считайте, что списки смежности дважды связаны и что массив `LINK` помещает вершину v в список смежности вершины w, как описано в разделе 2.3.

2.6. Напишите алгоритм для построения списков смежности для неориентированного графа. Каждое ребро (v, w) необходимо представить дважды: в списках смежности для v и для w. Оба экземпляра каждого ребра должны связываться между собой так, что если один из них удаляется, то другой также можно легко удалить.

***2.7.** (*Топологическая сортировка*). Пусть $G = (V, E)$ — ориентированный ациклический граф. Напишите алгоритм, присваивающий целые числа вершинам графа G по следующему правилу: если из вершины с номером i в вершину с номером j идет ориентированное ребро, то $i < j$. (*Подсказка*: ациклический граф должен иметь вершину, в которую не входит ни одно ребро. Почему? Одно из решений этой задачи можно получить так: находим вершину, в которую не входят ребра, приписываем ей наименьший номер и удаляем из графа вместе со всеми ребрами, выходящими из нее. Повторяем этот процесс для графа, полученного в результате такого удаления, присваиваем

следующий наименьший номер и т.д. Для того чтобы обеспечить эффективность этого алгоритма, т.е. чтобы он имел сложность $O(\|E\| + \|V\|)$, необходимо, чтобы в поисках вершины, в которую не входит ни одно ребро, не приходилось просматривать каждый новый граф.

***2.8.** Пусть $G = (V, E)$ — ориентированный ациклический граф с двумя выделенными вершинами — начальной и целевой. Напишите алгоритм для нахождения такого множества путей из начальной вершины в целевую, чтобы

а) ни одна вершина, кроме начальной и целевой, не лежала на двух путях;

б) в этом множество нельзя было добавить ни одного нового пути, не нарушив условия 1.

Заметим, что этим условиям удовлетворяет много множеств. Не обязательно находить множество с наибольшим числом путей, достаточно какого-нибудь множества, удовлетворяющего приведенным выше условиям. Ваш алгоритм должен иметь временную сложность $O(\|E\| + \|V\|)$.

***2.9.** (*Задача об устойчивом паросочетании*). Пусть B — множество из n юношей, а G — множество из n девушек. Каждый юноша оценивает девушек числами от 1 до n, и каждая девушка оценивает юношей числами от 1 до n. *Паросочетанием* называется взаимно однозначное соответствие между юношами и девушками. Паросочетание устойчиво, если для любых двух юношей b_1 и b_2 и соответствующих им в этом паросочетании девушек g_1 и g_2 выполняются следующие два условия:

а) либо b_1 оценивает g_1 выше, чем g_2, либо g_2 оценивает b_2 выше, чем b_1;

б) либо b_2 оценивает g_2 выше, чем g_1, либо g_1 оценивает b_1 выше, чем b_2.

Докажите, что устойчивое паросочетание всегда существует. Напишите алгоритм для нахождения одного из таких паросочетаний.

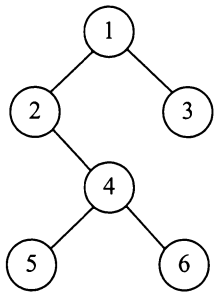
2.10. Рассмотрим бинарное дерево, вершинам которого приписаны имена. Напишите алгоритм, печатающий эти имена в (*а*) прямом порядке, (*б*) обратном и (*в*) симметричном.

***2.11.** Напишите алгоритм для вычисления значений арифметических выражений, содержащих операции + и × и представленных в (*а*) префиксной польской записи, (*б*) инфиксной записи и (*в*) постфиксной польской записи.

***2.12.** Разработайте метод, который позволял бы приравнивать к нулю те элементы матрицы, которые выбираются первый раз: таким способом устраняется работа по инициализации исходной записи матрицы смежности, занимающая время $O(\|V\|^2)$. (*Подсказка*: предусмотрите в каждом инициализируемом элементе матрицы указатель на обратный указатель в стеке. Всякий раз, когда выбирается какой-то элемент, проверяйте, что он имеет неслу-

чайное значение, т.е. что его указатель указывает на активную часть стека, а соответствующий обратный указатель — на выбираемый элемент.)

2.13. Рассмотрите работу алгоритмов 2.1 и 2.2 на бинарном дереве рис. 2.17.



	LEFTSON	RIGHTSON
1	2	3
2	0	4
3	0	0
4	5	6
5	0	0
6	0	0

Рис. 2.17. Бинарное дерево

***2.14.** Докажите, что алгоритм 2.2 корректен.

***2.15.** (*Ханойские башни*). Имеются три стержня, *A*, *B* и *C*, и *n* дисков разных размеров. Вначале все диски нанизаны на стержень *A* в порядке убывания размеров, так что наибольший диск находится внизу. Цель задачи — переместить диски со стержня *A* на стержень *B*, чтобы никакой больший диск ни разу не оказался над меньшим диском. За один шаг можно перемещать только один диск. Стержень *C* можно использовать для временного хранения дисков. Напишите рекурсивный алгоритм решения этой задачи. Каково время работы вашего алгоритма в терминах числа перемещений дисков?

****2.16.** Решите упражнение 2.15 с помощью алгоритма без рекурсии. Какой из алгоритмов легче понять и корректность какого из них проще доказать?

****2.17.** Докажите, что для решения упражнения 2.15 необходимо и достаточно 2^{n-1} перемещений.

2.18. Напишите алгоритм генерирования всех перестановок целых чисел от 1 до *n*. (*Подсказка*: множество перестановок целых чисел от 1 до *n* можно получить из множества перестановок целых чисел от 1 до $n - 1$, вставляя *n* во все возможные позиции в каждой перестановке.)

2.19. Напишите алгоритм нахождения высоты бинарного дерева, представленного на рис. 2.7, *б*.

2.20. Напишите алгоритм подсчета числа потомков каждой вершины дерева.

****2.21.** Рассмотрим двухпозиционный переключатель с двумя входами и двумя выходами, показанный на рис. 2.18. В одной позиции входы 1 и 2 соединяются

соответственно с выходами 1 и 2, в другой — с выходами 2 и 1. Используя эти переключатели, создайте сеть с n входами и n выходами, на которой можно получить любую из возможных перестановок входов. В этой сети должно быть не более $O(n\log n)$ переключателей.

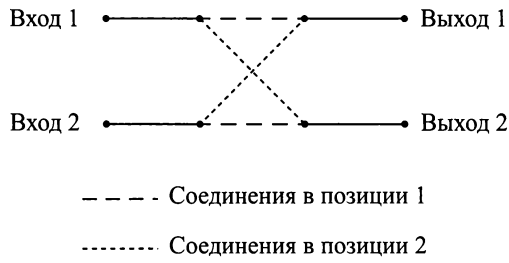


Рис. 2.18. Двухпозиционный переключатель

2.22. Напишите программу RASP, которая выполняла бы следующую процедуру вычисления программы, вычисляющей $\binom{n}{m}$ — число сочетаний по m из n:

```
procedure COMB(n, m):
if m = 0 or n = m then return 1
else return (COMB(n - 1, m) + COMB(n - 1, m - 1)
```

Текущие значения n и m, а также адреса возвращаемого значения и значений, передаваемых при вызове, можно хранить в стеке.

***2.23.** Иногда задачу размера n удобно разбить на $\sqrt{n}$ подзадач размера $\sqrt{n}$. В результате получается рекуррентное уравнение вида

$$T\left(\frac{n^2}{2^r}\right) = nT(n) + bn^2,$$

где r — целое число, $r \geq 1$. Покажите, что решение этого рекуррентного уравнения имеет порядок $O\left(n(\log n)^r \log\log n\right)$.

2.24. Вычислите следующие суммы:

а) $\sum_{i=1}^{n} i$; **б)** $\sum_{i=1}^{n} a^i$; **в)** $\sum_{i=1}^{n} ia^i$;

г) $\sum_{i=1}^{k} 2^{k-i} i^2$; **д)** $\sum_{i=1}^{n} \binom{n}{i}$; **е)** $\sum_{i=1}^{n} i\binom{n}{i}$.

*2.25. Решите следующие рекуррентные уравнения при $T(1) = 1$:

а) $T(n) = aT(n - 1) + bn$,

б) $T(n) = aT(n/2) + bn \log n$,

в) $T(n) = aT(n - 1) + bn^c$,

г) $T(n) = aT(n/2) + bn^c$.

*2.26. Модифицируйте алгоритм 2.3 для нахождения наибольшего и наименьшего элементов множества, разрешив рекурсию до уровня $\|S\| = 1$. Какова асимптотическая скорость роста числа сравнений?

*2.27. Покажите, что для одновременного нахождения наибольшего и наименьшего элементов n-элементного множества необходимо и достаточно $\left\lceil \frac{3}{2}n - 2 \right\rceil$ сравнений.

*2.28. Модифицируйте алгоритм для умножения целых чисел с помощью разбиения каждого целого числа на (*а*) три и (*б*) четыре части. Какова сложность каждого из этих алгоритмов?

*2.29. Пусть A — массив размера n, состоящий из положительных и отрицательных целых чисел, причем $A[1] < A[2] < \ldots < A[n]$. Напишите алгоритм нахождения числа i, для которого $A[i] = i$ (если такое число существует). Какова временная сложность вашего алгоритма? Докажите, что $O_C(\log n)$ — наилучший возможный порядок.

**2.30. Для числа n, не являющегося степенью двойки, из алгоритма 2.4 можно получить корректный алгоритм сортировки слиянием, если заменить оператор $m \leftarrow (i + j - 1)/2$ на рис. 2.14 оператором $m \leftarrow \lfloor (i + j)/2 \rfloor$. Пусть $T(n)$ — число сравнений, необходимых для сортировки n элементов этим методом.

а) Покажите, что

$$T(1) = 0,$$
$$T(n) = T\lfloor n/2 \rfloor + T\lceil n/2 \rceil + n - 1.$$

б) Покажите, что решением этого рекуррентного уравнения является функция

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

*2.31. Покажите, что решением рекуррентного уравнения

$$X(1)=1,$$
$$X(n)=\sum_{i=1}^{n-1}X(i)X(n-i),\ \text{при } n>1$$

является функция

$$X(n+1)=\frac{1}{n+1}\binom{2n}{n}.$$

где $X(n)$— это число способов правильной расстановки скобок в цепочке из n символов. Числа $X(n)$ называются числами Каталана. Покажите, что $X(n)\geq 2^{n-2}$.

2.32. Модифицируйте алгоритм 2.5 так, чтобы он выдавал порядок перемножения матриц, при котором количество умножений их элементов становится минимальным.

2.33. Напишите эффективный алгоритм для определения порядка, в котором следует вычислять произведение матриц $M_1 \times M_1 \times ... \times M_n$, чтобы минимизировать число умножений элементов в случае, когда каждая матрица M_i имеет один из размеров 1×1, $1 \times d$, $d \times 1$ или $d \times d$ при некотором фиксированном d.

Определение. *Контекстно-свободной грамматикой* G в нормальной форме Хомского называется четверка (N, Σ, P, S), где (1) N — конечное множество *нетерминальных символов*; (2) Σ — конечное множество *терминальных символов*; (3) P — конечное множество пар, называемых *правилами вывода* и имеющими вид $A \to BC$ или $A \to a$ (A, B, C — символы из N и a — символ из множества Σ); и (4) S — выделенный символ из N. Мы будем писать $\alpha A\gamma \Rightarrow \alpha\beta\gamma$, если α, β, γ — строки, состоящие из терминальных и нетерминальных символов, и пара $A \to \beta$ принадлежит P. *Языком* $L(G)$, *порождаемым грамматикой* G, называется множество строк терминальных символов $\left\{w \middle| S \overset{*}{\Rightarrow} w\right\}$, где $\overset{*}{\Rightarrow}$ обозначает рефлексивное и транзитивное замыкание отношения $\Rightarrow$.

***2.34.** Напишите алгоритм сложности $O(n^3)$, который определял бы принадлежность данной строки $w = a_1a_2...a_n$ языку $L(G)$, где $G = (N, \Sigma, P, S)$ — контекстно-свободная грамматика в нормальной форме Хомского. (*Подсказка.* Пусть $m_{ij} = \left\{A \middle| A \in N \text{ и } A \overset{*}{\Rightarrow} a_ia_{i+1}...a_j\right\}$. Слово w принадлежит языку $L(G)$ тогда и только тогда, когда $S \in m_{1n}$. Для вычисления m_{ij} примените динамическое программирование.)

***2.35.** Пусть x и y — строки символов из некоторого алфавита. Рассмотрите операции удаления символа из строки x, вставки символа в строку x и замены символа в строке x другим символом. Опишите алгоритм нахождения минимального числа таких операций, необходимых для преобразования строки x в строку y.

Библиографические замечания

Дополнительную информацию о структурах данных и их реализации можно найти в книгах Knuth [1968] и Stone [1972]. Книга Pratt [1975] содержит описание реализации рекурсии в языках, подобных Алголу. Теория графов изложена в книгах Berge [1958] и Harary [1969]. Knuth [1968] приводит информацию о деревьях и алгоритмах их обхода. Дальнейшие сведения об алгоритмах обхода деревьев можно найти у Burkhard [1973].

Оптимальность алгоритма 2.3 (для нахождения максимума и минимума) показал Pohl [1072]. Алгоритм сложности $O(n^{1.59})$ для целых чисел (см. раздел 2.6) приведен в работе Karatsuba and Ofman [1962]. Winograd [1973] рассматривает аналогичные методы ускорения с более общей точки зрения.

Понятие динамического программирования развивал Bellman [1957]. Алгоритм 2.5 представляет собой известное приложение этого понятия, опубликованное Godbole [1973], а также Muraoka и Kuck [1973]. Приложением динамического программирования к распознаванию принадлежности строки к контекстно-свободному языку (см. упражнение 2.34) независимо друг от друга занимались J. Cocke and Kasami [1965], а также Younger [1967]. Решение упражнения 2.35 приведено в работе Wagner and Fisher [1974].

Дальнейшую информацию о решении рекуррентных уравнений см. в книгах Liu [1968] или Sloane [1973].

ГЛАВА 3

Сортировка и порядковые статистики

В этой главе изучаются две важные взаимосвязанные задачи — сортировка последовательности элементов и выбор k-го наименьшего элемента последовательности. Под сортировкой последовательности подразумевается такая перестановка, после которой все элементы последовательности расположены в порядке невозрастания или неубывания. Для того чтобы найти k-й наименьший элемент последовательности, можно упорядочить ее элементы в порядке неубывания и выбрать из полученной последовательности k-й элемент. Однако мы увидим, что существуют более быстрые способы поиска k-го наименьшего элемента.

Сортировка — практически важная и теоретически интересная задача. Поскольку значительную часть коммерческой обработки данных составляет сортировка больших объемов данных, эффективные алгоритмы сортировки имеют большое экономическое значение. Кроме того, процесс сортировки последовательности элементов является неотъемлемой частью многих алгоритмов.

В главе рассматриваются два класса алгоритмов сортировки. Алгоритмы первого класса основаны на использовании структуры сортируемых элементов. Например, если сортируемые элементы представляют собой числа от 0 до $m - 1$, то последовательность из n элементов можно упорядочить за время $O(n + m)$, а если строками в некотором алфавите — то за время, пропорциональное сумме длин этих слов.

Второй класс алгоритмов не предполагает, что у сортируемых элементов существует какая-то структура. Их основная операция — сравнение двух элементов. В алгоритмах этого типа, как мы увидим, для сортировки последовательности, состоящей из n элементов, необходимо выполнить, по крайней мере, $n\log n$ сравнений. Мы приведем два алгоритма сортировки алгоритма сложности $O_C(n\log n)$: пирамидальная сортировка со сложностью $O_C(n\log n)$ в худшем случае и быстрая сортировка со средней сложностью $O_C(n\log n)$.

3.1. Задача сортировки

Определение. *Частичным порядком* на множестве S называется такое двухместное отношение R, что для любых a, b и c из S выполняются следующие условия:

1. aRa (R — *рефлексивно*).
2. Из aRb и bRc следует aRc (R — транзитивно).
3. Из aRb и bRa следует $a = b$ (R — антисимметрично).

Примерами частичных порядков являются отношение $\leq$ на множестве целых чисел и отношение $\subseteq$ (включение множеств).

Линейным, или *полным*, порядком на множестве S называется такой частичный порядок R на множестве S, что для любых двух элементов a и b выполняется либо aRb, либо bRa. Отношение $\leq$ на множестве целых чисел является линейным порядком, а включение множеств $\subseteq$ — нет.[1]

Задачу сортировки можно сформулировать следующим образом. Пусть задана последовательность из n элементов $a_1, a_2, \ldots, a_n$, выбранных из множества, на котором задан линейный порядок (обычно мы будем обозначать его символом $\leq$). Необходимо найти перестановку π этих n элементов, которая отобразит данную последовательность в неубывающую последовательность $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$, т.е. $a_{\pi(i)} \leq a_{\pi(i+1)}$ при $1 \leq i < n$. Как правило, мы будем создавать саму упорядоченную последовательность, а не упорядочивающую перестановку π.

Методы сортировки разделяются на *внутренние* (когда данные размещаются в памяти с произвольным доступом) и *внешние* (когда данные размещаются преимущественно за пределами памяти с произвольным доступом). Внешняя сортировка является частью таких приложений, как обработка счетов, при которой в работу вовлекается гораздо больше элементов, чем можно сразу запомнить в оперативной памяти. Поэтому методы внешней сортировки, применимые к данным, находящимся на внешних устройствах памяти (таких, как диски или магнитные ленты), имеют огромное коммерческое значение.

Внутренняя сортировка играет важную роль в проектировании алгоритмов и коммерческих приложений. В тех случаях, когда сортировка является частью другого алгоритма, количество сортируемых элементов обычно невелико, и они помещаются в оперативной памяти. Тем не менее мы будем предполагать, что элементов, подлежащих сортировке, довольно много. Если же необходимо упоря-

[1] Если $\leq$ — линейный порядок, то пишут $a < b$, если $a < b$ и $a \neq b$ (вполне естественно). Кроме того, выражение $b > a$ — это синоним $a < b$, а $b \geq a$ эквивалентно $a \leq b$.

дочить только небольшое количество элементов, то гораздо выгоднее применять простую стратегию вроде метода пузырька, имеющего сложность $O(n^2)$ (упражнение 3.5).

Существует много алгоритмов сортировки. Мы не будем пытаться описать все важные алгоритмы, а ограничимся методами, оказавшимися полезными при разработке алгоритмов. Сначала мы рассмотрим вариант, в котором сортируемые элементы являются целыми числами или (что почти эквивалентно) строками в конечном алфавите. В этом случае сортировку можно выполнить за линейное время. Затем мы исследуем задачу сортировки, в которой специальные свойства чисел или строк не используются, и приходится строить разветвления в программе только на основе результатов сравнения сортируемых элементов. В таком случае необходимо выполнить $O(n \log n)$ сравнений. Как мы покажем, это не только необходимое, но и достаточное условие для упорядочения последовательности из n элементов.

3.2. Поразрядная сортировка

Начнем с последовательности целых чисел $a_1, a_2, ..., a_n$ из диапазона от 0 до $m - 1$. Если m не слишком велико, то ее можно упорядочить следующим образом.

1. Организуем m пустых очередей по одной для каждого целого числа от 0 до $m - 1$. Каждую такую очередь назовем *блоком*.[2]
2. Просмотрим последовательность $a_1, a_2, ..., a_n$ слева направо, помещая элемент a_i в очередь с номером a_i.
3. Выполним конкатенацию этих очередей (добавим содержимое $(i + 1)$-й очереди в конец i-й очереди) и получим в результате упорядоченную последовательность.

Поскольку любой элемент можно вставить в i-ю очередь за постоянное время, то n элементов можно вставить в очереди за время $O(n)$. Конкатенация m очередей требует времени $O(m)$. Если m имеет порядок $O(n)$, то этот алгоритм сортирует n целых чисел за время $O(n)$. Назовем его *блочной сортировкой*.

Блочную сортировку можно обобщить так, чтобы она упорядочивала последовательность кортежей (т.е. списков) целых чисел в лексикографическом порядке. Пусть $\leq$ — линейный порядок на множестве S. *Лексикографическим порядком* называется такое расширение отношения $\leq$ на кортежи элементов из S, при котором $(s_1, s_2, \ldots, s_p) \leq (t_1, t_2, \ldots, t_q)$ означает, что выполнено одно из условий.

---

[2] В качестве синонимов перевода слова "bucket" используются также синонимы карман и корзина. Соответственно, этот алгоритм сортировки называют и корзинной, и карманной сортировкой. — *Примеч. ред.*

1. Существует такое целое число j, что $s_j < t_j$ и для всех $i < j$ выполняется условие $s_i = t_j$.
2. $p \le q$ и $s_i = t_i$ при $1 \le i \le p$.

Например, если рассматривать строки как кортежи из букв с естественным алфавитным порядком, то слова в словаре расположены в лексикографическом порядке.

Сначала обобщим блочную сортировку на последовательности, состоящие из k-элементных кортежей, элементами которых являются целые числа от 0 до $m - 1$. Сортировка осуществляется с помощью k обходов данной последовательности, в каждом из которых применяется блочная сортировка. Во время первого обхода кортежи упорядочиваются по их k-м элементам. Во время второго обхода полученная последовательность упорядочивается по $(k - 1)$-м элементам. При третьем обходе последовательность, полученная после второго обхода, упорядочивается по $(k - 2)$-м элементам и т.д. При k-м (последнем) обходе последовательность, полученная после $(k - 1)$-го обхода, упорядочивается по первым элементам.[3] Теперь элементы последовательности расположены в лексикографическом порядке. Дадим точное описание этого алгоритма.

Алгоритм 3.1. Лексикографическая сортировка

Вход. Последовательность $A_1, A_2, ..., A_n$, где A_i — k-элементный кортеж

$$(a_{i1}, a_{i2}, ..., a_{ik}),$$

в котором a_{ij} — целое число от 0 до $m - 1$. (Удобной структурой данных для этой последовательности кортежей является массив размером $n \times k$.)

Выход. Последовательность $B_1, B_2, ..., B_n$, представляющая собой такую перестановку $A_1, A_2, ..., A_n$, что $B_i \le B_j$ при $1 \le i < n$.

Метод. Чтобы поместить k-элементный кортеж A_i в некоторый блок, достаточно передвинуть указатель на A_i. Поэтому кортеж A_i можно добавить в блок за фиксированное время, а не только за время, ограниченное числом k. Для хранения "текущей" последовательности элементов применяется очередь с именем `QUEUE`. Используется также массив `Q`, состоящий из m блоков, в котором блок `Q`[i] предназначен для хранения тех k-элементных кортежей, у которых число i стоит в элементе, рассматриваемом в данный момент. Алгоритм приведен на рис. 3.1. □

[3] Во многих практических ситуациях достаточно применить блочную сортировку только по нескольким первым элементам. Если число элементов, попавших в каждый блок, невелико, то можно упорядочить слова в каждом блоке с помощью какого-нибудь прямолинейного алгоритма сортировки, например сортировка методом пузырька.

```
begin
    поместить A_1, A_2, ..., A_n в список QUEUE:
    for j ← k step -1 until 1 do
        begin
            for l ← 0 until m - 1 do опустошить Q[l];
            while список QUEUE не пустой do
                begin
                    пусть A_i — первый элемент в списке QUEUE:
                    переместить A_i из списка QUEUE в блок Q[a_ij]
                end;
            for l ← 0 until m - 1 do
                конкатенировать блок Q[l] с концом списка QUEUE
        end
end
```

Рис. 3.1. Алгоритм лексикографической сортировки

Теорема 3.1. Алгоритм 3.1 лексикографически упорядочивает последовательность n k-элементных кортежей, каждый элемент которых является целым числом от 0 и $m - 1$, за время $O((m + n)k)$.

Доказательство. Доказательство корректности алгоритма 3.1 проводится индукцией по количеству выполнений внешнего цикла. Индуктивное предположение заключается в том, что после r выполнений этого цикла кортежи в списке QUEUE будут расположены в лексикографическом порядке по их r последним элементам. Искомый результат легко получить, если заметить, что $(r + 1)$-е выполнение внешнего цикла упорядочивает множество кортежей по их $(r + 1)$-му с конца элементу, а в ситуации, когда два кортежа оказались в одном и том же блоке, первый из них предшествует второму в лексикографическом порядке, определяемом r последними элементами.

Временная сложность внешнего цикла алгоритма 3.1 имеет порядок $O(m + n)$. Цикл повторяется k раз, и это дает временную сложность $O((m + n)k)$. □

Алгоритм 3.1 имеет множество приложений. Долгое время он применялся в устройствах, сортирующих перфокарты. Кроме того, его можно использовать для упорядочения множества из $O(n)$ целых чисел, лежащих между 0 и $n^k - 1$, за время $O(kn)$, поскольку любое такое число можно считать k-элементным кортежем цифр от 0 до $n - 1$ (в системе счисления с основанием n).

Нашим последним обобщением блочной сортировки будет ее распространение на кортежи неодинаковой длины, которые мы будем называть строками. Если самая длинная строка имеет длину k, то каждую строку можно дополнить специальным символом до длины k и затем применить алгоритм 3.1. Однако, если

длинных строк мало, такой подход по двум причинам окажется неоправданно неэффективным. Во-первых, при каждом обходе просматривается каждая строка, а во-вторых, каждый блок $Q[i]$ анализируется даже в том случае, если почти все блоки пусты. Мы опишем алгоритм, сортирующий последовательность строк различной длины, элементами которых являются числа от 0 до $m - 1$. Время работы этого алгоритма на такой последовательности равно $O(m + l_{\text{total}})$, где l_i — длина i-й строки и $l_{\text{total}} = \sum_{i=1}^{n} l_i$. Этот алгоритм полезен в ситуациях, когда числа m и l_{total} имеют одинаковый порядок $O(n)$.

Суть этого алгоритма заключается в том, что сначала он упорядочивает строки в порядке убывания их длин. Пусть $l_{\max}$ — длина самой длинной строки. Тогда, как и в алгоритме 3.1, блочная сортировка применяется $l_{\max}$ раз. Однако при первом обходе он сортирует лишь строки длины $l_{\max}$ (по самому правому элементу). При втором обходе он сортирует те строки, длина которых не менее $l_{\max} - 1$ (по $(l_{\max} - 1)$-му элементу), и т.д.

Допустим, нам нужно упорядочить последовательность из трех строк: *bab*, *abc* и *a*. (Хотя мы считаем элементами кортежей целые числа, для удобства записи часто будут использоваться буквы. Это не вызывает трудностей, потому что при необходимости всегда можно заменить *a*, *b* и *c* на 0, 1 и 2.) В этом примере $l_{\max} = 3$, так что при первом обходе последовательности сортируются только первые две строки по третьему элементу. При этом строка *bab* помещается в *b*-блок, *abc* — в *c*-блок, *a*-блок остается пустым. При втором обходе эти же две строки сортируются по второму элементу. Теперь *a*-блок и *b*-блок становятся занятыми, а *c*-блок — пустым. При третьем (последнем) обходе сортируются все три строки по первому элементу. На этот раз *a*-блок и *b*-блок оказываются занятыми, а *c*-блок— пустым.

Заметим, что в общем случае многие блоки при обходе могут оказаться пустыми. Следовательно, целесообразно предусмотреть предварительный этап, чтобы выяснить, какие блоки при данном обходе будут непустыми. Список непустых блоков для каждого обхода формируется в порядке возрастания номеров блоков. Это позволяет выполнить конкатенацию непустых блоков за время, пропорциональное их количеству.

Алгоритм 3.2. Лексикографическая сортировка строк разной длины.

Вход. Последовательность строк (кортежей) $A_1, A_2, ..., A_n$, элементы которых представлены целыми числами от 0 до $m-1$. Пусть l_i — длина строки $A_i = (a_{i1}, a_{i2}, ..., a_{il_i})$ и $l_{\max}$ — наибольшее из чисел l_i.

Выход. Перестановка $B_1, B_2, ..., B_n$ строк A_i, такая, что $B_1 \le B_2 \le ... \le B_n$.

Метод.

1. Начнем с формирования списков (по одному для каждого $1 \le l \le l_{max}$) тех символов, которые появляются в l-м элементе одной или более строк. Для этого сначала создадим пары (l, a_{il}) для каждого элемента a_{il} каждой строки A_i, где $1 \le i \le n$, $1 \le l \le l_i$. Такая пара указывает, что в l-м элементе некоторой строки стоит число a_{il}. Множество этих пар лексикографически упорядочивается с помощью очевидного обобщения алгоритма 3.1.[4] Затем, просматривая упорядоченный список слева направо, формируем упорядоченные списки NONEMPTY[l] для $1 \le l \le l_{max}$, такие, что NONEMPTY[l] содержит только те символы, которые появляются в l-м элементе некоторой строки. Иными словами, список NONEMPTY[l] содержит в упорядоченном виде все целые числа j, для которых $a_{il} = j$ при некотором i.

2. Определяем длину каждой строки. Затем формируем списки LENGTH[l] для $1 \le l \le l_{max}$, такие, что список LENGTH[l] содержит все строки длины l. (Говоря о перемещении строк, мы имеем в виду перемещение указателей, ссылающихся на них. Следовательно, каждую строку можно добавить в список LENGTH[l] за фиксированное число шагов.)

3. Теперь строки сортируются по элементам, как в алгоритме 3.1, начиная с элемента с номером l_{max}. Однако после i-го обхода строк список QUEUE содержит только те из них, длина которых не менее $l_{max} - i + 1$, и они уже упорядочены в соответствии с элементами, имеющими номера от $l_{max} - i + 1$ до l_{max}. Списки типа NONEMPTY, сформированные на шаге 1, помогают определить непустые блоки при каждом применении блочной сортировки. Эта информация используется для того, чтобы ускорить конкатенацию блоков. Соответствующая часть алгоритма на псевдо-Алголе приведена на рис. 3.2. □

Пример 3.1. Упорядочим последовательность строк *a*, *bab* и *abc* с помощью алгоритма 3.2. Эти строки можно представить в виде структуры данных, изображенной на рис. 3.3. STRING — это массив, в котором элемент STRING[i] содержит указатель на представление i-й строки, у которой длина и элементы хранятся в массиве DATA. Ячейка в массиве DATA, на которую ссылается указатель из элемента STRING[i], содержит число j символов в i-й строке. Эти символы содержат следующие j ячеек массива DATA.

[4] В алгоритме 3.1 предполагалось, что все элементы выбираются из одного и того же алфавита. Здесь же второй элемент принимает значения от 0 до $m - 1$, а первый — от 1 до l_{max}.

```
    begin
1.      опустошаем список QUEUE:
2.      for j ← 0 until m - 1 do опустошить Q[j]:
3.      for l ← l_max step -1 until 1 do
            begin
4.              конкатенировать LENGTH[l] с началом списка QUEUE;[5]
5.              while список QUEUE не пустой do
                    begin
6.                      пусть A_i — первая строка в списке QUEUE;
7.                      переместить A_i из списка QUEUE в блок Q[a_il]
                    end;
8.              for each j в NONEMPTY[l] do
                    begin
9.                      конкатенировать Q[j] с концом списка QUEUE;
10.                     опустошить блок Q[j]
                    end
            end
    end
```

Рис. 3.2. Лексикографическая сортировка строк разной длины

Списки строк, используемые алгоритмом 3.2, в действительности являются списками указателей того же типа, что и в массиве STRING. В остальной части этого примера мы будем для удобства говорить о строках, а не об указателях на них. Однако необходимо помнить, что в очередях хранятся именно указатели, а не сами строки.

В части 1 алгоритма 3.2 формируется пара (1, *a*) из первой строки, пары (1, *b*), (2, *a*), (3, *b*) из второй и (1, *a*), (2, *b*), (3, c) — из третьей. Упорядоченный список этих пар выглядит следующим образом:

$$(1, a)\ (1, a)\ (\mathrm{l}, b)\ (2, a)\ (2, b)\ (3, b)\ (3, c)$$

Обходя его слева направо, приходим к выводу, что

$$\mathrm{NONEMPTY}[1] = a, b$$

$$\mathrm{NONEMPTY}[2] = a, b$$

$$\mathrm{NONEMPTY}[3] = b, c$$

[5] С формальной точки зрения конкатенацию можно выполнить только в конце очереди, но конкатенация в начале не должна вызывать никаких трудностей. Эффективнее было бы сначала выбрать строки A_i из списка LENGTH[*l*] в строке 6, а затем из списка очереди QUEUE и совсем не выполнять конкатенации списков LENGTH[*l*] и QUEUE.

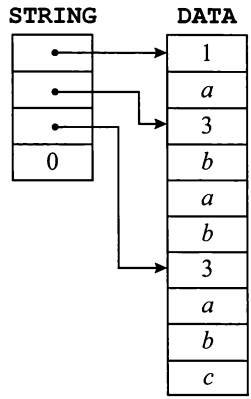


Рис. 3.3. Структура данных для строк

В части 2 алгоритма 3.2 мы вычисляем $l_1 = 1$, $l_2 = 3$ и $l_3 = 3$. Следовательно, LENGTH[1] = *a*, список LENGTH[2] пуст и LENGTH[3] = *bab*, *abc*. Поэтому часть 3 начинаем с того, что полагаем QUEUE = *bab*, *abc*, а затем упорядочиваем эти строки по их третьему элементу. Равенство NONEMPTY[3] = *b*, *c* позволяет не выполнять конкатенацию блока $Q[a]$ и конца списка QUEUE при построении упорядоченного списка в соответствии со строками 8–10, показанными на рис. 3.2. Таким образом, после первого прохода цикла в строках 3–10, показанных на рис. 3.2, выполняется условие QUEUE = *bab*, *abc*.

При втором проходе список QUEUE не меняется, так как список LENGTH[2] пуст, а упорядочение по второму элементу не изменяет порядок. При третьем обходе мы, согласно строке 4, получаем QUEUE = *a*, *bab*, *abc*. Упорядочение по первому элементу создает список QUEUE = *a*, *abc*, *bab*. Итак, мы получили правильный порядок. Заметим, что при третьем проходе список $Q[c]$ становится пустым, а поскольку *c* не входит в список NONEMPTY[1], нет необходимости выполнять конкатенацию $Q[c]$ и конца списка QUEUE. □

Теорема 3.2. Алгоритм 3.2 упорядочивает входную последовательность за время $O(l_{\text{total}} + m)$, где

$$l_{\text{total}} = \sum_{i=1}^{n} l_i.$$

Доказательство. Простая индукция по числу проходов внешнего цикла в программе, приведенной на рис. 3.2, показывает, что после *i* проходов список QUEUE содержит строки, длина которых не меньше $l_{\max} - i + 1$, расположенные в соответствии с их элементами с номерами от $l_{\max} - i + 1$ до $l_{\max}$. Следовательно, рассматриваемый алгоритм лексикографически упорядочивает входную последовательность.

Что касается временной сложности, то в части 1 затрачивается время порядка $O(l_{\text{total}})$ на образование списка пар и $O(m_{\text{total}}+l_{\text{total}})$ на его упорядочение. Аналогично временная сложность части 2 имеет порядок $O(l_{\text{total}})$ времени.

Теперь исследуем часть 3 и программу, представленную на рис. 3.2. Обозначим через n_i — количество строк, имеющих i-й элемент, и m_i — количество различных символов, появляющихся в i-х элементах строк (т.е. m_i — длина списка NONEMPTY[i]).

Рассмотрим фиксированное значение l в строке 3 на рис. 3.2. Цикл в строках 5–7 занимает $O(n_l)$ времени, а в строках 8–10 — $O(m_l)$ времени. Шаг 4 выполняется за постоянное время, так что один проход цикла в строках 3–10 занимает $O(m_l + n_l)$ времени. Следовательно, временная сложность всего цикла составляет

$$O\left(\sum_{l=1}^{l_{max}}(m_l + n_l)\right).$$

Поскольку

$$\sum_{l=1}^{l_{\max}} m_l \le l_{\text{total}} \text{ и } \sum_{l=1}^{l_{\max}} n_l = l_{\text{total}},$$

мы видим, что строки 3–10 выполняются за время $O(l_{\text{total}})$. Теперь, учитывая, что на строку 1 затрачивается постоянное время, а на строку 2 — время порядка $O(m)$, получаем требуемый результат. □

Рассмотрим пример, когда упорядочение является частью другого алгоритма.

Пример 3.2. Два дерева называются *изоморфными*, если одно из них можно отобразить на другое, изменив порядок сыновей его вершин. Рассмотрим задачу распознавания изоморфизма двух деревьев T_1 и T_2. Следующий алгоритм делает это за время, пропорциональное числу вершин. Этот алгоритм присваивает целые числа вершинам двух данных деревьев, начиная с вершин уровня 0 и двигаясь вверх до корней, так что деревья изоморфны тогда и только тогда, когда их корням присвоено одно и то же число. Алгоритм работает следующим образом.

1. Приписываем всем листьям деревьев T_1 и T_2 целое число 0.
2. По индукции предположим, что всем вершинам деревьев T_1 и T_2 на уровне $i-1$ уже приписаны целые числа. Пусть L_1 — список вершин .уровня $i-1$ в дереве T_1, расположенных в порядке неубывания присвоенных им чисел, а L_2 — соответствующий список для дерева T_2.[6]
3. Присваиваем внутренним вершинам уровня i в дереве T_1 кортеж целых чисел следующим образом: просматриваем список L_1 слева направо и для ка-

[6]Убедитесь, что временная сложность присваивания чисел вершинам при прямом обходе дерева, имеет порядок $O(n)$.

ждой вершины v из L_1 берем присвоенное ей число в качестве очередного элемента кортежа, который ставится в соответствии отцу вершины v. После выполнения этого шага каждой внутренней вершине w уровня i в дереве T_1 будет поставлен в соответствие кортеж $(i_1, i_2, ..., i_k)$, где $i_1, i_2, ..., i_k$ — целые числа, присвоенные сыновьям вершины w и расположенные в порядке неубывания. Обозначим через S_1 последовательность кортежей, построенных для вершин уровня i в дереве T_1.

4. Повторяем шаг 3 для дерева T_2. Пусть S_2 — последовательность кортежей, построенных для вершин уровня i в дереве T_2.
5. Упорядочиваем последовательности S_1 и S_2 с помощью алгоритма 3.2. Пусть соответственно S_1' и S_2' — упорядоченные последовательности кортежей.
6. Если последовательности S_1' и S_2' не совпадают, то останавливаемся, делая вывод, что исходные деревья не изоморфны. В противном случае присваиваем число 1 тем вершинам уровня i в дереве T_1, которые представлены первым отличающимся кортежем в последовательности S_1', число 2 — второму отличающемуся кортежем и т.д. Так как эти целые числа присвоены вершинам уровня i в дереве T_1, создаем список L_1 из вершин, которым присвоены числа. Добавляем в начало списка L_1 все листья дерева T_1, расположенные на уровне i. Пусть L_2 — соответствующий список вершин дерева T_2. Эти два списка теперь можно использовать для присваивания кортежей вершинам уровня $i + 1$, возвращаясь на шаг 3.
7. Если корням деревьев T_1 и T_2 присвоено одно и то же число, то T_1 и T_2 изоморфны. □

Процесс присваивания чисел и кортежей узлам двух изоморфных деревьев продемонстрирован на рис. 3.4.

Теорема 3.3. Изоморфизм двух деревьев с n вершинами можно определить за время $O(n)$.

Доказательство. Теорема следует из формализации алгоритма, изложенного в примере 3.2. Доказательство корректности алгоритма мы не приводим. Время работы можно оценить, если заметить, что присваивание целых чисел внутренним вершинам уровня i занимает время, пропорциональное числу вершин уровня $i - 1$. Временная сложность суммирования по всем уровням имеет порядок $O(n)$. Время присваивания чисел листьям также пропорционально n. Следовательно, временная сложность всего алгоритма составляет $O(n)$. □

Помеченным называется дерево, в котором вершинам присвоены метки. Предположим, что метками вершин являются целые числа от 1 до n. Тогда изо-

морфизм двух помеченных деревьев можно определить за линейное время, если включить метку каждой вершины в качестве первого элемента кортежа, присвоенного этой вершине. Таким образом, справедливо следующее утверждение.

Следствие. Определение изоморфизма двух размеченных деревьев с n вершинами, метками которых служат целые числа от 1 до n, имеет временную сложность $O(n)$.

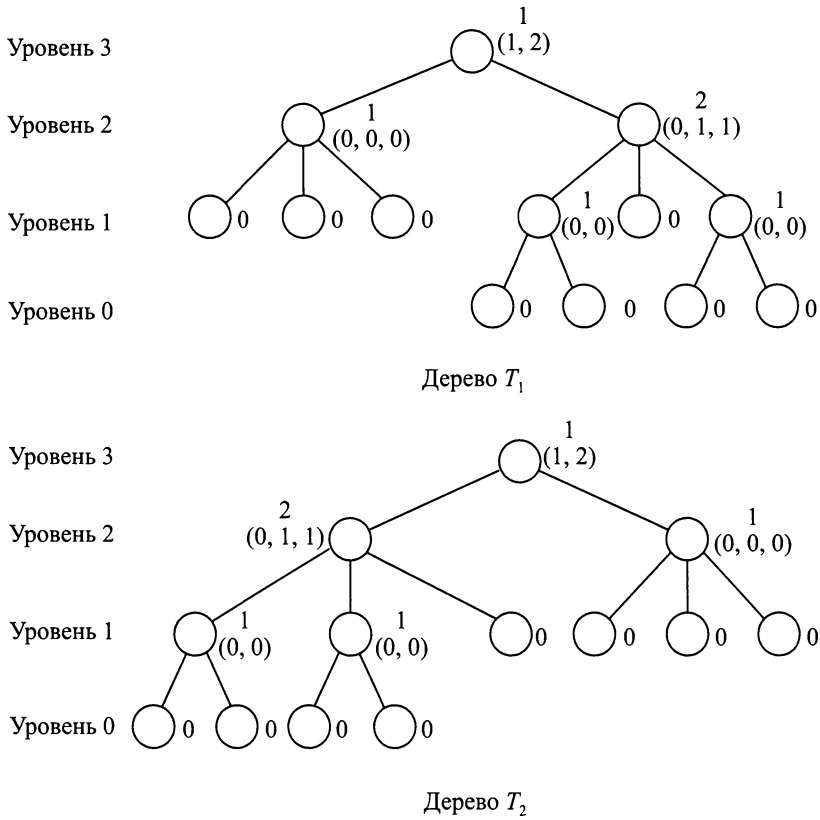


Рис. 3.4. Числа, присвоенные алгоритмом определения изоморфизма деревьев

3.3. Сортировка с помощью сравнений

В этом разделе мы рассмотрим задачу упорядочения последовательности, состоящей из n элементов, взятых из линейно упорядоченного множества S, о структуре которых ничего не известно. Информацию об этой последовательности мож-

но получить только путем сравнения двух элементов. Сначала мы покажем, что любой алгоритм, упорядочивающий последовательность элементов с помощью сравнений, выполняет, по крайней мере, $O(n\log n)$ сравнений на последовательности длины n.

Предположим, необходимо упорядочить последовательность, состоящую из n различных элементов a_1, a_2, ..., a_n. Алгоритм сортировки с помощью сравнений можно представить в виде дерева решений так, как описано в разделе 1.5. На рис. 1.18 показано дерево решений, упорядочивающее последовательность a, b, c. Далее мы предполагаем, что если элемент a сравнивается с элементом b в некотором узле v дерева решений, то необходимо перейти к левому сыну вершины v, если $a < b$, и к правому — если $a \geq b$.

Как правило, алгоритмы сортировки, в которых для разветвления используются сравнения, ограничиваются сравнением двух входных элементов за один раз. В самом деле, алгоритм, который работает на произвольном линейно упорядоченном множестве, не может преобразовывать входные данные, поскольку в общей постановке задачи операции над данными не имеют смысла. Так или иначе, мы докажем сильный результат о высоте любого дерева решений, упорядочивающего последовательность, состоящую из n элементов.

Лемма 3.1. Бинарное дерево высоты h содержит не более 2^h листьев.

Доказательство. Применим элементарную индукцию по h. Достаточно заметить, что бинарное дерево высоты h состоит из корня и самое большее двух поддеревьев, каждое из которых имеет высоту не более $h - 1$. □

Теорема 3.4. Высота любого дерева решений, упорядочивающего последовательность, состоящую из n различных элементов, не меньше $\log n!$.

Доказательство. Поскольку результатом упорядочения последовательности, состоящей из n элементов, может быть любая из $n!$ перестановок элементов входной последовательности, то в дереве решений должно быть, по крайней мере, $n!$ листьев. По лемме 3.1 высота такого дерева должна быть не меньше $\log n!$. □

Следствие. В любом алгоритме сортировки с помощью сравнений на упорядочение последовательности, состоящей из n элементов, выполняется не меньше $cn\log n$ сравнений при некотором $c > 0$ и достаточно большом n.

Доказательство. Заметим, что при $n > 1$

$$n! \geq n(n-1)(n-2)\ldots\left(\left\lceil\frac{n}{2}\right\rceil\right) \geq \left(\frac{n}{2}\right)^{n/2},$$

так что $\log n! \geq (n/2)\log(n/2) \geq (n/4)\log n$ при $n \geq 4$. □

По формуле Стирлинга можно получить более точное приближение факториала $n!$ с помощью функции $(n/e)^n$, так что выражение $n(\log n - \log e) = n\log n - 1{,}44n$ является хорошим приближением нижней границы числа сравнений, необходимых для упорядочения последовательности из n элементов.

3.4. Пирамидальная сортировка с помощью $O(n \log n)$ сравнений

Поскольку любой алгоритм сортировки с помощью сравнений затрачивает $n\log n$ сравнений для упорядочения хотя бы одной последовательности длины n, возникает вопрос: существуют ли алгоритмы сортировки, затрачивающие не более $O(n\log n)$ сравнений для упорядочения любой последовательности длины n? Один такой алгоритм мы уже видели — это сортировка слиянием из раздела 2.7. Другой алгоритм — пирамидальная сортировка. Помимо того, что это полезный алгоритм сортировки, в нем используется интересная структура данных, которая имеет и другие приложения.

Алгоритм пирамидальной сортировки лучше всего понять в терминах бинарного дерева (рис. 3.5), у которого каждый лист имеет глубину d или $d-1$. Вершины дерева помечаются элементами упорядочиваемой последовательности. Алгоритм пирамидальной сортировки меняет размещение этих элементов на дереве до тех пор, пока элемент, соответствующий произвольной вершине, станет не меньше элементов, соответствующих его сыновьям. Такое размеченное дерево мы будем называть *кучей* (heap).

Пример 3.3. Куча изображена на рис. 3.5. Заметим, что последовательность элементов, лежащих на пути из любого листа в корень, линейно упорядочена и наибольший элемент в поддереве всегда соответствует его корню. □

На следующем шаге алгоритма пирамидальной сортировки из кучи удаляется наибольший элемент — он соответствует корню дерева. Метка одного из листов переносится в корень, а сам лист удаляется. Затем полученное дерево преобразовывается в кучу, и процесс повторяется. Последовательность элементов, удаленных из кучи, упорядочена по невозрастанию.

Удобной структурой данных для кучи служит такой массив A, что $A[1]$ —элемент в корне; $A[2i]$ и $A[2i+1]$ — элементы в левом и правом сыновьях (если они существуют) той вершины, в которой хранится $A[i]$. Например, кучу на рис. 3.5 можно представить массивом

16	11	9	10	5	6	8	1	2	4

Заметим, что вершины наименьшей глубины стоят в этом массиве первыми, а вершины равной глубины — в порядке слева направо.

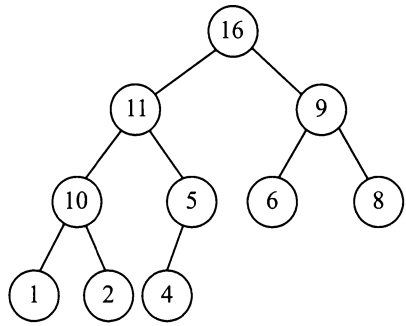


Рис. 3.5. Куча

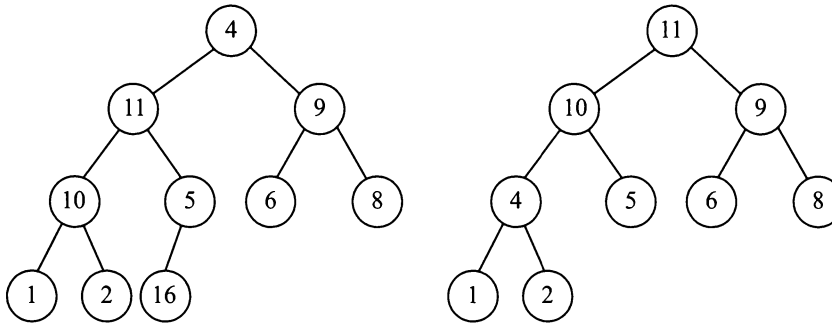
Но не каждую кучу можно представить таким способом. На языке представления деревьев можно сказать, что для образования такого массива требуется, чтобы листья самого низкого уровня было расположены как можно левее (см. рис. 3.5).

Если куча представляется описанным массивом, то некоторые операции из алгоритма пирамидальной сортировки выполнить нетрудно. Например, следуя алгоритму, необходимо удалить элемент из корня, запомнить его, переделать оставшееся дерево в кучу и удалить непомеченный лист. Можно удалить наибольший элемент из кучи и запомнить его, поменяв местами $A[1]$ и $A[n]$, и затем не считать более ячейку n нашего массива частью кучи. Ячейка n рассматривается как лист, удаленный из этого дерева. Для того чтобы переделать дерево, хранящееся в ячейках 1, 2, ... , $n - 1$, в кучу, необходимо взять новый элемент $A[1]$ и провести его вдоль подходящего пути в дереве. Затем можно повторить процесс, меняя местами $A[1]$ и $A[n - 1]$ и считая, что дерево занимает ячейки 1, 2, ..., $n - 2$ и т.д.

Пример 3.4. Рассмотрим на примере кучи, показанной на рис. 3.5, что происходит, когда мы поменяем местами первый и последний элементы массива, представляющего эту кучу. Новый массив

4	11	9	10	5	6	8	1	2	16

соответствует помеченному дереву на рис. 3.6, *а*. Элемент 16 исключается из дальнейшего рассмотрения. Для того чтобы превратить полученное дерево в кучу, необходимо поменять местами элемент 4 с бо́льшим из его сыновей, т.е. с элементом 11.



а) б)

Рис. 3.6. Результат перестановки элементов 4 и 16 в куче из рис. 3.5 (*а*); результат реконструкции кучи и исключения элемента 16 (*б*)

В своей новой позиции элемент 4 имеет сыновей 10 и 5. Так как они больше 4, то 4 меняется местами с 10, бо́льшим сыном. После этого сыновьями элемента 4 в новой позиции становятся 1 и 2. Поскольку 4 превосходит их обоих, дальнейшие перестановки не нужны.

Полученная в результате куча показано на рис. 3.6, *б*. Заметим, что хотя элемент 16 был удален из кучи, он все же присутствует в конце массива A. □

Перейдем к формальному описанию алгоритма пирамидальной сортировки.

Пусть a_1, a_2, ..., a_n — последовательность сортируемых элементов. Предположим, что вначале они помещаются в массив A именно в этом порядке, т.е. $A[i] = a_i$, $1 \le i \le n$. Первый шаг состоит в построении кучи, т.е. элементы в массиве A перераспределяются так, чтобы удовлетворялось свойство кучи: $A[i] \ge A[2i]$ при $1 \le i \le n/2$ и $A[i] \ge A[2i + 1]$ при $1 \le i < n/2$. Этот процесс начинается с листьев и создает все бо́льшие и бо́льшие кучи. Всякое поддерево, состоящее из листа, уже является кучей. Чтобы сделать поддерево высоты h кучей, необходимо поменять местами элемент в корне и наибольший из элементов, соответствующих его сыновьям, если, конечно, он меньше какого-то из них. Такое действие может испортить кучу высоты $h - 1$, и тогда ее необходимо снова перестроить. Приведем точное описание этого алгоритма.

Алгоритм 3.3. Построение кучи

Вход. Массив элементов $A[i]$, $1 \le i \le n$.

Выход. Элементы массива A, организованные в виде кучи, т.е. $A[i] \le A[\lfloor i/2 \rfloor]$ для $1 < i \le n$.

Метод. В основе алгоритма лежит рекурсивная процедура HEAPIFY. Ее параметры i и j задают область ячеек массива A, обладающую свойством кучи, а корень создаваемого дерева помещается в ячейку i.

```
      procedure HEAPIFY(i, j):
1.    if i — не лист, и если сын вершины i
          содержит больший элемент, чем вершина i then
              begin
2.                пусть k — сын вершины i,
                  содержащий наибольший элемент;
3.                меняем местами A[i] и A[k];
4.                HEAPIFY(k, j)
              end
```

Параметр j используется, чтобы определить, является ли i листом и имеет ли одного или двух сыновей. Если $i > j/2$, то i — лист, и процедуре HEAPIFY(i, j) ничего не нужно делать, поскольку $A[i]$ — и так куча.

Алгоритм, превращающий весь массив A в кучу, довольно простой.

```
procedure BUILDHEAP:
for i ← n[7] step -1 until 1 do HEAPIFY(i, n)   □
```

Покажем, что алгоритм 3.3 преобразует A в кучу за линейное время.

Лемма 3.2. Если вершины $i + 1$, $i + 2$, ... , n являются корнями куч, то после вызова процедуры HEAPIFY(i, n) все вершины i, $i + 1$, ..., n будут корнями куч.

Доказательство. Доказательство проводится обратной индукцией по i. База, т.е. случай $i = n$, представляет собой тривиальный случай, так как вершина n должна быть листом и условие, проверяемое в строке 1, гарантирует, что процедура HEAPIFY(n, n) ничего не делает.

Выполняя шаг индукции заметим, что если вершина i является листом или не имеет сына с бо́льшим элементом, то доказывать нечего (как и в случае базы). Однако если у вершины i есть один сын (т. е. если $2i = n$) и $A[i] < A[2i]$, то строка 3 процедуры HEAPIFY меняет местами $A[i]$ и $A[2i]$. В строке 4 вызывается HEAPIFY($2i$, n), поэтому из предположения индукции вытекает, что дерево с корнем $2i$ будет преобразовано в кучу. При этом вершины $i + 1$, $i + 2$, ..., $2i - 1$ остаются корнями куч. Так как после этой новой перестановки в массиве A выполняется неравенство $A[i] > A[2i]$, то дерево с корнем i также оказывается кучей.

Аналогично, если вершина i имеет двух сыновей (т.е. если $2i + 1 \leq n$) и наибольший из элементов $A[2i]$ и A[$2i + 1$] больше элемента $A[i]$, то, рассуждая аналогично

[7] На практике следует начать с $\lfloor n/2 \rfloor$

предыдущему случаю, можно показать, что после вызова процедуры HEAPIFY(i, n) все вершины i, $i + 1$, ..., n будут корнями куч. □

Теорема 3.5. Алгоритм 3.3 преобразует дерево A в кучу за линейное время.

Доказательство. Применяя лемму 3.2, можно с помощью простой обратной индукции по i показать, что вершина i становится корнем кучи для всех i, $1 \le i \le n$.

Пусть $T(h)$ — время выполнения процедуры HEAPIFY на вершине высоты h. Тогда $T(h) \le T(h - 1) + c$, где c — константа. Отсюда вытекает, что $T(h)$ имеет порядок $O(h)$.

Если пренебречь рекурсивными вызовами, то алгоритм 3.3 вызывает процедуру HEAPIFY один раз для каждой вершины. Поэтому время, затрачиваемое на процедуру BUILDHEAP, имеет тот же порядок, что и сумма высот всех вершин. Однако количество вершин высоты i не превышает $\lceil n/2^{i+1} \rceil$. Следовательно, общее время, затрачиваемое процедурой BUILDHEAP, имеет порядок

$$\sum_{i=1}^{h} i \frac{n}{2^i},$$

т.е. $O(n)$. □

Теперь можно завершить подробное описание алгоритма пирамидальной сортировки. После того как элементы массива A преобразованы в кучу, некоторые элементы удаляются из корня по одному за раз. Это делается путем обмена местами $A[1]$ и $A[n]$ и последующего преобразования элементов $A[1]$, $A[2]$, ..., $A[n - 1]$ в кучу. Затем местами меняются элементы $A[1]$ и $A[n - 1]$, а в кучу преобразуются элементы $A[1]$, ..., $A[n - 2]$. Этот процесс продолжается до тех пор, пока в куче не останется один элемент. Тогда последовательность $A[1]$, $A[2]$, ..., $A[n]$ оказывается упорядоченной.

Алгоритм 3.4. Пирамидальная сортировка

Вход. Массив элементов $A[i]$, $1 \le i \le n$.

Выход. Элементы массива A, расположенные в порядке неубывания.

Метод. Применяется процедура BUILDHEAP, т.е. алгоритм 3.3. Алгоритм выглядит следующим образом:

```
begin
    BUILDHEAP;
    for i ← n step -1 until 2 do
        begin
            поменять местами A[1] и A[i];
            HEAPIFY(1, i-1)
        end
end □
```

Теорема 3.6. Алгоритм 3.4 упорядочивает последовательность, состоящую из n элементов, за время $O(n \log n)$.

Доказательство. Корректность алгоритма доказывается индукцией по числу проходов основного цикла, которое обозначим через m. Индуктивное предположение состоит в том, что после n итераций список $A[n-m+1], ..., A[n]$ содержит m наибольших элементов, расположенных в порядке неубывания, а список $A[1], ..., A[n-m]$ образует кучу. Детали доказательства оставляем читателям в качестве упражнения. Временная сложность процедуры HEAPIFY(1, i) имеет порядок $O(\log i)$. Следовательно, алгоритм 3.4 имеет сложность $O(n \log n)$. □

Следствие. Алгоритм пирамидальной сортировки имеет временную сложность $O_C(n \log n)$.

3.5. Быстрая сортировка с ожидаемой временной сложностью $O(n \log n)$

До сих пор мы рассматривали поведение алгоритмов сортировки только в худшем случае. Для многих приложений более реалистичной мерой временной сложности является ожидаемое время работы. В случае сортировки с помощью деревьев решений никакой алгоритм сортировки не может иметь ожидаемую временную сложность, существенно меньшую, чем $n \log n$. Однако известны алгоритмы сортировки, у которых временная сложность в худшем случае имеет порядок cn^2, где c — константа, но ожидаемое время работы является одной из лучших. Примером такого алгоритма является быстрая сортировка, рассматриваемая в этом разделе.

Чтобы рассуждать об ожидаемом времени работы алгоритма, мы должны договориться о вероятностном распределении входных данных. Для сортировки естественно допустить, что мы и сделаем, что любая перестановка упорядочиваемой последовательности равновероятна в качестве входных данных. При таком допущении можно оценить снизу ожидаемое число сравнений, необходимых для упорядочения последовательности, состоящей из n элементов.

Общий метод состоит в том, чтобы назначить каждому листу v дерева решений вероятность быть достигнутым при заданных входных данных. Зная распределение вероятностей на входных данных, можно найти вероятности, назначенные листьям. Таким образом, можно определить ожидаемое число сравнений, выполняемых данным алгоритмом сортировки, если вычислить сумму $\sum_i p_i d_i$, взятую по всем листьям дерева решений данного алгоритма, в которой p_i — вероятность достижения i-го листа, а d — его глубина. Это число называется *ожидаемой глубиной* дерева решений. Итак, мы пришли к следующему обобщению теоремы 3.4.

Теорема 3.7. Если все перестановки последовательности, состоящей из n элементов, появляются на входе с равными вероятностями, то любое дерево решений, упорядочивающее последовательность, состоящую из n элементов, имеет ожидаемую глубину не менее $\log n!$.

Доказательство. Обозначим через $D(T)$ сумму глубин листьев бинарного дерева T. Пусть $D(m)$ — ее наименьшее значение, взятое по всем бинарным деревьям T с m листьями. Покажем индукцией по m, что $D(m) \geq m \log m$.

База, т.е. случай $m = 1$, является тривиальным случаем. Допустим, индуктивное предположение верно для всех значений m, меньших k. Рассмотрим дерево решений T с k листьями. Оно состоит из корня с левым поддеревом T_i с i листьями и правым поддеревом T_{k-i} с $(k-i)$ листьями при некотором i, $1 \leq i < k$. Очевидно, что

$$D(T) = i + D(T_i) + (k-i) + D(T_{k-i}).$$

Следовательно, наименьшее значение $D(T)$ задается равенством

$$D(k) = \min_{1 \leq i \leq k} \left[k + D(i) + D(k-i) \right]. \tag{3.1}$$

Учитывая индуктивное предположение, получаем, что

$$D(k) \geq k + \min_{1 \leq i \leq k} \left[i \log i + (k-i) \log (k-i) \right]. \tag{3.2}$$

Легко показать, что этот минимум достигается при $i = k/2$. Следовательно,

$$D(k) \geq k + k \log \frac{k}{2} = k \log k.$$

Следовательно, $D(m) \geq m \log m$ для всех $m \geq 1$.

Теперь мы утверждаем, что дерево решений T, упорядочивающее n случайных элементов, имеет не меньше $n!$ листьев. Более того, в точности $n!$ листьев появляются с вероятностью $1/n!$ каждый, а вероятность остальных равна нулю. Из дерева T можно удалить все вершины, потомками которых являются только листья с нулевой вероятностью, не изменяя его ожидаемой глубины. Тогда останется дерево T', имеющее $n!$ листьев, каждый из которых достигается с вероятностью $1/n!$. Поскольку $D(T') \geq n! \log n!$, то ожидаемая глубина дерева T' (а значит, и T) не меньше $(1/n!)n! \log n! = \log n!$. □

Следствие. Любая сортировка с помощью сравнений выполняет в среднем не менее $cn \log n$ сравнений, где c — положительная константа.

Существует эффективный алгоритм, называемый быстрой сортировкой, ожидаемое время работы которого хотя и ограничено снизу функцией $cn \log n$, где c — константа (как и у любой сортировки с помощью сравнений), но меньше времени

работы других известных алгоритмов при их реализации на большинстве существующих машин. В худшем случае быстрая сортировка имеет квадратичное время работы, но для многих приложений это не существенно.

Алгоритм 3.5. Быстрая сортировка

Вход. Последовательность S, состоящая из n элементов $a_1, a_2, ..., a_n$.

Выход. Элементы последовательности S, расположенные по порядку.

Метод. Рекурсивная процедура QUICKSORT показана на рис. 3.7. Алгоритм заключается в вызове QUICKSORT(S). □

```
      procedure QUICKSORT(S):
1.        if S содержит не больше одного элемента then return S:
          else
              begin
2.                случайным образом выбрать произвольный
                      элемент a из S;
3.                пусть S1, S2 и S3 – последовательности элементов
                      из S, элементы которых соответственно меньше,
                      равны и больше a;
4.                return (QUICKSORT(S1), затем S2,
                      затем QUICKSORT(S3))
              end
```

Рис. 3.7. Программа быстрой сортировки

Теорема 3.8. Алгоритм 3.5 упорядочивает последовательность, состоящую из n элементов, за ожидаемое время $O(n \log n)$.

Доказательство. Корректность алгоритма 3.5 доказывается прямой индукцией по длине последовательности S. Чтобы упростить анализ времени работы, допустим, что все элементы в S не равны друг другу. Это предположение максимизирует размеры последовательностей S_1 и S_3, которые создаются в строке 3, и тем самым максимизирует ожидаемое время, затрачиваемое в рекурсивных вызовах в строке 4. Пусть $T(n)$ — ожидаемое время, затрачиваемое алгоритмом QUICKSORT на упорядочение последовательности, состоящей из n элементов. Ясно, что $T(0) = T(1) = b$, где b — константа.

Предположим, что элемент a, выбираемый в строке 2, является i-м наименьшим элементом среди n элементов последовательности S. Тогда на два рекурсивных вызова QUICKSORT в строке 4 затрачивается ожидаемое время $T(i - 1)$ и $T(n - i)$ соответственно. Поскольку параметр i равновероятно принимает любое

значение от 1 до n, а балансировка последовательности QUICKSORT(S) очевидно занимает время cn, где c — константа, получаем следующее соотношение:

$$T(n) \le cn + \frac{1}{n}\sum_{i=1}^{n}\left[T(i-1)+T(n-i)\right] \text{ при } n \ge 2. \tag{3.3}$$

Выполняя алгебраические преобразования в (3.3), получаем неравенство

$$T(n) \le cn + \frac{2}{n}\sum_{i=0}^{n-1}T(i). \tag{3.4}$$

Покажем, что при $n \ge 2$ выполняется неравенство $T(n) \le kn\ln n$, где $k = 2c + 2b$ и $b = T(0) = T(1)$. Для базы $n = 2$ неравенство $T(2) \le 2c + 2b$ непосредственно вытекает из (3.4). Для выполнения шага индукции запишем (3.4) в виде

$$T(n) \le cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1}ki\ln i. \tag{3.5}$$

Так как функция $i\ln i$ вогнута, легко показать, что

$$\sum_{i=2}^{n-1} i\ln i \le \int_{2}^{n} x\ln x dx \le \frac{n^2\ln n}{2} - \frac{n^2}{4}. \tag{3.6}$$

Подставляя (3.6) в (3.5), получаем

$$T(n) \le cn + \frac{4b}{n} + kn\ln n - \frac{kn}{2}. \tag{3.7}$$

Поскольку $n \ge 2$ и $k = 2c + 2b$, то $cn + 4b/n \le kn/2$. Таким образом, из (3.7) следует неравенство $T(n) \le kn\ln n$. □

Для практической реализации алгоритма быстрой сортировки необходимо учесть два фактора. Первый — это способ выбора элемента a, упоминаемого в строке 2 процедуры QUICKSORT. При реализации этой инструкции может возникнуть соблазн принять простое решение, например, всегда выбирать, первый элемент последовательности S. Такой выбор мог бы значительно ухудшить работу алгоритма QUICKSORT. Последовательность, к которой применяется подпрограмма сортировки, часто бывает уже упорядочена, так что первый элемент с вероятностью выше средней является небольшим. Нетрудно убедиться, что, в крайнем случае, когда процедура QUICKSORT начинает работу на уже упорядоченной последовательности без дубликатов, а в качестве элемента a, упоминаемого в строке 2, всегда выбирается первый элемент последовательности S, в последовательности S_3 всегда будет на один элемент меньше, чем в S. В этом случае QUICKSORT выполняет квадратичное число шагов.

При выборе разделяющего элемента a в строке 2 намного лучше использовать датчик случайных чисел, сгенерировать целое число i, $(1 \le i \le |S|)$,[8] а затем вы-

[8]Выражение $|S|$ обозначает длину последовательности S.

брать i-й элемент последовательности S. Еще проще извлечь выборку элементов из S, а затем взять в качестве разделяющего элемента ее медиану. Например, в качестве разделяющего элемента можно было бы взять медиану выборки, состоящей из первого, среднего и последнего элементов последовательности S.

Второй фактор — способ эффективного разбиения S на три последовательности: S_1, S_2 и S_3. Можно и нужно хранить в массиве A все n исходных элементов. Поскольку процедура `QUICKSORT` вызывается рекурсивно, ее аргумент S всегда будет находиться в последовательных элементах массива, скажем, $A[f]$, $A[f+1]$, ..., $A[l]$ для некоторых $1 \le f \le l \le n$. Выбрав случайный элемент a, можно выполнить разбиение последовательности S на этом же месте. Иначе говоря, можно записать последовательность S_1 в элементах $A[f]$, $A[f+1]$, ..., $A[k]$, а $S_2 \cup S_3$ — в элементах $A[k+1]$, $A[k+2]$, ..., $A[l]$ при некотором $k, f \le k \le l$. Затем можно, если нужно, расщепить $S_2 \cup S_3$, но обычно эффективнее просто рекурсивно применить процедуру `QUICKSORT` к последовательностям S_1 и $S_2 \cup S_3$, если оба этих множества не пустые.

По-видимому, легче всего разбить последовательность S на том же месте с помощью двух указателей на массив, назовем их i и j. Вначале $i = f$, и все время в $A[f]$, ..., $A[i-1]$ будут находиться элементы из последовательности S_1. Аналогично вначале $j = l$, а в элементах $A[j+1]$, ..., $A[l]$ будут находиться элементы последовательности $S_2 \cup S_3$. Это разбиение производит подпрограмма, приведенная на рис. 3.8.

```
     begin
1.       i ← f;
2.       j ← l;
3.       while i ≤ j do
             begin
4.               while A[j] ≥ a и j ≥ f do j ← j - 1;
5.               while A[i] < a и i ≤ l do i ← i + 1;
6.               if i < j then
                     begin
7.                       поменять местами A[i] и A[j];
8.                       i ← i + 1;
9.                       j ← j - 1
                     end
             end
     end
```

Рис. 3.8. Разбиение последовательностей S_1 и $S_2 \cup S_3$ на месте

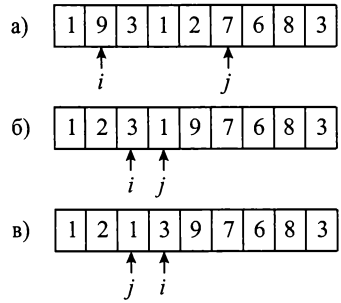


Рис. 3.9. Разбиение массива

Затем можно вызвать процедуру `QUICKSORT` для массива $A[f]$, ..., $A[i-1]$, т.е. S_1, и для массива $A[j+1]$, ..., $A[l]$, т.е. $S_2 \cup S_3$. Но если $i = f$ (и тогда $S_1 = \varnothing$), то необходимо сначала удалить из последовательности $S_2 \cup S_3$ хотя бы один элемент, равный a. Удобно удалять тот элемент, по которому производилось разбиение. Подчеркнем, что если это представление в виде массива применяется для последовательностей, то можно передать аргументы для процедуры `QUICKSORT`, просто установив указатели на первую и последнюю ячейку используемого фрагмента массива.

Пример 3.5. Разобьем массив A по элементу $a = 3$.

1	2	3	4	5	6	7	8	9
6	9	3	1	2	7	1	8	3

Инструкция **`while`** в строке 4 уменьшает j с 9 до 7, поскольку числа $A[9] = 3$ и $A[8] = 8$ оба не меньше a, но $A[7] = 1 < a$. Строка 5 не увеличивает i по сравнению с его начальным значением 1, поскольку $A[1] = 6 \geq a$. Таким образом, мы переставляем элементы $A[1]$ и $A[7]$, полагаем $i = 2, j = 6$ и получаем массив, показанный на рис. 3.9, *а*. Результаты, получаемые после следующих двух проходов цикла в строках 3–9, показаны на рис. 3.9, *б-в*. В этот момент $i > j$ и выполнение инструкции **`while`**, стоящей в строке 3, заканчивается. □

3.6. Порядковые статистики

Сортировка тесно связана с задачей нахождения k-го наименьшего элемента в последовательности, состоящей из n элементов.[9] Одно из очевидных решений со-

[9]Строго говоря, k-м наименьшим элементом последовательности $a_1, a_2, ..., a_n$ называется такой элемент b этой последовательности, что существует не более чем для $k-1$ значений i, при которых что $a_i < b$ и не менее чем для k значений i, при которых $a_i \leq b$. Например, 4 — второй и третий наименьший элемент последовательности 7, 4, 2, 4.

стоит в следующем: упорядочить эту последовательность в порядке неубывания ее элементов и найти k-й элемент. Как мы уже видели, это потребует $n \log n$ сравнений. Аккуратно применяя стратегию "разделяй и властвуй", можно найти k-й наименьший элемент за $O(n)$ шагов. Важный частный случай возникает, когда $k = \lceil n/2 \rceil$; в этом случае середина последовательности находится за линейное время.

Алгоритм 3.6. Нахождение k-го наименьшего элемента

Вход. Последовательность S, состоящая из n элементов, принадлежащих линейно упорядоченному множеству, и целое число k, $1 \le k \le n$.

Выход. k-й наименьший элемент последовательности S.

Метод. Применяется рекурсивная процедура SELECT, приведенная на рис. 3.10. □

```
      procedure SELECT(k, S):
1.    if |S| < 50 then
          begin
2.            упорядочить S:
3.            return k-й наименьший элемент в S
          end
      else
          begin
4.            разбить S на ⌊|S|/5⌋ последовательностей по 5 элементов
5.            за их пределами останется не более четырех элементов;
6.            упорядочить каждую 5-элементную последовательность;
7.            пусть M — последовательность медиан 5-элементных
              множеств;
8.            m ← SELECT (⌈|M|⌉/2, M);
9.            пусть S1, S2 и S3 — последовательности элементов из S,
              которые соответственно меньше, больше и равны m;
10.           if |S1| > k then return SELECT(k, S1)
              else
11.               if (|S1| + |S2| ≥ k) then return m
12.               else return SELECT(k - |S1| - |S2|, S3)
          end
```

Рис. 3.10. Алгоритм поиска k-го наименьшего элемента

Попробуем интуитивно понять, как работает алгоритм 3.6. Его основная идея заключается в том, что заданная последовательность разбивается по некоторому элементу m на такие три подпоследовательности S_1, S_2, S_3, что S_1 содержит все элементы, меньшие m, S_2 — все элементы, равные m, и S_3 — все элементы, большие m. Подсчитав число элементов в S_1 и S_2, можно узнать, в каком из множеств S_1, S_2 и S_3 лежит k-й наименьший элемент. Так можно свести данную задачу к задаче меньшего размера.

Чтобы получить алгоритм с линейной сложностью, необходимо уметь находить за линейное время разделяющий элемент так, чтобы длина каждой из подпоследовательностей S_1 и S_2 была не больше фиксированной доли длины последовательности S. Главная трудность заключается в способе выбора разделяющего элемента m. Последовательность S разбивается на подпоследовательности по пять элементов в каждой. Каждая подпоследовательность упорядочивается, после чего из медиан всех этих подпоследовательностей формируется последовательность M, которая содержит только $\lfloor n/5 \rfloor$ элементов, и поэтому ее медиану можно найти в пять раз быстрее, чем медиану последовательности из n элементов.

Далее, не меньше четверти всех элементов последовательности S не превосходят m и не меньше четверти ее элементов больше или равны m (рис. 3.11). Возникает вопрос: какую роль играет "магическое число" 5? Ответ состоит в том, что процедура `SELECT` рекурсивно вызывается два раза, каждый раз для последовательности, длина которой равна некоторой части длины последовательности S. Чтобы алгоритм работал линейное время, сумма длин этих двух последовательностей должна быть меньше $|S|$. Числа, кроме 5, также подходят, но для некоторых из них процесс упорядочения подпоследовательностей будет выполняться дольше. В качестве упражнения предлагаем выяснить, какие именно числа можно использовать вместо 5.

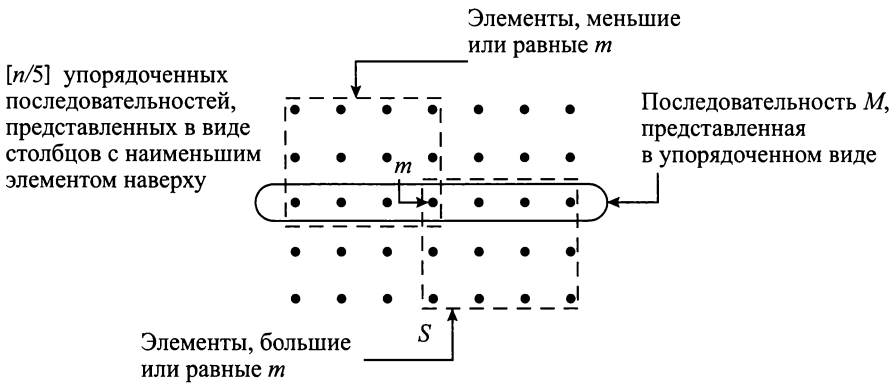


Рис. 3.11. Разбиение множества S по алгоритму 3.6.

Теорема 3.9. Алгоритм 3.6 находит k-й наименьший элемент в последовательности S, состоящей из n элементов, за время $O(n)$.

Доказательство. Корректность алгоритма доказывается непосредственно индукцией по $|S|$, и эту часть доказательства мы оставляем читателям в качестве упражнения. Пусть $T(n)$ — время, затрачиваемое на выбор k-го наименьшего элемента из последовательности, состоящей из n элементов. Длина последова-

тельности M, составленной из медиан подпоследовательностей, не превышает $n/5$, поэтому рекурсивный вызов процедуры SELECT($\lceil |M|/2 \rceil$, M) занимает время, не превышающее $T(n/5)$.

Длина каждой из последовательностей S_1 и S_3 не превышает $3n/4$. Чтобы убедиться в этом, заметим, что не менее $\lfloor n/10 \rfloor$ элементов последовательности M больше или равны m и для каждого из них в S найдутся два различных элемента, которые тоже больше или равны m. Следовательно, $|S_1| \le n - 3\lfloor n/10 \rfloor$, т.е. при $n \ge 50$ длина последовательности S_1 меньше $3n/4$. Аналогичное рассуждение применимо и к S_3. Таким образом, затраты времени на рекурсивные вызовы в строке 10 или 12 не превышают $T(3n/4)$. Все остальные инструкции занимают не более $O(n)$ времени. Таким образом, для некоторой константы c

$$T(n) \le \begin{cases} cn, & \text{если } n \le 49, \\ T(n/5) + T(3n/4) + cn, & \text{если } n \ge 50. \end{cases} \tag{3.8}$$

Из (3.8) по индукции следует, что $T(n) \le 20cn$. □

3.7. Ожидаемое время для порядковых статистик

В этом разделе мы изучим ожидаемое время, затрачиваемое на поиск k-го наименьшего элемента в последовательности, состоящей из n элементов. Мы увидим, что для идентификации k-го наименьшего элемента требуется выполнить по меньшей мере $n - 1$ сравнений как в худшем случае, так и в среднем. Следовательно, алгоритм, описанный в предыдущем разделе, с точностью до постоянного множителя оптимален в классе деревьев решений. Здесь мы приведем другой алгоритм, который имеет квадратичную сложность в худшем случае, но среднее время работы которого составляет лишь долю времени работы алгоритма 3.6.

Пусть $S = \{a_1, a_2, \ldots, a_n\}$ — множество, состоящее из n различных элементов, а T — дерево решений какого-нибудь алгоритма для поиска k-го наименьшего элемента в последовательности S. Каждый путь p в дереве T определяет такое отношение R_p на множестве S, что $a_i R_p a_j$, если два различных элемента a_i и a_j сравниваются в некоторой вершине, лежащей на пути p, и в результате этого сравнения выясняется, что либо $a_i < a_j$, либо $a_i \le a_j$.[10] Пусть R_p^+ — транзитивное замыкание отношения R_p.[11] Интуитивно ясно, что если $a_i R_p^+ a_j$, то последовательность сравнений, представленная путем p, выясняет, что $a_i < a_j$, поскольку ни один элемент не сравнивается сам с собой

---

[10] Предполагается, что каждое сравнение a и b дает результат $a < b$ или $b \le a$. Если $a_i < a_j$, то сравниваются элементы a_i и a_j, а если $a_i \le a_j$, то сравниваются элементы a_j и a_i.

[11] *Транзитивным замыканием* отношения R называется такое отношение R^+, что cR^+d тогда и только тогда, когда существует последовательность истинных утверждений вида $e_1 R e_2$, $e_2 R e_3$, ..., $e_{m-1} R e_m$, где $m \ge 2$, $c = e_1$ и $d = e_m$.

Лемма 3.3. Если на пути p выясняется, что a_m является k-м наименьшим элементом в S, то для любого $i \neq m$, $1 \leq i \leq n$ либо $a_i R_p^+ a_m$, либо $a_m R_p^+ a_i$.

Доказательство. Допустим, что некоторый элемент a_u не связан с a_m отношением R_p^+. Покажем, что, поместив a_u либо перед, либо после a_m в линейном порядке, заданном на множестве S, мы получим противоречие с предположением, что путь p правильно определил, что a_m является k-м наименьшим элементом в последовательности S. Пусть $S_1 = \{a_j \mid a_j R_p^+ a_u\}$ и $S_2 = \{a_j \mid a_u R_p^+ a_j\}$, а все остальные элементы множества S входят в множество S_3. По предположению a_u и a_m принадлежат S_3.

Если a_j — произвольный элемент множества S_1 (соответственно S_2) и $a_i R_p^+ a_j$ (соответственно $a_j R_p^+ a_i$), то в силу транзитивности a_i также принадлежит множеству S_1 (соответственно S_2). Следовательно, можно построить линейный порядок R, согласованный с R_p^+ так, что все элементы множества S_1 предшествуют всем элементам множества S_3, которые, в свою очередь, предшествуют всем элементам множества S_2.

В соответствии с предположением, элемент a_u не связан отношением R_p^+ ни с одним элементом из множества S_3. Допустим, что элемент a_u предшествует элементу a_m в смысле линейного порядка R, т.е. $a_u R a_m$. Тогда можно определить новый порядок R', совпадающий с R, за исключением того, что a_u непосредственно следует за a_m. Порядок R' также согласован с порядком R_p^+. Для каждого порядка R и R' можно найти разные целочисленные значения, удовлетворяющие соответственно R или R'. Однако в обоих случаях a_m не может быть k-м наименьшим элементом, так как в R' элементу a_m предшествует на один элемент меньше, чем в R. Следовательно, если какой-то элемент из S не связан с a_m отношением R_p^+, то дерево T не может правильно выбрать k-й наименьший элемент из множества S. Случай $a_m R a_u$ исследуется аналогично. □

Теорема 3.10. Если T — дерево решений, выбирающее k-й наименьший элемент в множестве S, $||S|| = n$, то глубина любого листа дерева T не меньше $n - 1$.

Доказательство. Рассмотрим путь p в дереве T из корня к листу. По лемме 3.3 либо $a_i R_p^+ a_m$, либо $a_m R_p^+ a_i$ для каждого $i \neq m$ где a_m — элемент, выбранный в качестве k-го наименьшего. Для элемента a_i, $i \neq m$, определим *ключевое* сравнение как первое сравнение на пути p, содержащее a_i, для которого выполнено одно из условий:

1) a_i сравнимо с a_m,

2) a_i сравнимо с a_j, $a_i R_p a_j$ и $a_i R_p^+ a_m$,

3) a_i сравнимо с a_j, $a_j R_p a_i$ и $a_m R_p^+ a_j$.

Интуитивно ясно, что ключевое сравнение для a_i — это первое сравнение, с помощью которого можно определить, предшествует ли элемент a_i элементу a_m или следует за ним.

Очевидно, что всякий элемент a_i, кроме a_m, обладает ключевым сравнением, иначе не выполнилось бы ни условие $a_i R_p^+ a_m$, ни условие $a_m R_p^+ a_i$. Кроме того, легко видеть, что никакое сравнение не может быть ключевым для обоих сравниваемых элементов. Так как в ключевые сравнения должны вовлекаться $n - 1$ элементов, то длина пути p должна быть не меньше $n - 1$. □

Следствие. Нахождение k-го наименьшего элемента в множестве S требует не меньше $n - 1$ сравнений как в среднем, так и в худшем случае.

На самом деле для всех k, кроме 1 и n, можно доказать более сильный результат, чем теорема 3.10 (см. упражнения 3.21–3.23).

Если необходимо найти алгоритм с хорошим ожидаемым временем, который вычисляет k-й наименьший элемент в множестве S, то можно применить стратегию, похожую на применявшуюся в алгоритме быстрой сортировки.

Алгоритм 3.7. Нахождение k-го наименьшего элемента.

Вход. Последовательность S, состоящая из n элементов произвольного множества с линейным порядком $\leq$, и целое число k, $1 \leq k \leq n$.

Выход. k-й наименьший элемент в множестве S.

Метод. Применяется рекурсивная процедура SELECT, приведенная на рис. 3.12. □

```
     procedure SELECT(k, S):
1.   if |S| = 1 then return единственный элемент множества S
     else
         begin
2.           выбрать случайным образом элемент множества S:
3.           пусть S1, S2 и S3 — последовательности элементов
             множества S, которые меньше, равны или больше a
                 соответственно
4.           if |S1| ≥ k then return SELECT(k, S1)
             else
5.               if |S1| + |S2| ≥ k then return a
6.               else return SELECT(k - |S1| - |S2|, S3)
         end
```

Рис. 3.12. Алгоритм выбора

Теорема 3.11. Ожидаемое время работы алгоритма 3.7 линейно.

Доказательство. Пусть $T(n)$ — ожидаемое время работы процедуры SELECT на последовательности из n элементов. Для простоты предположим, что все элементы множества S различны. (При наличии повторяющихся элементов результат не изменится.)

Пусть элемент a, выбранный в строке 2, является i-м наименьшим элементом в множестве S. Число i может принимать значения 1, 2, ..., n с одинаковой вероятностью. Если $i > k$, то процедура SELECT вызывается для последовательности, состоящей из $i - 1$ элементов, а если $i < k$, то для последовательности, состоящей из $n - i$ элементов. Следовательно, ожидаемое время работы рекурсии в строке 4 или 6 равно

$$\frac{1}{n}\left[\sum_{i=1}^{k-1} T(n-i) + \sum_{i=k+1}^{n} T(i-1)\right] = \frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i)\right].$$

Остальная часть процедуры SELECT занимает время cn, где c — константа, поэтому для $n \geq 2$ получаем следующее неравенство:

$$T(n) \leq cn + \max_k \left\{\frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i)\right]\right\}. \tag{3.9}$$

В качестве упражнения читатели могут самостоятельно доказать, используя индукцию, что если $T(1) \leq c$, то $T(n) \leq 4cn$ для всех $n \geq 2$. □

Упражнения

3.1. Примените алгоритм 3.1 для упорядочения последовательности строк *a*, *acb*, *bca*, *bbc*, *acc*, *bac*, *baa*.

3.2. Примените алгоритм 3.2 для упорядочения последовательности строк *a*, *bc*, *aab*, *baca*, *cbc*, *cc*.

3.3. Проверьте, изоморфны ли два дерева на рис. 3.13 в смысле примера 3.2.

3.4. Упорядочьте список 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9 7, применяя (*а*) пирамидальную сортировку, (*б*) быструю сортировку, (*в*) сортировку слиянием (алгоритм 2.4). Сколько сравнений производится в каждом случае?

3.5. Рассмотрите следующий алгоритм, упорядочивающий последовательность элементов a_1, a_2, ..., a_n, хранящихся в массиве A, т.е. $A[i] = a$ для $1 \leq i \leq n$.

```
procedure BUBBLESORT(A):
for j = n - 1 step -1 until 1 do
    for i = 1 step 1 until j do
        if A[i + 1] < A[i] then поменять местами A[i] и A[i + 1]
```

а) Докажите, что процедура BUBBLESORT упорядочивает элементы массива A в порядке неубывания.

б) Определите худшее и ожидаемое время работы процедуры BUBBLESORT.

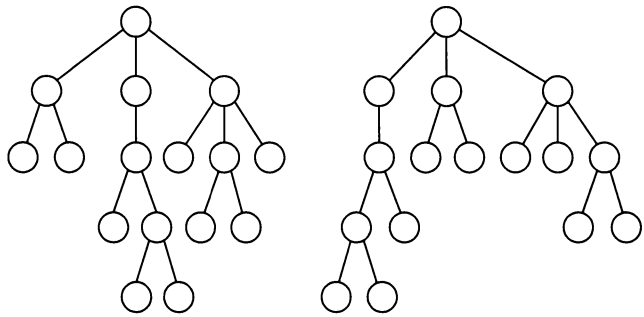


Рис. 3.13. Два дерева

3.6. Завершите доказательство того, что кучу можно создать за линейное время (теорема 3.5).

3.7. Завершите доказательство того, что временная сложность пирамидальной сортировки имеет порядок $O(n\log n)$ (теорема 3.6).

3.8. Покажите, что временная сложность алгоритма быстрой сортировки в худшем случае имеет порядок $O(n^2)$.

3.9. Какова временная сложность алгоритма 3.7, находящего k-й наименьший элемент, в худшем случае?

***3.10.** Докажите, что ожидаемое время упорядочения последовательности, состоящей из n элементов, ограничено снизу величиной $cn\log n$, где c — константа, завершив доказательство теоремы 3.7 и решив уравнение (3.2).

3.11. Завершите доказательство того, что алгоритм 3.6 находит k-й наименьший элемент за время $O(n)$ (теорема 3.9), решив уравнение (3.8).

3.12. Докажите корректность подпрограммы разбиения, приведенной на рис. 3.8, и проанализируйте время ее работы.

3.13. Завершите доказательство того, что ожидаемая временная сложность алгоритма 3.7 для нахождения k-го наименьшего элемента имеет порядок $O(n)$ (теорема 3.11), решив уравнение (3.9).

***3.14.** Покажите, что среднее количество сравнений, которые должен выполнить алгоритм 3.7, не превосходят $4n$. Можно ли улучшить эту оценку, если известно значение k, при котором будет применяться алгоритм?

***3.15.** Пусть S — последовательность элементов, содержащая m_i экземпляров i-го элемента, где $1 \le i \le k$. Пусть $n = \sum_{i=1}^{k} m_i$. Докажите, что

$$O\left(n + \log\left(\frac{n!}{m_1!\, m_2! \ldots m_k!}\right)\right)$$

сравнений необходимо и достаточно, чтобы упорядочить последовательность S.

3.16. Пусть S_1, S_2, ... , S_k — множество чисел, лежащих между 1 и n, и сумма мощностей всех S_i равна n. Опишите алгоритм, имеющий временную сложность порядка $O(n)$, упорядочивающий все множества S_i.

3.17. Пусть даны последовательность $a_1, a_2, \ldots, a_n$ и перестановка $\pi(1), \pi(2), \ldots, \pi(n)$. Напишите программу на псевдо-Алголе, которая располагала бы эту последовательность в порядке $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$, не копируя ее в новый массив. Какова худшая и ожидаема временная сложность этого алгоритма?

***3.18.** В процессе построения кучи размера $2^k - 1$ мы строим две кучи размера $2^{k-1} - 1$, затем объединяем их, добавляем корень и проталкиваем элемент, стоящий в корне, вниз на его надлежащее место. Так же легко можно было бы построить кучу, добавляя за один шаг один элемент в качестве листа и проталкивая этот новый элемент вверх по дереву. Напишите алгоритм, который строил бы кучу с помощью добавления листьев (по одному на каждом шаге), и сравните порядок временной сложности вашего алгоритма и алгоритма 3.3.

***3.19.** Рассмотрим прямоугольный массив. Расположим элементы в каждой строке в порядке возрастания. Затем расположим в порядке возрастания элементы каждого столбца. Докажите, что каждая строка останется упорядоченной.

***3.20.** Пусть $a_1, a_2, \ldots, a_n$ — последовательность элементов, а p и q — положительные целые числа. Рассмотрим подпоследовательности, образованные выбором каждого p-го элемента. Упорядочим их. Повторим этот процесс для q. Докажите, что подпоследовательности с шагом p останутся упорядоченными.

****3.21.** Рассмотрим нахождение с помощью сравнений наибольшего и второго по величине элемента в множестве, состоящем из n элементов. Докажите, что для этого необходимо и достаточно $n + \lceil \log n \rceil - 2$ сравнения.

****3.22.** Покажите, что среднее количество сравнений, необходимых для нахождения k-го наименьшего элемента в последовательности, состоящей из n

элементов, не меньше $(1 + 0{,}75\alpha(1 - \alpha))n$, где $\alpha = k/n$, а k и n достаточно велики.

**3.23. Покажите, что в худшем случае для нахождения k-го наименьшего элемента в множестве, состоящем из n элементов, требуется $n + \texttt{MIN}(k, n - k + 1) - 2$ сравнений.

**3.24. Пусть S — множество, состоящее из n целых чисел. Допустим, мы можем только складывать его элементы и сравнивать суммы. Сколько требуется сравнений для нахождения наибольшего элемента в множестве S при этих ограничениях?

*3.25. Алгоритм 3.6 разбивает свой аргумент на подпоследовательности, состоящие из 5 элементов. Будет ли этот алгоритм работать для других размеров, например 3, 7 или 9? Выберите размер, который минимизирует общее количество сравнений. На рис. 3.14 приведено наименьшее известное количество сравнений, достаточное для упорядочения множеств различных размеров. Известно, что для $n \leq 12$ эти числа оптимальны.

*3.26. Алгоритм 3.5 разбивает данную последовательность на подпоследовательности, состоящие из 5 элементов, а затем находит медиану медиан. Вместо того чтобы искать медиану медиан, не эффективнее ли искать другой элемент, например $\lfloor k/5 \rfloor$-й?

Размер n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Сравнения	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50

Рис. 3.14. Наименьшее известное количество сравнений для упорядочения последовательности из n элементов

*3.27. Рассмотрим следующий метод сортировки. Начнем с последовательности, состоящей из одного элемента. В эту последовательность будем по очереди вставлять другие элементы с помощью бинарного поиска. Придумайте структуру данных, позволяющую быстро выполнять бинарный поиск и вставлять элементы. Можете ли вы реализовать эту сортировку за время $O(n \log n)$?

**3.28. Вместо выбора произвольного элемента для разбиения множества размера n, как в алгоритме быстрой сортировки и алгоритме 3.7, можно было бы извлечь небольшую выборку, например, размера s, найти ее медиану и использовать ее, чтобы разделить все множество. Укажите, как выбрать s в зависимости от n, чтобы минимизировать ожидаемое количество сравнений, необходимых для сортировки.

**3.29. Обобщите идею упражнения 3.28 так, чтобы минимизировать количество сравнений, необходимых для нахождения порядковой статистики. (*Подсказка*: извлеките из выборки два элемента, между которыми с большой вероятностью лежит искомый элемент.)

3.30. Метод сортировки называется *устойчивым*, если одинаковые элементы остаются в упорядоченной последовательности в том же относительном порядке, в каком они были в исходной последовательности. Какой из перечисленных ниже алгоритмов устойчив?

а) Метод пузырька (упражнение 3.5).

б) Сортировка слиянием (алгоритм 2.4).

в) Пирамидальная сортировка (алгоритм 3.4).

г) Быстрая сортировка (алгоритм 3.5).

Проблема для исследования

Некоторые проблемы, касающиеся количества сравнений, необходимых в определенных ситуациях, все еще не решены. Например, пусть требуется найти k наименьших элементов множества, состоящего из n элементов. Случай $k = 3$ рассмотрен в работе Pratt and Yao [1973]. Для $n \geq 13$ неизвестно, оптимальны ли числа, приведенные на рис. 3.14, для упорядочения n элементов. Для малых n оптимальным относительно количества сравнений является алгоритм сортировки из работы Ford and Johnson [1959].

Библиографические замечания

Книга Knuth [1973a] представляет собой энциклопедическое руководство по методам сортировки. Пирамидальная сортировка была впервые предложена в работе Williams [1965]; некоторые улучшения сделаны в работе Floyd [1964]. Алгоритм быстрой сортировки изобрел Hoare [1962]. Улучшения быстрой сортировки по схеме упражнения 3.28 предложены в публикациях Singleton [1969], а также Frazer and McKellar [1970]. Алгоритм 3.6 для нахождения порядковой статистики, линейный в худшем случае, изложен в работе Blum, Floyd, Pratt, Rivest and Tarjan [1972]. Исследованию количества сравнений, необходимого для нахождения некоторых порядковых статистик, посвящены работы Hadian and Sobel [1969] и Pratt and Yao [1973].

Результат упражнения 3.21 получен в работе Kislitsyn [1964]. Упражнения 3.22, 3.23 и 3.29 взяты из работы Floyd and Rivest [1973], где также приведена более точная нижняя оценка, чем в упражнении 3.22. Упражнения 3.19, 3.20 и некоторые обобщения обсуждаются в работах Gale and Karp [1970] и Liu [1972]. Интересное приложение сортировки для построения выпуклой оболочки множества точек на плоскости дано в Graham [1972]. Устойчивая сортировка рассмотрена в работе Horvath [1974].

ГЛАВА 4

Структуры данных для работы с множествами

Для разработки эффективного алгоритма решения конкретной задачи полезно изучить ее сущность. Во многих ситуациях задачу можно сформулировать в терминах фундаментальных математических понятий, таких как множество, и тогда алгоритм для ее решения можно выразить через базовые операции над основными объектами. Преимущество такого подхода состоит в том, что он позволяет исследовать несколько различных структур данных и выбрать из них наиболее подходящую для данной задачи. Таким образом, с разработкой хорошего алгоритма тесно связана разработка хорошей структуры данных.

В настоящей главе изучаются семь базовых операций над множествами, типичных для многих задач информационного поиска. Мы приведем разнообразные структуры данных для представления множеств и рассмотрим пригодность каждой из них в ситуациях, когда выполняется та или иная последовательность базовых операций.

4.1. Базовые операции над множествами

Введем следующие базовые операции над множествами.

1. `MEMBER`(a, S). Устанавливает принадлежность элемента a множеству S; если a принадлежит S, то выдает ответ "да", в противном случае — "нет".
2. `INSERT`(a, S). Заменяет S на $S \cup \{a\}$.
3. `DELETE` (a, S). Заменяет S на $S - \{a\}$.
4. `UNION`(S_1, S_2, S_3). Заменяет множества S_1 и S_2 множеством $S_3 = S_1 \cup S_2$. В дальнейшем будем предполагать, что множества S_1 и S_2 не пересекаются, чтобы избежать необходимости удалять повторяющиеся экземпляры одного и того же элемента в $S_1 \cup S_2$.
5. `FIND`(a). Выдает имя того множества, которому в данный момент принадлежит элемент a. Если элемент a принадлежит более чем одному множеству, то операция не определена.

6. SPLIT(a, S). Здесь предполагается, что на множестве S введен линейный порядок. Операция разбивает S на два множества: $S_1 = \{b|b \leq a$ и $b \in S\}$ и $S_2 = \{b|b > a$ и $b \in S\}$.
7. MIN(S). Выдает наименьший (в смысле отношения $\leq$) элемент множества S.

Многие практические задачи можно свести к одной или нескольким подзадачам, которые можно абстрактно сформулировать как последовательность базовых инструкций над некоторой базой данных (универсальным множеством элементов). В данной главе мы будем изучать последовательности инструкций σ, состоящих из базовых операций.

Например, выполнение последовательностей, состоящих из операций MEMBER, INSERT и DELETE, является неотъемлемой частью многих задач поиска. Структура данных, которую можно использовать для выполнения последовательности этих операций, называется *словарем*. В настоящей главе рассматривается несколько структур данных (таких, как хеш-таблицы, деревья бинарного поиска и 2-3-деревья), которые могут быть словарями.

С этими операциями связано множество интересных вопросов. Нас в основном будет интересовать временная сложность последовательностей операций σ как функция их длины и размера базы данных. Мы исследуем худшую и ожидаемую временную сложность, а также проведем различия между сложностью онлайн- и офлайн-алгоритмов.

Определение. Выполнение последовательности инструкций σ в режиме *онлайн* означает, что инструкции в последовательности выполняются слева направо, причем i-я инструкция в последовательности должна выполняться без учета всех следующих инструкций. В режиме *офлайн* перед выполнением любой инструкции можно просматривать всю последовательность σ.

Очевидно, что всякий онлайн-алгоритм может работать как офлайн-алгоритм, но обратное верно не всегда. Мы рассмотрим ситуации, когда офлайн-алгоритм работает быстрее любого известного онлайн-алгоритма. Однако во многих приложениях приходится рассматривать только онлайн-алгоритмы.

Пусть дана последовательность инструкций, которую необходимо выполнить. Основной вопрос заключается в том, какую структуру данных использовать для представления соответствующей базы данных. Часто задача требует соблюдения тщательного баланса между противоположными целями. В типичной ситуации последовательность состоит из нескольких операций, которые необходимо выполнить многократно, причем зачастую в неизвестном порядке. Могут существовать несколько структур данных, каждая из которых позволяет одну операцию выполнить легко, а другие — с большим трудом. Во многих случаях наилучшим решением является компромисс. Часто мы будем использовать структуру, которая

не делает максимально легким выполнение ни одной из операций, но позволяет выполнить всю работу быстрее, чем при любом очевидном подходе.

Приведем пример, иллюстрирующий формулировку задачи о построении остовного дерева для графа в терминах последовательности операций над множествами.

Определение. Пусть $G = (V, E)$ — неориентированный граф. *Остовным деревом графа* G называется неориентированное дерево $S = (V, T)$, где $T \subseteq E$ *Остовным лесом графа* $G = (V, E)$ называется множество неориентированных деревьев $\{(V_1, T_1), (V_2, T_2), ..., (V_k, T_k)\}$, в котором множества вершин V_i образуют разбиение множества V,[1] а каждое множество T_i является (возможно, пустым) подмножеством множества E.

Функцией стоимости графа $G = (V, E)$ называется отображение c из E в множество вещественных чисел. Стоимостью подграфа $G' = (V', E')$ графа G является функция $c(G') = \sum_{e \in E'} c(e)$.

Пример 4.1. Рассмотрим алгоритм, приведенный на рис. 4.1, для нахождения остовного дерева $S = (V, T)$ наименьшей стоимости в заданном графе $G = (V, E)$. Этот алгоритм нахождения минимального остовного дерева подробно обсуждается в разделе 5.1, а его применение проиллюстрировано в примере 5.1.

```
      begin
1.        T ← ∅
2.        VS ← ∅
3.        for для каждой вершины v ∈ V do
              добавить в VS одноэлементное множество {v}:
4.        while ||VS|| > 1 do
              begin
5.                выбрать из E ребро (v, w) наименьшей стоимости;
6.                удалить ребро (v, w) из E:
7.                if v и w принадлежат разным множествам W1 и W2
                     из VS then
                      begin
8.                        заменить W1 и W2 в VS на W1 ∪ W2
9.                        добавить ребро (v, w) в T
                      end
              end
      end
```

Рис. 4.1. Алгоритм нахождения минимального остовного дерева

[1] Т.е. $V_1 \cup V_2 \cup ... \cup V_k = V$ и $V_i \cap V_j = \varnothing$ при $i \neq j$.

В алгоритме, представленном на рис. 4.1 упоминаются три множества — E, T и VS. Множество E содержит ребра графа G, а T используется для накапливания ребер окончательного остовного дерева. Работа алгоритма состоит в преобразовании остовного леса графа G в единое остовное дерево. Множество VS состоит из множества вершин, входящих в деревья этого остовного леса. Вначале VS содержит все вершины графа G в виде одноэлементных множеств.

Этот алгоритм можно рассматривать как последовательность операций над тремя множествами: E, T и VS. В строке 1 инициализируется множество T, в строках 2 и 3 — множество VS, причем элементы множества VS сами являются множествами. В строке 3 в множество VS добавляются исходные одноэлементные множества. Строка 4, управляющая основным циклом алгоритма, предполагает наличие счетчика множеств, входящих в множество VS. В строке 7 мы проверяем, соединяет ли ребро (v, w) два остовных дерева в остовном лесу. Если да, то в строке 8 эти два дерева сливаются, а в строке 9 ребро (v, w) добавляется в окончательное остовное дерево.

Строка 7 предполагает, что мы можем найти имя множества в VS, которое содержит конкретную вершину. (Вид имен, используемых для множеств в VS, может быть произвольным.) По существу, строка 7 предполагает, что мы можем эффективно выполнить базовую операцию `FIND`. Аналогично строка 8 предполагает, что над непересекающимися множествами вершин можно выполнить операцию `UNION`.

Найти структуру данных, на которой быстро выполняется операция `UNION` или `FIND`, довольно легко. Однако в данном случае требуется, чтобы на одной структуре данных можно было легко реализовать обе операции — `UNION` и `FIND`. Более того, поскольку выполнение операции `UNION` в строке 8 зависит от результата выполнения операций `FIND` в строке 7, искомая последовательность операций `UNION` и `FIND` должна выполняться в режиме онлайн. Две такие структуры изучаются в разделах 4.6 и 4.7.

Рассмотрим последовательность операций, выполняемых на множестве ребер E. В строке 5 необходимо выполнить базовую операцию `MIN`, а в строке 6 — операцию `DELETE`. Хорошая структура данных для этих двух операций нам уже встречалась — это куча из раздела 3.4. (Хотя там с помощью кучи мы искали наибольший элемент, очевидно, что с его помощью можно также найти и наименьший элемент.)

Наконец, в строке 9 с помощью базовой операции `INSERT` в множество T ребер остовного дерева добавляется новое ребро. Здесь достаточно простого списка. □

4.2. Хеширование

Кроме операций, из которых состоит последовательность σ, не менее важно правильно выбрать подходящую структуру данных для ее выполнения. Этот вопрос связан с размером базы данных (универсального множества), с которой работают операции из σ. Например, в главе 3 мы видели, что с помощью поразрядной сортировки последовательность, состоящую из n элементов, можно упорядочить за линейное время, если элементами рассматриваемого множества являются целые числа из подходящего диапазона. Однако, если эти элементы принадлежат произвольному линейно упорядоченному множеству, то наилучшая временная сложность, которой мы смогли добиться, имела порядок $O(n \log n)$.

В данном разделе мы исследуем проблему обработки изменяющегося множества S. В него будут добавляться новые элементы, из него будут удаляться старые элементы, и время от времени необходимо будет отвечать на вопрос: принадлежит ли в данный момент элемент x множеству S? Эта задача естественно решается с помощью словаря. Нам нужна структура данных, которая позволит удобно выполнять последовательности инструкций, состоящие из операций `MEMBER`, `INSERT` и `DELETE`. Будем предполагать, что элементы, которые могут появиться в множестве S, принадлежат очень большому универсальному множеству, так что представлять S в виде бинарного вектора неразумно с практической точки зрения.

Пример 4.2. Компилятор или ассемблер поддерживает таблицу символов всех идентификаторов, которые используются в транслируемой программе. В большинстве языков программирования множество всех возможных идентификаторов очень велико. Например, в Фортране около $1{,}62 \cdot 10^9$ возможных идентификаторов. Следовательно, нереально представить таблицу символов в виде массива с отдельными ячейками для всех возможных идентификаторов, независимо от того, появляются они в программе или нет.

Операции, производимые компилятором с таблицей символов, разделяются на две категории. Во-первых, в таблицу необходимо добавлять новые обнаруженные идентификаторы. Для этого в таблице создается ячейка, в которую записывается конкретный идентификатор и данные о нем (например, действительный или целочисленный). Во-вторых, время от времени компилятор может запрашивать информацию о каком-нибудь идентификаторе (например, имеет ли этот идентификатор целочисленный тип).

Таким образом, для реализации таблицы символов, вероятно, подойдет структура данных, способная обеспечить выполнение операций `INSERT` и `MEMBER`. Структура данных, обсуждаемая в этом разделе, действительно часто используется для реализации таблицы символов. □

Рассмотрим *хеширование* — метод, обеспечивающий выполнение не только операций INSERT и MEMBER, необходимых для построения таблицы символов, но и операции DELETE. Существует множество вариантов хеширования, но мы обсудим здесь только основную идею.

Схема хеширования показана на рис. 4.2. *Функция хеширования* h отображает элементы универсального множества (например, в случае таблицы символов — все возможные идентификаторы) в множество целых чисел от 0 до $m - 1$. В дальнейшем будем предполагать, что для всех элементов a значение $h(a)$ можно вычислить за постоянное время. Элементами массива A размера m служат указатели на списки элементов множества S. Список, на который указывает $A[i]$, состоит из всех тех элементов $a \in S$, для которых $h(a) = i$.

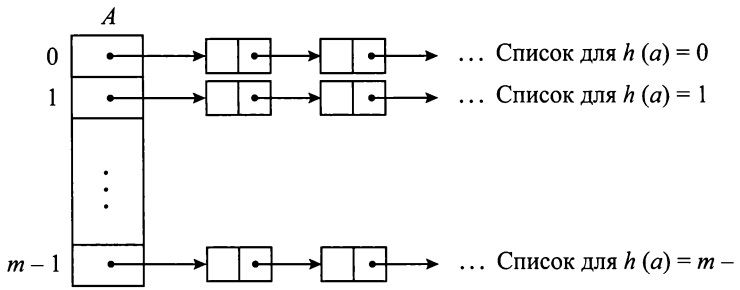


Рис. 4.2. Схема хеширования

Для того чтобы выполнить операцию INSERT(a, S), необходимо вычислить $h(a)$ и затем просмотреть список, на который указывает $A[h(a)]$. Если элемента a в этом списке нет, его необходимо добавить в конец списка. Чтобы выполнить операцию DELETE(a, S), необходимо также просмотреть список $A[h(a)]$ и удалить элемент a, если он там есть. Аналогично просматривается список $A[h(a)]$ при выполнении операции MEMBER(a, S).

Вычислительную сложность этой схемы хеширования нетрудно оценить. В худшем случае она не очень хорошая. Например, допустим, что последовательность σ состоит из n различных операций INSERT. Допустим, что на всех элементах, которые следует вставить, функция h принимает одинаковые значения, так что все элементы оказываются в одном и том же списке. В этой ситуации для выполнения i-й операции из последовательности σ требуется время, пропорциональное i. Таким образом, чтобы добавить все n элементов в множество S, хеширование может потребовать времени порядка n^2.

Однако ожидаемая временная сложность этого процесса выглядит намного лучше. Если значение $h(a)$ с одинаковой вероятностью может быть любым целым числом от 0 и $m - 1$ и в множество вставляется $n \leq m$ элементов, то при вставке i-го

элемента средняя длина того списка, в который он помещается, равна $(i - 1)/m$, т.е. всегда меньше 1. Следовательно, ожидаемое время, необходимое для вставки n элементов, имеет порядок $O(n)$. Если вместе с операциями INSERT выполняются $O(n)$ операций DELETE и MEMBER, то общее ожидаемое время по-прежнему составляет $O(n)$.

Следует иметь в виду, что в этих расчетах размер m хеш-таблицы предполагался не меньшим максимального размера n множества S. Однако число n, как правило, заранее не известно. В таком случае разумнее всего создавать последовательности хеш-таблиц T_0, T_1, T_2, ...

Сначала выбирается подходящее значение m для размера исходной таблицы T_0. Затем, когда количество элементов, вставленных в T_0, превзойдет m, создается новая таблица T_1 размера $2m$ и с помощью новой функции хеширования[2] все элементы, находящиеся в данный момент в таблице T_0, перемещаются в таблицу T_1. Теперь старая таблица T_0 больше не нужна. Вставка новых элементов в таблицу T_1 продолжается до тех пор, пока число элементов не превзойдет $2m$. В этот момент создается новая таблица T_2 размера $4m$ и функция хеширования меняется, чтобы переместить старые элементы из T_1 в T_2. В общем, всякий раз, когда в таблице T_{k-1} оказывается $2^{k-1}m$ элементов, создается таблица T_k размера $2^k m$. Этот процесс продолжается до тех пор, пока не будут вставлены все элементы.

Подсчитаем ожидаемое время, требуемое для вставки в хеш-таблицу 2^k элементов, если применять изложенную только что схему и считать, что $m = 1$. Легко видеть, что этот процесс описывается рекуррентным уравнением

$$T(1) = 1,$$

$$T(2^k) = T(2^{k-1}) + 2^k,$$

решением которого, очевидно, является функция $T(2^k) = 2^{k+1} - 1$.

Таким образом, с помощью хеширования последовательность из n операций INSERT, MEMBER и DELETE можно выполнить за ожидаемое время порядка $O(n)$.

Здесь важен выбор функции хеширования h. Если элементы, добавляемые в множество S, представлены целыми числами, равномерно распределенными между 0 и некоторым числом r, где $r \gg n$, то в качестве $h(a)$ можно взять $a \bmod m$, где m — размер текущей хеш-таблицы. Другие примеры функций хеширования приведены в работах, упоминаемых в конце главы.

4.3. Бинарный поиск

В данном разделе сравниваются три различных решения одной простой задачи поиска. Дано множество S, состоящее из n элементов, принадлежащих большому

[2]Эта функция принимает значения от 0 до $2m - 1$.

универсальному множеству. Требуется выполнить последовательность σ, состоящую только из операций MEMBER.

Наиболее прямолинейным способом решения было бы запомнить элементы множества S в некотором списке. В ходе последовательных просмотров этого списка каждая операция MEMBER(a, S) выполняется до тех пор, пока не будет найден элемент a или не будут просмотрены все элементы списка. Выполнение всех операций из σ заняло бы тогда время порядка $n \cdot |\sigma|$ как в худшем случае, так и в среднем. Основное преимущество этой схемы заключается в том, что она почти не требует предварительной работы.

Другой способ — разместить элементы множества S в хеш-таблице размера $\|S\|$. Операция MEMBER(a, S) выполняется с помощью поиска в списке $h(a)$. Если удастся найти хорошую функцию h, то выполнение последовательности σ займет время $O(|\sigma|)$ в среднем и $O(n|\sigma|)$ в худшем случае. Основная трудность здесь связана с нахождением функции хеширования, равномерно распределяющей элементы из S в хеш-таблице.

Если на множестве S задан линейный порядок ≤, то третьим решением будет бинарный поиск. Элементы из S хранятся в массиве A. Этот массив упорядочивается так, чтобы выполнялось условие $A[1] < A[2] < ... < A[n]$. Теперь, чтобы установить, принадлежит ли элемент a множеству S, необходимо сравнить его с элементом b, который хранится в ячейке $\lfloor(1+n)/2\rfloor$. Если $a = b$, то процедура прекращает работу и выдает ответ "да". В противном случае она повторяется на первой половине массива, если $a < b$, и на второй, если $a > b$. Повторно разбивая область поиска пополам, можно не более чем за $\lfloor \log(n+1) \rfloor$ сравнений либо найти элемент a, либо установить, что его нет в множестве S.

Рекурсивная процедура SEARCH(a, f, l), приведенная на рис. 4.3, ищет элемент a в ячейках $f, f+1, f+2, ..., l$ массива A. Для установления принадлежности элемента a множеству S вызывается процедура SEARCH(a, 1, n).

```
procedure SEARCH(a, f, l):
if f > l then return "нет"
else
    if a = A[⌊(f + l)/2⌋] then return "да"
    else
        if a < A[⌊(f + l)/2⌋] then
            return SEARCH(a, f, ⌊(f + l)/2⌋ - 1)
        else return SEARCH(a, ⌊(f + l)/2⌋ + 1, l)
```

Рис. 4.3. Процедура бинарного поиска

Чтобы понять, почему эта процедура работает, представим массив A в виде бинарного дерева. Корень находится в ячейке $\lfloor(1+n)/2\rfloor$, а его левый и правый сыно-

вья — в ячейках $\lfloor(1+n)/4\rfloor$ и $\lfloor 3(1+n)/4\rfloor$ соответственно и т.д. Эта интерпретация бинарного поиска станет яснее в следующем разделе.

Легко показать, что на поиск элемента в A процедура SEARCH тратит не более $\lceil \log(n+1)\rceil$ сравнений, поскольку никакой путь в рассматриваемом дереве не длиннее $\lceil \log(n+1)\rceil$. Если все элементы являются равновероятными, то можно также показать (упражнение 4.4), что процедура SEARCH выполняет оптимальное среднее число сравнений (а именно $|\sigma|\log n$), необходимое для выполнения операций MEMBER в последовательности σ.[3]

4.4. Деревья бинарного поиска

Рассмотрим следующую задачу. В множестве S выполняются операции вставки и удаления элементов. Время от времени необходимо узнать, принадлежит ли указанный элемент множеству S, или найти текущий наименьший элемент в S. Предполагается, что элементы добавляются в S из большого универсального множества с линейно упорядоченным отношением $\leq$. Эту задачу можно сформулировать в общем виде как выполнение последовательности операций INSERT, DELETE, MEMBER и MIN.

Выше было показано, что для выполнения последовательности операций INSERT, DELETE и MEMBER целесообразно использовать хеш-таблицу. Однако невозможно найти наименьший элемент, не просмотрев всю хеш-таблицу. Структурой данных, подходящей для всех четырех операций, является дерево бинарного поиска.

Определение. *Деревом бинарного поиска* для множества S называется размеченное бинарное дерево, каждая вершина v которого помечена элементом $l(v) \in S$ так, что

1. $l(u) < l(v)$ для каждой вершины u в левом поддереве вершины v.
2. $l(u) > l(v)$ для каждой вершины u из правого поддерева вершины v.
3. Для каждого элемента $a \in S$ существует только одна вершина v, для которого $l(v) = a$.

Заметим, что из условий 1 и 2 вытекает, что метки этого дерева расположены в соответствии с симметричным порядком. Кроме того, условие 3 следует из условий 1 и 2.

[3]Разумеется, хеширование не сводится исключительно к операциям сравнения, поэтому, возможно, оно в каком-то смысле лучше бинарного поиска и во многих случаях это действительно так.

Пример 4.3. На рис. 4.4 изображено возможное бинарное дерево для ключевых слов Алгола **begin**, **else**, **end**, **if**, **then**. Здесь линейным порядком является лексикографический порядок. □

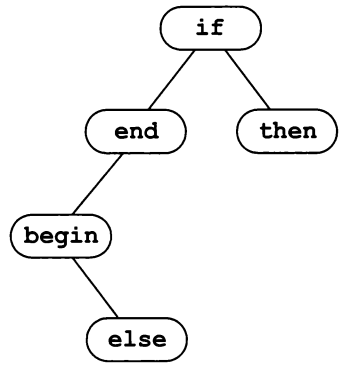


Рис. 4.4. Дерево бинарного поиска

Для того чтобы выяснить, принадлежит ли элемент a множеству S, представленному деревом бинарного поиска, необходимо сравнить a с меткой корня. Если метка корня равна a, то очевидно, что $a \in S$. Если a меньше метки корня, то переходим к левому поддереву корня (если оно есть). Если a больше метки корня, то переходим к правому поддереву корня. Если элемент a присутствует в дереве, то он в конце концов будет найден. В противном случае процесс прекращается, когда необходимо будет найти несуществующее левое или правое поддерево.

Алгоритм 4.1. Просмотр дерева бинарного поиска

Вход. Дерево бинарного поиска T, множество S и элемент a.

Выход. "Да", если $a \in S$, и "нет" в противном случае.

Метод. Если дерево T пусто,[4] выдать "нет". В противном случае пусть r — корень дерева T. Тогда алгоритм состоит из единственного вызова рекурсивной процедуры SEARCH(a, S),приведенной на рис. 4.5. □

Очевидно, что алгоритм 4.1 достаточен для выполнения операции MEMBER(a, S). Более того, его легко модифицировать так, чтобы он выполнял операцию INSERT(a, S). Если дерево пустое, то создается корень с меткой a. Если дерево не пустое и элемента, который требуется вставить, в дереве нет, то процедуре SEARCH

[4]Хотя наше определение понятия дерева требует, чтобы дерево содержало хотя бы одну вершину, т.е. корень, во многих алгоритмах мы будем трактовать пустое дерево (дерево без вершин) как бинарное.

не удается найти сына в строке 3 или 5. Вместо того чтобы выдать "нет" в строке 4 или 6 соответственно, для этого элемента создается новая вершина там, где должен был быть отсутствующий сын.

```
      procedure SEARCH(a, v):
1.    if a = l(v) then return "да"
      else
2.        if a < l(v) then
3.            if v имеет левого сына w then return SEARCH(a, w)
4.            else return "нет"
          else
5.            if v имеет правого сына w then return SEARCH(a, w)
6.            else return "нет"
```

Рис. 4.5. Обход дерева бинарного поиска

Кроме того, деревья бинарного поиска удобны для выполнения операций MIN и DELETE. Для нахождения наименьшего элемента в дереве бинарного поиска T выполняется обход пути v_0, v_1, ..., v_p, где v_0 — корень дерева T; v_1 — левый сын вершины v_{i-1}, $1 \le i \le p$ и у вершины v_p нет левого сына. Метка в вершине v_p является наименьшим элементом в дереве T. В некоторых задачах удобно использовать указатель на v_p, чтобы обеспечить доступ к наименьшему элементу за постоянное время.

Реализовать операцию DELETE(a, S) немного труднее. Допустим, элемент a, подлежащий удалению, расположен в вершине v. Возможны три случая.

1. Вершина v — лист; в этом случае вершина v удаляется из дерева.
2. У вершины v в точности один сын. В этом случае делаем отца вершины v отцом ее сына, удаляя v из дерева (если вершина v — корень, то ее сына делаем новым корнем).
3. У вершины v два сына. В этом случае находим в левом поддереве вершины v наибольший элемент b, рекурсивно удаляем из этого поддерева вершину, содержащую b, и помечаем вершину v элементом b. (Заметим, что среди элементов, меньших a, элемент b будет наибольшим элементом всего дерева.)

В качестве упражнения предлагаем написать программу на псевдо-Алголе для операции DELETE. Заметим, что одно выполнение любой из операций MEMBER, INSERT, DELETE и MIN можно осуществить за время $O(n)$.

Пример 4.4. Предположим, требуется удалить слово **if** из дерева бинарного поиска, изображенного на рис. 4.4. Слово **if** расположено в корне, у которого два сына. Наибольшее слово, лексикографически меньшее **if**, расположенное в левом поддереве корня, — это **end**. Удаляем из дерева вершину с меткой **end** и за-

меняем **if** на **end** в корне. Затем, поскольку вершина с меткой **end** имеет одного сына (**begin**), делаем вершину с меткой **begin** сыном корня. Полученное дерево показано на рис. 4.6. □

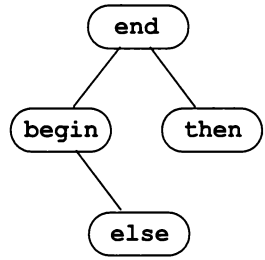


Рис. 4.6. Дерево бинарного поиска после выполнения операции DELETE

Оценим временную сложность последовательности, состоящей из n операций INSERT, когда базовое множество представлено деревом бинарного поиска. Время, необходимое для вставки элемента a в дерево бинарного поиска, пропорционально числу сравнений a с элементами, уже находящимися в дереве. Следовательно, время можно измерять количеством сравнений.

В худшем случае временная сложность вставки n элементов может оказаться квадратичной. Например, допустим, что последовательность добавляемых элементов оказалась упорядоченной в порядке возрастания. Тогда дерево поиска будет состоять из одной цепи правых сыновей. Однако если в дерево вставляются n случайных элементов, то, как показывает следующая теорема, временная сложность вставки составит $O(n\log n)$.

Теорема 4.1. Среднее количество сравнений, необходимых для вставки $n \geq 1$ случайных элементов в пустое дерево бинарного поиска, равно $O(n\log n)$.

Доказательство. Пусть $T(n)$ — число сравнений между элементами последовательности $a_1, a_2, ..., a_n$ при построении дерева бинарного поиска, $T(0) = 0$. Пусть $b_1, b_2, ..., b_n$ — та же последовательность в порядке возрастания.

Если $a_1, a_2, ..., a_n$ — случайная последовательность элементов, то a_1 с равной вероятностью совпадает с b_j для любого j, $1 < j < n$. Элемент a_1 становится корнем дерева бинарного поиска, и в окончательном дереве $j - 1$ элементов $b_1, b_2, ..., b_{j-1}$ будут находиться в левом поддереве корня и $n - j$ элементов $b_{j+1}, b_{j+2}, .., b_n$ — в правом поддереве.

Подсчитаем среднее число сравнений, необходимых для вставки в дерево элементов $b_1, b_2, ..., b_{j-1}$. Каждый из этих элементов обязательно сравнивается с корнем. Это сравнение выполняется $j - 1$ раз. Затем по индукции получаем, что еще потребуется $T(j - 1)$ сравнений, чтобы вставить элементы $b_1, b_2, ..., b_{j-1}$ в левое поддерево. Итак, необходимо $j - 1 + T(j - 1)$ сравнений, чтобы вставить элементы $b_1, b_2, ..., b_{j-1}$ в дерево бинарного поиска. Аналогично $n - j + T(n - j)$ сравнений потребуется, чтобы вставить в дерево элементы $b_{j+1}, b_{j+2}, ..., b_n$. Поскольку j с равной вероятностью принимает любое значение от 1 до n, то

$$T(n) = \frac{1}{n}\sum_{j=1}^{n}\bigl(n - 1 + T(j-1) + T(n-j)\bigr). \tag{4.1}$$

После простых алгебраических преобразований получаем

$$T(n) = n - 1 + \frac{2}{n}\sum_{j=0}^{n-1} T(j). \tag{4.2}$$

Используя приемы, описанные в разделе 3.5, можно показать, что

$$T(n) \le kn\log n,$$

где $k = \ln 4 = 1{,}39$. Таким образом, среднее количество сравнений при вставке n элементов в дерево бинарного поиска имеет порядок $O(n\log n)$. □

Итак, с помощью методов, описанных в этом разделе, можно выполнить случайную последовательность, состоящую из n операций `INSERT`, `DELETE`, `MEMBER` и `MIN` за ожидаемое время $O(n\log n)$. В худшем случае ее временная сложность является квадратичной. Однако даже это худшее время можно улучшить до $O(n\log n)$ с помощью одной из схем сбалансированных деревьев (2-3-деревьев, AVL-деревьев или деревьев с ограниченной балансировкой), которые обсуждаются в разделе 4.9 и упражнениях 4.30–4.37.

4.5. Оптимальные деревья бинарного поиска

В разделе 4.3 по заданному множеству $S = \{a_1, a_2, ..., a_n\}$, являющемуся подмножеством некоторого большого универсального множества U, строилась структура данных, позволяющая эффективно выполнить последовательность σ, состоящую только из операций `MEMBER`. Рассмотрим эту задачу снова, но теперь предположим, что кроме множества S задается для каждого $a \in U$ вероятность появления в последовательности σ операции `MEMBER`(a, S). Теперь нам хотелось бы построить дерево бинарного поиска для S так, чтобы последовательность σ операций `MEMBER` можно было выполнить в онлайн-режиме с наименьшим средним числом сравнений.

Пусть $a_1, a_2, ..., a_n$ — элементы множества S, перечисленные в порядке возрастания и p_i — вероятность операции MEMBER(a_i, S) в последовательности σ. Пусть q_0 — вероятность операции MEMBER(a, S) в последовательности для некоторого $a < a_1$, а q_i — вероятность операции MEMBER(a, S) для некоторого a, $a_i < a < a_{i+1}$. Наконец, пусть q_n — вероятность операции MEMBER(a, S) в последовательности σ для некоторого $a > a_n$. Для определения стоимости дерева бинарного поиска удобно добавить к нему $n + 1$ фиктивных листьев, чтобы учесть элементы из $U - S$. Мы будем обозначать эти листья числами 0, 1, ..., n.

На рис. 4.7 показано дерево бинарного поиска, изображенное на рис. 4.4, с добавленными фиктивными листьями. Например, лист с меткой 3 представляет те элементы a, для которых **end** $< a <$ **if**.

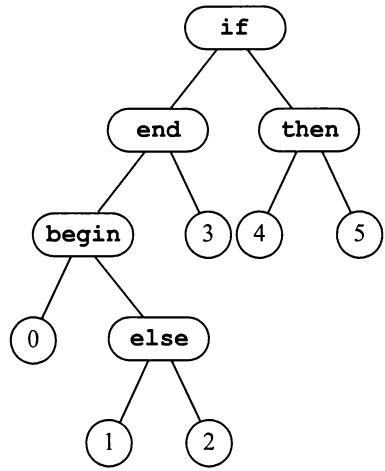


Рис. 4.7. Дерево бинарного поиска с добавленными листьями

Нам необходимо определить стоимость дерева бинарного поиска. Если элемент a равен метке $l(v)$ некоторой вершины v, то число вершин, посещенных во время выполнения операции MEMBER(a, S), на единицу больше глубины вершины v. Если $a \notin S$ и $a_i < a < a_{i+1}$, то число вершин, посещенных при выполнении операции MEMBER(a, S), равно глубине фиктивного листа i. Следовательно, *стоимость* дерева бинарного поиска можно определить по формуле

$$\sum_{i=1}^{n} p_i \left(\texttt{DEPTH}(a_i)+1\right) + \sum_{i=0}^{n} q_i \texttt{DEPTH}(i).$$

Имея дерево бинарного поиска T, обладающее наименьшей стоимостью, мы можем выполнить последовательность операций MEMBER с наименьшим средним

числом посещений вершин, применив для выполнения каждой операции MEMBER алгоритм 4.1.

Как найти дерево наименьшей стоимости по заданным числам p_i и q_i? Подход "разделяй и властвуй" предлагает определить элемент a_i, который необходимо разместить в корне. Это позволило бы свести данную задачу к двум подзадачам построения левого и правого поддеревьев. Однако, похоже, трудно найти корень, не решив всю задачу. Следовательно, придется решать $2n$ подзадач: по две для каждого возможного корня. Это естественно приводит к решению с помощью динамического программирования.

Для $0 \le i < j \le n$ обозначим через T_{ij} дерево наименьшей стоимости для подмножества $\{a_{i+1}, a_{i+2}, ..., a_j\}$. Пусть c_{ij} — стоимость дерева T_{ij}, а r_{ij} — его корень. Вес w_{ij} дерева T_{ij} равен $q_i + (p_{i+1} + q_{i+1}) + ... + (p_j + q_j)$.

Как показано на рис. 4.8, дерево T_{ij} состоит из корня a_k, левого поддерева $T_{i,k-1}$ (т.е. дерева наименьшей стоимости для $\{a_{i+1}, a_{i+2}, ..., a_{k-1}\}$ и правого поддерева T_{kj} (т.е. дерева наименьшей стоимости для $\{a_{k+1}, a_{k+2}, ..., a_j\}$. Если $i = k - 1$, то нет левого поддерева, а при $k = j$ нет правого поддерева. Для удобства мы будем трактовать T_{ii} как пустое дерево. Вес w_{ii} дерева T_{ii} равен q_i, а его стоимость c_{ii} равна 0.

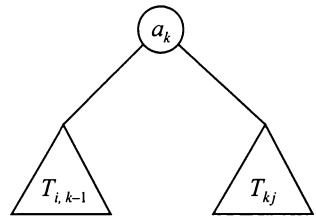


Рис. 4.8. Поддерево T_{ij}

В дереве T_{ij}, $i < j$, глубина каждой вершины в левом и правом поддеревьях на единицу больше глубины этих вершин в поддеревьях $T_{i,k-1}$ и T_{kj}. Таким образом, стоимость c_{ij} дерева T_{ij} можно выразить формулой

$$c_{ij} = w_{i,k-1} + p_k + w_{kj} + c_{i,k-1} + c_{kj} = w_{ij} + c_{i,k-1} + c_{kj}.$$

Здесь следует использовать значение k, которое минимизирует сумму $c_{i,k-1} + c_{kj}$. Таким образом, для нахождения оптимального дерева T_{ij} для каждого k, $i < k \le j$, необходимо вычислить стоимость дерева с корнем a_k, левым поддеревом $T_{1,k-1}$ и правым поддеревом T_{kj}, а затем выбрать дерево наименьшей стоимости. Приведем соответствующий алгоритм.

Алгоритм 4.2. Построение оптимального дерева бинарного поиска

Вход. Множество элементов $S = \{a_1, a_2, ..., a_n\}$, где $a_1 < a_2 < ... < a_n$, а также числа $q_0, q_1, ..., q_n$ и $p_1, p_2, ..., p_n$, где $1 \le i < n$; q_i — вероятность выполнения операции

MEMBER(a, S) для элементов a, таких, что $a_i < a < a_{i+1}$; q_0 — вероятность выполнения операции MEMBER(a, S) для $a < a_1$; q_n — вероятность выполнения операции MEMBER(a, S) для $a > a_n$; p_i, $1 \le i \le n$ — вероятность выполнения операции MEMBER(a_i, S).

Выход. Дерево бинарного поиска для S, обладающее наименьшей стоимостью.

Метод.

1. Для $0 \le i < j \le n$ вычисляются числа r_{ij} и c_{ij} в порядке возрастания разности $j - i$ с помощью алгоритма динамического программирования, приведенного на рис. 4.9.
2. После вычисления всех чисел r_{ij} вызывается процедура BUILDTREE(0, n) для рекурсивного построения оптимального дерева для T_{0n}. Процедура BUILDTREE приведена на рис. 4.10. □

```
      begin
1.        for i ← 0 until n do
              begin
2.                w_ii ← q_i;
3.                c_ii ← 0
              end;
4.        for l ← 1 until n do
5.            for i ← 0 until n - l do begin
                  begin
6.                    j ← i + l;
7.                    w_ij ← w_i,j-1 + p_j + q_j;
8.                    пусть m — значение k, i < k ≤ j,
                          для которого сумма c_i,k-1 + c_kj минимальна;
9.                    c_ij ← w_ij + c_i,m-1 + c_mj;
10.                   r_ij ← a_m
                  end
      end
```

Рис. 4.9. Алгоритм вычисления корней оптимальных поддеревьев

```
procedure BUILDTREE(i, j):
begin
      создать корень v_ij дерева T_ij;
      пометить v_ij меткой r_ij;
      пусть m — индекс числа r_ij (т.е. r_ij = a_m);
      if i < m - 1 then сделать дерево BUILDTREE(i, m - 1)
          левым поддеревом вершины v_ij;
      if m < j then сделать дерево BUILDTREE(m, j)
          правым поддеревом вершины v_ij
end
```

Рис. 4.10. Процедура для построения оптимального бинарного дерева

Пример 4.5. Рассмотрим четыре элемента $a_1 < a_2 < a_3 < a_4$ с $q_0 = 1/8$, $q_1 = 3/16$, $q_2 = q_3 = q_4 = 1/16$ и $p_1 = 1/4, p_2 = 1/8, p_3 = p_4 = 1/16$.

На рис. 4.11 приведены значения w_{ij}, r_{ij} и c_{ij}, вычисленные алгоритмом, приведенным на рис. 4.9. Для удобства записи значения w_{ij} и c_{ij} в этой таблице были умножены на 16.

$l = j - i$ ↓	$i \rightarrow$ 0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$
1	$w_{01} = 9$ $c_{01} = 9$ $r_{01} = a_1$	$w_{12} = 6$ $c_{12} = 6$ $r_{12} = a_2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = a_3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = a_4$	
2	$w_{02} = 12$ $c_{02} = 18$ $r_{02} = a_1$	$w_{13} = 8$ $c_{13} = 11$ $r_{13} = a_2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = a_4$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = a_1$	$w_{14} = 10$ $c_{14} = 18$ $r_{14} = a_2$			
4	$w_{04} = 16$ $c_{04} = 33$ $r_{04} = a_2$				

Рис. 4.11. Значения w_{ij}, r_{ij} и c_{ij}

Например, чтобы вычислить r_{14}, необходимо сравнить значения $c_{11} + c_{24}$, $c_{12} + c_{34}$ и $c_{13} + c_{44}$, равные (после умножения на 16) соответственно 8, 9 и 11. Таким образом, в строке 8 рис. 4.9 минимум стоимости достигается при $k = 2$, следовательно $r_{14} = a_2$.

Вычислив таблицу, показанную на рис. 4.11, построим дерево T_{04}, вызывая процедуру `BUILDTREE(0, 4)`. Полученное в результате дерево бинарного поиска показано на рис. 4.12. Его стоимость равна 33/16. □

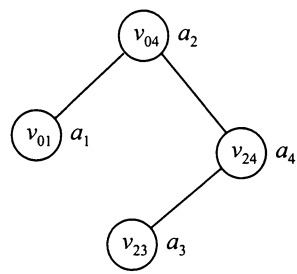


Рис. 4.12. Дерево минимальной стоимости

Теорема 4.2. Временная сложность алгоритма 4.2 построения оптимального дерева бинарного поиска имеет порядок $O(n^3)$.

Доказательство. Вычислив таблицу значений r_{ij}, мы строим по ней оптимальное дерево за время $O(n)$, вызывая процедуру BUILDTREE. Эта процедура вызывается только n раз, и каждый вызов занимает постоянное время.

Наибольшее время занимает алгоритм динамического программирования (рис. 4.9). Строка 8 находит значение k, минимизирующее $c_{i,k-1} + c_{kj}$ за время $O(j - i)$. Остальные шаги цикла в строках 5–10 занимают постоянное время. Внешний цикл в строке 4 выполняется n раз, а внутренний — не более n раз для каждой итерации внешнего цикла. Таким образом, суммарная сложность составляет $O(n^3)$.

Корректность алгоритма доказывается простой индукцией по $l = j - i$ и сводится к доказательству правильности вычисления чисел r_{ij} и c_{ij} в строках 9 и 10.

Чтобы показать, что оптимальное дерево правильно строится процедурой BUILDTREE, заметим, что если v_{ij} — корень поддерева для элементов $\{a_{i+1}, a_{i+2}, ..., a_j\}$. то его левый сын будет корнем оптимального дерева для элементов $\{a_{i+1}, a_{i+2}, ..., a_{m-1}\}$, где $r_{ij} = a_m$, а правый будет корнем оптимального дерева для элементов $\{a_{m+1}, a_{m+2}, ..., a_j\}$. Теперь очевидно, что процедура BUILDTREE(i, j) правильно строит оптимальное дерево для элементов $\{a_{i+1}, a_{i+2}, ..., a_j\}$. □

В алгоритме 4.2 поиск m в строке 8 рис. 4.9 можно ограничить диапазоном между позициями корней деревьев $T_{i,j-1}$ и $T_{i+1,j}$, гарантируя нахождение минимума. При такой модификации алгоритм 4.2 может находить оптимальное дерево за время $O(n^2)$.

4.6. Простой алгоритм объединения непересекающихся множеств

Рассмотрим обработку вершин в алгоритме для нахождения остовного дерева (пример 4.1). Возникающая здесь задача преобразования множеств имеет следующие особенности.

1. Объединяются только непересекающиеся множества.
2. Элементы множеств можно считать целыми числами от 1 до n.
3. Над множествами выполняются операции UNION и FIND.

В этом и следующем разделах мы рассмотрим структуры данных для задач такого вида. Пусть заданы n элементов, которые мы будем считать целыми числами 1, 2, ..., n. Предположим, что вначале каждый элемент образует одноэлементное

множество. Предположим, что необходимо выполнить последовательность операций UNION и FIND. Напомним, что операция UNION имеет вид UNION(A, B, C), указывающий, что два непересекающихся множества с именами A и B необходимо заменить их объединением C. В приложениях часто неважно, что выбирается в качестве имени множества, так что мы будем предполагать, что множества можно именовать целыми числами от 1 до n. Кроме того, будем предполагать, что никакие два множества никогда не имеют одинаковых имен.

Для решения этой задачи можно использовать несколько интересных структур данных. Здесь мы представим структуру данных, благодаря которой последовательность инструкций, содержащую до $n - 1$ операций UNION и $O(n \log n)$ операций FIND, можно выполнить за время $O(n \log n)$. В следующем разделе описывается структура данных, позволяющая обрабатывать последовательность из $O(n)$ операций UNION и FIND в худшем случае за время, почти линейное по n. Эти структуры данных позволяют выполнять последовательности операций INSERT, DELETE и MEMBER с такой же вычислительной сложностью.

Напомним, что в алгоритмах поиска, изложенных в разделах 4.2–4.5, предполагалось, что элементы принадлежат универсальному множеству, которое намного больше, чем множество выполняемых операций. В этом разделе универсальное множество будет приблизительно того же размера, что и последовательность выполняемых операций.

Вероятно, простейшей структурой данных для задач типа UNION–FIND служит массив, представляющий собой коллекцию множеств. Пусть R — массив размера n, а $R[i]$ — имя множества, содержащего элемент i. Поскольку имена множеств значения не имеют, можно вначале взять $R[i] = i, 1 \le i \le n$, чтобы отметить тот факт, что в начальный момент коллекция множеств имеет вид $\{\{1\}, \{2\}, ..., \{n\}\}$, а множество $\{i\}$ имеет имя i.

Операция FIND(i) выполняется путем вывода текущего значения $R[i]$. Следовательно, сложность выполнения операции FIND постоянна, а это лучшее, на что можно рассчитывать.

Для того чтобы выполнить операцию UNION(A, B, C), необходимо последовательно просмотреть массив R и заменить каждый его элемент, содержащий A или B, на C. Следовательно, сложность выполнения такой операции имеет порядок $O(n)$. Таким образом, последовательность из n операций UNION могла бы потребовать $O(n^2)$ времени, что нежелательно.

Этот упрощенный алгоритм можно улучшить несколькими способами. Во-первых, можно воспользоваться преимуществом связанных списков. Во-вторых, необходимо понять, что всегда эффективнее включать меньшее множество в большее. Для этого необходимо отличать "внутренние имена" множеств в массиве R от "внешних имен", упоминаемых в операциях UNION. И те и другие являются числами от 1 до n, но не обязательно одинаковыми.

Рассмотрим следующую структуру данных для этой задачи. Как и ранее, возьмем массив R, в котором элемент $R[i]$ содержит "внутреннее" имя множества, которому принадлежит элемент i. Однако теперь для каждого множества A мы построим связанный список LIST[A], содержащий его элементы. Для реализации этого связанного списка применяются два массива LIST и NEXT. Элемент LIST[A] представляет собой целое число j, указывающее, что j — первый элемент в множестве с внутренним именем A. Элемент NEXT[j] содержит следующий элемент множества A, NEXT[NEXT[j]] — следующий за ним элемент и т.д.

Кроме того, будем использовать массив SIZE, такой, что SIZE[A] — количество элементов в множестве A. Множества будут менять внутренние имена, а два массива INTERNAL_NAME и EXTERNAL_NAME будут устанавливать соответствие между внутренними и внешними именами. Иначе говоря, EXTERNAL_NAME[A] — это настоящее имя (задаваемое операциями UNION) множества с внутренним именем A. INTERNAL_NAME[j] — это внутреннее имя множества с внешним именем j. Внутренние имена — это имена, используемые в массиве R.

Пример 4.6. Пусть $n = 8$ и у нас есть коллекция из трех множеств {1, 3, 5, 7}, {2, 4, 8} и {6} с внешними именами 1, 2 и 3 соответственно. Структуры данных для этих трех множеств показаны на рис. 4.13, где 2, 3 и 1 — внутренние имена для 1, 2 и 3 соответственно. □

	R	NEXT
1	2	3
2	3	4
3	2	5
4	3	8
5	2	7
6	1	0
7	2	0
8	3	0

	LIST	SIZE	EXTERNAL_NAME
1	6	1	3
2	1	4	1
3	2	3	2

Множества (с внешними именами)		INTERNAL_NAME
1 = {1, 3, 5, 7}	1	2
2 = {2, 4, 8}	2	3
3 = {6}	3	1

Рис. 4.13. Структура данных для алгоритма UNION-FIND

Операция FIND(i) выполняется, как и ранее, путем обращения к элементу $R[i]$ для определения внутреннего имени множества, содержащего элемент i в данный

момент, а элемент EXTERNAL_NAME[$R[i]$] задает настоящее имя множества, которому принадлежит i.

Операция объединения UNION(I, J, K) выполняется следующим образом. (Номера строк см. на рис. 4.14.)

1. Определяем внутренние имена множеств I и J (строки 1 и 2).
2. Сравниваем относительные размеры множеств I и J с помощью массива SIZE (строки 3, 4).
3. Обходим список элементов меньшего множества и изменяем соответствующие элементы в массиве R на внутреннее имя большего множества (строки 5–9).
4. Включаем меньшее множество в большее, добавляя список элементов меньшего множества в начало списка для большего множества (строки 10–12).
5. Присваиваем полученному множеству внешнее имя K (строки 13 и 14).

Включая меньшее множество в большее, мы делаем время выполнения операции UNION пропорциональным мощности меньшего множества. Все детали приведены на рис. 4.14.

```
      procedure UNION(I, J, K):
      begin
1.        A ← INTERNAL_NAME[I];
2.        B ← INTERNAL_NAME[J];
3.        wlg предполагаем SIZE[A] ≤ SIZE[B]
4.            otherwise поменять местами A и B in
              begin
5.                ELEMENT ← LIST[A];
6.                while ELEMENT ≠ 0 do
                      begin
7.                        R[ELEMENT] ← B;
8.                        LAST ← ELEMENT;
9.                        ELEMENT ← NEXT[ELEMENT]
                      end;
10.               NEXT[LAST] ← LIST[B];
11.               LIST[B] ← LIST[A];
12.               SIZE[B] ← SIZE[A] + SIZE[B];
13.               INTERIMAL_NAME[K] ← B;
14.               EXTERNAL_NAME[B] ← K
              end
      end
```

Рис. 4.14. Реализация операции UNION

Пример 4.7. После выполнения операции UNION(1, 2, 4) структура данных, показанная на рис. 4.13, превратится в структуру данных, представленную на рис. 4.15. □

	R	NEXT
1	2	3
2	2	4
3	2	5
4	2	8
5	2	7
6	1	0
7	2	0
8	2	1

	LIST	SIZE	EXTERNAL_ NAME
1	6	1	3
2	2	7	4
3	–	–	–
4	–	–	–

Множества (с внешними именами)

3 = {6}

4 = {1, 2, 3, 4, 5, 7, 8}

	INTERNAL_ NAME
1	–
2	–
3	1
4	2

Рис. 4.15. Структура данных после выполнения алгоритма UNION

Теорема 4.3. С помощью алгоритма рис. 4.14 можно выполнить $n-1$ (максимально возможное число) операций UNION за $O(n\log n)$ шагов.

Доказательство. Поскольку сложность каждой операции UNION пропорциональна количеству перемещаемых элементов, то при равномерном распределении ее стоимости по элементам сложность перемещения любого элемента остается постоянной. Главное, что всякий раз, когда элемент перемещается из списка, он оказывается в списке, по крайней мере в два раза длиннее прежнего. Поэтому никакой элемент нельзя переместить более чем $\log n$ раз и, следовательно, сложность перемещения любого элемента составляет $O(\log n)$. Общие затраты времени равны сумме затрат времени на перемещение отдельных элементов. Таким образом, общая сложность имеет порядок $O(n\log n)$. □

Из теоремы 4.3 следует, что при выполнении m операций FIND и не более $n-1$ операций UNION общие затраты времени имеют порядок $O(\max(m, n\log n))$. Если m имеет порядок $n\log n$ или больше, то этот алгоритм действительно оптимален с точностью до постоянного множителя. Однако во многих ситуациях m имеет

порядок $O(n)$, а в таком случае, как мы увидим в следующем разделе, можно добиться лучшего времени, чем $O(\max(m, n \log n))$.

4.7. Древовидные структуры для задачи UNION-FIND

В предыдущем разделе мы представили структуру данных для задачи UNION–FIND, позволяющую выполнить $n - 1$ операций UNION и $O(n \log n)$ операций FIND за время $O(n \log n)$. В данном разделе описывается структура данных, которая состоит из деревьев, образующих лес, и предназначена для представления коллекции множеств. Эта структура данных позволит выполнить $O(n)$ операций UNION и FIND за почти линейное время.

Предположим, что каждое множество A представлено корневым неориентированным деревом T_A, вершинам которого поставлены в соответствие элементы из A. Корню этого дерева присвоено имя самого множества. Операцию UNION(A, B, C) можно выполнить, преобразуя корень дерева T_A в сына корня дерева T_B и заменяя имя в корне дерева T_B на C. Операцию FIND(i) можно выполнить, определив положение вершины, представляющей элемент i в некотором дереве T из леса, и пройдя путь из этой вершины в корень дерева T, в котором записано имя множества, содержащего i.

В такой схеме сложность слияния двух деревьев является постоянной. Однако сложность выполнения операции FIND(i) пропорциональна длине пути из вершины i в корень. Эта длина может быть равной $n - 1$. Поэтому сложность выполнения $n - 1$ операций UNION, за которыми следуют n операций FIND, может достигать $O(n^2)$. Например, вычислим сложность выполнения последовательности операций

```
UNION(1, 2, 2)
UNION(2, 3, 3)
 ...
UNION(n - 1, n, n)
FIND(1)
FIND(2)
 ...
FIND(n)
```

Выполнение $n - 1$ операций UNION приводит к дереву, изображенному на рис. 4.16. Сложность выполнения n операций FIND пропорциональна

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

Однако ее можно уменьшить, если поддерживать балансировку деревьев. Для этого можно в каждом дереве подсчитывать количество вершин и, объединяя два множества, всегда присоединять меньшее дерево к корню большего. Этот способ аналогичен включению меньшего множества в большее, который мы применяли в предыдущем разделе.

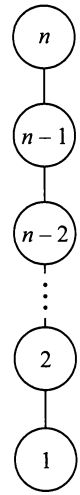


Рис. 4.16. Дерево после выполнения операций `UNION`

Лемма 4.1. Если при выполнении каждой операции `UNION` корень дерева с меньшим числом вершин (при равенстве вершин выбирается любое дерево) преобразуется в сына корня дерева с большим числом вершин, то высота дерева в лесу может достичь значения h, только если в нем не менее 2^h вершин.

Доказательство. Доказательство проведем индукцией по h. Для $h = 0$ утверждение верно, поскольку в каждом дереве есть по крайней мере одна вершина. Допустим, что утверждение верно для всех значений параметра, меньших $h \geq 1$. Пусть T — дерево высоты i с наименьшим числом вершин. Тогда оно должно было получиться в результате слияния деревьев T_1 и T_2, где T_1 — дерево высоты $h - 1$ и у него не больше вершин, чем у T_2. По предположению индукции дерево T_1 имеет не менее 2^{h-1} вершин, и, значит, дерево T_2 имеет не менее 2^{h-1} вершин, откуда следует, что дерево T имеет не менее 2^h вершин. □

Оценим худшее время выполнения последовательности из n операций `UNION` и `FIND` на основе структуры данных в виде леса, модифицированной так, что в операции `UNION` корень меньшего дерева становится сыном корня большего дерева. Никакое дерево не может иметь высоту больше, чем $\log n$. Следовательно,

порядок временной сложности выполнения операций UNION и FIND не превосходит $O(n\log n)$. Эта граница точна в том смысле, что существуют последовательности из n операций, занимающие время, пропорциональное $n\log n$.

Изложим еще одну модификацию этого алгоритма, называемую сжатием путей. Поскольку в общей сложности преобладает сложность операций FIND, попробуем уменьшить ее. Всякий раз, когда выполняется операция FIND(i), мы проходим путь от вершины i до корня r. Пусть i, v_1, v_2, ..., v_n, r — вершины на этом пути. Сделаем каждую из вершин i, v_1, v_2, ..., v_{n-1} сыном корня. На рис. 4.17, *б*, показан эффект операции FIND(i), выполненной над деревом, приведенном на рис. 4.17, *а*.

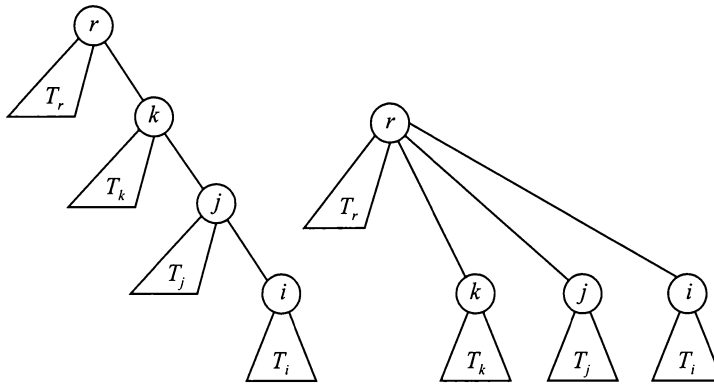


Рис. 4.17. Эффект сжатия пути

Вся процедура объединения деревьев для задачи UNION–FIND, включая сжатие путей, выражена в следующем алгоритме.

Алгоритм 4.3. Алгоритм быстрого объединения непересекающихся множеств.

Вход. Последовательность σ операций UNION и FIND, выполняемых на коллекции множеств, элементы которых представлены целыми числами от 1 до n. Пусть имена множеств также являются целыми числами от 1 до n и вначале множество с именем i состоит из единственного элемента — самого числа i.

Выход. Последовательность ответов на операции FIND из последовательности σ. Ответ на каждую операцию FIND необходимо получить перед просмотром очередной операции, входящей в σ.

Метод. Описание алгоритма состоит из трех частей: инициализация, выполнение операции FIND и выполнение операции UNION.

1. *Инициализация.* Для каждого элемента i, $1 \le i \le n$, создадим вершину v_i. Установим COUNT[v_i] = 1, NAME[v_i] = i и FATHER[v_i] = 0. Вначале каждая вершина v_i сама является деревом. Чтобы найти корень множества i, создадим такой массив ROOT, что ROOT[i] ссылается на v_i. Чтобы определить вершину, содержащую элемент i, создадим массив ELEMENT. Вначале ELEMENT[i] = v_i.
2. *Выполнение операции* FIND(i). Программа приведена на рис. 4.18. Начиная с вершины ELEMENT[i], идем по пути, ведущему к корню дерева, создавая список встретившихся вершин. По достижении корня выводится имя множества, а каждая вершина из пройденного пути делается сыном корня.
3. *Выполнение операции* UNION(i, j, k). С помощью массива ROOT находим корни деревьев, представляющих множества i и j. Затем делаем корень меньшего дерева сыном корня большего дерева. См. рис. 4.19. □

```
begin
    опустошить LIST;
    v ← ELEMENT[i];
    while FATHER[v] ≠ 0 do
        begin
            добавить v в LIST;
            v ← FATHER[v]
        end;
    comment теперь v — корень;
    print NAME[v];
    for для каждой вершины w в LIST do FATHER[w] ← v
end
```

Рис. 4.18. Выполнение инструкций FIND(i)

```
begin
    wlg при условии COUNT[ROOT[i]] ≤ COUNT[ROOT[j]]
        otherwise поменять местами i и j in
        begin
            LARGE ← ROOT[j];
            SMALL ← ROOT[i];
            FATHER[SMALL] ← LARGE:
            COUNT[LARGE] ← COUNT[LARGE] + COUNT[SMALL];
            NAME[LARGE] ← k;
            ROOT[k] ← LARGE
        end
end
```

Рис. 4.19. Выполнение инструкций UNION(i, j, k)

Покажем, что сжатие путей существенно ускоряет работу алгоритма. Чтобы оценить это улучшение, введем функции F и G:

$$F(0)=1,$$
$$F(i)=2^{F(i-1)} \text{ при } i>0.$$

Как показано на рис. 4.10, функция F растет чрезвычайно быстро. Функция $G(n)$ определяется как наименьшее число k, для которого $F(k) \geq n$. Функция G растет очень медленно. Действительно, $G(n) \leq 5$ для всех "практических" значений n, а именно для всех $n \leq 2^{65536}$.

n	$F(n)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65636}

Рис. 4.20. Некоторые значения функции F

Теперь покажем, что алгоритм 4.3 выполнит последовательность инструкций σ, состоящую из cn операций `UNION` и `FIND`, за время, не большее $c'nG(n)$, где c и c' — константы, причем c' зависит от c. Для простоты предположим, что выполнение операции `UNION` занимает "единицу времени", а операции `FIND`(i) — время, пропорциональное количеству вершин, лежащих на пути из вершины с меткой i в корень дерева, содержащего эту вершину.[5]

Определение. *Ранг вершины* относительно последовательности σ, состоящей из операций `UNION` и `FIND`, удобно определить следующим образом.

1. Удалить из σ операции `FIND`.
2. Выполнить получившуюся последовательность σ операций `UNION`.
3. Ранг вершины v — это ее высота в получившемся лесу. Докажем некоторые важные свойства ранга вершины.

[5]Таким образом, в данном случае "единица времени" — это некоторое постоянное количество шагов, выполняемых машиной RAM. Поскольку мы пренебрегаем постоянными множителями, то результаты о порядке величины также можно выражать через "единицы времени".

Лемма 4.2. Существует не более $n/2^r$ вершин ранга r.

Доказательство. По лемме 4.1 каждая вершина ранга r имеет по крайней мере 2^r потомков в лесу, построенном при выполнении последовательности σ'. Поскольку множества потомков любых двух различных вершин одинаковой высоты в лесу не пересекаются, а общее число непересекающихся множеств, содержащих не менее 2^r вершин, не превосходит $n/2^r$, то вершин ранга r не может быть больше $n/2^r$. □

Следствие. Ранг любой вершины не превосходит $\log n$.

Лемма 4.3. Если в какой-то момент выполнения последовательности σ вершина w оказалась подлинным потомком вершины v, то ее ранг меньше ранга вершины v.

Доказательство. Если в какой-то момент выполнения последовательности σ вершина w стала потомком вершины v, то w будет потомком вершины v и в лесу, построенном при выполнении последовательности σ'. Следовательно, высота вершины w должна быть меньше высоты вершины v, так что ранг вершины w меньше ранга вершины v. □

Разобьем ранги на *группы*. Отнесем ранг r к группе $G(r)$. Например, ранги 0 и 1 находятся в группе 0, ранг 2 — в группе 1, ранги 3 и 4 — в группе 2, ранги от 5 до 16 — в группе 3. Для $n > 1$ наибольший возможный ранг, т.е. $\lfloor \log n \rfloor$, попадает в группу $G(\lfloor \log n \rfloor) \le G(n) - 1$.

Оценим сложность выполнения последовательности σ, состоящей из cn операций `UNION` и `FIND`. Поскольку каждую операцию `UNION` можно выполнить за единицу времени, то все операции `UNION` из последовательности σ можно выполнить за время $O(n)$. Для того чтобы оценить сложность выполнения всех операций `FIND`, применим один важный "бухгалтерский" прием. За каждое выполнение операции `FIND` наложим штраф как на само выполнение, так и на некоторые перемещаемые вершины леса, встречающиеся на пути. Искомая общая сложность будет равна сумме всех штрафов, налагаемых на выполнение операций `FIND` и на перемещаемые вершины.

Штрафы за выполнение операции `FIND`(i) назначаются следующим образом. Пусть v — вершина на пути из вершины, представляющей элемент i, в корень дерева, содержащий элемент i.

1. Если v — корень или ранг вершины `FATHER`[i] отличается от ранга вершины v, то выполнение штрафуется на одну единицу времени.
2. Если ранги вершины v и ее отца принадлежат одной группе, то на вершину v налагается штраф в одну единицу времени.

По лемме 4.3 ранг вершин по мере прохождения вверх по пути монотонно возрастает. Поскольку различных групп рангов не больше, чем $G(n)$, то по правилу 1 никакое выполнение операции FIND не штрафуется более чем на $G(n)$ единиц времени. Если применяется правило 2, то вершина v будет перемещена и сделана сыном вершины с бо́льшим рангом, чем ее предыдущий отец. Если вершина v принадлежит группе $g > 0$, то она может перемещаться и штрафоваться не более чем $F(g) - F(g-1)$ раз, прежде чем приобретет отца из группы с более высоким рангом. Если ранг вершины принадлежит группе 0, то она может перемещаться не более одного раза, прежде чем приобретет отца из более высокой группы. С этого момента перемещение вершины v будет штрафоваться по правилу 1 сложностью выполнения операции FIND.

Чтобы оценить верхнюю границу штрафов, налагаемых на вершины, умножим наибольший штраф, который можно наложить на вершину с рангом из данной группы, на число вершин в этой группе и просуммируем по всем группам рангов. Пусть $N(g)$ — количество вершин, ранг которых принадлежит группе $g > 0$. Тогда по лемме 4.2

$$\begin{aligned} N(g) &\le \sum_{r=F(g-1)+1}^{F(g)} n/2^r \le \\ &\le \left(n/2^{F(g-1)+1}\right)\left[1+\frac{1}{2}+\frac{1}{4}+\ldots\right] \le \\ &\le n/2^{F(g-1)} \le \\ &\le n/F(g). \end{aligned}$$

Наибольший штраф на вершину с рангом из группы $g > 0$ не превосходит $F(g) - F(g-1)$. Следовательно, наибольший штраф для вершин, ранги которых принадлежат группе g, ограничен числом n. Это утверждение, очевидно, верно и для $g = 0$. Поскольку количество групп рангов не превышает $G(n)$, суммарный штраф, налагаемый на все вершины, не превосходит $nG(n)$. Следовательно, общее время, требуемое для выполнения cn операций FIND, состоит из времени, не большего $cnG(n)$, на которое оштрафовано выполнение этих операций, и времени, не большего $nG(n)$, на которое оштрафованы вершины. Таким образом, мы доказали следующую теорему.

Теорема 4.4. Пусть c — произвольная константа. Найдется другая константа c', зависящая от c и такая, что алгоритм 4.3 выполнит последовательность σ, состоящую из cn операций UNION и FIND на n элементах не более чем за $c'nG(n)$ единиц времени.

Доказательство. См. выше. □

В качестве упражнения предлагаем читателям показать, что если в последовательности σ наряду с операциями UNION и FIND допускаются основные операции INSERT и DELETE, то ее все равно можно выполнить за время $O(nG(n))$.

Неизвестно, насколько точной является оценка времени работы алгоритма 4.3, даваемая теоремой 4.4. Однако в оставшейся части раздела мы докажем теоретически интересный факт, что время работы алгоритма 4.3 нелинейно зависит от n. Для этого построим конкретную последовательность операций UNION и FIND, на которой алгоритм 4.3 работает нелинейное время.

Удобно ввести новую операцию над деревьями, которую мы будем называть *частичной операцией FIND* — сокращенно PF (partial find). Пусть T — дерево, в котором вершины v, v_1, v_2, ..., v_m, w образуют путь из вершины v к его предку w (w не обязательно корень). Операция PF(v, w) делает каждую из вершин v, v_1, v_2, ..., v_{m-1} сыном вершины w. Мы будем говорить, что эта операция PF имеет *длину* $m + 1$ (если $v = w$, то длина равна 0). На рис. 4.21, *б*, отражено влияние операции PF(v, w) на дерево, приведенное на рис. 4.21, *а*.

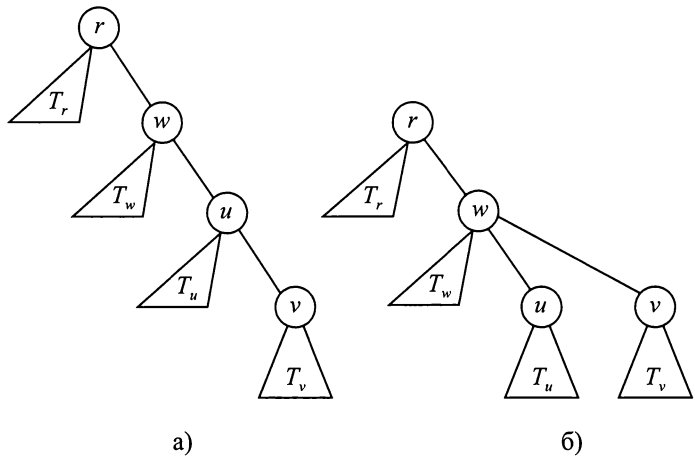


Рис. 4.21. Эффект частичной операции FIND

Рассмотрим последовательность σ, состоящую из операций UNION и FIND. Когда выполняется конкретная операция FIND из последовательности σ, мы находим вершину v в некотором дереве T и идем по пути из v в корень w. Теперь предположим, что выполняются только операции UNION из последовательности σ, а операции FIND игнорируются. В результате получается некоторый лес F. Результат данной операции FIND из последовательности σ можно обнаружить, если найти в F положение вершин v и w, которые должна была использовать эта

операция и затем выполнить операцию PF(v, w). Заметим, что вершина w может больше не быть корнем в лесе F.

Для получения нижней оценки времени работы алгоритма 4.3 исследуем его поведение на последовательности операций FIND, за которыми следуют операции PF. Эту последовательность можно заменить последовательностью операций UNION и FIND с тем же временем работы. Из описанных ниже специальных деревьев мы построим последовательности операций UNION и PF, на которых время работы алгоритма 4.3 более чем линейно.

Определение. Для $k \geq 0$ пусть $T(k)$ — дерево, в котором

1. глубина каждого листа равна k;
2. каждая вершина высоты h имеет 2^h сыновей, $h \geq 1$.

Таким образом, корень дерева $T(k)$ имеет 2^k сыновей, каждый из которых является корнем экземпляра дерева $T(k-1)$. На рис. 4.22 изображено дерево $T(2)$.

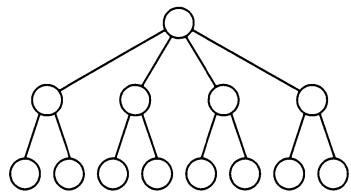


Рис. 4.22. Дерево $T(2)$

Лемма 4.4. Для каждого $k \geq 0$ можно построить дерево $T'(k)$, которое порождается некоторой последовательностью операций UNION и содержит в качестве подграфа дерево $T(k)$, причем по меньшей мере четверть вершин дерева $T'(k)$ являются листьями в дереве $T(k)$.

Доказательство. Доказательство проводится индукцией по k. Для $k = 0$ лемма тривиальна, поскольку $T(0)$ состоит из одной вершины. Чтобы построить $T'(k)$ для $k > 0$, сначала построим $2^k + 1$ экземпляров деревьев $T'(k-1)$. Образуем дерево $T'(k)$, выбрав один экземпляр $T'(k-1)$ и затем включив в него один за другим каждый из оставшихся экземпляров. Корень полученного дерева имеет (среди других) 2^k сыновей, каждый из которых является корнем дерева $T'(k-1)$.

Пусть $N'(k)$ — общее число вершин в дереве $T'(k)$ и $L(k)$ — число листьев в дереве $T(k)$. Тогда

$$N'(0) = 1,$$

$$N'(k) = (2^k + 1)N'(k-1) \text{ для } k \geq 1,$$

и

$$L(0) = 1,$$

$$L(k) = 2^k L(k-1) \text{ для } k \geq 1,$$

так что

$$\frac{L(k)}{N'(k)} = \frac{\prod_{i=1}^{k} 2^i}{\prod_{i=1}^{k} (2^i + 1)} = \frac{2}{3} \prod_{i=2}^{k} \frac{1}{1 + 2^{-i}} \text{ при } k \geq 1. \tag{4.3}$$

Заметим, что для $i \geq 2$ справедливо неравенство $\ln(1 + 2^{-i}) < 2^{-i}$, поэтому

$$\ln\left(\prod_{i=2}^{k} \frac{1}{1 + 2^{-i}}\right) \geq -\sum_{i=2}^{k} 2^{-i} \geq -\frac{1}{2}. \tag{4.4}$$

Используя (4.3) и (4.4), получаем

$$\frac{L(k)}{N'(k)} \geq \frac{2}{3} e^{-1/2} \geq \frac{1}{4}.$$

Лемма доказана. □

Построим последовательность операций `UNION` и `PF`, которая сначала создаст дерево $T'(k)$, а затем выполнит операцию `PF` на листьях подграфа $T(k)$. Сначала покажем, что для каждого $l > 0$ найдется такое число k, что можно выполнить операцию PF длины l последовательно на каждом листе дерева $T(k)$.

Определение. Пусть $D(c, l, h)$ — такое наименьшее значение k, что если заменить каждое поддерево в $T(k)$ с корнем высоты h любым деревом, имеющим l листьев и высоту не меньше 1, то на каждом листе полученного дерева можно будет выполнить операцию `PF` длины c.

Лемма 4.5. Значение $D(c, l, h)$ определено (т.е. конечно) для всех положительных c, l и h.

Доказательство. В доказательстве используется двойная индукция. Мы хотим доказать наше утверждение индукцией по c. Однако для доказательства этого утверждения для c, исходя из его истинности для $c - 1$, необходимо провести индукцию по l.

База индукции, т.е. случай $c = 1$, доказывается легко. $D(1, l, h) = h$ для всех l и h, поскольку операция `PF` длины 1 не перемещает никаких вершин.

Теперь, чтобы провести индукцию по c, предположим, что для всех l и h определено значение $D(c-1, l, h)$. Мы должны показать, что $D(c, l, h)$ определено для всех l и h. Это делается индукцией по l.

Для базы индукции по l докажем неравенство

$$D(c, 1, h) \le D(c-1, 2^{h+1}, h+1).$$

Заметим, что при $l = 1$ мы заменяем деревьями с единственным листом поддеревья в дереве $T(k)$ с корнями высоты h при некотором k. Пусть H — множество вершин высоты h в дереве $T(k)$. Очевидно, что в этом преобразованном дереве каждый лист является подлинным потомком единственного элемента множества H. Поэтому, если бы мы смогли выполнить операции PF длины $c-1$ на всех элементах множества H, то, конечно, смогли бы выполнить операции PF длины c на всех листьях.

Пусть $k = D(c-1, 2^{h+1}, h+1)$. По предположению индукции для c число k существует. Все вершины высоты $h+1$ в дереве $T(k)$ имеют по 2^{h+1} сыновей, причем все сыновья принадлежат множеству H. Если удалить из дерева $T(k)$ всех подлинных потомков вершин, входящих в H, то каждое поддерево с корнями на высоте $h+1$ будет заменено деревом высоты 1 с 2^{h+1} листьями. По определению D число $k = D(c-1, 2^{h+1}, h+1)$ достаточно велико, так что операции PF длины $c-1$ можно выполнить на всех листьях, т.е. элементах множества H.

Теперь, чтобы завершить индукцию по c, осталось сделать шаг индукции по l. В частности, покажем, что

$$D(c,l,h) \le D\left(c-1,\ 2^{D(c,l-1,h)\left(1+D(c,l-1,h)\right)/2},\ D(c,\ l-1,\ h)\right) \text{ при } l > 1. \qquad (4.5)$$

Чтобы доказать неравенство (4.5), положим $k = D(c, l-1, h)$ и обозначим через k' его правую часть. Нам необходимо найти способ заменить каждую вершину высоты h в $T(k')$ деревом с l листьями, а затем на каждом листе выполнить операцию PF длины c. Начнем с выполнения операций PF на $l-1$ листьях каждого подставляемого дерева. По предположению индукции по l это можно сделать.

Выполнив операции PF на $l-1$ листьях, находим, что l-й лист каждого подставляемого дерева теперь имеет отца, который не совпадает с отцом l-го листа любого другого подставляемого дерева. Множество таких отцов обозначим через F. Если мы смогли выполнить операции PF длины $c-1$ на этих отцах, то можем выполнить операции PF длины c на листьях. Пусть S —поддерево с корнем высоты k в дереве $T(k')$. Легко проверить, что S содержит $2^{k(k+1)/2}$ листьев дерева $T(k')$. Следовательно, после выполнения операций PF число вершин в S, которые также принадлежат множеству F, не превосходит $2^{k(k+1)/2}$. Следовательно, оставшееся множество можно рассматривать как произвольное дерево с $2^{k(k+1)/2}$ листьями, которые принадлежат F. По предположению индукции для c и l неравенство (4.5) справедливо. □

Теорема 4.5. Временная сложность алгоритма 4.3 больше cn для любой константы c.

Доказательство. Пусть c — такая константа, что алгоритм 4.3 выполнит любую последовательность, состоящую из $n - 1$ операций UNION и n операций FIND, не более чем за cn единиц времени. Выберем $d > 4c$ и вычислим $k = D(d, 1, 1)$. Построим $T'(k)$ с помощью последовательности операций UNION. Поскольку мы можем выполнить операции PF длины d на каждом листе вложенного дерева $T(k)$, листья которого занимают более четверти вершин дерева $T'(k)$, эти затраты времени на выполнение последовательности операций UNION и PF превысят cn единиц времени. Полученное противоречие доказывает теорему. □

4.8. Приложение и обобщения алгоритма UNION-FIND

Как мы видели, при решении задачи построения остовного дерева, описанной в примере 4.1, последовательность базовых операций UNION и FIND возникает естественным образом. В этом разделе мы рассмотрим несколько других задач, которые приводят к последовательностям операций UNION и FIND. В первой задаче необходимо осуществить вычисления в режиме офлайн, т.е. прочесть всю последовательность операций перед выдачей ответа.

Приложение 1. Задача MIN в режиме офлайн

Даны два типа операций: INSERT(i) и EXTRACT_MIN. Начнем с пустого множества S. Каждый раз, когда встречается операция INSERT(i), в множество S включается целое число i. Каждый раз, когда выполняется операция EXTRACT_MIN, находится и удаляется наименьший элемент множества S.

Пусть σ — последовательность операций INSERT и EXTRACT_MIN, в которой для каждого i, $1 \le i \le n$, операция INSERT(i) встречается не более одного раза. По данной последовательности σ необходимо найти последовательность целых чисел, удаляемых с помощью операции EXTRACT_MIN. Задача решается в офлайн-режиме, поскольку предполагается, что вся последовательность σ известна до того, как необходимо вычислить даже первый элемент выходной последовательности.

Задачу MIN в режиме офлайн можно решить следующим методом. Пусть k — число операций EXTRACT_MIN в последовательности σ. Запишем σ в виде $\sigma_1 E \sigma_2 E \sigma_3 E \ldots \sigma_k E \sigma_{k+1}$, где каждая подпоследовательность σ_j, $1 \le j \le k + 1$, состоит только из операций INSERT, а E обозначает операцию EXTRACT_MIN. Смоделируем последовательность σ с помощью алгоритма 4.3. Начальная последовательность множеств для работы алгоритма объединения строится так: если в последовательности σ_j встречается операция INSERT(i), то считаем, что множе-

ство с именем j, $1 \le j \le k+1$, содержит элемент i. Для того чтобы создать дважды связанный упорядоченный список для тех значений j, для которых существует множество с именем j, воспользуемся двумя массивами: PRED и SUCC. Вначале PRED[j] = $j-1$ для $1 \le j \le k+1$ и SUCC[j] = $j+1$ для $0 \le j \le k$. Затем выполняется программа, приведенная на рис. 4.23.

```
for i ← 1 until n do
    begin
        j ← FIND(i);
        if j ≤ k then
            begin
                print i "удаляется j-й операцией EXTRACT_MIN":
                UNION(j, SUCC[j], SUCC[j]);
                SUCC[PRED[j]] ← SUCC[j];
                PRED[SUCC[j]] ← PRED[j];
            end
    end
```

Рис. 4.23. Программа решения задачи MIN в режиме офлайн

Нетрудно убедиться, что время выполнения этой программы ограничено временем работы алгоритма объединения множеств. Следовательно, задача MIN в офлайн-режиме имеет временную сложность $O(nG(n))$.

Пример 4.8. Рассмотрим последовательность операций σ = 4 3 E 2 E 1 E, где j означает INSERT(j), а E — EXTRACT_MIN. Тогда σ_1 = 4 3, σ_2 = 2, σ_3 = 1 и σ_4 — пустая последовательность. Начальная структура данных представляет собой последовательность множеств

$$1 = \{3, 4\},\ \ 2 = \{2\},\ \ 3 = \{1\},\ \ 4 = \varnothing.$$

При первом выполнении цикла **for** выясняется, что FIND(1) = 3. Следовательно, результатом третьей операции EXTRACT_MIN в последовательности σ является 1. Последовательность множеств превращается в

$$1 = \{3, 4\},\ \ 2 = \{2\},\ \ 4 = \{1\}.$$

В этот момент SUCC[2] = 4 и PRED[4] = 2, поскольку множества с именами 3 и 4 были объединены в одно множество с именем 4.

При следующем прохождении цикла с i = 2 выясняется, что FIND(2) = 2. Таким образом, результатом второй операции EXTRACT_MIN будет 2. Множество с именем 2 объединяется со следующим множеством (с именем 4) и получается последовательность множеств

$$1 = \{3, 4\},\ \ 4 = \{1, 2\}.$$

Два последних прохождения устанавливают, что результатом операции EXTRACT_MIN является 3 и что 4 никогда не извлекается. □

Следующее приложение — задача определения глубины. Она возникает, в частности, при отождествлении идентификаторов в программе на языке ассемблера. Во многих языках ассемблера есть инструкции, объявляющие, что два идентификатора представляют одну и ту же ячейку памяти. Как только ассемблер обнаруживает инструкцию, отождествляющую два идентификатора α и β, он должен найти два множества S_α и S_β идентификаторов, эквивалентных соответственно α и β, и заменить эти два множества их объединением. Очевидно, задачу можно смоделировать последовательностью операций UNION и FIND.

Однако, если глубже проанализировать эту задачу, можно по-другому применить структуры данных из предыдущего раздела. Каждому идентификатору соответствует ячейка таблицы символов, и если несколько идентификаторов являются эквивалентными, то данные о них удобно хранить только в одной ячейке таблицы символов. Это значит, что для каждого множества эквивалентных идентификаторов существует *начало отсчета*, т.е. место в таблице символов, где хранится информация об этом множестве, и каждый элемент этого множества имеет *смещение* от начала. Для того чтобы найти позицию идентификатора в таблице символов, необходимо прибавить его смещение к началу отсчета множества, которому принадлежит данный идентификатор. Однако когда два множества идентификаторов отождествляются, смещение необходимо изменить. Эта задача корректировки смещений в абстрактном виде решается в приложении 2.

Приложение 2. Задача определения глубины

Дана последовательность операций двух типов: LINK(v, r) и FIND_DEPTH(v). Начнем с n неориентированных корневых деревьев, каждое из которых состоит из единственной вершины i, $1 \le i \le n$. Операция LINK(v, r), где r — корень дерева, а v — вершина другого дерева, делает корень r сыном вершины v. Условия, что v и r принадлежат различным деревьям и что r — корень, гарантируют, что получаемый в результате граф также будет лесом. Операция FIND_DEPTH(v) состоит в том, чтобы найти и выдать текущую глубину вершины v.

Если при работе с лесом использовать обычное представление в виде списков смежности и вычислять глубину вершин очевидным образом, то сложность алгоритма составит $O(n^2)$. Вместо этого мы представим исходный лес в другом виде, который будем называть D-лесом. Его единственное предназначение — быстро вычислять глубины. Каждой вершине в D-лесу приписывается целочисленный вес так, чтобы сумма весов вдоль пути в D-лесу от вершины v к корню равнялась глубине вершины v в исходном лесу. Для каждого дерева в D-лесу хранится счетчик числа вершин в этом дереве.

Вначале D-лес состоит из n деревьев, каждое из которых содержит единственную вершину, соответствующую целому числу i, $1 \le i \le n$. Начальный вес каждой вершины равен нулю.

Для выполнения операции FIND_DEPTH(v) мы проходим путь из вершины v в корень r. Пусть $v_1, v_2, \ldots, v_k$ — вершины на этом пути ($v_1 = v$, $v_k = r$). Тогда

$$\mathtt{DEPTH}(v) = \sum_{i=1}^{k} \mathtt{WEIGHT}[v_i].$$

Кроме того, применим сжатие путей. Каждая вершина v_i, $1 \le i \le k-2$, становится сыном корня r. Чтобы сохранялось сформулированное выше свойство весов, новый вес вершины v_i должен равняться $\sum_{j=i}^{k-1} \mathtt{WEIGHT}[v_j]$ для $1 \le i < k$. Поскольку новые веса можно вычислить за время $O(k)$, операция FIND_DEPTH имеет ту же временную сложность, что и операция FIND.

Чтобы выполнить операцию LINK(v, r), соединим деревья, содержащие вершины v и r, снова включая меньшее дерево в большее. Пусть T_v и T_r — деревья в D-лесу, содержащие v и r соответственно, а v' и r' — их корни. Деревья в D-лесу не обязательно изоморфны деревьям в исходном лесу, так что, в частности, r может не быть корнем дерева T_r. Пусть COUNT(T) — количество вершин в дереве T. Рассмотрим отдельно два варианта.

Вариант 1. $\mathtt{COUNT}(T_r) \le \mathtt{COUNT}(T_v)$ Делаем r' сыном вершины v'. Необходимо также скорректировать вес старого корня r' дерева T так, чтобы глубина каждой вершины w в T_r правильно вычислялась при прохождении пути из w в v' в объединенном дереве. Для этого выполним операцию FIND_DEPTH(v), а затем

$$\mathtt{WEIGHT}[r'] \leftarrow \mathtt{WEIGHT}[r'] - \mathtt{WEIGHT}[v'] + \mathtt{DEPTH}(v) + 1.$$

Таким образом, глубина каждой вершины дерева T_r увеличивается на глубину вершины v плюс 1. Наконец, полагаем счетчик числа вершин объединенного дерева равным сумме счетчиков для T_r и T_v.

Вариант 2. $\mathtt{COUNT}(T_v) < \mathtt{COUNT}(T_r)$. Здесь вычисляется DEPTH(v), затем вершина v' становится сыном вершины r' и выполняются следующие операции:

$$\mathtt{WEIGHT}[r'] \leftarrow \mathtt{WEIGHT}[r'] + \mathtt{DEPTH}(v) + 1;$$

$$\mathtt{WEIGHT}[v'] \leftarrow \mathtt{WEIGHT}[v'] - \mathtt{WEIGHT}[r'];$$

$$\mathtt{COUNT}[T_r] \leftarrow \mathtt{COUNT}[T_v] + \mathtt{COUNT}[T_r].$$

Итак, $O(n)$ операций LINK и FIND_DEPTH можно выполнить за время $O(nG(n))$.

Приложение 3. Эквивалентность конечных автоматов

Детерминированным конечным автоматом называется машина, распознающая строки символов. Она имеет входную ленту, разбитую на ячейки, головку, считывающую данные с входной ленты, и управляющее устройство с конечным числом состояний (рис. 4.24). Конечный автомат M можно представить в виде пятерки (S, I, δ, s_0, F), где

1. S — множество состояний *управляющего устройства*.
2. I — *входной алфавит* (каждая ячейка входной ленты содержит символ из I).
3. δ — отображение из $S \times I$ в S (если $\delta(s, a) = s'$, то каждый раз, когда M находится в состоянии s, а считывающая головка указывает на символ a, конечный автомат M сдвигает считывающую головку вправо и переходит в состояние s').
4. s_0 — выделенное состояние в S, называемое *начальным*.
5. F — подмножество множества состояний S, называемое множеством *допускающих* (или *заключительных*) состояний.

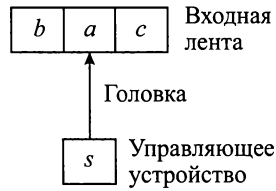


Рис. 4.24. Конечный автомат

Пусть I^* — множество строк (слов) конечной длины, состоящих из символов алфавита I. Пустая строка ε также принадлежит этому множеству. Продолжим функцию δ до отображения из $S \times I^*$ в S.

1. $\delta(S, \varepsilon) = S$.
2. $\delta(S, xa) = \delta(\delta(s, x), a)$ для всех $x \in I^*$ и $a \in I$.

Входная строка x *допускается* автоматом M, если $\delta(s_0, x) \in F$. *Языком* $L(M)$, *допускаемым автоматом* M, называется множество всех строк, допускаемых M. Более подробное введение в теорию конечных автоматов изложено в разделе 9.1.

Два состояния s_1 и s_2 считаются *эквивалентными*, если для каждого $x \in I^*$ состояние $\delta(s_1, x)$ является допускающим тогда и только тогда, когда $\delta(s_2, x)$ является допускающим состоянием.

Два конечных автомата M_1 и M_2 считаются *эквивалентными*, если $L(M_1) = L(M_2)$. Мы покажем, что с помощью алгоритма `UNION-FIND` можно распознать эквива-

лентность двух конечных автоматов $M_1 = (S_1, I, \delta_1, s_1, F_1)$ и $M_2 = (S_2, I, \delta_2, s_2, F_2)$ за $O(nG(n))$ шагов, где $n = ||S_1|| + ||S_2||$.

Отношение эквивалентности двух состояний имеет важное свойство: если два состояния s и s' эквивалентны, то для всех входных символов a состояния $\delta(s, a)$ и $\delta(s', a)$ также эквивалентны. Кроме того, благодаря наличию пустой строки ни одно допускающее состояние не может оказаться эквивалентным недопускающему. Следовательно, если начальные состояния s_1 и s_2 автоматов M_1 и M_2 эквивалентны, то можно вывести другие пары эквивалентных состояний. Если в одну из таких пар попадет допускающее состояние вместе с недопускающим, то s_1 и s_2 были неэквивалентными. Если же это не произойдет, то верно обратное утверждение.

Чтобы узнать, эквивалентны ли два конечных автомата $M_1 = (S_1, I, \delta_1, s_1, F_1)$ и $M_2 = (S_2, I, \delta_2, s_2, F_2)$, выполним следующие действия.

1. С помощью программы, приведенной на рис. 4.25, найдем все множества состояний, которые должны быть эквивалентными, если эквивалентны состояния s_1 и s_2. Список `LIST` содержит такие пары состояний (s, s'), что s и s' эквивалентны, а следующие за ними состояния $(\delta(s, a), \delta(s', a))$ еще не рассматривались. Вначале список `LIST` содержит только пару (s_1, s_2). Чтобы найти множества эквивалентных состояний, программа применяет алгоритм объединения непересекающихся множеств. Список `COLLECTION` представляет собой некоторое семейство множеств. Вначале каждое состояние из $S_1 \cup S_2$ образует одноэлементное множество. (Без потери общности можно считать, что множества S_1 и S_2 не пересекаются.) Затем каждый раз, когда s и s' оказываются эквивалентными, содержащие их множества A и A', входящие в `COLLECTION`, объединяются, и новое множество получает имя A.

2. После выполнения этой программы множества, входящие в `COLLECTION`, представляют разбиение множества $S_1 \cup S_2$ на блоки состояний, которые должны быть эквивалентными. Автоматы M_1 и M_2 эквивалентны тогда и только тогда, когда никакой блок не содержит допускающего состояния вместе с недопускающим.

Временная сложность этого алгоритма (как функция от числа состояний $n = ||S_1|| + ||S_2||$) определяется в основном временной сложностью алгоритма объединения множеств. Операций `UNION` может быть не более $n - 1$, поскольку каждая такая операция уменьшает на единицу число множеств, входящих в `COLLECTION`, а вначале их было только n. Количество операций `FIND` пропорционально числу пар, помещенных в `LIST`. Это число не больше $n||I||$, поскольку любая пара, кроме начальной (s_1, s_2), попадает в `LIST` только после выполнения операции `UNION`. Следовательно, при постоянном размере входного алфавита временная сложность распознавания эквивалентности конечных автоматов M_1 и M_2 составляет $O(nG(n))$.

```
begin
    LIST ← (s1, s2);
    COLLECTION ← ∅;
    for для каждого состояния s ∈ S1 ∪ S2 do добавить {s}
                                в COLLECTION;
    comment мы инициализировали исходные множества
        для каждого состояния из S1 ∪ S2;
    while существует пара (s, s'), входящая в LIST do
        begin
            удалить (s, s') из LIST;
            пусть A и A' обозначают FIND(s) и FIND(s') соответственно;
                if A ≠ A' then
                    begin
                        UNION(A, A', A);
                        for все a ∈ I do
                            добавить (δ(s, a), δ(s', a)) в LIST
                    end
        end
end
```

Рис. 4.25. Алгоритм для распознавания множеств эквивалентных состояний в предположении, что состояния s_1 и s_2 эквивалентны

4.9. Схемы сбалансированных деревьев

При решении ряда важных классов задач, аналогичных задаче UNION–FIND, приходится вернуться к способам, при которых последовательность из n операций выполняется за время $O(n\log n)$ в худшем случае. Один из таких классов состоит из задач, в которых последовательность, состоящая из операций MEMBER, INSERT и DELETE, выполняется над элементами, мощность базового множества которых намного больше количества реально используемых элементов. В этом случае нельзя выбирать элемент, непосредственно индексируя массив указателей. Необходимо применять хеширование или дерево бинарного поиска.

Если n элементов уже вставлены, то время выполнения одной операции выбора в методе хеширования является постоянным в среднем и $O(n)$ в худшем случае. В дереве бинарного поиска среднее время одной операции выбора имеет порядок $O(\log n)$, но, если множество имен не является статичным, временная сложность в худшем случае тоже может оказаться высокой. Если просто добавлять имена в дерево, не поддерживая его сбалансированность, то можно в результате получить дерево с n вершинами, глубина которого близка к n. Тогда в худшем случае временная сложность операции в дереве бинарного поиска будет составлять $O(n)$. Методом, изложенным в этом разделе, можно уменьшить временную сложность в худшем случае до $O(\log n)$ шагов на операцию.

Другой класс задач, требующих $O(n \log n)$ времени, состоит из задач выполнения последовательности. состоящей из n операций INSERT, DELETE и MIN, в онлайн-режиме. Еще один, третий, класс задач возникает, когда нужно представлять упорядоченные списки и уметь конкатенировать и расщеплять их.

В этом разделе описываются методы выполнения в онлайн-режиме последовательностей, содержащих важные подмножества семи основных операций на множествах, введенных в разделе 4.1. Структурой данных, лежащей в основе метода, является *сбалансированное дерево*, под которым мы понимаем дерево с высотой, приблизительно равной логарифму числа его вершин.

Вначале сбалансированное дерево строится легко. Однако в процессе выполнения последовательности операций INSERT и DELETE трудно предотвратить нарушение баланса. Например, если периодически не осуществлять балансировку, то при выполнении последовательности операций DELETE, которые удаляют вершины только из левой части дерева, получится дерево, смещенное вправо.

Существует много способов балансировки дерева. Некоторые из них оставляют структуру дерева довольно гибкой, так что число вершин в дереве высоты h может изменяться от 2^h до 2^{h+1} или 3^h. Такие методы позволяют, по меньшей мере, удвоить число вершин в поддереве, прежде чем что-то менять выше его корня.

Мы обсудим два метода такого вида, называемые методами 2-3-деревьев и AVL-деревьев. Алгоритмы, работающие с 2-3-деревьями, интуитивно понятнее, поэтому мы обсудим их в первую очередь, а похожие на них алгоритмы работы с AVL-деревьями вынесем в упражнения.

Определение. *2-3-деревом* называется дерево, в котором каждая вершина, не являющаяся листом, имеет двух или трех сыновей, а длины всех путей из корня в листья одинаковы. Заметим, что дерево, состоящее из единственной вершины, является 2-3-деревом. На рис. 4.26 приведены два 2-3-дерева с шестью листьями.

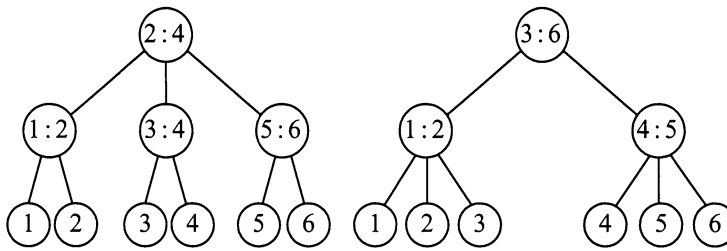


Рис. 4.26. 2-3-деревья

Следующая лемма устанавливает связь между количеством вершин и листьев в 2-3-дереве и его высотой.

Лемма 4.6. Пусть T — 2-3-дерево высоты h. Количество вершин дерева T лежит в диапазоне от $2^{h+1} - 1$ до $(3^{h+1} - 1)/2$, а число листьев — от 2^h и 3^h.

Доказательство. Простая индукция по h. □

Линейно упорядоченное множество S можно представить в виде 2-3-дерева, присвоив листьям дерева его элементы. Обозначим элемент, присвоенный листу l, через $E[l]$. Существуют два основных метода присваивания элементов листьям, применение которых зависит от рассматриваемой задачи.

Если базовое множество элементов гораздо больше, чем количество реально используемых элементов, и дерево будет использоваться как словарь, то, вероятно, лучше присваивать элементы в порядке возрастания слева направо. В каждой вершине v, не являющейся листом, нам потребуется еще два элемента: $L[v]$ — наибольший элемент множества S в поддереве, корнем которого служит самый левый сын вершины v, и $M[v]$ — наибольший элемент множества S в поддереве, корнем которого служит второй сын вершины v (см. рис. 4.26). Значения L и M, присвоенные вершинам, позволяют искать элемент, начиная с корня, способом, напоминающим бинарный поиск. Время обнаружения произвольного элемента пропорционально высоте дерева, поэтому операцию `MEMBER` на множестве из n элементов можно выполнить за время $O(\log n)$, если представить его в виде 2-3-дерева такого вида.

Во втором методе присваивания элементов листьям порядок выполнения этих операций ничем не ограничен. Этот метод особенно полезен для реализации операции `UNION`. Однако для выполнения операций типа `DELETE` необходим механизм для определения положения листа, представляющего данный элемент. Если элементами множества являются целые числа из фиксированного диапазона, например, от 1 до n, то лист, представляющий элемент i, можно найти с помощью i-й ячейки некоторого массива. Если же элементы рассматриваемых множеств принадлежат некоторому большому базовому множеству, то лист, представляющий элемент i, можно найти с помощью вспомогательного словаря.

Рассмотрим следующие наборы операций.

1) `INSERT`, `DELETE`, `MEMBER`.

2) `INSERT`, `DELETE`, `MIN`.

3) `INSERT`, `DELETE`, `UNION`, `MIN`.

4) `INSERT`, `DELETE`, `FIND`, `CONCATENATE`, `SPLIT`.

Структуру данных, обеспечивающую выполнение операций из множества 1, будем называть *словарем* (dictionary), из множества 2 — *очередью с приоритетами* (priority queue), из множества 3 — *сливаемой кучей* (mergeable heap), из множества 4 — *сцепляемой очередью* (concatenable queue).

Покажем, что 2-3-деревья могут служить для реализации словарей, очередей с приоритетами, сцепляемых очередей и сливаемых деревьев, применение которых обеспечивает выполнение n операций за время $O(n \log n)$. Описанные здесь методы достаточно мощные, чтобы выполнять последовательности, составленные из любого совместимого подмножества семи операций, перечисленных в начале главы. Единственная несовместимость состоит в том, что операция `UNION` предполагает неупорядоченное множество, а `SPLIT` и `CONCATENATE` предполагают наличие порядка.

4.10. Словари и очереди с приоритетами

В этом разделе мы изучим основные операции, необходимые для реализации словарей и очередей с приоритетами. На протяжении всего раздела будем предполагать, что элементы присвоены листьям 2-3-дерева в порядке слева направо и в каждой внутренней вершине v определены функции $L[v]$ и $M[v]$, введенные в предыдущем разделе.

Чтобы вставить новый элемент a в 2-3-дерево, необходимо найти позицию для нового листа l, который будет содержать элемент a. Для этого необходимо выполнить поиск элемента a в дереве. Если дерево содержит более одного элемента, то поиск элемента a закончится в вершине f, имеющей двух или трех сыновей, которые являются листьями.

Если вершина f имеет только два листа, l_1 и l_2, то l делаем сыном вершины f. Если $a < E[l_1]$,[6] то l делаем самым левым сыном вершины f и полагаем $L[f] = a$ и $M[f] = E[l_1]$; если $E[l_1] < a < E[l_2]$, то l делаем средним сыном вершины f и полагаем $M[f] = a$; если $E[l_2] < a$, то l делаем третьим сыном вершины f. Значения L и M на некоторых прямых предках вершины f, возможно, придется изменить.

Пример 4.9. Если в 2-3-дерево, показанное на рис. 4.27, *а*, вставляется элемент 2, то получается 2-3-дерево, изображенное на рис. 4.27, *б*. □

[6] $E[v]$ — элемент, хранящийся в листе v.

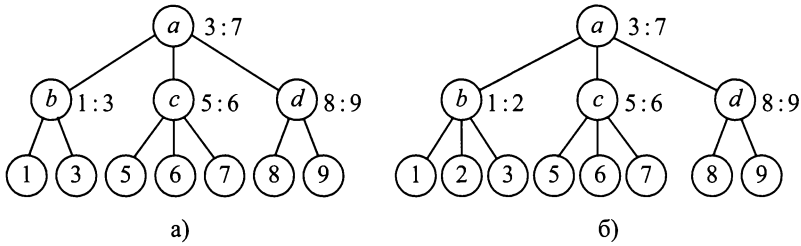


Рис. 4.27. Вставка в 2-3-дерево: (*а*) дерево до вставки; (*б*) дерево после вставки элемента 2

Теперь допустим, что у вершины f уже есть три листа: l_1, l_2 и l_3. Сделаем l соответствующим сыном вершины f. Теперь вершина f имеет четырех сыновей. Чтобы сохранить 2-3-свойство, создадим новую вершину g. Затем сделаем g братом вершины f, сделав его сыном отца вершины f. Если отец вершины f имел двух сыновей, то на этом процесс останавливается. Если трех, то необходимо рекурсивно повторять эту процедуру до тех пор, пока у всех вершин в дереве останется не более трех сыновей. Если у корня окажется четыре сына, создадим новый корень с двумя новыми сыновьями, каждый из которых будет иметь в качестве двух своих сыновей двух из четырех сыновей старого корня.

Пример 4.10. Если в 2-3-дерево на рис. 4.27, *а*, вставляется элемент 4, то новый лист с меткой 4 необходимо сделать самым левым сыном вершины c. Поскольку у вершины c уже есть три сына, создаем новую вершину c'. Затем делаем листья 4 и 5 сыновьями вершины c, а листья 6 и 7 — сыновьями вершины c'. Теперь делаем вершину c' сыном вершины a. Однако поскольку у вершины a уже есть три сына, создаем новую вершину a'. Делаем вершины b и c сыновьями старой вершины a, а вершины c' и d — сыновьями новой вершины a'. Наконец, создаем новый корень r и делаем вершины a и a' ее сыновьями. Полученное дерево изображено на рис. 4.28. □

Алгоритм 4.4. Вставка нового элемента в 2-3-дерево

Вход. Непустое 2-3-дерево T с корнем r и новый элемент $a \notin T$.

Выход. Преобразованное 2-3-дерево с новым листом, которому присвоен элемент a.

Метод. Предполагается, что дерево T содержит хотя бы один элемент. Чтобы упростить описание алгоритма, опустим детали корректировки L и M в разных вершинах

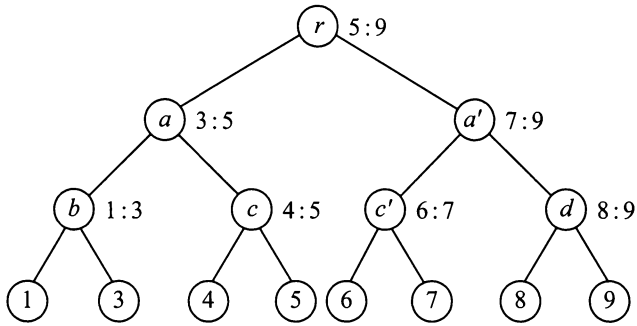


Рис. 4.28. Дерево с рис. 4.27, *а* после вставки элемента 4

1. Если дерево T состоит из единственной вершины l с меткой b, создаем новый корень r'. Образуем новую вершину v с меткой a. Делаем вершины l и v сыновьями корня r', причем l будет левым сыном, если $b < a$, и правым в противном случае.
2. Если дерево T содержит более одной вершины, положим f = SEARCH(a, r). Процедура SEARCH приведена на рис. 4.29. Создаем новый лист l с меткой a. Если у вершины f есть два сына с метками b_1 и b_2, то делаем вершину l соответствующим сыном вершины f: Вершина l будет левым сыном, если $a < b_1$, средним, если $b_1 < a < b_2$, и правым, если $b_2 < a$. Если у вершины f три сына, делаем вершину l соответствующим сыном вершины f, а затем, чтобы вставить в дерево T вершину f и его четырех сыновей, вызываем процедуру ADDSON(f). Процедура ADDSON приведена на рис. 4.30. Чтобы учесть наличие вершины a, корректируем значения L и M вдоль пути из вершины a в корень.[7] Другие очевидные изменения, корректирующие значения L и M, производятся процедурой ADDSON (предлагаем их в качестве упражнения). □

```
procedure SEARCH(a, r):
if любой сын вершины r является листом then return r
else
    begin
        пусть s_i — i-й сын вершины r;
        if a ≤ L[r] then return SEARCH(a, s_1)
        else
            if r имеет двух сыновей или a ≤ M[r] then return
                                                  SEARCH(a, s_2)
            else return SEARCH(a, s_3)
    end
```

Рис. 4.29. Процедура SEARCH

[7] Достаточно пройти путь от вершины a до такой вершины, что a — не наибольший элемент в своем поддереве.

```
procedure ADDSON(v):
begin
    создать новую вершину v';
    сделать двух самых правых сыновей вершины v левым и правым
        сыновьями вершины v';
    if вершина v не имеет отца then
        begin
            создать новый корень r;
            сделать вершину v левым, а v' — правым сыном корня r
        end
    else
        begin
            пусть f — отец вершины v;
            сделать v' сыном вершины f, расположенным
                непосредственно справа от v;
            if теперь вершина f имеет четыре сына then ADDSON(f)
        end
end
```

Рис. 4.30. Процедура ADDSON

Теорема 4.6. Алгоритм 4.4 вставляет новый элемент в 2-3-дерево с n листьями за время, не превосходящее $O(\log n)$. Кроме того, этот алгоритм сохраняет порядок исходных листьев и структуру 2-3-дерева.

Доказательство. Прямая индукция по числу вызовов процедуры SEARCH показывает, что новый лист становится сыном правильной вершины. Порядок исходных листьев не изменяется. В силу леммы 4.6 высота 2-3-дерева с n листьями не превосходит $\log n$. Поскольку процедура ADDSON(v) рекурсивно вызывает себя только на отце вершины v, может произойти не более $\log n$ рекурсивных вызовов. Каждый вызов процедуры ADDSON занимает постоянное время, так что общее время не превосходит $O(\log n)$. □

Элемент a можно удалить из 2-3-дерева способом, по существу обратным к вставке. Пусть a — метка листа l. Рассмотрим отдельно три варианта.

Вариант 1. Если l — корень, удаляем его. (В этом случае элемент a был единственным элементом в дереве.)

Вариант 2. Если l — сын вершины, имеющей трех сыновей, удаляем ее.

Вариант 3. Если l — сын вершины f, имеющей двух сыновей s и l, то существуют две возможности:

а) f — корень; удаляем вершины l и f и делаем корнем второго сына s;

б) f — не корень. Допустим, что вершина f имеет брата слева от себя.[8]

Вариант, когда брат находится справа, рассматривается аналогично. Если у вершины g только два сына, делаем вершину s самым правым сыном вершины g, удаляем l и рекурсивно вызываем процедуру удаления, чтобы удалить вершину f. Если у вершины g три сына, то самого правого сына делаем левым сыном вершины f и удаляем вершину l.

Пример 4.11. Из 2-3-дерева на рис. 4.28 удалим элемент 4. Лист с меткой 4 является сыном вершины c, у которой два сына, поэтому делаем лист с меткой 5 самым правым сыном вершины b, удаляем лист с меткой 4 и затем рекурсивно удаляем вершину c.

Вершина c — сын вершины a, у которой два сына. Вершина a' — правый брат вершины a, поэтому по симметрии делаем b самым левым сыном вершины a', удаляем вершину c и затем рекурсивно удаляем вершину a.

Вершина a — сын корня. Применяя вариант 3a, делаем вершину a' корнем остающегося дерева (рис. 4.31). □

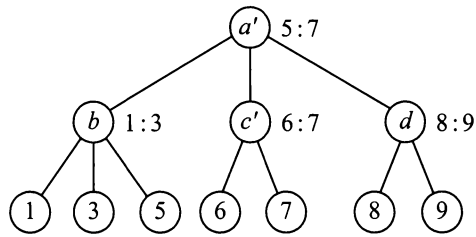


Рис. 4.31. Дерево, представленное на рис. 4.28, после удаления элемента 4

Формальную детализацию процесса удаления вершин, а также доказательство того, что на 2-3-дереве с n листьями его можно выполнить не более чем за $O(\log n)$ шагов, оставляем в качестве упражнения.

Таким образом, операции MEMBER, INSERT и DELETE на 2-3-дереве с n листьями можно выполнить не более чем за $O(\log n)$ шагов. Следовательно, 2-3-дерево может служить словарем с временной сложностью $O(n \log n)$, поскольку оно может обеспечить выполнение последовательности инструкций, состоящей из n операций MEMBER, INSERT и DELETE, не более чем за $O(n \log n)$ шагов.

Рассмотрим теперь операцию MIN. Наименьший элемент в 2-3-дереве расположен в самом левом листе, который, конечно, можно найти за $O(\log n)$ шагов.

[8] Братьями называются вершины, имеющие одного и того же отца.

Следовательно, с помощью 2-3-дерева любую последовательность инструкций, состоящую из n операций INSERT, DELETE и MIN, можно выполнить за время $O(n \log n)$. Это значит, что временная сложность реализации очереди с приоритетами на основе 2-3-дерева имеет порядок $O(n \log n)$. Для этой же цели подходит и куча, используемая в алгоритме пирамидальной сортировки, и AVL-дерево, обсуждаемое в упражнениях 4.30–4.33.

4.11. Сливаемые кучи

В этом разделе мы рассмотрим структуру данных, с помощью которой можно выполнить последовательность, состоящих из операций INSERT, DELETE, UNION и MIN, за время $O(n \log n)$. В этой структуре, которую можно считать обобщением кучи, рассмотренной в разделе 3.4, множество элементов S представляется 2-3-деревом T. Каждый элемент из S является меткой листа дерева T, но множество листьев не упорядочено, как это было в двух предыдущих разделах. Каждая внутренняя вершина дерева T помечена значением SMALLEST[v], т.е. значением наименьшего элемента, хранящегося в поддереве с корнем v. Для этого приложения 2-3-деревьев функции $L[v]$ и $M[v]$ не нужны.

Наименьший элемент множества S можно найти, двигаясь вниз по дереву T от его корня и переходя от внутренних вершин v к их сыновьям, помеченным наименьшим значением функции SMALLEST. В таком случае, если дерево T содержит n листьев, операция MIN занимает $O(\log n)$ шагов.

Во многих приложениях из множества S удаляется только наименьший элемент. Однако если мы хотим удалить из множества S произвольный элемент, то необходимо найти лист, содержащий его. В приложениях, допускающих нумерацию элементов целыми числами 1, 2, ..., n, можно пронумеровать сами листья. Если же элементы имеют произвольную природу, то можно воспользоваться вспомогательным 2-3-словарем, листья которого содержат указатели на листья дерева T. С помощью этого словаря можно найти любой лист за $O(\log n)$ шагов. Словарь необходимо корректировать при каждом выполнении операции INSERT, но это требует не более $O(\log n)$ шагов.

После удаления листа l из дерева T необходимо для каждого его прямого предка v заново вычислить значения функции SMALLEST. Новым значением SMALLEST[v] будет наименьшее из значений SMALLEST[s] для двух или трех сыновей s вершины v. Если всегда выполнять вычисления снизу вверх, то индукцией по числу повторных вычислений можно показать, что каждое вычисление дает для функции SMALLEST правильный ответ. Поскольку эта функция изменяется только в предках удаленного листа, операцию DELETE можно выполнить за $O(\log n)$ шагов.

Рассмотрим операцию UNION. Каждое множество представлено отдельным 2-3-деревом. Чтобы объединить два множества S_1 и S_2, вызываем процедуру IMPLANT(T_1, T_2), приведенную на рис. 4.32, где T_1 и T_2 — это 2-3-деревья, представляющие S_1 и S_2.[9]

```
procedure IMPLANT(T1, T2):
if HEIGHT(T1) = HEIGHT(T2) then
    begin
        создать новый корень r;
        сделать ROOT[T1] и ROOT[T2] соответственно левым
            и правым сыновьями вершины
    end
else
    wlg HEIGHT(T1) > HEIGHT(T2) otherwise
        поменять местами T1 и T2, а также левых и правых сыновей in
        begin
            пусть  v — такая вершина на самом правом пути в T1,
                что DEPTH(v) = HEIGHT(T1) - HEIGHT(T2);
            пусть f — отец вершины v;
            сделать ROOT[T2] сыном вершины f,
                расположенным непосредственно справа от v;
            if вершина f имеет четырех сыновей then ADDSON(f)[10]
        end
```

Рис. 4.32. Процедура IMPLANT

Пусть высота h_1 дерева T_1 не меньше высоты h_2 дерева T_2. Операция IMPLANT находит на самом правом пути в T_1 вершину v с высотой h_2 и делает корень дерева T_2 его самым правым братом. Если у его отца f окажется четыре сына, операция IMPLANT вызовет процедуру ADDSON(f). Значения функции SMALLEST на вершинах, потомки которых изменяются в процессе выполнения процедуры IMPLANT, можно скорректировать тем же способом, что и в операции DELETE.

В качестве упражнения предлагаем показать, что процедура IMPLANT соединяет деревья T_1 и T_2 в одно 2-3-дерево за время $O(h_1 - h_2)$ при $h_1 > h_2$. Если учитывать время на корректировку значений L и M, то временная сложность процедуры IMPLANT может составить $O\left(\max\left(\log\|S_1\|, \log\|S_2\|\right)\right)$. Рассмотрим теперь приложение, в котором операции UNION, MIN и DELETE возникают естественным образом.

---

[9] Если нам нужны значения L и M для новой вершины, созданной процедурой ADDSON(f), то сначала необходимо найти наибольшего потомка вершины v, следуя по пути, идущему в самый правый лист.

[10] Здесь различие между левым и правым сыновьями неважно; это имеет значение для сцепляемых очередей, которые обсуждаются в следующем разделе.

Пример 4.12. В примере 4.1 мы описали алгоритм для нахождения остовных деревьев наименьшей стоимости. Он создавал из вершин все бо́льшие и бо́льшие множества, такие, что элементы каждого из них соединялись ребрами, выбранными для остовного дерева наименьшей стоимости. Стратегия поиска новых ребер для этого остовного дерева состояла в том, чтобы перебирать ребра (сначала наименьшей стоимости) и проверять, соединяют ли они какие-нибудь еще не соединенные вершины.

В качестве альтернативной стратегии каждому множеству вершин V_i поставим в соответствие множество E_i всех нерассмотренных ребер, инцидентных какой-то вершине из V_i. Если выбрать не рассмотренное ранее ребро e, инцидентное вершине из относительно малого множества V_i, то другой конец ребра e с большой вероятностью не будет лежать в V_i, и можно будет добавить e к остовному дереву. Если это нерассмотренное ребро e обладает наименьшей стоимостью среди всех ребер, инцидентных вершинам из V_i, то можно показать, что включение его в остовное дерево приведет к остовному дереву наименьшей стоимости.

Для реализации этого алгоритма необходимо сначала создать для каждой вершины множество инцидентных ей ребер. Чтобы среди ребер, инцидентных вершинам из V_i, найти нерассмотренное ребро наименьшей стоимости, применим оператор `MIN` к множеству E_i нерассмотренных ребер для V_i. Затем удалим из E_i найденное ребро e. Если окажется, что ребро e имеет в V_i только один конец, а другой лежит в множестве V_i', отличном от V_i, то применим операцию `UNION` к множествами V_i и V_i' (например, с помощью структуры данных из алгоритма 4.3), а также к множествам E_i и E_i'.

В качестве структуры данных для представления каждого множества ребер E_i можно взять 2-3-дерево, каждый лист которого помечен ребром и его стоимостью. В множестве ребер нет никакого специального порядка. Каждой внутренней вершине приписана наименьшая стоимость ее потомков-листьев, обозначаемая `SMALLEST`[v].

Вначале для каждой вершины строим 2-3-дерево, содержащее все инцидентные ему ребра. Чтобы построить такое дерево, начнем с листьев. Затем добавим вершины высоты 1, объединяя листья в группы по два или три так, чтобы групп, состоящих из двух листьев, было не больше двух. Сделав это, вычислим для каждой вершины высоты 1 наименьшую стоимость листа-потомка. Затем соберем вершины высоты 1 в группы по два или три и будем продолжать процесс до тех пор, пока на некотором уровне не возникнет одна вершина — корень. Время, затрачиваемое на такое построение дерева, пропорционально числу листьев. Реализация остальной части алгоритма теперь очевидна. Общее время работы составляет $O(e \log e)$, где e — число всех ребер. □

4.12. Сцепляемые очереди

В разделе 4.10 было показано, как на 2-3-дереве с n листьями выполнить каждую из операций INSERT, DELETE, MIN и MEMBER за время $O(\log n)$, если пользоваться значениями L и M. Теперь покажем, как за время $O(\log n)$ выполнить каждую из операций CONCATENATE и SPLIT. Снова будем предполагать, что элементы расположены на листьях 2-3-дерева в порядке возрастания слева направо и для каждой вершины v вычисляются значения $L[v]$ и $M[v]$.

Операция CONCATENATE(S_1, S_2) получает такие две последовательности S_1 и S_2, что каждый элемент из S_1 меньше каждого элемента из S_2. Результатом этой операции является конкатенация этих последовательностей, т.е. последовательность S_1S_2. Если S_1 и S_2 представлены 2-3-деревьями T_1 и T_2 соответственно, то мы хотим объединить T_1 и T_2 в одно дерево T, листьями которого являются листья дерева T_1 в исходном порядке и следующие за ними листья дерева T_2 в исходном порядке. Это можно сделать, вызвав процедуру IMPLANT(T_1, T_2), приведенную на рис. 4.32.

Наконец, рассмотрим операцию SPLIT. Напомним, что операция SPLIT(a, S) разбивает множество S на два множества: $S_1 = \{b | b \leq a$ и $b \in S\}$ и $S_2 = \{b | b > a$ и $b \in S\}$. Для ее реализации определим процедуру SPLIT(a, T), которая расщепляет 2-3-дерево на два таких 2-3-дерева T_1 и T_2, что метки всех листьев в T_1 не больше a, а метки всех листьев в T_2 больше a.

Этот метод можно описать неформально. Пусть дано 2-3-дерево T, содержащее элемент a. Пройдем по пути из корня в лист с меткой a. Этот путь разбивает наше дерево на поддеревья, корнями которых являются не сами вершины, лежащие на нем, а их сыновья. Это иллюстрирует рис. 4.33, где слева от пути находятся деревья T_1, T_2, T_3 и тривиальное дерево, состоящее из одной вершины v_1, а справа — деревья T_4, T_5 и v_2.

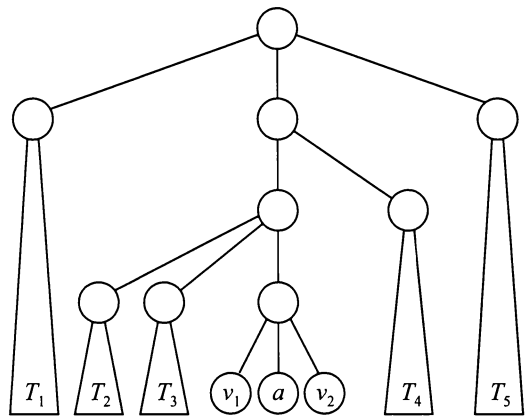


Рис. 4.33. Расщепление 2-3-дерева

Деревья, расположенные слева от рассматриваемого пути, и дерево, состоящее из одной вершины a, соединяются с помощью описанного выше алгоритма конкатенации деревьев. Аналогично соединяются деревья, расположенные справа от пути. Необходимые детали даны в процедуре SPLIT (рис. 4.34).

```
procedure SPLIT(a, T):
begin
    на пути из вершины ROOT[T] к листу с меткой a
        удалить все вершины, кроме этого листа;
    comment В данный момент дерево T оказалось разделенным
        на два леса — левый, состоящий из всех деревьев,
        листья которых лежат слева от a, и вершины с меткой a,
        и правый, состоящий из всех деревьев, листья которых
        лежат справа от a;
    while в левом лесу более одного дерева do
        begin
            пусть T' и T'' — два самых правых дерева в левом лесу;
            IMPLANT(T', T'')[11]
        end;
    while в правом лесу более одного дерева do
        begin
            пусть T' и T''  — два самых левых дерева в правом лесу;
            IMPLANT(T', T'')
        end
end
```

Рис. 4.34. Процедура расщепления 2-3-дерева

Теорема 4.7. Процедура SPLIT разбивает 2-3-дерево T по листу a так, что все листья, расположенные слева от a, и сам лист a оказываются в одном 2-3-дереве, а все листья, расположенные справа от a, — в другом. Эта процедура занимает время $O(\text{HEIGHT}(T))$. Порядок листьев сохраняется.

Доказательство. Свойства процедуры IMPLANT гарантируют, что деревья объединены правильно. Оценка временной сложности получается из следующих соображений. Вначале число деревьев любой данной высоты не превышает два и только деревьев нулевой высоты может быть три. Когда два дерева соединяются, получается дерево, высота которого не более чем на единицу больше наибольшей из высот двух исходных деревьев. Если получается

[11] Результат процедуры IMPLANT(T', T'') относится к левому лесу. Аналогично при применении к деревьям из правого леса результат процедуры IMPLANT относится к правому лесу.

дерево высоты, на единицу большей высоты любого из исходных деревьев, его корень имеет степень двойки. Таким образом, если объединяются три дерева высоты h, то в результате получается дерево высоты не более $h + 1$. Следовательно, на каждой стадии процесса количество деревьев одинаковой высоты не превосходит трех.

Время, необходимое для объединения двух деревьев разной высоты, пропорционально разности их высот, а одинаковой высоты — постоянно. Следовательно, объединение всех деревьев происходит за время, пропорциональное сумме числа деревьев и наибольшей из разностей высот любых двух деревьев. Таким образом, всего общий объем затраченного времени имеет порядок высоты исходного дерева. □

Заметим, что с помощью сцепляемой очереди последовательность S_2 можно вставить между парой элементов последовательности S_1 за время $O\left(\max\left(\log|S_1|, \log|S_2|\right)\right)$. Если $S_2 = b_1, b_2, ..., b_n$; $S_1 = a_1, a_2, ..., a_m$ и последовательность S_2 необходимо вставить между элементами a_i и a_{i+1}, то можно применить операцию SPLIT(a_1, S_1) и расщепить последовательность S_1 по элементу a_i на последовательности $S_1' = a_1, ..., a_i$ и $S_1'' = a_{i+1}, ..., a_m$, применить операцию CONCATENATE(S_1', S_2), получив последовательность $S_3 = a_1, ..., a_i, b_1, ..., b_n$, и, наконец, операцию CONCATENATE(S_3, S_1''), дающую искомую последовательность.

4.13. Разбиение

Рассмотрим специальный тип расщепления, называемый *разбиением*. Задача разбиения множества возникает довольно часто, и решение, которое мы продемонстрируем здесь, поучительно само по себе. Пусть даны множество S и разбиение π множества S на непересекающиеся блоки $\{B_1, B_2, ..., B_p\}$. Кроме того, дана функция f, отображающая S на S.

Задача состоит в том, чтобы найти такое *грубейшее* (с наименьшим числом блоков) разбиение $\pi' = \{E_1, E_2, ..., E_q\}$ множества S, что

а) π' согласовано с π (т.е. каждое множество E_i является подмножеством некоторого блока B_j);

б) если a и b принадлежат E_i, то $f(a)$ и $f(b)$ принадлежат некоторому множеству E_j.

Будем называть π' *грубейшим разбиением множества S, совместимым с разбиением* π *и функцией f*.

Очевидное решение состоит в повторном дроблении блоков исходного разбиения следующим способом. Пусть B_i — некоторый блок. Рассмотрим $f(a)$ для каждого a из B_i. Разобьем B_i так, чтобы два элемента a и b попадали в один блок тогда

и только тогда, когда $f(a)$ и $f(b)$ оба принадлежат некоторому блоку B_j. Будем повторять этот процесс до тех пор, пока дальнейшее дробление станут невозможными. Этот алгоритм имеет сложность $O(n^2)$, поскольку каждое дробление занимает время $O(n)$, а всего может быть $O(n)$ уточнений. Пример 4.13. демонстрирует, что для разбиения действительно может потребоваться квадратичное число шагов.

Пример 4.13. Пусть $S = \{1, 2, ..., n\}$ и $B_1 = \{1, 2, ..., n-1\}$, $B_2 = \{n\}$ — исходное разбиение. Пусть f — такая функция на S, что $f(i) = i + 1$ для $1 \le i < n$ и $f(n) = n$. На первой итерации разбиваем B_1 на $\{1, 2, \ldots, n-2\}$ и $\{n-1\}$. Эта итерация занимает $n-1$ шагов, поскольку необходимо просмотреть каждый элемент в B_1. На следующей итерации разбиваем последовательность $\{1, 2, ..., n-2\}$ на $\{1, 2, ..., n-3\}$ и $\{n-2\}$. Продолжая аналогично, выполняем всего n–2 итерации, причем i-я итерация состоит из $n-i$ шагов. Следовательно, общее количество шагов равно

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

Окончательным разбиением будет $E_i = \{i\}$ для $1 \le i \le n$. □

Недостаток этого метода состоит в том, что утончение блока может потребовать $O(n)$ шагов, даже если из него удаляется только один элемент. Опишем алгоритм разбиения, который для разделения блока на два подблока требует время, пропорциональное размеру меньшего подблока. Этот подход приводит к алгоритму, имеющему временную сложность $O(n \log n)$.

Для каждого блока $B \subseteq S$ положим $f^{-1}(B) = \{b \mid f(b) \in B\}$. Вместо того чтобы разбивать B_i по значениям $f(a)$ для $a \in B_i$, разобьем относительно B_i те блоки B_j, которые содержат хотя бы один элемент из $f^{-1}(B_i)$ и один элемент не из $f^{-1}(B_i)$. Иначе говоря, каждый такой блок B_j разбивается на множества $\{b | b \in B_j$ и $f(b) \in B_i\}$ и $\{b | b \in B_j$ и $f(b) \notin B_i\}$.

Как только проведено разбиение относительно блока B_i, больше не требуется проводить разбиения относительно него, пока он сам не будет расщеплен. Если вначале $f(b) \in B_i$ для каждого элемента $b \in B_j$ и B_j расщеплен на B_i' и B_i'', то можно разбить B_j относительно B_i' или B_i'' и получить тот же результат, поскольку множество $\{b | b \in B_j$ и $f(b) \in B_i'\}$ совпадает с множеством $B_j - \{b | b \in B_j$ и $f(b) \in B_i''\}$.

Поскольку существует выбор, по отношению к какому из блоков B_i' или B_i'' проводить разбиение, будем разбивать последовательность относительно того блока, для которого это сделать проще. Точнее говоря, будем выбирать меньшее из множеств $f^{-1}(B_i')$ и $f^{-1}(B_i'')$. Алгоритм приведен на рис. 4.35.

Алгоритм 4.5. Разбиение

Вход. Множество S из n элементов, начальное разбиение $\pi = \{B[1], ..., B[p]\}$ и функция f: $S \to S$.

Выход. Грубейшее разбиение $\pi' = \{B[1], B[2], ..., B[q]\}$ множества S, совместимое с разбиением π и функцией f.

Метод. К разбиению π применяется программа, приведенная на рис. 4.35. В ней опущены некоторые детали, важные для ее реализации. Мы обсудим эти детали при анализе времени работы алгоритма. □

```
      begin
          WAITING ← {1, 2, ... , p};
2.        q ← p;
3.        while WAITING не пусто do
              begin
4.                выбрать и удалить любое целое число i из WAITING
5.                INVERSE ← f⁻¹(B[i]);
6.                for j, для которых выполняются условия
                      B[j] ⋂ INVERSE ≠ ∅ и B[j] ⊄ INVERSE do
                          begin
7.                            q ← q + 1;
8.                            создать новый блок B[q];
9.                            B[q] ← B[j] ⋂ INVERSE;
10.                           B[j] ← B[j] - B[q];
11.                           if  j ∈ WAITING then
                                  добавить q в WAITING
                              else
12                                if ||B[j]|| ≤ ||B[q]|| then
13.                               добавить j в WAITING
14.                           else добавить q в WAITING
                          end
              end
      end
```

Рис. 4.35. Алгоритм разбиения

Анализ алгоритма 4.5 начнем с доказательства того, что он правильно разбивает множество S. Пусть π — разбиение множества S и f — отображение множества S на себя. Множество $T \subseteq S$ будем называть *допустимым* для разбиения π, если для любого блока $B \in \pi$ либо $B \subseteq f^{-1}(T)$, либо $B \bigcap f^{-1}(T) = 0$. Например, на рис. 4.35 строки 9 и 10 гарантируют, что множество $B[i]$ является допустимым для получающегося разбиения, поскольку если $B \bigcap f^{-1}(T) \neq \varnothing$ для некоторого блока B, то либо $B \subseteq$ INVERSE и тогда включение $B \subseteq f^{-1}(B[i])$ очевидно, либо в строках 9 и 10 множество B разбивается на два блока, один из которых является подмножеством множества $f^{-1}(B[i])$, а другой с ним не пересекается.

Частью доказательства корректности алгоритма 4.5 является доказательство того, что окончательное разбиение не оказывается слишком грубым. Иначе говоря, необходимо доказать следующее утверждение.

Лемма 4.7. После окончания работы алгоритма 4.5 каждый блок B в получающемся разбиении π' является допустимым для π'.

Доказательство. Для доказательства леммы покажем, что для каждого блока $B[l]$ справедливо следующее утверждение:

после каждого выполнения цикла в строках 4–14 на рис. 4.35, если $l \notin$ WAITING и $l \le q$, то существует список $q_1, q_2, ..., q_k$ (возможно, пустой), что $q_i \in$ WAITING, $1 \le i \le k$, и множество в $B[l] \cup B[q_1] \cup ... \cup B[q_k]$ безопасно для текущего разбиения. (4.6)

Неформально это можно сформулировать так: когда число l удаляется из множества WAITING, строки 6–14 делают блок $B[l]$ допустимым для разбиения, которое получается после строки 14. Блок $B[l]$ остается допустимым до тех пор, пока он не будет разбит. Когда $B[l]$ разбивается, индекс одного подблока, назовем его $B[q]$, помещается в множество WAITING. Другой подблок по-прежнему называется $B[l]$. Очевидно, что объединение этих двух подблоков, т.е. $B[l] \cup B[q]$, является допустимым для рассматриваемого разбиения, поскольку оно совпадает со старым блоком $B[l]$. Дальнейшее разбиение образует такие блоки $B[q_1], ..., B[q_k]$, что числа $q_1, ..., q_k$ входят в множество WAITING, а объединение $R = B[l] \cup B[q_1] \cup ... \cup B[q_k]$ является допустимым. Когда некоторое множество q_i, $(1 \le i \le k)$ удаляется из множества WAITING, строки 6–14 снова делают множества $B[q_i]$ и $R - B[q_i]$ допустимыми. Теперь перейдем к формальному доказательству.

Докажем справедливость утверждения (4.6) для всех l индукцией по числу выполнений строк 4–14. Когда алгоритм заканчивает работу, множество WAITING пусто, и, значит, из (4.6) будет следовать, что каждый блок окончательного разбиения π' является допустимым для π'.

Для доказательства базы индукции допустим, что строки 4–14 вообще не выполняются. В таком случае утверждение (4.6) тривиально, поскольку $l \in$ WAITING для всех $1 \le l \le q = p$.

Для проведения шага индукции предположим, что l не входит в множество WAITING после завершения строки 14. Если число l ранее всегда входило в множество WAITING, то оно имеет значение i, определенное в строке 4. Легко показать, что цикл в строках 6–14 делает блок $B[i]$ допустимым для разбиения, получающегося после выполнения строки 14. Это утверждение мы уже обосновали после определения понятия “допустимый”.

Если число l не входило в множество WAITING во время предыдущего выполнения цикла до строки 14, то по предположению индукции найдется такой список q_1, ..., q_k, что утверждение (4.6) было справедливо для i на предыдущем этапе. Кроме того, в строке 4 обязательно выполняется неравенство $i \neq l$.

Вариант 1. Число i не входит в множество $L = \{q_1, q_2, ..., q_k\}$. В строках 9, 10 могло произойти разбиение нескольких блоков. Для каждого такого блока $B[q_r]$, $1 \leq r \leq k$, (т.е. $j = q_r$) добавим в L индекс блока, созданного в строке 8. В результате выполнения строки 11 список L по-прежнему будет состоять только из индексов, входящих в множество WAITING. Если сам блок $B[l]$ не разбит, то $B[l]$ и множество блоков с индексами из L все еще образуют множество, допустимое для текущего разбиения, так что утверждение (4.6) выполняется. Если же блок $B[l]$ разбит, то необходимо также добавить в множество L индекс q, выбранный в строке 8, когда $j = l$. Следовательно, множество $B[l] \cup \bigcup_{r \in L} B[r]$ будет допустимым для текущего разбиения.

Вариант 2. Число i принадлежит множеству $L = \{q_1, q_2, ..., q_k\}$. Не ограничивая общность, будем считать, что $i = q_1$. Рассуждение проводится почти так же, как в случае 1, но q_1 в конце рассматриваемой итерации строк 4–14 может не входить в множество WAITING. Однако мы знаем, что каждый блок B текущего разбиения будет либо подмножеством множества $f^{-1}(B[q_1])$, либо не будет с ним пересекаться. Пусть $T = B[l] \cup \bigcup_{r \in L} B[r]$, где список L модифицирован, как в варианте 1. Если $B \subseteq f^{-1}(B[q_1])$, то, разумеется, $B \cap f^{-1}(T) = \varnothing$. Если $B \cap f^{-1}(B[q_1]) = \varnothing$, то аналогично варианту 1 можно доказать, что $B \cap f^{-1}(T) = \varnothing$ или $B \subseteq f^{-1}(T)$.

Наконец, когда алгоритм 4.5 заканчивает работу, множество WAITING должно быть пустым. Следовательно, из утверждения (4.6) вытекает, что для каждого l блок $B[l]$ является допустимым для окончательного разбиения. □

Теорема 4.8. Алгоритм 4.5 правильно вычисляет грубейшее разбиение множества S, совместимое с разбиением π и функцией f.

Доказательство. Лемма 4.7 показывает, что результат π' алгоритма 4.5 совместим с разбиением π и функцией f. Необходимо доказать, что разбиение π' является максимально грубым. Простая индукция по числу блоков, разбиваемых в строках 9, 10, показывает, что каждое такое разбиение, сделанное алгоритмом, является необходимым для совместимости. Оставляем эту индукцию в качестве упражнения. □

Теперь мы должны детально проанализировать реализацию алгоритма 4.5, чтобы показать, что время его работы составляет $O(n \log n)$, где $n = \|S\|$. При оценке времени работы главное — показать, что цикл в строках 6–14 можно выпол-

нить за время, пропорциональное ||INVERSE||. Наша первая задача —эффективно найти подходящее множество чисел j в строке 6. Нам понадобится такой массив INBLOCK, что INBLOCK[l] — индекс блока, содержащего l. Массив INBLOCK можно инициализировать за $O(n)$ шагов и после строки 9 скорректировать его за время, не большее того, что тратится на формирование списка $B[q]$ в этой строке. Следовательно, поскольку нас интересует только порядок временной сложности, можно не учитывать время, затрачиваемое на работу с массивом INBLOCK.

С помощью массива INBLOCK легко построить список JLIST чисел j, необходимых в строке 6, за $O(\|\text{INVERSE}\|)$ шагов. Для каждого элемента a, входящего в множество INVERSE, индекс блока, содержащего a, добавляется в список JLIST, если его там еще нет. Для каждого j хранится счетчик числа элементов, входящих в множество INVERSE, которые входят также и в блок $B[j]$. Если этот счетчик достигает величины $\|B[j]\|$, то $B[j] \subseteq$ INVERSE и j удаляется из списка JLIST.

Используя массив INBLOCK, можно для каждого j, входящего в список JLIST, построить также список INTERSECTION[j], в который войдут все целые числа, принадлежащие обоим множествам $B[j]$ и INVERSE. Чтобы быстро удалить элементы списка INTERSECTION[j] из $B[j]$ и добавить их в $B[q]$, необходимо поддерживать двусвязные списки $B[l]$, $1 \le l \le q$, т.е. списки, содержащие указатели как на следующий, так и к предыдущий элементы.

Выполнение строк 9 и 10 требует выполнения $O(\|B[q]\|)$ шагов. Для данного выполнения цикла **for** суммарное время, затрачиваемое на нахождение подходящих чисел j и выполнение строк 7–10, составляет $O(\|\text{INVERSE}\|)$. Кроме того, легко видеть, что проверка в строках 12–14, если ее выполнять должным образом, занимает $O(\|B[q]\|)$ времени, так что общая временная сложность составляет $O(\|\text{INVERSE}\|)$.

Осталось рассмотреть строку 11. Чтобы быстро проверить, входит ли j в множество WAITING, образуем вспомогательный массив INWAITING[j]. Массив INWAITING можно инициализировать за $O(n)$ шагов и без труда корректировать в строках 11–14. Таким образом, справедлива следующая лемма.

Лемма 4.8. Цикл **for** в строках 6–14 на рис. 4.35 можно реализовать за время $O(\|\text{INVERSE}\|)$.

Доказательство. См выше. □

Теорема 4.9. Алгоритм 4.5 можно реализовать за время $O(n \log n)$.

Доказательство. Рассмотрим условия, при которых блок, содержащий целое число S и не представленный в множестве WAITING, может туда попасть.

В строке 1 это может случиться лишь один раз. В строке 11 это произойти не может, даже если $s \in B[q]$, поскольку тогда число s было бы в $B[j]$ еще раньше, а индекс j уже содержался бы в множестве WAITING. Если это случилось в строках 13 и 14, то число s находится в блоке, размер которого не больше половины размера того блока, куда оно входило в тот предыдущий момент, когда индекс множества, содержащего s, попал в множество WAITING. Отсюда можно заключить, что индекс множества, содержащего число s, попадает в множество WAITING не больше $1+\log n$ раз. Следовательно, число s не может быть в блоке i, который выбирался в строке 4 более чем $1+\log n$ раз.

Допустим, что на каждый элемент s из $B[i]$ каждый раз налагается штраф, пропорциональный $\|f^{-1}(s)\|$, так что сумма всех штрафов равна сложности выполнения цикла в строках 6–14. Тогда существует такая константа c, что за одно выполнение цикла элемент s штрафуется не более чем на $c\|f^{-1}(s)\|$. Как мы уже показали, элемент s не может попасть в выбираемый список $B[i]$ более чем $O(\log n)$ раз, так что суммарный штраф, налагаемый на него, составляет $O\left(\left\|f^{-1}(s)\right\|\log n\right)$. Поскольку сумма $\sum_{s\in S}\left\|f^{-1}(s)\right\|$ должна равняться n, общая сложность всех выполнений цикла **for** составляет $O(n\log n)$. Легко видеть, что сложность остальной части алгоритма 4.5 имеет порядок $O(n)$. Теорема доказана. □

Алгоритм 4.5 имеет несколько приложений. Одно из важных приложений — минимизация числа состояний конечного автомата. Пусть задан конечный автомат $M = (S, I, \delta, s_0, F)$ и требуется найти эквивалентный ему автомат M' с наименьшим числом состояний. Для каждого состояния S и входного символа a обозначим через $\delta(S, a)$ очередное состояние автомата. Вначале все состояния автомата M можно разбить на множество F допускающих состояний и множество $S - F$ недопускающих состояний. Задача минимизации числа состояний автомата M эквивалентна нахождению грубейшего разбиения π' множества S, совместимого с начальным разбиением $\{F, S - F\}$ и такого, что если состояния s и t находятся в одном блоке разбиения π', то состояния $\delta(s, a)$ и $\delta(t, a)$ также находятся в одном блоке автомата M' для каждого входного символа a.

Единственное отличие этой задачи от задачи, решаемой алгоритмом 4.5, состоит в том, что δ — отображение из $S \times I$ в S, а не из S в S. Однако δ можно интерпретировать как множество $\{\delta_{a_1}, \delta_{a_2}, ..., \delta_{a_m}\}$ функций, заданных на S, где δ_a — сужение функции δ на входной символ a.

Алгоритм 4.5 легко модифицировать так, что он решит эту более общую задачу, помещая пары (i, δ_a) в множество WAITING. Каждая пара (i, δ_a) состоит из индекса i некоторого блока разбиения и функции δ_a, по которой проводится разбиение. Вначале WAITING = $\{(i, \delta_a) | i = 1$ или 2 и $a \in I\}$, поскольку начальное разбиение

$\{F, S - F\}$ состоит из двух блоков. Всякий раз, когда блок $B[j]$ разбивается на $B[j]$ и $B[q]$, каждая допустимая функция δ_a образует пару с числами j и q. Остальные детали оставляем читателям в качестве упражнения.

4.14. Резюме

На рис. 4.36 приведены различные структуры данных, рассмотренные в этой главе, типы операций, которые можно выполнять на их основе, и предположения об их размерах и природе базового множества, откуда извлекаются элементы.

			Время выполнения n операций над множеством размера n	
	Структура данных	**Тип базового множества**	**Допустимые операции**	**Ожидаемое время**	**Худшее время**
1.	Хеш-таблица	Произвольное множество, на котором можно вычислить хеш-функцию	MEMBER, INSERT, DELETE	$O(n)$	$O(n^2)$
2.	Дерево бинарного поиска	Произвольное упорядоченное множество	MEMBER, INSERT, DELETE, MIN	$O(n \log n)$	$O(n^2)$
3.	Дерево из алгоритма 4.3	Целые числа от 1 до n	MEMBER, INSERT, DELETE, UNION, FIND	не более $O(nG(n))$	не более $O(nG(n))$
4.	2-3-деревья с неупорядоченными листьями	Произвольное упорядоченное множество	MEMBER, INSERT, DELETE, UNION, FIND, MIN	$O(n \log n)$	$O(n \log n)$
5.	2-3-деревья с упорядоченными листьями	Произвольное упорядоченное множество	MEMBER, INSERT, DELETE, FIND, SPLIT, MIN, CONCATENATE	$O(n \log n)$	$O(n \log n)$

Рис. 4.36. Свойства структур данных

Упражнения

4.1. Укажите подмножество семи основных операций из раздела 4.1, достаточное для упорядочения любой последовательности, состоящей из n элементов. Что можно сказать о сложности выполнения потока из n операций, выбранных из вашего подмножества?

4.2. Пусть элементами являются строки из букв и используется следующая функция хеширования для таблицы размера $m = 5$: сложить "значения" букв, где A имеет значение 1, B — значение 2 и т.д., а затем разделить результат на 5 и взять остаток. Выпишите содержимое таблицы хеширования и списков, при условии, что вставляются строки `DASHER`, `DANCER`, `PRANCER`, `VIXEN`, `COMET`, `CUPID`, `DONNER`, `BLITZEN`.

4.3. Вставьте восемь строк из упражнения 4.2 в дерево бинарного поиска. Какую последовательность вершин необходимо посетить, чтобы проверить, принадлежит ли `RUDOLPH` этому множеству?

4.4. Покажите, что процедура `SEARCH` (см. рис. 4.3) имеет наименьшее возможное ожидаемое время поиска, если все элементы универсального множества разыскиваются с одинаковой вероятностью.

4.5. Найдите оптимальное дерево бинарного поиска для a, b, ..., h, если эти элементы имеют вероятности соответственно 0,1; 0,2; 0,05; 0,1; 0,3; 0,05; 0,15; 0,05, а вероятность всех остальных равна нулю.

****4.6.** Покажите, что в алгоритме 4.2 можно ограничить поиск числа m в строке 8 на рис. 4.9 диапазоном позиций от $r_{i,j-1}$ до $r_{i+1,j}$ и все равно гарантированно находить максимум.

***4.7.** Используйте упражнение 4.6 для такой модификации алгоритма 4.2, чтобы его временная сложность составляла $O(n^2)$.

4.8. Закончите доказательство теоремы 4.2, показав, что алгоритм построения дерева на рис. 4.10 работает корректно.

4.9. Закончите доказательство теоремы 4.3.

***4.10.** Постройте интерфейс для перевода множества n внешних имен, лежащих между 1 и r, в множество внутренних имен, состоящее из целых чисел от 1 до n. Этот интерфейс должен переводить в обе стороны в предположении, что $r \gg n$.

а) Разработайте интерфейс с хорошей ожидаемой временной сложностью.

б) Разработайте интерфейс с хорошей временной сложностью в худшем случае.

***4.11.** Найдите эффективную структуру данных для представления подмножества S целых чисел, лежащих между 1 и n. Мы хотим выполнять на множестве следующие операции.

а) Выбирать и удалять произвольное целое число.

б) Добавлять целое число i.

Предполагается, что механизм не должен добавлять в S целые числа, которые там уже есть. Структура данных должна быть такой, чтобы время на выбор и удаление элемента и время на добавление элемента не зависели от $||S||$.

4.12. Найдите дерево, которое получается, когда алгоритм 4.3 выполняет последовательность операций UNION и FIND, порождаемую следующей программой. Допустим, что множество i вначале имеет вид $\{i\}$ для $1 \le i \le 16$.

```
begin
     for i ← 1 step 2 until 15 do UNION(i, i + 1, i);
     for i ← 1 step 4 until 13 do UNION(i, i + 2, i);
     UNION (1, 5, 1);
     UNION (9, 13, 9);
     UNION (1 , 9, 1);
     for i ← 1 step 4 until 13 do FIND(i)
end
```

4.13. Пусть σ — последовательность операций UNION и FIND, причем все операции UNION предшествуют операциям FIND. Докажите, что алгоритм 4.3 выполняет последовательность σ за время, пропорциональное длине последовательности σ.

4.14. Назовем S_0-деревом такое дерево, которое состоит из единственной вершины. S_i-дерево для $i > 0$ получается операцией, которая делает корень одного S_{i-1}-дерева сыном корня другого S_{i-1}-дерева. Докажите, что

а) S_n-дерево имеет $\binom{n}{h}$ вершин высоты h.

б) S_n-дерево можно получить из S_m-дерева ($m \le n$), заменив каждую вершину S_m-дерева на S_{n-m}-дерево так, что сыновья этой вершины становятся сыновьями корня подставляемого S_{n-m}-дерева.

в) S_n-дерево содержит вершину с n сыновьями, которые служат корнями S_0- S_1-, ..., S_{n-1}-деревьев.

4.15. Рассмотрим алгоритм объединения непересекающихся множеств, который делает корень дерева с меньшим числом вершин (в случае одинако-

вого количества вершин выбор дерева произволен) сыном корня большего дерева, но не использует сжатие путей. Докажите, что верхнюю границу $O(n \log n)$ нельзя улучшить. Иначе говоря, покажите, что для некоторой константы c и произвольно больших значений n существуют последовательности операций UNION и FIND, требующие $cn \log n$ шагов.

**4.16. Пусть $T(n)$ — временная сложность выполнения n операций UNION и FIND в худшем случае при условии, что применяется древовидная структура из раздела 4.7 и сжатие путей для операций FIND, но операция UNION(A, B, C) выполняется так, что корень дерева A становится сыном корня дерева B независимо от того, какое из множеств больше. Покажите, что $T(n) \geq k_1 n \log n$ для некоторой константы $k_1 > 0$.

**4.17. Покажите, что $T(n) \leq k_2 n \log n$ для некоторой константы k_2, где $T(n)$ обозначает то же, что и в упражнении 4.16.

**4.18. Покажите, как можно выполнить последовательность из n операций UNION, FIND, MEMBER, INSERT, DELETE на множестве целых чисел 1, 2, ..., n за время $O(nG(n))$. Будем предполагать, что операция DELETE(i, S) делает i членом нового множества $\{i\}$, которому необходимо присвоить (произвольное) имя. Это множество можно впоследствии объединить с другим. Кроме того, предположим, что никакой элемент не принадлежит более чем одному множеству.

4.19. Обобщите офлайн-задачу MIN так, чтобы можно было выполнять операцию MIN вида MIN(i), которая находит все целые числа, меньшие i, вставленные до нее и еще не найденные предыдущей операцией MIN.

4.20. Разработайте полную структуру данных для офлайн-задачи MIN, включающую представление деревьев массивами, и напишите программу, в которой употребляются массивы, а не команды высокого уровня, используемые алгоритмом объединения непересекающихся множеств.

4.21. Обобщите офлайн-задачу MIN следующим образом. Пусть T — дерево с n вершинами. Каждой вершине ставится в соответствие целое число от 1 до n. Некоторым вершинам приписаны операции EXTRACT_MIN. Обойдите дерево в обратном порядке. Встретив операцию EXTRACT_MIN в вершине v, найдите в дереве с корнем v наименьшее целое число (исключая число в самой вершине v) среди тех чисел, которые не были удалены ранее, и удалите его. Укажите алгоритм для этого процесса, работающий в режиме офлайн и имеющий сложность $O(nG(n))$.

**4.22. Разработайте алгоритм сложности $O(n \log \log n)$ для задачи UNION-FIND, пользуясь структурой данных, состоящей из дерева, каждый лист которо-

го отстоит от корня на расстояние 2. Ограничьте степень корня числом между 1 и $n/\log n$, а степень каждого его сына — числом между 1 и $\log n$. Как модифицировать этот алгоритм, чтобы его временная сложность не превышала $O(n \log\log\log n)$? Какую наилучшую оценку временной сложности можно получить, обобщая этот метод?

4.23. Покажите, как в таблице символов можно хранить смещения от начала отсчета, упомянутые в приложении 2 раздела 4.8. (*Подсказка*: примените прием, использованный для определения глубины.)

4.24. Напишите программу для операций `UNION` и `FIND`, осуществляющую вычисления с весом, как в приложении 2 раздела 4.8.

4.25. С помощью алгоритма, приведенного на рис. 4.25, проверьте эквивалентность конечных автоматов. Начальными состояниями являются 1 и A соответственно, а множествами заключительных состояний —множества $\{5\}$ и $\{C, E\}$ соответственно.

Текущее состояние	Вход 0	Вход 1
1	2	3
2	3	5
3	4	5
4	2	5
5	1	2

Следующее состояние

Текущее состояние	Вход 0	Вход 1
A	D	B
B	D	C
C	A	B
D	B	E
E	A	D

Следующее состояние

4.26. Напишите полные программы для выполнения операций на 2-3-деревьях:

а) `DELETE`

б) `UNION`

в) `MEMBER` (в предположении, что множество листьев упорядочено и каждая вершина помечена номером самого высокого листа среди всех листьев, с которыми она соединена);

г) `SPLIT` (в предположении, что задан лист, по которому производится расщепление, и множество листьев упорядочено).

4.27. Напишите полную программу для вставки новой вершины в 2-3-дерево, предполагая, что множество листьев упорядочено.

4.28. Напишите программы для основных операций MEMBER, INSERT, DELETE, MIN, UNION и FIND, используя 2-3-деревья с метками SMALLEST из раздела 4.11. Базовым множеством будем считать $\{1, 2, ..., n\}$.

4.29. Рассмотрим реализацию сливаемого дерева в виде 2-3-дерева (INSERT, DELETE, UNION, MIN). Будем считать, что базовое множество, из которого берутся элементы, велико. Опишите, как реализовать операцию FIND за $O(\log n)$ шагов, где n — общее число элементов во всех соединяемых деревьях.

Определение. *AVL-деревом*[12] называется такое дерево бинарного поиска, в котором для каждой вершины v высоты его левого и правого поддеревьев отличаются не более чем на единицу. Если какого-то поддерева нет, его высота считается равной -1.

Пример 4.14. Дерево на рис. 4.37 не является AVL-деревом, потому что высота левого поддерева вершины * равна 2, а правого — 0. Во всех остальных вершинах AVL-условие выполняется. □

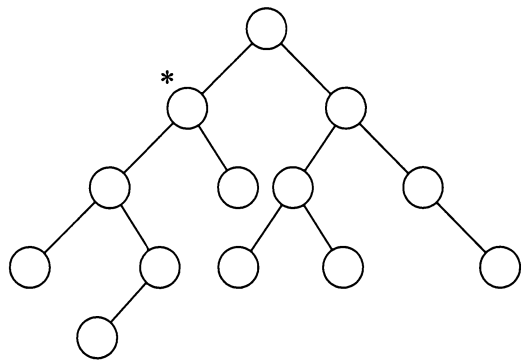


Рис. 4.37. Не AVL-дерево

4.30. Покажите, что AVL-дерево высоты h содержит не более $2^{h+1} - 1$ вершину и не менее $\frac{5+2\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^h + \frac{5-2\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^h - 1$ вершин.

[12]Аббревиатура *AVL* образована из первых букв фамилий авторов метода — советских математиков Г. М. Адельсона-Вельского и Е. М. Ландиса.

*4.31. Пусть T — дерево бинарного поиска с n вершинами, обладающее AVL-свойством. Напишите алгоритм сложности $O(\log n)$ для операций INSERT и DELETE, которые сохраняют AVLсвойство. Можно считать, что высоту каждой вершины можно найти в ней самой и что информация о высотах корректируется автоматически.

*4.32. Напишите алгоритмы для расщепления AVL-дерева и конкатенации двух AVL-деревьев. Эти алгоритмы должны иметь временную сложность, пропорциональную высотам деревьев.

*4.33. Используйте AVL-дерево в качестве базиса алгоритма, выполняющего операции MIN, UNION и DELETE над множествами целых чисел от 1 до n, затрачивая $O(\log n)$ шагов на операцию.

Определение. *Балансом* вершины v бинарного дерева называется число $(1 + L) / (2 + L + R)$, где L и R — количество вершин в левом и правом поддеревьях вершины v. Дерево бинарного поиска называется *αсбалансированным*, если баланс каждой вершины заключен между α и $1\ \alpha$.

Пример 4.15. Баланс вершины, отмеченной знаком * на рис. 4.37, равен 5/7. Ни одна другая вершина не имеет баланса, столь сильно уклоняющегося от 1/2, так что дерево на рис. 4.37 2/7-сбалансировано.

*4.34. Укажите верхнюю и нижнюю границы числа вершин в α-сбалансированном дереве высоты h.

**4.35. Повторите упражнения 4.31–4.33 для деревьев с балансом α, где $\alpha \leq 1/3$.

*4.36. Можно ли выполнить упражнения 4.31–4.33, если $\alpha > 1/3$?

*4.37. Разработайте концепцию сбалансированного дерева с данными на листьях, в котором балансировка достигается за счет того, что разность высот поддеревьев поддерживается в пределах фиксированной константы.

*4.38. Напишите алгоритм сложности $O(nG(n))$, который по заданному дереву с n вершинами и списку из n пар вершин находил бы для каждой пары (v, w) ближайшего общего предка вершин v и w.

*4.39. Длина *внешних путей* бинарного дерева определяется как сумма глубин всех его листьев, длина *внутренних путей* — как сумма глубин всех его вершин. Каково соотношение между длинами внешних и внутренних путей, если у каждой вершины либо два сына, либо ни одного?

*4.40. Разработайте структуру данных для реализации очередей, которая позволяла бы выполнять в онлайн-режиме следующие операции.

а) ENQUEUE(i, A) — добавить целое число i в очередь A. Допускаются различные вхождения одного и того же числа.

б) DEQUEUE(A) — извлечь из A тот элемент, который находится там дольше всех.

в) MERGE(A, B, C), т.е. объединить очереди A и B в новую очередь C. Предполагается, что элементы находятся в объединенной очереди так же долго, как они находились в соответствующей очереди A или B. Заметьте, что одно и то же число может несколько раз входить в A и B и каждое вхождение следует рассматривать как отдельный элемент.

Сколько времени нужно для выполнения последовательности, состоящей из n операций?

***4.41.** Разработайте структуру данных для офлайн-реализации операций из упражнения 4.40. Какое время потребуется для выполнения последовательности, состоящей из n таких операций?

4.42. В алгоритме 4.5 предполагалось, что начальное разбиение $\pi = \{B_1, ..., B_q\}$ было таким, что для каждого $1 \le i \le q$, выполнялось условие $f^{-1}(B_i) \subseteq B_j$. Уменьшает ли временную сложность алгоритма 4.5 это допущение?

Проблема для исследования

4.43. Существует много нерешенных вопросов, связанных со скоростью выполнения последовательностей, состоящих из n операций, выбранных из семи операций, приведенных в разделе 4.1 (или других основных операций того же типа). Если элементы извлекаются из произвольного множества и единственный способ получения информации о них — это попарное сравнение, то любое множество базовых операций, с помощью которых их можно упорядочивать, потребует время $O_C(n \log n)$. Однако, если базовое множество представляет собой множество целых чисел $\{1, 2, ..., n\}$ (и даже $\{1, 2, ..., n^k\}$ для фиксированного k), то сложно доказать, что для сортировки потребуется более $O(n)$ времени, если только базовых операций для нее достаточно. Таким образом, существует возможность улучшить ожидаемое время или время в худшем случае, приведенное на рис. 4.36. И наоборот, нельзя ли для некоторого подмножества (или даже всего множества) базовых операций на целых числах от 1 до n получить нижние оценки, лучшие, чем $O(n)$?

Библиографические замечания

Информация о разнообразии методов хеширования приведена в книгах Morris (1968), Aho and Ulman (1973) и Knuth [1973a]. В последней книге содержится также информация о деревьях бинарного поиска. Алгоритм 4.2 для построения статического дерева бинарного поиска заимствован у Gilbert and Moore [1959]. Алгоритм сложности $O(n^2)$ для той же задачи можно найти в Knuth [1971], где приведено также и решение упражнения 4.6. Hu and Tucker [1971] показали, что временной сложности $O(n^2)$ и пространственной сложности $O(n)$ достаточно, чтобы построить оптимальное дерево бинарного поиска, в котором данные появляются только на листьях. Knuth [1973a] привел реализацию этого алгоритма за время $O(n \log n)$.

Reingold [1972] приводит оптимальные алгоритмы для многих основных преобразований множеств, таких, как объединение и пересечение. Алгоритм 4.3 для задачи `UNION -FIND` был, по-видимому, впервые применен M. D. McIlroy and R. Morris. Knuth [1969] считает автором алгоритма сжатия путей A. Tritter. Теорема 4.4 взята у Hopcroft and Ullman [1974], а теорема 4.5 — у Tarjan [1974]. Отождествление идентификаторов и вычисление смещения, обсуждавшиеся в разделе 4.8, см. в Galler and Fisher [1964]. Эквивалентность конечных автоматов описана в статьи Hopcroft and Karp [1971]. Упражнения 4.16 и 4.17, касающиеся сложности алгоритма 4.3 без сжатия путей, заимствованы из Fisher [1972] и Paterson [1973] соответственно.

Описание AVL-деревьев приведено по работе Adel'son-Vel'skii and Landis [1962], а ответы на упражнение 4.31 можно найти в Crane [1972]. Понятие α сбалансированного дерева изложено в Nievergelt and Reingold [1973]. Читатели могут обратиться к этой работе при выполнении упражнений 4.34–4.37. Дальнейшие применения 2-3-деревьев можно найти в работе Ullman [1974].

Упражнение 4.22 представляет собой раннее решение задачи `UNION –FIND`, приведенное в Stearns and Rosenkrantz [1969]. Упражнение 4.38 см. в Aho, Hopcroft and Ullman [1974].

Различие между онлайн- и офлайн-режимами работы алгоритмов ввели Hartmania, Lewis and Stearns [1965]. Rabin [1963] изучил важный частный случай онлайн-вычислений, называемый вычислениями в реальном времени.

Алгоритм разбиения и задача о минимизации числа состояний конечного автомата взяты из статьи Hopcroft [1971].

ГЛАВА 5

Алгоритмы на графах

Многие научные и технические задачи можно сформулировать в терминах неориентированных или ориентированных графов. В этой главе мы обсудим некоторые из основных задач на графах, время решения которых полиномиально зависит от количества вершин и, следовательно, ребер графа. В центре нашего внимания будут находиться задачи, в которых речь идет о связности графа. К ним относятся алгоритмы для нахождения остовных деревьев, двусвязных компонентов, сильно связных компонентов и путей между вершинами. В главе 10 мы изучим более сложные задачи о графах.

5.1. Остовное дерево минимальной стоимости

Пусть $G = (V, E)$ — связный неориентированный граф, для которого задана функция стоимости, отображающая ребра во множество действительных чисел. Напомним, что остовным деревом данного графа называется неориентированное дерево, содержащее все вершины графа. Стоимость остовного дерева определяется как сумма стоимостей его ребер. Наша цель — найти в графе G остовное дерево минимальной стоимости. Мы увидим, что остовное дерево минимальной стоимости в графе с e ребрами можно найти за время $O(e \log e)$ в общем случае и $O(e)$, если число e достаточно велико по сравнению с количеством вершин (см. упражнение 5.3). Многие алгоритмы нахождения остовного дерева основаны на следующих двух леммах.

Лемма 5.1. Пусть $G = (V, E)$ — связный неориентированный граф и $S = (V, T)$ — его остовное дерево. Тогда

а) для любых двух вершин v_1 и v_2 из V путь между v_1 и v_2 в S является единственным;

б) если в S добавить ребро из $E - T$, то возникнет ровно один цикл.

Доказательство. Утверждение (*а*) очевидное, поскольку если бы существовало более одного пути, то в дереве S существовал бы цикл.

Утверждение (*б*) тоже очевидное, поскольку между концами добавляемого ребра уже существовал один путь. □

Лемма 5.2. Пусть $G = (V, E)$ — связный неориентированный граф и c — функция стоимости, заданная на его ребрах. Пусть $\{(V_1, T_1), (V_2, T_2), ..., (V_k, T_k)\}$ — произвольный остовный лес графа G при $k > 1$ и $T = \bigcup_{i=1}^{k} T_i$. Допустим, что $e = (v, w)$ — ребро минимальной стоимости в $E - T$, причем $v \in V_1$ и $w \notin V_1$. Тогда найдется остовное дерево графа G, содержащее $T \cup \{e\}$, стоимость которого не больше стоимости любого остовного дерева графа G, содержащего T.

Доказательство. Допустим обратное и обозначим через $S' = (V, T')$ остовное дерево графа G, содержащее остовный лес T и не содержащее ребра e, стоимость которого меньше стоимости любого остовного дерева графа G, содержащего $T \cup \{e\}$.

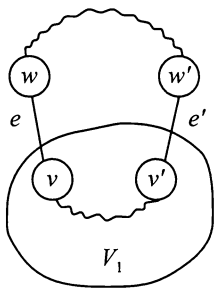


Рис. 5.1. Цикл в графе G

По лемме 5.1, *б*, добавление ребра e к остовному дереву S' создает цикл (рис. 5.1). Этот цикл должен содержать такое ребро $e' = (v', w')$, отличное от e, что $v' \in V_1$ и $w \notin V_1$. По условию $c(e) \le c(e')$.

Рассмотрим граф S, созданный добавлением ребра e к остовному дереву S' и удалением ребра e' из остовного дерева S'. В дереве S нет циклов, поскольку единственный цикл e' был разорван удалением ребра. Кроме того, все вершины в множестве V все еще соединены, поскольку существует путь между v' и w' в S. Следовательно, S — остовное дерево графа G. Поскольку $c(e) \le c(e')$, стоимость дерева S не больше стоимости дерева S'. Однако S содержит и лес T, и ребро e, что противоречит минимальности дерева S'.□

Опишем алгоритм поиска остовного дерева минимальной стоимости для неориентированного графа $G = (V, E)$. Этот алгоритм по существу не отличается от алгоритма из примера 4.1. Он работает с коллекцией VS непересекающихся множеств вершин. Каждое множество W из VS представляет связное множество вершин, образующее остовное дерево в остовном лесу, представленном коллекцией VS. Ребра выбираются из множества E в порядке возрастания их стоимости. Ребра (v, w) рассматриваются по очереди. Если v и w принадлежат одному и тому же множеству из VS, то ребро (v, w) исключается из рассмотрения. Если v и w принадлежат разным множествам W_1 и W_2 (это означает, что W_1 и W_2 еще не соединены), то объединяем их в одно множество и добавляем (v, w) к множеству T ребер, входящих в окончательное остовное дерево. Здесь можно воспользоваться алгоритмом объединения непересекающихся множеств, описанным в разделе 4.7. Применив лемму 5.2 и проведя несложную индукцию по числу выбранных ребер, заключаем, что это ребро содержится по крайней мере в одном остовном дереве минимальной стоимости графа G.

```
    begin
1.      T ← ∅;
2.      VS ← ∅;
3.      создать очередь с приоритетами Q, содержащую все ребра из E;
4.      for v ∈ V do добавить {v} к VS;
5.      while ||VS|| > 1 do
            begin
6.              выбрать ребро (v, w) минимальной стоимости в Q;
7.              удалить (v, w) из Q;
8.              if v и w принадлежат различным множествам W1 и W2
                        из VS then
                    begin
9.                      заменить W1 и W2 на W1 ∪ W2 в VS;
10.                     добавить (v, w) к T
                    end
            end
    end
```

Рис. 5.2. Алгоритм построения остовного дерева минимальной стоимости

Алгоритм 5.1. Остовное дерево минимальной стоимости (алгоритм Крускала)

Вход. Неориентированный граф $G = (V, E)$ с функцией стоимости c, заданной на его ребрах.

Выход. Остовное дерево $S = (V, T)$ минимальной стоимости для G.

Метод. Программа приведена на рис. 5.2. □

Пример 5.1. Рассмотрим неориентированный граф, показанный на рис. 5.3. Перечислим его ребра в порядке возрастания их стоимостей.

Ребро	Стоимость	Ребро	Стоимость
(v_1, v_7)	1	(v_4, v_5)	17
(v_3, v_4)	3	(v_1, v_2)	20
(v_2, v_7)	4	(v_1, v_6)	23
(v_3, v_7)	9	(v_5, v_7)	25
(v_2, v_3)	15	(v_5, v_6)	28
(v_4, v_7)	16	(v_6, v_7)	36

На самом деле на шаге 3 алгоритма 5.1 ребра не упорядочиваются, а хранятся в виде кучи, 2-3-дерева или какой-нибудь другой удобной структуры данных до востребования. Куча из раздела 3.4 представляет собой почти идеальный способ реализации очереди с приоритетами. Повторное обращение в строке 6 с целью найти ребро минимальной стоимости является основной операцией с кучей. Кроме того, число ребер, выбираемых в строке 6 для построения остовного дерева, часто меньше $||E||$. В таких ситуациях мы экономим время, поскольку никогда не упорядочиваем E полностью.

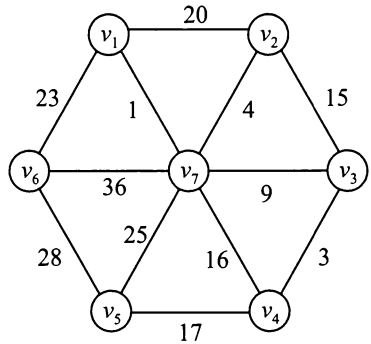


Рис. 5.3. Неориентированный граф с указанными стоимостями его ребер

Вначале каждое ребро находится в одноэлементном множестве, входящем в коллекцию *VS* и состоящем из самого этого ребра. Наименьшую стоимость имеет ребро (v_1, v_7), поэтому оно добавляется к дереву и множества $\{v_1\}$ и $\{v_7\}$ в коллекции *VS* объединяются. Затем рассматривается ребро (v_3, v_4). Поскольку вершины v_3 и v_4 принадлежат разным множествам в *VS*, добавляем ребро (v_3, v_4) в дерево и объединяем множества $\{v_3\}$ и $\{v_4\}$. Далее мы добавляем ребро (v_2, v_7) и объединяем множества $\{v_2\}$ и $\{v_1, v_7\}$. Кроме того, добавляем четвертое ребро (v_3, v_7) и объединяем множества $\{v_1, v_2, v_7\}$ и $\{v_3, v_4\}$.

Затем рассматриваем ребро (v_2, v_3). Обе вершины, v_2 и v_3, принадлежат одному и тому же множеству $\{v_1, v_2, v_3, v_4, v_7\}$. Следовательно, из v_2 в v_3 идет путь из ребер, уже вошедших в остовное дерево, и потому ребро (v_2, v_3) не добавляется. Вся последовательность шагов приведена на рис. 5.4. Остовное дерево, получающееся в результате, изображено на рис. 5.5. □

Теорема 5.1. Алгоритм 5.1 находит остовное дерево минимальной стоимости в связном графе G. Если в цикле, описанном строками 5–10, рассматривается d ребер, то затрачивается время $O(d \log e)$, где $e = ||E||$. Следовательно, временная сложность алгоритма 5.1 не превышает $O(e \log e)$.

Ребро	Операция	Множества в *VS* (связные компоненты)
(v_1, v_7)	Добавить	$\{v_1, v_7\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_4)	Добавить	$\{v_1, v_7\}, \{v_2\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_2, v_7)	Добавить	$\{v_1, v_2, v_7\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_7)	Добавить	$\{v_1, v_2, v_3, v_4, v_7\}, \{v_5\}, \{v_6\}$
(v_2, v_3)	Отклонить	
(v_4, v_7)	Отклонить	
(v_4, v_5)	Добавить	$\{v_1, v_2, v_3, v_4, v_5, v_7\}, \{v_6\}$
(v_1, v_2)	Отклонить	
(v_1, v_6)	Добавить	$\{v_1, \ldots, v_7\}$

Рис. 5.5. Последовательность действий по построению остовного дерева

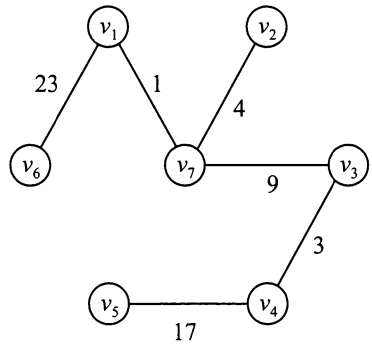


Рис. 5.4. Остовное дерево минимальной стоимости

Доказательство. Корректность алгоритма вытекает непосредственно из леммы 5.2 и того факта, что строки 8 и 9 сохраняют вершины в одном и том же множестве тогда и только тогда, когда они принадлежат одному и тому же дереву остовного леса, представленного коллекцией VS.

Для оценки временной сложности допустим, что в строках 5–10 требуется d итераций цикла. Если множество Q реализуется в виде очереди с приоритетами, то нахождение ребра минимальной стоимости в Q (строка 6) занимает $O(\log e)$ шагов. Если применяется быстрый алгоритм объединения непересекающихся множеств, то общее время нахождения всех множеств W_1 и W_2, содержащих v и w (строка 8), и замены их на их объединение (строка 9) не превосходит $O(eG(e))$. Остальная часть цикла, очевидно, занимает постоянное время, не зависящее от размера графа G. Инициализация очереди Q занимает время $O(e)$, а инициализация коллекции VS — $O(n)$, где n — число вершин в множестве V. □

5.2. Метод поиска в глубину

Рассмотрим обход вершин неориентированного графа в следующем порядке. Выбираем и посещаем некоторую начальную вершину v. Затем выбираем произвольное ребро (v, w), инцидентное v, и посещаем w. Вообще, пусть x — последняя посещенная вершина. Для продолжения процесса выбираем какое-нибудь не рассмотренное еще ребро (x, y), инцидентное x. Если вершина y уже посещалась, ищем другое новое ребро, инцидентное x. Если вершина y раньше не посещалась, переходим в y и заново начинаем поиск, начиная с нее. Пройдя все пути, начинающиеся в вершине y, возвращаемся в вершину x, т.е. в ту вершину, из которой впервые была достигнута вершина y. Затем продолжаем выбор нерассмотренных ребер, инцидентных вершине x, до тех пор, пока их список не будет исчерпан. Этот метод обхода вершин неориентированного графа называется *поиском в глубину*, поскольку процесс поиска идет в направлении вперед (вглубь) до тех пор, пока это возможно.

Метод поиска в глубину можно также осуществлять и в ориентированном графе. Если граф является ориентированным, то, находясь в вершине x, мы выбираем только ребра (x, y), выходящие из x. Обойдя все ребра, выходящие из вершины y, мы возвращаемся в вершину x даже тогда, когда в вершину y входят другие, еще не рассмотренные, ребра.

Если метод поиска в глубину применяется к связному неориентированному графу, то, как нетрудно показать, посещается каждая вершина и проходится каждое ребро. Если граф является несвязным, то отыскивается его связный компонент. После обхода связного компонента в качестве новой начальной вершины выбирается та, которая еще не посещалась, и начинается новый поиск.

Поиск в глубину на неориентированном графе $G = (V, E)$ разбивает ребра из множества E на два множества — T и B. Ребро (v, w) помещается в множество T, если вершина w не посещалась до того момента, когда мы, рассматривая ребро (v, w), оказались в вершине v. В противном случае ребро (v, w) включается в множество B. Ребра из множества T называются *древесными*, а ребра из множества B — *обратными*. Подграф (V, T) называется неориентированным лесом, который называется *остовным лесом поиска в глубину графа* G. Если этот лес состоит из единственного дерева, то подграф (V, T) называется *глубинным остовным деревом*. Если граф G — связный, то глубинный остовный лес является деревом. Корнем каждого дерева в глубинном остовном лесу считается вершина, в которой начинался этот поиск.

Алгоритм поиска в глубину приведен ниже.

Алгоритм 5.2. Метод поиска в глубину в неориентированном графе

Вход. Граф $G = (V, E)$, представленный списками смежности $L[v]$, $v \in V$.

Выход. Разбиение множества E на множества древесных и обратных ребер T и B соответственно.

Метод. Рекурсивная процедура SEARCH(v) на рис. 5.6 добавляет ребро (v, w) в множество T, если вершина w достигается первой при поиске на ребре, исходящем из вершины v.

```
      procedure SEARCH(v):
      begin
1.        помечаем вершину v как "старую";
2.        for для каждой вершины w в L[v] do
3.        if w помечена как "новая" then
              begin
4.                add(v, w) в T;
5.                SEARCH(w)
              end
      end
```

Рис. 5.6. Поиск в глубину

Полный алгоритм выглядит следующим образом.

```
      begin
6.        T ← ∅;
7.        for все v ∈ V do помечаем вершину v как "новую";
8.         while существует вершина v ∈ V, помеченная как "новая" do
9.             SEARCH(v)
      end
```

Все вершины во множестве E, не принадлежащие множеству T, считаются принадлежащими множеству B. Отметим, что если ребро (v, w) принадлежит множеству E, то вершины v и w принадлежат множествам $L[w]$ и $L[v]$ соответственно. Следовательно, мы не можем просто поместить ребро (v, w) в множество B, находясь в вершине v, если вершина w помечена как “старая”, поскольку вершина w может быть отцом вершины v.□

Пример 5.2. Будем изображать древесные ребра, образующие множество T, сплошными линиями, а обратные ребра, из которых состоит множество B, — пунктирными. Кроме того, корень дерева (или деревьев), т.е. начальная вершина, выбираемая в строке 8, будет изображаться выше всех, а сыновей каждой вершины будем располагать слева направо в том порядке, в котором их ребра добавлялись в строке 4 процедуры SEARCH. На рис. 5.7, *б*, показано возможное разбиение графа, изображенного на рис. 5.7, *а*, на множества древесных и обратных ребер, построенное в процессе поиска в глубину.

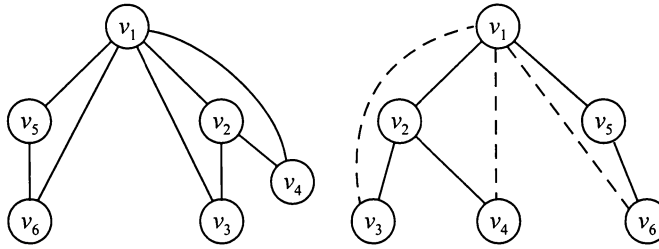


Рис. 5.7. Граф и глубинное остовное дерево

Вначале все вершины помечены как "новые". Допустим, что в строке 8 выбирается вершина v_1. Выполняя процедуру SEARCH(v_1), можно в строке 2 выбрать $w = v_2$. Поскольку вершина v_2 помечена как "новая", добавляем ребро (v_1, v_2) в T и вызываем SEARCH(v_2). Теперь можно было бы выбрать вершину v_1 из $L[v_2]$, но она уже помечен как "старая". Тогда выберем $w = v_3$. Поскольку v_3 — "новая" вершина, добавляем (v_2, v_3) в T и вызываем SEARCH(v_3). Каждая вершина, смежная с v_3, является теперь "старой", так что снова вызываем SEARCH(v_2).

Выполняя процедуру SEARCH(v_2), находим ребро (v_2, v_4), добавляем его к T и вызываем SEARCH(v_4). Заметим, что на рис. 5.7, *б*, вершина v_4 расположена справа от найденного ранее сына v_3 вершины v_2. Никакие "новые" вершины не являются смежными с v_4, поэтому опять вызываем SEARCH(v_2). На этот раз мы не обнаруживаем "новых" вершин, смежных с v_2, поэтому вызываем SEARCH(v_1). Далее процедура SEARCH(v_1) находит v_5, а SEARCH(v_5) находит v_6. Все вершины оказываются на дереве и помечены как "старые". Таким образом, выполнение алгоритма завершается. Если бы граф не был связным, то цикл в строках 8-9 надо было повторить для каждой компоненты. □

Теорема 5.2. Временная сложность алгоритма 5.2 на графе с n вершинами и e ребрами составляет O(max(n, e)).

Доказательство. Если список вершин составлен и один раз просмотрен, то строка 7 и поиск "новой" вершины в строке 8 требуют $O(n)$ шагов. Время, затрачиваемое на SEARCH(v), если не считать рекурсивных вызовов этой процедуры, пропорционально количеству вершин, смежных с вершиной v. Процедура SEARCH(v) для заданной вершины v вызывается только один раз, поскольку после вызова вершина v помечается как "старая". Таким образом, временная сложность всей процедуры SEARCH составляет O(max(n, e)). Теорема доказана. □

Мощность метода поиска в глубину частично отражена в приведенной ниже лемме, в которой утверждается, что каждое ребро неориентированного графа G либо принадлежит глубинному остовному лесу, либо соединяет предка с потомком в некотором дереве, входящем в глубинный остовный лес. Таким образом, все ребра графа G, как древесные, так и обратные, соединяют две вершины, одна из которых является предком другой в остовном лесу.

Лемма 5.3. Если (v, w) — обратное ребро, то либо v — предок вершины w, либо w — предок вершины v в остовном лесу.

Доказательство. Без потери общности будем считать, что v посещается раньше w, т.е. SEARCH(v) вызывается раньше, чем SEARCH(w). Следовательно, в тот момент, когда будет достигнута вершина v, вершина w все еще будет помечена как "новая". Все "новые" вершины, посещенные во время вызова процедуры SEARCH(v), станут потомками вершины v в остовном лесу. Но процедура SEARCH(v) не может закончиться, пока не будет достигнута вершина w, поскольку w находится в списке $L[v]$. □

Метод поиска в глубину задает на вершинах остовного леса естественный порядок: вершины можно пометить в том порядке, в каком они посещались, если инициализировать COUNT единицей между строками 6 и 7 алгоритма 5.2 и вставить в начало процедуры SEARCH инструкции

```
DFNUMBER[v] ← COUNT;
COUNT ← COUNT + 1;
```

Тогда вершины леса получат метки от 1 до их числа в лесу.

Очевидно, что для графа с n вершинами эти метки можно присвоить за время $O(n)$. Такой порядок соответствует прямому порядку обхода каждого дерева в результирующем остовном лесу. В дальнейшем будем считать, что все глубинные остовные леса размечены именно так. Часто мы будем интерпретировать метки вершин как их имена. Таким образом, мы можем использовать выражения вида "$v < w$", где v и w — вершины.

Пример 5.3. Метод поиска в глубину задает на графе, изображенном на рис. 5.7, *а*, следующий порядок: v_1, v_2, v_3, v_4, v_5, v_6. В этом можно убедиться, либо проследив порядок вызовов процедуры SEARCH, либо обойдя дерево на рис. 5.7, *б*, в прямом порядке.

Подчеркнем, что если v — прямой предок вершины w, то $v < w$. Аналогично, если вершина v находится слева от вершины w в дереве, то $v < w$.

5.3. Двусвязность

Рассмотрим приложение метода поиска в глубину к выявлению двусвязных компонентов в неориентированном графе. Пусть $G = (V, E)$ — связный неориентированный граф. Вершину a называют *точкой сочленения* графа G, если существуют такие вершины v и w, что v, w и a отличаются друг от друга и любой путь между v и w содержит вершину a. Иначе говоря, a — точка сочленения графа G, если удаление вершины a расщепляет G на две или более части. Граф G называется *двусвязным*, если для любой тройки несовпадающих вершин v, w и a существует путь между v и w, не содержащий a. Таким образом, неориентированный связный граф является двусвязным тогда и только тогда, когда в нем нет точек сочленения.

На множестве ребер графа G можно задать естественное отношение, полагая, что для ребер e_1 и e_2 это отношение выполняется, если $e_1 = e_2$ или существует цикл, содержащий e_1 и e_2. Легко показать, что это отношение является отношением эквивалентности,[1] разбивающим множество ребер графа G на такие классы эквивалентности E_1, E_2, ..., E_k, что два различных ребра принадлежат одному и тому же классу тогда и только тогда, когда они лежат на общем цикле. Для $1 \le i \le k$ обозначим через V_i множество вершин, лежащих на ребрах из класса E_i. Каждый граф $G_i = (V_i, E_i)$ называется *двусвязным компонентом* графа G.

Пример 5.4. Рассмотрим неориентированный граф на рис. 5.8, *а*. Вершина v_4 является точкой сочленения, поскольку каждый путь между v_1 и v_7 проходит через v_4. Классы эквивалентности, состоящие из ребер, лежащих на общих циклах, имеют следующий вид:

$$\{(v_1, v_2), (v_1, v_3), (v_2, v_3)\},$$
$$\{(v_2, v_4), (v_2, v_5), (v_4, v_5)\},$$
$$\{(v_4, v_6)\},$$
$$\{(v_6, v_7), (v_6, v_8), (v_6, v_9), (v_7, v_8), (v_8, v_9)\}.$$

Эти классы порождают двусвязные компоненты, изображенные на рис. 5.8, *б*. Обратите внимание на то, что ребро (v_4, v_6), будучи само классом эквивалентности (оно не входит ни в один цикл), порождает двусвязный компонент, состоящий из v_4 и v_6. □

[1] R называется *отношением эквивалентности* на множестве S, если R *рефлексивно* (aRa для всех $a \in S$), *симметрично* (из aRb следует bRa для всех a, $b \in S$) и *транзитивно* (из aRb и bRc следует aRc) . Легко показать, что отношение эквивалентности на множестве S разбивает его на непересекающиеся классы эквивалентности. (*Классом эквивалентности* называется подмножество $[a] = \{b \mid bRa\}$.)

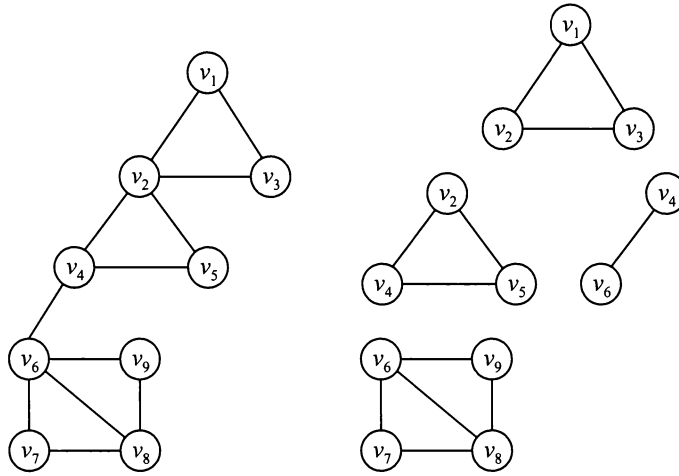


Рис. 5.8. Неориентированный граф (*а*) и его двусвязные компоненты (*б*)

Полезную информацию о двусвязности дает следующая лемма.

Лемма 5.4. Пусть $G_i = (V_i, E_i)$, $1 \le i \le k$ — двусвязные компоненты связного неориентированного графа $G = (V, E)$. Тогда

1) граф G_i является двусвязным для каждого i, $1 \le i \le k$;
2) для всех $i \ne j$ множество $V_i \cap V_j$ содержит не более одной вершины;
3) точка a является точкой сочленения графа G тогда и только тогда, когда $a \in V_i \cap V_j$ для некоторых $i \ne j$.

Доказательство.

1. Допустим, найдутся такие три неодинаковые вершины v, w и a в V_i, что все пути в графе G_i между v и w проходят через вершину a. Тогда, очевидно, что (v, w) не является ребром из E_i. Следовательно, во множестве E_i есть неодинаковые ребра (v, v') и (w, w'), а в G_i есть цикл, который их содержит. По определению двусвязного компонента все ребра и вершины, принадлежащие этому циклу, лежат в E_i и V_i соответственно. Следовательно, существуют два пути между вершинами v и w, причем только один из них содержит вершину a. Это — противоречие.
2. Допустим, множеству $V_i \cap V_j$ принадлежат две разные вершины v и w. Тогда существуют цикл C_1 в G_i, содержащий v и w, и цикл C_2 в G_j, также содержащий v и w. Поскольку множества E_i и E_j не пересекаются, множества ребер, входящих в C_1 и C_2, тоже не пересекаются. Тем не менее из ребер этих ци-

клов можно построить новый цикл, содержащий вершины v и w. Отсюда следует, что хотя бы одно ребро в E_i совпадает с каким-то ребром в E_j. Таким образом, вопреки предположению E_i и E_j не являются классами эквивалентности.

3. Пусть a — точка сочленения графа G. Тогда существуют такие две вершины v и w, что v, w и a различны и каждый путь между v и w содержит a. Поскольку граф G является связным, хотя бы один такой путь существует. Пусть (x, a) и (y, a) — два ребра, лежащие на некотором пути между v и w, инцидентные вершине a. Если существует цикл, содержащий эти два ребра, то некоторый путь между v и w не содержит a. Следовательно, (x, a) и (y, a) входят в разные двусвязные компоненты, а вершина a принадлежит пересечению множеств их вершин.

И наоборот, если $a \in V_i \bigcap V_j$, то существуют ребра (x, a) и (y, a) соответственно в E_i и E_j. Поскольку они не лежат оба на одном и том же цикле, всякий путь из x в y содержит вершину a. Следовательно, a — точка сочленения. □

Метод поиска в глубину особенно полезен для нахождения двусвязных компонентов неориентированного графа. Отчасти это связано с тем, что, согласно лемме 5.3, в нем нет поперечных ребер. Иначе говоря, если вершина v — не предок и не потомок вершины w в остовном лесу, то v и w не могут соединяться никаким ребром.

Если a — точка сочленения, то удаление ребра a и всех ребер, инцидентных ей, расщепляет граф G по крайней мере на две части.

Одна из них состоит из сына s вершины a и всех ее потомков в глубинном остовном дереве. Следовательно, в этом остовном дереве вершина a должна иметь сына s, потомки которого не соединены обратными ребрами с прямыми предками вершины a. И наоборот, из-за отсутствия поперечных ребер вершина a, не являющаяся корнем, является точкой сочленения, если никакой потомок некоторого ее сына не соединен обратным ребром ни с каким прямым предком вершины a. Корень глубинного остовного дерева является точкой сочленения тогда и только тогда, когда он имеет не менее двух сыновей.

Пример 5.5. Остовное дерево, построенное методом поиска в глубину, для графа, изображенного на рис. 5.8, *а*, показано на рис. 5.9. Точками сочленения являются v_2, v_4 и v_6. Вершина v_2 имеет сына v_4, и ни из какого потомка вершины v_4 не выходит обратного ребра к прямому предку вершины v_2. Аналогично вершина v_4 имеет сына v_6, а вершина v_6 — сына v_8 с такими же свойствами. □

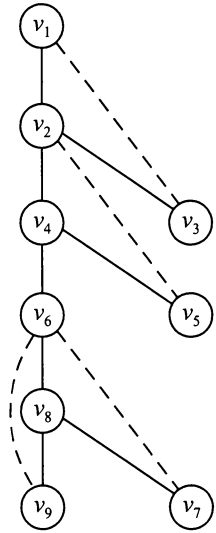


Рис. 5.9. Глубинное остовное дерево

Высказанные выше идеи сформулированы в следующей лемме.

Лемма 5.5. Пусть $G = (V, E)$ — связный неориентированный граф, а $S = (V, T)$ — его глубинное остовное дерево. Вершина a является точкой сочленения графа G тогда и только тогда, когда выполнено одно из условий:

1) a — корень и имеет более одного сына;
2) a — не корень, и у некоторого его сына s нет обратных ребер между потомками вершины s (в том числе самим s) и прямыми предками вершины a.

Доказательство. Легко показать, что корень является точкой сочленения тогда и только тогда, когда у него больше одного сына. Эту часть оставляем в качестве упражнения.

Предположим, что условие 2 выполнено. Пусть f — отец вершины a. Из леммы 5.3 следует, что каждое обратное ребро идет из некоторой вершины к его же предку. Следовательно, любое обратное ребро, выходящее из потомка v вершины s, идет к предку вершины v. По условию леммы обратное ребро не может идти к прямому предку вершины a. Значит, оно идет либо к вершине a, либо к потомку вершины s. Таким образом, всякий путь из s в f содержит вершину a, откуда следует, что a — точка сочленения.

Для доказательства обратного утверждения допустим, что a — точка сочленения, отличная от корня. Пусть x и y — различные вершины, отличные от a и такие, что всякий путь в G между x и y содержит a. По крайней мере, одна из вершин x и y, скажем x, является прямым потомком вершины a в дереве S; в противном случае в графе G существовал бы путь между x и y, проходящий по ребрам из множества T и не содержащий вершину a. Пусть s — такой сын вершины a, что x — потомок вершины s (возможно, $x = s$). Тогда либо между потомком v вершины s и прямым предком w вершины a нет обратного ребра и, значит, условие 2 выполняется, либо такое ребро (v, w) существует. В этой ситуации надо отдельно рассмотреть два варианта.

Вариант 1. Предположим, что y — не потомок вершины a. Тогда из x в v и далее в w и y идет путь, не содержащий a. Это — противоречие (рис. 5.10, *а*).

Вариант 2. Предположим, что y — потомок вершины a. Ясно, что y — не потомок вершины s, поскольку иначе существовал бы путь из x в y, не содержащий a. Пусть s' — такой сын вершины a, что y — потомок вершины s'. Тогда либо между потомком v' вершины s' и прямым предком w' вершины a нет обратного ребра и, значит, условие 2 выполняется, либо такое ребро (v', w') существует. В последнем случае из x в v и далее в w', v' и y идет путь, не содержащий вершину a. Это — противоречие (см. рис. 5.10, *б*). Приходим к выводу, что условие 2 выполнено.□

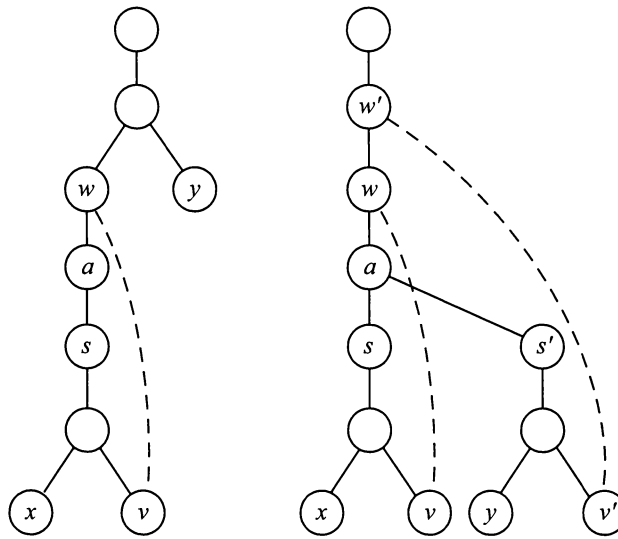


Рис. 5.10. Контрпримеры путей

Пусть T и B — множества соответственно древесных и обратных ребер, построенных методом поиска в глубину в связном неориентированном графе $G = (V, E)$. Будем считать, что номера вершин в множестве V присвоены в ходе поиска в глубину. Для каждой вершины v из множества V определим функцию

$$\texttt{LOW}[v] = \texttt{MIN}(\{v\} \cup \{w \mid \text{существует такое обратное ребро } (x, w) \in B, \text{ что } x \text{ — потомок вершины } v, \text{ а } w \text{ — предок вершины } v \text{ в глубинном остовном лесу } (V, T)\}). \quad (5.1)$$

Прямая нумерация означает, что если x — потомок вершины v, а (x, w) — обратное ребро, причем $w < v$, то w — прямой предок вершины v. Следовательно, по лемме 5.5, если v — не корень, то v является точкой сочленения тогда и только тогда, когда имеет сына s, для которого $\texttt{LOW}[s] \geq v$.

Выразив значение $\texttt{LOW}[v]$ через вершины, смежные с вершиной v, с помощью обратных ребер и значений функции LOW на сыновьях вершины v, функцию LOW можно включить в процедуру SEARCH. Точнее говоря, значение $\texttt{LOW}[v]$ можно вычислить, определив минимальное значение среди вершин w, удовлятворяющих одному из следующих свойств:

1) $w = v$, или

2) $w = \texttt{LOW}[s]$ и s — сын вершины v, или

3) (v, w) — обратное ребро в множестве B.

Минимальное значение w можно будет определить, как только будет исчерпан список $L[v]$ вершин, смежных с вершиной v. Таким образом, определение (5.1) эквивалентно равенству

$$\texttt{LOW}[v] = \texttt{MIN}(\{v\} \cup \{\texttt{LOW}[s] | s \text{ — сын вершины } v\} \cup \{w | (v, w) \in B\}). \quad (5.2)$$

В новый вариант процедуры SEARCH, приведенный на рис. 5.11, мы включили и переименование вершин при первом посещении, и вычисление функции LOW. В строке 4 элемент $\texttt{LOW}[v]$ инициализируется его максимально возможным значением. Если v имеет сына w в глубинном остовном лесу, то значение $\texttt{LOW}[v]$ в строке 11 корректируется, если оно оказывается больше, чем $\texttt{LOW}[w]$. Если вершина v соединена обратным ребром с вершиной w, то в строке 13 элементу $\texttt{LOW}[v]$ присваивается значение $\texttt{DFNUMBER}[w]$ при условии, что номер вершины w при поиске в глубину меньше текущего значения $\texttt{LOW}[v]$. Инструкция в строке 12 проверяет, не является ли ребро (v, w) обратным из-за того, что w — отец вершины v в глубинном остовном дереве. Таким образом, рис. 5.11 реализует формулу (5.2).

```
      procedure SEARCHB(v):
      begin
1.        пометить вершину v как "старую";
2.        DFNUMBER[v] ← COUNT;
3.        COUNT ← COUNT + 1;
4.        LOW[v] ← DFNUMBER[v];
5.        for для каждой вершины w ∈ L[v] do
6.            if вершина w помечена как "новая" then
                  begin
7.                    добавить (v, w) в T;
8.                    FATHER[w] ← v;
9.                    SEARCHB(w);
10.                   if LOW[w] ≥ DFNUBMER[v] then
                          обнаружен двусвязный компонент;
11.                   LOW[v] ← MIN(LOW[v], LOW[w])
                  end
12.           else if w - не FATHER[v] then
13.               LOW[v] ← MIN(LOW[v], DFNUMBER[w])
      end
```

Рис. 5.11. Поиск в глубину с вычислением функции LOW

Вычислив значение LOW[v] для каждой вершины v, легко найти точки сочленения. Сначала изложим полный алгоритм, а затем докажем его корректность и покажем, что его временная сложность имеет порядок $O(e)$.

Алгоритм 5.3. Нахождение двусвязных компонентов

Вход. Связный неориентированный граф $G = (V, E)$.

Выход. Список ребер каждого двусвязного компонента графа G.

Метод.

1. Вначале полагаем T равным $\varnothing$ и COUNT = 1. Помечаем каждую вершину во множестве V как "новую", выбираем произвольную вершину v_0 в V и вызываем процедуру SEARCHB(v_0) (рис. 5.11), чтобы построить глубинное остовное дерево $S = (V, T)$ и вычислить LOW(v) для каждого v из V.
2. Когда в строке 5 процедуры SEARCHB выбирается вершина w, помещаем ребро (v, w) в STACK (т.е. стек для ребер), если его там еще нет.[2] Когда в строке 10 обнаруживается пара (v, w), для которой w — сын вершины v и LOW[w] ≥ v, выталкиваем из STACK все ребра, включая (v, w). Эти ребра образуют двусвязный компонент графа G.□

[2]Заметим, что если (v, w) — ребро, то $v \in L[w]$ и $w \in L[v]$. Поэтому ребро (v, w) встречается дважды: один раз при посещении вершины v и второй раз — при посещении вершины w. Узнать, попало ли ребро (v, w) в STACK, можно, проверив, что $v < w$ и w — "старая вершина" или $v > w$ и w = FATHER[v].

Пример 5.6. Остовное дерево с рис. 5.9, построенное методом поиска в глубину, воспроизведено на рис. 5.12. Его вершины переименованы в соответствии с функцией DFNUMBER и указаны значения функции LOW. Например, процедура SEARCHB(6) устанавливает, что LOW[6] = 4, поскольку существует обратное ребро (6, 4). Тогда процедура SEARCHB(5), которая вызвала SEARCHB(6), полагает LOW[5] = 4, так как 4 меньше начального значения LOW[5], равного 5.

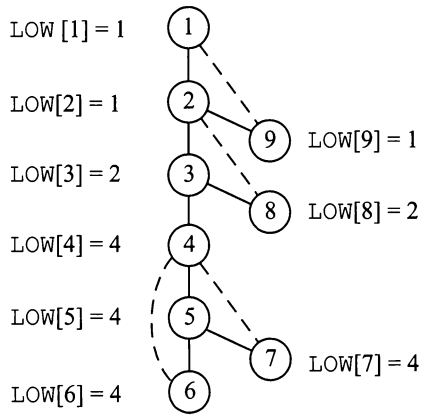


Рис. 5.12. Остовное дерево с рис. 5.9 со значениями функции LOW

Завершив выполнение процедуры SEARCHB(5), обнаруживаем (строка 10), что LOW[5] = 4. Следовательно, 4 — точка сочленения. В этот момент стек содержит такие ребра (от дна к вершине):

$$(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 4), (5, 7), (7, 4).$$

Поэтому выталкиваем все ребра, включая (4, 5). Таким образом, мы выводим ребра (7, 4), (5, 7), (6, 4), (5, 6) и (4, 5), которые являются ребрами первого найденного двусвязного компонента. Закончив выполнение процедуры SEARCHB(2), обнаруживаем, что LOW[2] = 1, и опустошаем стек, хотя 1 не является точкой сочленения. Это гарантирует, что двусвязный компонент, содержащий корень, тоже будет обнаружен. □

Теорема 5.3. Алгоритм 5.3 правильно находит двусвязные компоненты графа G с e ребрами за время $O(e)$.

Доказательство. Для доказательства того, что шаг 1 выполняется за время $O(e)$, необходимо просто обобщить аналогичное свойство процедуры SEARCH (теорема 5.2). Шаг 2 просматривает каждое ребро один раз, помещает его в стек и впоследствии выталкивает его оттуда. Следовательно, временная сложность шага 2 равна $O(e)$.

Что касается корректности алгоритма, то лемма 5.5 гарантирует, что точки сочленения будут найдены правильно. Даже если корень не является точкой сочленения, алгоритм обращается с ним как с точкой сочленения, чтобы выделить двусвязный компонент, содержащий корень.

Мы должны доказать, что если LOW[w] $\geq v$, то по завершении процедуры SEARCHB(w) ребра, расположенные в стеке над (v, w), будут в точности ребрами из двусвязного компонента, содержащего (v, w). Это доказывается индукцией по числу b двусвязных компонентов в G. Базис, т.е. вариант $b = 1$, тривиален, поскольку тогда v — корень дерева, (v, w) — единственное древесное ребро, выходящее из v, и по окончании процедуры SEARCHB(w) все ребра графа G находятся в стеке.

Теперь допустим, что предположение индукции выполняется для всех графов, имеющих b двусвязных компонентов. Пусть вызов процедуры SEARCHB(w) — первый вызов процедуры SEARCHB, по окончании которого LOW[w] $\geq v$, где (v, w) — древесное ребро. Поскольку никакие ребра не были удалены из стека, множество ребер, расположенных в стеке выше (v, w), состоит из всех ребер, инцидентных потомкам вершины w. Легко показать, что эти ребра в точности совпадают с ребрами того двусвязного компонента, куда входит (v, w). Удалив их из стека, алгоритм работает так, как работал бы на графе G', получающемся из G, если удалить двусвязный компонент, содержащий ребро (v, w). Осталось только сделать шаг индукции, поскольку G' состоит из b двусвязных компонентов. □

5.4. Метод поиска в глубину в ориентированном графе

Алгоритм 5.2 может также находить ориентированный остовный лес в ориентированном графе $G = (V, E)$, если определить список $L[v]$ вершин, смежных с вершиной v, как $\{w|(v, w)$ — ребро$\}$, т.е. $L[v]$ — это список вершин, которые являются началами ребер с хвостом v.

Пример 5.7. На рис. 5.13, *а*, изображен ориентированный граф, а на рис. 5.13, *б*, — его глубинный остовный лес. Как и ранее, будем изображать древесные ребра сплошными линиями, а остальные ребра — пунктиром.

Чтобы построить остовный лес, начнем с вершины v_1. Процедура SEARCH(v_1) вызывает SEARCH(v_2), а та — SEARCH(v_3). Последний вызов завершается, ничего не добавляя к дереву, поскольку единственное ребро с началом v_3 идет в вершину v_1, уже помеченную как "старая". Поэтому возвращаемся к процедуре SEARCH(v_2), которая добавляет v_4 в качестве второго сына вершины v_2. Процедура SEARCH(v_4) заканчивает работу, ничего не добавляя к лесу, поскольку вершина v_3 уже "старая".

Затем завершается выполнение процедуры SEARCH(v_2), поскольку все ребра, выходящие из v_2, к данному моменту просмотрены. Следовательно, возвращаемся обратно в v_1 и вызываем процедуру SEARCH(v_5). Последний вызов ничего не добавляет к дереву. Аналогично ничего не может добавить процедура SEARCH (v_1).

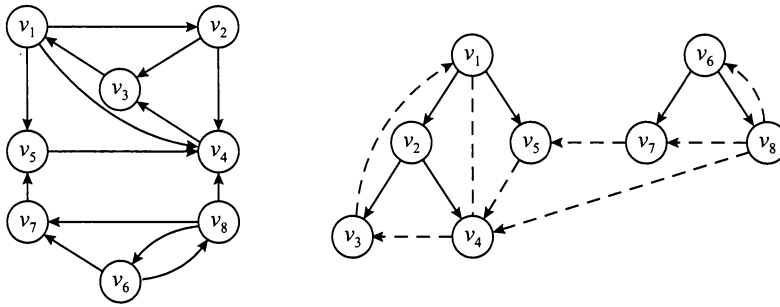


Рис. 5.13. Глубинный поиск ориентированного графа:
(*а*) ориентированный граф, (*б*) остовной лес

Теперь возьмем вершину v в качестве корня нового глубинного остовного дерева. Его построение аналогично предыдущему, и мы оставляем его читателям в качестве упражнения. Заметим, что вершины посещались в порядке $v_1, v_2, ..., v_8$. Следовательно, в процессе поиска в глубину вершине v_i был присвоен номер i, $1 \le i \le 8$. □

При поиске в глубину на ориентированном графе кроме древесных возникают еще три типа ребер. Это обратные ребра, такие, как (v_3, v_1) на рис. 5.13, *б*, поперечные справа налево ребра, такие, как (v_4, v_3) на том же рисунке, и прямые ребра, такие, как (v_1, v_4). Однако ни одно ребро не идет из вершины с меньшим номером, присвоенным в процессе поиска в глубину, в ребро с бо́льшим номером, если только последнее не является потомком первого. Это не случайно.

Объяснение аналогично объяснению того, почему в неориентированном графе нет поперечных ребер. Пусть (v, w) — ребро и вершина v была посещена до вершины w (т.е. $v < w$). Каждая вершина, которой приписывается номер между началом и концом процедуры SEARCH(v), становится потомком вершины v. Однако вершина w должна получить свой номер, когда рассматривается ребро (v, w), если только не получила его раньше. Если вершина w получает номер в то время, когда рассматривается ребро (v, w), то (v, w) становится древесным ребром, в противном случае — прямым ребром. Таким образом, не может быть такого поперечного ребра (v, w), что $v < w$.

Метод поиска в глубину в графе G разбивает множество его ребер на четыре класса.

1. *Древесные ребра*, идущие к новым вершинам в процессе поиска.
2. *Прямые ребра*, идущие от предков к подлинным потомкам, но не являющиеся ребрами дерева.
3. *Обратные ребра*, идущие от потомков к предкам (возможно, из вершины в себя).
4. *Поперечные ребра*, соединяющие вершины, которые не являются ни предками, ни потомками друг друга.

Ключевое свойство поперечных ребер устанавливается в следующей лемме.

Лемма 5.6. Если (v, w) — перекрестное ребро, то $v > w$, т.е. перекрестное ребро направлено справа налево.

Доказательство. Доказательство аналогично доказательству леммы 5.3 и остается в качестве упражнения. □

5.5. Сильная связность

В качестве примера эффективного алгоритма, который стал возможным благодаря методу поиска в глубину на ориентированном графе, рассмотрим задачу распознавания сильной связности ориентированного графа, т.е. существования на нем пути из каждой вершины в любую другую.

Определение. Пусть $G = (V, E)$ — ориентированный граф. Разобьем множество его вершин V на такие классы эквивалентности V_i, $1 \le i \le r$, что вершины v и w эквивалентны тогда и только тогда, когда на графе существуют пути из v в w и обратно. Пусть E_i, $1 \le i \le r$, — множество ребер, соединяющих пары вершин в множестве V_i. Графы $G_i = (V_i, E_i)$ называются *сильно связными компонентами* графа G. Хотя каждая вершина графа G принадлежит некоторому множеству V_i, в графе G могут быть ребра, не принадлежащие ни одному множеству E_i. Граф называется *сильно связным*, если он имеет только один сильно связный компонент.

Применим метод поиска в глубину для нахождения сильно связных компонентов графа. Сначала покажем, что вершины каждого сильно связного компонента образуют связный подграф в глубинном остовном лесу. Этот связный подграф представляет собой дерево, и его корень называется *корнем сильно связного компонента*. Однако не каждое дерево в глубинном остовном лесу непременно представляет какой-то сильно связный компонент.

Лемма 5.7. Пусть $G_i = (V_i, E_i)$ — сильно связный компонент ориентированного графа G, а $S = (V, T)$ — глубинный остовный лес графа G. Тогда вершины графа G_i вместе с ребрами из $E_i \cap T$ образуют дерево.

Доказательство. Пусть v и w — вершины из множества V_i. (Будем предполагать, что именами вершин являются номера, присвоенные им в ходе поиска в глубину.) Без потери общности будем считать, что $v < w$. Поскольку вершины v и w принадлежат одному и тому же сильно связному компоненту, то существует путь P в графе G_i, идущий из v в w. Пусть x — вершина на пути P с наименьшим номером (возможно, это сама вершина v). Путь P, дойдя до какого-нибудь потомка вершины x, не сможет выйти за пределы поддерева потомков вершины x, поскольку за пределы этого поддерева выходят лишь поперечные ребра и обратные ребра, идущие в вершины с номерами, меньшими x. (Поскольку потомки вершины x пронумерованы последовательно, начиная с x, поперечное или обратное ребро, выходящее из поддерева потомков вершины x, должно идти в вершину с номером, меньшим x.) Следовательно, w — потомок вершины x. Поскольку вершины пронумерованы в прямом порядке, то все вершины, номера которых лежат в диапазоне от x до w, также являются потомками вершины x. Поскольку $x \le v < w$, то v — потомок вершины x.

Итак, любые две вершины в графе G_i имеют общего предка в G_i. Пусть r — общий предок вершин графа G_i с наименьшим номером. Если v — вершина графа G_i, то любая вершина на пути из r в v в остовном дереве также является вершиной графа G_i. □

Сильно связные компоненты ориентированного графа G можно найти, построив корни компонентов в том порядке, в каком они встретились в последний раз в процессе поиска в глубину на графе G. Пусть $r_1, r_2, \ldots, r_k$ — эти корни в том порядке, в каком заканчивался их поиск в глубину (т.е. поиск вершины r_i заканчивался перед поиском вершины r_{i+1}). Тогда для каждого $i < j$ либо r_i лежит слева от r_j, либо r_i — потомок вершины r_j в глубинном остовном лесу.

Пусть G_i — сильно связный компонент с корнем r_i, $1 \le i \le k$. Тогда граф G_1 состоит из всех потомков вершины r_1, поскольку никакая вершина $r_j, j > 1$, не может быть потомком вершины r_1. Аналогично можно доказать следующую лемму.

Лемма 5.8. Для любого i, $1 \le i \le k$, граф G_i состоит из вершин, являющихся потомками вершины r_i и не принадлежащих ни одному из графов $G_1, G_2, \ldots, G_{i-1}$

Доказательство. Корень r_j для $j > 1$ не может быть потомком вершины r_i, поскольку вызов процедуры SEARCH(r_j) завершается после работы процедуры SEARCH(r_i). □

Для нахождения корней введем следующую функцию.

$$\texttt{LOWLINK}[v] = \texttt{MIN}(\{v\} \cup \{w|\text{существует поперечное или обратное ребро из некоторого потомка вершины } v \text{ в вершину } w, \text{ а корень сильно связного компонента, содержащего } w, \text{ является предком вершины } v\}). \quad (5.3)$$

На рис. 5.14 изображено поперечное ребро из потомка вершины v в вершину w, причем корень r сильно связного компонента, содержащего w, является предком вершины v.

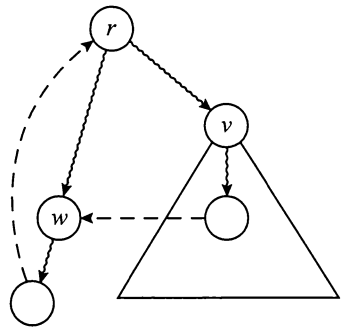


Рис. 5.14. Поперечное ребро, удовлетворяющее условию из определения функции LOWLINK

Мы вскоре увидим, как вычислять функцию LOWLINK в ходе поиска в глубину. Однако прежде охарактеризуем корни сильно связных компонентов в терминах значений этой функции.

Лемма 5.9. Пусть G — ориентированный граф. Для того чтобы вершина v была корнем сильно связного компонента графа G, необходимо и достаточно, чтобы выполнялось условие $\texttt{LOWLINK}[v] = v$.

Доказательство. Необходимость. Пусть v — корень сильно связного компонента графа G. По определению $\texttt{LOWLINK}[v] \le v$. Допустим, $\texttt{LOWLINK}[v] < v$. Тогда найдутся такие вершины w и r, что выполняются следующие условия.

1. В вершину w входит поперечное или обратное ребро, выходящее из потомка вершины v.
2. Вершина r — корень сильно связного компонента, содержащего w.
3. Вершина r — предок вершины v.
4. Выполняется неравенство $w < v$.

По условию 2 вершина r — предок вершины w, так что $r \leq w$. Следовательно, по условию 4 справедливо неравенство $r < v$, откуда в силу условия 3 вытекает, что r — прямой предок вершины v. Однако r и v должны принадлежать одному и тому же сильно связному компоненту, поскольку в графе G существуют пути из r в v и из v в r через w. Следовательно, вершина v не является корнем сильно связного компонента. Это — противоречие. Таким образом, LOWLINK[v] = v.

Достаточность. Пусть LOWLINK[v] = v. Если вершина v не является корнем сильно связного компонента, содержащего v, то корнем будет некоторый прямой предок r вершины v. Следовательно, существует путь P из v в r. Рассмотрим первое ребро в P, идущее из потомка вершины v в вершину w, не являющуюся потомком вершины v. Это ребро либо обратное и идет к предку вершины v, либо поперечное и идет в вершину, номер которой меньше v. В любом случае $w < v$.

Осталось показать, что вершины r и w принадлежат одному и тому же сильно связному компоненту графа G. Должен существовать путь из r в v, поскольку r — предок вершины v. Путь P идет из v через w в r. Следовательно, вершины r и w принадлежат одному и тому же сильно связному компоненту. Таким образом, LOWLINK[v] $\leq w < v$. Это — противоречие.

Значения функции LOWLINK легко вычислить в ходе поиска в глубину. Сильно связные компоненты также нетрудно найти. Для этого вершины графа G помещаются в стек в том порядке, в каком они посещаются во время поиска. Всякий раз, когда обнаруживается корень, все вершины соответствующего сильно связного компонента находятся в верхней части стека и выталкиваются наружу. Эта стратегия основана на лемме 5.8 и свойствах нумерации, порождаемой поиском в глубину.

Попав в вершину v в первый раз, полагаем LOWLINK[v] = v. Когда встречается обратное или поперечное ребро (v, w) и w принадлежит тому же сильно связному компоненту, что и некоторый предок вершины v, полагаем значение функции LOWLINK в вершине v равным меньшему из ее текущего значения и w. Когда встречается древесное ребро (v, w), рекурсивно просматривается поддерево с корнем w и вычисляется значение LOWLINK[w]. По возвращении к v значение функции LOWLINK в вершине v полагается равным меньшему из ее текущего значения и LOWLINK[w].

Чтобы определить, принадлежит ли w тому же компоненту, что и некоторый предок вершины v, достаточно проверить, находится ли еще вершина w в стеке для вершин. Эта проверка осуществляется с помощью массива, указывающего наличие вершины в стеке. Заметим, что по лемме 5.8, если w еще в стеке, то корень сильно связного компонента, содержащего w, является предком вершины v.

Рассмотрим модификацию процедуры SEARCH, которая вычисляет функцию LOWLINK. Она использует магазинный список STACK для вершин. Эта процедура приведена на рис. 5.15.

```
      procedure SEARCHC(v):
      begin
1.        пометить вершину v как "старую";
2.        DFNUMBER[v] ← COUNT;
3.        COUNT ← COUNT + 1;
4.        LOWLINK[v] ← DFNUMBER[v];
5.        затолкнуть v в СТЕК;
6.        for для каждой вершины w ∈ L[v] do
7.            if вершина w помечен как "новая" then
                  begin
8.                    SEARCHC(w);
9.                    LOWLINK[v] ← MIN(LOWLINK[v], LOWLINK[w])
                  end
              else
10.               if DFNUMBER[w] < DFNUMBER[v] и w ∈ STACK then
11.                   LOWLINK[v] ← MIN(DFNUMBER[w], LOWLINK[v]);
12.       if LOWLINK[v] = DFNUMBER[v] then
              begin
                  repeat
                      begin
13.                       вытолкнуть x из вершины STACK;
14.                       print x
                      end
15.               until x = v;
16.               print "конец сильно связного компонента"
              end
      end
```

Рис. 5.15. Процедура вычисления функции LOWLINK

Функция LOWLINK вычисляется в строках 4, 9 и 11. В строке 4 LOWLINK[v] инициализируется номером вершины v, присвоенным ей в ходе поиска в глубину. В строке 9 выполняется присваивание LOWLINK[v] = LOWLINK[w], если для некоторого сына w вершина LOWLINK[w] оказалась меньше текущего значения LOWLINK[v]. В строке 10 узнаем, является ли (v, w) обратным или поперечным ребром, и проверяем, не найден ли уже сильно связный компонент, содержащий w. Если нет, то корень сильно связного компонента, содержащего w, будет предком вершины v. В строке 11 значение LOWLINK[v] полагается равным номеру вершины w, присвоенному ему в ходе поиска в глубину, если оно раньше не получило меньшее значение.

Теперь сформулируем весь алгоритм нахождения сильно связных компонентов ориентированного графа.

Алгоритм 5.4. Нахождение сильно связных компонентов ориентированного графа

Вход. Ориентированный граф $G = (V, E)$.

Выход. Список сильно связных компонентов графа G.

Метод.

```
begin
    COUNT ← 1;
    for все v ∈ V do пометить вершину v как "новую";
    опустошить STACK;
    while существует вершина v, помеченная как "новая" do
        SEARCHC(v)
end □
```

Пример 5.8. Рассмотрим глубинный остовный лес на рис. 5.13. Этот лес воспроизведен на рис. 5.16, где указаны также значения функции LOWLINK. Вызов процедуры SEARCHC(3) заканчивает работу первым среди всех вызовов процедуры SEARCHC. Исследуя ребро (3, 1), полагаем значение LOWLINK[3] равным MIN(1, 3) = 1. По возвращении к SEARCHC(2) полагаем LOWLINK[2] = MIN(2, LOWLINK[3]) = 1. Затем вызываем SEARCHC(4) для исследования ребра (4, 3). Поскольку 3 < 4 и 3 еще находится в STACK, полагаем LOWLINK[4] = MIN(3, 4) = 3.

Затем возвращаемся к вызову SEARCHC(2) и полагаем LOWLINK[2] равным меньшему из чисел: LOWLINK[4] и текущего значения LOWLINK[2], который равен 1. Поскольку последнее значение меньше, то LOWLINK[2] не меняется. Возвращаемся к вызову SEARCHC(1), полагая LOWLINK[l] = MIN(1, LOWLINK[2]) = 1. Исследовав далее ребро (1, 4), ничего не делаем, поскольку (1, 4) — прямое ребро, и условие строки 10 процедуры SEARCHC не выполняется, так как 4 > 1.

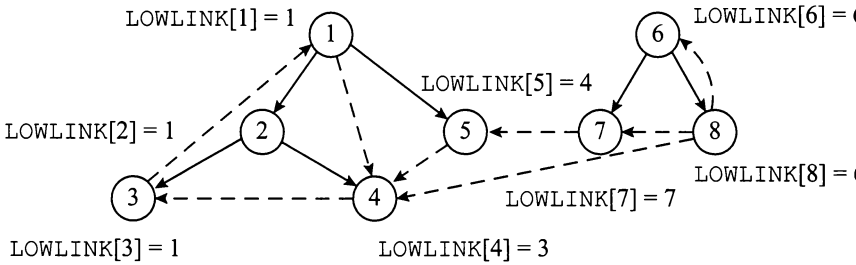


Рис. 5.16. Остовный лес с вычисленной функцией LOWLINK

Далее вызываем процедуру SEARCHC(5) и рассматриваем поперечное ребро (5, 4). Это вынуждает нас положить LOWLINK[5] = 4, поскольку 4 < 5 и 4 нет в STACK. Возвращаясь к вызову SEARCHC(1), полагаем значение LOWLINK[1] равным меньшему из предыдущего значения, равного 1, и LOWLINK[5], т.е. LOWLINK[5] = 1.

Затем, поскольку все ребра, выходящие из 1, уже рассмотрены и LOWLINK[1] = 1, выясняется, что 1 — корень сильно связного компонента. Этот компонент состоит из 1 и всех вершин, расположенных над ним в стеке. Поэтому стек опустошается и список вершин 1, 2, 3, 4, 5 печатается в качестве сильно связного компонента графа G.

Другие сильно связные компоненты — {7} и {6, 8}. Оставляем читателям закончить вычисление функции LOWLINK и сильно связных компонент, отправляясь от вершины 6. Заметим, что последние посещения корней сильно связных компонентов происходили в порядке 1, 7, 6. □

Теорема 5.4. Алгоритм 5.4 правильно находит сильно связные компоненты графа G за время $O(\max(n, e))$, где n — число вершин, а e — число ребер ориентированного графа G.

Доказательство. Нетрудно проверить, что время выполнения одного вызова процедуры SEARCHC(v), не считая рекурсивных вызовов SEARCHC, равно сумме константы и времени, пропорционального количеству ребер, выходящих из вершины v. Следовательно, все вызовы процедуры SEARCHC в совокупности занимают время, пропорциональное сумме количества вершин и количества ребер, поскольку SEARCHC вызывается только один раз для каждой вершины. Участки алгоритма 5.4, отличные от процедуры SEARCHC, очевидно, можно реализовать за время $O(n)$. Таким образом, оценка временной сложности получена.

Чтобы доказать корректность алгоритма, достаточно показать индукцией по числу завершенных вызовов процедуры SEARCHC, что по окончании SEARCHC(v) значение LOWLINK[v] вычислено правильно. После строк 12–16 процедуры SEARCHC вершина v становится корнем сильно связного компонента тогда и только тогда, когда LOWLINK[v] = v. Далее, в качестве результата выводятся только те потомки вершины v, которые не входят в компоненты, корни которых были найдены раньше v (по лемме 5.8). Иначе говоря, вершины, находящиеся в стеке выше вершины v, являются его потомками, а их корни не были найдены ранее v, поскольку эти вершины все еще находятся в стеке.

Чтобы доказать корректность вычисления функции LOWLINK, заметим, что на рис. 5.15 есть два места, где значение LOWLINK[v] могло быть меньше v, — это строки 9 и 11 процедуры SEARCHC. В первом случае w — сын вершины v и LOWLINK[w] < v. Тогда существует вершина x = LOWLINK[w], которую можно

достичь из потомка y вершины w через поперечное или обратное ребро. Кроме того, корень r сильно связного компонента, содержащего x, является предком вершины w. Поскольку $x < v$, то $r < v$, и, значит, r — прямой предок вершины v. Таким образом, значение LOWLINK[v] должно быть не больше, чем LOWLINK[w].

Во втором случае в строке 11 из вершины v идет поперечное или обратное ребро в вершину $w < v$, сильно связный компонент C которого еще не найден. Вызов процедуры SEARCHC на корне r из графа C еще не закончился, так что r должен быть предком вершины v. (Поскольку $r \leq w < v$, то либо r лежит слева от v, либо r — предок вершины v. Но если бы r был слева от v, то вызов процедуры SEARCHC(r) завершился бы.) Отсюда следует, что значение LOWLINK[v] должно быть не больше w.

Осталось доказать, что SEARCHC вычисляет значение LOWLINK[v] таким малым, каким оно должно быть. Предположим, что это не так, т.е. существует предок x вершины v, из которого в некоторую вершину y идет поперечное или обратное ребро, а корень r сильно связного компонента, содержащего y, является предком вершины v. Надо показать, что LOWLINK[v] оказывается не больше y.

Вариант 1. $x = v$. В силу индуктивного предположения и леммы 5.9 можно считать, что все сильно связные компоненты, найденные до сих пор, корректны. Тогда вершина y все еще должна быть в стеке, поскольку выполнение процедуры SEARCHC(r) еще не закончилось. Следовательно, в строке 11 значение LOWLINK[v] полагается равным y или меньшему числу.

Вариант 2. $x \neq v$. Пусть z — сын вершины v, потомком которого является x. Тогда в силу индуктивного предположения по окончании вызова SEARCHC(z) значение LOWLINK[z] должно быть равно y или меньшему числу. В строке 9 значение LOWLINK[v] становится именно таким, если раньше оно не было еще меньше. □

5.6. Задачи поиска путей

В этом разделе мы рассмотрим две часто встречающиеся задачи о путях, соединяющих вершины. В дальнейшем G считается ориентированным графом. *Транзитивным (и рефлексивным) замыканием* графа G называется граф G^*, содержащий то же множество вершин, что и G, но в котором из v в w идет ребро тогда и только тогда, когда в G существует путь из v в w, длина которого больше или равна нулю.

Задача поиска транзитивного замыкания графа тесно связана с *задачей о кратчайшем пути*. Поставим в соответствие каждому ребру e графа G неотрицательную стоимость $c(e)$. *Стоимость пути* — это сумма стоимостей образующих его

ребер. В задаче о кратчайшем пути для каждой пары вершин (v, w) среди всех путей из v в w необходимо найти пути минимальной стоимости.

Оказывается, что идеи, лежащие в основе лучших алгоритмов поиска транзитивного замыкания и кратчайшего пути, известных в настоящее время, являются (простыми) частными случаями идей, лежащих в основе решения задач поиска (бесконечного) множества всех путей между каждой парой вершин. Чтобы обсудить эту задачу во всей ее общности, введем специальную алгебраическую структуру.

Определение. *Замкнутым полукольцом* называется система (S, +, ·, 0, 1), где S — множество элементов, + и · — бинарные операции на множестве S, удовлетворяющие следующим пяти свойствам.

1. (S, +, 0) — *моноид*, т.е. множество S *замкнуто* относительно операции + ($a + b \in S$ для всех a и b из S), а сама операция + является *ассоциативной* ($(a + (b + c) = (a + b) + c$ для всех a, b, c из S) и 0 служит *единичным* элементом (т.е. $a + 0 = 0 + a = a$ для всех a из S). Например, тройка (S, ·, 1) является моноидом. Кроме того, мы предполагаем, что 0 является *аннулятором*, т.е. $a \cdot 0 = 0 \cdot a = 0$.
2. Операция + *коммутативна* ($a + b = b + a$)·и *идемпотентна* ($a + a = a$).
3. Операция · *дистрибутивна* относительно + ($a \cdot (b + c) = a \cdot b + a \cdot c$ и $(b + c) \cdot a = b \cdot a + c \cdot a$).
4. Если a_1, a_2, ..., a_i, ...— счетная последовательность элементов из S, то сумма $a_1 + a_2 + ... + a_i + ...$ существует и является единственной. Более того, ассоциативность, коммутативность и идемпотентность выполняются не только для конечных, но и для бесконечных сумм.
5. Операция · дистрибутивна не только относительно конечных, но и счетных сумм (это не следует из свойства 3). Таким образом, из свойств 4 и 5 следует, что

$$\left(\sum_i a_i\right) \cdot \left(\sum_j b_j\right) = \sum_{i,j} a_i \cdot b_j = \sum_i \left(\sum_j \left(a_i \cdot b_j\right)\right).$$

Пример 5.9. Приведем три системы замкнутых полуколец.

1. Пусть S_1 =({0, 1}, + , ·, 0, 1), а сложение и умножение заданы таблицами

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

Свойства 1–3 легко проверить. Относительно свойств 4 и 5 заметим, что счетная сумма равна нулю тогда и только тогда, когда все слагаемые равны нулю.

2. Пусть $S_2 = (R, \texttt{MIN}, +, +\infty, 0)$, где R — множество неотрицательных действительных чисел, включая $+\infty$. Легко проверить, что $+\infty$ служит единичным элементом для операции MIN, а 0 — для операции +.

3. Пусть Σ — конечный алфавит (т.е. множество символов) и $S_3 = (F_\Sigma, \cup, \cdot, \varnothing, \{\varepsilon\})$, где F_Σ — семейство множеств, состоящих из конечных строк символов из Σ, включая пустую строку ε (т.е. строку нулевой длины). Здесь символ $\cup$ обозначает операцию объединения множеств, а символ $\cdot$ — их конкатенацию.[3] Единичным элементом для операции $\cup$ служит $\varnothing$, а для операции $\cdot$ — $\{\varepsilon\}$. Свойства 1–3 читатели могут проверить сами. Относительно свойств 4 и 5 достаточно заметить, что счетные объединения определяются так: $x \in (A_1 \cup A_2 \cup ...)$ равносильно $x \in A_i$ для некоторого i. □

Центральной операцией в нашем анализе замкнутых полуколец является унарная операция *, называемая *замыканием*. Если $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо и $a \in S$, то a^* определяется как $\sum_{i=0}^{\infty} a^i$, где $a^0 = 1$ и $a^i = a \cdot a^{i-1}$. Иначе говоря, a^* равно бесконечной сумме $1 + a + a \cdot a + a \cdot a \cdot a + ...$. Из свойства 4 определения замкнутого полукольца следует, что $a^* \in S$. Из свойств 4 и 5 вытекает равенство $a^* = 1 + a \cdot a^*$. Заметим, что $0^* = 1^* = 1$.

Пример 5.10. Рассмотрим полукольца S_1, S_2 и S_3 из примера 5.9. В S_1 для $a = 0$ или 1 выполняется условие $a^* = 1$. В S_2 для всех $a \in R$ справедливо равенство $a^* = 0$. В S_3 выполняется условие $A^* = \{\varepsilon\} \cup \{x_1 x_2 ... x_k | k \geq 1$ и $x_i \in A$ при $1 \leq i \leq k\}$ для всех $A \in F_\Sigma$. Например, $\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$, т.е. $\{a, b\}^*$ состоит из всех строк символов a и b, включая пустую строку. По существу, $F_\Sigma = \mathcal{P}(\Sigma^*)$, где $\mathcal{P}(X)$ обозначает множество всех подмножеств множества X. □

Теперь предположим, что $G = (V, E)$ — ориентированный граф, в котором каждое ребро помечено элементом некоторого замкнутого полукольца $(S, +, \cdot, 0, 1)$.[4] Будем называть *меткой пути* произведение $(\cdot)$ меток ребер, образующих этот путь, перечисленных в порядке их прохождения. В частном случае метка пути нулевой длины равна 1 (единичному элементу относительно операции $\cdot$ в полу-

[3] Конкатенацией $A \cdot B$ множеств A и B называется множество $\{x | x = yz, y \in A$ и $z \in B\}$.

[4] Обратите внимание на аналогию между рассматриваемой ситуацией и конечным автоматом (см. *Hopcroft and Ullman* [1969] или *Aho and Ullman* [1972]), который обсуждается в разделе 9.1. Там вершины представляют собой состояния, а метки ребер — символы нз некоторого конечного алфавита.

кольце). Для каждой пары вершин (v, w) определим функцию $c(v, w)$ как сумму меток всех путей между v и w. Будем называть $c(v, w)$ *стоимостью* прохода из v в w. Будем считать, что сумма по пустому множеству путей равна 0 (единичному элементу относительно операции + полукольца). Заметим, что если граф G имеет циклы, то между вершинами v и w могут существовать бесконечно много путей, но аксиомы замкнутого полукольца гарантируют корректность определения функции $c(v, w)$.

Пример 5.11. Рассмотрим ориентированный граф на рис. 5.17, в котором каждое ребро помечено элементом полукольца S_1 из примера 5.9. Метка пути v, w, x равна $1 \cdot 1 = 1$. Простой цикл из w в w имеет метку $1 \cdot 0 = 0$. Вообще, каждый путь положительной длины, идущий из w в w, имеет метку 0. В то же время стоимость пути нулевой длины из w в w равна 1. Следовательно, $c(w, w) = 1$. □

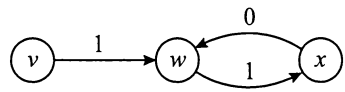


Рис. 5.17. Размеченый ориентированный граф

Построим алгоритм вычисления $c(v, w)$ для всех пар вершин v и w. Основными шагами этого алгоритма, выполоняемыми за единицу времени, будут операции +, · и * в произвольном замкнутом полукольце. Разумеется, для таких структур, как множество всех подмножеств строк над некоторым алфавитом, неясно, можно ли реализовать эти операции вообще, а не только за единичное время. Однако в полукольцах, которые мы будем изучать, эти операции легко выполнимы.

Алгоритм 5.5. Вычисление стоимости прохода между вершинами.

Вход. Ориентированный граф $G = (V, E)$, где $V = \{v_1, v_2, ..., v_n\}$ и функция разметки l: $(V \times V) \to S$, где $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо. Полагаем $l(v_i, v_j) = 0$, если $(v_i, v_j) \notin E$.

Выход. Для всех i и j в диапазоне от 1 до n элемент $c(v_i, v_j)$ из S, равный сумме меток всех путей из v_i в v_j.

Метод. Вычисляем C_{ij}^k для всех $1 \le i \le n$, $1 \le j \le n$ и $0 \le k \le n$. Величины C_{ij}^k — это суммы меток путей, идущих из v_i в v_j, все вершины которых, кроме, возиожно, концов, принадлежат множеству $\{v_1, v_2, ..., v_k\}$. Например, путь v_9, v_3, v_8 рассматривается при вычислении C_{98}^4 и C_{98}^3, но не C_{98}^2. Алгоритм формулируется следующим образом.

```
      begin
1.        for i ← 1 until n do C_ii^0 ← 1 + l(v_i, v_i)
2.        for 1 ≤ i, j ≤ n и i ≠ j do C_ij^0 ← l(v_i, v_j);
3.        for k ← 1 until n do
4.            for 1 ≤ i, j ≤ n do
5.                C_ij^k ← C_ij^(k-1) + C_ik^(k-1) · (C_kk^(k-1))* · C_kj^(k-1);
6.        for 1 ≤ i, j ≤ n do c(v_i, v_j) ← C_ij^n
      end
```

Теорема 5.5. Алгоритм 5.5 выполняет $O(n^3)$ операций +, · и * полукольца и вычисляет $c(v_i, v_j)$ для $1 \le i, j \le n$.

Доказательство. Нетрудно убедиться, что строка 5, состоящая из четырех операций, выполняется n^3 раз, а циклы **for** в строках 1, 2 и 6 проходятся не более n^2 раз каждый. Следовательно, достаточно $O(n^3)$ операций.

Чтобы доказать корректность алгоритма, необходимо доказать с помощью индукции по k, что C_{ij}^k — это сумма меток всех путей из v_i в v_j, не содержащих промежуточных (т.е. не концевых) вершин с индексами, больше, чем k. База индукции, т.е. вариант $k = 0$, тривиален в силу строк 1 и 2, поскольку любой такой путь имеет длину 0 или состоит из единственного ребра. Шаг индукции получается на основании строки 5, поскольку путь из v_i в v_j, не содержащий промежуточных вершин с индексами, больше, чем k, либо

1. не содержит промежуточных вершин с индексами, больше, чем $k-1$ (слагаемое C_{ij}^{k-1}), или
2. идет из v_i в v_k, затем несколько раз (а возможно, ни разу) из v_k в v_k и, наконец, из v_k в v_j, причем на этих путях нет промежуточных вершин с индексами, больше, чем $k-1$ (слагаемое $C_{ik}^{k-1} \cdot \left(C_{kk}^{k-1}\right)^* \cdot C_{kj}^{k-1}$).

Аксиомы замкнутого кольца гарантируют, что в строке 5 сумма меток всех этих путей вычисляется корректно. □

5.7. Алгоритм транзитивного замыкания

Уточним алгоритм 5.5 для двух интересных случаев. Первый случай — замкнутое полукольцо S_1, описанное в примере 5.9. В нем легко выполнять сложение

и умножение, а также и операцию *, поскольку 0* = 1* = 1. Действительно, поскольку 1 — это единичный элемент, строку 5 алгоритма 5.5 можно заменить на[5]

$$C_{ij}^{k} \leftarrow C_{ij}^{k-1} + C_{ik}^{k-1} \cdot C_{kj}^{k-1}. \tag{5.4}$$

Чтобы вычислить рефлексивно-транзитивное замыкание графа, определим функцию разметки:

$$l(v,w) = \begin{cases} 1, \text{ если } (v,w) \text{ — ребро,} \\ 0, \text{ если нет.} \end{cases}$$

Стоимость $c(v, w)$ равна 1 тогда и только тогда, когда из v в w идет путь длины 0 или больше. Это легко проверить, используя аксиомы замкнутого полукольца $\{0, 1\}$.

Пример 5.12. Рассмотрим граф, представленный на рис. 5.18, игнорируя числа на ребрах. Функция разметки $l(v_i, v_j)$ задается таблицей

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	0	0
v_3	0	1	0

$l(v_i, v_j)$

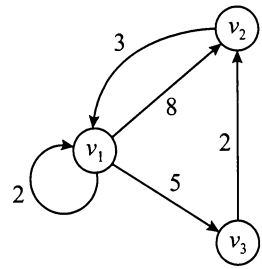


Рис. 5.18. Граф

Строка 1 алгоритма 5.5 выполняет присваивания $C_{11}^{0} = C_{22}^{0} = C_{33}^{0} = 1$. Строка 2 вычисляет $C_{ij}^{0} = l(v_i, v_j)$ для $i \neq j$, так что получаем для C_{ij}^{0} таблицу

[5]На первый взгляд, это утверждение означает, что достаточно исследовать только пути без циклов. Тем не менее, заметим, что с инструкцией (5.4) вместо строки 5 алгоритм 5.5 будет вычислять суммы по множествам путей, включающим не только все пути без циклов, но и некоторые другие.

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	0
v_3	0	1	1

C_{ij}^0

Теперь положим $k = 1$ и выполним цикл в строках 4 и 5, где вместо строки 5 стоит (5.4). Например, $C_{23}^1 = C_{23}^0 + C_{21}^0 \cdot C_{13}^0 = 0 + 1 \cdot 1 = 1$. Таблицы значений C_{ij}^1, C_{ij}^2 и C_{ij}^3 приведены на рис. 5.19. □

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	1
v_3	0	1	1

C_{ij}^1

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	1
v_3	1	1	1

$C_{ij}^2 = C_{ij}^3 = c(v_i, v_j)$

Рис. 5.19. Значения C_{ij}^k

5.8. Алгоритм поиска кратчайшего пути

Для вычисления кратчайшего пути воспользуемся вторым замкнутым полукольцом из примера 5.9, а именно: множеством неотрицательных действительных чисел с $+\infty$. Напомним, что аддитивной операцией в этом полукольце является операция MIN, а мультипликативной — сложение действительных чисел. Иначе говоря, мы рассматриваем структуру (R, MIN, +, $+\infty$, 0). В примере 5.10 отмечалось, что $a^* = 0$ для всех $a \in R$, так что снова можно обойтись без операции * в строке 5 алгоритма 5.5, заменив эту строку на

$$C_{ij}^k \leftarrow \mathtt{MIN}\left(C_{ij}^{k-1}, C_{ik}^{k-1} + C_{kj}^{k-1}\right). \tag{5.5}$$

Неформально оператор (5.5) означает, что кратчайшим путем из v_i в v_j, не проходящим через вершины, индексы которых больше k, является более короткий из следующих двух путей:

1) кратчайшего пути, не проходящего через вершины, индексы которых больше $k - 1$, и
2) кратчайшего пути из v_i в v_k и затем в v_j, не проходящего через другие вершины, индексы которых больше $k - 1$.

Чтобы превратить алгоритм 5.5 в алгоритм поиска кратчайшего пути, введем функцию $l(v_i, v_j)$, равную стоимости ребра (v_i, v_j), если оно есть, и $+\infty$ в противном случае. Затем заменим строку 5 оператором (5.5). По теореме 5.5 значение $c(v_i, v_j)$,

выданное алгоритмом 5.5, будет наименьшей стоимостью (т.е. суммой стоимостей ребер) пути из всех путей между v_i и v_j.

Пример 5.13. Рассмотрим граф, представленный на рис. 5.18. Пусть каждое ребро помечено так, как показано на рисунке. Тогда функции l, C_{ij}^0, C_{ij}^1, C_{ij}^2 и C_{ij}^3 примут значения, показанные на рис. 5.20. Например, $C_{12}^3 = \mathtt{MIN}\left(C_{12}^2,\ C_{13}^2 + C_{32}^2\right) = \mathtt{MIN}(8,\ 5+2) = 7$. □

	v_1	v_2	v_3
v_1	2	8	5
v_2	3	∞	∞
v_3	∞	2	∞

$l(v_i, v_j)$

	v_1	v_2	v_3
v_1	0	8	5
v_2	3	0	∞
v_3	∞	2	0

C_{ij}^0

	v_1	v_2	v_3
v_1	0	8	5
v_2	3	0	8
v_3	∞	2	0

C_{ij}^1

	v_1	v_2	v_3
v_1	0	8	5
v_2	3	0	8
v_3	5	2	0

C_{ij}^2

	v_1	v_2	v_3
v_1	0	7	5
v_2	3	0	8
v_3	5	2	0

$C_{ij}^3 = c\left(v_i,\ v_j\right)$

Рис. 5.20. Вычисление кратчайшего пути

5.9. Задачи о путях и умножение матриц

Пусть $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо. Определим матрицы размерностью $n \times n$, содержащие элементы из S с обычными операциями суммирования и умножения. Иначе говоря, пусть матрицы A, B и C состоят из элементов a_{ij}, b_{ij} и c_{ij} соответственно для $1 < i, j < n$. Тогда $C = A + B$ означает, что $c_{ij} = a_{ij} + b_{ij}$ для всех i и j, а $C = A \cdot B$ означает, что $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$ для всех i и j. Нетрудно доказать следующую лемму.

Лемма 5.10. Пусть $(S, +, \cdot, 0, 1)$ — замкнутое полукольцо и M_n — множество матриц $n \times n$ над S. Пусть $+_n$ и $\cdot_n$ обозначают соответственно сложение и умножение матриц; 0_n — матрица, все элементы которой равны 0; I_n — матрица с единицей на главной диагонали и нулями вне диагонали. Тогда $(M_n, +_n, \cdot_n, 0_n I_n)$ — замкнутое полукольцо.

Доказательство. Оставляем читателям в качестве упражнения. □

Пусть $G = (V, E)$ — ориентированный граф, где $V = \{v_1, v_2, ..., v_n\}$. Допустим, l: $(V \times V) \to S$ — функция разметки, как в алгоритме 5.5, и A_G — матрица $n \times n$, элемент ij которой равен $l(v_i, v_j)$. Следующая лемма связывает вычисление, выполняемое алгоритмом 5.5, с вычислением замыкания A_G^* в полукольце M_n матриц $n \times n$ над S.

Лемма 5.11. Пусть G и A_G такие, как указано выше. Тогда (i, j)-й элемент матрицы A_G^* равен $c(v_i, v_j)$.

Доказательство. $A_G^* = \sum_{i=0}^{\infty} A_G^i$, где $A_G^0 = I_n$ и $A_G^i = A_G \cdot A_G^{i-1}$ для $i \geq 1$. Индукцией по i легко показать, что ij-й элемент матрицы A_G^k равен сумме стоимостей путей длины k, идущих из v_i в v_j. Следовательно, ij-й элемент матрицы A_G^* равен сумме стоимостей всех путей из v_i в v_j. □

Из леммы 5.11 вытекает, что алгоритм 5.5 можно рассматривать как алгоритм вычисления замыкания матрицы. Теорема 5.5 утверждает, что алгоритм 5.5 требует $O(n^3)$ элементарных операций над замкнутым полукольцом. Мы покажем, что если скалярные величины принадлежат замкнутому полукольцу, то произведение двух матриц с вычислительной точки зрения эквивалентно вычислению замыкания матрицы. Иначе говоря, имея любой алгоритм для вычисления произведения, мы можем построить алгоритм замыкания, и наоборот. Этот результат глубже рассмотрен в главе 6, где показано, что, если скаляры образуют кольцо, умножение матриц можно выполнить, используя менее чем $O(n^3)$ операций. Эта конструкция состоит из двух частей.

Теорема 5.6. Если замыкание произвольной матрицы размерностью $n \times n$ над замкнутым полукольцом можно вычислить за такое время $T(n)$, что $T(3n) \leq 27T(n)$,[6] то существует константа c, такая, что время $M(n)$ умно-

[6]Это правдоподобное предположение, поскольку $T(n)$ в худшем случае имеет порядок $O(n^3)$. На самом деле в утверждении теоремы вместо числа 27 можно было бы использовать любую другую константу.

жения двух матриц A и B размерностью $n \times n$ удовлетворяет неравенству $M(n) \le cT(n)$.

Доказательство. Допустим, мы хотим вычислить произведение матриц $A \cdot B$, где A и B — матрицы размерностью $n \times n$. Сначала сформируем матрицу размерностью $3n \times 3n$:

$$X = \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}$$

и найдем замыкание матрицы X. Отметим, что

$$X^2 = \begin{pmatrix} 0 & 0 & A \cdot B \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ и } X^3 = X^4 = \ldots = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Следовательно

$$X^* = I_{3n} + X + X^2 = \begin{pmatrix} I_n & A & A \cdot B \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

Поскольку произведение $A \cdot B$ записано в правом верхнем углу, приходим к выводу, что $M(n) \le T(3n) \le 27T(n)$. □

Это доказательство теоремы 5.6 имеет следующую интерпретацию в виде графа. Рассмотрим граф G с $3n$ вершинами $\{1, 2, \ldots, 3n\}$. Разобьем их на три множества: $V_1 = \{1, 2, \ldots, n\}$, $V_2 = \{n + 1, n + 2, \ldots, 2n\}$ и $V_3 = \{2n + 1, 2n + 2, \ldots, 3n\}$. Будем считать, что все ребра графа G направлены либо из V_1 в V_2, либо из V_2 в V_3. Такой граф изображен на рис. 5.21. Пусть A и B — такие матрицы $n \times n$, что ij-й элемент матрицы A является меткой ребра, идущего из i в $n + j$, а ij-й элемент матрицы B является меткой ребра, идущего из $n + i$ в $2n + j$. Тогда ij-й элемент произведения $A \cdot B$ представляет собой сумму меток путей, идущих из i в $2n + j$, поскольку каждый такой путь образован ребром из i в некоторую вершину k, $n < k \le 2n$, и следующим за ним ребром из k в $2n + j$. Следовательно, сумма меток всех путей из i в $2n + j$ совпадает с ij-м элементом матрицы $A \cdot B$.

Теперь докажем обратное утверждение, согласно которому по заданному алгоритму умножения матриц можно найти алгоритм замыкания, время работы которого с точностью до постоянного множителя совпадает с временем работы заданного алгоритма.

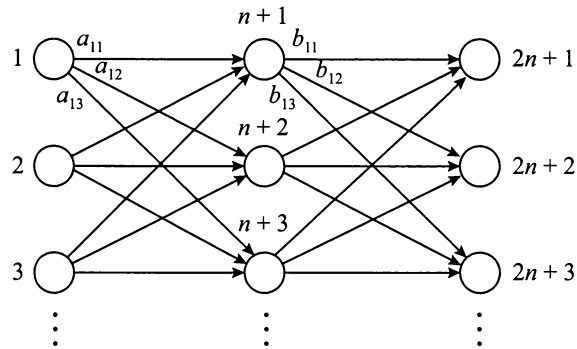


Рис. 5.21. Интерпретация теоремы 5.6 в виде графа

Теорема 5.7. Если произведение любых двух матриц $n \times n$ над замкнутым полукольцом можно вычислить за такое время $M(n)$, что $M(2n) \geq 4M(n)$, то существует алгоритм, вычисляющий замыкание матрицы за время $T(n)$, удовлетворяющее неравенству $T(n) \leq cM(n)$ для некоторой константы c.

Доказательство. Пусть X — матрица размерностью $n \times n$. Сначала рассмотрим случай, когда n — степень числа 2, т.е. 2^k. Разобьем матрицу X на четыре матрицы размера $2^{k-1} \times 2^{k-1}$:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

Из леммы 5.11 следует, что матрица X представляет граф $G = (V, E)$, в котором множество вершин разбито на два множества V_1 и V_2 размера 2^{k-1}. Матрица A представляет ребра между вершинами множества V_1, матрица D — ребра между вершинами из V_2, матрица B — ребра, идущие из вершин множества V_1 в вершины множества V_2, а матрица C — ребра, идущие из вершин множества V_2 в вершины множества V_1 (рис. 5.22).

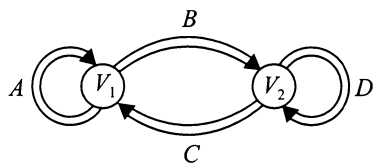


Рис. 5.22. Граф G

Путь из v_1 в v_2, концы которого принадлежат V_1,

1) либо никогда не выходит за пределы V_1,
2) либо для некоторого $k \geq 1$ найдутся такие вершины w_i, x_i, y_i и z_i, где $1 \leq i \leq k$, что все вершины w_i и z_i лежат в V_1, все x_i и y_i — в V_2, и рассматриваемый путь идет из v_1 в w_1 внутри V_1, затем по некоторому ребру уходит в x_1, потом вдоль некоторого пути внутри V_2 идет в вершину y_1. после чего по некоторому ребру уходит в z_1, затем по пути внутри V_1 идет в w_2 и т.д. и, наконец, приходит в z_k, откуда вдоль пути внутри V_1 идет в v_2. Этот вид пути показан на рис. 5.23.

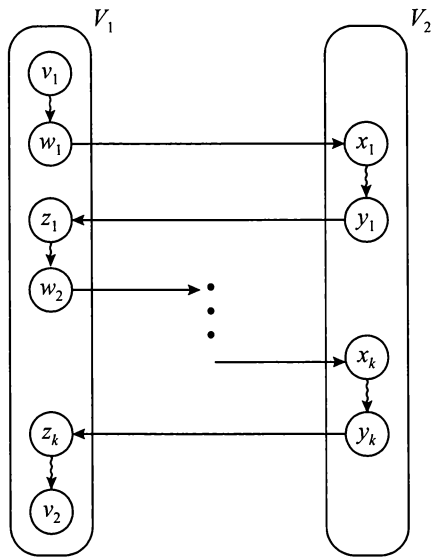


Рис. 5.23. Пути, удовлетворяющие условию 2

Сумма меток, приписанных путям, удовлетворяющим условию 1 или 2, очевидно, равна $(A + B \cdot D^* \cdot C)^*$. Иначе говоря, $B \cdot D^* \cdot C$ представляет пути, идущие из w_i в x_i, затем в y_i и далее в z_i на рис. 5.23. Следовательно, матрица $A + B \cdot D^* \cdot C$ представляет те пути, соединяющие вершины в V_1, которые либо состоят из одного ребра, либо прыгают прямо в V_2, остаются там на некоторое время и затем прыгают обратно в V_1. Все пути между вершинами из множества V_1 можно представить в виде последовательности путей, представленных матрицей $A + B \cdot D^* \cdot C$. Таким образом, матрица $(A + B \cdot D^* \cdot C)^*$ представляет все пути, соединяющие вершины из V_1. Следовательно, верхний левый угол матрицы X^* занимает матрица $(A + B \cdot D^* \cdot C)^*$.

Обозначим матрицу $(A + B \cdot D^* \cdot C)^*$ через E. С помощью аналогичных рассуждений можно представить матрицу X^* в виде

$$X^* = \begin{pmatrix} E & E \cdot B \cdot D^* \\ D^* \cdot C \cdot E & D^* + D^* \cdot C \cdot E \cdot B \cdot D^* \end{pmatrix} = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Матрицы E, F, G и H можно вычислить, проделав такую последовательность шагов:

$$\begin{aligned} T_1 &= D^*, \\ T_2 &= B \cdot T_1, \\ E &= (A + T_2 \cdot C)^*, \\ F &= E \cdot T_2, \\ T_3 &= T_1 \cdot C, \\ G &= T_3 \cdot E, \\ H &= T_1 + G \cdot T_2. \end{aligned}$$

Эти шаги требуют выполнения двух замыканий, шести умножений и двух сложений матриц размерностью $2^{k-1} \times 2^{k-1}$. Следовательно, можно записать:

$$\begin{aligned} T(1) &= 1, \\ T(2^k) &\le 2T(2^{k-1}) + 6M(2^{k-1}) + 2 \cdot 2^{2k-2}, \; k > 1. \end{aligned} \tag{5.6}$$

Три слагаемых в правой части неравенства (5.6) представляют сложность соответственно замыканий, умножений и сложений. Мы утверждаем, что существуют такие константа c и функция T, что $T(2^k) \le cM(2^k)$ и T удовлетворяет (5.6) для всех k. База индукции, т.е. вариант $k = 0$, тривиальна, поскольку константу c можно взять сколь угодно большой. Для выполнения шага индукции предположим, что $T(2^{k-1}) \le cM(2^{k-1})$ для некоторой константы c. Из условия теоремы, т.е. из неравенства $M(2n) \ge 4M(n)$, заключаем, что $M(2^{k-1}) \ge 2^{2k-2}$. (Заметим, что $M(1) = 1$.) Следовательно, из (5.6) вытекает, что

$$T(2^k) \le (2c+8)M(2^{k-1}).$$

Снова в силу условия теоремы $M(2^{k-1}) \le \frac{1}{4} M(2^k)$, так что

$$T(2^k) \le \left(\frac{1}{2}c + 2\right) M(2^k). \tag{5.7}$$

Если взять $c \ge 4$, то (5.7) влечет за собой $T(2^k) \le cM(2^k)$, а это и требовалось доказать.

Если n не является степенью числа 2, то можно вложить X в большую матрицу, размер которой является степенью числа 2:

$$\begin{pmatrix} X & 0 \\ 0 & I \end{pmatrix},$$

где I — единичная матрица минимально возможного размера. Это вложение увеличит размер матрицы не более чем вдвое, и, значит, константа увеличится не более чем в восемь раз. Таким образом, найдется такая константа c', что $T(n) \leq c'M(n)$ для всех n, независимо от того, являются ли они степенями числа 2 или нет. □

Следствие 1. Время, необходимое для вычисления замыкания булевой матрицы, имеет тот же порядок, что и время вычисления произведения двух булевых матриц того же размера.

В главе 6 мы опишем асимптотически более быстрые алгоритмы умножения булевых матриц и тем самым покажем, что транзитивное замыкание графа можно вычислить за время, меньшее $O(n^3)$.

Следствие 2. Время, необходимое для вычисления замыкания матрицы над множеством неотрицательных целых чисел с операциями `MIN` и + в качестве сложения и умножения элементов, имеет тот же порядок, что и время умножения двух матриц такого же типа.

В настоящее время[7] все известные методы решения задачи о нахождении кратчайших путей для всех пар вершин тратят не меньше cn^3 времени для некоторой константы c.

5.10. Задачи с одним источником

Во многих приложениях требуется найти кратчайший путь только из одной вершины (*источника*). Например, иногда необходимо найти кратчайший путь между двумя конкретными вершинами, однако до сих пор не изобретен алгоритм, который решал бы эту задачу эффективнее в худшем случае, чем лучший из известных алгоритмов для одного источника.

В задачах с одним источником нет объединяющего понятия, аналогичного замкнутым полукольцам и алгоритму 5.5. Если хотим всего лишь выяснить, к каким вершинам ведут пути из источника, то задача становится тривиальной и ее можно решить с помощью нескольких алгоритмов, имеющих временную сложность $O(e)$ на графах, содержащих e ребер. Поскольку любой алгоритм, который не просма-

[7] Первое издание книги вышло в 1974 г. — *Примеч. ред.*

тривает все ребра, не может быть корректным, нетрудно убедиться, что $O(e)$ — лучшая оценка, на которую можно надеяться.

В задаче нахождения кратчайших путей из одного источника ситуация иная. Хотя нет причин предполагать, что сложность ее решения имеет порядок, превышающий $O(e)$, ни один такой алгоритм не известен. Мы обсудим алгоритм сложности $O(n^2)$, работа которого основана на построении множества S вершин, кратчайшие расстояния до которых от источника известны. На каждом шаге в S добавляется та из оставшихся вершин v, расстояние до которой от источника меньше расстояния до всех других оставшихся вершин. Если стоимости всех ребер неотрицательны, то можно быть уверенным, что путь из источника в v проходит только через вершины из S. Следовательно, для каждой вершины v достаточно найти кратчайшее расстояние до нее от источника вдоль пути, проходящего только через вершины из S. Изложим алгоритм формально.

Алгоритм 5.6. Кратчайший путь из одного источника (алгоритм Дейкстры)

Вход. Ориентированный граф $G = (V, E)$, источник $v_0 \in V$ и функция l, отображающая множество $E \times E$ во множество неотрицательных действительных чисел. Полагаем $l(v_i, v_j) = +\infty$, если ребро (v_i, v_j), $i \neq j$ не принадлежит графу, и $l(v, v) = 0$.

Выход. Для каждой вершины $v \in V$ минимальная сумма меток на ребрах среди всех путей P, идущих из v_0 в v.

Метод. Создаем такое множество $S \subseteq V$, что кратчайший путь из источника в каждую вершину v из S целиком лежит в S. Массив $D[v]$ содержит стоимость текущего кратчайшего пути из v_0 в v, который проходит только через вершины из S. Алгоритм приведен на рис. 5.24. □

```
    begin
1.      S ← {v0};
2.      D[v0] ← 0;
3.      for each v in V − {v0} do D[v] ← l(v0, v);
4.      while S ≠ V do
            begin
5.              выбрать вершину w в V − S при D[w] − минимум;
6.              добавить w в S;
7.              for для каждой вершины v из V − S do
8.                  D[v] ← MIN(D[v], D[w] + l(w, v))
            end
    end
```

Рис. 5.24. Алгоритм Дейкстры

Пример 5.14. Рассмотрим граф, изображенный на рис. 5.25. Вначале $S = \{v_0\}$, $D[v_0] = 0$, а $D[v_i]$ равно 2, $+\infty$, $+\infty$, 10 для $i = 1, 2, 3, 4$ соответственно. На первой итерации цикла в строках 4–8 выбирается вершина $w = v_1$, поскольку $D[v_1] = 2$ — наименьшее значение D. Затем вычисляем $D[v_2] = \texttt{MIN}(+\infty, 2 + 3) = 5$ и $D[v_4] = \texttt{MIN}(10, 2 + 7) = 9$. Последовательность значений D и другие вычисления алгоритма 5.6 приведены на рис. 5.26.□

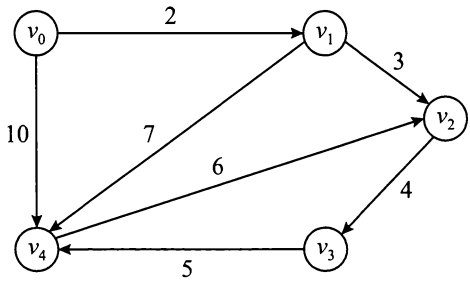


Рис. 5.25. Граф с помеченными ребрами

Итерация	S	w	$D[w]$	$D[v_1]$	$D[v_2]$	$D[v_3]$	$D[v_4]$
Начало	$\{v_0\}$	–	–	2	$+\infty$	$+\infty$	10
1	$\{v_0, v_1\}$	v_1	2	2	5	$+\infty$	9
2	$\{v_0, v_1, v_2\}$	v_2	5	2	5	9	9
3	$\{v_0, v_1, v_2, v_3\}$	v_3	9	2	5	9	9
4	Все	v_4	9	2	5	9	9

Рис. 5.26. Вычисления по алгоритму 5.6 на графе с рис. 5.25

Теорема 5.8. Алгоритм 5.6 вычисляет стоимость кратчайшего пути среди всех путей, идущих из v_0 в любую вершину, за время $O(n^2)$.

Доказательство. Цикл **for** в строках 7-8 требует выполнения $O(n)$ шагов, столько же шагов требует и выбор w в строке 5. В оценке сложности цикла **while** в строках 4–8 эти процессы преобладают над остальными. Цикл **while** выполняется $O(n)$ раз, так что его общая временная сложность имеет порядок $O(n^2)$. Строки 1–3, очевидно, выполняются за время $O(n)$.

Чтобы показать корректность, нужно доказать индукцией по размеру множества S, что для каждого v из S число $D[v]$ равно длине кратчайшего пути из v_0 в v. Более того, для всех v из $V - S$ число $D[v]$ равно длине кратчайшего пути из v_0 в v, лежащего целиком в S, за исключением самой вершины v.

База. Пусть $||S|| = 1$. Кратчайший путь из v_0 в себя имеет нулевую длину, а путь из v_0 в v, лежащий целиком в S, за исключением v, состоит из единственного ребра (v_0, v). Таким образом, строки 2 и 3 корректно формируют начальные значения массива D.

Шаг индукции. Пусть w — вершина, выбранная в строке 5. Если число $D[w]$ не равно длине кратчайшего пути из v_0 в w, то должен существовать более короткий путь P. Этот путь P должен содержать некоторую вершину, отличающуюся от w и не принадлежащую S. Пусть v — первая такая вершина на пути P. Однако тогда расстояние от v_0 до v меньше $D[w]$, а кратчайший путь в вершину v целиком лежит в S, за исключением самой вершины v (см. рис. 5.27). Следовательно, по предположению индукции при выбре вершины w выполняется неравенство $D[v] < D[w]$. Это — противоречие. Таким образом, такого пути P не существует и $D[w]$ — длина кратчайшего пути из v_0 в w.

Вторая часть индуктивного предположения о том, что значение $D[w]$ остается корректным, очевидно в силу строки 8. □

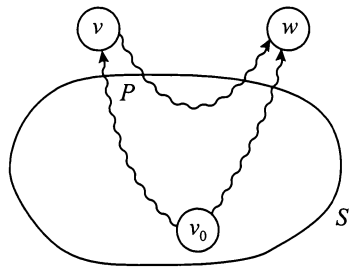


Рис. 5.27. Пути в вершину v

5.11. Доминаторы в ориентированных ациклических графах: сочетание понятий

В этой главе уже описано несколько методов разработки эффективных алгоритмов, например метод поиска в глубину и разумное упорядочение вычислений. В главе 4 мы изучили много структур данных, полезных для представления множеств, над которыми выполняются различные операции. Эту главу мы закончим примером, иллюстрирующим, как можно строить эффективные алгоритмы, комбинируя структуры данных из главы 4 с методами работы с графами, развитыми в данной главе. В частности, мы применим офлайн-алгоритм поиска минимума, алгоритм объединения непересекающихся множеств и 2-3-деревья с поиском в глубину для определения доминаторов в ориентированном ациклическом графе.

Ориентированный граф $G = (V, E)$ называется *корневым* относительно вершины r, если существуют пути из r в каждую его вершину. В остальной части раздела

будем предполагать, что $G = (V, E)$ — корневой ориентированный ациклический граф с корнем r.

Вершина v называется *доминатором* вершины w, если любой путь из корня в w проходит через v. Каждая вершина является доминатором самой себя, а корень доминирует над каждой вершиной. Множество доминаторов вершины w можно линейно упорядочить, расположив их в порядке следования на кратчайшем пути из корня в вершину w. Ближайший доминатор вершины w, не совпадающий с ней, называется ее *непосредственным доминатором*. Поскольку множество доминаторов каждой вершины линейно упорядочено, отношение доминирования можно представить деревом с корнем r, называемым *доминаторным деревом* для G. Вычисление доминаторов полезно в задаче оптимизации кодов (Aho and Ullman [1973], Hecht [1973]).

Мы разработаем алгоритм сложности $O(e \log e)$, вычисляющий доминаторное дерево для корневого ориентированного ациклического графа, содержащего e ребер. Основное предназначение этого алгоритма — продемонстрировать сочетание методов этой и предыдущей глав. Алгоритм основан на следующих трех леммах.

Пусть $G = (V, E)$ — граф и $G' = (V', E')$ — подграф G. Если $(a, b) \in E'$, будем писать $a \underset{G'}{\Rightarrow} b$. Через $\overset{+}{\underset{G'}{\Rightarrow}}$ и $\overset{*}{\underset{G'}{\Rightarrow}}$ обозначим соответственно транзитивное и рефлексивно-транзитивное замыкания отношения $\underset{G'}{\Rightarrow}$. Если $(a, b) \in E$, то будем писать $a \Rightarrow b$.

Лемма 5.12. Пусть $G = (V, E)$ — ориентированный ациклический граф с корнем r, а $S = (V, T)$ — его остовное дерево с корнем r, построенное методом поиска в глубину. Пусть a, b, c и d — такие вершины из ножества V, что $a \overset{*}{\underset{S}{\Rightarrow}} b \overset{+}{\underset{S}{\Rightarrow}} c \overset{*}{\underset{S}{\Rightarrow}} d$. Далее, пусть (a, c) и (b, d) — их прямые ребра. Тогда замена прямого ребра (b, d) прямым ребром (a, d) не меняет доминаторов ни одной вершины графа G (рис. 5.28).

Доказательство. Обозначим через G' граф, полученный при замене ребра (b, d) в графе G ребром (a, d). Предположим, что v — доминатор вершины w в G, но не в G'. Тогда в графе G' найдется путь P из r в w, не содержащий v. Очевидно, что этот путь должен идти по ребру (a, d), поскольку это единственное ребро в графе G', которого нет в графе G. Следовательно, можно написать $P: r \overset{*}{\Rightarrow} a \Rightarrow d \overset{*}{\Rightarrow} w$. Отсюда следует, что вершина v должна лежать на пути $a \overset{+}{\underset{S}{\Rightarrow}} d$ и что она не совпадает с a и d. Если $a < v \le b$ при нумерации поиска в глубину, то замена ребра $a \Rightarrow d$ в P на путь $a \Rightarrow c \overset{*}{\underset{S}{\Rightarrow}} d$ порождает путь в графе G, идущий из r в w и не содержащий v. Если $b < v < d$, то замена ребра $a \Rightarrow d$ в P на путь $a \overset{*}{\underset{S}{\Rightarrow}} b \Rightarrow d$ порождает путь в графе G, идущий из r в w и не содержащий v. Поскольку $a < v \le b$ или $b < v < d$, то найдется путь в графе G, идущий из r в w и не содержащий v. Это — противоречие.

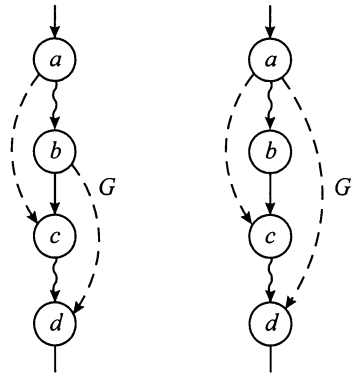


Рис. 5.28. Преобразование из леммы 5.12

Допустим, что v — доминатор вершины w в графе G', но не в графе G. Тогда в G существует путь P из r в w, не содержащий v. Поскольку этот путь не лежит целиком в G', то он должен содержать ребро $b \Rightarrow d$. Следовательно, вершина v лежит на пути $b \overset{*}{\underset{S}{\Rightarrow}} d$ и $b < v < d$. Таким образом, в графе G' существует путь $r \overset{*}{\underset{S}{\Rightarrow}} a \Rightarrow d \overset{*}{\underset{S}{\Rightarrow}} v$, не содержащий вершину v. Это — противоречие. □

Лемма 5.13. Пусть G и S те же, что и в лемме 5.12, $a \overset{*}{\underset{S}{\Rightarrow}} b$ и (a, b) — прямое ребро графа G. Если в G нет прямого ребра (c, d), для которого $c < a$ и $a < d \le b$, и поперечного ребра (e, d), для которого $a < d \le b$, то удаление древесного ребра, входящего в b, не меняет доминаторов никакой вершины в G (рис. 5.29).

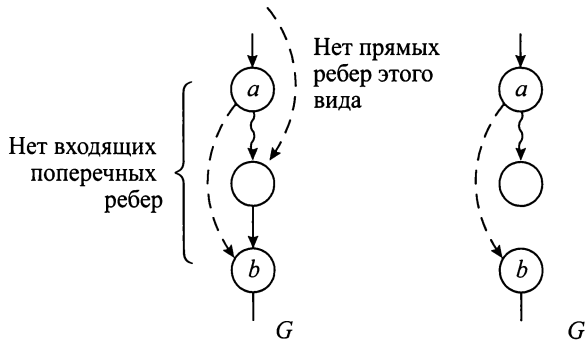


Рис. 5.29. Преобразование из леммы 5.13

Доказательство. Аналогично доказательству леммы 5.12. □

Лемма 5.14. Пусть G и S удовлетворяют условиям леммы 5.12, (c, d) — поперечное ребро графа G; b — общий предок для c и d с наибольшим номером. Пусть $a = \mathtt{MIN}(\{b\} \cup \{v | (v, w)$ — прямое ребро и $b < w \le c\})$. Если ни в одну вершину пути $b \overset{*}{\underset{S}{\Rightarrow}} c$ отличающуюся от b, не входит поперечное ребро, то замена поперечного ребра (c, d) прямым ребром (a, d) не меняет доминаторов никакой вершины в графе G (рис. 5.30).

Доказательство. Пусть G' — граф, полученный при замене ребра (c, d) в графе G ребром (a, d). Допустим, что v — доминатор вершины w в графе G, но не в графе G'. Тогда в графе G' существует путь P из r в w, не содержащий v. Очевидно, что этот путь должен идти по ребру (a, d). Следовательно, вершина v должна лежать на пути $a \overset{*}{\underset{S}{\Rightarrow}} b$, проходящем по остовному дереву, поскольку в противном случае замена ребра $a \Rightarrow d$ в P на $a \overset{*}{\underset{S}{\Rightarrow}} d$ или $a \overset{*}{\underset{S}{\Rightarrow}} c \Rightarrow d$ породила бы путь в графе G, идущий из r в w и не содержащий v. Однако тогда замена ребра $a \Rightarrow d$ в пути P на $a \Rightarrow u \overset{*}{\underset{S}{\Rightarrow}} c \Rightarrow d$, где u лежит на пути $b \overset{*}{\underset{S}{\Rightarrow}} c$ и $u > b$, породила бы путь в графе G, идущий из r в w и не содержащий v. Это — противоречие.

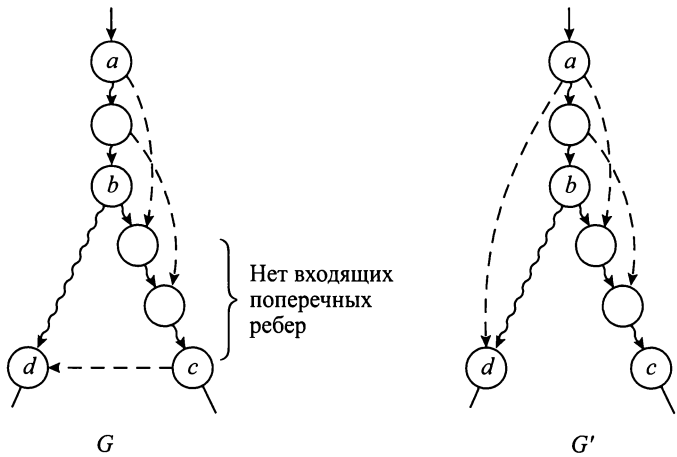


Рис. 5.30. Преобразование из леммы 5.14

Допустим, что v — доминатор вершины w в графе G', но не в G. Тогда в графе G найдется путь P из r в w, не содержащий v. Понятно, что путь P содержит ребро (c, d). Запишем путь P в виде $r \overset{*}{\underset{S}{\Rightarrow}} c \Rightarrow d \overset{*}{\underset{S}{\Rightarrow}} w$. Путь $r \overset{*}{\underset{S}{\Rightarrow}} c$ должен содержать вершину, лежащую на пути $a \overset{*}{\underset{S}{\Rightarrow}} b$, поскольку не существует поперечных ребер, входящих в вершины на пути $b \overset{*}{\underset{S}{\Rightarrow}} c$ (исключая вершину b). Пусть x — такая вер-

шина с наибольшим номером, а P_1 — участок пути P от r до x, за которым следует $x \overset{*}{\underset{S}{\Rightarrow}} d$, а за ним — участок пути P от d до w. Пусть P_2 — путь $r \overset{*}{\underset{S}{\Rightarrow}} a \Rightarrow d$, за которым следует участок пути P от d до w. Если вершина v лежит на пути P_1, то он должен лежать и на пути $x \overset{*}{\underset{S}{\Rightarrow}} d$, причем $v > x$. Если он лежит на пути P_2, то должен лежать на пути $r \overset{*}{\underset{S}{\Rightarrow}} a$. Поскольку $a \le x$, то один из этих путей в графе G' не содержит v. Это — противоречие. □

Нетрудно показать, что повторное применение преобразований из лемм 5.12–5.14 до тех пор, пока они возможны, превращает граф G в дерево. Поскольку эти преобразования не меняют отношения доминирования, то окончательное дерево должно быть доминаторным для графа G. Эти преобразования образуют алгоритм вычисления доминаторов графа G. Вся хитрость заключается в построении структуры данных, позволяющей эффективно находить подходящие ребра, к которым надо применять преобразования из лемм 5.12–5.14.

На интуитивном уровне, мы строим доминаторное дерево для заданного ориентированного ациклического графа $G = (V, E)$ следующим образом. Сначала методом поиска в глубину строим в графе G, начиная с корня, соответствующее остовное дерево $S = (V, T)$. Номера вершин графа G меняем на их номера, порожденные поиском в глубину. Затем применяем к графу G преобразования из лемм 5.12–5.14. Они выполняются двумя взаимосвязанными подпрограммами, одна из которых обрабатывает прямые ребра, а другая — поперечные.

I. Прямые ребра

Предположим на время, что в графе G нет поперечных ребер. Если в вершину v входит более одного прямого ребра, то по лемме 5.12 все прямые ребра, входящие в вершину v, за исключением одного с наименьшим началом, можно удалить, не изменив доминаторов ни одной вершины.

Составным ребром назовем упорядоченную пару (t, H), где t — вершина, называемая *началом* составного ребра, а H — множество вершин, называемое *концом* составного ребра. Составное ребро $(t, \{h_1, h_2, ..., h_k\})$ представляет множество ребер $(t, h_1), (t, h_2), ..., (t, h_k)$

Многократно применяя лемму 5.12, изменяем начала прямых ребер. В какой-то момент начало каждого ребра из множества ребер с общим началом t изменится на t'. Чтобы сделать это эффективно, представим одним составным ребром некоторые множества прямых ребер с общим началом. Вначале каждое прямое ребро (t, h) представляется составным ребром $(t, \{h\})$.

Каждой вершине v графа G поставим в соответствие множество $F[v]$. Это множество содержит такие пары $(t, \{h_1, h_2, ..., h_m\})$, что t — предок вершины v, каждая вершина h_i — потомок вершины v и (t, h_i) — прямое ребро в графе G. Вначале $F[v] = \{(t, \{v\})\}$, где t — начало с наименьшим номером прямого ребра с концом v.

Затем проходим остовное дерево в порядке, обратном к прямому. Возвращаясь по ребру (v, w) остовного дерева, обнаруживаем, что множество $F[w]$ содержит составное ребро для каждого прямого предка t вершины w, который в данный момент является началом прямого ребра для некоторого потомка вершины w. Далее выполняем следующие действия.

1. Пусть $(t, \{h_1, h_2, ..., h_m\})$ — составное ребро в $F[w]$, имеющее начало с наибольшим номером. Если $t = v$, удаляем его из $F[w]$. (Это составное ребро представляет множество прямых ребер, начала которых были перемещены в вершину v, но больше не будут перемещаться под действием преобразования из леммы 5.12.) Удалим из графа G ребро остовного дерева с концом h_i, $1 \le i \le m$. (Этот шаг соответствует применению леммы 5.13.)
2. Если в графе G есть прямое ребро (t, v),[8] то для каждого составного ребра $(u, \{h_1, ..., h_m\})$ из $F[w]$, для которого $u \ge t$, делаем следующее:
   а) удаляем $(u, \{h_1, ..., h_m\})$ из $F[w]$;
   б) объединяем $(u, \{h_1, ..., h_m\})$ с концом того составного ребра из $F[v]$, которое представляет среди прочих и ребро (t, v).

   (Этот шаг соответствует применению леммы 5.12.)
3. Заменяем $F[v]$ на $F[v] \cup F[w]$.

Пример 5.15. Рассмотрим корневой ориентированный ациклический граф G, изображенный на рис. 5.31. Поиск в глубину на графе G может породить граф, изображенный на рис. 5.32, *а*, где указаны также F-множества, поставленные в соответствие каждой вершине. На рис. 5.32, *б–г*, приведены результаты преобразований прямых ребер. Окончательное доминаторное дерево изображено на рис. 5.32, *г*. □

Предоставляем читателю доказать, что по достижении корня итоговый граф действительно является корректным доминаторным деревом (в предположении, что не существует поперечных ребер). Этот алгоритм можно эффективно реализовать с помощью алгоритма объединения непересекающихся множеств и обрабатывать им множества концов составных ребер. Множества $F[v]$ составных ребер можно представить 2-3-деревьями, поскольку необходимо уметь эффективно удалять составные ребра, выделять во множестве составных ребер ребро с наибольшим началом и строить объединение множеств составных ребер. При такой структуре данных для обработки графа, содержащего e ребер, требуется время $O(e \log e)$.

---

[8] Напомним, что по предположению все прямые ребра с концом v, кроме тех, у которых начало имеет наименьший номер, удалены из графа G.

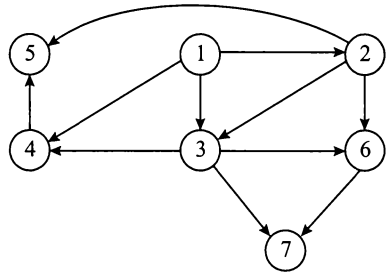


Рис. 5.31. Корневой ориентированный ациклический граф

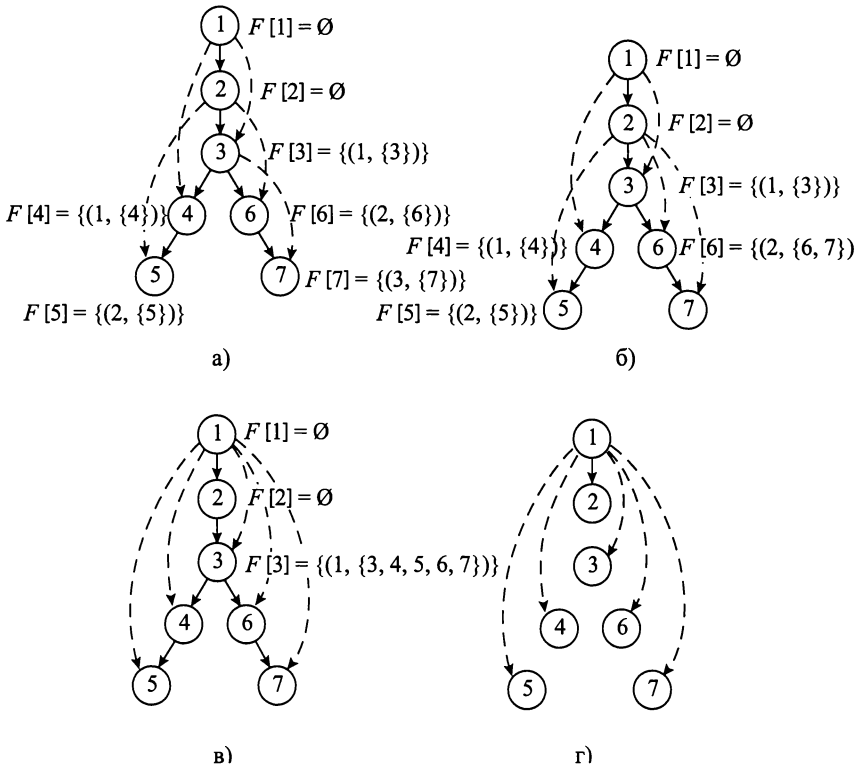


Рис. 5.32. Результаты преобразований прямых ребер: (*а*) начальное состояние; (*б*) после возвращения вдоль ребра (6, 7) остовного дерева; (*в*) после возвращения вдоль ребра (3, 4) остовного дерева; (*г*) после возвращения вдоль ребра (1, 2) остовного дерева

II. Поперечные ребра

В общем случае нет оснований предполагать, что поперечных ребер в графе нет, но их можно заменить прямыми ребрами с помощью метода, излагаемого ниже. Эту замену не следует выполнять до обработки прямых ребер, описанной в п 1, поскольку структура данных, которая строится в п. 1, помогает эффективно применить лемму 5.14. С другой стороны, не надо полностью выполнять алгоритм из п. 1 до удаления поперечных ребер, поскольку каждое удаленное поперечное ребро становится прямым. Мы должны добавить этапы обработки поперечных ребер при обходе в порядке, обратном прямому, который был описан применительно к прямым ребрам. Заметим, что в силу леммы 5.13 в п. 1 требуется, чтобы в определенные моменты времени в определенные вершины не входили поперечные ребра. Поскольку обход выполняется в порядке, обратном прямому, шаги, описанные ниже, заранее преобразуют поперечное ребро в прямое, поскольку их наличие сделало лемму 5.13 неприменимой.

Пусть S — глубинное остовное дерево графа G. Вначале для каждого поперечного ребра (v, w) вычисляем общего предка вершин v и w с наибольшим номером. Каждой вершине v поставим в соответствие множество $L[v]$ упорядоченных пар (u, w), где (u, w), $u > w$, представляет запрос о предке вершин u и w с наибольшим номером. Вначале $L[v] = \{(v, w) \mid$ существует поперечное ребро (v, w), $v > w\}$. Во время обхода дерева в соответствии с процедурой, описанной в п. 1, выполним следующие дополнительные действия.

1. При прохождении древесного ребра (v, w), в котором $v < w$, удаляем из $L[v]$ все пары (x, y), в которых $y \geq w$. Вершина v — общий предок с наибольшим номером вершин x и y.
2. После возвращения к вершине v по ребру (v, w) остовного дерева заменяем $L[v]$ на $L[v] \cup L[w]$.

Вычисление предков с наибольшим номером можно выполнить не более чем за $O(eG(e))$ шагов, где e — число ребер графа G. Для этого можно воспользоваться обобщением офлайн-алгоритма `MIN` из упражнения 4.21.

Поперечные ребра преобразуются в прямые по лемме 5.14. Этот процесс необходимо выполнять во время обработки прямых ребер. Каждой вершине v поставим в соответствие некоторое множество $C[v]$ составных ребер. Вначале $C[v] = \{(v, \{h_1, ..., h_m\}) | (v, h_i)$ — поперечное ребро при $1 \leq i \leq m\}$. После возвращения в вершину v вдоль древесного ребра (v, w) в дополнение к обработке прямых ребер выполним следующие шаги.

1. Заменяем $C[v]$ на $C[v] \cup C[w]$.
2. Удаляем из составных ребер все поперечные ребра (x, y), для которых v — предок вершин x и y с наибольшим номером. Если t — начало этого состав-

ного ребра, заменяем поперечное ребро (x, y) прямым ребром (t, y). Если в вершину y уже входит прямое ребро, оставляем только то прямое ребро, у которого начало имеет меньший номер.

3. Пусть (u, v) — прямое ребро, если оно есть, с концом v. Если такого ребра нет, то пусть (u, v) — древесное ребро в ходящее в вершину v. После поиска всех потомков вершины v удаляем из $C[v]$ все составные ребра, начала которых лежат выше u. Объединяем множества концов удаленных составнх ребер и образуем новое составное ребро с началом u. Добавляем новое составное ребро во множество $C[v]$.

Пример 5.16. Рассмотрим глубинное остовное дерево, представленное на рис. 5.33, *а*. Значения $C[v]$ показаны для выбранных вершин. Поскольку существует прямое ребро из вершины 2 в вершину 8, изменим составное ребро $(8, \{4\})$ в $C[8]$ на $(2, \{4\})$. Затем объединим $C[8]$ и $C[7]$. Поскольку $(1, 7)$ — прямое ребро, преобразуем составное ребро $(2, \{4\})$ в $(1, \{4\})$. После возвращения в вершину 6 множество $C[6]$ станет равным $\{(1, \{4\}), (6, \{5\})\}$.

После возвращения из вершины 6 в вершину 3 множество $C[3]$ станет равным $\{(3, \{5\}), (1, \{4\})\}$. Вершина 3 является предком вершин 6 и 5, а также вершин 8 и 4, с наибольшим номером. Поэтому удаляем из $C[3]$ составные ребра $(3, \{5\})$ и $(1, \{4\})$ и добавляем в граф G прямые ребра $(3, 5)$ и $(1, 4)$. Результат показан на рис. 5.33, *б*. Последующая часть поиска не порождает никаких изменений. □

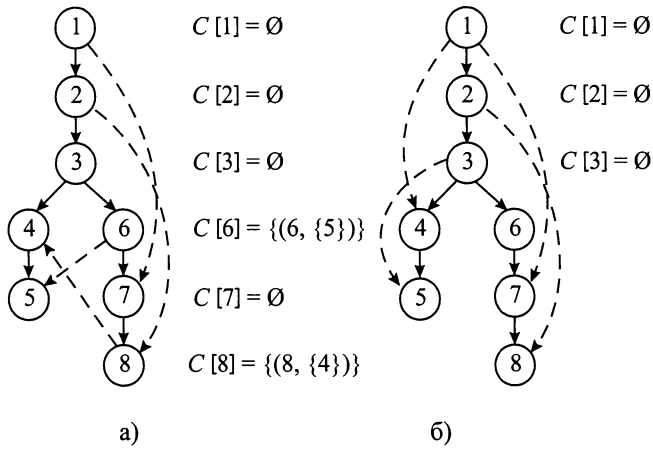


Рис. 5.33. Удаление поперечных ребер: (*а*) начальное состояние; (*б*) после обработки ребра (3, 6)

Составные поперечные ребра можно представить 2-3-деревьями. Попробуйте самостоятельно составить формальное описание доминаторного алгоритма. Если

сможете получить подходящие структуры, то считайте, что в полной мере освоили методики, рассмотренные в этой и предыдущей главах.

Упражнения

5.1. Найдите остовное дерево минимальной стоимости для графа, представленного на рис. 5.34, если все нарисованные ребра являются неориентированными.

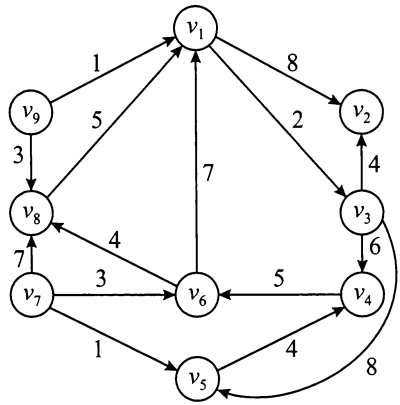


Рис. 5.34.

5.2. Пусть $S = (V, T)$ — остовное дерево минимальной стоимости, построенное с помощью алгоритма 5.1. Пусть $c_1 \le c_2 \le ... \le c_n$ — стоимости ребер в T. Пусть S' — произвольное остовное дерево со стоимостями ребер $d_1 \le d_2 \le ... \le d_n$. Покажите, что $c_i \le d_i$ для $1 \le i \le n$.

****5.3.** Чтобы за время $O(e)$ построить остовное дерево минимальной стоимости для графа с n вершинами и e ребрами в предположении, что $e \le f(n)$, где f — некоторая функция, которую необходимо найти, можно воспользоваться следующей стратегией. В различные моменты времени вершины будут группироваться во множества, связанные древесными ребрами, которые к тому времени уже найдены. Все ребра, инцидентные одной или двум вершинам из множества, будут храниться в приоритетной очереди для этого множества. Вначале каждая вершина находится во множестве, состоящем из нее самой. Шаг итерации состоит в том, чтобы найти наименьшее множество S и ребро с наименьшей стоимостью среди всех ребер, выходящих из S. Пусть это ребро ведет во множество T. Затем добавляем его к остовному дереву и объединяем множества S и T. Если все множества имеют размер не меньше $g(n)$, где g — еще одна функция, которую необходимо найти, то вместо этого образуем новый граф, где каждое множество представлено

одной вершиной. Вершины нового графа, соответствующие множествам S_1 и S_2, полагаются смежными, если некоторая вершина в S_1 была смежной некоторой вершине в S_2 в исходном графе. Стоимостью ребра, соединяющего S_1 и S_2 в новом графе, считается наименьшая из стоимостей ребер, соединяющих вершину из S_1 с вершиной из S_2 в исходном графе. Далее этот алгоритм рекурсивно применяется к новому графу.

Ваша задача — выбрать $g(n)$ так, чтобы минимизировать $f(n)$.

5.4. Постройте методом поиска в глубину остовный лес для неориентированного графа на рис. 5.35. Выбирайте в качестве исходной вершины для построения очередного дерева любую вершину. Найдите двусвязные компоненты этого графа.

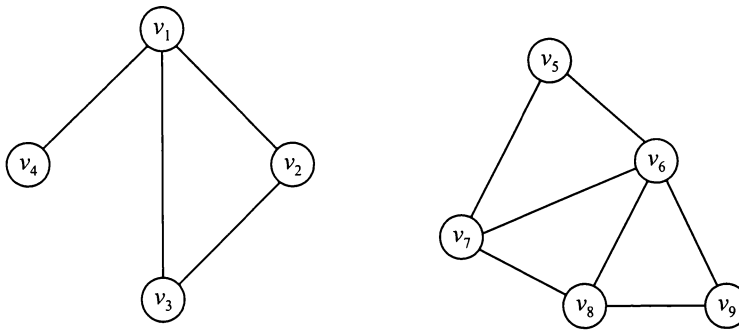


Рис. 5.35

5.5. Постройте методом поиска в глубину остовный лес для ориентированного графа, представленного на рис. 5.34. Затем найдите сильно связные компоненты.

5.6. Найдите двусвязные компоненты графа, представленного на рис. 5.36.

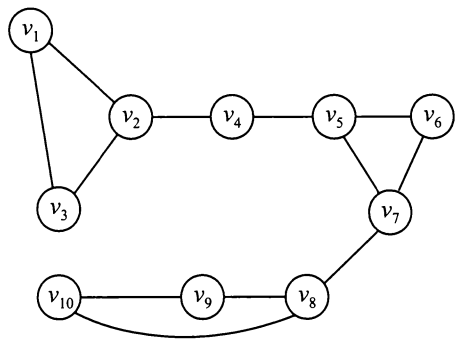


Рис. 5.36

5.7. С помощью метода поиска в глубину разработайте эффективные алгоритмы, которые решают следующие задачи.

а) Разбивают неориентированный граф на его связные компоненты.

б) Находят в неориентированном связном графе путь, проходящий через каждое ребро точно один раз в каждом направлении.

в) Проверяют, является ли ориентированный граф ацикличным.

г) Находят такой порядок вершин ациклического ориентированного графа, при котором $v < w$, если из v в w ведет путь ненулевой длины.

д) Выясняют, можно ли так ориентировать ребра связного неориентированного графа, чтобы получить сильно связный ориентированный граф. (*Подсказка*: покажите, что это можно сделать тогда и только тогда, когда удаление из графа G любого ребра оставляет его связным).

5.8. Пусть $G = (V, E)$ — *мультиграф* (т.е. ориентированный граф, у которого между двумя вершинами может быть более одного ребра). Напишите алгоритм сложности $O(\|E\|)$, который удаляет каждую вершину v степени 2 (заменяя ребра (u, v) и (v, w) ребром (u, w)) и уничтожает кратные ребра (заменяя их одним ребром). Заметим, что замена кратных ребер одним ребром может привести к образованию вершины степени 2, которую затем необходимо удалить. Аналогично удаление вершины степени 2 может привести к образованию кратных ребер, которые затем также необходимо удалить.

5.9. *Эйлеровым циклом* в неориентированном графе называется путь, который начинается и заканчивается в одной и той же вершине и проходит по каждому ребру точно один раз. Связный неориентированный граф G имеет эйлеров цикл тогда и только тогда, когда степень каждой вершины является четной. Постройте алгоритм сложности $O(e)$ для нахождения эйлерова цикла в графе с e ребрами при условии, что такой цикл существует.

***5.10.** Пусть $G = (V, E)$ — двусвязный неориентированный граф и (v, w) — его ребро. Пусть $G' = (\{v, w\}, \{(v, w)\})$. Укажите способ выполнения в онлайн-режиме последовательности операций `FINDPATH`(s, t), где s — вершина графа G'; (s, t) — ребро графа G, но не G'. Процедура `FINDPATH`(s, t) выполняется так: находим простой путь в графе G, начинающийся с ребра (s, t) и заканчивающийся в некоторой вершине графа G', отличающейся от s, и добавляем вершины и ребра этого пути к G'. Время выполнения любой последовательности должно составлять $O(\|E\|)$.

5.11. Для ориентированного графа G на рис. 5.37 найдите следующее.

а) Транзитивное замыкание.

б) Длину кратчайшего пути между каждой парой вершин (стоимости ребер приведены на рис. 5.37).

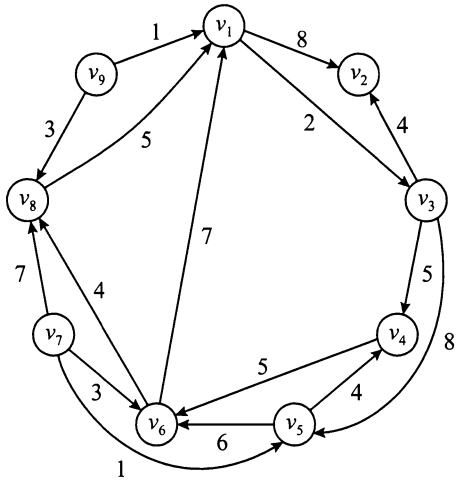


Рис. 5.37

***5.12.** Найдите замкнутое полукольцо из четырех элементов.

***5.13.** *Транзитивная редукция* ориентированного графа $G = (V, E)$ — это произвольный граф $G' = (V, E')$ с минимально возможным числом вершин, транзитивное замыкание которого совпадает с транзитивным замыканием графа G. Покажите, что если граф G ацикличен, то его транзитивная редукция является единственной.

****5.14.** Докажите, что время $R(n)$ вычисления транзитивной редукции ациклического графа с n вершинами имеет тот же порядок, что и время $T(n)$ вычисления транзитивного замыкания, если принять (вполне разумное) допущение, что $8R(n) \geq R(2n) \geq 4R(n)$ и $8T(n) \geq T(2n) \geq 4T(n)$.

***5.15.** Докажите, что время вычисления транзитивной редукции ациклического графа имеет тот же порядок, что и время вычисления транзитивной редукции произвольного графа, если принять те же допущения, что и в упражнении 5.14.

***5.16.** Повторите упражнении 5.15, но для транзитивного замыкания.

***5.17.** Пусть A — матрица размерностью $n \times n$ над замкнутым полукольцом $\{0, 1\}$. Не используя интерпретацию на графах, докажите следующие факты.

а) $A^* = I_n + A + A^2 + \ldots + A^n$,

б) $\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^* = \begin{pmatrix} A^* & A^*BC^* \\ 0 & C^* \end{pmatrix}$.

Подсказка: покажите, что

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^i = \begin{pmatrix} A^i & A^{i-1}B + A^{i-2}BC + \ldots + BC^{i-1} \\ 0 & C^i \end{pmatrix}.$$

***5.18.** Докажите, что алгоритм поиска кратчайшего пути из раздела 5.8 работает, когда некоторые ребра имеют отрицательную стоимость, но никакой цикл не имеет отрицательной стоимости. Что случится, если у какого-нибудь цикла будет отрицательная стоимость?

****5.19.** Докажите, что положительные и отрицательные действительные числа с $+\infty$ и $-\infty$ не образуют замкнутого полукольца. Как в этом свете объяснить упражнение 5.18? (*Подсказка*: какие свойства замкнутого полукольца фактически используются в алгоритме 5.5?)

5.20. Примените алгоритм 5.6 для поиска кратчайшего расстояния от вершины v_6 в каждую вершину v графа G, представленного на рис. 5.37.

***5.21.** Докажите, что задачу о поиске кратчайшего пути из одного источника в случае неотрицательных стоимостей ребер для графа с e ребрами и n вершинами можно решить за время $O(e \log n)$. (*Подсказка*: используйте подходящую структуру данных, чтобы строки 5 и 8 алгоритма 5.6 можно было эффективно выполнить при $e \ll n^2$.)

****5.22.** Докажите, что задачу о поиске кратчайшего пути из одного источника в случае неотрицательных стоимостей ребер для графа с e ребрами и n вершинами можно решить за время $O(ke + k^{1+1/k})$ для любой фиксированной постоянной k.

5.23. Докажите, что алгоритм поиска кратчайшего пути, приведенный на рис. 5.38, строит кратчайший путь из v_0 в каждую вершину v произвольного графа, у которого могут быть ребра с отрицательными стоимостями, но циклов с отрицательными стоимостями нет.

```
begin
    S ← {v0};
    D[v0] ← 0;
    for для каждой вершины v из V - {v0} do D[v] ← l(v0, v);
    while S ≠ V do
        begin
            выбрать такую вершину w из V - S, что D[w]  — минимальное;
            S  ← S ⋃ {w};
                for v ∈ V, для которых D[v] > D[w] + l(w, v) do
                    begin
                        D[v] = D[w] + l(w, v);
                        S  ← S - {v}
                    end
        end
end
```

Рис. 5.38. Алгоритм нахождения кратчайшего пути из одного источника

***5.24.** Каков порядок времени работы алгоритма, приведенного на рис. 5.38? (*Подсказка*: рассмотрите работу алгоритма на графе с пятью вершинами, стоимости ребер которого указаны на рис. 5.39.)

	1	2	3	4	5
1	0	7	8	9	10
2	0	0	8	9	10
3	0	–2	0	9	10
4	0	–4	–3	0	10
5	0	–7	–6	–5	0

Рис. 5.39. Стоимости ребер для графа с пятью вершинами

5.25. Постройте алгоритм, который распознавал бы, есть ли в данном ориентированном графе, ребрам которого приписаны положительные и отрицательные стоимости, цикл отрицательной стоимости.

5.26. Измените правило выбора вершины w в алгоритме, предствленном на рис. 5.38, так, чтобы гарантировать временную сложность $O(n^3)$ в случае, когда ребрам приписаны произвольные стоимости.

5.27. Напишите алгоритм, который по данной $(n \times n)$-матрице M положительных целых чисел строил бы последовательность смежных элементов, начинающуюся с $M[n, 1]$ и оканчивающуюся $M[1, n]$, так, чтобы минимизировать сумму абсолютных значений разностей между соседними элементами. Два элемента $M[i, j]$ и $M[k, l]$ считаются *смежными*, если (*а*) $i = k \pm 1$ и $j = l$ или (*б*) $i = k$ и $j = l \pm 1$. Например, для рис. 5.40 решением будет последовательность 7, 5, 8, 7, 9, 6, 12.

$$\begin{bmatrix} 1 & 9 & 6 & 12 \\ 8 & 7 & 3 & 5 \\ 5 & 9 & 11 & 4 \\ 7 & 3 & 2 & 6 \end{bmatrix}$$

Рис. 5.40. Матрица положительных целых чисел

5.28. Алгоритм на рис. 5.24 для каждого $v \in V$ вычисляет наименьшую стоимость пути среди всех путей из v_0 в v. Модифицируйте этот алгоритм так, чтобы он строил некоторый путь минимальной стоимости для каждой вершины v из V.

**5.29. Напишите программу, реализующую доминаторный алгоритм из раздела 5.11.

Проблемы для исследования

5.30. Для многих проблем, связанных с графами, поиск в глубину мог бы принести пользу. Одна из них касается *k-связности*. Неориентированный граф называется *k-связным*, если для каждой пары вершин v и w существуют такие k путей между ними, что ни одна вершина (кроме v и w) не входит более чем в один путь. Таким образом, бисвязность означает двухсвязность. Hopcroft and Tarjan [1973b] построили алгоритм, отыскивающий 3связные компоненты за линейное время. Естественно выдвинуть гипотезу, что для каждого k существует алгоритм, отыскивающий k-связные компоненты за линейное (по числу вершин и ребер) время. Можете ли вы найти такой алгоритм?

5.31. Другой интересной проблемой, для решения которой, возможно, полезен метод поиска в глубину (а может быть, и нет), является построение алгоритма, отыскивающего за линейное (по числу ребер) время остовное дерево минимальной стоимости.

5.32. Заслуживает рассмотрения и задача поиска кратчайшего пути из одного источника, когда $e \ll n^2$. Существует ли алгоритм, отыскивающий за время $O(e)$ хотя бы кратчайшее расстояние между двумя конкретными точками? Читателю было бы полезно ознакомиться с упражнениями 5.21 и 5.22, основанными на работах Johnson [1973], а также Spira and Pan [1973], в которых показано, что в общем случае для таких алгоритмов требуется $n^2/4$ сравнений, если алгоритмы используют только сравнения между суммами стоимостей ребер. Wagner [1974] применил метод поразрядной сортировки, чтобы получить алгоритм сложности $O(e)$ в случае, когда в качестве стоимостей ребер фигурируют малые целые числа.

5.33. Как доказал Kerr [1972], если допускаются только операции `MIN` и +, то проблема поиска кратчайших путей между всеми парами вершин требует kn^3 шагов, где k — некоторая положительная постоянная. M. O. Rabin усилил этот результат до $n^3/6$. Тем не менее может оказаться, что если допустить другие операции, то можно получить алгоритм с временной сложностью $O(n^3)$. Например, как мы увидим в следующей главе, можно построить транзитивное замыкание (или, что эквивалентно, перемножить две булевы матрицы) быстрее, чем за $O(n^3)$ шагов, если допустить операции, отличные от булевых.

Библиографические замечания

Теории графов посвящены книги Berge [1958] и Harary [1969]. Алгоритм 5.1 построения остовных деревьев минимальной стоимости взят из Kruskal [1956]. Prim [1957] указывает другой подход к этой задаче.

Алгоритм, касающийся двусвязности и использующий метод поиска в глубину, изобрел J. E. Hopcroft. Алгоритм для сильно связных компонентов взят из Tarjan [1972]. В литературе описаны многочисленные применения метода поиска в глубину для построения оптимальных или наилучших из известных алгоритмов. Hopcroft and Tarjan [1973a] приводят линейный тест планарности. В работе Hopcroft and Tarjan [1973b] описан линейный алгоритм нахождения 3связных компонент. Tarjan [1973a] на основе той же идеи разработал наилучший из известных алгоритмов нахождения доминаторов. Кроме того, Tarjan [1973b] предложил тест для "редуцируемости потоковых графов".

Алгоритм 5.5, общий алгоритм нахождения путей, по существу, принадлежит Kleene [1956], который использовал его в связи с регулярными выражениями (раздел 9.1). Данная здесь форма этого алгоритма заимствована у McNaughton and Yamada [1960]. Алгоритм сложности $O(n^3)$, строящий транзитивное замыкание, взят из работы Warshall [1962]. Аналогичный алгоритм поиска кратчайших путей для всех пар вершин заимствован у Floyd (см. также Hu [1968]). Алгоритм для одного источника изложен по работе Dijkstra [1959]. Spira and Pan [1973] показали, что алгоритм Дейкстры, по существу, оптимален, если в качестве модели брать деревья решений.

Danzig, Blattner и Rao [1967] заметили, что наличие ребер с отрицательными стоимостями не влияет на решение задачи поиска кратчайших путей между всеми парами точек, если в графе нет цикла с отрицательной стоимостью. Johnson [1973] обсуждает задачи с одним источником при наличии ребер с отрицательными стоимостями и приводит решения упражнений 5.21 и 5.22. Spira [1973] приводит алгоритм плоиска кратчайшего пути за среднее время $O\left(n^2 \log^2 n\right)$.

Связь между задачей поиска путей и умножением матриц описана в работах Munro [1971] и Furman [1970] (теорема 5.7), а также Fischer and Mayer [1971] (теорема 5.6). Связь с транзитивной редукцией (упражнения 5.13–5.15) заимствована у Aho, Garey and Ullman [1972]. Even [1973] обсуждает *k*связность графов.

ГЛАВА 6

Умножение матриц и связанные с ним операции

В этой главе мы исследуем асимптотическую вычислительную сложность умножения матриц, элементы которых принадлежат произвольному кольцу. Мы увидим, что обычный алгоритм умножения матриц, имеющий сложность $O(n^3)$, можно асимптотически улучшить. Для того чтобы перемножить две матрицы размерностью $n \times n$, достаточно времени $O(n^{2,81})$. Более того, мы увидим, что и другие операции, такие как разложение LUP, обращение матрицы и вычисление определителя, сводятся к умножению матриц, и потому их можно выполнить так же быстро, как и умножение матриц. Затем мы покажем, что умножение матриц сводится к обращению матрицы, поэтому любое улучшение асимптотического времени решения одной из этих задач привело бы к аналогичному улучшению другой. Мы закончим главу двумя алгоритмами умножения булевых матриц, асимптотическая временна*я* сложность которых меньше $O(n^3)$.

С учетом технических возможностей современных вычислительных машин алгоритмы этой главы в большинстве случаев не стоит рассматривать как практические. Одна из причин состоит в том, что из-за скрытых постоянных множителей они работают быстрее обычных алгоритмов сложности $O(n^3)$ только при достаточно больших значениях n. Кроме того, для этого семейства алгоритмов недостаточно хорошо изучено поведение ошибок округления. Тем не менее идеи данной главы, на наш взгляд, заслуживают рассмотрения, поскольку они демонстрируют, что очевидные алгоритмы не всегда наилучшие, а также потому, что могут служить основой разработки еще более эффективных и действительно практических алгоритмов для этого важного класса задач.

6.1. Основные понятия

В этом разделе представлены основные алгебраические понятия, полезные при исследовании задач умножения матриц. Читатели, знакомые с кольцами и линейной алгеброй, могут сразу перейти к разделу 6.2.

Определение. *Кольцом* называется алгебраическая структура $(S, +, \cdot, 0, 1)$, где S — множество элементов, + и · — бинарные операции на S. Для всех a, b и c из S выполняются следующие соотношения.

1. $(a + b) + c = a + (b + c)$ и $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (операции + и · являются ассоциативными).
2. $a + b = b + a$ (операция + является коммутативной).
3. $(a + b) \cdot c = a \cdot c + b \cdot c$ и $a \cdot (b + c) = a \cdot b + a \cdot c$ (операция · является дистрибутивной относительно +).
4. $a + 0 = 0 + a = a$ (0 является единичным элементом для операции +).
5. $a \cdot 1 = 1 \cdot a = a$ (1 является единичным элементом для операции ·).
6. Для каждого $a \in S$ существует *обратный элемент* $-a$, т.е. $a + (-a) = (-a) + a = 0$.

Заметим, что свойство существования обратного элемента относительно сложения в замкнутом полукольце может не выполняться (раздел 5.6). Свойство 4 замкнутого полукольца — существование и единственность бесконечных сумм — не всегда выполняется в кольце. Кольцо, в котором операция является коммутативной, называется *коммутативным*.

Если в коммутативном кольце для каждого ненулевого элемента a существует элемент a^{-1}, обратный относительно умножения, т.е. $a \cdot a^{-1} = a^{-1} \cdot a = 1$, то кольцо называется *полем*.

Пример 6.1. Действительные числа образуют кольцо, если в качестве операций + и · используются обычные сложение и умножение. В то же время действительные числа не образуют замкнутого полукольца.

Система $(\{0, 1\}, +, \cdot, 0, 1)$, где + — сложение по модулю 2 и обычное умножение, образует кольцо, но не замкнутое полукольцо, поскольку значение 1 + 1 + ... не определено. Однако если переопределить операцию + так, что $a + b = 0$, если $a = b = 0$ и $a + b = 1$ в прочих случаях, то получится замкнутое полукольцо S_2 из упражнения 5.1. S_2 не является кольцом, поскольку 1 не имеет обратного. □

Введем важный класс колец, образованных матрицами.

Определение. Пусть $R = (S, +, \cdot, 0, 1)$ — кольцо и M_n — множество матриц размерностью $n \times n$, элементы которых принадлежат R. Пусть 0_n обозначает матрицу размерностью $n \times n$, все элементы которой равны 0, и I_n — *единичную матрицу* размерностью $n \times n$, в которой на главной диагонали стоят 1, а на остальных местах 0. Для матриц A и B из M_n обозначим через $A +_n B$ такую

матрицу C размерностью $n \times n$, что $C[i,j] = A[i,j] + B[i,j]$,[1] а через $A \cdot_n B$ — такую матрицу D размерностью $n \times n$, что $D[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$.

Лемма 6.1. $(M_n, +_n, \cdot_n, 0_n, I_n)$ — кольцо.

Доказательство. Элементарное упражнение. □

Заметим, что определенная выше операция умножения матриц $\cdot_n$ не коммутативна при $n > 1$, даже если умножение в исходном кольце R коммутативно. Мы будем использовать обозначения + и · вместо $+_n$ и $\cdot_n$, если это не будет вызывать путаницы со сложением и умножением в исходном кольце R. Кроме того, мы будем часто опускать знак умножения, если он очевиден.

Пусть R — кольцо и M_n — кольцо матриц размерностью $n \times n$ с элементами из R. Допустим, n — четное число. Тогда матрицу M_n размерностью $n \times n$ можно разбить на четыре матрицы размерностью $(n/2) \times (n/2)$. Пусть $R_{2,n/2}$ — кольцо матриц размерностью 2×2 с элементами из $M_{n/2}$. Легко непосредственно проверить, что умножение и сложение матриц размерностью $n \times n$ в кольце M_n эквивалентно умножению и сложению соответствующих матриц размерностью 2×2 в $R_{2,n/2}$, элементами которых являются матрицы размерностью $(n/2) \times (n/2)$.

Лемма 6.2. Пусть f — такое отображение из M_n в $R_{2,n/2}$, что $f(A)$ — матрица

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

из $R_{2,n/2}$, где A_{11} — левый верхний квадрант матрицы A; A_{12} — правый верхний квадрант; A_{21} — левый нижний квадрант; A_{22} — правый нижний квадрант. Тогда

1. $f(A + B) = f(A) + f(B)$,

2. $f(AB) = f(A) \cdot f(B)$.

Доказательство. Лемма доказывается путем элементарной подстановки определений операций + и · в кольце $M_{n/2}$ в определения операций + и · в кольце $R_{2,n/2}$. □

Важность леммы 6.2 обусловлена тем, что она позволяет создать алгоритм умножения матриц размерностью $n \times n$ из алгоритмов умножения матриц размерностью 2×2 и $(n/2) \times (n/2)$. Мы используем эту лемму в следующем разделе,

---

[1] $M[i,j]$ — элемент, стоящий в матрице M на пересечении i-й строки и j-го столбца.

где построим асимптотически быстрый алгоритм умножения матриц. А пока подчеркнем тот факт, что алгоритм умножения матриц размерностью 2 × 2 будет не произвольным, а специально ориентированным на кольцо $R_{2,n/2}$. Поскольку кольцо $R_{2,n/2}$ не коммутативно, даже если кольцо R коммутативно, то этот алгоритм умножения матриц размерностью 2 × 2 не может предполагать коммутативности умножения матриц размерностью $(n/2) \times (n/2)$). Однако он может использовать любое из свойств кольца.

Определение. Пусть A — матрица размерностью $n \times n$ с элементами из некоторого поля. Матрицей A^{-1}, *обратной* к матрице A, называется такая матрица размерностью $n \times n$, что $AA^{-1} = I_n$.

Легко показать, что если матрица A^{-1} существует, то она единственная и $AA^{-1} = A^{-1}A = I_n$. Кроме того, $(AB)^{-1} = B^{-1}A^{-1}$.

Определение. Пусть A — матрица размерностью $n \times n$. *Определителем* $\det(A)$ матрицы A называется сумма, взятая по всем перестановкам $p = (i_1, i_2, ..., i_n)$ целых чисел от 1 до n, произведений

$$(-1)^{k_p} \prod_{j=1}^{n} A(j, i_j),$$

где $k_p = 0$, если перестановка p четная (т.е. ее можно получить из (1, 2, ..., n) четным числом перестановок), и $k_p = 1$, если перестановка p нечетная (получается нечетным числом перестановок).

Нетрудно показать, что каждая перестановка является либо четной, либо нечетной (но не одновременно).

Пример 6.2. Пусть

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Для чисел 1, 2, 3 существует шесть перестановок: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). Очевидно, перестановка (1, 2, 3) является четной, поскольку получается с помощью 0 перестановок. Перестановка (2, 3, 1) является четной, поскольку, переставляя 2 и 3 в (1, 2, 3), получаем (1, 3, 2), затем переставляем 1 и 2 и получаем (2, 3, 1). Аналогично (3, 1, 2) — четная перестановка, а остальные три — нечетные. Таким образом, $\det(A) = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$. □

Пусть матрица A размерностью $n \times n$ образована элементами из некоторого поля. Можно показать, что матрица A^{-1} существует тогда и только тогда, когда $\mathbf{det}(A) \neq 0$, и что $\mathbf{det}(AB) = \mathbf{det}(A)\mathbf{det}(B)$. В случае $\mathbf{det}(A) \neq 0$ матрицу A называют *невырожденной*.

Определение. Матрица A размерностью $m \times n$ называется *верхней треугольной*, если $A[i,j] = 0$ при $1 \le j < i \le m$, и *нижней треугольной*, если $A[i,j] = 0$ при $1 \le i < j \le n$.

Умножение двух верхних треугольных матриц дает верхнюю треугольную матрицу. Аналогичное утверждение верно и для нижних треугольных матриц.

Лемма 6.3. Если A — квадратная верхняя или нижняя треугольная матрица, то

а) определитель $\mathbf{det}(A)$ равен произведению элементов, стоящих на главной диагонали (т.е. $\prod_i A[i,i]$);

б) матрица A является невырожденной тогда и только тогда, когда все элементы на главной диагонали отличны от нуля.

Доказательство. (*а*) Любая перестановка $(i_1, i_2, ..., i_n)$, кроме $(1, 2, ..., n)$, содержит такой элемент i_j, что $i_j < j$, и такой элемент i_k, что $i_k > k$. Следовательно, в сумме, определяющей $\mathbf{det}(A)$, все слагаемые, кроме того, которое соответствует перестановке $(1, 2, ..., n)$, равны 0. (*б*) Непосредственно следует из (*а*). □

Определение. *Нормированной* называется матрица, у которой на главной диагонали стоят 1 (элементы вне главной диагонали произвольны).

Заметим, что определитель нормированной верхней треугольной или нижней треугольной матрицы равен 1.

Определение. *Матрицей перестановки* называется такая матрица из 0 и 1, что в каждой строке и каждом столбце стоит ровно одна единица.

Определение. *Подматрицей* матрицы A называется матрица, получаемая удалением некоторых строк и столбцов матрицы A. *Главной подматрицей* матрицы A размерностью $n \times n$ называется квадратная подматрица матрицы A, состоящая из первых k строк первых k столбцов матрицы A, $1 \le k \le n$.

Рангом $\mathbf{rank}(A)$ матрицы A называется размер ее наибольшей квадратной невырожденной подматрицы. Например, ранг матрицы перестановки размерностью $n \times n$ равен n.

Заметим, что если $A = BC$, то **rank**$(A) \le$ MIN(**rank**(B), **rank**(C)). Кроме того, если матрица A ранга m состоит из m строк, то любые ее k строк образуют матрицу ранга k.

Определение. Матрицей A^T, *транспонированной* к матрице A, называют матрицу, получаемую перестановкой элементов $A[i, j]$ и $A[j, i]$ для всех i и j.

6.2. Алгоритм Штрассена для умножения матриц

Пусть A и B — две матрицы размерностью $n \times n$, причем n — степень числа 2. По лемме 6.2 можно разбить каждую матрицу A и B на четыре матрицы размерностью $(n/2) \times (n/2)$ и через них выразить произведение матриц A и B:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

где

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned} \tag{6.1}$$

Если рассматривать A и B как матрицы размерностью 2×2, элементами каждой из которых служат матрицы размерностью $(n/2) \times (n/2)$, то их произведение можно выразить через суммы и произведения матриц размерностью $(n/2) \times (n/2)$ по формуле (6.1). Допустим, что матрицы C_{ij} вычисляются с помощью m умножений и a сложений (или вычитаний) матриц размерностью $(n/2) \times (n/2)$. Рекурсивно применяя этот алгоритм, можно вычислить произведение двух матриц размерностью $n \times n$ за время $T(n)$, удовлетворяющее неравенству

$$T(n) \le mT\left(\frac{n}{2}\right) + \frac{an^2}{4}, \quad n > 2, \tag{6.2}$$

если n — степень числа 2. Первое слагаемое в правой части неравенства (6.2) — это сложность умножения m пар матриц размерностью $(n/2) \times (n/2)$, а второе — временная сложность выполнения a сложений при условии, что каждое сложение или вычитание двух матриц размерностью $(n/2) \times (n/2)$ занимает $n^2/4$ времени. Рассуждения, аналогичные проведенным в теореме 2.1 для $c = 2$, показывают, что при $m > 4$ решение неравенства (6.2) ограничено сверху величиной $kn^{\log m}$, где k — некоторая константа. Вид этого решения не зависит от числа a, т.е. от количества сложений. Таким образом, если $m < 8$, то мы получаем метод, который асимпто-

тически лучше обычного метода умножения матриц, имеющего временную сложность $O(n^3)$.

Штрассен изобрел остроумный метод умножения двух матриц размерностью 2×2 с элементами из произвольного кольца, в котором достаточно семи умножений. Рекурсивно применяя свой метод, он смог умножить две матрицы размерностью $n \times n$ за время $O\left(n^{\log 7}\right)$, что по порядку примерно равно $n^{2,81}$.

Лемма 6.4. Произведение двух матриц размерностью 2×2 с элементами из произвольного кольца можно вычислить, выполнив 7 умножений и 18 сложений (вычитаний).

Доказательство. Чтобы вычислить произведение матриц

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

сначала вычислим произведения

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}), \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}), \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}), \\ m_4 &= (a_{11} + a_{12})b_{22}, \\ m_5 &= a_{11}(b_{12} - b_{22}), \\ m_6 &= a_{22}(b_{21} - b_{11}), \\ m_7 &= (a_{21} + a_{22})b_{11}. \end{aligned}$$

Затем вычисляем величины c_{ij} по формулам

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6, \\ c_{12} &= m_4 + m_5, \\ c_{21} &= m_6 + m_7, \\ c_{22} &= m_2 - m_3 + m_5 - m_7. \end{aligned}$$

Количество операций подсчитывается непосредственно. Доказательство того, что в результате получаются требуемые величины c_{ij}, представляет собой простое алгебраическое упражнение на использование аксиом кольца. □

В упражнении 6.5 приведен способ вычисления произведения двух матриц размерностью 2×2 за 7 умножений и 15 сложений.

Теорема 6.1. Две матрицы размерностью $n \times n$ с элементами из произвольного кольца можно перемножить за $O\left(n^{\log 7}\right)$ арифметических операций.

Доказательство. Сначала рассмотрим случай $n = 2^k$. Пусть $T(n)$ — количество арифметических операций, необходимых для умножения двух матриц размерностью $n \times n$. По лемме 6.4

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \quad \text{при } n \geq 2.$$

Следовательно, вследствие простой модификации теоремы 2.1 $T(n)$ составляет $O\left(7^{\log n}\right)$, или $O\left(n^{\log 7}\right)$.

Если число n не является степенью числа 2, то каждую матрицу вложим в матрицу, порядок которой равен наименьшей степени числа 2, большей n. Это увеличит порядок матриц не более чем вдвое и, значит, увеличит константу не более чем в 7 раз. Таким образом, $T(n)$ имеет порядок $O\left(n^{\log 7}\right)$ для всех $n \geq 1$. □

Теорема 6.1 касается только порядка роста функции $T(n)$. Однако, для того, чтобы выяснить, для каких значений n алгоритм Штрассена работает быстрее обычного алгоритма, необходимо найти соответствующую мультипликативную константу. Если же число n не является степенью числа 2, то простое вложение каждой исходной матрицы в матрицу, порядок которой равен наименьшей степени числа 2, большей n, даст слишком большую константу. Вместо этого можно вложить каждую матрицу в матрицу порядка $2^p r$ для некоторого небольшого числа r и p раз применить лемму 6.4, умножая матрицы размерностью $r \times r$ обычным методом, имеющим сложность $O(r^3)$. Альтернативно можно было бы написать более универсальный рекурсивный алгоритм, который при четном n расщеплял бы каждую матрицу на четыре подматрицы, как и раньше, а при нечетном n сначала увеличивал бы порядок матриц на единицу.

Следует подчеркнуть, что, вычислив мультипликативную константу, мы найдем всего лишь оценку числа арифметических операций. Чтобы сравнить метод Штрассена с обычным методом умножения матриц, надо также учесть дополнительную сложность процедур, связанных с поиском элементов матриц.

6.3. Обращение матриц

В данном разделе мы покажем, что задачу обращения матриц из определенного класса можно свести к задаче умножения матриц. В этот класс входят все обратимые треугольные матрицы, но не входят их обратные матрицы. Позднее мы обобщим этот результат на произвольные обратимые матрицы. В этом разделе и в разделах 6.4 и 6.5 рассматриваются матрицы с элементами из некоторого поля.

Лемма 6.5. Пусть матрицу A можно представить в виде

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.$$

Допустим, что существует матрица A_{11}^{-1}. Определим $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12}$ и предположим, что матрица Δ^{-1} существует. Тогда

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{bmatrix}. \tag{6.3}$$

Доказательство. Непосредственные алгебраические манипуляции приводят к следующему результату:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & \Delta \end{bmatrix} \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix},$$

где $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12}$. Следовательно,

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & \Delta^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix} =$$

$$= \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{bmatrix}. \qquad \square$$

Лемму 6.5 нельзя применять ко всем невырожденным матрицам. Например, матрица перестановки A размерностью $n \times n$, у которой $A[i,j] = 1$, если $j = n - i + 1$, и $A[i,j] = 0$ в противном случае, является невырожденной, но $\mathbf{det}(A_{11}) = 0$ для любой главной подматрицы A_{11}. Тем не менее лемма 6.5 применима ко всем невырожденным нижним и верхним треугольным матрицам.

Лемма 6.6. Если матрица A является невырожденной верхней (нижней) треугольной матрицей, то матрицы A_{11} и Δ в лемме 6.5 имеют обратные невырожденные верхние (нижние) треугольные матрицы.

Доказательство. Предположим, что матрица A является верхней треугольной матрицей. Доказательство для случая, когда матрица A является нижней треугольной, проводится аналогично. Очевидно, матрица A_{11} является невырожденной верхней треугольной матрицей, тогда матрица A_{11}^{-1} существует. Очевидно, что $A_{21} = 0$. Таким образом, матрица $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12} = A_{22}$ является невырожденной верхней треугольной матрицей. $\square$

Теорема 6.2. Пусть $M(n)$ — время, необходимое для умножения двух матриц размерностью $n \times n$ над кольцом. Если для всех m выполняются неравенства $8M(m) \geq M(2m) \geq 4M(m)$, то существует константа c такая, что обратную матрицу к любой невырожденной верхней (нижней) треугольной матрице A можно вычислить за время $cM(n)$.

Доказательство. Мы докажем это утверждение для случая, когда n — степень числа 2. Очевидно, если число n не является степенью числа 2, мы можем вложить матрицу A в матрицу вида

$$\begin{bmatrix} A & 0 \\ 0 & I_m \end{bmatrix},$$

где $m + n \leq 2n$ — степень числа 2. Тогда, увеличивая константу c не более чем в 8 раз, этот результат можно получить для произвольного числа n.[2]

Предполагая, что n — степень числа 2, мы можем разбить матрицу A на четыре подматрицы размерностью $(n/2) \times (n/2)$ и рекурсивно применить формулу (6.3). Отметим, что $A_{21} = 0$, поэтому $\Delta = A_{22}$. Следовательно, для обращения треугольных матриц A_{11} и Δ требуется $2T(n/2)$ времени; $2M(n/2)$ времени — для умножения двух нетривиальных умножений и $n^2/4$ времени — чтобы поставить "минус" в правом верхнем углу. Из условий теоремы и неравенства $M(1) \geq 1$ получаем, что $n^2/4 \leq M(n/2)$. Следовательно,

$$\begin{aligned} &T(1) = 1, \\ &T(n) \leq 2T\left(\frac{n}{2}\right) + 3M\left(\frac{n}{2}\right) \quad \text{при } n \geq 2. \end{aligned} \tag{6.4}$$

Нетрудно показать, что из (6.4) следует неравенство $T(n) \leq 3M(n)/2$. □

6.4. LUP-разложение матриц

Один из эффективных методов решения системы линейных уравнений основан на применении так называемого LUP-разложения.

Определение. *LU-разложением* матрицы A размерностью $m \times n$, $m \leq n$, называется ее представление в виде $A = LU$, где L — нормированная нижняя треугольная матрица размерностью $m \times m$; U — верхняя треугольная матрица размерностью $m \times n$.

[2] И снова более тщательный анализ дает для произвольного n константу c, мало отличающуюся от наилучшей известной константы для случая, когда n — степень числа 2.

Уравнение $A\mathbf{x} = \mathbf{b}$, где A — это матрица размерностью $n \times n$; $\mathbf{x}$ — n-мерный вектор-столбец неизвестных; $\mathbf{b}$ — n-мерный вектор-столбец, можно решить, сначала выполнив разложение матрицы A в произведение LU, если это возможно. Затем представим систему $A\mathbf{x} = \mathbf{b}$ в виде $LU\mathbf{x} = \mathbf{b}$. Чтобы получить решение $\mathbf{x}$, решим сначала систему $L\mathbf{y} = \mathbf{b}$ относительно вектора $\mathbf{y}$, а затем — систему $U\mathbf{x} = \mathbf{y}$ относительно вектора $\mathbf{x}$.

Трудность применения этого метода заключается в том, что LU-разложение матрицы A может оказаться невозможным, даже если она является невырожденной. Например, матрица

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

является невырожденной, но у нее нет LU-разложения. Однако если матрица A является невырожденной, то найдется такая матрица перестановки P, что матрица AP^{-1} имеет LU-разложение. Сформулируем алгоритм, который по любой невырожденной матрице A находит такие матрицы L, U и P, что $A = LUP$. Матрицы L, U и P образуют *LUP-разложение* матрицы A.

Алгоритм 6.1. LUP-разложение

Вход. Невырожденная матрица M размерностью $n \times n$, где n — степень числа 2.

Выход. Матрицы L, U и P, для которых выполняется равенство $M = LUP$, где L — нормированная нижняя треугольная матрица; U — верхняя треугольная; P — матрица перестановки.

Метод. Вызываем процедуру `FACTOR`(M, n, n), где `FACTOR` — рекурсивная процедура, показанная на рис. 6.4. При описании этой процедуры используются рис. 6.1–6.3, где затемненная область матрицы состоит из одних нулей.

Каждый рекурсивный вызов процедуры `FACTOR` применяется к подматрице A размерностью $m \times p$ матрицы M размерностью $n \times n$. При каждом вызове число m является степенью числа 2 и $m \le p \le n$. Результатами этой процедуры являются три матрицы: L, U и P (рис. 6.1). □

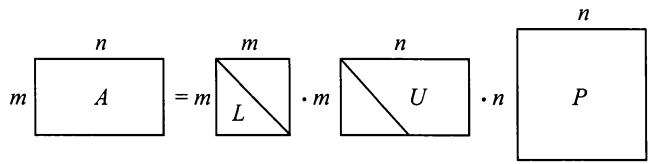
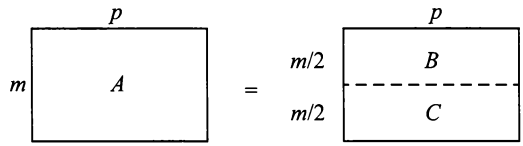
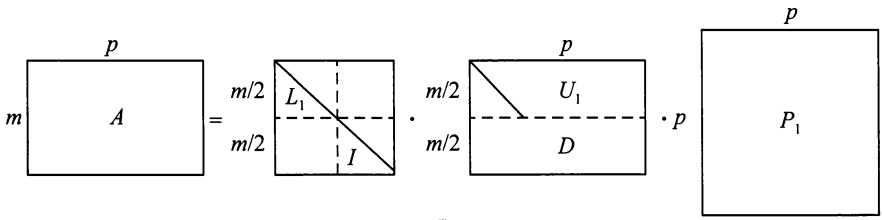


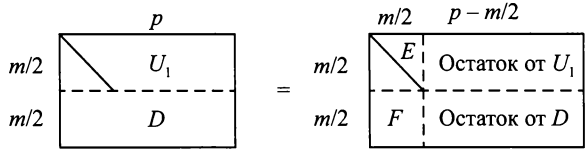
Рис. 6.1. Искомый результат процедуры `FACTOR`



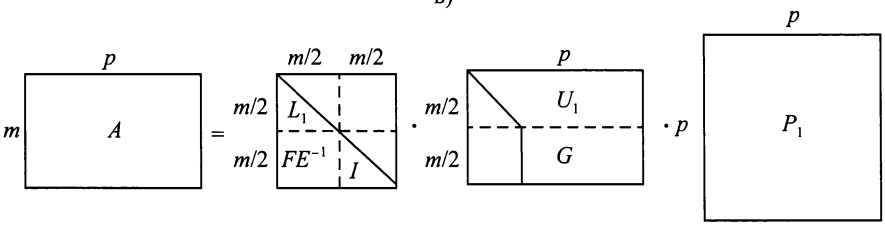
а)



б)



в)



г)

Рис. 6.2. Этапы процедуры FACTOR: *а*) начальное разложение матрицы A; (*б*) факторизация после первого вызова процедуры FACTOR; (*в*) разложение матриц U_1 и D; (*г*) обнуление нижнего левого угла матрицы D [3]

[3]Напомним, что I — нормированная нижняя или верхняя треугольная матрица.

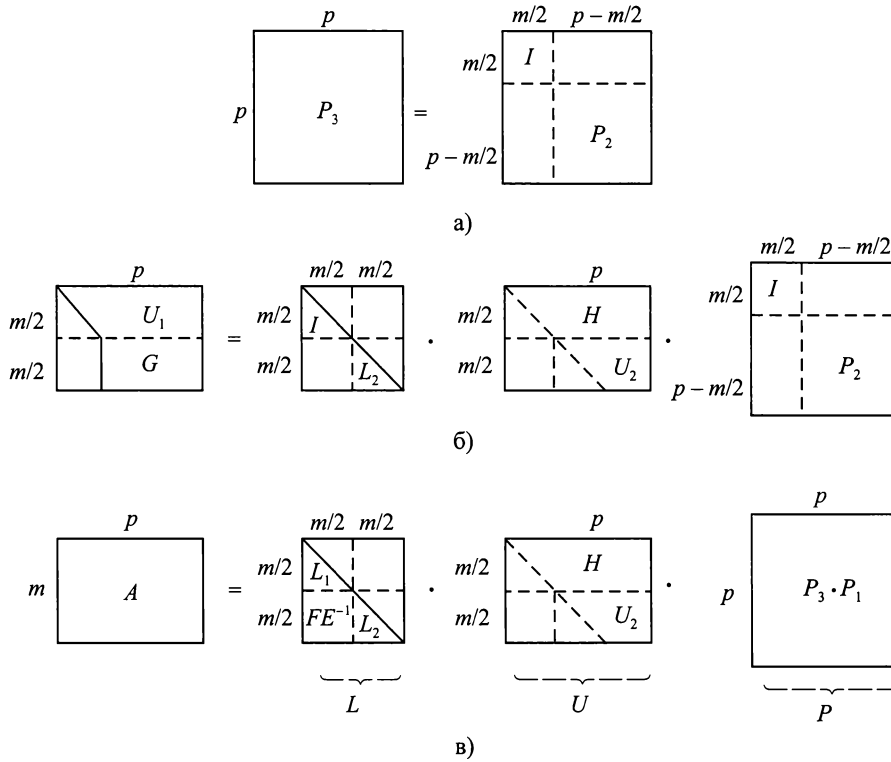


Рис. 6.3. Завершение процедуры `FACTOR`: (*а*) создание матрицы P_3; (*б*) разложение матриц U_1 и G; (*в*) разложение матрицы A

```
      procedure  FACTOR(A, m, p):
      if m = 1  then
          begin
1.          пусть L = [1] (т.е. L — нормированная матрица 1 × 1;
2.          найти, если можно, столбец c матрицы A с ненулевым
                элементом, и пусть P будет матрицей p × p,
                меняющей местами столбцы 1 и c;
            comment Обратите внимание на то, что P = P^-1;
3.          пусть U = AP;
4.          return (L, U, P)
          end
      else
          begin
5.          разбить A на матрицы B и C размерностью (m/2) × p,
                как показано на рис. 6.2, а;
6.          вызвать FACTOR(B, m/2, p), чтобы получить L1, U1, P1;
7.          вычислить D = CP1^-1;
            comment в этот момент матрицу A можно записать
            как произведение трех матриц, показанных на рис. 6.2, б;
8.          пусть E и F — первые m/2 столбцов соответственно
                матриц U1 и D (рис. 6.2, в);
9.          вычислить G = D - FE^-1 U1;
            comment обратите внимание на то, что первые m/2 столбцов
                матрицы G состоят из одних нулей. Матрицу A
                можно представить в виде произведения матриц,
                показанных на рис. 6.2, г;
10.         пусть G'— самые правые p - m/2 столбцов матрицы G;
11.         вызвать FACTOR(G', m/2, p - m/2), чтобы получить L2, U2 и P2;
12.         пусть P3 — матрица перестановки p × p, у которой в левом
                верхнем углу стоит I_m/2, а в правом нижнем - P2
                (рис. 6.3, а);
13.         вычислить H = U1 P3^-1;
            comment в этот момент матрицу, составленную из U1 и G,
                можно представить так, как показано на рис. 6.3, б.
                Если в рис. 6.2, г, подставить правую часть
                равенства из рис. 6.3,6, то получится представление
                матрицы A в виде произведения пяти матриц. Первые
                две из них — нормированные нижние треугольные
                матрицы, третья — верхняя  треугольная, последние
                две — матрицы перестановок. Умножим первые две
                и последние две, чтобы получить искомое
                разложение матрицы A;
14.         пусть L — матрица m × m, состоящая из L1, O_m/2, FE^-1 и L2,
                как показано на рис. 6.3, в;
15.         пусть U — это матрица m × p, у которой в верхней половине
                стоит H, а в нижней — O_m/2 и U2 (рис. 6.3, в);
16.         пусть P — произведение P3 P1;
17.         return (L, U, P)
          end
```

Рис. 6.4. Процедура FACTOR

Пример 6.3. Найдем LUP-разложение матрицы

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix}.$$

Для начала вызовем процедуру FACTOR(M, 4, 4), которая немедленно вызывает процедуру

$$\text{FACTOR}\left(\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix}, 2, 4\right).$$

Взяв эту матрицу в качестве матрицы A, вызовем процедуру FACTOR([0 0 0 1], 1, 4). В результате получим

$$L_1 = [1], U_1 = [1 \quad 0 \quad 0 \quad 0], P_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

где матрица P_1 меняет местами столбцы 1 и 4.

В строке 7 вычислим[4]

$$D = CP_1^{-1} = [0 \quad 0 \quad 2 \quad 0]\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = [0 \quad 0 \quad 2 \quad 0].$$

Затем в строке 8 получаем $E = [1]$ и $F = [0]$, так что $G = D = [0\ 0\ 2\ 0]$. В строке 10 получаем $G' = [0\ 2\ 0]$, поэтому после строки 11 имеем

$$L_2 = [1], \quad U_2 = [2 \quad 0 \quad 0], \quad P_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

В строке 12

$$P_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

[4] В этом примере все матрицы перестановок совпали со своими обратными.

в строке 13

$$H = U_1 P_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}.$$

Таким образом, процедура

$$\texttt{FACTOR}\left(\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix}, 2, 4 \right)$$

возвращает следующий результат:

$$L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \text{ и } P = P_3 P_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Теперь вернемся к процедуре FACTOR(M, 4, 4) в строке 6, где матрицы L, U и P превращаются в матрицы L_1, U_1, P_1 соответственно. Тогда в строке 7 получаем

$$D = CP_1^{-1} = \begin{bmatrix} 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}.$$

В строке 8

$$E = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \text{ и } F = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Следовательно, в строке 9

$$G = D = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix},$$

а в строке 10

$$G' = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix}.$$

Читатели могут самостоятельно проверить, что вызов процедуры FACTOR(G', 2, 2) приводит к следующим результатам:

$$L_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad U_2 = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix} \text{ и } P_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Таким образом, в строке 12 матрица P_3 равна I_4, а в строке 13

$$H = U_1 P_3^{-1} = U_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}.$$

Следовательно, в строках 14–16

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \quad \text{и} \quad P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \quad \square$$

Перейдем к анализу и доказательству корректности алгоритма 6.1.

Теорема 6.3. Для любой невырожденной матрицы A алгоритм 6.1 вычисляет матрицы L, U и P, такие, что $A = LUP$.

Доказательство. Предоставим читателям в качестве упражнения доказать корректность разложений, изображенных на рис. 6.2 и 6.3. Необходимо лишь показать, что

1) в строке 2 процедуры `FACTOR` всегда существует ненулевой столбец и
2) в строке 9 всегда существует матрица E^{-1}.

Пусть A — матрица размерностью $m \times n$. Покажем индукцией по m, где m — степень числа 2, что если матрица A имеет ранг m, то процедура `FACTOR` вычислит такие матрицы L, U и P, что $A = LUP$ и L, U, P — нижняя треугольная матрица, верхняя треугольная матрица и матрица перестановки с рангами m, m и n соответственно. Кроме того, первые m столбцов матрицы U образуют подматрицу ранга m. Если $m = 1$, то в матрице A должен быть ненулевой элемент, так что база индукции, очевидно, является истинной. Допустим, что $m = 2^k$, $k \geq 1$. Поскольку матрица A имеет m столбцов и ранг m, то каждая из матриц B и C, появляющихся в строке 5, имеет $m/2$ столбцов и ранг $m/2$. По индуктивному предположению вызов процедуры `FACTOR` в строке 6 выдает требуемые матрицы L_1,U_1 и P_1, причем первые $m/2$ столбцов матрицы U_1 образуют матрицу ранга $m/2$. Следовательно, матрица E^{-1}, необходимая в строке 9, существует.

На рис. 6.2, *г*, видно, что матрица A равна произведению трех матриц, у одной из которых в верхней части стоит U_1, а в нижней — G. Ранг этой матрицы должен быть равен m, поскольку матрица A имеет ранг m. Следовательно, матрица G имеет ранг $m/2$. Поскольку первые $m/2$ столбцов матрицы G состоят из нулей, а матрица G' получается из G вычеркиванием ее первых $m/2$ столбцов, то ранг

матрицы G' также равен $m/2$. Следовательно, по предположению индукции вызов процедуры FACTOR в строке 11 дает искомые матрицы L_2, U_2, P_2. Отсюда непосредственно следует индуктивная гипотеза. □

Прежде чем переходить к анализу временной сложности, заметим, что матрицу перестановки можно представить в виде такого массива P, что $P[i] = j$ тогда и только тогда, когда единица в столбце i стоит в строке j. Таким образом, две матрицы перестановок размерностью $n \times n$ можно перемножить за время $O(n)$, положив $P_1P_2[i] = P_1[P_2[i]]$. При таком представлении обращение матрицы перестановки также можно вычислить за время $O(n)$.

Теорема 6.4. Пусть для каждого n можно умножить две матрицы размерностью $n \times n$ за такое время $M(n)$, что при всех m и некотором $\varepsilon > 0$ неравенство $M(2m) \geq 2^{2+\varepsilon} M(m)$.[5] Тогда найдется такая константа k, что алгоритм 6.1 тратит не более $kM(n)$ времени для любой невырожденной матрицы.

Доказательство. Применим алгоритм 6.1 к матрице размерностью $n \times n$. Пусть $T(m)$ — время, требуемое для выполнения процедуры FACTOR(A, m, p), где A — матрица размерностью $m \times p$, $m \leq p \leq n$. В силу строк 1–4 процедуры FACTOR $T(1) = bn$ для некоторой константы b. Рекурсивные вызовы в строках 6 и 11 занимают $T(m/2)$ времени каждый. В каждой из строк 7 и 13 вычисляется матрица, обратная к матрице перестановки (что занимает $O(n)$ времени), и произвольная матрица умножается на матрицу перестановки. Это умножение просто переставляет столбцы первой матрицы. Представляя матрицу перестановки в виде массива P, легко видеть, что $P[i]$-й столбец первой матрицы становится i-м столбцом произведения. Таким образом, произведение матриц можно найти за время $O(mn)$, а значит, строки 7 и 13 выполняются за время $O(mn)$.

Строка 9 занимает $O(M(m/2))$ времени на вычисление матрицы E^{-1} (в силу теоремы 6.2) и такое же время требуется для вычисления произведения FE^{-1}. Поскольку матрица U_1 имеет не более $m/2$ строк и не более n столбцов, то произведение $(FE^{-1})U_1$ можно вычислить за время

$$O\left(\frac{n}{m} M\left(\frac{m}{2}\right)\right).$$

Заметим, что n делится на m, поскольку m и n являются степенями числа 2 и $m \leq n$. Легко видеть, что остальные шаги занимают $O(mn)$ времени в худшем случае. Таким образом, получаем рекуррентные соотношения

---

[5] Говоря неформально, значение $M(n)$ должно лежать в диапазоне от $n^{2+\varepsilon}$ и n^3. Может оказаться, например, что $M(n) = kn^2 \log n$ для некоторой константе k и условие теоремы не удовлетворяется.

$$T(m) \le \begin{cases} 2T\left(\frac{m}{2}\right) + \frac{cn}{m} M\left(\frac{m}{2}\right) + dmn, & \text{если } m > 1, \\ bn, & \text{если } m = 1, \end{cases} \tag{6.5}$$

где b, c и d — константы.

В силу условия теоремы и равенства $M(1) = 1$ справедливо неравенство $M(m/2) \ge (m/2)^2$. Следовательно, в формуле (6.5) второе и третье слагаемые можно объединить. Для некоторой константы e

$$T(m) \le \begin{cases} 2T\left(\frac{m}{2}\right) + \frac{en}{m} M\left(\frac{m}{2}\right), & \text{если } m > 1, \\ bn, & \text{если } m = 1, \end{cases} \tag{6.6}$$

Из (6.6) следует

$$T(m) \le \frac{en}{4m}\left[4M\left(\frac{m}{2}\right) + 4^2 M\left(\frac{m}{2^2}\right) + \ldots + 4^{\log m} M(1)\right] + bnm \le$$

$$\le \frac{en}{4m} \sum_{i=1}^{\log m} 4^i M\left(\frac{m}{2^i}\right) + bnm.$$

Из условия теоремы следует, что $4^i M(m/2^i) \le (1/2^\varepsilon)^i M(m)$. Таким образом,

$$T(m) \le \frac{en}{4m} M(m) \sum_{i=1}^{\infty} \left(\frac{1}{2^\varepsilon}\right)^i + bnm.$$

Поскольку сумма в правой части сходится и $M(m) \ge m^2$, существует такая константа k, что $T(m) \le (kn/m)M(m)$. В алгоритме 6.1 $n = m$ и, значит, $T(n) \le kM(n)$. □

Следствие. LUP-разложение любой невырожденной матрицы размерностью $n \times n$ можно найти за $O(n^{2,81})$ шагов.

Доказательство. Следует из теорем 6. 1, 6.3 и 6.4. □

6.5. Приложения LUP-разложения матриц

В этом разделе мы покажем, как можно использовать LUP-разложение для вычисления обратных матриц и определителей, а также решения систем линейных уравнений. Мы увидим, что каждая из этих задач сводится к задаче умножения двух матриц и, следовательно, любое улучшение асимптотической временной сложности умножения матриц приводит к улучшению асимптотической временной сложности решения этих задач. И наоборот, умножение матриц сводится к обращению матриц, и, следовательно, задачи умножения и обращения матриц с вычислительной точки зрения являются эквивалентными.

Теорема 6.5. Пусть $\varepsilon > 0$ и $a \geq 1$. Пусть $M(n)$ — время, необходимое для умножения двух матриц, и $M(2m) \geq 2^{2+\varepsilon}M(m)$ для некоторого $\varepsilon > 0$. Тогда матрицу, обратную к данной невырожденной матрице, можно найти за время $O(M(n))$.

Доказательство. Пусть A — невырожденная матрица размерностью $n \times n$. По теоремам 6.3 и 6.4 можно найти LUP-разложение $A = LUP$ за время $O(M(n))$. Тогда $A^{-1} = P^{-1}U^{-1}L^{-1}$. Матрицу P^{-1} можно вычислить за $O(n)$ шагов. В силу теоремы 6.2 матрицы U^{-1} и L^{-1} существуют, и их можно вычислить за $O(M(n))$ шагов. Аналогично за $O(M(n))$ шагов можно вычислить произведение $P^{-1}U^{-1}L^{-1}$. □

Следствие. Обращение матрицы A размерностью $n \times n$ можно выполнить за время $O(n^{2,81})$.

Теорема 6.6. Если функция $M(n)$ та же, что и в теореме 6.5, и матрица A имеет размерность $n \times n$, то $\mathbf{det}(A)$ можно вычислить за $O(M(n))$ шагов.

Доказательство. Применим алгоритм 6.1, чтобы найти LUP-разложение матрицы A. Если он не работает, потому что в строке 2 не удалось найти ненулевой столбец или в строке 9 не существует матрица E^{-1}, то матрица A является вырожденной и $\mathbf{det}(A) = 0$. В противном случае существует разложение $A = LUP$. Тогда $\mathbf{det}(A) = \mathbf{det}(L)\mathbf{det}(U)\mathbf{det}(P)$. Определители $\mathbf{det}(L)$ и $\mathbf{det}(U)$ можно найти, вычислив произведения диагональных элементов матриц L и U. Поскольку L — нормированная нижняя треугольная матрица, то $\mathbf{det}(L) = 1$. Поскольку U — верхняя треугольная, то можно вычислить $\mathbf{det}(U)$ за $O(n)$ шагов. Так как P — матрица перестановки, то $\mathbf{det}(P) = \pm 1$, в зависимости от того, представляет ли P четную или нечетную перестановку. Вопрос о четности или нечетности перестановки можно выяснить, построив ее из вектора $(1, 2, ..., n)$. Потребуется не более $n - 1$ перестановок, и их число можно подсчитать во время выполнения. □

Следствие. Определитель матрицы размерностью $n \times n$ можно вычислить за $O(n^{2,81})$ шагов.

Пример 6.4. Вычислим определитель матрицы M из примера 6.3. Там мы нашли LUP-разложение

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Определители первого и второго сомножителей равны произведениям диагональных элементов, т.е. 1 и 24 соответственно. Осталось установить, какую перестановку — четную или нечетную — представляет последняя матрица P. Поскольку P представляет перестановку (4, 3, 2, 1) и ее можно получить двумя операциями обмена (1, 2, 3, 4) $\Rightarrow$ (4, 2, 3, 1) $\Rightarrow$ (4, 3, 2, 1), заключаем, что она является четной и $\det(P) = 1$. Таким образом, $\det(M) = 24$. □

Теорема 6.7. Пусть функция $M(n)$ та же, что и в теореме 6.5, A —невырожденная матрица размерностью $n \times n$ и b — вектор-столбец размерности n. Пусть $\mathbf{x} = [x_1, x_2, ..., x_n]^T$ — вектор-столбец неизвестных. Тогда систему линейных уравнений $A\mathbf{x} = \mathbf{b}$ можно решить за $O(M(n))$ шагов.

Доказательство. С помощью алгоритма 6.1 построим LUP-разложение $A = LUP$. Тогда система $LUP\mathbf{x} = \mathbf{b}$ решается в два шага. Сначала решаем систему $L\mathbf{y} = \mathbf{b}$ относительно $\mathbf{y}$, затем — систему $UP\mathbf{x} = \mathbf{y}$ относительно $\mathbf{x}$. Каждую из этих подзадач можно решить за $O(n^2)$ шагов, применив метод исключения, т.е. сначала найти значение y_1, подставить его вместо неизвестной переменной y_1, затем найти значение y_2 и т.д. LUP-разложение можно выполнить за $O(M(n))$шагов в силу теоремы 6.4, а систему $LUP\mathbf{x} = \mathbf{b}$ можно решить за $O(n^2)$ шагов. □

Следствие. Систему из n уравнений с n неизвестными можно решить за $O(n^{2,81})$ шагов.

В заключение покажем, что умножение и обращение матриц имеет одну и ту же вычислительную сложность.

Теорема 6.8. Пусть для умножения двух матриц размерностью $n \times n$ и обращения матрицы размерностью $n \times n$ требуется $M(n)$ и $I(n)$ времени. Предположим, что $8M(m) \geq M(2m) \geq 2^{2+\varepsilon}M(m)$ при некотором $\varepsilon > 0$ и аналогичные неравенства верны для $I(n)$. Тогда $M(n)$ и $I(n)$ совпадают с точностью до постоянного множителя.

Доказательство. Теорема 6.5 показывает, что $I(n) \leq c_1 M(n)$ для некоторой константы c_1. Чтобы доказать неравенство $M(n) \leq c_2 I(n)$ для некоторой константы c_2, рассмотрим произвольные матрицы A и B размерностью $n \times n$. Тогда

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}.$$

Следовательно, можно вычислить произведение AB, обращая матрицу размерностью $3n \times 3n$. Отсюда следует, что $M(n) \le I(3n) \le I(4n) \le 64I(n)$. □

6.6. Умножение булевых матриц

В разделе 5.9 мы рассмотрели задачу умножения двух булевых матриц с элементами из замкнутого полукольца $\{0, 1\}$, в котором сложение и умножение определяются таблицами

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

Мы показали, что умножение двух булевых матриц эквивалентно вычислению транзитивного замыкания графа. К сожалению, замкнутое полукольцо $\{0, 1\}$ не является кольцом, поэтому к умножению булевых матриц неприменимы ни алгоритм Штрассена для умножения матриц, ни другие результаты, изложенные ранее в этой главе.

Очевидно, что обычный алгоритм умножения матриц требует $O(n^3)$ шагов.[6] Тем не менее есть по крайней мере два способа умножения булевых матриц менее чем за $O(n^3)$ шагов. Первый из них асимптотически лучше, но второй, по-видимому, практичнее при умеренных значениях n. Опишем сейчас первый способ в следующей теореме.

Теорема 6.9. Произведение двух булевых матриц размерностью $n \times n$ можно вычислить за $O_A(n^{2,81})$ шагов.

Доказательство. Целые числа по модулю $n + 1$ образуют кольцо Z_{n+1}. Для умножения матриц A и B в кольце Z_{n+1} можно применить алгоритм Штрассена. Пусть C — произведение матриц A и B в кольце Z_{n+1}; D — их произведение как булевых матриц. Нетрудно показать, что если $D[i,j] = 0$, то $C[i,j] = 0$, и если $D[i,j] = 1$, то $1 \le C[i,j] \le n$. Следовательно, матрицу D легко получать из матрицы C. □

Следствие 1. Если для умножения двух k-разрядных двоичных чисел требуется $m(k)$ битовых операций, то две булевы матрицы можно перемножить за $O_B\left(n^{2,81} m \log n\right)$ шагов.

---

[6] Если почти все элементы матрицы-произведения равны 1, то две булевы матрицы можно перемножить в среднем за время $O(n^2)$, вычисляя каждую сумму $\sum_{k=1}^{n} a_{ik} b_{kj}$, только пока не будет найден член, равный 1. Как известно, сумма с такими слагаемыми равна 1. Если, например, каждый элемент a_{ik} и b_{kj} равен 1 с вероятностью p и эти события независимы, то среднее число слагаемых, которое необходимо просмотреть, не превышает $1/p^2$ независимо от n.

Доказательство. Поскольку все арифметические операции можно выполнить в кольце Z_{n+1}, для представления чисел достаточно $\lfloor \log n \rfloor + 1$ разрядов. Умножение двух таких чисел занимает не более $O_{\text{В}}(m \log n)$ времени, а сложение и вычитание — не более $O_{\text{В}}(\log n)$, что, очевидно, не превосходит $O_{\text{В}}(m \log n)$. □

В главе 7 мы обсудим алгоритм умножения чисел, для которого $m(k)$ имеет порядок $O_{\text{В}}(k \log k \log\log k)$. Учитывая этот результат, получаем такое следствие.

Следствие 2. Для умножения булевых матриц требуется не более $O_{\text{В}}(n^{2,81} \log n \log\log n \log\log\log n)$ шагов.

Второй метод, часто называемый алгоритмом четырех русских в честь его изобретателей,[7] в какой-то мере практичнее алгоритма из теоремы 6.9. Кроме того, он легко переносится на вычисления с битовыми векторами, чего нельзя сказать об алгоритме из теоремы 6.9.

Допустим, требуется перемножить две булевы матрицы A и B размерностью $n \times n$. Для простоты будем считать, что n делится на $\log n$. Можно разбить A на подматрицы размерностью $n \times \log n$, а B — на подматрицы размерностью $\log n \times n$, как показано на рис. 6.5. Можно записать

$$AB = \sum_{i=1}^{n/\log n} A_i B_i.$$

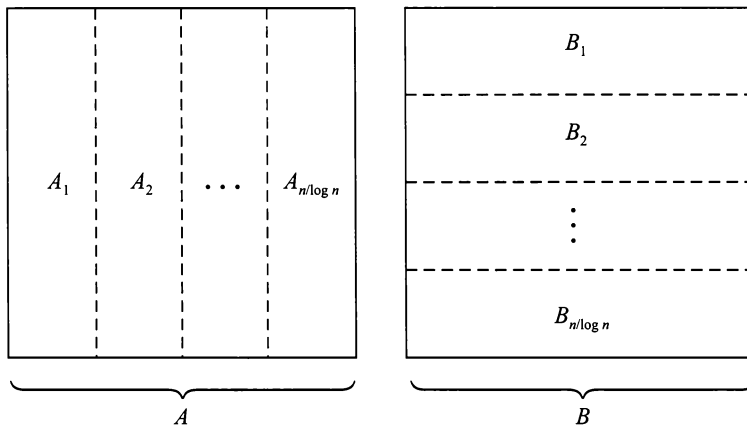


Рис. 6.5. Разбиение булевых матриц A и B

[7] См. библиографические замечания в конце главы. — *Примеч. ред.*

Заметим, что каждое произведение A_iB_j само является матрицей размерностью $n \times n$. Если мы сможем вычислить каждое произведение A_iB_j за $O(n^2)$ шагов, то сможем вычислить матрицу AB за $O\left(n^3/\log n\right)$ шагов, поскольку существует всего $n/\log n$ таких произведений.

Перейдем к вычислению произведений A_iB_j. Каждое такое произведение можно вычислить за $O\left(n^2 \log n\right)$ шагов. Для этого вычисляем произведение $\mathbf{a}_jB_i$ для каждой строки $\mathbf{a}_j$ матрицы A_i. Чтобы найти $\mathbf{a}_jB_i$, выберем в матрице B_i все строки с такими номерами k, что в $\mathbf{a}_j$ в k-м столбце стоит 1. Затем просуммируем их, обращаясь с ними как с n-мерными битовыми векторами.

Например, пусть

$$A_i = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \text{ и } B_i = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Тогда

$$C_i = A_iB_i = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Первая строка в матрице C_i точно совпадает с третьей строкой в матрице B_i, поскольку в первой строке в матрице A единица стоит только в третьем столбце. Вторая строка в матрице C_i равна сумме первой и третьей строк в матрице B_i, поскольку во второй строке в матрице A единицы стоят только в первом и третьем столбцах и т.д.

Вычисление $\mathbf{a}_j B_i$ для каждой строки $\mathbf{a}_j$ матрицы A_i занимает $O(n \log n)$ времени. Поскольку матрица A_i содержит n строк, общий объем работы при вычислении произведения $A_i B_i$ составляет $O(n^2 \log n)$.

Чтобы быстрее вычислить матрицу $A_i B_i$, заметим, что в каждой строке матрицы A_i содержится $\log n$ элементов, каждый из которых равен 0 или 1. Следовательно, все матрицы A_i могут содержать в общей сложности не более $2^{\log n} = n$ разных строк.

Таким образом, из строк матрицы B_i можно составить лишь n разных сумм. Можно заранее вычислить все возможные суммы строк матрицы B_i и вместо вычисления $\mathbf{a}_j B_i$ находить в ней ответ по $\mathbf{a}_j$.

Этот метод требует лишь $O(n^2)$ времени на вычисление $A_i B_i$, что объясняется следующими соображениями. Любое подмножество строк матрицы B_i или пусто, или состоит из одного элемента, или равно объединению одноэлементного множества и множества, меньшего исходного. Выбрав правильный порядок, можно вычислить все суммы строк, прибавляя одну строку матрицы B_i к уже найденной сумме строк. Так можно получить все n сумм строк матрицы B_i за $O(n^2)$ шагов. После вычисления сумм и расположения их в массиве можно выбирать соответствующую сумму для каждой из n строк матрицы A_i.

Алгоритм 6.2. Алгоритм четырех русских для умножения булевых матриц.

Вход. Две булевы матрицы A и B размерностью $n \times n$.

Выход. Произведение $C = AB$.

Метод. Положим $m = \lfloor \log n \rfloor$. Разобьем матрицу A на матрицы $A_1, A_2, \ldots, A_{\lceil n/m \rceil}$, где матрицы A_i, $1 \le i < \lceil n/m \rceil$, состоят из столбцов матрицы A с номерами от $m(i-1)+1$ до mi, а матрица $A_{\lceil n/m \rceil}$ — из оставшихся последних столбцов, к которым добавлены столбцы из нулей, если это нужно, чтобы в матрице $A_{\lceil n/m \rceil}$ было m столбцов. Разобьем матрицу B на матрицы $B_1, B_2, \ldots, B_{\lceil n/m \rceil}$, где матрицы B_i, $1 \le i < \lceil n/m \rceil$, состоят из строк матрицы B с номерами от $m(i-1)+1$ до mi, а матрица $B_{\lceil n/m \rceil}$ — из оставшихся последних строк, к которым добавлены строки из нулей, если это нужно, чтобы в матрице $B_{\lceil n/m \rceil}$ было m строк. Эта ситуация представлена на рис. 6.5. Вычисления приведены на рис. 6.6. NUM(v) — это целое число, представленное битовым вектором v, записанным в двоичной системе счисления с обратным порядком разрядов. Например, NUM([0, 1, 1]) = 6. □

```
     begin
1.       for i ← 1 until ⌈n/m⌉ do
             begin
                 comment сумма строк b_1^(i), ..., b_m^(i) матрицы B_i;
2.               ROWSUM[0] ← [0,0,...,0] (n нулей);
3.               for j ← 1 until 2^m - 1 do
                     begin
4.                       пусть k такое, что 2^k < j < 2^(k+1) ;
5.                       ROWSUM[j] ← ROWSUM[j - 2^k] + b_(k+1)^(i) [8]
                     end;
6.               пусть C_i — матрица, в которой j-я строка равна
                     ROWSUM[NUM(a_j)], где a_j — j-я строка матрицы A_i,
                     1 ≤ j ≤ n.
             end;
7.       пусть C равно Σ_{i=1}^{⌈n/m⌉} C_i
     end
```

Рис. 6.6. Алгоритм четырех русских

Теорема 6.10. Алгоритм 6.2 вычисляет матрицу $C = AB$ за $O\left(n^3/\log n\right)$ шагов.

Доказательство. Индукция по j показывает, что в строках 2–5 ROWSUM[j] становится равной побитовой булевой сумме таких строк $\mathbf{b}_k$ матрицы B_i, что в двоичном представлении числа j на k-м месте справа стоит 1. Отсюда вытекает, что в строке 6 $C_i = A_i B_i$ и, значит, в строке 7 $C = AB$.

Для подсчета временной сложности алгоритма сначала рассмотрим цикл, описанный строками 3–5. Оператор присваивания в строке 5, очевидно, выполняется за $O(n)$ шагов. Вычисление значения k в строке 4 занимает время $O(m)$, меньшее, чем $O(n)$, так что все тело цикла (строки 4–5) занимает время $O(n)$. Цикл повторяется $2^m - 1$ раз, поэтому его сложность равна $O(n2^m)$. Поскольку $m \le \log n$, цикл в строках 3–5 занимает время $O(n^2)$.

В строке 6 вычисление NUM($\mathbf{a}_j$) имеет сложность $O(m)$, а копирование вектора ROWSUM[NUM($\mathbf{a}_j$)] имеет сложность $O(n)$, так что строка 6 выполняется за $O(n^2)$ шагов. Поскольку $\lceil n/m \rceil \le 2n/\log n$, цикл в строках 1–6, который повторяется $\lceil n/m \rceil$ раз, занимает время $O\left(n^3/\log n\right)$. Аналогично в строке 7 требуется найти не более $2n/\log n$ сумм матриц размерностью $n \times n$, что дает сложность $O\left(n^3/\log n\right)$. Таким образом, весь алгоритм требует выполнения $O\left(n^3/\log n\right)$ шагов. □

[8] Здесь, конечно, подразумевается побитовая булева сумма.

Вероятно, интереснее то, что алгоритм 6.2 можно реализовать за $O_{BV}(n^2/\log n)$ вычислений с битовыми векторами, если в нашем распоряжении есть логические и арифметические операции над битовыми строками.

Теорема 6.11. Алгоритм 6.2 можно реализовать за $O_{BV}(n^2/\log n)$ операций с битовыми векторами.

Доказательство. Для того чтобы определить, когда увеличивать число k, используем счетчик. Вначале счетчик равен 1, а $k = 0$. Каждый раз, когда j увеличивается, счетчик уменьшается на 1, если его значение отличалось от 1, в противном случае значение счетчика полагается равным новому значению j, а число k увеличивается на 1.

Присваивания в строках 2 и 5 на рис. 6.6 занимают постоянное время. Следовательно, цикл в строках 3–5 имеет сложность $O_{BV}(n)$. Поскольку в модели RAM битовый вектор представляется целым числом,[9] на вычисление $\texttt{NUM}(\mathbf{a}_i)$ в строке 6 время не затрачивается. Следовательно, каждую строку матрицы C_i можно найти за фиксированное число операций над битовыми векторами и строка 6 имеет сложность $O_{BV}(n)$. Таким образом, цикл в строках 1–6 занимает время $O_{BV}(n^2/\log n)$ и такое же время требуется для выполнения строки 7.□

Упражнения

6.1. Покажите, что целые числа по модулю n образуют кольцо. Иначе говоря, Z_n — это кольцо $(\{0, 1, ..., n-1\}, +, \cdot, 0, 1)$, где $a + b$ и $a \cdot b$ — обычные сложение и умножение по модулю n.

6.2. Покажите, что матрицы размерностью $n \times n$ с элементами из некоторого кольца R образуют кольцо.

6.3. Приведите пример, показывающий, что произведение матриц некоммутативно, даже если их элементы берутся из кольца с коммутативным умножением.

6.4. Примените алгоритм Штрассена для вычисления произведения

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

---

[9] Можно не обращать внимание на то, что $\texttt{NUM}(\mathbf{a}_i)$ соответствует не самому вектору $\mathbf{a}_i$, а перестановке его элементов в обратном порядке, если в качестве j-й строки матрицы B взять j-ю строку снизу, а не сверху, как мы делали раньше.

6.5. Альтернативный вариант алгоритма Штрассена использует для вычисления произведения двух матриц размерностью 2×2 равенства

$$
\begin{aligned}
s_1 &= a_{21} + a_{22}, & m_1 &= s_2 s_6, & t_1 &= m_1 + m_2,\\
s_2 &= s_1 - a_{11}, & m_2 &= a_{11} b_{11}, & t_2 &= t_1 + m_4,\\
s_3 &= a_{11} - a_{21}, & m_3 &= a_{12} b_{21},\\
s_4 &= a_{12} - s_2, & m_4 &= s_3 s_7,\\
s_5 &= b_{12} - b_{11}, & m_5 &= s_1 s_5,\\
s_6 &= b_{22} - s_5, & m_6 &= s_4 b_{22},\\
s_7 &= b_{22} - b_{12}, & m_7 &= a_{22} s_8,\\
s_8 &= s_6 - b_{21}.
\end{aligned}
$$

Элементы произведения получаются следующим образом:

$$
\begin{aligned}
c_{11} &= m_2 + m_3; & c_{21} &= t_2 - m_7;\\
c_{12} &= t_1 + m_5 + m_6; & c_{22} &= t_2 + m_5.
\end{aligned}
$$

Покажите, что эти элементы равны соответствующим элементам из (6.1). Заметьте, что алгоритм состоит только из 7 умножений и 15 сложений.

6.6. Докажите следующие соотношения для матриц A, B и C размерностью $n \times n$.

а) Если $AB = I$ и $AC = I$, то $B = C$.

б) $A^{-1}A = I$.

в) $(AB)^{-1} = B^{-1}A^{-1}$.

г) $(A^{-1})^{-1} = A$.

д) $\mathbf{det}(AB) = \mathbf{det}(A)\mathbf{det}(B)$.

6.7. Теорема 6.2 утверждает, что обратную матрицу к невырожденной верхней треугольной матрице можно вычислять за не более чем $cn^{2,81}$ арифметических операций, где c — константа. Найдите эту константу c в предположении, что матрицы умножаются по алгоритму Штрассена и n — степень числа 2.

6.8. Представим матрицу перестановки в виде массива P, для которого $P[i] = j$ тогда и только тогда, когда в i-м столбце на j-й строке стоит 1. Пусть P_1 и P_2 — такие представления матриц перестановки размерностью $n \times n$.

а) Докажите, что $P_1P_2[i] = P_1[P_2[i]]$.

б) Постройте алгоритм, вычисляющий P_1^{-1} за время $O(n)$.

в) Измените представление так, чтобы $P[i] = j$ тогда и только тогда, когда на i-й строке в j-м столбце стоит 1. Напишите правильную формулу для P_1P_2 и постройте алгоритм для вычисления P_1^{-1}.

6.9. Примените алгоритм 6.1, чтобы найти LUP-разложение матрицы

$$M = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 3 & 0 \\ 1 & -1 & 0 & 1 \\ 2 & 0 & -1 & 3 \end{bmatrix}.$$

6.10. Мы показали, что нахождение LUP-разложения, вычисление обратной матрицы, вычисление определителя и решение системы линейных уравнений имеют сложность $O_A(n^{2.81})$. Найдите наилучшие постоянные множители для каждой из этих задач в предположении, что матрицы умножаются по алгоритму Штрассена, n — степень числа 2 и применяются алгоритм 6.1 и теоремы 6.4–6.7.

6.11. Для матрицы M из упражнения 6.9 найдите (*а*) обратную матрицу и (*б*) определитель, используя методы, описанные в этой главе.

6.12. С помощью LUP-разложения решите систему

$$\begin{aligned} x_3 + 2x_4 &= 7 \\ 3x_3 &= 9 \\ x_1 - x_2 + x_4 &= 3 \\ 2x_1 - x_3 + 3x_4 &= 10 \end{aligned}$$

6.13. Покажите, что каждая перестановка может быть четной или нечетной, но не одновременно.

6.14. Вычислите произведение булевых матриц, используя (*а*) метод теоремы 6.9 и (*б*) алгоритм четырех русских.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

6.15. Завершите доказательство теоремы 6.3, показав корректность взаимосвязей, изображенных на рис. 6.2 и 6.3.

****6.16.** Рассмотрим поле[10] F_2 целых чисел по модулю 2. Найдите алгоритм умножения матриц размерностью $n \times n$ над полем F_2 с асимптотической слож-

[10]Определение поля см. в разделе 12.1.

ностью не более $n^{2,81}/(\log n)^{0,4}$. (*Подсказка*: разбейте матрицы на блоки размера $\sqrt{\log n} \times \sqrt{\log n}$.)

6.17. Оцените значение n, начиная с которого $n^{2,81}$ меньше $n^3/\log n$.

***6.18.** Пусть $L(n)$ — время умножения двух нижних треугольных матриц размерностью $n \times n$; $T(n)$ — время умножения двух произвольных матриц. Докажите, что найдется такая константа c, что $T(n) \leq cL(n)$.

6.19. Докажите, что матрица, обратная к верхней (нижней) треугольной, является верхней (нижней) треугольной.

***6.20.** Пусть $I(n)$ — число шагов, необходимое для обращения матрицы размерностью $n \times n$, а $U(n)$ — для обращения верхней треугольной матрицы. Докажите, что найдется такая константа c, что $I(n) \leq cU(n)$ для всех n.

****6.21.** Чтобы вычислить произведение матриц $C = AB$, можно было бы определить сначала матрицу $D = (PAQ)(Q^{-1}BR)$, а затем $C = P^{-1}DR^{-1}$. Если P, Q и R — специальные матрицы, например матрицы перестановок, то вычисление произведений PAQ, $Q^{-1}BR$ и $P^{-1}DR^{-1}$ не требует умножения элементов кольца. Воспользуйтесь этой идеей, чтобы найти другой метод перемножения матриц 2×2 с помощью семи умножений элементов кольца.

6.22. Докажите, что если LU-разложение невырожденной матрицы A существует, то оно является единственным. (*Подсказка*: пусть $A = L_1U_1 = L_2U_2$; покажите, что $L_2^{-1}L_1 = U_2U_1^{-1} = I$.)

6.23. Докажите, что если матрица A и каждая ее главная подматрица являются невырожденными, то A имеет LU-разложение.

6.24. Может ли вырожденная матрица обладать LUP-разложением?

****6.25.** Пусть A — матрица размерностью $n \times n$ с действительными элементами. Матрица A называется *положительно определенной*, если для каждого ненулевого вектора-столбца $\mathbf{x}$ выполнено неравенство $\mathbf{x}^T A\mathbf{x} > 0$.

а) Покажите, что лемму 6.5 можно применить для обращения произвольной невырожденной симметричной положительно определенной матрицы.

б) Покажите, что если матрица A является невырожденной, то матрица AA^T является положительно определенной и симметричной.

в) Используя (*а*) и (*б*), постройте алгоритм сложности $O(M(n))$ для обращения любой невырожденной матрицы с действительными элементами.

г) Будет ли ваш алгоритм из пункта (*в*) работать в случае поля целых чисел по модулю 2?

***6.26.** Матрица A размерностью $n \times n$ называется *тёплицевой*, если

$$A[i, j] = A[i - 1, j - 1], 2 \le i, j \le n.$$

а) Найдите представление тёплицевых матриц, при котором сумму двух тёплицевых матриц можно найти за $O(n)$ шагов.

б) С помощью метода "разделяй и властвуй" постройте алгоритм умножения тёплицевой матрицы размерностью $n \times n$ на вектор-столбец. Сколько арифметических операций он требует? (*Подсказка*: с помощью метода "разделяй и властвуй" можно получить алгоритм сложности $O(n^{1,59})$.) В главе 7 будет описан метод, с помощью которого этот результат можно улучшить.

в) Постройте асимптотически эффективный алгоритм умножения двух тёплицевых матриц размерностью $n \times n$. Сколько арифметических операций он требует? Заметьте, что произведение тёплицевых матриц может не быть тёплицевой матрицей.

Проблемы для исследования

6.27. Естественно попытаться непосредственно улучшить метод Штрассена. Hopcroft and Kerr [1971] показали, что для вычисления произведения матриц размерностью 2×2 над произвольным кольцом необходимы семь умножений. Однако рекурсивный алгоритм может быть основан на умножении матриц какого-нибудь другого небольшого размера. Например, можно было бы улучшить порядок алгоритма Штрассена, если бы можно было перемножать матрицы размерностью 3×3 с помощью 21 умножений или матрицы размерностью 4×4 с помощью 48 умножений.

6.28. Можно ли находить кратчайшие пути меньше, чем за $O(n^3)$ шагов? Алгоритм Штрассена неприменим к замкнутым полукольцам, состоящим из неотрицательных вещественных чисел и $+\infty$, но, возможно, удастся свести операции в этом замкнутом полукольце к операциям в некотором кольце, как это было сделано для булевых матриц.

Библиографические замечания

Алгоритм Штрассена заимствован из работы Strassen [1969]. Winograd [1973] уменьшил количество необходимых сложений до 15, что улучшило постоянный множитель, но не порядок сложности (см. упражнение 6.5).

Strassen [1969] также описал методы сложности $O(n^{2,81})$ для обращения матриц, вычисления определителей и решения систем линейных уравнений в предполо-

жении, что каждая матрица, встречающаяся в процессе вычислений, является невырожденной. Bunch and Hopcroft [1974] показали, что LUP-разложение можно выполнить за $O(n^{2,81})$ шагов при единственном предположении, что исходная матрица является невырожденной. A. Schönhage независимо показал, что обращение любой невырожденной матрицы над упорядоченным полем можно выполнить за $O(n^{2,81})$ шагов (см. упражнение 6.25).

Результат о том, что умножение матриц не сложнее обращения матрицы, получен в работе Winograd [1973a] Алгоритм сложности $O(n^{2,81})$ для умножения булевых матриц построен в публикации Fischer and Meyer [1971], а алгоритм четырех русских — в работе Arlazarov, Dinic, Kronrod and Faradzev [1974].

Дополнительные сведения по теории матриц можно найти в Hohn [1958]. Алгебраические концепции, такие, как теория колец, изложены в книге MacLane and Birkhoff [1970]. Решение упражнения 6.13 приведено в работе Birkhoff and Bartee [1970], а упражнение 6.16 принадлежит J. Hopcroft.

ГЛАВА 7

Быстрое преобразование Фурье и его применения

Преобразование Фурье естественным образом возникает во многих областях науки и техники, поэтому эффективный алгоритм его вычисления представляет большой интерес. Широкое распространение преобразования Фурье будет показано с помощью демонстрации его применимости к построению эффективных алгоритмов. Во многих приложениях удобно свести поставленную задачу к другой, более простой. Такие примеры возникают при вычислении произведения двух полиномов. С вычислительной точки зрения целесообразно сначала применить линейное преобразование к векторам коэффициентов полиномов, затем к образам коэффициентов применить операцию, более простую, чем свертка, и, наконец, к результату применить обратное преобразование, чтобы получить искомое произведение. В данном случае подходящим линейным преобразованием является дискретное преобразование Фурье.

В этой главе мы изучим преобразование Фурье и обратное к нему и обсудим его роль в вычислении сверток и произведений разного вида. Мы опишем эффективный алгоритм, называемый быстрым преобразованием Фурье (Fast Fourier Transform — FFT). Он основан на методе вычисления полиномов с помощью деления, и в нем учитывается, что полиномы вычисляются для аргументов, равных корням из единицы.

Затем мы докажем теорему о свертке. Свертку будем интерпретировать как вычисление полиномов в корнях из единицы, умножение этих значений и последующую интерполяцию полиномов. С помощью быстрого преобразования Фурье мы разработаем эффективный алгоритм для свертки и применим его к символьному умножению полиномов и умножению целых чисел. Получающийся алгоритм умножения целых чисел, называемый алгоритмом Шёнхаге–Штрассена, является асимптотически самым быстрым из известных способов умножения двух целых чисел.

7.1. Дискретное преобразование Фурье и обратное к нему

Обычно преобразование Фурье определяется на комплексных числах. По причинам, которые станут ясными позже, мы будем определять преобразование Фурье над произвольным коммутативным кольцом $(R, +, \cdot, 0, 1)$.[1] Элемент ω из R, обладающий свойствами

1) $\omega \neq 1$,

2) $\omega^n = 1$,

3) $\sum_{j=0}^{n-1} \omega^{jp} = 0$ при $1 \le p \le n$,

называется *примитивным корнем n-й степени из единицы*. Элементы $\omega^0, \omega^1, ..., \omega^{n-1}$ называются *корнями n-й степени из единицы*.

Например, $e^{2\pi i/n}$, где $i = \sqrt{-1}$, является примитивным корнем n-й степени из единицы в кольце комплексных чисел.

Пусть $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$ — n-мерный вектор-столбец с элементами из R. Будем считать, что элемент n обладает в R обратным элементом относительно умножения[2] и что ω — примитивный корень n-й степени из единицы в этом кольце. Пусть A — такая матрица размерностью $n \times n$, что $A[i, j] = \omega^{ij}$ для $0 \le i, j < n$. *Дискретным преобразованием Фурье вектора* $\mathbf{a}$ называется вектор $F(\mathbf{a}) = A\mathbf{a}$, i-я компонента b_i которого равна $\sum_{k=0}^{n-1} a_k \omega^{ik}$, $0 \le i < n$. Матрица A является невырожденной, и, значит, существует обратная к ней матрица A^{-1}. Ее простой вид описывается в лемме 7.1.

Лемма 7.1. Пусть R — коммутативное кольцо, в котором существует примитивный корень n-й степени из единицы ω, причем n имеет обратный элемент в кольце R. Пусть A — такая матрица размерностью $n \times n$, что $A[i, j] = \omega^{ij}$, $0 \le i, j < n$. Тогда существует матрица A^{-1} и ее (i, j)-й элемент равен $(1/n)\omega^{-ij}$.

Доказательство. Пусть $\delta_{ij} = 1$, если $i = j$, и $\delta_{ij} = 0$ в противном случае. Достаточно показать, что если матрица A^{-1} определена, как выше, то $A \cdot A^{-1} = I_n$, т.е. (i, j)-й элемент матрицы $A \cdot A^{-1}$ удовлетворяет равенству

---

[1] Напомним, что коммутативным называется кольцо, в котором умножение и сложение являются коммутативными.

[2] Целые числа появляются в любом кольце, даже конечном. Под элементом n кольца подразумевается сумма 1 + 1 +...+ 1 (n раз), где 1 — мультипликативная единица. Заметим также, что примитивный корень ω обладает обратным элементом, поскольку $\omega\omega^{-1} = 1$, поэтому можно говорить об отрицательных степенях ω.

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{ik}\omega^{-kj}=\delta_{ij} \text{ для } 0\le i,j<n. \tag{7.1}$$

Если $i=j$, то левая часть равенства (7.1) принимает вид

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{0}=1.$$

Пусть $i\neq j$ и $q=i-j$. Тогда левая часть в (7.1) равна

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{qk},\ -n<q<n,\ q\neq 0.$$

Если $q>0$, то

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{qk}=0,$$

поскольку ω — примитивный корень n-й степени из единицы. Если $q<0$, то, умножив левую часть на $\omega^{-q(n-1)}$, изменив порядок слагаемых и подставив $-q$ вместо q, получим

$$\frac{1}{n}\sum_{k=0}^{n-1}\omega^{qk},\ 0<q<n.$$

Эта сумма тоже равна нулю, поскольку ω — примитивный корень n-й степени из единицы. Отсюда непосредственно следует (7.1). □

Обратным дискретным преобразованием Фурье вектора **a** называется вектор $F^{-1}(\mathbf{a})=A^{-1}\mathbf{a}$, i-й компонент которого, $0<i<n$, равен

$$\frac{1}{n}\sum_{k=0}^{n-1}a_k\omega^{-ik}.$$

Очевидно, что в результате применения обратного преобразования к преобразованию вектора **a** получается сам вектор **a**, т.е. $F^{-1}F(\mathbf{a})=\mathbf{a}$.

Преобразование Фурье тесно связано с вычислением полиномов и их интерполяцией. Рассмотрим полином $(n-1)$-й степени

$$p(x)=\sum_{i=0}^{n-1}a_i x^i.$$

Его можно однозначно представить двумя способами: списком его коэффициентов $a_0, a_1, \ldots, a_{n-1}$ и списком его значений в n различных точках $x_0, x_1, \ldots, x_{n-1}$. Процесс нахождения коэффициентов полинома по его значениям в точках $x_0, x_1, \ldots, x_{n-1}$ называется *интерполяцией*.

Вычисление преобразования Фурье вектора $[a_0, a_1, \ldots, a_{n-1}]^T$ эквивалентно переходу от представления полинома $\sum_{i=0}^{n-1}a_i x^i$ в виде списка его коэффициентов к его

представлению в виде списка значений в точках ω^0, ω^1, ..., ω^{n-1}. Точно так же вычисление обратного преобразования Фурье эквивалентно интерполяции полинома по его значениям в корнях n-й степени из единицы.

Можно было бы определить преобразование, которое вычисляло бы значения полинома на множестве точек, отличных от корней из единицы. Например, можно было бы использовать целые числа 1, 2, ..., n. Однако мы увидим, что, выбор степеней ω значительно упрощает вычисление значений и интерполяцию. В главе 8 преобразование Фурье применяется для вычисления значений и интерполяции полиномов в произвольных точках.

Одно из основных приложений преобразования Фурье — вычисление свертки двух векторов. Рассмотрим два вектора-столбца

$$\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T \text{ и } \mathbf{b} = [b_0, b_1, ..., b_{n-1}]^T.$$

Их *сверткой* $\mathbf{a} \circledast \mathbf{b}$ называется такой вектор $\mathbf{c} = [c_0, c_1, ..., c_{2n-1}]^T$, где $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}$. (Если $k < 0$ или $k \geq n$, то положим $a_k = b_k = 0$.) Таким образом,

$$c_0 = a_0 b_0;\ c_1 = a_0 b_1 + a_1 b_0;\ c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0 \text{ и т.д.}$$

Заметим, что $c_{2n-1} = 0$; этот компонент включен только для симметрии.

Чтобы обосновать полезность свертки, снова обратимся к представлению полинома его коэффициентами. Произведение двух полиномов степени $n - 1$

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \text{ и } q(x) = \sum_{j=0}^{n-1} b_j x^j$$

является полиномом степени $2n - 2$

$$p(x)q(x) = \sum_{i=0}^{2n-2} \left[\sum_{j=0}^{i} a_j b_{i-j} \right] x^i.$$

Заметим, что коэффициенты произведения равны компонентам свертки векторов $\mathbf{a} = [a_0, a_1, ..., a_n]^T$ и $\mathbf{b} = [b_0, b_1, ..., b_n]^T$, составленных из коэффициентов исходных полиномов, если пренебречь коэффициентом c_{2n-1}, равным нулю.

Если два полинома степени $n - 1$ представлены своими коэффициентами, то, чтобы вычислить коэффициенты их произведения, можно вычислить свертку векторов их коэффициентов. С другой стороны, если $p(x)$ и $q(x)$ представлены своими значениями в корнях n-й степени из единицы, то, чтобы вычислить представление их произведения, можно просто перемножить пары значений этих полиномов в соответствующих корнях. Отсюда следует, что свертка двух векторов $\mathbf{a}$ и $\mathbf{b}$ представляет собой обратное преобразование покомпонентного произведения их образов, т.е. $\mathbf{a} \circledast \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$. Иначе говоря, свертку векторов можно вычислить, выполнив их преобразования Фурье, вычислив покомпонентное произведение и затем выполнив обратное преобразование. Единственная трудность

состоит в том, что произведение двух полиномов степени $n-1$, вообще говоря, является полиномом степени $2n-2$ и для его представления требуются его значения в $2n-2$ различных точках. В теореме 7.1 показано, как преодолеть эту трудность. Можно рассматривать $p(x)$ и $q(x)$ как полиномы степени $2n-1$, у которых коэффициенты при n старших степенях x равны нулю (т.е. полиномы степени $n-1$ можно интерпретировать как полиномы степени $2n-1$).

Теорема 7.1. (Теорема о свертке.) Пусть

$$\mathbf{a} = [a_0, a_1, ..., a_{n-1}, 0, ..., 0]^T$$

и

$$\mathbf{b} = [b_0, b_1, ..., b_{n-1}, 0, ..., 0]^T$$

являются векторами-столбцами размерности $2n$, а

$$F(\mathbf{a}) = [a'_0, a'_1, ..., a'_{2n-1}]^T$$

и

$$F(\mathbf{b}) = [b_0, b'_1, ..., b'_{2n-1}]^T$$

являются их преобразованиями Фурье. Тогда $\mathbf{a} \circledast \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$.

Доказательство. Поскольку $a_i = b_i = 0$ для $n \le i < 2n$, то для $0 \le l < 2n$

$$a'_l = \sum_{j=0}^{n-1} a_j \omega^{lj} \quad \text{и} \quad b'_l = \sum_{k=0}^{n-1} b_k \omega^{lk}.$$

Следовательно,

$$a'_l b'_l = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)}. \tag{7.2}$$

Пусть $\mathbf{a} \circledast \mathbf{b} = [c_0, c_1, ..., c_{2n-1}]^T$ и $F(\mathbf{a} \circledast \mathbf{b}) = [c'_0, c'_1, ..., c'_{2n-1}]^T$. Поскольку $c_p = \sum_{j=0}^{2n-1} a_j b_{p-j}$, получаем

$$c'_i = \sum_{p=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{p-j} \omega^{lp}. \tag{7.3}$$

Меняя порядок суммирования в (7.3) и подставляя k вместо $p-j$, получаем

$$c'_l = \sum_{j=0}^{2n-1} \sum_{k=-j}^{2n-1-j} a_j b_k \omega^{l(j+k)}. \tag{7.4}$$

Поскольку $b_k = 0$ при $k < 0$, нижний предел внутреннего суммирования можно поднять до $k = 0$. Аналогично, поскольку $a_j = 0$ для $j \ge n$, верхний предел внешнего суммирования также можно понизить до $n-1$. Верхний предел внутреннего суммирования не меньше n, независимо от значения j. Таким образом, верхний предел можно заменить на $n-1$, поскольку $b_k = 0$ при $k \ge n$. После этих

изменений формула (7.4) превратится в (7.2). Отсюда следует, что $c'_l = a'_l b'_l$. Итак, $F(\mathbf{a} \circledast \mathbf{b}) = F(\mathbf{a}) \cdot F(\mathbf{b})$, откуда вытекает, что $\mathbf{a} \circledast \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$. □

Свертка двух n-мерных векторов является $2n$-мерным вектором. Это требует, чтобы в теореме о свертке векторы $\mathbf{a}$ и $\mathbf{b}$ были дополнены n нулями. Чтобы этого избежать, введем понятие обернутой свертки.

Определение. Пусть $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$ и $\mathbf{b} = [b_0, b_1, ..., b_{n-1}]^T$ — два n-мерных вектора. *Положительно обернутой сверткой* векторов $\mathbf{a}$ и $\mathbf{b}$ называется такой вектор $\mathbf{c} = [c_0, c_1, ..., c_{n-1}]^T$, что

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}.$$

Отрицательно обернутой сверткой векторов $\mathbf{a}$ и $\mathbf{b}$ называется такой вектор $\mathbf{d} = [d_0, d_1, ..., d_{n-1}]^T$, что

$$d_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}.$$

В разделе 7.5 мы используем обернутую свертку в алгоритме быстрого умножения целых чисел Шёнхаге–Штрассена. Пока отметим, что, вычислив значения двух полиномов степени $n - 1$ в корнях n-й степени из единицы и перемножив пары значений в соответствующих точках, мы получим n значений, по которым можно однозначно интерполировать полином степени $n - 1$. Вектор коэффициентов этого единственного полинома как раз и будет положительно обернутой сверткой векторов коэффициентов исходных полиномов.

Теорема 7.2. Пусть $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$ и $\mathbf{b} = [b_0, b_1, ..., b_{n-1}]^T$ — два n-мерных вектора, а ω — примитивный корень n-й степени из единицы. Пусть $\psi^2 = \omega$. Предположим, что элемент n имеет обратный элемент. Тогда справедливы следующие утверждения.

1. Положительно обернутая свертка векторов $\mathbf{a}$ и $\mathbf{b}$ равна $F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$.
2. Пусть $\mathbf{d} = [d_0, d_1, ..., d_{n-1}]^T$ — отрицательно обернутая свертка векторов $\mathbf{a}$ и $\mathbf{b}$, $\hat{\mathbf{a}} = \left[a_0, \psi a_1, ..., \psi^{n-1} a_{n-1}\right]^T$, $\hat{\mathbf{b}} = \left[b_0, \psi b_1, ..., \psi^{n-1} b_{n-1}\right]^T$ и $\hat{\mathbf{d}} = \left[d_0, \psi d_1, ..., \psi^{n-1} d_{n-1}\right]^T$. Тогда $\hat{\mathbf{d}} = F^{-1}\left(F(\hat{\mathbf{a}}) \cdot F(\hat{\mathbf{b}})\right)$.

Доказательство. Теорема доказывается аналогично теореме 7.1 с учетом равенства $\psi^n = -1$. Детали предоставляем доказать читателям в качестве упражнения. □

7.2. Алгоритм быстрого преобразования Фурье

Очевидно, что преобразование Фурье вектора **a** из R^n и обратное к нему можно вычислить за время $O_A(n^2)$, если предположить, что каждая арифметическая операция над элементами кольца R выполняется за один шаг. Однако если n — степень числа 2, то это можно сделать быстрее: существует алгоритм сложности $O_A(n\log n)$, вычисляющий преобразование Фурье или обратное к нему, и мы предполагаем, что эту оценку улучшить невозможно. Мы приведем только алгоритм прямого преобразования Фурье. Алгоритм обратного преобразования аналогичен, и читатели могут самостоятельно сформулировать его. Основная идея быстрого преобразования Фурье (FFT) по своей природе является алгебраической; мы находим сходство между частями тех n сумм, которые порождаются умножением $A\mathbf{a}$. В этом разделе мы считаем, что $n = 2^k$ для некоторого целого числа k.

Вспомним, что вычисление вектора $A\mathbf{a}$ эквивалентно вычислению $n - 1$ полинома $p(x)=\sum_{j=0}^{n-1} a_j x^j$ в точках x, равных ω^0, ω^1, ..., ω^{n-1}. Однако вычисление $p(x)$ в точке $x = a$ равносильно нахождению остатка от деления $p(x)$ на $x - a$. (Чтобы убедиться в этом, можно записать $p(x)$ в виде $(x - a)q(x) + c$, где c — константа. Тогда $p(a) = c$.) Таким образом, вычисление преобразования Фурье сводится к нахождению остатка от деления полинома $p(x)=\sum_{j=0}^{n-1} a_j x^j$ степени $n - 1$ на каждый из полиномов $x - \omega^0$, $x - \omega^1$, ..., $x - \omega^{n-1}$.

Обычное последовательное деление $p(x)$ на каждый из полиномов $x - \omega^i$ имеет временную сложность $O(n^2)$. Чтобы построить более быстрый алгоритм, перемножим попарно полиномы $x - \omega^i$, затем перемножим попарно получившиеся $n/2$ полиномов и т.д., пока не останется два полинома q_1 и q_2, каждый из которых равен произведению половины полиномов $x - \omega^i$. Затем поделим $p(x)$ на q_1 и q_2. Соответствующие остатки $r_1(x)$ и $r_2(x)$ имеют степень не более $n/2 - 1$ каждый. Для каждого корня ω^i, для которого полином q_1 делится без остатка на $x - \omega^i$, нахождение остатка от деления $p(x)$ на $x - \omega^i$ равносильно нахождению остатка от деления $r_1(x)$ на $x - \omega^i$. Аналогичное утверждение верно для каждого корня ω^i, для которого полином q_2 делится без остатка на $x - \omega^i$. Следовательно, вычисление остатков от деления $p(x)$ на каждый из полиномов $x - \omega^i$ равносильно вычислению остатков от деления $r_1(x)$ и $r_2(x)$ на каждый из $n/2$ подходящих полиномов $x - \omega^i$. Рекурсивное применение метода "разделяй и властвуй" гораздо эффективнее прямолинейного метода деления $p(x)$ на каждый полином $x - \omega^i$.

При перемножении полиномов $x - \omega^i$ можно ожидать, что произведения будут содержать результаты перекрестных умножений одночленов. Однако при подходящем упорядочении полиномов $x - \omega^i$ можно добиться, чтобы все произведения имели вид $x^j - \omega^i$, и это еще уменьшит время, затрачиваемое на умножение и деление полиномов. Изложим все эти идеи более подробно.

Пусть $c_0, c_1, ..., c_{n-1}$ — перестановка элементов $\omega^0, \omega^1, ..., \omega^{n-1}$, которую мы уточним позже. Определим полиномы q_{lm}, где $0 \le m \le k$ и l — целое число, кратное 2^m, $0 \le l \le 2^k - 1$:

$$q_{lm} = \prod_{j=l}^{l+2^m-1} (x - c_j).$$

Таким образом, $q_{0k} = (x - c_0)(x - c_1)...(x - c_{n-1})$, $q_{l0} = x - c_l$ и в общем случае

$$q_{lm} = q_{l,m-1} q_{l+2^{m-1},m-1}.$$

Существует 2^{k-m} полиномов со вторым нижним индексом m, и каждый полином $x - c_l$ делит один из них без остатка. Полиномы q_{lm} изображены на рис. 7.1. (См. также разделы 8.4 и 8.5.)

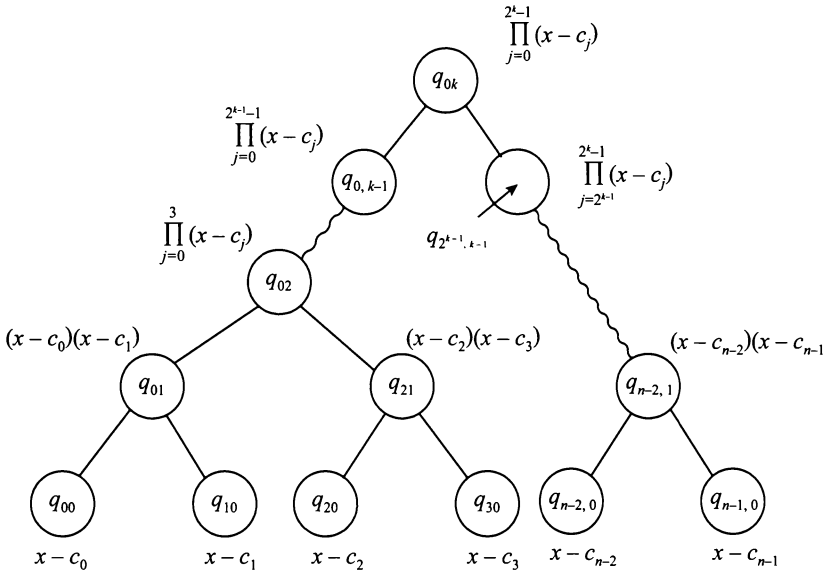


Рис. 7.1. Полиномы q_{lm}

Наша цель — вычислить остаток от деления $p(x)$ на $q_{l0}(x)$ для каждого l. Для этого вычислим остатки от деления $p(x)$ на $q_{lm}(x)$ для каждого q_{lm}, начиная с $m = k - 1$ и заканчивая $m = 0$.

Допустим, мы уже вычислили полиномы r_{lm} степени $2^m - 1$, являющиеся остатками от деления $p(x)/q_{lm}(x)$. (Предполагаем, что $r_{0k} = p(x)$.) Поскольку $q_{lm} = q'q''$, где $q' = q_{l,m-1}$ и $q'' = q_{l+2^{m-1},m-1}$, можно утверждать, что деление $p(x)/q'(x)$ дает тот же остаток, что и $r_{lm}/q'(x)$, и аналогичное утверждение справедливо для $q''(x)$. Для доказательства положим

$$p(x) = h_1(x)q'(x) + r_{l,m-1},$$

где степень полинома $r_{l,m-1}$ не превышает $2^{m-1}-1$. Поскольку

$$p(x) = h_2(x)q_{lm}(x) + r_{lm},$$

получаем

$$h_1(x)q'(x) + r_{l,m-1} = h_2(x)q_{lm}(x) + r_{lm}. \tag{7.5}$$

Разделим обе части равенства (7.5) на $q'(x)$. Поскольку полиномы $h_1(x)q'(x)$ и $h_2(x)q_{lm}(x)$ делятся без остатка, то остаток от деления r_{lm} на $q'(x)$ равен $r_{l,m-1}$.

Таким образом, можно получить остатки от деления $p(x)$ на $q'(x)$ и $p(x)$ на $q''(x)$, разделив на $q'(x)$ и $q''(x)$ полином r_{lm} степени $2^{k-m}-1$, а не полином $p(x)$ степени 2^k-1. Этот метод выполнения деления сам по себе дает экономию времени. Однако можно делать еще больше. Выбрав подходящий порядок элементов $c_0, c_1, \ldots, c_{n-1}$ для степеней ω, можно добиться, чтобы каждый полином q_{lm} имел вид $x^{2^m} - \omega^s$ при некотором s. Деление на такие полиномы выполняется особенно просто.

Лемма 7.2. Пусть $n = 2^k$ и ω — примитивный корень степени n из единицы. Пусть $[d_0 d_1 \ldots d_{k-1}]$ — двоичное представление целого числа j,[3] где $0 \le j \le 2^k$, а **rev**(j) — целое число с двоичным представлением $[d_{k-1} d_{k-2} \ldots d_0]$. Обозначим $c_j = \omega^{\mathbf{rev}(j)}$ и $q_{lm} = \prod_{j=l}^{l+2^{m-1}} \left(x - c_j\right)$. Тогда $q_{lm} = x^{2^m} - \omega^{\mathbf{rev}\left(l/2^m\right)}$.

Доказательство. Доказательство проводится индукцией по m. База, т.е. случай $m = 0$, является тривиальной, поскольку по определению $q_{l0} = x - c_l = x - \omega^{\mathbf{rev}(l)}$. Для проведения шага индукции заметим, что при $m > 0$

$$q_{lm} = q_{l,m-1} q_{l+2^{m-1},m-1} = \\ = \left(x^{2^{m-1}} - \omega^{\mathbf{rev}\left(l/2^{m-1}\right)}\right)\left(x^{2^{m-1}} - \omega^{\mathbf{rev}\left(l/2^{m-1}+1\right)}\right),$$

где $l/2^{m-1}$ — четное число от 0 до $2^k - 1$. Далее,

$$\omega^{\mathbf{rev}\left(l/2^{m-1}+1\right)} = \omega^{2^{k-1}+\mathbf{rev}\left(l/2^{m-1}\right)} = -\left(\omega^{\mathbf{rev}\left(l/2^{m-1}\right)}\right),$$

поскольку $\omega^{2^{k-1}} = \omega^{n/2} = -1$. Следовательно,

$$q_{lm} = x^{2^m} - \omega^{2\mathbf{rev}\left(l/2^{m-1}\right)} = x^{2^m} - \omega^{\mathbf{rev}\left(l/2^m\right)},$$

так как $\mathbf{rev}(2t) = \frac{1}{2}\mathbf{rev}(t)$. □

---

[3] Т.е. $j = \sum_{i=0}^{k-1} d_{k-1-i} 2^i$.

Пример 7.1. Если $n=8$, то список c_0, c_1, ..., c_7 имеет вид $\omega^0, \omega^4, \omega^2, \omega^6, \omega^1, \omega^5, \omega^3, \omega^7$. Полиномы q_{lm} представлены на рис. 7.2.

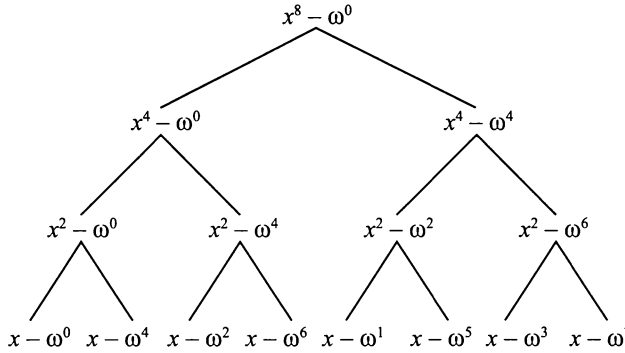


Рис. 7.2. Полиномы q_{lm} из леммы 7.2

Покажем, как использовать полиномы q_{lm} для вычисления остатков. Вначале рассчитываются остатки r_{02} и r_{42} от деления $p(x)$ на $x^4 - \omega^0$ и $x^4 - \omega^4$, где $p(x) = \sum_{j=0}^{7} a_j x^j$. Затем вычисляются остатки r_{01} и r_{21} от деления r_{02} на $x^2 - \omega^0$ и $x^2 - \omega^4$, а также остатки r_{41} и r_{61} от деления r_{42} на $x^2 - \omega^2$ и $x^2 - \omega^6$. Наконец, вычисляются r_{00}, r_{10}, r_{20}, ..., r_{70}, где r_{00} и r_{10} — остатки от деления r_{01} на $x - \omega^0$ и $x - \omega^4$, r_{20} и r_{30} — остатки от деления r_{21} на $x - \omega^2$ и $x - \omega^6$ и т.д.

Другие примеры применения этого подхода приведены в разделе 8.5. □

Доказав, что полиномы q_{lm} имеют вид $x^s - c$, покажем, что остаток от деления $p(x)$ на $x^s - c$ найти легко.

Лемма 7.3. Пусть

$$p(x) = \sum_{j=0}^{2t-1} a_j x^j$$

и c — константа. Тогда остаток от деления $p(x)/(x^t - c)$ равен

$$r(x) = \sum_{j=0}^{t-1} \left(a_j + c a_{j+t}\right) x^j.$$

Доказательство. Достаточно заметить, что $p(x)$ можно представить в виде

$$\left[\sum_{j=0}^{t-1} a_{j+t} x^j\right]\left(x^t - c\right) + r(x). \quad \square$$

Таким образом, вычисление остатка от деления произвольного полинома степени $2t - 1$ на $x^t - c$ можно выполнить за $O_A(t)$ шагов. Это меньше, чем в любом

известном алгоритме вычисления остатка от деления произвольного полинома степени $2t - 1$ на произвольный полином степени t.

Алгоритм 7.1. Быстрое преобразование Фурье

Вход. Вектор $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$, где $n = 2^k$ для некоторого целого числа k.

Выход. Вектор $F(\mathbf{a}) = [b_0, b_1, ..., b_{n-1}]^T$, где $b_i = \sum_{j=0}^{n-1} a_j \omega^{ji}$ для $0 \le i < n$.

Метод. См. рис. 7.3. □

```
      begin
1.        пусть r_{0k} = Σ_{j=0}^{n-1} a_j x^j;
          comment Полином представлен своими коэффициентами,
               так что в строке 1 не производится никаких
               вычислений. Остаток от деления полинома Σ_{j=0}^{n-1} a_j x^j на q_{1m}
               будет представлен полиномом r_{1m};
2.        for m ← k - 1 step -1 until 0 do
3.            for l ← 0 step 2^{m+1} until n - 1 do
                  begin
4.                    пусть r_{l,m+1} = Σ_{j=0}^{2^{m+1}-1} a_j x^j;
                      comment Вычисляем остатки меньшей степени,
                          используя коэффициенты полинома r_{l,m+1};
5.                    S ← rev(l/2^m);
6.                    r_{lm}(x) ← Σ_{j=0}^{2^m-1} (a_j + ω^s a_{j+2^m}) x^j;
7.                    r_{l+2^m,m}(x) ← Σ_{j=0}^{2^m-1} (a_j + ω^{s+n/2} a_{j+2^m}) x^j;
                  end;
8.            for l ← 0 until n - 1 do b_{rev(l)} ← r_{l0}
      end
```

Рис. 7.3. Алгоритм быстрого преобразования Фурье

Модификация алгоритма 7.1 для вычисления обратных преобразований состоит в замене ω на ω^{-1} (для этого в строках 6 и 7 знаки показателей степеней элемента ω меняются на противоположные). Кроме того, $b_{\mathbf{rev}(i)}$ делится на n в строке 8.

Пример 7.2. Пусть $n = 8$ и, следовательно, $k = 3$. При $m = 2$ цикл в строках 3–7 выполняется только для $l = 0$. В строке 4 r_{03} равно $\sum_{j=0}^{7} a_j x^j$. В строке 5 $s = 0$, в строках 6 и 7

$$r_{02} = (a_3 + a_7)x^3 + (a_2 + a_6)x^2 + (a_1 + a_5)x + (a_0 + a_4)$$

и

$$r_{42} = (a_3 + \omega^4 a_7)x^3 + (a_2 + \omega^4 a_6)x^2 + (a_1 + \omega^4 a_5)x + (a_0 + \omega^4 a_4).$$

Если $m = 1$, то l принимает значения 0 и 4. Когда $l = 0$, s = 0 в строке 5, в строках 6 и 7

$$r_{01} = (a_1 + a_3 + a_5 + a_7)x + (a_0 + a_2 + a_4 + a_6),$$

и

$$r_{21} = (a_1 + \omega^4 a_3 + a_5 + \omega^4 a_7)x + (a_0 + \omega^4 a_2 + a_4 + \omega^4 a_6).$$

Если $l = 4$, то $s = 2$. В силу строк 6 и 7 и формулы для r_{42}, приведенной выше,

$$r_{41} = (a_1 + \omega^2 a_3 + \omega^4 a_5 + \omega^6 a_7)x + (a_0 + \omega^2 a_2 + \omega^4 a_4 + \omega^6 a_6),$$
$$r_{61} = (a_1 + \omega^6 a_3 + \omega^4 a_5 + \omega^2 a_7)x + (a_0 + \omega^6 a_2 + \omega^4 a_4 + \omega^2 a_6),$$

Наконец, если $m = 0$, то l принимает значения 0, 2, 4 и 6. При $l = 4$, например, $s = 1$, поэтому

$$r_{40} = a_0 + \omega a_1 + \omega^2 a_2 + \ldots + \omega^7 a_7.$$

К моменту входа в цикл **for** в строке 8 полином r_{l0} всегда будет иметь степень 0, т.е. будет константой. Например, **rev**(l) = 1 при $l = 4$ и r_{40} станет значением b_1. Эта формула для b_1 согласуется с его определением. □

Покажем, что алгоритм 7.1 корректен.

Теорема 7.3. Алгоритм 7.1 вычисляет дискретное преобразование Фурье.

Доказательство. В строке 6: $r_{lm} = r_{l,m+1}/q_{lm}$, а в строке 7: $r_{l+2^m,m} = r_{l,m+1}/q_{l+2^m,m}$. Поэтому с помощью лемм 7.2 и 7.3 нетрудно доказать индукцией по $k - m$, что r_{lm} — остаток от деления $\sum_{j=0}^{n-1} a_j x^j$ на q_{lm}. Тогда для $m = 0$ лемма 7.3 гарантирует, что цикл **for** в строке 8 присвоит всем b_i правильные значения (остатки, равные константам). □

Теорема 7.4. Временная сложность алгоритма 7.1 составляет $O_A(n \log n)$.

Доказательство. Каждое выполнение строк 6 и 7 занимает $O_A(2^m)$ шагов. При фиксированном m цикл в строках 3–7 повторяется $n/2^{m+1}$ раз, занимая в целом время $O_A(n)$, не зависящее от m. Внешний цикл, начинающийся в строке 2, выполняется $\log n$ раз, занимая в целом время $O_A(n \log n)$. Цикл в строке 8 не требует выполнения арифметических операций. □

В теореме 7.4 мы предполагали, что число n фиксировано. Поэтому степени элемента ω, а также значения s и **rev**(l), вычисляемые по l в строках 5 и 8, можно вычислять заранее и использовать в качестве констант в неветвящейся программе. Если же надо, чтобы n было параметром, можно вычислить степени элемента ω

и запомнить их в виде таблицы за $O(n)$ шагов работы машины RAM. Более того, хотя на вычисление s и **rev**(l) в строках 5 и 8 затрачивается $O(\log n)$ шагов, всего производится не более $3n$ таких вычислений, так что весь алгоритм при реализации на машине RAM имеет временную сложность $O(n \log n)$.

Следствие 1. Свертку $\mathbf{a} \circledast \mathbf{b}$, где $\mathbf{a}$ и $\mathbf{b}$ — векторы размерности n, можно вычислить за $O_A(n \log n)$ шагов.

Доказательство. Следует из теорем 7.1, 7.3 и 7.4. □

Следствие 2. Положительно и отрицательно обернутые свертки векторов $\mathbf{a}$ и $\mathbf{b}$ можно вычислить за $O_A(n \log n)$ шагов.

Алгоритм 7.1 для вычисления FFT изложен для того, чтобы пояснить интуитивные соображения, лежащие в основе его конструкции. На самом деле при вычислении FFT можно работать лишь с коэффициентами и тем самым упростить алгоритм. Это реализовано в алгоритме 7.2.

Алгоритм 7.2. Упрощенный алгоритм FFT.

Вход. Вектор $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$, где $n = 2^k$ для некоторого целого числа k.

Выход. Вектор $F(\mathbf{a}) = [b_0, b_1, ..., b_{n-1}]^T$, где $b_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ для $0 \le i < n$.

Метод. Применяем программу на рис. 7.4. Для облегчения понимания вводим временный массив S, в котором будут храниться результаты предыдущего шага. На практике эти вычисления можно осуществлять на том же месте. □

```
      begin
1.        for i ← 0 until 2^k - 1 do R[i] ← a_i;
2.        for l ← 0 until k - 1 do
              begin
3.                for i ← 0 until 2^k - 1 do S[i] ← R[i];
4.                for i ← 0 until 2^k - 1 do
                      begin
5.                        пусть [d_0 d_1 ... d_{k-1}] — двоичное представление
                              целого числа i;
6.                        R[[d_0 d_1 ... d_{k-1}]] ← S[[d_0 d_1 ... d_{l-1} 0 d_{l+1} ... d_{k-1}]] +
                          + ω^[d_l d_{l-1} ... d_0 0 ... 0] S[[d_0 ... d_{l-1} 1 d_{l+1} ..., d_{k-1}]]
                      end
              end;
7.        for i ← 0 until 2^k - 1 do
              b_[d_0 ... d_{k-1}] ← R[[d_{k-1} ... d_0]]
      end
```

Рис. 7.4. Алгоритм быстрого преобразования Фурье

Когда строка 3 выполняется первый раз, коэффициенты полинома $p(x) = \sum_{i=1}^{n-1} a_i x^i$ сохраняются в массиве S. Во время первого выполнения строки 6 полином $p(x)$ делится на $x^{n/2} - 1$ и $x^{n/2} - \omega^{n/2}$. Остатки имеют вид

$$\sum_{i=0}^{n/2-1} \left(a_i + a_{i+n/2}\right) x^i \quad \text{и} \quad \sum_{i=0}^{n/2-1} \left(a_i + \omega^{n/2} a_{i+n/2}\right) x^i.$$

Их коэффициенты хранятся в массиве S при втором выполнении строки 3. Коэффициенты первого остатка занимают первую половину массива S, а второго остатка — вторую.

При втором выполнении строки 6 каждый из этих двух полиномов-остатков делится на два полинома вида $x^{n/4} - \omega^s$. Это дает четыре остатка, каждый степени $n/4 - 1$. При третьем выполнении строки 3 их коэффициенты запоминаются в массиве S и т.д. Строка 7 реорганизует компоненты ответа так, чтобы они следовали в правильном порядке. Она необходима потому, что корни из единицы были переставлены, чтобы устранить члены, получающиеся в результате перекрестных умножений при вычислении произведений полиномов $x - \omega^i$. Этот процесс при $n = 8$ частично проиллюстрирован на рис. 7.5.

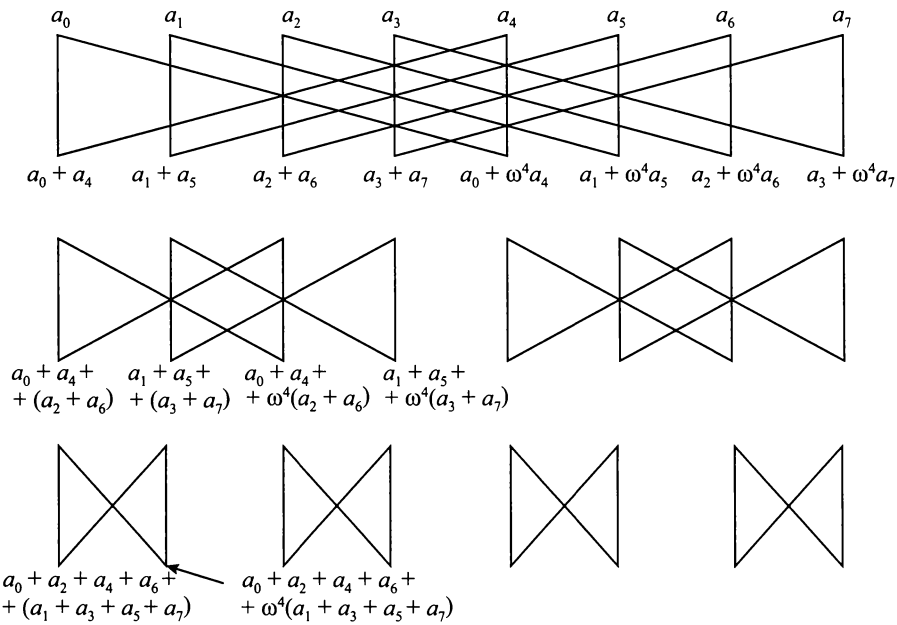


Рис. 7.5. Вычисления FFT по алгоритму 7.2.
Для экономии места коэффициенты полиномов-остатков не указаны

7.3. Быстрое преобразование Фурье с битовыми операциями

В приложениях, где преобразование Фурье используется для упрощения вычисления свертки, часто требуется точный результат. Если элементы принадлежат кольцу действительных чисел, их приходится аппроксимировать с конечной точностью, что порождает ошибки. Избежать этих ошибок можно, если производить вычисления в конечном поле.[4] Например, чтобы свернуть векторы $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$ и $\mathbf{b} = [b_0, b_1, ..., b_{n-1}]^T$, можно взять 2 в качестве корня пятой степени из единицы и вычислять по модулю 31. Тогда преобразование векторов **a** и **b**, попарные умножения и нахождение обратного преобразования дают точное значение $\mathbf{a} \circledast \mathbf{b}$ по модулю 31.[5] Часто бывает трудно выбрать подходящее конечное поле с подходящим корнем n-й степени из единицы. Вместо этого мы будем использовать кольцо R_m целых чисел по модулю m, где m будет таким, чтобы в R_m был примитивный корень n-й степени из единицы ω.[6] Сразу не очевидно, что по заданному n можно найти такие ω и m, что ω — примитивный корень n-й степени из единицы в кольце вычетов по модулю m. Кроме того, слишком большие m не походят, поскольку в таком случае вычисления по модулю m становятся громоздкими. К счастью, если n — степень числа 2, то подходящее число m существует всегда и равно примерно $2n$. В частности, мы покажем (теорема 7.5), что, когда n и $\omega > 1$ являются степенями числа 2, можно вычислять свертки в кольце вычетов по модулю $(\omega^{n/2} + 1)$ с помощью преобразования Фурье, покомпонентного умножения и обратного преобразования. Сначала установим два предварительных результата, сформулированных в виде лемм 7.4 и 7.5. В них мы будем предполагать, что $R = (S, +, \cdot, 0, 1)$ — коммутативное кольцо и $n = 2^k$, $k \geq 1$.

Лемма 7.4. Для всех $a \in S$

$$\sum_{i=0}^{n-1} a^i = \prod_{i=0}^{k-1} \left(1 + a^{2^i}\right).$$

Доказательство. Доказательство проводится индукцией по k. База, т.е. случай $k = 1$, является тривиальной. Теперь заметим, что

---

[4] Определение поля см. в разделе 12.1.

[5] Разумеется, компоненты ответа должны быть заключены в диапазоне от 0 до 30, поскольку иначе их невозможно будет восстановить. В общем случае, чтобы можно было восстановить ответ, необходимо выбирать модули достаточно большими.

[6] До сих пор число ω можно было считать комплексным числом $e^{2\pi i/n}$ и выполнять арифметические операции в поле комплексных чисел. Однако с этого момента число ω следует считать целым и все арифметические операции выполнять в конечном кольце вычетов по модулю m.

$$\sum_{i=0}^{n-1} a^i = (1+a)\sum_{i=0}^{n/2-1} (a^2)^i. \tag{7.6}$$

Индуктивное предположение и подстановка a^2 вместо a приводят к равенству

$$\sum_{i=0}^{n/2-1} (a^2)^i = \prod_{i=0}^{k-2} \left(1+(a^2)^{2^i}\right) = \prod_{i=1}^{k-1} \left(1+a^{2^i}\right). \tag{7.7}$$

Подставляя (7.7) в правую часть равенства (7.6), получаем требуемый результат. □

Лемма 7.5. Пусть $m = \omega^{n/2}+1$, где $\omega \in S$, $\omega \neq 0$. Тогда для $1 \le p \le n$ выполняется тождество $\sum_{i=0}^{n-1} \omega^{ip} \equiv 0 \pmod m$.

Доказательство. По лемме 7.4 достаточно показать, что $1+\omega^{2^j p} \equiv 0 \pmod m$ для некоторого $0 \le j \le k$. Пусть $p = 2^s p'$, где p' нечетно. Очевидно, что $0 \le s < k$. Выберем j так, чтобы $i + s = k - 1$. Тогда $1+\omega^{2^j p} = 1+\omega^{2^{k-1} p'} = 1+(m-1)^{p'}$. Однако $m - 1 \equiv -1 \pmod m$ и p' нечетно, так что $(m-1)^{p'} \equiv -1 \pmod m$. Отсюда следует, что $1+\omega^{2^j p} \equiv 0 \pmod m$ для $j = k - 1 - s$. □

Теорема 7.5. Пусть n и ω — положительные степени числа 2 и $m = \omega^{n/2} + 1$. Пусть R_m — кольцо вычетов по модулю m. Тогда в кольце R_m элемент n имеет обратный (по модулю m) элемент и ω — примитивный корень n-й степени из единицы.

Доказательство. Поскольку n — степень числа 2, а m — нечетно, то m и n являются взаимно простыми числами. Поэтому число n имеет обратное по модулю m.[7] Поскольку $\omega \neq 1$, то $\omega^n = \omega^{n/2}\omega^{n/2} \equiv (-1)(-1) = 1 \bmod (\omega^{n/2}+1)$. Тогда из леммы 7.5 следует, что ω — примитивный корень n-й степени из единицы в R_m. □

Важность теоремы 7.5 объясняется тем, что теорема о свертке верна для кольца целых чисел по модулю $2^{n/2} + 1$. Если необходимо вычислить свертку двух n-мерных векторов с целыми компонентами, причем компоненты свертки лежат в диапазоне от 0 до $2^{n/2}$, мы можем быть уверены, что ответ будет точным. Если же компоненты свертки не лежат в диапазоне от 0 до $2^{n/2}$, то они точны по модулю $2^{n/2} + 1$.

---

[7] Это утверждение — одна из основных теорем теории чисел. В разделе 8.8 будет показано, что если a и b — взаимно простые число, то существуют такие целые числа x и y, что $ax + by = 1$. Тогда $ax = 1 \pmod b$. Полагая $b = m$ и $a = n$, получаем наш результат.

Мы почти готовы установить, сколько битовых операций требуется для вычисления свертки по модулю m. Сначала, однако, определим, сколько битовых операций требуется для вычисления вычета целого числа по модулю m, поскольку это существенный этап в подсчете количества битовых операций на основе количества арифметических операций по модулю m.

Пусть $m = \omega^p + 1$ для некоторого целого числа p. Для вычисления a по модулю m применим обобщение приема "отбрасывания девяток". Если число a разложено по основанию ω^p, т.е. записано в виде последовательности из l блоков по p цифр в каждом, то a по модулю m можно сосчитать, попеременно складывая и вычитая эти l блоков из p цифр.

Лемма 7.6. Пусть $m = \omega^p + 1$ и $a = \sum_{i=0}^{l-1} a_i \omega^{pi}$, где $0 \le a_i \le \omega^p$ для каждого i. Тогда $a \equiv \sum_{i=0}^{l-1} a_i (-1)^i \pmod{m}$.

Доказательство. Достаточно заметить, что $\omega^p \equiv -1 \pmod{m}$. □

Заметим, что если число блоков l в лемме 7.6 фиксировано, то вычет числа a по модулю m можно найти за $O_{\text{в}}(p \log \omega)$ битовых операций.

Пример 7.3. Пусть $n = 4$, $\omega = 2$ и $m = 2^2 + 1$. Тогда, применяя лемму 7.6, положим $p = 2$. Рассмотрим число $a = 101100$ в двоичной системе счисления. В данном случае $a_0 = 00$, $a_1 = 11$ и $a_2 = 10$. Вычислим $a_0 - a_1 + a_2 = -1$, а затем, прибавляя m, находим, что $a \equiv 4 \pmod{5}$. Поскольку $a = 44$, то результат правильный. □

Лемма 7.6 дает эффективный метод вычисления a по модулю m. Она играет важную роль в следующей теореме, устанавливающей верхнюю границу количества битовых операций, требуемых для вычисления дискретного преобразования Фурье и обратного к нему.

Теорема 7.6. Пусть ω и n — степени числа 2 и $m = \omega^{n/2}+1$. Пусть $[a_0, a_1, ..., a_{n-1}]^T$ — вектор с целочисленными компонентами, где $0 \le a_i < m$ для каждого i. Тогда дискретное преобразование Фурье для вектора $[a_0, a_1, ..., a_{n-1}]^T$ и обратное к нему преобразование можно вычислить по модулю m за $O_{\text{в}}(n^2 \log n \log \omega)$ шагов.

Доказательство. Применим алгоритм 7.1 или 7.2. Для обратного преобразования подставим ω^{-1} вместо ω и умножим каждый результат на n^{-1}. Целое число по модулю m можно представить в виде строки, состоящей из $b = ((n/2)\log \omega) + 1$ битов. Поскольку $m = 2^{b-1} + 1$, то вычеты по модулю m можно представить в виде двоичных чисел от 00...0 до 100...0.

В алгоритме 7.1 используется сложение целых чисел по модулю m и умножение по модулю m целого числа на степень числа ω. Эти операции выполняются за время $O(n\log n)$. Вследствие леммы 7.6 сложение по модулю m занимает $O_{\text{в}}(b)$ шагов, где $b=\big((n/2)\log\omega\big)+1$. Умножение на ω^p, $0 \le p < n$, эквивалентно сдвигу влево на $p\log\omega$ разрядов, поскольку ω — степень числа 2. Получающееся целое число содержит не более $3b-2$ разрядов, так что по лемме 7.6 сдвиг и последующее вычисление вычета занимают $O_{\text{в}}(b)$ шагов. Таким образом, прямое преобразование Фурье можно выполнить за время $O_{\text{в}}\big(bn\log n\big)$, т.е. $O_{\text{в}}\big(n^2\log n\log\omega\big)$.

Для вычисления обратного преобразования необходимо выполнить умножение на ω^{-p} и n^{-1}. Поскольку $\omega^{n-p}\omega^p \equiv 1$, то $\omega^{n-p} \equiv \omega^{-p} \pmod m$. Следовательно, вместо умножения на ω^{-p} можно умножать на ω^{n-p}, что эквивалентно сдвигу влево на $(n-p)\log\omega$ разрядов, причем получающееся целое число содержит не более $3b-2$ разрядов. По лемме 7.6 вычеты можно найти за $O_{\text{в}}(b)$ шагов. Наконец, рассмотрим умножение на n^{-1}. Если $n = 2^k$, то оно сводится к сдвигу влево на $n\log\omega-k$ разрядов, в результате которого получается число не более чем с $3b-2$ двоичными разрядами, и вычислению вычета по лемме 7.6. Таким образом, выполнение обратного преобразования Фурье также требует $O_{\text{в}}\big(n^2\log n\log\omega\big)$ шагов. □

Пример 7.4. Пусть $\omega = 2$, $n = 4$ и $m = 5$. Вычислим преобразование Фурье вектора $[a_0, a_1, a_2, a_3]^T$, где $a_i = i$. Поскольку $a_i < 5$ для каждого i, можно ожидать, что этот вектор можно будет восстановить, проведя вычисления по модулю m. Для представления чисел мы используем три бита, но в действительности у нас будут лишь числа 000, ..., 100, если не считать промежуточных результатов.

В соответствии с алгоритмом 7.1 мы должны вычислить коэффициенты, указанные на рис. 7.6, где вместо ω подставлено число 2.

a_0	a_1	a_2	a_3
a_0+a_2	a_1+a_3	a_0+4a_2	a_1+4a_3
$a_0+a_2+(a_1+a_3)\equiv$ $\equiv a_0+a_1+a_2+a_3\equiv$ $\equiv b_0$	$a_0+a_2+4(a_1+a_3)\equiv$ $\equiv a_0+4a_1+a_2+4a_3\equiv$ $\equiv b_2$	$a_0+4a_2+2(a_1+4a_3)\equiv$ $\equiv a_0+2a_1+4a_2+8a_3\equiv$ $\equiv b_1$	$a_0+4a_2+8(a_1+4a_3)\equiv$ $\equiv a_0+8a_1+4a_2+2a_3\equiv$ $\equiv b_3$

Рис. 7.6. Вычисление быстрого преобразования Фурье для $n = 4$

Приведем значения переменных и выражений на рис. 7.6:

$a_0 = 000$	$a_1 = 001$	$a_2 = 010$	$a_3 = 011$
$a_0+a_2 = 010$	$a_1+a_3 = 100$	$a_0+4a_2 = 011$	$a_1+4a_3 = 011$
$b_0 = 001$	$b_2 = 011$	$b_1 = 100$	$b_3 = 010$

Таким образом, преобразованием вектора $[0, 1, 2, 3]^T$ является вектор $[1, 4, 3, 2]^T$ mod 5. Рассмотрим последний элемент b_3 в нижней строке. Он вычисляется по двум последним элементам в средней строке:

Берем $a_1 + 4a_3$	011
Сдвигаем на три разряда влево (умножаем на 8)	11000
Расщепляем на три блока по 2 разряда	1 10 00
Складываем первый и третий блоки и вычитаем второй	−1
Прибавляем $m = 5$	100
Прибавляем $a_0 + 4a_2 = 011$	111
Вычитаем m	010

Для вычисления обратного преобразования заметим, что $2^{-1} \equiv 8 \pmod 5$, $4^{-1} \equiv 4 \pmod 5$ и $8^{-1} \equiv 2 \pmod 5$. Таким образом, формулы для обратного преобразования можно вывести из рис. 7.6, поменяв местами a_i и b_i, а также 2 и 8. Это вычисление выглядит так:

$b_0 = 001$	$b_1 = 100$	$b_2 = 011$	$b_3 = 010$
$b_0 + b_2 = 100$	$b_1 + b_3 = 001$	$b_0 + 4b_2 = 011$	$b_1 + 4b_3 = 010$
$4a_0 = 000$	$4a_2 = 011$	$4a_1 = 100$	$4a_3 = 010$

Наконец, разделим каждый ответ на 4 (умножим на 4, поскольку $4^{-1} \equiv 4 \pmod 5$), и получаем $[0, 1, 2, 3]^T$ в качестве $[a_0, a_1, a_2, a_3]^T$. □

Следствие теоремы 7.6. Пусть на вычисление произведения двух kразрядных двоичных целых чисел тратится $O_B(M(k))$ шагов. Пусть **a** и **b** — n-мерные векторы длины n с целочисленными компонентами, лежащими в диапазоне от 0 и ω^n, где n и ω — степени числа 2. Тогда свертку $\mathbf{a} \circledast \mathbf{b}$, а также положительно и отрицательно обернутые свертки векторов **a** и **b** по модулю $\omega^n + 1$ можно вычислить за время

$$O_B\left(\max\left(n^2 \log n \log \omega,\ nM\left(n \log \omega\right)\right)\right).$$

Здесь первый аргумент представляет время вычисления преобразований, а второй — выполнения $2n$ умножений $\left(n \log \omega + 1\right)$ разрядных двоичных целых чисел. Наилучшее известное значение $M(k)$ равно $k \log k \log\log k$ (см. раздел 7.5). При этом значении второй аргумент больше первого, так что для вычисления соответствующей свертки требуется $O_B\left(n^2 \log n \log\log n \log \omega \log\log \omega \log\log\log \omega\right)$ шагов.

7.4. Произведение полиномов

Задача умножения двух полиномов от одной переменной по существу эквивалентна задаче вычисления свертки двух последовательностей. Иначе говоря,

$$\left(\sum_{i=0}^{n-1} a_i x^i\right)\left(\sum_{j=0}^{n-1} b_j x^j\right) = \sum_{k=0}^{2n-2} c_k x^k, \text{ где } c_k = \sum_{m=0}^{n-1} a_m b_{k-m}.$$

Как и ранее, коэффициенты a_p и b_p считаются нулями, если $p < 0$ или $p \geq n$. Напомним, что коэффициент c_{2n-1} должен быть нулем. Поэтому существуют дополнительные следствия теоремы 7.4.

Следствие 3 теоремы 7.4. Коэффициенты произведения двух полиномов степени n можно вычислить за $O_A(n \log n)$ шагов.

Доказательство. Вытекает из следствия 1 теоремы 7.4 и соображений, приведенных выше. □

Следствие 4 теоремы 7.4. Допустим, произведение двух k-битовых двоичных целых чисел можно вычислить за $M(k)$ шагов и

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \text{ и } q(x) = \sum_{j=0}^{n-1} b_j x^j,$$

где a_i и b_i — целые числа в диапазоне от 0 и $\omega^{n/2}/\sqrt{n}$ для всех i и j, а n и ω — степени числа 2. Тогда коэффициенты полинома $p(x)q(x)$ можно вычислить за $O_B\left(\max\left(n^2 \log n \log \omega,\ nM(n \log \omega)\right)\right)$ шагов.

Доказательство. Вытекает из теоремы 7.4 и следствия теоремы 7.6. □

Снова заметим, что в следствии 4 доминирует второй аргумент.

Фактически теорема 7.1 допускает такую интерпретацию. Предположим, что $p(x)$ и $q(x)$ — полиномы степени $n - 1$. Можно вычислить полиномы p и q в любых $2n - 1$ или более точках, скажем c_0, c_1, ..., а затем перемножить $p(c_j)$ и $q(c_j)$, получив значения pq в тех же точках. По этим значениям можно выполнить интерполяцию и построить единственный полином степени $2n - 2$, являющийся произведением $p(x)q(x)$.

Когда преобразование Фурье применяют к вычислению свертки (или, что эквивалентно, к умножению полиномов), выбирают $c_j = \omega^j$, где ω — примитивный корень степени $2n$ из единицы. Далее находят преобразования Фурье полиномов p и q (т.е. вычисляют p и q в точках c_0, c_1, ...), попарно перемножают результаты этих преобразований (т.е. умножают $p(c_j)$ на $q(c_j)$, чтобы получить значение произведения в точке c_j), и вычисляют pq, применяя обратное преобразование. Лемма 7.1

на самом деле гарантирует, что вычисление обратного преобразования сводится к формулам интерполяции. Иначе говоря, мы действительно восстанавливаем полином по его значениям в точках ω^0, ω^1, ..., ω^{2n-1}.

7.5. Алгоритм Шёнхаге–Штрассена для умножения целых чисел

Теперь изучим важное приложение теоремы о свертке — алгоритм быстрого побитового умножения целых чисел. В разделе 2.6 мы видели, как перемножить два n-разрядных двоичных числа за $O\left(n^{\log 3}\right)$ шагов, разбивая каждое двоичное число на два $(n/2)$-разрядных числа. Этот метод можно обобщить, разбивая числа на b блоков по l разрядов в каждом. Если рассматривать эти b блоков как коэффициенты полинома, получатся выражения, аналогичные (2.4). Чтобы определить коэффициенты произведения полиномов, вычислим полиномы на некотором подходящем множестве точек, перемножим найденные значения и интерполируем. Выбрав в качестве точек, в которых вычисляются полиномы, примитивные корни из единицы, можно воспользоваться преобразованием Фурье и теоремой о свертке. Сделав b функцией от n и применив рекурсию, можно умножить два n-разрядных двоичных числа за $O_{\text{в}}\left(n \log n \log\log n\right)$ шагов.

Для простоты ограничимся случаем, когда n — степень числа 2. В случае, когда n не является степенью числа 2, необходимо добавить к входным векторам достаточное количество ведущих нулей, чтобы n стало степенью числа 2 (это увеличивает только постоянный множитель). Кроме того, мы вычислим произведение двух n-разрядных двоичных чисел по модулю $2^n + 1$. Чтобы найти точное значение произведения двух n-разрядных двоичных чисел, необходимо добавить ведущие нули и перемножать $2n$-разрядные числа по модулю $2^{2n} + 1$, тем самым увеличивая время опять не более чем в постоянное число раз.

Пусть u и v — двоичные целые числа в диапазоне от 0 до 2^n, которые требуется перемножить по модулю $2^n + 1$. Заметим, что двоичное представление числа 2^n занимает $n + 1$ битов. Если число u или v равно 2^n, то оно представляется специальным символом -1, и в этом особом случае умножение выполняется легко: если $u = 2^n$, то uv по модулю $2^n + 1$ получается путем вычисления $2^n + 1 - v$ по модулю $2^n + 1$.

Допустим, что $n = 2^k$. Положим $b = 2^{k/2}$, если k четное, и $b = 2^{(k-1)/2}$ в противном случае. Пусть $l = n/b$. Заметим, что $l \geq b$ и l делится на b без остатка. Первый шаг состоит в разбиении u и v на b блоков по l битов в каждом. Таким образом,

$$u = u_{b-1}2^{(b-1)l} + \ldots + u_1 2^l + u_0$$

и

$$v = v_{b-1}2^{(b-1)l} + \ldots + v_1 2^l + v_0.$$

Произведение uv можно представить в виде

$$uv = y_{2b-2}2^{(2b-2)l} + ... + y_1 2^l + y_0, \tag{7.8}$$

где

$$y_i = \sum_{j=0}^{b-1} u_j v_{i-j}, \quad 0 \le i < 2b.$$

(Для $j < 0$ или $j > b - 1$ положим $u_j = v_j = 0$. Член y_{2b-1} равен 0 и включен только для симметрии.)

Произведение uv можно вычислить с помощью теоремы о свертке. Перемножение преобразований Фурье требует $2b$ умножений. С помощью обернутой свертки количество умножений можно уменьшить до b. Именно по этой причине мы вычисляем uv по модулю $2^n + 1$. Поскольку $bl = n$, то $2^{bl} \equiv -1 \pmod{(2^n + 1)}$. Отсюда в силу (7.8) и леммы 7.6, вычислив uv по модулю $2^n + 1$, находим, что

$$uv \equiv (w_{b-1}2^{(b-1)l} + ... + w_1 2^l + w_0) \pmod{(2^n + 1)},$$

где $w_i = y_i - y_{b+i}$, $i < b$.

Поскольку произведение двух l-битовых двоичных чисел меньше 2^{2l}, а y_i и y_{b+i} представляют собой суммы, составленные из $i+1$ и $b - (i + 1)$ таких произведений соответственно, то $w_i = y_i - y_{b+i}$ удовлетворяет неравенствам $-(b - 1 - i)2^{2l} < w_i < (i + 1)2^{2l}$. Следовательно, w_i может принимать не более чем $b2^{2l}$ значений. Если мы сможем вычислить все w_i по модулю $b2^{2l}$, то сможем вычислить uv по модулю $2^n + 1$, сделав $O\left(b\log\left(b2^{2l}\right)\right)$ дополнительных шагов для сложения b чисел w_i с необходимыми сдвигами.

Чтобы вычислить все w_i по модулю $b2^{2l}$, вычислим эти w_i дважды — по модулю b и по модулю $2^{2l} + 1$. Пусть w_i' — это w_i по модулю b, а w_i'' — это w_i по модулю $2^{2l} + 1$. Поскольку b — степень числа 2, а число $2^{2l} + 1$ нечетно, то b и $2^{2l} + 1$ являются взаимно простыми числами. Поэтому w_i можно получить из w_i' и w_i'' по формуле

$$w_i = \left(2^{2l} + 1\right)\left(\left(w_i' - w_i''\right) \bmod b\right) + w_i'',^{8}$$

где w_i лежит в диапазоне от $-(b - 1 - i)2^{2l}$ до $(i + 1)2^{2l}$. Вычисление w_i по w_i' и w_i'' требует $O\left(l + \log b\right)$ шагов для каждого w_i, что дает в целом $O\left(bl + b\log b\right)$, или $O(n)$ шагов.

Значения w_i по модулю b вычисляются так: берем $u_i' = u_i \pmod b$ и $v_i' = v_i \pmod b$ и формируем два $3b\log b$-битовых числа $\hat{u}$ и $\hat{v}$, как показано на рис. 7.7. Произведение $\hat{u}\hat{v}$ вычисляется с помощью алгоритма из раздела 2.6 не более чем

---

[8] Если для взаимно простых p_1 и p_2 выполнены сравнения $w \equiv q_1 \bmod p_1$, $w \equiv q_2 \bmod p_2$ и $0 < w < p_1p_2$, то $w = p_2\left(p_2^{-1} \bmod p_1\right)\left(q_1 - q_2 \bmod p_1\right) + q_2$. Пусть $p_1 = b$ и $p_2 = 2^2l + 1$. Поскольку b — степень числа 2 и $b \le 2^{2l}$, то b делит 2^{2l} и, значит, элементом, обратным к $2^{2l} + 1$ по модулю b, является 1.

за $O\left((3b\log b)^{1,6}\right)$ шагов, т.е. менее чем за $O(n)$ шагов. Далее, $\hat{u}\hat{v} = \sum_{i=0}^{2b-1} y_i' 2^{(3\log b)i}$, где $y_i' = \sum_{j=0}^{2b-1} u_i' v_{i-j}'$. Заметим, что $y_i' < 2^{3\log b}$, так что все y_i' можно легко восстановить по произведению $\hat{u}\hat{v}$. Тогда w_i по модулю b легко можно найти, вычисляя $y_i' - y_{b+i}'$ по модулю b.

$$\hat{u} = u_{b-1}' 00\ldots0 u_{b-2}' 00\ldots0 u_{b-3}' \ldots 00\ldots0 u_0'$$
$$\hat{v} = v_{b-1}' 00\ldots0 v_{b-2}' 00\ldots0 v_{b-3}' \ldots 00\ldots0 v_0'$$

Рис. 7.7. Составные числа, используемые при вычислении w_i по модулю b. В каждом блоке, состоящем из нулей, содержатся $2\log b$ нулей.

Вычислив все w_i по модулю b, вычисляем w_i по модулю $2^{2l} + 1$ с помощью обернутой свертки. Для этого надо вычислить преобразование Фурье, выполнить попарные умножения и осуществить обратное преобразование. Пусть $\omega = 2^{4l/b}$ и $m = 2^{2l} + 1$. По теореме 7.5 элементы ω и b имеют обратные по модулю m, а ω — примитивный корень b-й степени из единицы. Поэтому отрицательно обернутая свертка векторов $[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]$ и $[v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]$, где $\psi = 2^{2l/b}$ (ψ — корень степени $2b$ из единицы), равна

$$[(y_0 - y_b), \psi(y_1 - y_{b+1}), \ldots, \psi^{b-1}(y_{b-1} - y_{2b-1})] \ (\mathrm{mod}(2^{2l} + 1)),$$

где $y_i = \sum_{j=0}^{b-1} u_j v_{i-j}$ для $0 \le i \le 2b - 1$. Значения w_i по модулю $2^{2l} + 1$ можно получить соответствующим сдвигом. Итак, опишем полный алгоритм.

Алгоритм 7.3. Алгоритм Шёнхаге–Штрассена для умножения целых чисел

Вход. Два n-разрядных двоичных целых числа u и v, где $n = 2^k$.

Выход. $(n + 1)$-разрядное произведение uv по модулю $2^n + 1$.

Метод. Если n мало, умножьте u на v по модулю $2^n + 1$ с помощью любого алгоритма. Для большого n положим $b = 2^{k/2}$, если k — четное число, и $b = 2^{(k-1)/2}$, если k — нечетное. Пусть $l = n/b$. Представим $u = \sum_{i=0}^{b-1} u_i 2^{li}$ и $v = \sum_{i=0}^{b-1} v_i 2^{li}$, где u_i и v_i — целые числа между 0 и $2^l - 1$ (т.е. числа u_i — это блоки, составленные из l битов числа u, а v_i — блоки из l битов числа v).

1. Вычисляем преобразование Фурье по модулю $2^{2l} + 1$ векторов

$$[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]^T \text{ и } [v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]^T,$$

где $\psi = 2^{2l/b}$ и ψ^2 — примитивный корень b-й степени из единицы.

2. Вычисляем по модулю $2^{2l} + 1$ покомпонентное произведение преобразований Фурье, полученных на шаге 1, с помощью алгоритма 7.3, который рекурсивно применяется для вычисления произведения каждой пары соответствующих компонентов. (Ситуация, когда одно из чисел равно 2^{2l}, рассматривается как простой частный случай.)

3. Вычисляем обратное преобразование Фурье по модулю $2^{2l} + 1$ вектора, равного покомпонентному произведению, полученному на шаге 2. Результатом будет вектор $[w_0, \psi w_1, ..., \psi^{b-1} w_{b-1}]^T$ по модулю $2^{2l} + 1$, где w_i обозначает i-й компонент отрицательно обернутой свертки векторов $[u_0, u_1, ..., u_{b-1}]^T$ и $[v_0, v_1, ..., v_{b-1}]^T$. Вычисляем $w_i'' = w_i$ по модулю $2^{2l} + 1$, умножая $\psi^i w_i$ на ψ^{-i} по модулю $2^{2l} + 1$.

4. Вычисляем $w_i' = w_i \bmod b$ следующим образом.

   а) Пусть $u_i' = u_i \bmod b$ и $v_i' = v_i \bmod b$ для $0 \le i < b$.

   б) Построим числа $\hat{u}$ и $\hat{v}$, выписывая числа u_i' и v_i' в виде строки и вставляя между ними $2\log b$ нулей, т.е. $\hat{u} = \sum_{i=0}^{b-1} u_i' 2^{(3\log b)i}$ и $\hat{v} = \sum_{i=0}^{b-1} v_i' 2^{(3\log b)i}$.

   в) Вычисляем произведение $\hat{u}\hat{v}$ с помощью алгоритма из раздела 2.6.

   г) Произведение $\hat{u}\hat{v}$ имеет вид $\sum_{i=0}^{2b-1} y_i' 2^{(3\log b)i}$, где $y_i' = \sum_{j=0}^{2b-1} u_j' v_{i-j}'$. С помощью этого произведения числа w_i по модулю b можно восстановить $w_i' = (y_i' - y_{b+i}') \bmod b$ при $0 \le i < b$.

5. Вычисляем точные значения w_i по формуле

$$w_i = \left(2^{2l} + 1\right)\left(\left(w_i' - w_i''\right) \bmod b\right) + w_i'',$$

   учитывая, что w_i лежит между $(b - 1 - i)2^{2l}$ и $(i + 1)2^{2l}$.

6. Вычисляем $\sum_{j=0}^{b-1} w_i 2^{li} \bmod \left(2^n + 1\right)$. Это и есть искомый результат. □

Теорема 7.7. Алгоритм 7.3 вычисляет произведение uv по модулю $(2^n + 1)$.

Доказательство. По теореме 7.2 шаги 1–3 алгоритма 7.3 правильно вычисляют значения w_i по модулю $2^{2l} + 1$. В качестве упражнения предлагаем читателям доказать, что шаг 4 вычисляет w_i по модулю b, а шаг 5 — w_i по модулю $b(2^{2l} + 1)$, т.е. точное значение w_i. □

Теорема 7.8. Временная сложность алгоритма 7.3 составляет

$$O_{\text{B}}\left(n \log n \log\log n\right).$$

Доказательство. В силу следствия теоремы 7.6 шаги 1–3 выполняются за время $O_{\text{в}}\left(bl\log b + bM\left(2l\right)\right)$, где $M(m)$ — время умножения двух m-битовых двоичных целых чисел путем рекурсивного применения этого алгоритма. На шаге 4 формируются и умножаются числа $\hat{u}$ и $\hat{v}$ длины $3b\log b$ за $O_{\text{в}}\left(\left(3b\log b\right)^{1{,}59}\right)$ шагов. Поскольку $\left(3b\log b\right)^{1{,}59} < b^2$ для достаточно больших b, то временем, затрачиваемым на шаге 4, можно пренебречь, поскольку шаги 1 и 3 занимают $O_{\text{в}}\left(b^2\log b\right)$ времени. Временная сложность шагов 5 и 6 составляет $O(n)$, так что и ими можно пренебречь.

Поскольку $n = bl$ и b не превышает $\sqrt{n}$, мы получаем рекуррентное соотношение

$$M\left(n\right) \le cn\log n + bM\left(2l\right) \tag{7.9}$$

для некоторой константы c и достаточно больших n. Пусть $M'(n) = M(n)/n$. Тогда (7.9) можно записать в виде

$$M'\left(n\right) \le c\log n + 2M'\left(2l\right). \tag{7.10}$$

Поскольку $l \le 2\sqrt{n}$, то

$$M'\left(n\right) \le c\log n + 2M'\left(4\sqrt{n}\right), \tag{7.11}$$

откуда следует, что $M'\left(n\right) \le c'\log n\log\log n$ при подходящем выборе константы c'. Чтобы убедиться в этом, подставим в (7.11) $c'\log\left(4\sqrt{n}\right)\log\log\left(4\sqrt{n}\right)$ вместо $M'\left(4\sqrt{n}\right)$. Очевидные преобразования приводят к неравенству

$$M'\left(n\right) \le c\log n + 4c'\log\left(2+\frac{1}{2}\log n\right) + c'\log n\log\left(2+\frac{1}{2}\log n\right).$$

При больших n выполняется неравенство $2+\frac{1}{2}\log n \le \frac{2}{3}\log n$. Следовательно,

$$M'\left(n\right) \le c\log n + 4c'\log\frac{2}{3} + 4c'\log\log n + c'\log\frac{2}{3}\log n + c'\log n\log\log n. \tag{7.12}$$

Для больших n и для достаточно большой константы c' первые три слагаемых не превышают абсолютную величину четвертого, а оно отрицательно. Таким образом, $M'\left(n\right) \le c'\log n\log\log n$. Отсюда заключаем, что $M\left(n\right) \le c'n\log n\log\log n$. □

Упражнения

7.1. Чему равны примитивные корни n-й степени из единицы в кольце комплексных чисел при $n = 3, 4, 5$?

7.2. Покажите, как реализовать алгоритм 7.2 без вспомогательного массива S.

7.3. Вычислите дискретное преобразование Фурье следующих последовательностей в кольце комплексных чисел:

а) [0, 1, 2, 3];

б) [1, 2, 0, 2, 0, 0, 0, 1].

7.4. Обобщите алгоритм быстрого преобразования Фурье на случай, когда n не является степенью числа 2.

Определение. Тройка целых чисел (ω, n, m) называется *допустимой*, если ω и n имеют обратные элементы относительно умножения и ω — примитивный корень n-й степени из единицы в кольце целых чисел по модулю m.

7.5. Какие из следующих троек являются допустимыми?

а) (3, 4, 5).

б) (2, 6, 21).

в) (2, 6, 7).

7.6. Покажите, что если (ω, n, m) — допустимая тройка, то $\omega^n \equiv 1 \pmod m$ и $\omega^p \not\equiv 1$, если $1 \le p < n$.

***7.7.** Покажите, что если n — степень числа 2, а числа $2^n \equiv 1 \pmod m$ и $2^p - 1$ являются взаимно простыми при $1 \le p < n$, то $(2, n, m)$ — допустимая тройка.

****7.8.** Покажите, что если m — простое число, а ω — произвольное положительное целое число, то найдется такое число n, что (ω, n, m) —допустимая тройка.

***7.9.** Покажите, что если a и b — взаимно простые числа, то $ac \equiv 1 \pmod b$ для некоторого числа c, и наоборот. Докажите, что c по модулю b является единственным.

7.10. Найдите $(10101110011110)_2$ по модулю $2^5 + 1$.

7.11. Пусть t — целое число, заданное своим десятичным представлением. Покажите, что если сложить все цифры числа t, затем сложить все цифры результата и т.д. до тех пор, пока не останется одна цифра, то в конце концов получится t по модулю 9.

7.12. С помощью теоремы о свертке вычислите свертку последовательностей [1, 2, 3, 4] и [4, 3, 2, 1] по модулю 17, используя допустимую тройку (2, 8, 17).

7.13. Используя алгоритм 7.3, вычислите произведение двоичных чисел $(1011011)_2$ и $(10001111)_2$.

7.14. Покажите, что в результате извлечения $\log\log n$ раз квадратного корня из $n = 2^{2^k}$ получится число 2.

**7.15. Разработайте алгоритм быстрого умножения целых чисел на основе обычной, а не обернутой свертки. Каково асимптотическое время работы вашего алгоритма?

*7.16. *Циклической разностью* $\Delta\mathbf{a}$ вектора $\mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T$ называется вектор $[a_0 - a_{n-1}, a_1 - a_0, a_2 - a_1, ..., a_{n-1} - a_{n-2}]^T$. Пусть

$$F(\mathbf{a}) = [a'_0, a'_1, ..., a'_{n-1}]^T.$$

Покажите, что $F(\Delta\mathbf{a}) = \left[0, a'_1(1-\omega), a'_2(1-\omega^2), ..., a'_{n-1}(1-\omega^{n-1})\right]$, где ω — соответствующий корень n-й степени из единицы.

**7.17. С помощью упражнения 7.16 покажите, что из $X(n) - X(n-1) = n$ и $X(0) = 0$ следует $X(n) = n(n+1)/2$.

*7.18. *Циркулянтом* называется матрица, в которой каждая строка получается из строки, стоящей над ней, циклическим сдвигом на одну позицию вправо. Например, матрица

$$\begin{bmatrix} a & b & c \\ c & a & b \\ b & c & a \end{bmatrix}$$

является циркулянтом размерностью 3×3. Покажите, что вычисление дискретного преобразования Фурье n-мерного вектора, где n — простое число, эквивалентно умножению на циркулянт размерностью $(n-1) \times (n-1)$.

**7.19. Покажите, что преобразование Фурье над конечным полем для простого числа n можно вычислить за $O_A(n \log n)$ шагов.

*7.20. Рассмотрите представление полинома значениями всех его производных в некоторой точке. Является ли преобразование коэффициентов полинома в значения его производных линейным?

**7.21. Дайте физическую интерпретацию обернутой свертки в терминах операций над полиномами.

*7.22. Используя быстрое преобразование Фурье, постройте алгоритм сложности $O(n \log n)$, который умножал бы теплицеву матрицу на вектор. Сравните ваш алгоритм с решением упражнения 6.26, *б*.

Проблема для исследования

7.23. Постройте более быстрые алгоритмы умножения целых чисел и дискретного преобразования Фурье. Или, наоборот, покажите, что алгоритм Шёнхаге–Штрассена или быстрого преобразования Фурье —

наилучший из возможных при некоторых ограничениях на модель вычислений. Paterson, Fisher and Meyer [1974] показали, что при определенных ограничениях для побитового умножения целых чисел требуется $O_{\text{B}}(n\log n/\log\log n)$. Аналогично Morgenstern [1973] показал, что при определенных ограничениях дискретное преобразование Фурье требует $O_{\text{A}}(n\log n)$ шагов.

Библиографические замечания

Cooley, Lewis and Welch [1967] указывают, что способ быстрого преобразования Фурье описан еще в книге Runge and König [1924]. Его применяли Danielsona and Lanczosh [1942] и Good [1958, 1960]. прирлда этого метода хорошо описана в фундаментальной работе Cooley and Tukey [1965]. Интерпретация быстрого преобразования Фурье как деления полиномов, описанная в книге, принадлежит Fisuccia [1972]. Ввиду важности этого алгоритма для вычислений много внимания уделялось его эффективной реализации (см., например, работу Gentleman and Sande [1966] и многочисленные статьи в сборнике под редакцией Rabiner and Rader [1972]). Алгоритм умножения целых чисел заимствован из работы Schönhage and Strassen [1971]. Упражнения 7.18 и 7.19 взяты у Rader [1968].

ГЛАВА 8

Арифметические операции над целыми числами и полиномами

Арифметические операции над целыми числами и полиномами одной переменной целесообразно изучать вместе потому, что они по существу идентичны. Это относится не только к умножению и делению, но и к более сложным операциям. Например, вычет целого числа по модулю, задаваемому другим целым числом, равен значению полинома в точке. Представление целого числа его вычетами эквивалентно представлению полинома его значениями в нескольких точках. Восстановление целого числа по его вычетам ("китайская теорема об остатках") эквивалентно интерполированию полинома.

В этой главе мы покажем, что для выполнения некоторых операций над целыми числами и полиномами, например деления и возведения в квадрат, требуется время того же порядка, что и для умножения. Время выполнения других операций, таких, операции с вычетами или вычисление наибольших общих делителей, как упомянутые выше, может превосходить время умножения не более чем в $\log n$ раз, где n — длина двоичного представления целого числа или степень полинома. Мы будем чередовать результаты для целых чисел с соответствующими результатами для полиномов, причем обычно будем доказывать эти результаты только для чего-то одного, а доказательство для другого будем оставлять в качестве упражнения. Как и в остальных главах, основное внимание будет уделяться асимптотически наиболее эффективным алгоритмам.

В конце главы мы кратко обсудим различие между моделями полиномов, в одной из которых предполагается, что большинство коэффициентов отличны от нуля (плотная модель), а в другой считается, что большинство коэффициентов равны нулю (разреженная модель). Разреженная модель особенно полезна в случае полиномов от многих переменных (этот случай мы не обсуждаем).

8.1. Аналогии между целыми числами и полиномами

Наиболее очевидным образом схожесть между неотрицательными целыми числами и полиномами от одной переменной проявляется в возможности представлять каждый из них в виде конечного степенного ряда $\sum_{i=0}^{n-1} a_i x^i$. В случае целых чисел коэффициенты a_i можно выбирать из множества $\{0, 1\}$ при $x = 2$. В случае полиномов коэффициенты a_i можно выбирать из некоторого множества коэффициентов,[1] считая x переменной.

Существует естественный "размер" — это по существу длина степенного ряда, представляющего целое число или полином. В случае двоичного целого числа размер — это количество битов, необходимых для его представления; в случае полинома размер — это количество его коэффициентов. Таким образом, мы приходим к следующему определению.

Определение. Если i — неотрицательное целое число, то $\mathtt{SIZE}(i) = \lfloor \log i \rfloor + 1$. Если $p(x)$ — полином, то $\mathtt{SIZE}(p) = \mathtt{DEG}(p) + 1$, где $\mathtt{DEG}(p)$ — степень полинома p, т.е. наибольшая степень переменной x с ненулевым коэффициентом.

Над целыми числами и полиномами можно выполнять приближенное деление. Если a и b — два целых числа и $b \neq 0$, то найдется единственная пара целых чисел q и r, для которых

1) $a = bq + r$,
2) $r < b$,

где q и r — соответственно частное и остаток от деления a на b. Аналогично, если a и b — полиномы, причем b — не константа, то можно найти такие полиномы q и r, что

1) $a = bq + r$,
2) $\mathtt{DEG}(r) < \mathtt{DEG}(b)$.

Другое сходство между целыми числами и полиномами — существование удивительно быстрых алгоритмов умножения. В предыдущей главе мы показали, что два полинома степени n с вещественными коэффициентами можно перемножить

---

[1] Далее множество коэффициентов можно считать полем вещественных чисел (см. раздел 12.1), хотя результаты верны для любого поля коэффициентов, если измерять вычислительную сложность количеством операций в этом поле. Таким образом, в нашей модели размер коэффициентов не учитывается.

с помощью быстрого преобразования Фурье за время $O_A(n\log n)$. Сложность измеряется количеством арифметических операций, поскольку на практике полиномы с фиксированной точностью представляются их коэффициентами, и операции над полиномами сводятся к арифметическим операциям над их коэффициентами.

С помощью алгоритма Шёнхаге–Штрассена из раздела 7.5 можно умножить два n-разрядных двоичных целых числа за время $O_B(n\log n\log\log n)$. Мы утверждаем, что в случае целых чисел интересны лишь битовые операции. Фактически существуют только две ситуации, в которых умножение целых чисел не рассматривается как элементарная операция. Первая — это разработка аппаратной реализации умножения, в которой количество битовых операций соответствует числу элементов, необходимых для схемы умножения. Вторая — разработка алгоритмов произвольной точности для операций с фиксированной запятой для вычислительных машин со словами фиксированной длины, где количество битовых операций соответствует числу машинных команд, необходимых для умножения с точностью до n разрядов.

Таким образом, если пользоваться двумя различными мерами сложности (арифметической и битовой), результаты арифметических операций над полиномами и целыми числами окажутся очень похожими. Эти две меры аналогичны в том смысле, что битовые операции — это операции над коэффициентами степенных рядов, представляющих целые числа, а арифметические — это операции над коэффициентами полиномов.

8.2. Умножение и деление целых чисел

Покажем, что время, затрачиваемое на умножение целых чисел, выраженное с помощью количества битовых операций, с точностью до постоянного множителя равно времени, затрачиваемому на деление, и оно так же связано с операциями возведения в квадрат и обращения. В данном разделе мы будем использовать следующие обозначения.

Символ	Описание
$M(n)$	Время умножения двух целых чисел размера n
$D(n)$	Время деления целого числа размера не более $2n$ на целое число размера n
$S(n)$	Время возведения в квадрат целого числа размера n
$R(n)$	Время обращения целого числа размера n

Здесь время измеряется количеством битовых операций. Мы будем придерживаться разумного предположения, что $M(n)$ удовлетворяет неравенствам

$$a^2M(n) \geq M(an) \geq aM(n)$$

для $a \geq 1$. Будем предполагать, что остальные три функции также обладают этим свойством.

Сначала покажем, что для вычисления числа, обратного к заданному n-разрядному двоичному целому числу, требуется столько же времени, сколько для умножения двух n-разрядных двоичных чисел. Поскольку $1/i$ не является целым числом при $i > 1$, то под "обратным" к числу i мы на самом деле понимаем приближение к дроби $1/i$, имеющее n значащих битов. Так как предполагается, что изменение масштаба (сдвиг двоичной запятой) не требует временных затрат, мы можем сказать, что "обратное" к числу i — это частное от деления 2^{2n-1} на i. В оставшейся части этого раздела термин "обратное число" употребляется именно в этом смысле.

Сначала рассмотрим, как найти последовательные приближения к числу $A = 1/P$, где P — целое число. Пусть A_i — i-е приближение к $1/P$. Тогда точное значение A можно представить в виде

$$A = A_i + (1/P)(1 - A_i P). \tag{8.1}$$

Если в равенстве (8.1) в качестве приближения к $1/P$ взять число A_i, то получим формулу, которую можно использовать для нахождения $(i + 1)$-го приближения числа A по его i-му приближению:

$$A_{i+1} = A_i + A_i(1 - A_i P) = 2A_i - A_i^2 P. \tag{8.2}$$

Заметим, что если $A_i P = 1 - S$, то

$$A_{i+1} P = 2A_i P - A_i^2 P^2 = 2(1-S) - (1-S)^2 = 1 - S^2.$$

Это показывает, что итерационный процесс (8.2) имеет квадратичную скорость сходимости. Если $S \leq 1/2$, то при каждой итерации количество правильных битов удваивается.

Поскольку число P предположительно содержит много битов, то для первых приближений не обязательно брать все его цифры, потому что на правильные цифры числа A_{i+1} влияют только старшие биты числа P. Если первые k цифр справа от запятой в числе в A_i верны, то $2k$ цифр числа A_{i+1} можно найти по формуле (8.2). Иначе говоря, для того чтобы вычислить число $A_{i+1} = 2A_i - A_i^2 P$, мы округляем число $2A_i$ до k цифр справа от запятой, вычисляем $A_i^2 P$, используя $2k$-разрядное приближение числа P и отбрасываем в произведении все разряды правее $2k$-го после запятой. Конечно, такое использование приближенных значений может повлиять на сходимость, поскольку выше предполагалось, что число P является точным.

Применим эти соображения в алгоритме, вычисляющем частное $\lfloor 2^{2n-1}/P \rfloor$, где $P = p_1 p_2 \ldots p_n$ — это n-разрядное двоичное целое число с $p_1 = 1$. Этот метод по существу определяется формулой (8.2), к которой добавлен перенос запятой, чтобы можно было работать только с целыми числами. Это позволяет не заботиться о положении запятой.

Алгоритм 8.1. Вычисление чисел, обратных к целым числам

Вход. n-разрядное двоичное целое число $P = [p_1 p_2 ... p_n]$ при $p_1 = 1$. Для удобства будем предполагать, что n — степень числа 2, а $[x]$ — целое число, заданное битовой строкой (например, [110] = 6).

Выход. Целое число $A = [a_0 a_1 ... a_n] = \lfloor 2^{2n-1}/P \rfloor$.

Метод. Вызываем процедуру RECIPROCAL($p_1 p_2 ... p_n$), где RECIPROCAL — рекурсивная процедура, приведенная на рис. 8.1. Она вычисляет приближение числа $\left\lfloor 2^{2k-1}/[p_1 p_2 ... p_k] \right\rfloor$ для любого k, представляющего собой степень числа 2. Результатом обычно является целое k-разрядное число за исключением ситуаций, в которых P является степенью числа 2. В последнем случае результат представляет собой $(k + 1)$-разрядное число.

По заданным k битам числа, обратного к $[p_1 p_2 ... p_k]$, строки 2–4 вычисляют $2k - 3$ битов числа, обратного к $[p_1 p_2 ... p_{2k}]$. Строки 5–7 корректируют последние три бита. На практике можно было бы пропустить строки 5–7 и получить нужную точность дополнительным применением формулы (8.2) в конце. Мы предпочли включить в программу цикл в строках 5–7, чтобы упростить описание алгоритма и доказательство правильности его работы. □

```
      procedure RECIPROCAL ([p1p1...pk]):
1.    if k = 1 then return [10]
      else
          begin
2.            [c0c1...ck/2] ← RECIPROCAL ([p1p1...pk/2]);
3.            [d1d2...d2k] ← [c0c1...ck/2] * 2^(3k/2) -
                  [c0c1...ck/2]^2 * [p1p2...pk];
              comment Хотя правая часть оператора в строке 3
                  предназначена для вычисления (2k + 1)-разрядного
                  числа, старший (2k + 1)-й бит всегда равен нулю;
4.            [a0a1...ak] ← [d1d2...dk+1];
              comment [a0a1...ak] — хорошее приближение
                  к 2^(2k-1)/[p1p2...pk]. Следующий цикл уточняет это
                  приближение, прибавляя, если необходимо,
                  трехзначное число к младшим разрядам;
5.            for i ← 2 step -1 until 0 do
6.                if ([a0a1...ak] + 2^i) * [p1p2...pk] ≤ 2^(2k-1) then
7.                    [a0a1...ak] ← [a0a1...ak] + 2^i;
8.            return [a0a1...ak]
          end
```

Рис. 8.1. Процедура для обращения целых чисел

Пример 8.1. Вычислим $2^{15}/153$. Здесь

$$n = 8 \text{ и } 153 = [p_1 p_2 ... p_8] = [10011001].$$

Вызываем процедуру RECIPROCAL([10011001]), которая по очереди рекурсивно вызывает RECIPROCAL с аргументами [1001], [10] и [1]. В строке 1 находим RECIPROCAL[1] = [10] и возвращаемся к RECIPROCAL([10]) в строке 2, где полагаем $[c_0 c_1] \leftarrow [10]$. Затем в строке 3 вычисляем $[d_1 ... d_4] \leftarrow [10] * 2^3 - [10]^2 * [10] = [1000]$. Далее в строке 4 полагаем $[a_0 a_1 a_2] = [100]$. Цикл в строках 5–7 ничего не меняет. Возвращаемся к вызову RECIPROCAL([1001)) с [100] в качестве приближения к $2^3/[10]$.

В строке 2 при $k = 4$ имеем $[c_0 c_1 c_2] = [100]$. Тогда $[d_1 ... d_8] = [01110000]$ и $[a_0 ... a_4] = [01110]$. Снова цикл в строках 5–7 не приводит к изменениям. Вернемся к RECIPROCAL([10011001] в строке 2, причем $[c_0 ... c_4] = [01110]$. Далее

$$[d_1 ... d_{16}] = [0110101011011100].$$

Следовательно, в строке 4 $[a_1 ... a_8] = [011010101]$. В строке 6 находим, что $[011010101] * [10011001]$ (т.е. 213 * 153 в десятичной записи) равно 32589, тогда как $2^{15} = 32768$. Таким образом, цикл в строках 5–7 добавляет 1 к 213, что дает 214, или [011010110] в двоичной записи. □

Теорема 8.1. Алгоритм 8.1 находит такое число $[a_0 a_1 ... a_k]$, что

$$[a_0 a_1 ... a_k] * [p_1 p_2 ... p_k] = 2^{2k-1} - S$$

и $0 \le S < [p_1 p_2 ... p_k]$.

Доказательство. Проводится индукцией по k. База ($k = 1$) тривиальна в силу строки 1. Для проведения шага индукции положим $C = [c_0 c_1 ... c_{k/2}]$, $P_1 = [p_1 p_2 ... p_{k/2}]$ и $P_2 = [p_{k/2+1} p_{k/2+2} ... p_k]$. Тогда $P = [p_1 p_2 ... p_k] = P_1 2^{k/2} + P_2$. По предположению индукции

$$CP_1 = 2^{k-1} - S,$$

где $0 \le S < P_i$. Вследствие строки 3 число $D = [d_1 d_2 ... d_{2k}]$ определяется равенством

$$D = C 2^{3k/2} - C^2 (P_1 2^{k/2} + P_2). \tag{8.3}$$

Поскольку $p_1 = 1$, то $P_1 \ge 2^{k/2-1}$ и, следовательно, $C \le 2^{2/k}$. Отсюда следует, что $D < 2^{2k}$ и для представления числа D достаточно $2k$ битов.

Рассмотрим произведение $PD = (P_1 2^{k/2} + P_2) D$, равное

$$PD = CP_1 2^{2k} + CP_2 2^{3k/2} - (CP_1 2^{k/2} + CP_2)^2. \tag{8.4}$$

Подставив $2^{k-1} - S$ вместо CP_1 в (8.4) и сделав некоторые алгебраические упрощения, получаем

$$PD = 2^{3k-2} - (S 2^{k/2} - CP_2)^2. \tag{8.5}$$

Разделив (8.5) на 2^{k-1}, получаем

$$2^{2k-1} = \frac{PD}{2^{k-1}} + T, \tag{8.6}$$

где $T = (S2^{k/2} - CP_2)^2 2^{-(k-1)}$. По предположению индукции и в силу неравенства $P_1 < 2^{k/2}$ имеем $S < 2^{k/2}$. Поскольку $C \le 2^{k/2}$ и $P_2 < 2^{k/2}$, то $0 \le T < 2^{k+1}$.

В строке 4: $A = [a_0 a_1 ... a_k] = \lfloor D/2^{k-1} \rfloor$. Далее,

$$P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor > P\left(\frac{D}{2^{k-1}} - 1\right),$$

так что из (8.6) вытекает

$$2^{2k-1} \ge \frac{PD}{2^{k-1}} \ge P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor > \frac{PD}{2^{k-1}} - P = 2^{2k-1} - T - P >$$
$$> 2^{2k-1} - 2^{k+1} - 2^k.$$

Следовательно,

$$P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor = 2^{2k-1} - S',$$

где $0 \le S' < 2^{k+1} + 2^k$. Так как $P \ge 2^{k-1}$, то прибавление к $\lfloor D/2^{k-1} \rfloor$ числа, не превосходящего 6, дает число, удовлетворяющее предположению индукции для k. Поскольку эта инструкция выполняется в строках 5–7, шаг индукции выполнен. □

Теорема 8.2. Существует такая константа c, что $R(n) \le cM(n)$.

Доказательство. Достаточно показать, что временная сложность алгоритма 8.1 составляет $O_B(M(n))$. На строку 2 уходит $R(k/2)$ битовых операций. Строка 3 состоит из возведения в квадрат и умножения, что требует соответственно $M(k/2 + 1)$ и $M(k + 2)$ времени, а также вычитания, расходующего $O_B(k)$ времени. Заметим, что умножение на степени числа 2 не тратит битовых операций; мы считаем, что разряды сомножителей просто занимают новые положения, т.е. они сдвигаются. В силу нашего предположения, справедливо неравенство $M(k/2+1) \le \frac{1}{2} M(k+2)$. Кроме того, $M(k + 2) - M(k) = O_B(k)$ (см., например, раздел 2.6) и, значит, сложность строки 3 ограничена величиной $3/2M(k)+c'k$ для некоторой константы c'. Сложность строки 4 равна, очевидно, $O_B(k)$.

Может показаться, что цикл в строках 5–7 требует трех умножений, но можно выполнить необходимые вычисления за одно умножение $[a_0 a_1 ... a_k] * [p_1 p_2 ... p_k]$

и несколько сложений и вычитаний не более чем $2k$-разрядных двоичных целых чисел. Следовательно, сложность строк 5–7 ограничена величиной $M(k) + c''k$ для некоторой константы c''. Объединяя все эти оценки, приходим к выводу, что

$$R(k) \le R\left(\frac{k}{2}\right) + \frac{5}{2}M(k) + c_1 k \tag{8.7}$$

для некоторой константы c_1.

Мы утверждаем, что найдется такая константа c, что $R(k) \le cM(k)$. Константу c можно выбрать так, чтобы $c \ge R(1)/M(1)$ и $c \ge 5 + 2c_1$. Докажем это утверждение индукцией по k. База ($k = 1$) очевидна. Шаг индукции следует из (8.7), поскольку

$$R(k) \le cM\left(\frac{k}{2}\right) + \frac{5}{2}M(k) + c_1 k\,. \tag{8.8}$$

Поскольку $M(k/2) \le \frac{1}{2}M(k)$, в силу нашего предположения об M и очевидного неравенства $k \le M(k)$ можно переписать (8.8) в виде

$$R(k) \le \left(\frac{c}{2} + \frac{5}{2} + c_1\right)M(k). \tag{8.9}$$

Поскольку $c \ge 5 + 2c_1$, то из (8.9) следует, что $R(k) \le cM(k)$. □

Очевидно, что алгоритм 8.2 позволяет вычислить $1/P$ с n значащими битами, если P имеет столько же цифр, при этом неважно, где расположена запятая. Например, если $1/2 < P < 1$ и P содержит n битов, можно очевидным образом изменить масштаб и представить $1/P$ как 1 и последующие $n - 1$ битов дробной части.

Теперь покажем, что время $S(n)$, необходимое для возведения в квадрат целого числа размера n, не превышает по порядку времени $R(n)$ обращения целого числа размера n. Метод основан на тождестве

$$P^2 = \frac{1}{\frac{1}{P} - \frac{1}{P+1}} - P. \tag{8.10}$$

Приведем алгоритм, использующий тождество (8.10) с подходящим изменением масштаба.

Алгоритм 8.2. Возведение в квадрат с помощью обратных величин

Вход. n-разрядное целое число P в двоичной записи.

Выход. Двоичная запись числа P^2.

Метод.

1. Применим алгоритм 8.1 для вычисления $A = \lfloor 2^{4n-1}/P \rfloor$. Для этого добавим $2n$ нулей к P, применим процедуру RECIPROCAL[2] для вычисления $\lfloor 2^{6n-1}/P2^{2n} \rfloor$ и сдвигаем результат.
2. Аналогично вычисляем $B = \lfloor 2^{4n-1}/(P+1) \rfloor$.
3. Полагаем $C = A - B$. Заметим, что $C = 2^{4n-1}/(P^2 + P) + T$, где $|T| \le 1$. Слагаемое T возникает из-за того, что отбрасывание знаков при вычислении A и B может привести к ошибке вплоть до 1. Поскольку $2^{2n-2} < P^2 + P < 2^{2n}$, то $2^{2n+1} \ge C \ge 2^{2n-1}$.
4. Вычисляем $D = \lfloor 2^{4n-1}/C \rfloor - P$.
5. Пусть Q — четыре последних бита числа P. Увеличиваем или уменьшаем D на минимально возможную величину так, чтобы последние четыре бита результата совпали с последними четырьмя битами в числе Q^2. □

Теорема 8.3. Алгоритм 8.2 вычисляет P^2.

Доказательство. Вследствие усечения, выполняемого на этапах 1 и 2, можно гарантировать лишь то, что

$$C = \frac{2^{4n-1}}{P^2 + P} + T, \text{ где } |T| \le 1.$$

Поскольку $2^{2n-1} \le C \le 2^{2n+1}$ и ошибка в числе C заключена между -1 и 1, связанная с ней ошибка в $2^{4n-1}/C$ не превосходит

$$\left| \frac{2^{4n-1}}{C} - \frac{2^{4n-1}}{C-1} \right| = \left| \frac{2^{4n-1}}{C^2 - C} \right|.$$

Поскольку $C^2 - C \ge 2^{4n-3}$, то эта ошибка не превосходит 4. Отбрасывание цифр в строке 4 может увеличить ошибку до 5. Таким образом, $|P^2 - D| \le 5$. Следовательно, вычисление последних четырех битов в числе P^2 на шаге 5 гарантирует, что число D корректируется так, что становится равным P^2. □

Пример 8.2. Пусть $n = 4$ и $P = [1101]$. Тогда

$$A = \lfloor 2^{15}/[1101] \rfloor = [100111011000]$$

и

$$B = \lfloor 2^{15}/[1110] \rfloor = [100100100100].$$

---

[2]Процедура RECIPROCAL определена в алгоритме 8.1 только для целых чисел, длина которых равна степени числа 2. Обобщение на любые целые числа очевидно: добавляем нули и при необходимости изменяем масштаб.

Далее,

$$C = A - B = [10110100].$$

Тогда

$$D = [2^{15}/C] - P = [10110110] - [1101] = [10101001].$$

Таким образом, $D = 169$ — квадрат числа 13, и на шаге 5 не нужна никакая коррекция. □

Теорема 8.4. Найдется такая постоянная c, что $S(n) \le cR(n)$.

Доказательство. В алгоритме 8.2 трижды вычисляются обратные к числам, задаваемым строками длины не более $3n$. Вычитания на шагах 3 и 4 требуют $O_B(n)$ времени, а на шаге 5 выполняется работа фиксированного объема, не зависящего от n. Следовательно,

$$S(n) \le 3R(3n) + c_1 n \tag{8.11}$$

для некоторой константы c_1. Таким образом, $S(n) \le 27R(n) + c_1 n$. Поскольку $R(n) \ge n$, то, полагая $c = 27 + c_1$, завершаем доказательство теоремы. □

Теорема 8.5. Функции $M(n)$, $R(n)$, $D(n)$ и $S(n)$ равны с точностью до постоянных множителей.

Доказательство. Мы уже показали, что $R(n) \le c_1 M(n)$ и $S(n) \le c_2 R(n)$ для некоторых постоянных c_1 и c_2. Легко вывести неравенство $M(n) \le c_3 S(n)$, заметив, что

$$AB = \frac{1}{2}\left[(A+B)^2 - A^2 - B^2\right].$$

Следовательно, M, R и S равны с точностью до постоянных множителей.

Обсуждая деление n-разрядных двоичных чисел, мы имеем в виду деление числа, содержащего до $2n$ битов, на число, содержащее в точности n битов, причем ответ содержит не более $n + 1$ битов. Очевидно, что $R(n) \le D(n)$, так что $M(n) \le c_2 c_3 D(n)$. Более того, изменяя масштаб с помощью тождества $A/B = A * (1/B)$, можно показать, что для некоторой константы c выполняется неравенство

$$D(n) \le M(2n) + R(2n) + cn. \tag{8.12}$$

Поскольку $R(2n) \le c_1 M(2n)$ и, как легко показать, $M(2n) \le 4M(n)$, можно переписать (8.12) в виде

$$D(n) \le 4(1 + c_1)M(n) + cn. \tag{8.13}$$

Поскольку $M(n) \geq n$, то в силу (8.13) справедливо неравенство $D(n) \leq c_4 M(n)$, где $c_4 = 4 + 4c_1 + c$. Итак, мы доказали, что все рассматриваемые функции заключены между $dM(n)$ и $eM(n)$ для некоторых положительных постоянных d и e. □

Следствие. Деление $2n$-разрядного двоичного целого числа на n-разрядное можно выполнить за время $O_{\text{в}}(n \log n \log\log n)$.

Доказательство. Следует из теорем 7.8 и 8.5. □

8.3. Умножение и деление полиномов

Все методы предыдущего раздела переносятся на арифметические операции над полиномами от одной переменной. В этом разделе функции $M(n)$, $D(n)$, $R(n)$ и $S(n)$ обозначают время, требуемое соответственно для умножения, деления, обращения и возведения в квадрат полиномов степени n. Как и раньше, мы предполагаем, что $a^2 M(n) \geq M(an) \geq aM(n)$ для $a \geq 1$ и аналогичные неравенства справедливы и для других функций.

Обратным к полиному $p(x)$ степени n считается полином $\lfloor x^{2n}/p(x) \rfloor$.[3] $D(n)$ — это время вычисления полинома $\lfloor s(x)/p(x) \rfloor$, где $p(x)$ имеет степень n, а $s(x)$ — степень, не превосходящая $2n$. Заметим, что подобно тому, как в предыдущем разделе мы изменяли масштаб целых чисел с помощью степеней числа 2, масштаб полиномов можно изменять, умножая и деля их на степени переменной x.

Поскольку результаты настоящего раздела очень похожи на результаты, полученные для целых чисел, мы рассмотрим подробно только один из них: алгоритм обращения полиномов, аналогичный алгоритму 8.1 для целых чисел. Алгоритмы для полиномов в определенной степени проще соответствующих алгоритмов для целых чисел — в основном благодаря тому, что в степенных рядах в отличие от целых чисел нет переносов. Поэтому в алгоритмах для полиномов не надо корректировать младшие значащие биты, как это требовалось, например, в строках 5–7 алгоритма 8.1.

Алгоритм 8.3. Обращение полиномов

Вход. Полином $p(x)$ степени $n - 1$, где n — степень числа 2 (т.е. $p(x)$ имеет 2^i членов, где i — некоторое целое число.

Выход. Обратный полином $\lfloor x^{2n-2}/p(x) \rfloor$.

[3] По аналогии с обозначениями для целых чисел частное от деления полиномов обозначается с помощью функции округления с недостатком. Иначе говоря, если $p(x)$ не константа, то $\lfloor s(x)/p(x) \rfloor$ — это единственный полином $q(x)$, для которого $s(x) = p(x)q(x) + r(x)$ и $\text{DEG}(r(x)) < \text{DEG}(p(x))$

Метод. На рис. 8.2 приведена новая процедура

$$\texttt{RECIPROCAL}\left(\sum_{i=0}^{k-1} a_i x^i\right),$$

где k — степень числа 2 и $a_{k-1} \neq 0$. Эта процедура вычисляет

$$\left\lfloor \frac{x^{2k-2}}{\sum_{i=0}^{k-1} a_i x^i} \right\rfloor.$$

Заметим, что если $k = 1$, то аргумент является константой и обратное к нему число также является константой для вычисления $1/a_0$. Мы предполагаем, что каждая операция над коэффициентами выполняется за один шаг, и нет необходимости вызывать процедуру RECIPROCAL.

Сам алгоритм состоит в вызове процедуры RECIPROCAL с аргументом $p(x)$. □

```
      procedure RECIPROCAL(Σ_{i=0}^{k-1} a_i x^i);
1.    if k = 1 then return 1/a_0
      else
          begin
2.            q(x) ← RECIPROCAL(Σ_{i=k/2}^{k-1} a_i x^{i-k/2});
3.            r(x) ← 2q(x)x^{(3/2)k-2} - (q(x))^2 (Σ_{i=0}^{k-1} a_i x^i);
              return ⌊r(x)/x^{k-2}⌋
          end
```

Рис. 8.2. Процедура для обращения полиномов

Пример 8.3. Вычислим $\lfloor x^{14}/p(x) \rfloor$, где $p(x) = x^7 - x^6 + x^5 + 2x^4 - x^3 - 3x^2 + x + 4$. В строке 2 обращаем полином $x^3 - x^2 + x + 2$, т.е. находим $q(x) = \lfloor x^6/(x^3 - x^2 + x + 2) \rfloor$. Легко убедиться, что $q(x) = x^3 + x^2 - 3$. Поскольку $k = 8$, в строке 3 вычисляется $r(x) = 2q(x)x^{10} - (q(x))^2 p(x) = x^{13} + x^{12} - 3x^{10} - 4x^9 + 3x^8 + 15x^7 + 12x^6 - 42x^5 - 34x^4 + 39x^3 + 51x^2 - 9x - 36$. Затем в строке 4 получаем результат $s(x) = x^7 + x^6 - 3x^4 - 4x^3 + 3x^2 + 15x + 12$. Заметим, что полином $s(x)p(x)$ представляет собой сумму x^{14} и полинома шестой степени.

Теорема 8.6. Алгоритм 8.6 правильно вычисляет полином, обратный к данному.

Доказательство. Докажем индукцией по k, где k — степень числа 2, что если $s(x) = \texttt{RECIPROCAL}(p(x))$ и $p(x)$ имеет степень $k - 1$, то $s(x)p(x) = x^{2k-2} + t(x)$, где $t(x)$ — полином степени, меньшей $k - 1$. База ($k = 1$) очевидна, поскольку $p(x) = a_0$, $s(x) = 1/a_0$ и слагаемого $t(x)$ нет.

Для шага индукции положим $p(x) = p_1(x)x^{k/2} + p_2(x)$, где $\texttt{DEG}(p_1) = k/2 - 1$ и $\texttt{DEG}(p_2) \le k/2 - 1$. Тогда по предположению индукции, если $s_1(x) = \texttt{RECIPROCAL}(p_1(x))$, то $s_1(x)p_1(x) = x^{k-2} + t_1(x)$, где $\texttt{DEG}(t_1) < k/2 - 1$. В строке 3 вычисляем

$$r(x) = 2s_1(x)x^{(3/2)k-2} - (s_1(x))^2(p_1(x)x^{k/2} + p_2(x)). \tag{8.14}$$

Достаточно показать, что $r(x)p(x)$ равно сумме x^{3k-4} плюс члены, степень которых не превышает $2k - 3$. Тогда деление на x^{k-2} в строке 4 дает искомый результат.

Вследствие равенств (8.14) и $p(x) = p_1(x)x^{k/2} + p_2(x)$ имеем

$$r(x)p(x) = 2s_1(x)p_1(x)x^{2k-2} + 2s_1(x)p_2(x)x^{(3/2)k-2} - (s_1(x)p_1(x)x^{k/2} + s_1(x)p_2(x))^2. \tag{8.15}$$

Подставив в (8.15) $x^{k-2} + t_1(x)$ вместо $s_1(x)p_1(x)$, получим

$$r(x)p(x) = x^{3k-4} - (t_1(x)x^{k/2} + s_1(x)p_2(x))^2. \tag{8.16}$$

Поскольку $\texttt{DEG}(t_1) < k/2 - 1$ и степени полиномов $s_1(x)$ и $p_2(x)$ не превышают $k/2 - 1$, то степени всех членов в (8.16), отличных от x^{3k-4}, не превышают $2k - 4$.

Очевидно, что время работы алгоритмов 8.3 и 8.1 оценивается аналогично, если использовать две меры сложности (арифметическую и битовую соответственно). Аналогично можно доказать, что другие оценки временной сложности, полученные в разделе 8.2, переносятся на полиномы, если вместо битовых шагов рассматривать арифметические. Таким образом, верна следующая теорема.

Теорема 8.7. Пусть $M(n)$, $D(n)$, $R(n)$ и $S(n)$ — арифметическая сложность умножения, деления, обращения и возведения в квадрат полиномов от одной переменной соответственно. Все эти функции равны с точностью до постоянных множителей.

Доказательство. Аналогично доказательству теоремы 8.5 и результатов, из которых она следует. □

Следствие. Полином степени $2n$ можно разделить на полином степени n за время $O_A(n \log n)$.

Доказательство. Вытекает из теоремы 8.7 и следствия 3 теоремы 7.4. □

8.4. Модульная арифметика

В некоторых приложениях удобно выполнять арифметические операции над целыми числами в модульном представлении. Это значит, что вместо того, чтобы представлять целое число в системе счисления с фиксированным основанием, его представляют вычетами по модулям из множества попарно взаимно простых чисел. Если $p_0, p_1, ..., p_{k-1}$ — попарно взаимно простые числа и $p = \prod_{i=0}^{k-1} p_i$, то любое целое число и $0 \le u < p$ можно однозначно представить множеством его вычетов $u_0, u_1, ..., u_{k-1}$, где $u_i = u \bmod p_i$, где $0 < i < k$. Если $p_0, p_1, ..., p_{k-1}$ фиксированы, то пишут $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$.

Сложение, вычитание и умножение легко выполняются, если их результаты лежат в диапазоне от 0 до $p - 1$ (иначе говоря, если эти вычисления можно рассматривать как вычисления по модулю p). Пусть

$$u \leftrightarrow (u_0, u_1, ..., u_{k-1}) \text{ и } v \leftrightarrow (v_0, v_1, ..., v_{k-1}),$$

тогда

$$u + v \leftrightarrow (w_0, w_1, ..., w_{k-1}), \text{ где } w_i = (u_i + v_i) \bmod p_i, \tag{8.17}$$

$$u - v \leftrightarrow (x_0, x_1, ..., x_{k-1}), \text{ где } x_i = (u_i - v_i) \bmod p_i, \tag{8.18}$$

$$uv \leftrightarrow (y_0, y_1, ..., y_{k-1}), \text{ где } y_i = u_i v_i \bmod p_i. \tag{8.19}$$

Пример 8.4. Пусть $p_0 = 5, p_1 = 3$ и $p_2 = 2$. Тогда $4 \leftrightarrow (4, 1, 0)$, поскольку $4 = 4 \bmod 5$, $1 = 4 \bmod 3$ и $0 = 4 \bmod 2$. Аналогично $7 \leftrightarrow (2, 1, 1)$ и $28 \leftrightarrow (3, 1, 0)$. Заметим, что вследствие (8.19) $4 \times 7 \leftrightarrow (3, 1, 0)$, где $(3, 1, 0)$ — представление числа 28. Первый компонент произведения 4×7 равен $4 \times 2 \bmod 5$, т.е. 3; второй компонент равен $1 \times 1 \bmod 3$, т.е. 1; третий равен $0 \times 1 \bmod 2$, т.е. 0. Кроме того, $4 + 7 \leftrightarrow (1, 2, 1)$ (представление числа 11), $7 - 4 \leftrightarrow (3, 0, 1)$ (представление числа 3). □

Однако неясно, как в модульной арифметике экономно выполнять деление. Заметим, что отношение u/v может не быть целым числом, а если бы и было, то в общем случае нельзя найти его модульное представление, вычисляя u_i / v_i по модулю p_i для каждого i. Действительно, если p_i не является простым числом, то в диапазоне от 0 до $p - 1$ может оказаться несколько целых чисел w, равных u_i/v_i по модулю p_i в том смысле, что $wv_i \equiv u \bmod p_i$. Например, если $p_i = 6$, $v_i = 3$ и $u_i = 3$, то в качестве w можно было бы взять 1, 3 или 5, поскольку $1 \times 3 = 3 \times 3 = 5 \times 3 = 3 \bmod 6$. Поэтому $(u_i/v_i) \bmod p_i$ может не иметь смысла.

Преимущество модульного представления в основном заключается в том, что арифметические операции можно реализовать с меньшими аппаратными ресурсами, чем при обычном представлении, поскольку вычисления выполняются неза-

висимо для любого модуля. В отличие от обычного (позиционного) представления чисел здесь не нужны никакие переносы. К сожалению, проблемы эффективного деления и контроля переполнений (т.е. выхода результата за пределы диапазона от 0 до $p - 1$) оказываются непреодолимыми, и поэтому такие системы редко реализуются в компьютерном аппаратном обеспечении общего назначения.

Тем не менее излагаемые здесь идеи находят применение, главным образом, при изучении полиномов. Скорее всего, делить полиномы нам не придется. Кроме того, как мы увидим в следующем разделе, вычисление полиномов и их вычетов (по модулю других полиномов) тесно связаны. Сначала покажем, что модульная арифметика целых чисел работает так, как от нее ожидается.

Первая часть доказательства состоит в том, чтобы доказать, что выполняются соотношения (8.17)–(8.19). Этот факт очевиден, и мы оставляем его читателям в качестве упражнения. Вторая часть доказательства — показать, что соответствие $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$ взаимно однозначно (т.е. является изоморфизмом). Хотя этот результат несложен, сформулируем его в виде леммы.

Лемма 8.1. Пусть $p_0, p_1, ..., p_{k-1}$ — попарно взаимно простые $k - 1$ целые числа, $p = \prod_{i=0}^{k-1} p_i$ и $u_i = u \bmod p_i$. Тогда соответствие $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$ между целыми числами u в интервале $[0, p)$ и наборами вида

$$(u_0, u_1, ..., u_{k-1}),\ 0 \le u_i < p_i \text{ при } 0 \le i < k$$

взаимно однозначно.

Доказательство. Очевидно, что для каждого u найдется соответствующий kчленный кортеж. Поскольку в указанном интервале лежит ровно p значений переменной u и количество допустимых k-членных кортежей также равно p, достаточно показать, что каждый такой кортеж соответствует не более чем одному целому числу u. Допустим, что два числа u и v, $0 \le u < v < p$, соответствуют кортежу $(u_0, u_1, ..., u_{k-1})$. Тогда разность $v - u$ должна делиться на каждое число p_i. Поскольку все p_i являются попарно взаимно простыми, разность $v - u$ должна делиться и на p. Однако $u \ne v$ и $v - u$ делится на p, так что u и v должны отличаться не менее чем на p и, значит, не могут оба лежать между 0 и $p - 1$. □

Для того чтобы можно было пользоваться модульной арифметикой, необходимы алгоритмы, осуществляющие переход от позиционного представления к модульному и обратно. Один из методов перехода от позиционного представления целого числа и к его модульному представлению состоит в том, чтобы разделить u на каждое из чисел p_i, $0 \le i < k$.

Допустим, каждое из чисел p_i содержит b разрядов в двоичном представлении. Тогда для представления $p = \prod_{i=0}^{k-1} p_i$ требуется примерно bk битов, а деление u на каждое из чисел p_i, где $0 \le u < p$, могло бы потребовать k делений kb-битового числа на b-битовое число. Разбив каждое деление на k делений $2b$-битовых чисел на b-битовые, можно перейти к модульному представлению за время $O_{\text{В}}\left(k^2 D(b)\right)$, где $D(n)$ — время деления целых чисел (не превосходящее $O_{\text{В}}(n \log n \log\log n)$ в силу следствия теоремы 8.5).

Однако эту работу можно выполнить за значительно меньшее время, если применить метод, напоминающий метод деления полиномов из раздела 7.2. Вместо того чтобы делить число u на каждый из k модулей $p_0, p_1, \ldots, p_{k-1}$, сначала вычислим произведения $p_0p_1, p_2p_3, \ldots, p_{k-2}p_{k-1}$, затем $p_0p_1p_2p_3, p_4p_5p_6p_7\ldots$ и т.д. Далее вычисляем вычеты с помощью метода "разделяй и властвуй". Выполняя деление, получаем вычеты u_1 и u_2 числа u по модулям $p_0\ldots p_{k/2-1}$ и $p_{k/2}\ldots p_{k-1}$ соответственно. Теперь задача вычисления $u \bmod p_i$, $0 \le i < k$, сведена к двум подзадачам половинного размера, а именно $u \bmod p_i = u_1 \bmod p_i$ для $0 \le i < k/2$, и $u \bmod p_i = u_2 \bmod p_i$ для $k/2 \le i < k$.

Алгоритм 8.4. Вычисление вычетов

Вход. Модули $p_0, p_1, \ldots, p_{k-1}$ и такое целое число u, что $0 \le u < p = \prod_{i=0}^{k-1} p_i$.

Выход. Числа u_i, $0 \le i < k$, такие, что $u_i = u \bmod p_i$.

Метод. Допустим, что k — степень числа 2, например $k = 2^t$. (При необходимости добавим к входу лишние модули, равные 1, чтобы k степенью числа 2.) Начинаем с вычисления определенных произведений модулей, аналогичных полиномам q_{tm} из раздела 7.2.

Пусть $0 \le j < t$, число i кратно числу 2^j и $0 \le i < k$, положим

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m.$$

Таким образом, $q_{i0} = p_i$ и $q_{ij} = q_{i,j-1}q_{i+2^{j-1},j-1}$.

Сначала вычисляем числа q_{ij}, затем находим остатки u_{ij} от деления u на каждое из чисел q_{ij}. Искомым ответом будут числа u_{i0}. Детали приведены в программе на рис. 8.3. □

```
    begin
1.      for i ← 0 until k - 1 do q_i0 ← p_i;
2.      for j ← 1 until t - 1 do
3.          for i ← 0 step 2^j until k - 1 do
4.              q_ij ← q_i,j-1 * q_i+2^(j-1),j-1;
5.          u_0t ← u;
6.          for j ← t step -1 until 1 do
7.              for i ← 0 step 2^j until k - 1 do
                    begin
8.                      u_i,j-1 ← REMAINDER(u_ij / q_i,j-1);
9.                      u_i+2^(j-1),j-1 ← REMAINDER(u_ij / q_i+2^(j-1),j-1);
                    end;
10.             for i ← 0 until k - 1 do u_i ← u_i0
    end
```

Рис. 8.3. Вычисление вычетов

Теорема 8.8. Алгоритм 8.4 правильно вычисляет числа u_i.

Доказательство. Доказательство повторяет доказательство теоремы 7.3, где вычислялись значения полинома в корнях n-й степени из единицы. Легко показать индукцией по j, что строка 4 правильно вычисляет числа q_{ij}. Затем обратной индукцией по j в строке 6 доказываем, что $u_{ij} = u \bmod q_{ij}$. Строки 8 и 9 позволяют легко сделать это. Полагая $j = 0$, получаем $u_i = u \bmod p_i$. Детали оставляем читателям в качестве упражнения. □

Теорема 8.9. Алгоритм 8.4 требует $O_{\text{в}}\left(M\left(bk\right)\log k\right)$ времени, если для представления каждого из чисел p_i достаточно b битов.

Доказательство. Легко видеть, что больше всего времени требуют циклы в строках 3, 4 и 7–9. Каждый из них занимает $O_{\text{в}}(2^{t-j}M(2^{j-1}b))$ шагов.[4] Так как мы предположили, что $M(an) \geq aM(n)$ для $a \geq 1$, то сложность выполнения этих циклов ограничена величиной $O_{\text{в}}(M(2^t b)) = O_{\text{в}}(M(kb))$. Поскольку каждый из циклов повторяется не более $t = \log k$ раз, получаем нужный результат. □

Следствие. Если для представления каждого из модулей p_0, p_1, ..., p_{k-1} требуется b битов, то вычеты по этим модулям можно вычислить за время $O_{\text{в}}\left(bk \log k \log bk \log\log bk\right)$.

[4]Напомним, что функции $D(n)$ и $M(n)$, по существу, совпадают.

8.5. Модульная арифметика полиномов и вычисление их значений

Результаты, аналогичные результатам для целых чисел, справедливы и для полиномов. Пусть $p_0, p_1, ..., p_{k-1}$ — полиномы и $p = \prod_{i=0}^{k-1} p_i$. Тогда каждый полином u можно представить в виде последовательности $u_0, u_1, ..., u_{k-1}$ остатков от деления u на каждый полином p_i. Полином u_i — это единственный полином, для которого $\mathrm{DEG}(u_i) < \mathrm{DEG}(p_i)$ и $u = p_i q_i + u_i$ для некоторого полинома q_i. В такой ситуации мы будем писать $u_i = u \bmod p_i$, совершенно аналогично модульной записи целых чисел.

По аналогии с леммой 8.1 можно показать, что соответствие $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$ взаимно однозначно, если полиномы p_i попарно взаимно простые и степень полинома u меньше степени полинома p, т.е. $\mathrm{SIZE}(u) < \mathrm{SIZE}(p)$. Более того, алгоритм 8.4 для вычисления вычетов применим к полиномам, если в качестве p_i взять полиномы, а не целые числа. Вместо b (числа битов в каждом p_i) следует рассматривать максимальную степень полиномов p_i. Разумеется, сложность измеряется количеством арифметических, а не битовых операций. Тогда справедлива теорема, аналогичная теореме 8.9.

Теорема 8.10. Перенеся алгоритм 8.4 на полиномы, можно найти вычеты полинома $u(x)$ относительно полиномов $p_0, p_1, ..., p_{k-1}$ за время $O_A\left(M(dk)\log k\right)$, где d — верхняя граница степеней полиномов p_i и степень полинома u меньше степени полинома $\prod_{i=0}^{k-1} p_i$.

Доказательство. Аналогично теореме 8.9. Оставляем его читателям в качестве упражнения. □

Следствие 1. Для вычисления вычетов полинома u относительно полиномов $p_0, p_1, ..., p_{k-1}$ требуется не более $O_A\left(dk \log k \log dk\right)$ времени, где d — верхняя граница степеней полиномов p_i и степень полинома u меньше степени полинома $\prod_{i=0}^{k-1} p_i$.

Пример 8.5. Рассмотрим четыре полиномиальных модуля:

$$p_0 = x - 3,$$
$$p_1 = x^2 + x + 1,$$
$$p_2 = x^2 - 4,$$
$$p_3 = 2x + 2$$

и положим $u = x^5 + x^4 + x^3 + x^2 + x + 1$. Сначала вычислим произведения

$$p_0 p_1 = x^3 - 2x^2 - 2x - 3,$$
$$p_2 p_3 = 2x^3 + 2x^2 - 8x - 8.$$

Затем вычислим

$$u' = u \bmod p_0 p_1 = 28x^2 + 28x + 28,$$
$$u'' = u \bmod p_2 p_3 = 21x + 21.$$

Иначе говоря,

$$u' = p_0 p_1 (x^2 + 3x + 9) + 28x^2 + 28x + 28,$$
$$u'' = p_2 p_3 \left(\frac{1}{2}x^2 + \frac{5}{2}\right) + 21x + 21.$$

Далее вычислим

$$u \bmod p_0 = u' \bmod p_0 = 364,$$
$$u \bmod p_1 = u' \bmod p_1 = 0,$$
$$u \bmod p_2 = u'' \bmod p_2 = 21x + 21,$$
$$u \bmod p_3 = u'' \bmod p_3 = 0. \quad \square$$

Заметим, что алгоритм быстрого преобразования Фурье из раздела 7.2, по существу, является реализацией этого алгоритма, когда в качестве $p_0, p_1, \ldots, p_{k-1}$ выбираются $x - \omega^0$, $x - \omega^1$, ..., $x - \omega^{n-1}$. Алгоритм быстрого преобразования Фурье извлекает выгоду из выбора в качестве p_i полинома $x - \omega^i$. Благодаря подходящему упорядочению множества полиномов p_i каждое произведение равно разности между степенью x и степенью ω, поэтому деление выполняется особенно легко.

Как было отмечено в разделе 7.2, если p_i — полином первой степени $x - a$, то $u \bmod p_i = u(a)$. Следовательно, случай, когда все полиномы p_i имеют степень 1, особенно важен. Из теоремы 8.10 вытекает следующий результат.

Следствие 2. Значения полинома n-й степени в n точках можно вычислить за время $O_A\left(n \log^2 n\right)$.

Доказательство. Чтобы найти значения $u(x)$ в n точках $a_0, a_1, \ldots, a_{n-1}$, вычислим $u(x) \bmod (x - a_i)$ для $0 \le i < n$. Это занимает время $O_A\left(n \log^2 n\right)$ в силу следствия 1, поскольку здесь $d = 1$. $\square$

8.6. Применение китайской теоремы об остатках

Изучим задачу преобразования модульного представления целого числа в его позиционное представление.[5] Пусть даны попарно взаимно простые модули $p_0, p_1, ..., p_{k-1}$ и вычеты $u_0, u_1, ..., u_{k-1}$, где $k = 2^t$. Требуется найти такое целое число u, что $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$. Это можно сделать с помощью целочисленного аналога *интерполяционной формулы Лагранжа* для полиномов.

Лемма 8.2. Пусть c_i — произведение всех p_j, кроме p_i (т.е. $c_i = p/p_i$, где $p = \prod_{j=0}^{k-1} p_j$). Положим $d_i = c_i^{-1} \bmod p_i$ (т.е. $d_i c_i \equiv 1 \pmod{p_i}$ и $0 \le d_i < p_i$). Тогда

$$u \equiv \sum_{i=0}^{k-1} c_i d_i u_i \pmod{p}. \tag{8.20}$$

Доказательство. Поскольку числа p_j попарно взаимно просты, то, как известно, числа d_i существуют и являются единственными (упражнение 7.9). Кроме того, c_i делится на p_j при $j \neq i$, так что $c_i d_i u_i \equiv 0 \pmod{p_j}$ при $j \neq i$. Следовательно,

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv c_j d_j u_j \pmod{p_j}.$$

Поскольку $c_j d_j \equiv 1 \pmod{p_j}$, то

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv u_j \pmod{p_j}.$$

Так как p_j делит p, то эти соотношения выполняются даже тогда, когда все арифметические операции производятся по модулю p. Таким образом, равенство (8.20) справедливо. □

Наша задача состоит в эффективном вычислении формулы (8.20). Сразу трудно понять, как вычислить d_i по p_i, не используя метод проб и ошибок. Позже мы увидим, что эта задача не является сложной, если использовать алгоритм Евклида из раздела 8.8 и его быструю реализацию из раздела 8.10. В данном разделе мы изучим только ту форму китайской задачи об остатках, которая основана на предобусловливании. Если часть входных данных фиксирована для ряда задач, то все величины, зависящие только от этой фиксированной части, можно вычислить заранее и считать частью входа. Алгоритм, в котором сделаны такие предварительные вычисления, называют *предобусловленным*.

---

[5] Процесс восстановления числа по его остаткам был известен китайцам более 2000 лет назад, и потому соответствующую теорему называют китайской теоремой об остатках.

Для китайского алгоритма с предобусловливанием данных входом будут служить не только вычеты и модули p_i, но и обратные к ним числа d_i. Эта ситуация вполне реалистична. Если часто применять китайский алгоритм для перевода чисел, представленных вычетами по фиксированному множеству модулей, то разумно заранее вычислить те функции от этих модулей, значения которых используются в алгоритме. Следствие теоремы 8.21 утверждает, что на оценку сложности наличие или отсутствие предобусловливания влияет незначительно.

Анализируя (8.20), замечаем, что слагаемые $c_i d_i u_i$ при разных i имеют много общих сомножителей. Например, $c_i d_i u_i$ имеет сомножителем число $p_0 p_1 ... p_{k/2-1}$ если $i \geq k/2$, и число $p_{k/2} p_{k/2+1} ... p_{k-1}$, если $i < k/2$. Поэтому можно записать (8.20) в виде

$$u = \left(\sum_{i=0}^{k/2-1} c_i' d_i u_i \right) \prod_{i=k/2+1}^{k-1} p_i + \left(\sum_{i=k/2+1}^{k-1} c_i'' d_i u_i \right) \prod_{i=0}^{k/2-1} p_i,$$

где c_i' — произведение $p_0 p_1 ... p_{k/2-1}$ без p_i, а c_i'' — произведение $p_{k/2} p_{k/2+1} ... p_{k-i}$ без p_i. Это наблюдение подсказывает подход "разделяй и властвуй", аналогичный тому, который применялся при вычислении вычетов. Найдем произведения

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$$

(как в алгоритме 8.4), а затем — целые числа

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

Если $j = 0$, то $s_{i0} = d_i u_i$. Если $j > 0$, то вычисляем s_{ij} по формуле

$$s_{ij} = s_{i,j-1} q_{i+2^{j-1},j-1} + s_{i+2^{j-1},j-1} q_{i,j-1}.$$

Наконец, получаем $s_{i0} = u$, что и требовалось доказать.

Алгоритм 8.5. Быстрый предобусловленный китайский алгоритм

Вход.

1. Попарно взаимно простые целочисленные модули $p_0 p_1 ... p_{k-1}$, где $k = 2^t$ для некоторого t.
2. Множество обратных к ним чисел $d_0, d_1, ..., d_{k-1}$, таких что $d_i = (p/p_i)^{-1} \bmod p_i$, где $p = \prod_{i=0}^{k-1} p_i$.
3. Последовательность вычетов $(u_0, u_1, ..., u_{k-1})$.

Выход. Единственное целое число u, $0 \leq u < p$, для которого $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$.

Метод. Сначала вычисляем $q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$ как в алгоритме 8.4.[6] Затем выполняем программу, представленную на рис. 8.4, в предположении, что в ней

$$s_{ij} = \sum_{m=i}^{i+2^j-1} \frac{q_{ij} d_m u_m}{p_m}. \quad \square$$

```
      begin
1.        for i ← 0 until k - 1 do s_i0 ← d_i * u_i;
2.        for j ← 1 until t do
3.            for i ← 0 step 2^j until k - 1 do
4.                s_ij ← s_i,j-1 * q_i+2^(j-1),j-1 + s_i+2^(j-1),j-1 * q_i,j-1;
5.         write s_0t mod q_0t
       end
```

Рис. 8.4. Программа вычисления целого числа по его модульному представлению

Пример 8.6. Пусть p_0, p_1, p_2, p_3 равны соответственно 2, 3, 5, 7 и $(u_0, u_1, u_2, u_3) = (1, 2, 4, 3)$. Тогда $q_{i0} = p_i$ для $0 \le i \le 4$, $q_{01} = 6$, $q_{21} = 35$ и $q_{02} = p = 210$. Далее,

$$d_0 = (3 * 5 * 7)^{-1} \bmod 2 = 1, \text{ поскольку } 1 * 105 \equiv 1 \pmod 2$$
$$d_1 = (2 * 5 * 7)^{-1} \bmod 3 = 1, \text{ поскольку } 1 * 70 \equiv 1 \pmod 3,$$
$$d_2 = (2 * 3 * 7)^{-1} \bmod 5 = 3, \text{ поскольку } 3 * 42 \equiv 1 \pmod 5,$$
$$d_3 = (2 * 3 * 5)^{-1} \bmod 7 = 4, \text{ поскольку } 4 * 30 \equiv 1 \pmod 7.$$

Таким образом, в строке 1 вычисляются

$$s_{00} = 1 * 1 = 1, s_{10} = 1 * 2 = 2,$$
$$s_{02} = 3 * 4 = 12, s_{30} = 4 * 3 = 12.$$

Затем выполняется цикл в строках 3, 4 с $j = 1$. Здесь i принимает значения 0 и 2, следовательно,

$$s_{01} = s_{00} * q_{10} + s_{10} * q_{00} = 1 * 3 + 2 * 2 = 7,$$
$$s_{21} = s_{20} * q_{30} + s_{30} * q_{20} = 12 * 7 + 12 * 5 = 144.$$

Далее выполняется цикл в строках 3–4 при $j = 2$ и i, равном 0. Получаем

$$s_{02} = s_{01} * q_{21} + s_{21} * q_{01} = 7 * 35 + 144 * 6 = 1109.$$

[6] Заметим, что числа q_{ij} зависят только от чисел p_i. Можно включить их во входные данные, а не вычислять, поскольку допускается предобуславливание. Однако легко показать, что на порядок времени наличие или отсутствие предобуславливания не влияет.

Таким образом, результатом строки 5 будет 1109 mod 210, т.е. 59. Легко проверить, что вычеты числа 59 по модулям 2, 3, 5 и 7 равны 1, 2, 4 и 3 соответственно. На рис. 8.5 эти вычисления представлены графически. □

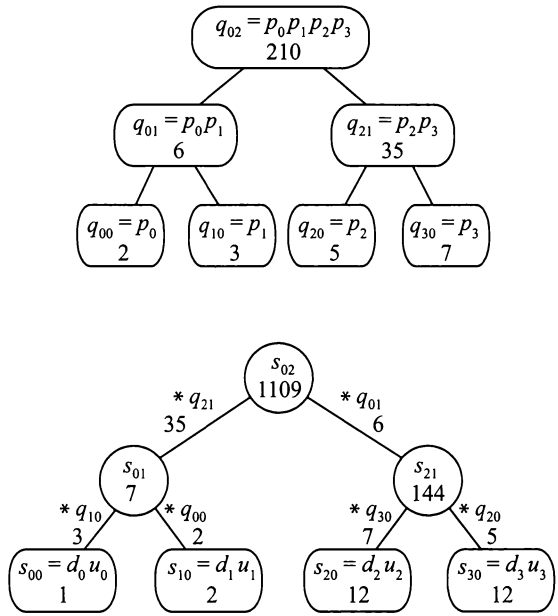


Рис. 8.5. Вычисления из примера 8.6

Теорема 8.11. Алгоритм 8.5 правильно вычисляет целое число u, для которого $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$.

Доказательство Элементарной индукцией по j можно доказать, что s_{ij} принимает правильное значение, т.е.

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

Правильность алгоритма непосредственно следует из леммы 8.2, т.е. из корректности формулы (8.20).

Теорема 8.12. Пусть даны k попарно взаимно простых целочисленных модулей $p_0, p_1, ..., p_{k-1}$ и вычеты $(u_0, u_1, ..., u_{k-1})$. Если каждое из чисел p_i содержит не более b битов, то существует предобусловленный алгоритм, вычисляющий число u, такое, что $0 \le u < p = \prod_{i=0}^{k-1} p_i$ и $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$ за время $O_{\text{в}}\left(M(bk)\log k\right)$, где $M(n)$ — время умножения двух n-битовых чисел.

Доказательство. Вычисление чисел q_{ij} требует $O_{\text{в}}\left(M(bk)\log k\right)$ времени.[7] Для анализа тела алгоритма заметим, что s_{ij} содержит не более $b2^j + b + j$ битов, поскольку является суммой 2^j слагаемых, каждое из которых равно произведению $2^j + 1$ целых чисел, состоящих не более чем из b битов. Следовательно, каждое слагаемое содержит не более $b(2^j + 1)$ битов, а сумма 2^j таких слагаемых — не более чем $b\left(2^j + 1\right) + \log 2^j = b2^j + b + j$ битов. Таким образом, строка 4 занимает время $O_{\text{в}}(M(b2^j))$. Цикл в строках 3-4 повторяется $k/2^j$ раз для фиксированного j, так что весь цикл выполняется за

$$O_{\text{в}}\left(\frac{k}{2^j} M\left(b2^j\right)\right)$$

шагов, а в силу обычного предположения о росте функции $M(n)$ эта величина не превосходит $O_{\text{в}}(M(bk))$. Поскольку цикл в строках 2–4 повторяется $\log k$ раз, то общая сложность составляет $O_{\text{в}}\left(M(bk)\log k\right)$ шагов. Нетрудно показать, что сложность строки 5 меньше этой величины. □

Следствие. Временная сложность китайского предобусловленного алгоритма с k модулями по b битов в каждом составляет $O_{\text{в}}\left(bk \log k \log bk \log\log bk\right)$.

8.7. Китайская теорема об остатках и интерполяция полиномов

Очевидно, что все результаты, приведенные в предыдущем разделе, справедливы не только для целочисленных, но и для полиномиальных модулей. Следовательно, выполняются следующая теорема и ее следствие.

Теорема 8.13. Пусть $p_0(x), p_1(x), \ldots, p_{k-1}(x)$ — полиномы степени не больше d и $M(n)$ — количество арифметических операций, необходимых для умножения двух полиномов степени n. Пусть $u_0(x), u_1(x), \ldots, u_{k-1}(x)$ — такие полиномы, что степень полинома $u_i(x)$ меньше степени полинома $p_i(x)$, $0 \le i < k$. Тогда существует предобусловленный алгоритм, вычисляющий за время $O_{\text{А}}\left(M(dk)\log k\right)$ единственный полином $u(x)$ степени $k - 1$, меньшей степени полинома $p(x) = \prod_{i=0}^{k-1} p_i(x)$, для которого

$$u(x) \leftrightarrow (u_0(x), u_1(x), \ldots, u_{k-1}(x)).$$

Доказательство. Аналогично алгоритму 8.5 и теореме 8.12.

---

[7] Поскольку $D(n)$ и $M(n)$ по существу равны, мы предпочитаем везде употреблять обозначение $M(n)$.

Следствие. Существует предобусловленный алгоритм восстановления полиномов по китайской теореме об остатках с временной сложностью $O_{\text{A}}\left(dk \log k \log dk\right)$.

Рассмотрим один важный частный случай, в котором все модули имеют степень 1. Если $p_i = x - a_i$ для $0 \le i < k$, то вычеты (числа u_i) являются константами, т.е. полиномами степени 0. Если $u(x) = u_i \bmod (x - a_i)$, то $u(x) = q(x)(x - a_i) + u_i$ и, значит, $u(a_i) = u_i$. Таким образом, единственным полиномом $u(x)$ степени, меньшей k, для которого $u(x) \leftrightarrow (u_0, u_1, ..., u_{k-1})$, будет тот единственный полином степени, меньшей k, для которого $u(a_i) = u_i$ для каждого i, $0 \le i < k$. Иначе говоря, $u(x)$ — это полином, интерполируемый по точкам (a_i, u_i) при $0 \le i < k$.

Поскольку при работе с полиномами интерполяция очень важна, с удовольствием отметим, что интерполяцию по значениям в k точках можно выполнить за время $O_{\text{A}}\left(k \log^2 k\right)$ даже без предобусловливания. Это объясняется тем, что в данном случае, как мы увидим из следующей леммы, значения коэффициентов d_i легко найти из равенства (8.20).[8]

Лемма 8.3. Пусть $p_1(x) = x - a_i$ для $0 \le i < k$, где все a_i различны (т.е. все $p_i(x)$ взаимно простые). Пусть $p(x) = \prod_{j=0}^{k-1} p_j(x)$, $c_i(x) = p(x)/p_i(x)$ и $d_i(x)$ — постоянный полином, причем $d_i(x)c(x) \equiv 1 \pmod{p_i(x)}$. Тогда $d_i(x) = 1/b$, где $b = \left.\frac{d}{dx} p(x)\right|_{x=a_i}$.

Доказательство. Запишем $p(x) = c_i(x)p_i(x)$, так что

$$\frac{d}{dx} p(x) = p_i(x) \frac{d}{dx} c_i(x) + c_i(x) \frac{d}{dx} p_i(x). \tag{8.21}$$

Далее, $dp_i(x)/dx = 1$ и $p_i(a_i) = 0$, поэтому

$$\left.\frac{dp(x)}{dx}\right|_{x=a_i} = c_i(a_i). \tag{8.22}$$

Заметим, что $d_i(x)c_i(x) \equiv 1 \bmod (x - a_i)$, и, значит, $d_i(x)c_i(x) = q_i(x)(x - a_i) + 1$ при некотором $q_i(x)$. Таким образом, $d_i(a_i) = 1/c_i(a_i)$. Теперь лемма немедленно следует из (8.22), поскольку $d_i(x)$ — константа. □

Теорема 8.14. Интерполяцию полинома по значениям в k точках можно выполнить за время $O_{\text{A}}\left(k \log^2 k\right)$ без предобусловливания.

---

[8] Как упоминалось в разделе 8.6, эта задача в общем случае несложна, однако для ее решения требуются методы из следующего раздела.

Доказательство. Вследствие леммы 8.3 вычисление полиномов d_i эквивалентно вычислению значений производной некоторого полинома степени $k-1$ в k точках. Полином $p(x)=\prod_{i=0}^{k-1} p_i(x)$ можно получить за время $O_A\left(k\log^2 k\right)$, если сначала вычислить p_0p_1, p_2p_3, ..., а затем $p_0p_1p_2p_3, p_4p_5p_6p_7$, ... и т.д. Производную полинома $p(x)$ можно найти за $O_A(k)$ шагов. В силу следствия 2 теоремы 8.1 вычисление значений производной занимает $O_A\left(k\log^2 k\right)$ времени. Таким образом, теорема вытекает из следствия теоремы 8.13 при $d=1$. □

Пример 8.7. Проинтерполируем полином по точкам (1, 2), (2, 7), (3, 4) и (4, 8). Здесь $a_i=i+1$ для $0\le i<4$, $u_0=2$, $u_1=7$, $u_2=4$ и $u_3=8$. Тогда $p_i(x)=x-i-1$, а полином $p(x)=\prod_{i=0}^{3} p_i(x)$ равен $x^4-10x^3+35x^2-50x+24$. Далее, $dp(x)/dx=4x^3-30x^2+70x-50$, и в точках 1, 2, 3, 4 этот полином принимает соответственно значения -6, 2, -2, 6. Таким образом, d_0, d_1, d_2 и d_3 равны соответственно $-1/6$, $1/2$, $-1/2$ и $1/6$. С помощью быстрого китайского алгоритма 8.5, настроенного на полиномы, получаем

$$s_{01}=d_0u_0p_1+d_1u_1p_0=-\frac{1}{6}\cdot 2\cdot(x-2)+\frac{1}{2}\cdot 7\cdot(x-1)=\frac{19}{6}x-\frac{17}{6},$$

$$s_{21}=d_2u_2p_3+d_3u_3p_2=-\frac{1}{2}\cdot 4\cdot(x-4)+\frac{1}{6}\cdot 8\cdot(x-3)=-\frac{2}{3}x+4,$$

$$s_{02}=u(x)=s_{01}q_{21}+s_{21}q_{01}=$$

$$=\left(\frac{19}{6}x-\frac{17}{6}\right)\left(x^2-7x-12\right)+\left(-\frac{2}{3}x+4\right)\left(x^2-3x+2\right),$$

$$u(x)=\frac{5}{2}x^3-19x^2+\frac{89}{2}x-26.$$

Как отмечалось в главе 7, арифметические операции над полиномами, например сложение, вычитание и умножение, можно выполнять, вычисляя полиномы в n точках, производя над найденными значениями арифметические операции и затем интерполируя полином по полученным в результате значениям. Если на выходе должен быть полином степени не более $n-1$, то мы получим правильный ответ.

Метод быстрого преобразования Фурье делает именно это, когда в качестве n точек выбираются числа ω^0, ω^1, ..., ω^{n-1}. Здесь алгоритмы вычисления значений и интерполяции оказались особенно простыми благодаря свойствам степеней числа ω и их специального упорядочения. Однако стоит подчеркнуть, что вместо

степеней числа ω можно было бы взять любой набор точек. Тогда мы получили преобразование, при котором его вычисление и обратное преобразование требуют $O_A\left(n\log^2 n\right)$, а не $O_A\left(n\log n\right)$.

8.8. Наибольший общий делитель и алгоритм Евклида

Определение. Пусть a_0 и a_1 — положительные целые числа. Положительное целое число g называется *наибольшим общим делителем* чисел a_0 и a_1 (его часто обозначают через $\mathrm{GCD}(a_0, a_1)$), если

1) g делит a_0 и a_1;

2) каждый общий делитель чисел a_0 и a_1 делит g.

Легко показать, что для положительных целых чисел a_0 и a_1 такое число g является единственным. Например, $\mathrm{GCD}(57, 33) = 3$.

Алгоритм Евклида для вычисления $\mathrm{GCD}(a_0, a_1)$ состоит в вычислении *последовательности остатков* $a_0, a_1, \ldots, a_k$, где a_i, $2 \le i \le k$ — ненулевой остаток от деления a_{i-2} на a_{i-1} и a_k нацело делит a_{k-1} (т.е. $a_{k+1} = 0$). В этой ситуации $\mathrm{GCD}(a_0, a_1) = a_k$.

Пример 8.8. В примере, приведенном выше, $a_0 = 57$, $a_1 = 33$, $a_2 = 24$, $a_3 = 9$, $a_4 = 6$ и $a_5 = 3$. Поэтому $k = 5$ и $\mathrm{GCD}(57, 33) = 3$. □

Теорема 8.15. Алгоритм Евклида правильно находит значение $\mathrm{GCD}(a_0, a_1)$.

Доказательство. Этот алгоритм вычисляет $a_{i+1} = a_{i-1} - q_i a_i$ для $1 \le i < k$, где $q_i = \lfloor a_{i-1}/a_i \rfloor$. Поскольку $a_{i+1} < a_i$ при $i \ge 1$, то алгоритм, очевидно, завершает работу. Кроме того, любой общий делитель чисел a_{i-1} и a_i делит a_{i+1} и любой общий делитель чисел a_i и a_{i+1} делит a_{i-1}. Следовательно, $\mathrm{GCD}(a_0, a_1) = \mathrm{GCD}(a_1, a_2) = \ldots = \mathrm{GCD}(a_{k-1}, a_k)$. Очевидно, $\mathrm{GCD}(a_{k-1}, a_k) = a_k$, так что теорема доказана. □

Алгоритм Евклида можно расширить так, чтобы он находил не только наибольший общий делитель чисел a_0 и a_1, но также и целые числа x и y, для которых $a_0x + a_1y = \mathrm{GCD}(a_0, a_1)$. Сформулируем этот алгоритм.

Алгоритм 8.6. Расширенный алгоритм Евклида

Вход. Положительные целые числа a_0 и a_1.

Выход. $\mathrm{GCD}(a_0, a_1)$ и такие целые числа x и y, что $a_0x + a_1y = \mathrm{GCD}(a_0, a_1)$.

Метод. Применяем программу, приведенную на рис. 8.6. □

```
      begin
1.        x_0 ← 1; y_0 ← 0; x_1 ← 0; y_1 ← 1; i ← 1;
2.        while a_i не делит a_{i-1} do
              begin
3.                q ← ⌊a_{i-1}/a_i⌋;
4.                a_{i+1} ← a_{i-1} - q * a_i;
5.                x_{i+1} ← x_{i-1} - q * x_i;
6.                y_{i+1} ← y_{i-1} ← q * y_i;
7.                i ← i + 1
              end;
8.        write a_i; write x_i; write y_i
      end
```

Рис. 8.6. Расширенный алгоритм Евклида

Пример 8.9. Если $a_0 = 57$ и $a_1 = 33$, то для чисел a_i, x_i, и y_i получаем значения

i	a_i	x_i	y_i
0	57	1	0
1	33	0	1
2	24	1	−1
3	9	−1	2
4	6	3	−5
5	3	−4	7

Заметим, что $57 \cdot (-4) + 33 \cdot 7 = 3$. □

Очевидно, что алгоритм 8.6 правильно вычисляет $\mathrm{GCD}(a_0, a_1)$, поскольку числа a_i образуют искомую последовательность остатков. В следующей лемме описывается важное свойство чисел x_i и y_i, вычисляемых алгоритмом 8.6.

Лемма 8.4. В алгоритме 8.6

$$a_0 x_i + a_1 y_i = a_i \text{ при } i \geq 0. \tag{8.23}$$

Доказательство. Равенство (8.23) справедливо для $i = 0$ и $i = 1$ в силу строки 1 алгоритма 8.6. Допустим, что (8.23) справедливо для $i - 1$ и i. Тогда $x_{i+1} = x_{i-1} - qx_i$ в силу строки 5 и $y_{i+1} = y_{i-1} - qy_i$ в силу строки 6. Следовательно,

$$a_0 x_{i+1} + a_1 y_{i+1} = a_0 x_{i-1} + a_1 y_{i-1} - q(a_0 x_i + a_1 y_i). \tag{8.24}$$

По предположению индукции

$$a_0 x_{i+1} + a_1 y_{i+1} = a_{i-1} - qa_i.$$

Так как $a_{i-1} - qa_i = a_{i+1}$ в силу строки 4, то лемма доказана. □

Заметим, что лемма 8.4 даже не зависит от способа вычисления q в строке, хотя строка существенна, чтобы гарантировать, что алгоритм 8.6 действительно вычисляет $\mathrm{GCD}(a_0, a_1)$. Объединим эти замечания и докажем следующую теорему.

Теорема 8.16. Алгоритм 8.6 вычисляет $\mathrm{GCD}(a_0, a_1)$ и такие числа x и y, что $a_0x + a_1y = \mathrm{GCD}(a_0, a_1)$.

Доказательство. Элементарное следствие леммы 8.4. □

Введем некоторые обозначения, которые понадобятся нам в следующем разделе.

Определение. Пусть a_0 и a_i — целые числа и $a_0, a_1, \ldots, a_k$ —соответствующая последовательность остатков. Пусть $q_i = \lfloor a_{i-1}/a_i \rfloor$, $1 \le i \le k$. Определим матрицы $R_{ij}^{(a_0,a_1)}$ или R_{ij} размерностью 2×2, где a_0 и a_1 при $0 \le i \le j \le k$ определяются условиями

1. $R_{ii} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ при $i \ge 0$.
2. Если $j > i$, то $R_{ij} = \begin{bmatrix} 1 & 0 \\ 0 & -q_j \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -q_{i-1} \end{bmatrix} * \ldots * \begin{bmatrix} 0 & 1 \\ 1 & -q_{i+1} \end{bmatrix}$.

Пример 8.10. Пусть $a_0 = 57$ и $a_1 = 33$, последовательность остатков имеет вид 57, 33, 24, 9, 6, 3, а частные q_i, $1 \le i \le 4$ равны 1, 1, 2, 1. Тогда

$$R_{04}^{(57,33)} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 3 & -5 \\ -4 & 7 \end{bmatrix}. \quad \square$$

Два интересных свойства этих матриц описаны в следующей лемме.

Лемма 8.5.

1. $\begin{bmatrix} a_j \\ a_{j+1} \end{bmatrix} = R_{ij} \begin{bmatrix} a_i \\ a_{i+1} \end{bmatrix}$ при $i < j < k$,
2. $R_{0j} = \begin{bmatrix} x_j & y_j \\ x_{j+1} & y_{j+1} \end{bmatrix}$ при $0 \le j < k$,

где числа a_i, x_i и y_i определяются расширенным алгоритмом Евклида.

Доказательство. Элементарное упражнение на использование индукции. □

Заметим, что все результаты этого раздела относятся и к полиномам от одной переменной. Надо лишь учесть, что, в то время как наибольший общий делитель двух целых чисел определяется однозначно, для полиномов с коэффициентами из некоторого поля наибольший общий делитель является единственным только с точностью до умножения на элемент поля. Иначе говоря, если $g(x)$ делит полиномы $a_0(x)$ и $a_1(x)$ и всякий другой их делитель тоже делит $g(x)$, то полином $cg(x)$, где c — ненулевая константа, также обладает этим свойством. Нас удовлетворит любой полином, который делит $a_0(x)$ и $a_1(x)$ и делится на любой их делитель. Чтобы обеспечить единственность, мы могли бы (но не будем) требовать, чтобы наибольший общий делитель был *нормированным*, т.е. чтобы коэффициент при его старшем члене был равен 1.

8.9. Асимптотически быстрый алгоритм нахождения наибольшего общего делителя полиномов

При построении алгоритмов для нахождения наибольшего общего делителя мы меняем нашу схему и рассматриваем сначала случай полиномов, поскольку в случае целых чисел приходится решать дополнительные задачи. Пусть $a_0(x)$ и $a_1(x)$ — два полинома, наибольший общий делитель которых необходимо вычислить. Мы предполагаем, что $\mathrm{DEG}(a_1(x)) < \mathrm{DEG}(a_0(x))$. Это условие легко выполнить, заменив полиномы a_0 и a_1 полиномами a_1 и $a_0 \bmod a_1$, если $\mathrm{DEG}(a_0) = \mathrm{DEG}(a_1)$, т.е. вторым и третьим членами последовательности остатков.

Разобьем нашу задачу на две части. Первая состоит в построении алгоритма, вычисляющего последний член последовательности остатков, степень которого больше половины степени полинома a_0. Формально пусть $l(i)$ — это единственное целое число, для которого $\mathrm{DEG}(a_{l(i)}) > i$ и $\mathrm{DEG}(a_{l(i)+1}) \le i$. Заметим, что если a_0 имеет степень n, то $l(i) \le n - i - 1$ в предположении, что $\mathrm{DEG}(a_1) < \mathrm{DEG}(a_0)$, поскольку $\mathrm{DEG}(a_i) \le \mathrm{DEG}(a_{i-1}) - 1$ для всех $i \ge 1$.

Введем рекурсивную процедуру HGCD (half GCD), которая по полиномам a_0 и a_1, таким, что $\mathrm{DEG}(a_0) > \mathrm{DEG}(a_1)$, формирует матрицу R_{0j} (см. раздел 8.8), где $j = l(n/2)$, т.е. a_j — последний член последовательности остатков, степень которого больше половины степени полинома a_0.

В основе алгоритма HGCD лежит тот факт, что частные от деления полиномов степеней d_1 и d_2, $d_1 > d_2$, зависят только от $2(d_1 - d_2) + 1$ старших членов делимого и $d_1 - d_2 + 1$ старших членов делителя. Процедура HGCD приведена на рис. 8.7.

```
      procedure HGCD(a_0, a_1):
1.        if DEG(a_1) ≤ DEG(a_0)/2 then return [1 0; 0 1]
          else
              begin
2.                пусть a_0 = b_0 x^m + c_0, m = ⌊DEG(a_0)/2⌋ и DEG(c_0) < m;
3.                пусть a_1 = b_1 x^m + c_1, где DEG(c_1) < m;
                  comment b_0 и b_1 — старшие члены полиномов a_0 и a_1.
                      Имеем DEG(b_0) = ⌈DEG(a_0)/2⌉ и
                      DEG(b_0) - DEG(b_1) = DEG(a_0) - DEG(a_1);
4.                R ← HGCD(b_0, b_1);
5.                [d; e] ← R [a_0; a_1];
6.                f ← d mod e;
                  comment e и f — соседние члены последовательности
                      остатков, их степень не превосходит ⌈3m/2⌉,
                      т.е. 3/4 степени полинома a_0;
7.                пусть e = g_0 x^⌊m/2⌋ + h_0, где DEG(h_0) < ⌊m/2⌋;
8.                пусть f = g_1 x^⌊m/2⌋ + h_1, где DEG(h_1) < ⌊m/2⌋;
                  comment степени полиномов g_0 и g_1 не превосходят
                      m + 1;
9.                S ← HGCD(g_0, g_1);
10.               q ← ⌊d/e⌋;
11.               return S * [0 1; 1 -q] * R
              end
```

Рис. 8.7. Процедура HGCD

Пример 8.11. Пусть

$$p_1(x) = x^5 + x^4 + x^3 + x^2 + x + 1$$
$$p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7.$$

Допустим, мы пытаемся вычислить HGCD(p_1, p_2). Если $a_0 = p_1$ и $a_1 = p_2$, то в строках 2 и 3 $m = 2$ и

$$b_0 = x^3 + x^2 + x + 1, \qquad c_0 = x + 1,$$
$$b_1 = x^2 - 2x + 3, \qquad c_1 = -x - 7.$$

Затем вызываем HGCD(b_0, b_1) в строке 4. Легко убедиться, что на этом шаге

$$R = \begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix}.$$

В строках 5 и 6 вычисляем

$$d = x^4 - 2x^3 + 3x^2 - x - 7,$$
$$e = 4x^3 - 7x^2 + 11x + 22,$$
$$f = -\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}.$$

Находим, что $\lfloor m/2 \rfloor = 1$ и, значит, в строках 7 и 8

$$g_0 = 4x^2 - 7x + 11, \qquad h_0 = 22,$$
$$g_1 = -\frac{3}{16}x - \frac{93}{16}, \qquad h_1 = -\frac{45}{8}.$$

Таким образом, в строке 9 получаем

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

В строке 10 обнаруживаем, что $q(x)$, т.е. частное $\lfloor d(x)/e(x) \rfloor$, равно $\frac{1}{4}x - \frac{1}{16}$.[9] Следовательно, в строке 11 получаем результат

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -\left(\frac{1}{4}x - \frac{1}{16}\right) \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix} = \begin{bmatrix} 1 & -(x+3) \\ -\left(\frac{1}{4}x - \frac{1}{16}\right) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Заметим, что

$$T \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix},$$

поскольку в последовательности остатков p_1 и p_2 член e является последним полиномом, степень которого больше половины степени p_1. □

Рассмотрим матрицу R, вычисляемую в строке 4 процедуры HGCD. Предположительно, матрица R равна

$$\prod_{j=1}^{l(\lceil m/2 \rceil)} \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix},$$

[9] Это вычисление, конечно, можно было бы выполнить сразу после строки 5.

где $q_j(x)$ — j-е частное из последовательности остатков для b_0 и b_1, т.е. $R = R_{0,l(\lceil m/2 \rceil)}^{(b_0;b_1)}$. Однако в строке 5 мы использовали R как матрицу $R_{0,l(\lceil 3m/2 \rceil)}^{(a_0;a_1)}$, чтобы получить d и e, где d — последний член последовательности остатков, степень которого больше $3m/2$. Необходимо показать, что обе эти интерпретации матрицы R корректны, т.е. $R_{0,l(\lceil 3m/2 \rceil)}^{(a_0;a_1)} = R_{0,l(\lceil m/2 \rceil)}^{(b_0;b_1)}$. Кроме того, мы должны показать, что матрица S, вычисляемая в строке 9, подходит для назначенной ей роли, т.е.

$$S = R_{0,l(\lceil m/2 \rceil)}^{(g_0,g_1)} = R_{0,l(m)}^{(e,f)}.$$

Эти результаты вытекают из следующих лемм.

Лемма 8.6. Пусть

$$f(x) = f_1(x)x^k + f_2(x), \tag{8.25}$$

где $\mathrm{DEG}(f_2) < k$ и

$$g(x) = g_1(x)x^k + g_2(x), \tag{8.26}$$

где $\mathrm{DEG}(g_2) < k$. Пусть q и q_1 — частные $\lfloor f(x)/g(x) \rfloor$ и $\lfloor f_1(x)/g_1(x) \rfloor$, а r и r_1 — остатки $f(x) - q(x)g(x)$ и $f_1(x) - q_1(x)g_1(x)$ соответственно. Если $\mathrm{DEG}(f) > \mathrm{DEG}(g)$ и $k \le 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$ (т.е. $\mathrm{DEG}(g_1) \ge \frac{1}{2}\mathrm{DEG}(f_1)$), то

а) $q(x) = q_1(x)$,

б) в полиномах $r(x)$ и $r_1(x)x^k$ все члены степени $k + \mathrm{DEG}(f) - \mathrm{DEG}(g)$ и выше совпадают.

Доказательство. Рассмотрим процесс деления $f(x)$ на $g(x)$ с помощью обычного алгоритма, который делит первый член полинома $f(x)$ на первый член полинома $g(x)$, чтобы получить первый член частного. Первый член частного умножается на $g(x)$ и вычитается из $f(x)$ и т.д. Первые $\mathrm{DEG}(g) - k$ членов, полученных таким способом, не зависят от $g_2(x)$. Однако частное содержит лишь члены степени $\mathrm{DEG}(f) - \mathrm{DEG}(g)$. Поэтому если $\mathrm{DEG}(f) - \mathrm{DEG}(g) \le \mathrm{DEG}(g) - k$, т.е. $k \le 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$, то частное не зависит от $g_2(x)$. Если $\mathrm{DEG}(f) - \mathrm{DEG}(g) \le \mathrm{DEG}(f) - k$, то частное не зависит от $f_2(x)$. Но неравенство $\mathrm{DEG}(f) - \mathrm{DEG}(g) \le \mathrm{DEG}(f) - k$ следует из неравенств $k \le 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$ и $\mathrm{DEG}(f) > \mathrm{DEG}(g)$. Таким образом, утверждение (*а*) доказано. Что касается утверждения (*б*), то аналогичные рассуждения показывают, что члены остатка, имеющие степень $\mathrm{DEG}(f) - (\mathrm{DEG}(g) - k)$ и выше, не зависят от $g_2(x)$. Аналогично члены остатка, имеющие степень k и выше, не зависят от $f_2(x)$. Однако $\mathrm{DEG}(f) - \mathrm{DEG}(g) + k > k$. Таким образом, в $r(x)$ и $r_1(x)x^k$ все члены степени $\mathrm{DEG}(f) - \mathrm{DEG}(g) + k$ и выше совпадают. □

Лемма 8.7. Пусть $f(x) = f_1(x)x^k + f_2(x)$ и $g(x) = g_1(x)x^k + g_2(x)$, где $\mathrm{DEG}(f_2) < k$ и $\mathrm{DEG}(g_2) < k$. Пусть $\mathrm{DEG}(f) = n$ и $\mathrm{DEG}(g) < \mathrm{DEG}(f)$. Тогда

$$R^{(f,g)}_{0,l(\lceil (n+k)/2 \rceil)} = R^{(f_1,g_1)}_{0,l(\lceil (n-k)/2 \rceil)},$$

т.е. частные, входящие в последовательности остатков для (f, g) и (f_1, g_1), совпадают, по крайней мере, до того места, где во второй последовательности появляется остаток степени, не большей половины степени полинома f_1.

Доказательство. Лемма 8.6 гарантирует совпадение частных и достаточного количества старших членов в соответствующих остатках, входящих в рассматриваемые последовательности. □

Теорема 8.17. Пусть $a_0(x)$ и $a_1(x)$ — такие полиномы, что $\mathrm{DEG}(a_0) = n$ и $\mathrm{DEG}(a_1) < n$. Тогда $\mathrm{HGCD}(a_0, a_1) = R_{0,\, l(n/2)}$.

Доказательство. Теорема доказывается простой индукцией по n. Из леммы 8.7 следует, что матрица R в строке 4 равна $R^{(a_0;a_1)}_{0,l(\lceil 3m/2 \rceil)}$, а матрица S в строке 9 равна $R^{(a_0;a_1)}_{l(\lceil 3m/2 \rceil)+1,l(m)}$. □

Теорема 8.18. Процедура HGCD выполняется за $O_{\mathrm{A}}\left(M(n)\log n\right)$ шагов, если ее аргументы имеют степени не выше n; здесь $M(n)$ — время умножения двух полиномов степени n.

Доказательство. Докажем теорему для случая, когда n — степень числа 4. Поскольку время выполнения процедуры HGCD, очевидно, не убывает с ростом n, то наша теорема будет тогда верна для всех n. Если $\mathrm{DEG}(a_0)$ — степень числа 4, то

$$\mathrm{DEG}(b_0) = \frac{1}{2}\mathrm{DEG}(a_0), \quad \mathrm{DEG}(g_0) \le \frac{1}{2}\mathrm{DEG}(a_0).$$

Таким образом, время $T(n)$ выполнения процедуры HGCD для аргумента степени n удовлетворяет условию

$$T(n) \le 2T\left(\frac{n}{2}\right) + cM(n) \tag{8.27}$$

для некоторой константы c. Иначе говоря, тело процедуры HGCD включает в себя два вызова той же процедуры для аргументов половинного размера и постоянное количество других операций с временной сложностью $O_{\mathrm{A}}(n)$ или $O_{\mathrm{A}}(M(n))$. Решение неравенства (8.27) должно быть известно читателю; оно ограничено сверху функцией $c_1 M(n)\log n$ для некоторой константы c_1. □

Перейдем к изложению полного алгоритма нахождения наибольших делителей. В нем участвует процедура HGCD, вычисляющая $R_{0,n/2}$, затем $R_{0,3n/4}$, $R_{0,7n/8}$ и т.д., где n — степень входа.

Алгоритм 8.7. Алгоритм GCD

Вход. Полиномы $p_1(x)$ и $p_2(x)$, для которых $\mathrm{DEG}(p_2) < \mathrm{DEG}(p_1)$.

Выход. Наибольший общий делитель $\mathrm{GCD}(p_1, p_2)$ для p_1 и p_2.

Метод. Вызываем процедуру $\mathrm{GCD}(p_1, p_2)$, где GCD — рекурсивная процедура, приведенная на рис. 8.8. □

```
      procedure GCD(a0, a1)
1.        if a1 делит a0 then return a1
          else
              begin
2.                R ← HGCD(a0, a1);
3.                [b0; b1] ← R * [a0; a1];
4.                if b1 делит b0 then return b1
                  else
                      begin
5.                        c ← b0 mod b1;
6.                        return GCD(b1, c)
                      end
              end
```

Рис. 8.8. Процедура GCD

Пример 8.12. Продолжим пример 8.11, где $p_1(x) = x^5 + x^4 + x^3 + x^2 + x + 1$ и $p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7$. Мы уже нашли

$$\mathrm{HGCD}(p_1, p_2) = \begin{bmatrix} 1 & -(x+3) \\ -\left(\frac{1}{4}x - \frac{1}{16}\right) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Следовательно, в строке 3 получаем

$$b_0 = 4x^3 - 7x^2 + 11x + 22,$$
$$b_1 = -\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}.$$

Оказывается, b_1 не делит b_0. В строке 5 находим, что

$$b_0 \bmod b_1 = 3952x + 3952.$$

Поскольку последний полином делит b_1, то вызов процедуры GCD в строке 6 завершает работу в строке 1 и выдает в качестве ответа $3952x + 3952$. Разумеется, $x + 1$ также является наибольшим общим делителем для p_1 и p_2. □

Доказательство корректности алгоритма 8.7 тривиально, если доказать, что он всегда заканчивает свою работу. Таким образом, корректность алгоритма вытекает из анализа времени его работы, что составляет содержание следующей теоремы.

Теорема 8.19. Если $\mathrm{DEG}(p_1) = n$, то алгоритм 8.7 выполняется за время $O_A\left(M(n)\log n\right)$, где $M(n)$ — время умножения двух полиномов степени n.

Доказательство. Время работы алгоритма 8.7 описывает неравенство

$$T(n) \le T\left(\frac{n}{2}\right) + c_1 M(n) + c_2 M(n)\log n, \tag{8.28}$$

где c_1 и c_2 — константы. Иначе говоря, степень полинома b_1 меньше половины степени полинома a_0, так что первое слагаемое в (8.28) отражает рекурсивный вызов в строке 6. Слагаемое $c_1 M(n)$ учитывает операции деления и умножения в строках 1, 3–5, а $c_2 M(n)\log n$ — вызов процедуры HGCD в строке 2. Легко видеть, что решение неравенства (8.28) ограничено сверху функцией $kM(n)\log n$, где k — некоторая константа. □

Следствие. Наибольший общий делитель двух полиномов степени не выше n можно вычислить за время $O_A\left(n\log^2 n\right)$.

8.10. Наибольший общий делитель целых чисел

Обсудим кратко изменения, которые надо внести в процедурах HGCD и GCD для применения к целым числам. Чтобы понять, где возникают трудности, вернемся к лемме 8.6, показывающей, что при нахождении частных от деления полиномов степеней n и $n - d$ нам ни в одном из них не нужны члены степени, меньшей $n - 2d$.

По аналогии с леммой 8.6 можно рассмотреть два целых числа f и g, $f > g$, и записать их в виде $f = f_1 2^k + f_2$ и $g = g_1 2^k + g_2$, где $f_2 < 2^k$ и $g_2 < 2^k$. Вместо условия $\mathrm{DEG}(g_1(x)) \ge \frac{1}{2}\mathrm{DEG}(f_1(x))$ можно предполагать, что $f_1 \le (g_1)^2$. Тогда можно считать, что $f = qg + r$ и $f_1 = q_1 g_1 + r_1$. Комбинируя эти формулы, получаем

$$g_1 2^k (q - q_1) = f_2 + r_1 2^k - qg_2 - r. \tag{8.29}$$

Поскольку $r_1 < g_1, f_2 < 2^k$ и все эти целые числа неотрицательны, из (8.29) следует, что $q - q_i \le 0$. Пусть $q = q_1 - m$ для некоторого $m \ge 0$. Тогда из (8.29) следует, что

$$mg_1 2^k \le qg_2 + r = (q_1 - m)g_2 + r.$$

Таким образом,

$$mg = qg_1 2^k + mg_2 = q_1 g_2 + r, \tag{8.30}$$

поскольку $f_1 \le (g_1)^2$, то $q_1 \le g_1$. Кроме того, $g_2 < 2^k$ и $r < g$, так что из (8.30) вытекает, что

$$mg < g_1 2^k + g \le 2g. \tag{8.31}$$

Теперь из (8.31) непосредственно следует, что $m < 2$. Таким образом, мы приходим к выводу, что или $q = q_1$, или $q = q_1 - 1$.

В первом случае нет никаких проблем. Если же $q = q_1 - 1$, то нельзя рассчитывать, что процедура `HGCD` будет работать правильно. К счастью, можно показать, что $q_1 \ne q$, только если q — последнее частное в последовательности остатков, для которого процедура `HGCD` строит матрицу. Иначе говоря, подставляя $q - q_1 = -1$ в (8.29), получаем

$$r = f_2 + r_1 2^k - qg_2 + g_1 2^k. \tag{8.32}$$

Поскольку $r < g = g_1^{2k} + g_2$, то из (8.32) вытекает, что $r_1 2^k + f_2 < g_2(1 + q)$ и тем более $r_1 < 1 + q$, а значит, $r_1 < q_1$.

Итак, число r_1, которое стоит в последовательности остатков после f_1 и g_1, должно быть меньше f_1/g_1. Поскольку $g_1 \ge \sqrt{f_1}$, то $r_1 < \sqrt{f_1}$, а это значит, что процедура `HGCD` выдала бы матрицу, содержащую частные вплоть до f_1/g_1, но не далее. Если эта матрица используется в строке 5 процедуры `HGCD`, то может оказаться, что число f, вычисляемое в строке 6, будет не меньше $a_0^{3/4}$. Однако поскольку ошибка в последнем частном равна лишь 1, можно показать, что продолжение последовательности остатков на ограниченное число членов (не зависящее от размера a_0) после выполнения строки 6 будет достаточно для того, чтобы один из очередных членов этой последовательности стал меньше $a_0^{3/4}$. Аналогичный прием необходимо применить после строки 5 процедуры `GCD`.

Еще один источник проблем связан с леммой 8.6. В случае полиномов мы смогли показать, что в последовательности остатков, образованной старшими членами полиномов, определенные члены высокого порядка совпадают с соответствующими членами в последовательности, образованной по всем членам полиномов. Аналогичный результат приближенно выполняется и для целых чисел при условии $q = q_1$. Однако можно только ограничить разность между r и $r_1 2^k$ числом $2^k(q + 1)$. Невозможно гарантировать, что какие-нибудь конкретные разряды чисел r и $r_1 2^k$ будут совпадать. Тем не менее такой источник ошибок округления приведет к тому, что после выполнения строки 6 процедуры `HGCD` и строки 5 процедуры `GCD` в последовательности остатков понадобится лишь конечное число дополнительных членов.

Из-за продолжения последовательности остатков на конечное количество членов сложность алгоритма увеличивается на $O_B(M(n))$, где $M(n)$ — время умноже-

ния n-разрядных двоичных чисел, так что анализ времени работы процедур HGCD и GCD, по существу, не изменится. Поэтому справедлива следующая теорема.

Теорема 8.20. Если $M(n)$ — время умножения двух n-разрядных двоичных целых чисел, то существует алгоритм вычисления их наибольшего общего делителя за $O_{\text{В}}\left(M(n)\log n\right)$ шагов. □

Доказательство. Выполните предложенную выше модификацию процедур HGCD и GCD. □

Следствие. Наибольший общий делитель двух целых чисел можно найти за время $O_{\text{В}}\left(n\log^2 n\log\log n\right)$. □

8.11. Еще раз о китайской теореме об остатках

Как и было обещано, мы покажем, как применить алгоритм GCD для построения асимптотически быстрого алгоритма, который восстанавливал бы целое число по его вычетам без предобусловливания. Вспомним, что алгоритм 8.5, использующий предобусловливание, требует $O_{\text{В}}\left(M(bk)\log k\right)$ времени на восстановление числа u по вычетам, взятым по k модулям, содержащим по b двоичных разрядов. Задача состоит в вычислении $d_i = (p/p_i)^{-1} \bmod p_i$, где $p_0, p_1, \ldots, p_{k-1}$ — модули, а p — их произведение.

С помощью очевидного алгоритма, основанного на методе "разделяй и властвуй", можно вычислить p, найдя сначала произведения пар чисел p_i, затем четверок чисел p_i и т.д., за время $O_{\text{В}}\left(M(bk)\log k\right)$. Метод, положенный в основу алгоритма 8.5, позволяет вычислить $e_i = (p/p_i) \bmod p_i$ при $0 \le i < k$ за $O_{\text{В}}\left(M(bk)\log k\right)$ шагов без предобусловливания. Остается оценить время, необходимое для вычисления $d_i \equiv e_i^{-1} \bmod p_i$.

Поскольку p/p_i — произведение всех модулей, отличных от p_i, оно должно быть взаимно простым по отношению к p_i. Если представить p/p_i в виде $qp_i + e_i$, где q — некоторое целое число, то e_i и p_i будут взаимно простыми, т.е. $\text{GCD}(e_i, p_i) = 1$. Поэтому, если x и y удовлетворяют условию $e_i x + p_i y = 1$, то $e_i x \equiv 1 \pmod{p_i}$. Следовательно, $x \equiv e_i^{-1} = d_i \bmod p_i$. Однако такие числа x и y вычисляются с помощью расширенного алгоритма Евклида.

Хотя процедура GCD предназначена только для вычисления $\text{GCD}(p_1, p_2)$, мы разработали процедуру HGCD так, чтобы она вычисляла матрицу $R_{0,l(n/2)}$. Следовательно, простая модификация алгоритма GCD могла бы вычислять и $R_{0,l(0)}$. Тогда можно было бы получить x, поскольку это левый верхний элемент этой матрицы. Сформулируем результат времени работы китайского алгоритма без предобусловливания.

Теорема 8.21. В случае целых чисел китайский алгоритм имеет сложность $O_{\text{В}}\left(M\left(bk\right)\log k\right)+O_{\text{В}}\left(kM\left(b\right)\log b\right)$, где k — количество используемых модулей, содержащих b битов каждый.

Доказательство. В силу анализа, проведенного выше, первый член выражения для сложности соответствует вычислению чисел e_i и выполнению алгоритма 8.5. Второй соответствует вычислению чисел d_i, поскольку вычисление чисел x и y можно вести по модулю p_i, используя b-битовые операции. □

Следствие. Китайский алгоритм предобусловливания имеет временную сложность $O_{\text{В}}\left(bk\log^2 bk\log\log bk\right)$.[10]

8.12. Разреженные полиномы

Представляя полином от одной переменной в виде $\sum_{i=0}^{n-1}a_i x^i$, мы предполагали, что он *плотный*, т.е. почти все его коэффициенты отличны от нуля. Для многих приложений целесообразно предполагать, что полином *разрежен*, т.е. количество ненулевых коэффициентов намного меньше его степени. В такой ситуации логично представлять полином списком пар (a_i, j_i), состоящих из ненулевого коэффициента и соответствующего ему показателя степени переменной x.

Поскольку мы не можем углубляться в разнообразные методы выполнения арифметических операций над разреженными полиномами, упомянем лишь два интересных аспекта этой теории. Во-первых, поскольку неразумно применять преобразование Фурье для умножения, мы приведем один разумный алгоритм умножения разреженных полиномов. Во-вторых, на примере вычисления полинома $[p(x)]^4$ мы продемонстрируем удивительное различие между способами выполнения арифметических операций над плотными и разреженными полиномами.

Наиболее разумная из известных стратегий умножения разреженных полиномов состоит в том, что полином $\sum_{i=1}^{n}a_i x^{j_i}$ задается списком пар (a_1, j_1), (a_2, j_2), ..., (a_n, j_n), где все j_i различные и расположены в порядке убывания, т.е. $j_i > j_{i+1}$ для $1 \le i < n$. Чтобы умножить два полинома, представленных таким образом, вычислим произведения пар и расположим их по величине показателей (по вторым компонентам пар), объединяя все члены с одинаковыми показателями. Если этого не делать, то увеличится количество членов с одинаковыми показателями и по мере выполнения арифметических операций сложность начнет значительно увеличиваться.

[10] Интересно, что при $M(n)=n\log n\log\log n$ эта величина является наилучшей, которую можно получить из теоремы 8.21 независимо от того, как связаны b и k.

При умножении упорядоченных разреженных полиномов можно воспользоваться их упорядоченностью и тем фактом, что в одном из них может оказаться намного больше членов, чем в другом, чтобы максимально упростить упорядочение произведений. Приведем неформальный алгоритм умножения разреженных полиномов таким способом.

Алгоритм 8.8. Умножение упорядоченных разреженных полиномов

Вход. Полиномы

$$f(x)=\sum_{i=1}^{m} a_i x^{j_i} \quad \text{и} \quad g(x)=\sum_{i=1}^{n} b_i x^{k_i},$$

представленные списками пар

$$(a_1, j_1), (a_2, j_2), \ldots, (a_m, j_m) \text{ и } (b_1, k_1), (b_2, k_2), \ldots, (b_n, k_n),$$

где последовательности j и k монотонно убывают.

Выход. Полином

$$\sum_{i=1}^{p} c_i x^{l_i} = f(x)g(x),$$

представленный списком пар, в котором последовательность $l_1, \ldots, l_p$ монотонно убывает.

Метод. Не умаляя общности, предполагаем, что $m \geq n$.

1. Строим последовательности S_i, $1 \leq i \leq n$, в которых r-й член, $1 \leq r \leq m$, равен $(a_r b_i, j_r + k_i)$. Таким образом, S_i представляет произведение полинома $f(x)$ на i-й член полинома $g(x)$.
2. Объединяем S_{2i-1} с S_{2i} для $1 \leq i \leq n/2$, приводя подобные члены. Затем попарно объединяем полученные последовательности, приводя подобные члены, и повторяем процесс, пока не получим одну упорядоченную последовательность. □

Теорема 8.22. Алгоритм 8.8 занимает $O(mn\log n)$ времени[11] в предположении, что $m \geq n$.

Доказательство. Шаг 1, очевидно, имеет сложность $O(mn)$. Шаг 2 необходимо повторить $\lceil \log n \rceil$ раз, и общий объем работы при каждом проходе займет $O(mn)$ времени. □

[11] Заметим, что здесь фигурирует сложность на RAM-машине, а не арифметическая сложность, поскольку программа для алгоритма 8.8 неизбежно содержит разветвления, даже если m и n фиксированы.

Посмотрим теперь, как алгоритм 8.8 и его временная сложность влияют на выбор способа выполнения арифметических операций над разреженными полиномами.

Пример 8.13. Вычислим $p^4(x)$, где $p(x)$ — плотный или разреженный полином с n членами. Если полином плотный, то легко показать, что лучше всего вычислять $p^4(x)$, дважды возводя в квадрат. Иначе говоря, пусть $M(n) = cn\log n$, где $M(n)$ — время умножения плотных полиномов. Тогда можно вычислить $p^2(x)$ за $cn\log n$ шагов и результат возвести в квадрат за $2cn\log 2n$ шагов, всего затратив $3cn\log n + 2cn$ шагов. Для сравнения укажем, что если вычислять $p^4 = p \cdot (p \cdot (p \cdot p))$, то, как легко видеть, требуется $6cn\log n$ шагов в предположении, что на вычисление $p \cdot p^2$ тратится $2M(n)$ шагов, а на вычисление $p \cdot p^3$ — $3M(n)$ шагов. Таким образом, для плотных полиномов получаем, как и ожидали, что p^4 надо вычислять повторным возведением в квадрат.

Теперь пусть $p(x)$ — разреженный полином с n членами. Если вычислять $(p^2)^2$ по алгоритму 8.8, то на первое возведение в квадрат уходит $cn^2\log n$ времени, а в предположении, что подобных членов немного, второе возведение в квадрат занимает $cn^4\log n^2$ времени, так что всего тратится $c\left(2n^4 + n^2\right)\log n$ шагов. С другой стороны, $p \cdot (p \cdot (p \cdot p))$ вычисляется за $cn^2\log n + cn^3\log n + cn^4\log n = c\left(n^4 + n^3 + n^2\right)\log n$ шагов. Эта величина меньше времени повторного возведения в квадрат. Таким образом, повторное возведение в квадрат разреженного полинома не всегда дает хороший способ вычисления p^4. Особенно сильно этот эффект проявляется при вычислении p^{2^t} для больших значений t. □

Упражнения

8.1. Используйте алгоритм 8.1 для нахождения числа, обратного к 429.

8.2. Используйте алгоритм 8.2 для вычисления 429^2.

8.3. Используйте алгоритм 8.3 для нахождения полинома, обратного к

$$x^7 - x^6 + x^5 - 3x^4 + x^3 - x^2 + 2x + 1.$$

***8.4.** Опишите алгоритм, аналогичный алгоритму 8.2, для вычисления квадратов полиномов.

8.5. Используйте свой алгоритм из упражнения 8.4, чтобы вычислить $(x^3 - x^2 + x - 2)^2$.

8.6. Используйте алгоритм 8.4, чтобы найти представление для числа 1 000 000 по модулям 2, 3, 5, 7, 11, 13, 17, 19.

8.7. Напишите полный алгоритм нахождения вычетов полинома по набору полиномиальных модулей.

8.8. Найдите вычеты полинома $x^7 + 3x^6 + x^5 + 3x^4 + x^2 + 1$ по модулям $x + 3$, $x^3 - 3x + 1$, $x^2 + x - 2$ и $x^2 - 1$.

8.9. Полиномы в упражнении 8.8 были специально выбраны так, чтобы можно было легко выполнить вычисления вручную. Случайно выберите четыре полинома степеней 1, 2, 3 и 4 и найдите относительно них вычеты полинома $x^9 - 4$. Что произойдет?

8.10. Пусть 5, 6, 7, 11 — четыре модуля. Найдите такое число u, меньшее их произведения, что и $u \leftrightarrow (1, 2, 3, 4)$.

***8.11.** Обобщите лемму 8.3 так, чтобы применить ее к произвольным полиномиальным модулям, корни которых известны или могут быть найдены (например, полиномам степени меньше 5). Какова сложность восстановления полиномов по их вычетам, если использовать ваш алгоритм (сложность рассматривается как функция от числа и наибольшей степени модулей)?

8.12. Найдите полином, значения которого в точках 0, 1, 2, 3 равны 1, 1, 2, 2 соответственно.

8.13. Найдите наибольший общий делитель полиномов

$$x^6 + 3x^5 + 3x^4 + x^3 - x^2 - x - 1,$$
$$x^6 + 2x^5 + x^4 + 2x^3 + 2x^2 + x + 1.$$

8.14. Полиномы в упражнении 8.13 были специально выбраны так, чтобы можно было легко выполнить вычисления вручную. Выберите два произвольных полинома степеней 7 и 8 и попробуйте найти их `GCD`. Что произойдет? Каким, по вашему предположению, окажется `GCD`?

***8.15.** Напишите полный алгоритм нахождения `GCD` целых чисел с временной сложностью $O_{\text{в}}\left(n \log^2 n \log\log n\right)$ на n-разрядных двоичных целых числах.

8.16. Используйте свой алгоритм из упражнения 8.15, чтобы найти `GCD`(377, 233).

***8.17.** Пусть $p(x)$ — разреженный полином с n мономами. Найдите как функцию от n сложность наилучшего метода вычисления $p^8(x)$ при условии, что умножения выполняются по алгоритму 8.8.

****8.18.** Предлагается следующий метод вычисления $f(x)/g(x)$ по полиномам f и g в предположении, что g делит f. Пусть f и g имеют степень $n - 1$ или меньше.

1. Вычислить дискретные преобразования Фурье F и G полиномов f и g соответственно.
2. Разделить члены полинома F на соответствующие члены полинома G, получив последовательность H.

3. Взять обратное преобразование от H. Считаем, что результатом будет f/g. Будет ли работать этот алгоритм?

**8.19. Пусть $M(n)$ — время умножения двух n-разрядных двоичных чисел и $Q(n)$ — время вычисления $\lfloor \sqrt{i} \rfloor$ для n-разрядного двоичного целого числа i. Предположим, что $M(an) \geq aM(n)$ при $a \leq 1$ и аналогичное неравенство верно для $Q(n)$. Покажите, что $M(n)$ и $Q(n)$ равны с точностью до постоянного множителя.

*8.20. Обобщите упражнение 8.19 на (*а*) полиномы, (*б*) корни r-й степени при фиксированном r.

**8.21. Напишите алгоритм сложности $O_{\text{A}}(n \log n)$, который вычислял бы полином степени $n-1$ и все его производные в данной точке.

**8.22. Плотный полином от r переменных[12] можно представить в виде

$$\sum_{i_1=0}^{n-1} \sum_{i_2=0}^{n-1} \cdots \sum_{i_r=0}^{n-1} a_{i_1 i_2 \ldots i_r} x_1^{i_1} x_2^{i_2} \ldots x_r^{i_r}.$$

Покажите, что такие полиномы можно перемножать за время $O_{\text{A}}(n^r \log n)$, если для вычисления значений и интерполяции использовать точки $x_1 = \omega^{i_1}$, $x_2 = \omega^{i_2}$, ..., $x_r = \omega^{i_r}$ при $0 \leq j_k < 2n$, где ω — примитивный корень степени $2n$ из единицы.

**8.23. Покажите, что при разумных допущениях о гладкости функций $M(n)$ и $D(n)$ временная сложность умножения и деления плотных полиномов от r переменных одинакова с точностью постоянного множителя.

**8.24. Найдите алгоритм сложности $O_{\text{B}}(n \log^2 n \log\log n)$, который переводил бы (*а*) n-разрядные двоичные числа в десятичные; (*б*) n-разрядные десятичные числа в двоичные.

**8.25. *Наименьшим общим кратным* (Least Common Divisor — LCM) целых чисел (полиномов) x и y называется целое число (полином), которое делится на x и y и делит всякое другое целое число (всякий другой полином), делящееся (делящийся) на x и y. Покажите, что время нахождения LCM двух n-разрядных двоичных чисел не меньше времени их умножения.

8.26. Порождает ли метод умножения целых чисел из раздела 2.6 алгоритм умножения полиномов с временной сложностью $O_{\text{A}}(n^{1,59})$?

**8.27. Покажите, что полином степени $n-1$ можно вычислить в точках $a^0, a^1, \ldots, a^{n-1}$ за $O_{\text{A}}(n \log n)$ шагов. (*Подсказка*: покажите, что для некоторых функций f и g полином $c_j = \sum_{i=0}^{n-1} b_i a^{ij}$ можно представить в виде $c_j = \sum_{i=0}^{n-1} f(i) g(j-i)$.)

[12]По мере увеличения r предположение о плотности полинома становится все менее полезным.

**8.28. Покажите, что полином степени $n-1$ можно вычислить в n точках $ba^{2j} + ca^j + d$, $0 \le j \le n$ за $O_A(n \log n)$.

8.29. Постройте рекурсивные варианты алгоритмов 8.4 и 8.5.

Проблемы для исследования

8.30. В этой главе мы показали, что многие задачи о полиномах и целых числах

а) имеют, по существу, ту же сложность, что и умножение, или

б) сложнее умножения не более чем в логарифмический множитель.

Некоторые из этих задач приводятся в упражнениях к данной главе. Упражнение 9.9 ставит еще одну задачу из группы (*б*) — задачу об умножении "и/или".

Интересно было бы расширить множества задач (*а*) или (*б*). Другая область исследования — показать, что некоторые задачи из группы (*б*) должны иметь одинаковую сложность. Например, можно предположить, что это верно для задач нахождения наибольшего общего делителя и наименьшего общего кратного.

8.31. Еще одна проблема, которая кажется обманчиво простой: выяснить, является ли алгоритм 8.8 наилучшим алгоритмом умножения упорядоченных разреженных полиномов и упорядочения результата. Некоторые аспекты этой проблемы рассмотрел Johnson [1974].

8.32. Значение n, при котором многие из описанных алгоритмов становятся практичными, огромно. Однако их можно было бы тщательно скомбинировать с методами сложности $O(n^{1,59})$, упомянутыми в разделе 2.6 и упражнении 8.26, при небольших n и очевидными методами сложности $O(n^2)$ при очень малых n. Найдите верхние оценки значений n, для которых задачи этой главы обладают алгоритмами, лучшими очевидных. Некоторая работа на этом пути уже проделана в Kung [1973].

Библиографические замечания

Теорема 8.2, утверждающая, что обращение не сложнее умножения, взята у Cook [1966]. Любопытно, что существование полиномиального аналога алгоритма 8.1 не осознавали в течение нескольких лет. Moenk and Borodin [1972] описали алгоритм сложности $O_B(n \log^2 n \log\log n)$ для деления, а через короткое время алгоритм деления сложности $O_B(n \log n \log\log n)$ независимо изложили несколько авторов, в том числе Sieveking [1972].

Построение алгоритмов для модульной арифметики, а также для вычисления значений и интерполяции полиномов изложено в работе Moenck and Borodin [1972].

Heindel and Horowitz [1971] нашли алгоритм сложности $O_{\text{B}}\left(n\log^2 n\log\log n\right)$, восстанавливающий числа по китайской теореме об остатках и использующий предобусловливание. Borodin and Munro [1971] описали алгоритм сложности $O(n^{1.91})$ для вычисления значений полинома в нескольких точках, а Horowitz [1972] обобщил его на интерполяцию. Kung [1973] построил алгоритм сложности $O_{\text{A}}\left(n\log^2 n\right)$ для вычисления значений полинома без применения быстрого (т.е. имеющего сложность $n\log n$) алгоритма деления. Единство восстановления по вычетам, интерполяции, вычисления значений полиномов и вычисления вычетов целых чисел установил Lipson [1971].

Алгоритм сложности $O_{\text{B}}\left(n\log^2 n\log\log n\right)$ для нахождения GCD целых чисел принадлежит Schönhage [1971]. На полиномы и общие евклидовы области его перенес Moenck [1973].

Обзор классических методов вычисления GCD дан в Knuth [1969]. Некоторую подборку материала о сложности арифметических операций над разреженными полиномами можно найти в Brown [1971], Gentleman [1972] и Johnson [1973]. Дополнительные результаты о реализации арифметических операций над полиномами на вычислительных машинах приведены в Hall [1971], Brown [1973] и Collins [1973]. Алгоритм 8.8 со структурой данных в виде дерева реализовал S. Johnson на языке ALTRAN (см. Brown [1973]).

Упражнения 8.19 и 8.20 принадлежат R. Karp и J. Ullman. Упражнение 8.21 о вычислении значений полинома и его производных можно найти в работах Vari [1974] и Aho, Steiglitz, Ullman [1974]. Из последней работы взято также упражнение 8.28. Упражнение 8.27 приведено в Blueatein [1970] и Rabiner, Chafer and Rader [1969]. Подробно арифметические операции над полиномами и целыми числами обсуждаются в книге Borodin and Munro [1975].

ГЛАВА 9

Алгоритмы сопоставления с образцом

Алгоритмы сопоставления с образцом являются неотъемлемой частью многих задач, связанных с редактированием текстов, поиском данных и символьной обработкой. В типичной задаче сопоставления строк задаются текстовая строка x и множество строк-образцов $\{y_1, y_2, ...\}$. Требуется найти либо одно, либо все вхождения строк-образцов в строке x. Множество образцов часто является регулярным множеством, заданным регулярным выражением. В данной главе мы обсудим несколько методов решения задач сопоставления строк с образцом.

Мы начнем с краткого обзора регулярных выражений и конечных автоматов. Затем изложим алгоритм, выявляющий в данной строке вхождение какой-нибудь строки из множества, заданного регулярным выражением. Порядок временной сложности алгоритма равен произведению длин строки x и данного регулярного выражения. Затем мы приведем алгоритм линейной сложности, который распознает, является ли данная строка y подстрокой данной строки x. Далее докажем сильный теоретический результат, состоящий в том, что любую проблему распознавания вхождения строк, разрешимую на двустороннем детерминированном магазинном автомате, можно решить на машине RAM за линейное время. Это замечательный результат, поскольку магазинный автомат может потребовать для решения задачи квадратичное или даже экспоненциальное время. Наконец, введем понятие позиционного дерева и применим его к другим задачам сопоставления строк, таким, как поиск в данной строке самого длинного повторения.

9.1. Конечные автоматы и регулярные выражения

Многие задачи распознавания образцов и их решения можно выразить в терминах регулярных выражений и конечных автоматов. Поэтому начнем с обзора соответствующего материала.

Определение. *Алфавитом* называется конечное множество символов. *Строка* в алфавите I — это конечная последовательность символов из I. *Пустая строка* ε — это строка, не содержащая ни одного символа. *Конкатенацией* строк x и y называется строка xy. Если строка имеет вид xyz, то x — ее *префикс*; y — подстрока; z — суффикс. Длиной строки x называется общее количество $|x|$ вхождений символов в x. Например, длина строки *aab* равна 3; строка ε имеет длину 0.

Языком над алфавитом I называют произвольное множество строк в алфавите I. Пусть L_1 и L_2 — два языка. Язык L_1L_2, называемый *конкатенацией* L_1 и L_2, — это множество $\{xy | x \in L_1 \text{ и } y \in L_2\}$.

Пусть L — язык. Тогда положим $L^0 = \{\varepsilon\}$ и $L^i = LL^{i-1}$ при $i \geq 1$. *Замыканием Клини* L^* языка L называют язык $L^* = \bigcup_{i=0}^{\infty} L^i$, а *позитивным замыканием* L^+ — язык $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Регулярные выражения и языки, которые они представляют (регулярные множества), полезны во многих областях компьютерных наук. В этой главе мы увидим, что они полезны для описания образцов.

Определение. Пусть I — алфавит. *Регулярные выражения* над I и языки, представляемые ими, рекурсивно определяются следующим образом.

1. $\varnothing$ является регулярным выражением и представляет пустое множество.
2. ε является регулярным выражением и представляет множество $\{\varepsilon\}$.
3. Для каждого $a \in I$ символ a является регулярным выражением и представляет множество $\{a\}$.
4. Если p и q — регулярные выражения, представляющие множества P и Q соответственно, то $(p + q)$, (pq) и (p^*) являются регулярными выражениями и представляют множества $P \cup Q$, PQ и P^* соответственно.

При записи регулярного выражения можно опустить многие скобки, если предположить, что операция * имеет более высокий приоритет, чем конкатенация и операция +, а конкатенация имеет более высокий приоритет, чем операция +. Например, ((0(1*)) + 0) можно записать как 01* + 0. Кроме того, вместо pp^* будем кратко писать p^+.

Пример 9.1.

1. 01 — регулярное выражение, представляющее множество {01}.
2. (0 + 1)* представляет {0, 1}*.
3. 1(0 + 1)*1 + 1 представляет множество всех строк, начинающихся и кончающихся символом 1. □

Язык называется *регулярным*, если его можно представить регулярным выражением. Два регулярных выражения α и β называют *эквивалентными* (этот факт обозначается как $\alpha = \beta$), если они представляют одно и то же множество. Например, $(0 + 1)^* = (0^* 1^*)^*$.

Понятие детерминированного конечного автомата было введено в главе 4. Его можно интерпретировать как устройство, состоящее из блока управления, который всегда находится в одном из конечного числа состояний, и входной ленты, которая сканируется слева направо своей считывающей головкой. Детерминированный конечный автомат выполняет шаги, определяемые текущим состоянием его блока управления и входным символом, на котором установлена считывающая головка. Каждый шаг состоит из перехода в новое состояние и сдвига считывающей головки на одну ячейку вправо. Оказывается, что язык можно представить регулярным выражением тогда и только тогда, когда он допускается некоторым конечным автоматом.

Важным обобщением рассматриваемого понятия является недетерминированный конечный автомат. Для каждого состояния и каждого входного символа недетерминированный автомат имеет нуль или более вариантов выбора следующего шага. Он может также выбирать, сдвигать ему считывающую головку при изменении состояния или нет.

Определение. *Недетерминированным конечным автоматом* (Nondeterministic Finite Automation — NDFA) называется пятерка (S, I, δ, s_0, F), где

1) S — конечное множество состояний устройства управления;
2) I — алфавит входных символов;
3) δ — *функция переходов*, отображающая $S \times (I \cup \{\varepsilon\})$ в множество подмножеств множества S;
4) $s_0 \in S$ — *начальное состояние* устройства управления;
5) $F \subseteq S$ — множество *заключительных* (*допускающих*) состояний.

Мгновенным описанием (Instantaneous Description — ID) M называется пара (s, w), где $s \in S$ — состояние блока управления и $w \in I^*$ — неиспользованная часть входной строки (т.е. символ, обозреваемый считывающей головкой, и все символы справа от него). *Начальным мгновенным описанием* автомата M называется мгновенное описание, у которого первым компонентом является начальное состояние, а вторым — распознаваемая строка, т.е. (s_0, w) для некоторой строки $w \in I^*$. *Допускающее мгновенное описание* — это мгновенное описание вида (s, ε), где $s \in F$.

Представим шаги недетерминированного конечного автомата бинарным отношением $\vdash$ на множестве мгновенных описаний. Если $\delta(s, a)$ содержит s', то пишут $(s, aw) \vdash (s', w)$ для всех $w \in I^*$. Заметим, что a — это или ε, или символ из I. Если $a = \varepsilon$, то состояние изменяется независимо от обозреваемого входного символа. Если $a \neq \varepsilon$, то символ a должен стоять в очередной ячейке входной ленты, а считывающая головка сдвигается на одну ячейку вправо.

Рефлексивное и транзитивное замыкание отношения обозначается через $\vdash^*$. Говорят, что *строка w допускается автоматом M*, если $(s_0, w) \vdash^* (s, \varepsilon)$ для некоторого $s \in F$. Иначе говоря, входная строка w допускается автоматом M, если найдется последовательность из нуля или более шагов, переводящая M из начального мгновенного описания (s_0, w) в допускающее мгновенное описание (s, ε). Множество строк $L(M)$, допускаемых автоматом M, называют *языком, допускаемым автоматом M.*

Пример 9.2. Рассмотрим недетерминированный конечный автомат M, допускающий все те строки из символов a и b, которые оканчиваются строкой aba, т.е. $L(M) = (a + b)^* aba$. Пусть $M = (\{s_1, s_2, s_3, s_4\}, \{a, b\}, \delta, s_1, \{s_4\})$, где функция δ определена на рис. 9.1. (Здесь без ε-переходов можно обойтись.)

	Вход		
Состояние	a	b	ε
s_1	$\{s_1, s_2\}$	$\{s_1\}$	$\varnothing$
s_2	$\varnothing$	$\{s_3\}$	$\varnothing$
s_3	$\{s_4\}$	$\varnothing$	$\{s_1\}$
s_4	$\varnothing$	$\varnothing$	$\{s_2\}$

Рис. 9.1. Функция переходов δ

Пусть на вход автомата M подается строка $ababa$. Тогда M может сработать в соответствии с последовательностями мгновенных описаний, показанными на рис. 9.2. Поскольку $(s_1, ababa) \vdash^* (s_4, \varepsilon)$ и s_4 — заключительное состояние, строка $ababa$ допускается автоматом M. □

С каждым недетерминированным конечным автоматом связан ориентированный граф, естественным образом представляющий функцию переходов этого автомата.

Определение. Пусть $M = (S, I, \delta, s_0, F)$ — недетерминированный конечный автомат. *Диаграммой переходов* автомата M называют ориентированный граф $G = (S, E)$ с помеченными ребрами. Множество ребер E и метки определяются следующим

образом. Если $\delta(s, a)$ содержит s' для некоторого $a \in I \cup \{\varepsilon\}$, то ребро (s, s') принадлежит E. Меткой ребра (s, s') является множество тех $b \in I \cup \{\varepsilon\}$, для которых $\delta(s, b)$ содержит s'.

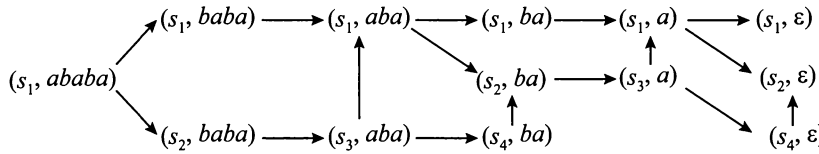


Рис. 9.2. Последовательность мгновенных описаний для входа *ababa*

Пример 9.3. Диаграмма переходов для недетерминированного конечного автомата M из примера 9.2 изображена на рис. 9.3. Заключительное состояние обведено двойным кружком. □

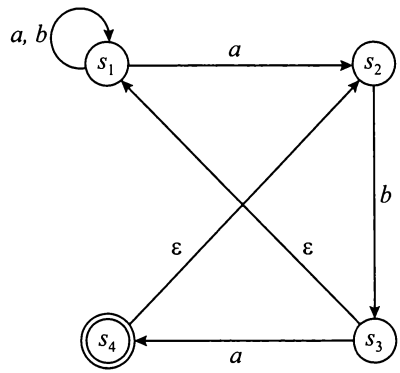


Рис. 9.3. Диаграмма переходов для примера 9.2

Диаграммы недетерминированного конечного автомата и задачи о путях на графах можно связать с помощью определенного замкнутого полукольца. Пусть I — алфавит; положим $S_I = (\mathcal{P}(I^*), \cup, \cdot, \varnothing, \{\varepsilon\})$. Из раздела 5.6 известно, что S_I — замкнутое полукольцо, в котором $\mathcal{P}(I^*)$ множество всех языков над алфавитом I; $\varnothing$ — единичный элемент относительно объединения $\cup$; $\{\varepsilon\}$ — единичный элемент относительно конкатенации.

Теорема 9.1. Любой язык, допускаемый недетерминированным конечным автоматом, является регулярным.

Доказательство. Пусть $M = (S, I, \delta, s_0, F)$ — недетерминированный конечный автомат и $G = (S, E)$ — его диаграмма переходов. Алгоритм 5.5 может найти для каждой пары вершин s и s' диаграммы язык $L_{ss'}$, являющийся множеством всех строк, помечающих пути из s в s'. Метка каждого ребра диаграммы является регулярным выражением. Более того, если множества C_{ij}^{k-1}, вычисленные алгоритмом 5.5, можно представить регулярными выражениями, то в силу строки 5 этого алгоритма множества C_{ij}^{k} можно представить регулярными выражениями. Следовательно, каждый язык $L_{ss'}$ можно представить регулярным выражением, а потому он регулярен. Тогда $L(M) = \bigcup_{s \in F} L_{s_0 s}$ — регулярное множество, поскольку по определению объединение регулярных множеств регулярно. □

Теорема, обратная теореме 9.1, также верна. Иначе говоря, для данного регулярного выражения найдется недетерминированный конечный автомат, допускающий язык, представленный этим регулярным выражением. С точки зрения сложности вычислений наиболее важно, что можно найти недетерминированный конечный автомат, у которого число состояний не больше удвоенной длины данного регулярного выражения и который из каждого своего состояния может перейти не более чем в два других.

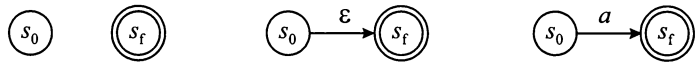
Теорема 9.2. Пусть α — регулярное выражение. Тогда найдется недетерминированный конечный автомат $M = (S, I, \delta, s_0, \{s_f\})$, допускающий язык, представленный регулярным выражением α, и обладающий следующими свойствами:

1. $\|S\| \le 2|\alpha|$, где $|\alpha|$ — длина выражения α.[1]
2. Для каждого $a \in I \cup \{\varepsilon\}$ множество $\delta(s_f, a)$ пусто.
3. Для каждого $s \in S$ сумма чисел $\|\delta(s, a)\|$ по всем $a \in I \cup \{\varepsilon\}$ не превосходит 2.

Доказательство. Применим индукцию по длине выражения α. Рассмотрим базу, т.е. случай $|\alpha| = 1$. Тогда выражение α должно быть одного из трех видов: (*а*) $\varnothing$, (*б*) ε или (*в*) a для некоторого $a \in I$. На рис. 9.4 изображены три автомата, имеющие по два состояния каждый, допускающие указанные языки и удовлетворяющие условиям теоремы.

---

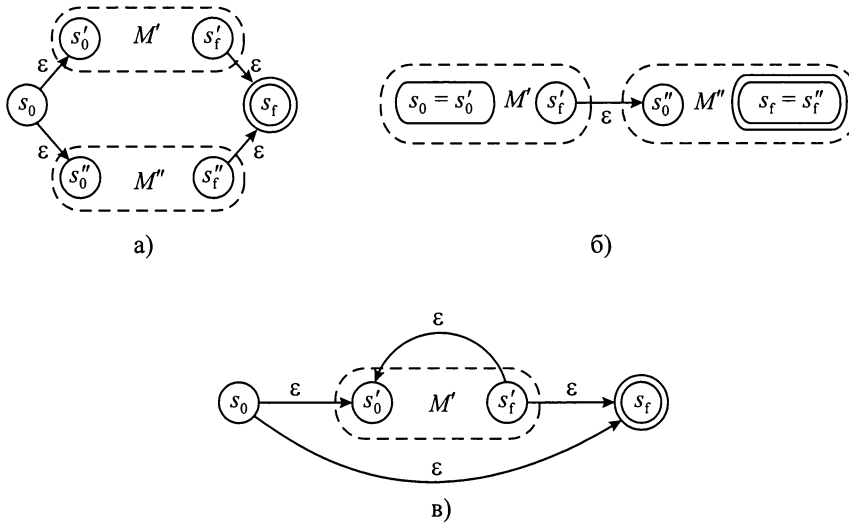
[1] Длиной регулярного выражения α называется число вхождений символов в строку α. Например, $|\varepsilon^*| = 2$, Можно было бы усилить свойство 1, игнорируя скобки при подсчете длины выражения α (например, регулярное выражение (a^*b^*), при таком условии оно имело бы длину 5, а не 7.



а) б) в)

Рис. 9.4. Конечные автоматы, допускающие языки, представленные регулярными выражениями длины 1: (*а*) $\varnothing$, (*б*) {ε} и (*в*) {*a*}

На шаге индукции выражение α должно быть одного из четырех видов: (*а*) $\beta+\gamma$; (*б*) $\beta\gamma$; (*в*) β^*; (*г*) (β), где β и γ — регулярные выражения. В случае (*г*) выражения α и β обозначают один и тот же язык, поэтому гипотеза индукции очевидна. Для остальных вариантов допустим, что M' и M'' —недетерминированные конечные автоматы, представленные выражениями β и γ соответственно, причем их множества состояния не пересекаются. Пусть s_0' и s_0'' — начальные состояния этих автоматов, а s_f' и s_f'' — их конечные состояния. Диаграммы переходов для случаев (*а*), (*б*) и (*в*) показаны на рис. 9.5.



а) б)

в)

Рис. 9.5. Диаграммы переходов автоматов, допускающих языки, представленные регулярными выражениями (*а*) $\beta+\gamma$, (*б*) $\beta\gamma$, (*в*) β^*

Пусть длины выражений α, β и γ равны $|\alpha|$, $|\beta|$ и $|\gamma|$, а n, n' и n'' — количество состояний автоматов M, M' и M''. В случае (*а*) $|\alpha| = |\beta| + |\gamma| + 1$ и $n = n' + n'' + 2$. По предположению индукции $n' \le 2|\beta|$ и $n'' \le 2|\gamma|$, поэтому $n \le 2|\alpha|$. Кроме того, в случае (*а*) добавляются только два ребра, выходящие из s_0, которые удовлетворяют ограничению, согласно которому из любой вершины исходит не более двух

ребер и еще по одному ребру, исходящему из каждой вершины s'_f и s''_f. Поскольку по индуктивной гипотезе из каждой вершины s'_f и s''_f раньше не выходило ни одного ребра, то это же ограничение на число выходящих ребер выполняется и для s'_f и s''_f. Наконец, из s_f, очевидно, никакие ребра не выходят. Случай (*б*) проверяется аналогично.[2]

В случае (*в*) $|\alpha| = |\beta| + 1$ и $n = n' + 2$. Поскольку $n' \le 2|\beta|$, то отсюда следует, что $n \le 2|\alpha|$. Ограничение на число ребер, выходящих из любой вершины, легко проверяется. □

Пример 9.4. Построим недетерминированный конечный автомат для регулярного выражения $ab^* + c$, длина которого равна 5. Недетерминированный конечный автомат для a, b и c изображен на рис. 9.4, *в*. С помощью конструкции, показанной на рис. 9.5, *б* построим автомат для b^*, как показано на рис. 9.6, *а*. Затем с помощью конструкции, представленной на рис. 9.5, *б*, построим автомат для ab^*, как показано на рис. 9.6, *б*. Наконец, с помощью конструкции, показанной на рис. 9.5, *а* построим недетерминированный конечный автомат для $ab^* + c$. Этот автомат, имеющий 10 состояний, изображен на рис. 9.6, *в*. □

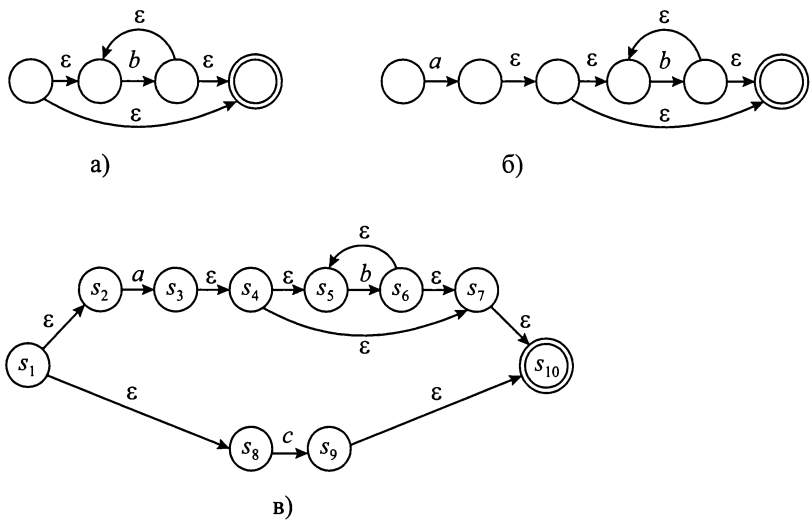


Рис. 9.6. Построение недетерминированного конечного автомата для выражения $ab^* + c$: (*а*) для b^*; (*б*) ab^*; (*в*) для $ab^* + c$

[2]На самом деле ребро из s'_f в s''_0 не нужно. Вместо него можно было бы просто отождествить состояния s'_f в s''_0. Точно так же на рис. 9.5, *а*, можно было бы отождествить s'_f в s''_f и считать одним заключительным состоянием.

Необходимо упомянуть один дополнительный результат. По заданному недетерминированному конечному автомату можно найти эквивалентную детерминированную машину. Однако детерминированный конечный автомат, эквивалентный данному недетерминированному конечному автомату, может иметь вплоть до 2^n состояний. Следовательно, переход к детерминированному автомату не всегда дает аффективный способ моделирования недетерминированного конечного автомата. Тем не менее детерминированные автоматы оказываются полными в распознавании образцов. Напомним их определение.

Определение. *Детерминированным конечным автоматом* (Deterministic Finite Automation — DFA) называется недетерминированный конечный автомат (S, I, δ, s_0, F), в котором

1. $\delta(s, \varepsilon) = \varnothing$ для всех $s \in S$.
2. $\|\delta(s, a)\| \leq 1$ для всех $s \in S$ и $a \in I$.

Теорема 9.3. Если L — регулярное множество, то оно допускается некоторым детерминированным конечным автоматом.

Доказательство. По теореме 9.2 множество L допускается некоторым недетерминированным конечным автоматом $M = (S, I, \delta, s_0, \{s_f\})$. Преобразуем M в детерминированный конечный автомат следующим образом. Сначала найдем такие пары состояний (s, t), что $(s, \varepsilon) \vdash_M^* (t, \varepsilon)$. Для этого построим ориентированный граф $G = (S, E)$, у которого $(s, t) \in E$ тогда и только тогда, когда $\delta(s, \varepsilon)$ содержит t. Затем вычислим рефлексивное и транзитивное замыкание $G' = (S, E')$ графа G. Мы утверждаем, что $(s, \varepsilon) \vdash_M^* (t, \varepsilon)$ тогда и только тогда, когда (s, t) принадлежит E'.

Теперь построим такой недетерминированный конечный автомат $M' = (S', I, \delta', s_0, F)$, что $L(M') = L(M)$, в M' нет ε-переходов и выполняются следующие условия.

1. $S' = \{s_0\} \cup \{t | \delta(s, a)$ содержит t для некоторого $s \in S$ и некоторого $a \in I$.
2. Для каждого $s \in S'$ и каждого $a \in I$

$$\delta'(s, a) = \{u | (s, t) \in E' \text{ и } \delta(t, a) \text{ содержит } u\}.$$

3. $F' = \{s | (s, f) \in E' \text{ и } f \in F\}$.

В качестве упражнения предлагаем доказать, что $L(M) = L(M')$. Очевидно, что в M' нет переходов по ε.

Далее по M' построим детерминированный конечный автомат M'', состояния которого образуют множество-степень для S'. Иначе говоря, $M'' = (\mathcal{P}(S'), I, \delta'', \{s_0\}, F'')$, где

а) для каждого подмножества S множества S' и каждого $a \in I$

$$\delta''(S, a) = = \{t | \delta'(s, a) \text{ содержит } t \text{ для некоторого } s \in S\};$$

б) $F'' = \{S | S \cap F \neq \varnothing\}$.

В качестве упражнения докажите индукцией по $|w|$, что $(\{s_0\}, w) \vdash^*_{M''} (S, \varepsilon)$ тогда и только тогда, когда $S = \{t | (s_0, w) \vdash^*_{M'} (t, \varepsilon)\}$. Таким образом, $L(M) = L(M') = L(M'')$. □

Пример 9.5. Рассмотрим недетерминированный конечный автомат M, представленный на рис. 9.7. Из начального состояния s_1 можно достичь s_3 и заключительное состояние s_4 по путям, помеченным символом ε. Следовательно, для вычисления рефлексивного и транзитивного замыкания G' ориентированного графа G, о котором шла речь в доказательстве теоремы 9.3, необходимо добавить ребро (s_1, s_4). Весь граф G' изображен на рис. 9.8. По M и G' построим недетерминированный конечный автомат M' (рис. 9.9). Поскольку в вершину s_4 входят ребра из всех вершин графа G', объявляем все состояния в M' заключительными. Поскольку единственное ребро, входящее в вершину s_3 в диаграмме для M, помечено символом ε, то s_3 не входит в M'.

При построении детерминированного конечного автомата M'' по автомату M' образуется восемь состояний. Однако только четырех из них можно достичь из начального состояния, так что остальные четыре можно выбросить. Детерминированный конечный автомат M'' изображен на рис. 9.10. □

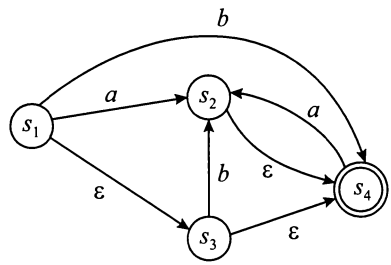


Рис. 9.7. Недетерминированный конечный автомат из примера 9.5

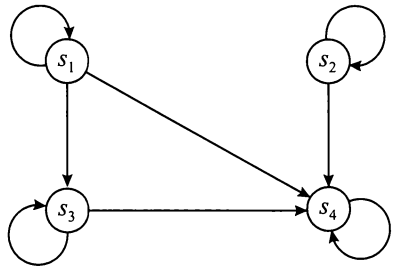


Рис. 9.8. Граф G'

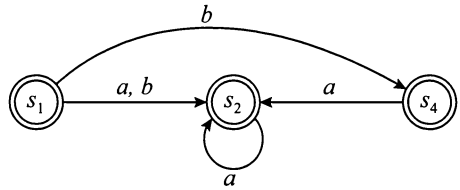


Рис. 9.9. Недетерминированный конечный автомат M'

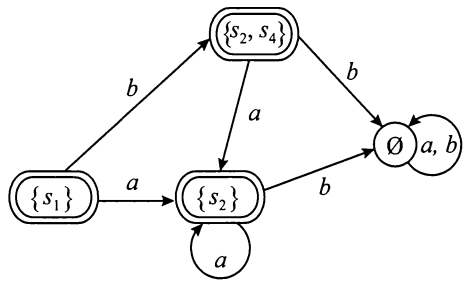


Рис. 9.10. Детерминированный конечный автомат M''

9.2. Распознавание образцов, задаваемых регулярными выражениями

Рассмотрим задачу распознавания образцов, в которой дана *текстовая строка* $x = a_1a_2...a_n$ и регулярное выражение α, называемое *образцом*. Требуется найти такой наименьший индекс j, что для некоторого i подстрока $a_ia_{i+1}...a_j$ строки x принадлежит языку, представленному выражением α.

Вопросы такого рода часто возникают при редактировании текстов. Многие программы для редактирования текстов разрешают пользователю задавать типы

замен в текстовой строке. Например, пользователь может сказать, что он хочет заменить слово y каким-то другим словом во фрагменте текста x. Чтобы выполнить такую команду, программа редактирования текста должна уметь находить вхождение слова y в x. Некоторые сложные редактирующие программы разрешают пользователю в качестве множества заменяемых строк указывать регулярное множество. Например, пользователь может указать: "Заменить [I*] в тексте x пустой строкой", имея в виду, что в тексте x необходимо стереть пару квадратных скобок и символы между ними.

Поставленную выше задачу можно переформулировать, заменив заданное регулярное выражение α выражением $\beta = I^*\alpha$, где I — алфавит текстовой строки. Можно найти первое вхождение строки из языка $L(a)$ в $x = a_1a_2...a_n$, обнаружив кратчайший префикс строки x, принадлежащий языку, представленному выражением β. Эту задачу можно решить, сначала построив недетерминированный конечный автомат M для распознавания множества, представленного выражением β, а затем применив алгоритм 9.1 (см. ниже) для определения последовательности множеств состояний S_j, в которые может перейти недетерминированный конечный автомат M после прочтения строки $a_1a_2...a_i$ при $i = 1, 2, ..., n$. Если во множестве S_j оказывается заключительное состояние, значит, в строке $a_1a_2...a_i$ есть суффикс $a_1a_{i+1}...a_j$, принадлежащий языку, представленному выражением α, при некотором $1 \le i \le j$. Способы нахождения i, т.е. левого конца образца, обсуждается в упражнениях 9.6–9.8.

Один из способов моделирования поведения недетерминированного конечного автомата M на текстовой строке x основан на превращении M в детерминированный конечный автомат, как в теореме 9.3. Однако такой путь может оказаться слишком дорогим, поскольку от регулярного выражения β можно перейти к недетерминированному конечному автомату с $2|\beta|$ состояниями, а затем к детерминированному конечному автомату с примерно $2^{2|\beta|}$ состояниями. Уже само построение детерминированного конечного автомата может оказаться невыполнимым.

Другой способ моделирования поведения недетерминированного конечного автомата M на строке x предусматривает предварительное исключение ε-переходов в M и построение недетерминированного конечного автомата M' без пустой строки, как это было сделано в теореме 9.3. Затем можно моделировать недетерминированный конечный автомат M' на входной строке $x = a_1a_2...a_n$, вычисляя для каждого i, $1 \le i \le n$ множество состояний S_i, в которые мог бы попасть автомат M' после прочтения строки $a_1a_2...a_i$. На самом деле каждое множество S_i — это то состояние, в которое пришел бы детерминированный конечный автомат M'', построенный в доказательстве теоремы 9.3, после прочтения строки $a_1a_2...a_i$.

Применяя этот способ, можно не строить автомат M'', а вычислять только те его состояния, которые появляются при обработке строки x. Чтобы объяснить, как найти множество S_i, необходимо показать, как построить S_i по S_{i-1}. Легко видеть, что

$$S_i = \bigcup_{s \in S_{i-1}} \delta'(s, a_i),$$

где δ' — функция переходов автомата M'. Тогда S_i — это объединение вплоть до $2|\beta|$ множеств, каждое из которых содержит не больше $2|\beta|$ членов. Поскольку при объединении необходимо исключить дубликаты (иначе представление множеств может стать громоздким), очевидное моделирование автомата M' занимает $O(|\beta|^2)$ шагов на входной символ, т.е. в общей сложности $O(n|\beta|^2)$ шагов.

Парадоксальным образом на практике часто гораздо эффективнее не исключать ε-переходы, а работать непосредственно с недетерминированным конечным автоматом M, построенным по теореме 9.2 из регулярного выражения β. Ключевое свойство в теореме 9.2 заключается в том, что на диаграмме переходов из каждого состояния автомата M выходит не более двух ребер. Это свойство позволяет показать, что на входной строке $x = a_1a_2...a_n$ алгоритм 9.1 (см. ниже) тратит на моделирование недетерминированного конечного автомата, построенного по β, $O(n|\beta|)$ шагов.

Алгоритм 9.1. Моделирование недетерминированного конечного автомата

Вход. Недетерминированный конечный автомат $M = (S, I, \delta, s_0, F)$ и строка $x = a_1a_2...a_n$ из I^*.

Выход. Последовательность $S_0, S_1, S_2, ..., S_n$, где

$$S_i = \{s | (s_0, a_1a_2...a_i) \vdash^* (s, \varepsilon)\}, \ 0 \le i \le n.$$

Метод. Чтобы вычислить S_i по S_{i-1}, сначала найдем множество состояний $T_i = \{t | \delta(s, a_i)$ содержит t для некоторого $s \in S_{i-1}\}$. Затем вычислим замыкание множества T_i, добавив к T_i все такие состояния u, что множество $\delta(t, \varepsilon)$ содержит u для некоторого t, ранее добавленного в T_i. Это замыкание, которое и является множеством S_i, строится с помощью очереди состояний $t \in T_i$, для которых множество $\delta(t, \varepsilon)$ еще не рассматривалось. Алгоритм приведен на рис. 9.11. □

Пример 9.6. Пусть M — недетерминированный конечный автомат, изображенный на рис. 9.6, *в*. Допустим, входная строка имеет вид $x = ab$. Тогда $S_0 = \{s_1\}$ в строке 2. В строках 9–12, рассмотрев состояния s_1, добавляем s_2 и s_8 в список QUEUE и в S_0. Рассмотрев s_2 и s_8, мы ничего не добавляем, поэтому $S_0 = \{s_1, s_2, s_8\}$. Затем в строке 3: $S_1 = \{s_3\}$. Рассмотрев s_3, добавляем s_4 в S_1, а рассмотрев s_4, добавляем s_5 и s_7. Рассмотрев s_5, мы не добавляем ничего, а рассмотрев s_7, добавляем s_{10}. Таким образом, $S_1 = \{s_3, s_4, s_5, s_7, s_{10}\}$. Затем в строке 3: $S_2 = \{s_6\}$. Рассмотрев s_6, добавляем s_5 и s_7 в S_2. Рассмотрев s_7, добавляем s_{10}. Следовательно, $S_2 = \{s_5, s_6, s_7, s_{10}\}$.

```
1.    for i ← 0 until n do
          begin
2.            if i = 0 then S_i ← {s_0}
3.            else S_i ← ⋃_{s∈S_{i-1}} δ(s, a_i);
              comment S_i еще не достигло своего конечного значения.
                  Сейчас оно соответствует множеству T_i, упомянутому
                  выше;
4.            пометить каждое состояние t ∈ S_i как "рассмотренное";
5.            пометить каждое состояние t ∈ S - S_i как
                                  "нерассмотренное";
6.            QUEUE ← S_i;
7.            while список QUEUE не пуст do
                  begin
8.                    найти и удалить первый элемент t, входящий
                                                      в QUEUE;
9.                    for u ∈ δ(t, ε) do
10.                       if u — "нерассмотренное" состояние then
                              begin
11.                               пометить u как "рассмотренное";
12.                               добавить u в QUEUE и в S_i
                              end
                  end
          end
```

Рис. 9.11. Моделирование недетерминированного конечного автомата

Теорема 9.4. Алгоритм 9.1 правильно вычисляет последовательность $S_0, S_1, ..., S_n$, где $S_i = \{s | (s_0, a_1a_2...a_i) \vdash^* (s, \varepsilon)\}$.

Доказательство. Доказательство корректности алгоритма 9.1 — это простое упражнение на применение индукции, которое мы оставляем читателю. □

Теорема 9.5. Пусть в диаграмме переходов автомата M с m состояниями из каждой вершины исходит не более e ребер. Тогда на входной строке длины n алгоритм 9.1 делает $O(emn)$ шагов.

Доказательство. Рассмотрим вычисление множества S_i для конкретного значения i. Строки 8–12 на рис. 9.11 требуют $O(e)$ шагов. Поскольку для данного i никакое состояние не попадает в список QUEUE дважды, то цикл в строках 7–12 требует $O(em)$ времени. Следовательно, легко показать, что тело основного

цикла, т.е. строки 2–12, требуют $O(em)$ времени. Таким образом, весь алгоритм занимает $O(emn)$ времени. □

Отсюда следует важный результат, связывающий распознавание вхождения элементов регулярного множества с алгоритмом 9.1.

Следствие. Если β — произвольное регулярное выражение и $x = a_1a_2....a_n$ — строка длины n, то найдется недетерминированный конечный автомат M, допускающий язык, представленный выражением β, причем алгоритм 9.1 затрачивает на построение последовательности $S_0, S_1, ..., S_n$, где $S_i = \{ |(s_0, a_1a_2... a_i) \vdash^* (s, \varepsilon)\}$ для $0 \le i \le n$, время $O(n|\beta|)$.

Доказательство. По теореме 9.2 можно построить автомат M, у которого не более $2|\beta|$ состояний и из каждого из них выходит не более двух ребер. Поэтому e в теореме 9.5 не превосходит 2. □

На основе алгоритма 9.1 можно построить различные алгоритмы распознавания образцов. Например, пусть даны регулярное выражение α и текстовая строка $x = a_1a_2...a_n$ и требуется найти наименьшее число k, для которого существует такое $j < k$, что $a_ja_{j+1}...a_k$ принадлежит множеству, представленному выражением α. С помощью теоремы 9.2 по выражению α можно построить недетерминированный конечный автомат M, допускающий язык $I^*\alpha$. Чтобы найти наименьшее число k, для которого $a_1a_2...a_k$ принадлежит $L(M)$, можно после блока, состоящего из строк 2–12 на рис. 9.11, вставить тест, проверяющий, содержит ли множество S_i состояние из F. В силу теоремы 9.2 можно сделать F одноэлементным, так что такая проверка не займет много времени: достаточно $O(m)$ шагов, где m — количество состояний автомата M. Если S_i содержит состояние из F, то прерываем основной цикл и $a_1a_2...a_i$ — кратчайший префикс строки x, который входит в $L(M)$.

Алгоритм 9.1 можно модифицировать так, чтобы он находил для каждого k такое наибольшее $j < k$ (или наименьшее j), что $a_ja_{j+1}...a_k$ входит в множество, представленное выражением α. Это делается путем сопоставления целого числа каждому состоянию из множеств S_i. Число, сопоставленное состоянию $s \in S_k$, указывает такое наибольшее (или наименьшее) j, что $(s_0, a_ja_{j+1}...a_k) \vdash^* (s, \varepsilon)$. Детали сопоставления этих целых чисел с помощью алгоритма 9.1 оставляем читателям в качестве упражнения.

9.3. Распознавание подстрок

Важным частным случаем общей задачи, описанной в предыдущем разделе, является случай, когда регулярное выражение α имеет вид $y_1 + y_2 + ... + y_k$, где каждый член y_i является строкой в алфавите I. Из следствия теоремы 9.5 вытекает, что первое вхождение шаблона y_i в текстовую строку $x = a_1a_2...a_n$ можно найти за

$O(ln)$ шагов, где l — сумма длин строк y_i. Однако возможно решение, имеющее сложность $O(l + n)$. Сначала рассмотрим случай, когда дана только одна строка-образец $y = b_1b_2...b_l$, где каждый символ b_i принадлежит I.

По строке y построим детерминированную машину M_y для сопоставления строк, которая распознает кратчайшую строку из I^* y, входящую в x. Чтобы построить M_y, сначала построим *скелетный детерминированный конечный автомат* с $l + 1$ состояниями 0, 1, ..., l, который переходит из состояния $i - 1$ в состояние i на входном символе b_i, как показано на рис. 9.12. Состояние 0 имеет также переход в себя при всех $b \neq b_1$. Состояние i можно считать указателем на iю позицию шаблонной строки y.

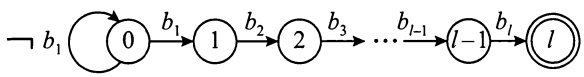


Рис. 9.12. Скелетная машина

Машина для распознавания строк M_y работает как детерминированный конечный автомат с той лишь разницей, что, просматривая один входной символ, она может сделать несколько переходов (изменений состояния). Машина M_y имеет то же множество состояний, что и скелетная машина. Поэтому состояние j машины M_y соответствует префиксу строки-образца y.

Машина M_y начинает работу в состоянии 0 и с входным указателем, установленным на a_1, т.е. на первый символ текстовой строки $x = a_1a_2...a_n$. Если $a_1 = b_1$, то машина M_y переходит в состояние 1 и сдвигает входной указатель на вторую позицию текстовой строки. Если $a_1 \neq b_1$, то машина M_y остается в состоянии 0 и сдвигает входной указатель на вторую позицию.

Допустим, что машина M_y после прочтения строки $a_1a_2...a_k$ находится в состоянии j. Это означает, что последние j символов строки $a_1a_2...a_k$ совпадают с $b_1b_2...b_k$ и последние m символов в строке $a_1a_2...a_k$ не являются префиксом строки $b_1b_2...b_j$ для $m > j$. Если $a_{k+1} = b_{j+1}$, т.е. следующий входной символ совпадает с b_{j+1}, то M_y переходит в состояние $j + 1$ и сдвигает входной указатель на a_{k+2}. Если $a_{k+1} \neq b_{j+1}$, то машина M_y переходит в состояние i с наибольшим номером, для которого $b_1b_2...b_i$ является суффиксом строки $a_1a_2...a_{k+1}$.

Чтобы облегчить нахождение состояния i, с машиной M_y ассоциируется функция f, принимающая целочисленные значения. Она называется *функцией отказов* и задается так: $f(j)$ — наибольшее число s, меньше, чем j, для которого $b_1b_2...b_s$ является суффиксом строки $b_1b_2...b_j$. Иначе говоря, $f(j)$ — наибольшее значение $s < j$, такое, что $b_1b_2...b_s = b_{j-s+1}b_{j-s+2}...b_j$. Если такого $s \geq 1$ нет, то $f(j) = 0$.

Пример 9.7. Допустим, что $y = aabbaab$. Функция f принимает значения

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

Например, $f(6) = 2$, поскольку aa — самый длинный собственный префикс строки $aabbaa$, который является ее суффиксом. □

Алгоритм вычисления функции отказов мы изложим несколько позже. Сейчас, чтобы увидеть, как используется функция отказов, определим функцию $f^{(m)}(j)$:

1. $f^{(1)}(j) = f(j)$ и

2. $f^{(m)}(j) = f(f^{(m-1)}(j))$, где $m > 1$.

Иначе говоря, $f^{(m)}(j)$ — это та же функция f, примененная m раз к j. (В примере 9.7 $f^{(2)}(6) = 1$.)

Допустим, что машина M_y после прочтения строки $a_1a_2...a_k$ находится в состоянии j и $a_{k+1} \neq b_{j+1}$.

Таким образом, машина M_y проходит через состояния $f^{(1)}(j)$, $f^{(2)}(j)$, ... до тех пор, пока не будет обнаружено состояние, в котором условие 1 или 2 будет выполнено для $f^{(m)}(j)$, но не для $f^{(m-1)}(j)$. Если выполнено условие 1, то машина M_y переходит в состояние $u + 1$. Если выполнено условие 2, то машина M_y переходит в состояние 0. В любом случае входной указатель перемещается на позицию a_{k+2}.

В случае 1 легко проверить, что поскольку $b_1b_2...b_j$ — самый длинный префикс строки y, одновременно являющийся суффиксом строки $a_1a_2...a_k$, то $b_1b_2...b_{f^{(m)}(j)+1}$ — самый длинный префикс строки $a_1a_2...a_{k+1}$. В случае 2 никакой префикс строки y не является суффиксом строки $a_1a_2...a_{k+1}$.

Затем машина M_y обрабатывает входной символ a_{k+2} и продолжает работать по такому алгоритму, пока не перейдет в заключительное состояние l, в котором l последних просмотренных входных символов образуют экземпляр образца $y = b_1b_2...b_l$, либо обработает последний входной символ из x и не попадет в состояние l, т.е. строка y не является подстрокой строки x.

Пример 9.8. Пусть $y = aabbaab$. Машина для распознавания строк-образцов M_y приведена на рис. 9.13. Штриховые стрелки указывают значения функции отказов для всех состояний. На входе $x = abaabaabbaab$ машина M_y пройдет через такую последовательность состояний:

Вход		a	b	a	a	b	a	a	b	b	a	a	b	
Состояние	0	1	0	1	2	3	1	2	3	4	5	6	7	
		0				0								

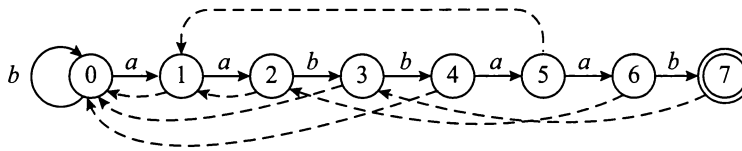


Рис. 9.13. Машина для распознавания строк-образцов

Например, вначале машина M_y находится в состоянии 0. Прочитав первый символ строки x, она переходит в состояние 1. Поскольку из состояния 1 нет перехода по второму входному символу b, то M_y переходит в состояние 0, т.е. в значение функции отказов для состояния 1, при этом входной указатель не сдвигается. Поскольку первый символ строки y отличается от b, то выполнено условие 2, и M_y остается в состоянии 0, и сдвигает входной указатель на позицию 3.

Прочитав 12-й входной символ, машина M_y перейдет в заключительное состояние 7. Таким образом, дойдя до позиции 12 в строке x, машина M_y нашла вхождение строки-образца y. □

Функцию f можно итеративно вычислить почти по той же схеме, по какой работает машина M_y. По определению $f(1) = 0$. Допустим, мы уже вычислили значения $f(1), f(2), ..., f(j)$. Пусть $f(j) = i$. Чтобы вычислить $f(j + 1)$, исследуем b_{j+1} и b_{i+1}. Если $b_{j+1} = b_{i+1}$, то $f(j + 1) = f(j) + 1$, поскольку

$$b_1 b_2 ... b_i b_{i+1} = b_{j-i+1} b_{j-i+2} ... b_j b_{j+1}.$$

Если $b_{j+1} \neq b_{i+1}$, то находим наименьшее m, для которого либо

1. $f^{(m)}(j) = u$ и $b_{j+1} = b_{u+1}$, либо

2. $f^{(m)}(j) = 0$ и $b_{j+1} \neq b_1$.

В случае 1 полагаем $f(j + 1) = u + 1$. В случае 2 полагаем $f(j + 1) = 0$. Детали даны в следующем алгоритме.

Алгоритм 9.2. Вычисление функции отказов

Вход. Шаблонная строка $y = b_1 b_2 ... b_l$, $l \geq 1$.

Выход. Функция отказов f для y.

Метод. Выполнение программы, представленной на рис. 9.14. □

Пример 9.9. Рассмотрим поведение алгоритма 9.2 на входе $y = aabbaab$. В результате инициализации имеем $f(1) = 0$. Поскольку $b_2 = b_1$, то $f(2) = 1$. Однако $b_3 \neq b_2$ и $b_3 \neq b_1$, так что $f(3) = 0$. Продолжая в том же духе, получаем значения f, указанные в примере 9.7. □

```
      begin
1.        f(1) ← 0;
2.        for j ← 2 until l do
              begin
3.                i ← f(j - 1);
4.                while b_j ≠ b_{i+1} и i > 0 do i ← f(i);
5.                if b_j ≠ b_{i+1} и i = 0 then f(j) ← 0
6.                else f(j) ← i + 1
              end
      end
```

Рис. 9.14. Вычисление функции отказов

Докажем, что алгоритм 9.2 правильно вычисляет функцию f за время $O(|y|)$. Сначала докажем корректность алгоритма.

Теорема 9.6. Алгоритм 9.2 вычисляет функцию f.

Доказательство. Докажем индукцией по j, что $f(j)$ — такое наибольшее целое $i < j$, что $b_1b_2...b_i = b_{j-i+1}b_{j-i+2}...b_j$. Если такого i нет, то $f(j) = 0$. По определению $f(1) = 0$. Допустим, что предположение индукции верно для всех $k < j$. При вычислении $f(j)$ алгоритм 9.2, выполняя строку 4, сравнивает b_j с $b_{f(j-1)+1}$.

Вариант 1. Пусть $b_j = b_{f(j-1)+1}$. Поскольку $f(j-1)$ — это такое наибольшее i, что $b_1...b_i = b_{j-i}...b_{j-1}$, равенство $f(j) = i + 1$ выполняется. Таким образом, в строках 5 и 6 $f(j)$ вычисляется правильно.

Вариант 2. Пусть $b_j \neq b_{f(j-1)+1}$. Тогда необходимо найти наибольшее значение i, для которого $b_1...b_i = b_{j-i}...b_{j-1}$ и $b_{i+1} = b_j$, если такое i существует. Если такого i нет, то очевидно, что $f(j) = 0$, и $f(j)$ правильно вычисляется в строке 5. Пусть i_1, i_2, ... — наибольшее, второе по величине и т.д. значения i, для которых

$$b_1...b_i = b_{j-i}...b_{j-1}.$$

С помощью простой индукции убеждаемся, что $i_1 = f(j-1)$, $i_2 = f(i_1) = f^{(2)}(j-1)$, ..., $i_k = f(i_{k-1}) = f^{(k)}(j-1)$, поскольку i_{k-1} — это $(k-1)$-е по величине значение i, для которого $b_1...b_i = b_{j-i}...b_{j-1}$, а i_k — наибольшее значение $i < i_{k-1}$, для которого $b_1...b_i = b_{i_{k-1}-i+1}...b_{i_{k-1}} = b_{j-i}...b_{j-1}$. Строка 4 просматривает i_1, i_2, ... по очереди, пока не найдет такое i, что $b_1...b_i = b_{j-i}...b_{j-1}$ и $b_{i+1} = b_j$, если такое i существует. По окончании выполнения оператора **while** выполняется условие $i = i_m$, если такое i_m существует, и, значит, $f(j)$ правильно вычисляется в строке 5.

Таким образом, значение $f(j)$ правильно вычисляется для всех j. □

Теорема 9.7. Алгоритм 9.2 вычисляет f за $O(l)$ шагов.

Доказательство. Строки 3 и 5 имеют фиксированную сложность. Сложность оператора **while** пропорциональна числу уменьшений значения i оператором $i \leftarrow f(i)$, который стоит после **do** в строке 4. Единственный способ увеличить i — это присвоить $f(j) = i + 1$ в строке 6, затем увеличить j на 1 в строке 2 и присвоить $i = f(j - 1)$ в строке 3. Поскольку вначале $i = 0$, а строка 6 выполняется не более $l - 1$ раз, заключаем, что оператор **while** в строке 4 не может выполняться более l раз. Поэтому строка 4 требует $O(l)$ времени. Остальная часть алгоритма, очевидно, имеет сложность $O(l)$, поэтому весь алгоритм тратит $O(l)$ времени. □

С помощью тех же рассуждений, что и в теореме 9.6, можно доказать, что после прочтения строки $a_1a_2...a_k$ машина для распознавания образцов M_y будет находиться в состоянии i тогда и только тогда, когда $b_1b_2...b_i$ — самый длинный префикс строки y, который является суффиксом строки $a_1a_2...a_k$. Поэтому машина M_y правильно находит самое левое вхождение строки y в текстовую строку $x = a_1a_2...a_n$.

С помощью тех же рассуждений, что и в теореме 9.7, можно доказать, что при обработке входной строки x машина M_y изменит свое состояние не более $2|x|$ раз. Поэтому можно узнать, является ли y подстрокой строки x, проследив изменения состояния машины M_y на входе x.[3] Для этого необходимо лишь знать значение функции отказов на строке y. По теореме 9.7 эти значения функции f можно найти за время $O(|y|)$. Следовательно, узнать, является ли y подстрокой строки x, можно за время $O(|x| + |y|)$, не зависящее от размера алфавита. Если же алфавит строки-образца мал, а текстовая строка значительно длиннее шаблона, то можно смоделировать некоторый детерминированный конечный автомат, допускающий язык I^*y. Этот детерминированный конечный автомат только один раз меняет состояние на каждом входном символе.

Алгоритм 9.3. Построение детерминированного конечного автомата для I^*y.

Вход. Строка-образец $y = b_1b_2...b_l$ в алфавите I. Для удобства вводим новый символ b_{l+1}, не совпадающий ни с одним символом из алфавита I.

Выход. Детерминированный конечный автомат M, для которого $L(M) = I^*y$.

Метод.

1. С помощью алгоритма 9.2 строим функцию отказов f для y.
2. Пусть $M = (S, I, \delta, 0, \{l\})$, где $S = \{0, 1, ..., l\}$, а функция δ определяется так:

[3] Напомним, что состояние машины M_y — это указатель позиции в шаблонной строке y. Поэтому изменение состояния машины M_y можно реализовать, непосредственно перемещая указатель по строке y.

```
begin
    for j = 1 until l do δ(j - 1, b_j) ← j;
    for b ∈ I, b ≠ b_1 do δ(0, b) ← 0;
    for j = 1 until l do
        for b ∈ I, b ≠ b_{j+1} do δ(j, b) ← δ(f(j), b)
end □
```

Теорема 9.8. Алгоритм 9.3 строит такой детерминированный конечный автомат M, что

$$(0, a_1a_2...a_k) \vdash^* (j, \varepsilon),$$

тогда и только тогда, когда $b_1b_2...b_j$ — суффикс строки $a_1a_2...a_k$, но $b_1b_2...b_i$ при $i > j$ не является суффиксом строки $a_1a_2...a_k$.

Доказательство. Проводится индукцией по k с помощью тех же рассуждений, что и в теореме 9.6. Предоставляем его читателю.

Пример 9.10. Детерминированный конечный автомат M для $y = aabbaab$, построенный алгоритмом 9.3, изображен на рис. 9.15.

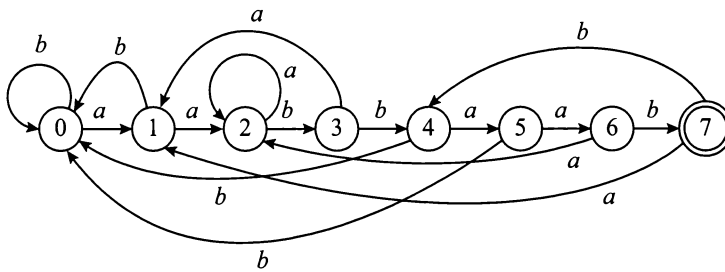


Рис. 9.15. Детерминированный конечный автомат M для $(a + b)^* aabbaab$

На входе $x = abaabaabbaab$ автомат M делает такие переходы:

Вход	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*b*	*a*	*a*	*b*
Состояние	0	1	0	1	2	3	1	2	3	4	5	6	7

Единственное отличие между машинами M и M_y состоит в том, что машина M заранее вычисляет состояние, в которое следует перейти в случае несовпадения. Таким образом, машина M делает только один переход на каждом входном символе. □

Основные результаты раздела формулируются в следующей теореме.

Теорема 9.9. Выяснить, является ли строка y подстрокой строки x, можно за время $O(|x| + |y|)$.

Теперь рассмотрим случай распознавания нескольких строк-образцов $y_1, y_2, ..., y_k$. Наша задача — распознать, входит ли одна из них в заданную строку $x = a_1a_2...a_n$. Для решения этой задачи также можно применить методы, описанные в этом разделе. Мы можем построить скелетную машину для распознавания строк-образцов $y_1, y_2, ..., y_k$. Эта машина будет деревом. Затем мы можем вычислить функцию отказов на этом дереве за время, пропорциональное $l = |y_1| + |y_2| + ... + |y_k|$. Машину для распознавания образцов можно построить точно так же, как было описано выше. Тогда за $O(l + n)$ шагов мы узнаем, является ли какая-нибудь строка подстрокой строки x. Детали оставляем читателям в качестве упражнения.

9.4. Двусторонний детерминированный магазинный автомат

Зная о существовании алгоритма, распознающего, входит ли строка y в строку x за время $O(|x| + |y|)$, его нетрудно построить. Но что дает нам основания предполагать существование такого алгоритма? Один из возможных подходов к ответу на этот вопрос основан на изучении двусторонних детерминированных магазинных автоматов.

Двусторонний детерминированный магазинный автомат представляет собой специальный тип машины Тьюринга, допускающей язык. Многие задачи распознавания образцов можно переформулировать в терминах задач распознавания языков. Например, пусть L — язык $\{xcy|x, y \in I^*, c \notin I$ и y — подстрока строки $x\}$. Тогда распознавание того, входит ли строка y в строку x, эквивалентно распознаванию принадлежности строки xcy языку L. В этом разделе мы покажем, что существует двусторонний детерминированный магазинный автомат, способный распознать язык L. Хотя этот автомат может затратить $O(n^2)$ времени, известен мощный метод моделирования, позволяющий моделировать поведение данного автомата на входной строке длины n на машине RAM, которая затратит на это $O(n)$ шагов. В настоящем разделе мы подробно изучим эту технику моделирования.

Двусторонний детерминированный магазинный автомат можно рассматривать как двухленточную машину Тьюринга, одна из лент которой используется в качестве магазинной памяти (рис. 9.16). Двусторонний детерминированный магазинный автомат имеет входную ленту, предназначенную только для чтения, у которой крайняя левая ячейка помечена концевым маркером ¢, а крайняя правая — концевым маркером $. Распознаваемая входная строка располагается между этими

двумя концевыми маркерами по одному символу в ячейке. Входные символы извлекаются из входного алфавита I, который по предположению не содержит ¢ и \$. Считывающая головка может за один шаг прочесть один символ и сдвинуться на одну ячейку влево, вправо или остаться на месте. Мы предполагаем, что головка не может сойти ни с одного конца входной ленты; наличие концевых маркеров ¢ и \$ позволяет так построить функцию переходов, чтобы она никогда не сдвигала считывающую головку влево от ¢ и вправо от \$.

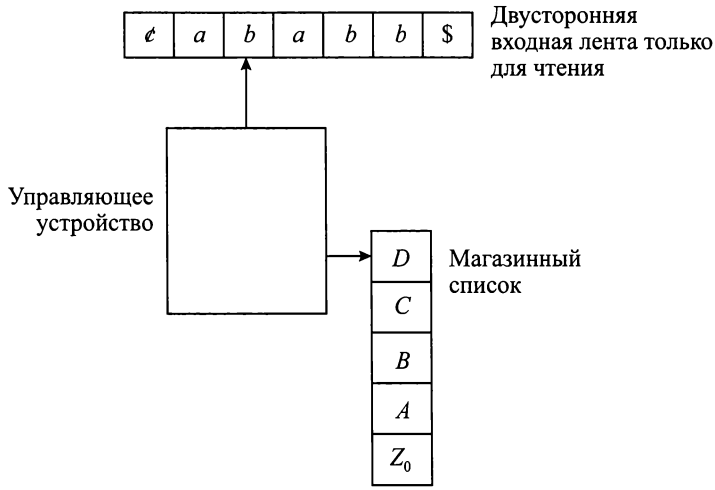


Рис. 9.16. Двусторонний детерминированный магазинный автомат

Магазинный список — это стек, содержащий символы из магазинного алфавита T. Нижняя ячейка магазинного списка содержит специальный символ Z_0, отмечающий дно стека. Мы предполагаем, что Z_0 не принадлежит алфавиту T.

Управляющее устройство всегда находится в одном из состояний, принадлежащих конечному множеству S. Работа машины определяется функцией переходов δ, которая для $s \in S$, $a \in I \cup \{¢, \$\}$ и $A \in T \cup \{Z_0\}$ указывает действие машины в случае, когда управляющее устройство находится в состоянии s, считывающая головка обозревает символ a и в вершине магазинного списка расположен символ A. Возможны три типа действий машины:

$$\delta(s,a,A) = \begin{cases} (s', d, \textbf{push}\ B), \text{ если } B \neq Z_0; \\ (s', d); \\ (s', d, \textbf{pop}), \text{ если } A \neq Z_0. \end{cases}$$

При любом из этих действий машина переходит в состояние s' и сдвигает считывающую головку в направлении d (где $d = -1$, $+1$ или 0, что означает сдвиг на одну ячейку влево, вправо или отсутствие сдвигов); **push** B означает добавление символа B в вершину магазинного списка; **pop** — удаление самого верхнего символа из магазина.

Определение. Формально двусторонний детерминированный магазинный автомат определяется как семерка

$$P = (S, I, T, \delta, s_0, Z_0, s_f),$$

где

1. S — конечное множество *состояний управляющего устройства.*
2. I — *входной алфавит* (не содержащий символов ¢ и $).
3. T — *алфавит магазинного списка* (не содержащий символ Z_0).
4. δ — отображение множества $(S - \{s_f\}) \times (I \cup \{¢, \$\}) \times (T \cup \{Z_0\})$. Значение $\delta(s, a, A)$, если оно определено, имеет один из следующих видов: $(s', d,$ **push** $B)$, (s', d) или $(s', d,$ **pop**$)$, где $s' \in S$, $B \in T$ и $d \in \{-1, 0, +1\}$. Будем предполагать, что в заключительном состоянии s_f двусторонний детерминированный магазинный автомат не делает никаких шагов, для некоторых других состояний шаги действия могут быть не определены. Кроме того, для любых s и A второй компонент значения $\delta(s, ¢, A)$ не может быть равен $d = -1$, а второй компонент значения $\delta(s, \$, A)$ не может быть равен $d = +1$. Наконец, третий компонент значения $\delta(s, a, Z_0)$ не содержит функцию **pop**.
5. $s_0 \in S$ — *начальное состояние* управляющего устройства.
6. Z_0 — *маркер дна* магазинного списка.
7. s_f — выделенное *заключительное состояние.*

Мгновенным описанием двустороннего детерминированного магазинного автомата P на входе $w = a_1a_2...a_n$ называется тройка (s, i, α), где

1. s — состояние из S.
2. i — целое число, $0 \le i \le n + 1$, указывающее положение считывающей головки (считаем, что $a_0 = ¢$ и $a_{n+1} = \$$).
3. α — строка, представляющая содержимое магазинного списка, причем самый левый символ в строке α расположен в вершине этого списка.

Шаг двустороннего детерминированного магазинного автомата на входе $a_1a_2...a_n$ определяется с помощью бинарного отношения $\vdash$ на мгновенных описаниях:

1. $(s, i, A\alpha) \vdash (s', i + d, BA\alpha)$, если $\delta(s, a_i, A) = (s', d, \textbf{push } B)$;
2. $(s, i, A\alpha) \vdash (s', i + d, A\alpha)$, если $\delta(s, a_i, A) = (s', d)$;
3. $(s, i, A\alpha) \vdash (s', i + d, \alpha)$, если $\delta(s, a_i, A) = (s', d, \textbf{pop})$.

Заметим, что добавление, считывание и удаление символов происходит только в вершине магазинного списка. Кроме того, требуется, чтобы в любой момент времени в магазинном списке двустороннего детерминированного магазинного автомата был только один маркер дна. Для обозначения последовательностей из нуля или более шагов двустороннего детерминированного магазинного автомата используется знак рефлексивного и транзитивного замыкания $\vdash^*$ отношения $\vdash$.

Начальное мгновенное описание двустороннего детерминированного магазинного автомата P для $w = a_1a_2...a_n$ (за исключением концевых маркеров) имеет вид $(s_0, 1, Z_0)$, показывающий, что P находится в своем начальном состоянии s_0, считывающая головка обозревает крайний левый символ из $a_1a_2...a_na_{n+1}$ и магазин содержит только маркер дна Z_0.[4]

Допускающее мгновенное описание для $w = a_1a_2...a_n$ имеет вид (s_f, i, Z_0) для некоторого $0 \le i \le n + 1$, показывающий, что автомат P перешел в заключительное состояние s_f и магазин содержит только маркер дна.

Говорят, что двусторонний детерминированный магазинный автомат P *допускает* входную строку w, если при входе w

$$(s_0, 1, Z_0) \vdash^* (s_f, i, Z_0)$$

для некоторого $0 \le i \le |w| + 1$. *Языком* $L(P)$, *допускаемым* автоматом P, называется множество всех строк, допускаемых автоматом P.

Рассмотрим несколько примеров двусторонних детерминированных магазинных автоматов и допускаемых ими языков. Двусторонний детерминированный магазинный автомат будем описывать неформально — в терминах их поведения, а не с помощью кортежа из семи элементов.

Пример 9.11. Рассмотрим язык $L = \{xcy | x, y \in \{a, b\}^*$ и y — подстрока строки $x\}$. Двусторонний детерминированный магазинный автомат P может распознать L следующим образом. Допустим, что автомату P дана входная строка w вида xcy, где $x = a_1a_2...a_n$ и $a_i \in \{a, b\}$, $1 \le i \le n$.

1. P сдвигает свою считывающую головку вправо, пока не встретит c.
2. P сдвигает свою считывающую головку влево, копируя $a_na_{n-1}...a_1$ в магазин, пока считывающая головка не достигнет левого концевого маркера. В этот

---

[4] Если $n = 0$, т.е. $w = \varepsilon$, то P обозревает $a_{n+1} = \$$, т.е. правый концевой маркер.

момент в магазинном списке содержится строка $xZ_0 = a_1a_2...a_nZ_0$ (самый левый символ строки xZ_0 расположен в вершине магазинного списка).

3. Затем P сдвигает свою считывающую головку к первому символу, стоящему справа от c (т.е. к началу строки y), не трогая магазинный список, и готовится сравнивать y с магазинным списком.

4. **while** символ в вершине магазина совпадает с символом, обозреваемым входной головкой **do**
   begin
   удалить верхний символ из магазинного списка;
   сдвинуть входную головку на одну ячейку вправо
   end

5. В этот момент возможны два случая.

   а) Считывающая головка достигла правого концевого маркера \$, и тогда автомат P допускает входную строку.

   б) Символ в вершине магазина не совпадает с текущим входным символом. Поскольку строка y и магазинный список до этого момента совпадали, автомат P может восстановить магазинный список, сдвигая считывающую головку влево и копируя входную в магазинный список, пока не встретит символ c. В этот момент в магазинном списке автомата P содержится строка $a_ia_{i+1}...a_nZ_0$ при некотором $1 \le i \le n$.[5] Если $i = n$, то P останавливается в состоянии, которое не является заключительным. В противном случае P удаляет a_i из магазина и сдвигает считывающую головку вправо к первому символу строки y\$. Затем P возвращается к шагу 4, пытаясь установить, не является ли строка y префиксом строки $a_{i+1}...a_n$.

Мы видим, что автомат P распознает, является ли y подстрокой строки $x = a_1a_2...a_n$, вполне естественным способом. Автомат P пытается по очереди сравнивать строку y с префиксом строки $a_ia_{i+1}...a_n$ для $i = 1, 2, ..., n$. Если это ему удается, то он допускает входную строку. Если нет — то отвергает. Заметим, что на эту процедуру автомат P может затратить $O(|x| \cdot |y|)$ шагов. □

Пример 9.12. Рассмотрим язык $L = \{w | w \in \{a, b, c\}^*, w = xy$, где $|x| \ge 2$ и $x^R = x\}$ (т.е. строка w имеет префикс длиной не менее двух символов, являющийся палиндромом). Здесь верхний индекс R обозначает обращение строки, например $(abb)^R = bba$. Требование $|x| \ge 2$ гарантирует, что всякая строка из $\{a, b, c\}^+$ начинается с одного из тривиальных палиндромов a, b или c. Рассмотрим двусторонний детерминиро-

---

[5] Когда мы попадаем в такую ситуацию первый раз, выполняется условие $i = 1$, но впоследствии i будет возрастать каждый раз на единицу.

ванный магазинный автомат P, который на заданной входной строке $a_1a_2...a_n$ ведет себя следующим образом.

1. P сдвигает считывающую головку к правому концевому маркеру, копируя входную строку в магазинный список. Теперь магазинный список содержит строку $a_n...a_2a_1Z_0$. Если $n < 2$, то автомат P сразу отвергает входную строку.
2. Затем P сдвигает считывающую головку к левому концевому маркеру, не трогая магазинный список. P устанавливает считывающую головку на символ, стоящий непосредственно справа от левого концевого маркера.
3. **while** символ в вершине магазина совпадает с символом, обозреваемым считывающей головкой **do**
   begin
   удалить верхний символ из магазина;
   сдвинуть считывающую головку на одну ячейку вправо
   end
4. В этот момент возможны два случая.
   а) Магазинный список содержит только символ Z_0, и тогда автомат P останавливается и допускает входную строку.
   б) Символ в вершине магазинного списка не совпадает с текущим входным символом. В этом случае автомат P сдвигает считывающую головку влево, копируя входную строку в магазинный список, пока головка не достигнет левого концевого маркера. Если первоначальная входная строка имела вид $a_1a_2 ...a_n$, то в этот момент в магазинном списке автомата P содержится строка $a_i...a_2a_1Z_0$ для некоторого i. Если $i = 2$, то P останавливается и отвергает входную строку. В противном случае P удаляет a_i из магазинного списка и сдвигает считывающую головку на одну ячейку вправо — к символу, стоящему непосредственно справа от левого концевого маркера. Затем автомат P возвращается к шагу 3.

Таким образом, автомат P устанавливает, начинается ли входная строка с палиндрома, пытаясь последовательно сравнивать строку $a_i...a_1$ с префиксом строки $a_1...a_n$ для каждого $2 \le i \le n$. На эту процедуру автомат P может затратить $O(n^2)$ шагов. □

Покажем, что за время $O(|w|)$ можно выяснить, допускает ли двусторонний детерминированный магазинный автомат P входную строку w, независимо от того, сколько шагов в действительности делает P. На протяжении последующего обсуждения мы считаем фиксированными двусторонний детерминированный магазинный автомат $P = (S, I, T, \delta, s_0, Z_0, s_f)$ и входную строку w длины n.

Определение. *Поверхностной конфигурацией* (кратко — *конфигурацией*) автомата P называют такую тройку (s, i, A), что $s \in S$, i — положение считывающей головки, $0 \le i \le n + 1$, и A — один из символов магазинного списка из алфавита $T \cup \{Z_0\}$.

Заметим, что конфигурация является мгновенным описанием, но не всякое мгновенное описание является конфигурацией. В некотором смысле каждая поверхностная конфигурация представляет многие мгновенные описания — те, у которых "поверхность", т.е. вершина магазинного списка, совпадает с третьим компонентом этой конфигурации. Говорят, что конфигурация $C' = (s', i', A)$ *выводится* из конфигурации $C = (s, i, A)$, и пишут $C \Rightarrow C'$, если найдется такая последовательность шагов автомата P, что при $m \ge 0$

$$\begin{aligned} C &\vdash (s_1, i_1, \alpha_1) \\ &\vdash (s_2, i_2, \alpha_2) \\ &\vdots \\ &\vdash (s_m, i_m, \alpha_m) \\ &\vdash C', \end{aligned}$$

где $|\alpha_i| \ge 2$ для каждого i.[6] В этой последовательности шагов промежуточные мгновенные описания (s_j, i_j, α_j) имеют в магазинном списке не менее двух символов. Мы будем работать с конфигурациями, а не с мгновенными описаниями, потому что конфигураций только $O(n)$, а количество различных мгновенных описаний может быть экспоненциальным.

Определение. Конфигурацию (s, i, A) называют *терминальной*, если значение $\delta(s, a_i, A)$ не определено или имеет вид $(s', d, \mathbf{pop})$. Иначе говоря, терминальная конфигурация приводит или к остановке автомата P, или к удалению символа из магазинного списка. *Терминатором* конфигурации C называется такая единственная терминальная конфигурация C' (если она существует), что $C \overset{*}{\Rightarrow} C'$, где $\overset{*}{\Rightarrow}$ — это рефлексивное и транзитивное замыкание отношения $\Rightarrow$.

Пример 9.13. Если изобразить длину магазинного списка как функцию числа шагов, сделанных автоматом P, то можно получить кривую, представленную на рис. 9.17. На этом рисунке каждая точка помечена конфигурацией (не мгновенным описанием) автомата P после каждого шага.

[6] Если $m = 0$, то $C \vdash C'$.

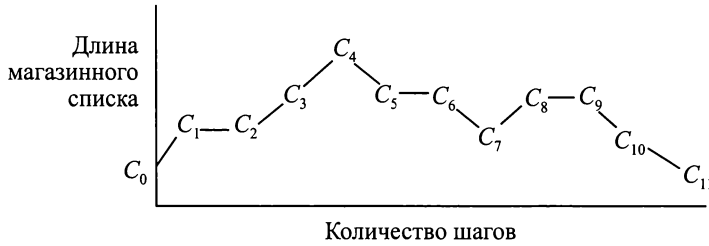


Рис. 9.17. Последовательность конфигураций

Из рис. 9.17 видно, например, что в конфигурациях C_0 и C_{11} в вершине стека находится Z_0. Поскольку между этими конфигурациями нет конфигурации с Z_0 в вершине стека, то $C_0 \Rightarrow C_{11}$. Если C_{11} — заключительная конфигурация, то C_{11} служит терминатором как для себя, так и для C_0. Из этого рисунка можно также сделать вывод, что $C_1 \Rightarrow C_2$, $C_2 \Rightarrow C_7$, $C_1 \overset{*}{\Rightarrow} C_{10}$ и C_{10} — терминальная конфигурация, поскольку она приводит к тому, что P удаляет символ из магазина. □

Ключевые соображения, необходимые для алгоритма моделирования, сформулированы в следующих двух простых леммах.

Лемма 9.1. Автомат P допускает w тогда и только тогда, когда некоторая конфигурация вида (s_f, i, Z_0), $0 \le i \le |w| + 1$, служит терминатором начальной конфигурации $(s_0, 1, Z_0)$.

Доказательство. Результат следует непосредственно из определения процесса допуска входной строки двусторонним детерминированным магазинным автоматом. □

Определение. Конфигурацию C называют *предшественницей* конфигурации D, если $C \overset{*}{\Rightarrow} D$, и *непосредственной предшественницей* конфигурации D, если $C \Rightarrow D$. Говорят, что пара конфигураций (C, D) лежит ниже пары (C', D') (необязательно различных) конфигураций, если

1) $C = (s, i, A), D = (t, j, A)$;

2) $C' = (s', i', B), D' = (t', j', B)$;

3) $(s, i, A) \vdash (s', i', BA)$;

4) $(t', j', BA) \vdash (t, j, A)$;

т.е. если автомат P может перейти из C в C' с помощью шага **push**, а из D' в D — с помощью шага **pop**.

Лемма 9.2. Если (C, D) лежит ниже (C', D') и $C' \stackrel{*}{\Rightarrow} D'$, то $C \Rightarrow D$.

Доказательство. Представляет собой несложное упражнение. □

Пример 9.14. На рис. 9.17 пара (C_3, C_5) лежит ниже пары (C_4, C_4), а (C_7, C_{10}) — ниже (C_8, C_9). Однако нельзя сказать с уверенностью, что (C_2, C_{10}) лежит ниже (C_3, C_9), поскольку символы в вершинах магазинов для конфигураций C_3 и C_9 могут быть разными. □

Работа моделирующего алгоритма основана на поиске терминатора для каждой конфигурации двустороннего детерминированного магазинного автомата P при входе w. Цель считается достигнутой, если терминатор начальной конфигурации $(s_0, 1, Z_0)$ найден.

Для хранения терминатора каждой конфигурации используется массив TERM. Мы предполагаем, что множество конфигураций линейно упорядочено (лексикографически). Тогда можно обращаться с именем конфигурации C как с целым числом и считать TERM[C] терминатором для C.

Используется также массив списков PRED. Он индексируется конфигурациями, и PRED[D] — список конфигураций C, для которых выполняется условие $C \Rightarrow D$.

Кроме массивов TERM и PRED используются два дополнительных списка: NEW и TEMP. Список NEW содержит пары еще не рассмотренных конфигураций (C, D), для которых TERM[C] = D. Список TEMP нужен в процедуре UPDATE(C, D), чтобы хранить предшественниц конфигурации C.

Мы поступаем следующим образом. Сначала присвоим TERM[C] = C, если C — терминальная конфигурация. (Каждая терминальная конфигурация является своим терминатором.) Добавим (C, C) в список NEW. Затем рассмотрим такие пары (C, D), что $C \vdash D$ за один шаг автомата P. (Заметим, что при таком шаге магазинный список не изменяется.)

Если терминатор для D уже известен, полагаем TERM[E] = TERM[D] для всех E, о которых в этот момент известно, что они — предшественницы конфигурации C, включая саму C. (Собственные предшественницы находятся в списке PRED[C].) Добавляем также пару (E, TERM[D]) в список NEW.

Если терминатор для D еще не известен, то C помещается в список PRED[D], т.е. в список непосредственных предшественниц конфигурации D.

В этот момент для каждой конфигурации C известна такая единственная конфигурация D, что $C \stackrel{*}{\Rightarrow} D$ без изменения магазинного списка, и либо D — терминатор для C, либо $D \vdash (s, i, \alpha)$ при $|\alpha| = 2$. Теперь рассматриваем все пары конфигураций, уже добавленные к списку NEW. В общем случае NEW содержит нерассмотренные

пары конфигураций (A, B), в которых $A \overset{*}{\Rightarrow} B$ и конфигурация B является терминальной. Допустим, что список NEW содержит пару (A, B). Удалим (A, B) из списка NEW и рассмотрим все пары (C, D) конфигураций, лежащих ниже (A, B). Если терминатор для конфигурации D уже вычислен, полагаем TERM[C] = TERM[D] и добавляем к списку NEW пару (C, TERM[D]). Для каждой конфигурации E из списка PRED[C] полагаем TERM[E] = TERM[D] и добавляем (E, TEPM[D]) в список NEW. Однако если терминатор для D еще не вычислен, помещаем C в список PRED[D]. Продолжаем эту процедуру, пока список NEW не опустеет. В этот момент будет найден терминатор (если он существует) для каждой конфигурации C.

Исчерпав весь список NEW, рассматриваем TERM[C_0], где C_0 — начальная конфигурация. Если TERM[C_0] — допускающая конфигурация, то двусторонний детерминированный магазинный автомат P допускает w. В противном случае P отвергает w.

Дадим более точное описание.

Алгоритм 9.4. Моделирование двустороннего детерминированного магазинного автомата

Вход. Двусторонний детерминированный магазинный автомат $P = (S, I, T, \delta, s_0, Z_0, s_f)$ и входная строка $w \in I^*$, $|w| = n$.

Выход. Ответ "да", если $w \in L(P)$, и "нет" в противном случае.

Метод.

1. Произведем начальную загрузку массивов и списков следующим образом. Для каждой конфигурации C положим TERM[C] = $\varnothing$ и PRED[C] = $\varnothing$. Положим NEW = $\varnothing$ и TEMP = $\varnothing$.
2. Для каждой терминальной конфигурации C полагаем TERM[C] = C и добавляем в список NEW пару (C, C).
3. Для каждой конфигурации C проверяем, существует ли такая конфигурация D, что $C \vdash D$ за один шаг. Если да, вызываем UPDATE(C, D). Процедура UPDATE приведена на рис. 9.18.
4. Пока список NEW не пуст, удаляем пару (C', D') из списка NEW. Для каждой пары (C, D), лежащей ниже (C', D'), вызываем UPDATE(C, D).
5. Если TERM[C_0], где C_0 — начальная конфигурация, является заключительной конфигурацией, получаем ответ "да". В противном случае "нет". □

```
      procedure UPDATE(C, D):
      begin
         comment Всякий раз, когда вызывается UPDATE(C, D),
             имеем C ⇒ D;
1.       if TERM[D] = ∅ then добавить C в список PRED[D]
         else
             begin
2.               TEMP ← {C};
3.               while TEMP ≠ ∅ do
                     begin
4.                       выбрать и удалить конфигурацию B из TEMP;
5.                       TERM[B] ← TERM[D];
6.                       добавить (B, TERM[D]) в список NEW;
7.                       for A ∈ PRED[B] do добавить A в TEMP
                     end
             end
      end
```

Рис. 9.18. Процедура UPDATE

Пример 9.15. Рассмотрим некоторые вычисления, выполняемые алгоритмом 9.4, когда он применяется к двустороннему детерминированному магазинному автомату, работающему, как показано на рис. 9.17.

На шаге 2 обнаруживаем, что C_4, C_6, C_9, C_{10} и C_{11} — терминальные конфигурации и потому свои же терминаторы. Добавляем в список NEW пары (C_4, C_4), (C_6, C_6), (C_9, C_9), (C_{10}, C_{10}) и (C_{11}, C_{11}).

На шаге 3 вызываем процедуру UPDATE(C_1, C_2). Поскольку в этот момент TERM[C_2] = ∅, то UPDATE всего лишь помещает C_1 в список PRED[C_2]. На шаге 3 также вызываем процедуру UPDATE(C_5, C_6). Поскольку TERM[C_6] = C_6, то UPDATE полагает TERM[C_5] = C_6 и добавляет (C_5, C_6) в список NEW. Кроме того, на шаге 3 вызываем процедуру UPDATE(C_8, C_9), и этот вызов полагает TERM[C_8] = C_9, и добавляет (C_8, C_9) в список NEW. Поэтому после шага 3 список NEW содержит пары

$$(C_4, C_4), (C_6, C_6), (C_9, C_9), (C_{10}, C_{10}), (C_{11}, C_{11}), (C_5, C_6), (C_8, C_8).$$

На шаге 4 удаляем (C_4, C_4) из списка NEW и вызываем UPDATE(C_3, C_5), так как (C_3, C_5) лежит ниже (C_4, C_4). Поскольку в этот момент TERM[C_5] = C_6, процедура UPDATE полагает TERM[C_3] = C_6 и добавляет (C_3, C_6) в список NEW. Затем на шаге 4 удаляем (C_6, C_6) из списка NEW и, поскольку (предположим, что это так) ниже (C_6, C_6) не лежит никакая пара, не вызываем UPDATE.[7] Аналогично не вызываем UPDATE для пар (C_9, C_9), (C_{10}, C_{10}), (C_{11}, C_{11}) и (C_5, C_6).

[7] Для простоты будем считать, что P не делает шагов, которые не указаны на рис. 9.17.

Когда пара (C_8, C_9) удаляется из списка NEW, вызываем процедуру UDATE(C_7, C_{10}), и этот вызов полагает TERM[C_7] = C_{10}, и добавляет пару (C_7, C_{10}) в список NEW. В этот момент список NEW содержит пары (C_3, C_6) и (C_7, C_{10}).

Удалив (C_3, C_6) из списка NEW, вызываем процедуру UPDATE(C_2, C_7), что приводит к TERM[C_2] = C_{10} и TERM[C_1] = C_{10}, поскольку PRED[C_2] содержит C_1. Добавляем (C_2, C_{10}) и (C_1, C_{10}) в список NEW.

Предлагаем читателям завершить это моделирование самостоятельно. □

Теорема 9.10. Алгоритм 9.4 правильно отвечает на вопрос "Принадлежит ли слово w языку $L(P)$?" за время $O(|w|)$.

Доказательство. Можно показать, что каждая конфигурация попадает в список TEMP не более одного раза. Следовательно, каждый вызов процедуры UPDATE всегда завершается. Также можно показать, что никакая пара конфигураций не попадает в список NEW более одного раза, следовательно, и сам алгоритм всегда завершает работу. Подробное доказательство этих двух утверждений оставляем читателям в качестве упражнения.

Покажем, что по окончании работы алгоритма 9.4 TERM[C_0] будет заключительной конфигурацией тогда и только тогда, когда $w \in L(P)$. Легко показать индукцией по числу шагов работы алгоритма 9.4, что

1) если TERM[C] полагается равным D, то D — терминатор для C;

2) если C добавляется в список PRED[D], то $C \Rightarrow D$;

3) если (C, D) помещается в список NEW, то TERM[C] = D.

Таким образом, если алгоритм 9.4 обнаруживает, что TERM[C_0] — заключительная конфигурация, то в силу леммы 9.1 утверждение $w \in L(P)$ является истинным.

Кроме того, необходимо показать, что если D — терминатор для C, то TERM[C] в итоге становится равным D. Доказательство проводится индукцией по числу шагов в последовательности $C \vdash^* D$. База индукции, соответствующая нулевому количеству шагов, тривиальна, поскольку $C = D$ и TERM[C] полагается равным D на шаге 2 алгоритма 9.4.

Для шага индукции предположим, что $C \vdash E \vdash^* D$, и рассмотрим отдельно два случая.

Вариант 1. E — конфигурация, т.е. шаг $C \vdash E$ не является ни шагом **push**, ни шагом **pop**. Тогда на шаге 3 вызывается процедура UPDATE(C, E). Если к этому моменту TERM[E] было присвоено значение D, то конфигурация C будет помещена в список TEMP в строке 2 и в итоге значение TERM[C] станет равным D в строке 5. Если значение TERM[E] еще не равно D, то добавляем C

к PRED[E] в строке 1 процедуры UPDATE(C, E). По предположению индукции мы в итоге получим, что TERM[E] = D. Если это случится в строке 5 процедуры UPDATE, то конфигурация C будет добавлена в список TEMP в строке 7, а значение TERM[C] станет равным D во время этого вызова процедуры UPDATE. Значение TERM[E] не может стать равным D на шаге 2 алгоритма 9.4, поскольку $E \neq D$. (Если бы оказалось, что $E = D$, то TERM[E] уже было бы присвоено значение D к тому моменту, когда мы стали рассматривать $C \vdash E$ на шаге 3.) Итак, в варианте 1 значение TERM[C] оказывается равным D.

Вариант 2. E — это такое мгновенное описание, что $C \vdash E$ является шагом **push**. Тогда можно найти такие конфигурации A, B и F, что (C, F) лежит ниже (A, B), $A \overset{*}{\vdash} B$ и $F \overset{*}{\vdash} D$, причем каждый из этих переходов совершается за меньшее число шагов, чем переход $C \overset{*}{\vdash}$ D. (Конфигурация A служит "поверхностью" мгновенного описания E.) По предположению индукции, значение TERM[A] полагается равным B, а значение TERM[F] — равным D.

Допустим, что последнее происходит раньше первого. Тогда (A, B) со временем оказывается в списке NEW, а на шаге 4 вызывается процедура UPDATE(C, F). Поскольку в этот момент TERM[F] = D, в строке 5 полагаем TERM[C] = D.

Допустим теперь, что TERM[A] присвоено значение B до того, как TERM[F] присвоено значение D. Тогда при вызове UPDATE(C, F) выполняется условие TERM[F] = $\varnothing$ и конфигурация C добавляется в список PRED[F]. Но тогда D становится значением TERM[C] при вычислении TERM[F]. Это завершает индукцию и доказательство корректности алгоритма 9.4.

Проанализируем время работы алгоритма 9.4. Шаги 1 и 2 занимают $O(n)$ времени, так как существует $O(n)$ конфигураций. Поскольку для каждой конфигурации двусторонний детерминированный магазинный автомат делает не более одного шага, то существует не более $O(n)$ пар конфигураций (C, D), в которых $C \vdash D$. Следовательно, процедура UPDATE вызывается на шаге 3 не более $O(n)$ раз.

Пара (C', D') попадает в список NEW, когда $C' \overset{*}{\Rightarrow} D'$ и уже известно, что D' — терминатор для C'. Поскольку каждая конфигурация имеет лишь один терминатор (если он вообще существует), то никакая пара не включается в список NEW более одного раза. Следовательно, общее число пар, записываемых в список NEW, не превосходит $O(n)$. Ниже каждой пары, попавшей в список NEW, лежит ограниченное число пар, поскольку если (C, D) лежит ниже (C', D'), то положения головок конфигураций C и C', а также D и D' различаются не более чем на единицу. Следовательно, процедура UPDATE вызывается $O(n)$ раз.

Теперь оценим общее время, затрачиваемое на процедуру UPDATE. Можно показать, что в массиве PRED каждая конфигурация появляется не более одного раза и в список TEMP никакая конфигурация не включается более одного раза. Общее время выполнения строк 4–6 процедуры UPDATE пропорционально количеству

конфигураций B, удаляемых из списка TEMP, а время выполнения строки 7 — количеству конфигураций A, добавляемых в список TEMP. Поскольку процедура UPDATE вызывается не более $O(n)$ раз, то общее время, занимаемое процедурой UPDATE, без учета времени на строки 4–6 и 7 составляет $O(n)$. Следовательно, алгоритм 9.4 работает линейное время. □

Результаты настоящего раздела применяются главным образом для доказательства существования алгоритмов линейной сложности, решающих определенные задачи. Мы уже видели, что некоторые задачи распознавания образцов можно сформулировать как задачи распознавания языков. Если мы сможем построить двусторонний детерминированный магазинный автомат, распознающий язык, соответствующий задаче распознавания образцов, то сможем утверждать, что существует алгоритм линейной сложности, решающий ту же задачу. Поскольку на входе длины n двусторонний детерминированный магазинный автомат может делать n^2 или даже 2^n шагов, то часто бывает проще найти алгоритм для распознавания языка с помощью двустороннего детерминированного магазинного автомата, чем алгоритм линейной сложности для решения задачи идентификации образцов прямо на машине RAM.

9.5. Позиционные деревья и идентификаторы подстрок

В предыдущем разделе мы показали, что если задачу распознавания образцов можно сформулировать как задачу распознавания языка, для решения которой можно найти соответствующий двусторонний детерминированный магазинный автомат, то исходную задачу распознавания образцов можно решить за линейное время. Можно было бы взять непосредственно алгоритм 9.4 как алгоритм линейной сложности, решающий исходную задачу. Однако мультипликативная константа, возникающая при таком моделировании, сделала бы этот подход в лучшем случае непривлекательным. В настоящем разделе мы изучим структуру данных, которую можно использовать для распознавания образцов с помощью более практичных алгоритмов, работающих линейное время. Эта структура данных применима, в частности, к следующим задачам.

1. По данным текстовой строки x и строки-образца найти все вхождения строки y в строку x.
2. Найти самую длинную повторяющуюся подстроку в заданной текстовой строке x.
3. Найти самую длинную подстроку, входящую как в заданные строки x и y.

Определение. *Позицией* в строке длины n считается любое число от 1 до n. Говорят, что символ a *входит в строку* x *в позиции* i, если $x = yaz$ и $|y| = i - 1$. Говорят, что подстрока u *идентифицирует* позицию i в строке x, если $x = yuz$, где $|y| = i - 1$, и строку x можно представить в виде $y'uz'$ только при условии $y' = y$, т.е. в позиции i начинается единственное вхождение строки u в x. Например, подстрока bba идентифицирует позицию 2 в строке $abbabb$, а подстрока bb не идентифицирует позицию 2.

В оставшейся части этой главы положим $x = a_1a_2...a_n$ — это строка в некотором алфавите I и $a_{n+1} = \$$ — символ, не принадлежащий алфавиту I. Тогда каждая позиция i строки $x\$ = a_1a_2...a_{n+1}$ идентифицируется по крайней мере одной строкой, а именно $a_ia_{i+1}...a_{n+1}$. Кратчайшая строка, идентифицирующая позицию i в строке $x\$$, называется *идентификатором позиции* i в *строке* x и обозначается через $s(i)$.

Пример 9.16. Рассмотрим строку $abbabb\$$. Идентификаторы позиций от 1 до 7 приведены на рис. 9.19. □

Позиция	Идентификатор подстроки
1	$abba$
2	bba
3	ba
4	$abb\$$
5	$bb\$$
6	$b\$$
7	$\$$

Рис. 9.19. Идентификаторы для $abbabb\$$

Идентификаторы позиций в строке $x\$$ удобно представлять с помощью дерева, называемого I-деревом, в котором помечены ребра и некоторые вершины.

Определение. *I-деревом* называют помеченное дерево T, ребра которого, выходящие из любой внутренней вершины, помечены различными символами из алфавита I. Если ребро $(v, w) \in T$ помечено символом a, то w называют a-сыном вершины v.

Позиционным деревом для строки $x\$ = a_1...a_na_{n+1}$, где $a_i \in I$, $1 \le i \le n$, и $a_{n+1} = \$$, называют $(I \cup \{\$\})$-дерево, для которого выполнены следующие условия.

1. T имеет $n + 1$ листьев, помеченных числами 1, 2, ..., $n + 1$. Листья дерева T взаимно однозначно соответствуют позициям в строке $x\$$.
2. Последовательность реберных меток, стоящих на пути из корня в лист с меткой i, является идентификатором $s(i)$ позиции i.

Пример 9.17. На рис. 9.20 изображено позиционное дерево для строки *abbabb*\$. Например, путь из корня в лист с меткой 2 помечен строкой *bba*, являющейся идентификатором позиции 2. □

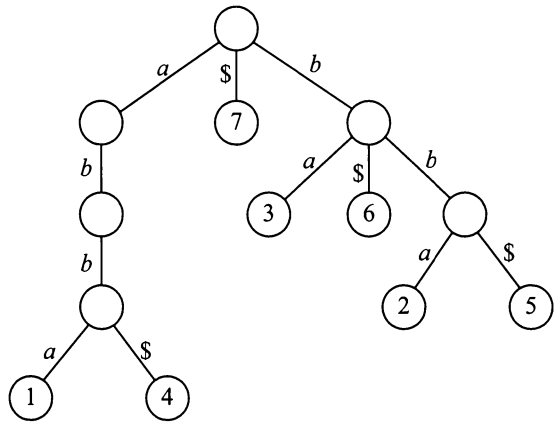


Рис. 9.20. Позиционное дерево

Сформулируем некоторые основные свойства идентификаторов.

Лемма 9.3. Пусть $s(i)$ — идентификатор позиции i строки $x\$ = a_1a_2...a_{n+1}$.

1. Если $s(i)$ имеет длину j, то длина подстроки $s(i-1)$ не превосходит $j+1$.
2. Никакой идентификатор не является собственным префиксом другого идентификатора.

Доказательство.

1. Если длина подстроки $s(i-1)$ больше $j+1$, то найдется такая позиция $k \neq i-1$, что $a_{i-1}a_i...a_{i+j-1} = a_ka_{k+1}...a_{k+j}$. Следовательно, $a_ia_{i+1}..a_{i+j-1} = a_{k+1}a_{k+1}...a_{k+j}$ и $a_ia_{i+1}..a_{i+j-1}$ не идентифицирует позицию i. Полученное противоречие доказывает исходное утверждение.
2. Легкое упражнение. □

Лемма 9.3.2, гарантирует, что для каждой строки $x\$$ действительно существует позиционное дерево.

С помощью позиционных деревьев можно решить многие задачи распознавания образцов, включая и упомянутые в начале раздела.

Пример 9.18. Исследуем основную задачу распознавания образцов: "Является ли строка $y = b_1b_2...b_p$ подстрокой строки $x = a_1a_2...a_n$?" Допустим, мы уже построили для строки $x\$$ позиционное дерево T. Чтобы узнать, входит ли строка $y = b_1b_2...b_p$ в строку x, рассмотрим это дерево как граф переходов некоторого детерминированного конечного автомата. Иначе говоря, отправляясь из корня дерева T, проследим максимально длинный возможный путь в позиционном дереве, которому приписана строка $b_1b_2...b_j$ для некоторого $0 \le j \le p$. Пусть этот путь оканчивается в вершине v. Существует несколько вариантов.

1. Если $j < p$ и v — не лист, то ответ — "нет". В этом случае $b_1b_2...b_j$ — подстрока строки x, а $b_1b_2...b_{j+1}$ — нет.
2. Если $j \le p$ и v — лист с меткой i, то $b_1b_2...b_j$ совпадает с j символами строки x, начиная с позиции i. Тогда необходимо сравнить $b_{j+1}b_{j+2}...b_p$ с $a_{i+j}a_{i+j+1}...a_{i+p-1}$. Если они не совпадают, то ответом будет "нет"; в противном случае "да", причем строка y входит в строку x, начиная с позиции i.
3. Если $j = p$ и v — не лист, то ответом будет "да". В этом случае y — подстрока с двумя или более вхождениями в строку x. Начальные позиции этих вхождений — метки листьев, принадлежащих поддереву вершины v позиционного дерева. □

Пример 9.19. С помощью позиционного дерева можно найти самую длинную повторяющуюся подстроку данной строки $x = a_1a_2...a_n$ (два вхождения этой подстроки могут перекрываться). Длиннейшая повторяющаяся подстрока соответствует внутренней вершине позиционного дерева с наибольшей глубиной. Такую вершину можно обнаружить очевидным способом.

Рассмотрим нахождение длиннейшей повторяющейся подстроки в строке *abbabb*. В позиционном дереве для строки *abbabb*\$ (рис. 9.20) есть одна внутренняя вершина глубины 3 и нет ни одной внутренней вершины большей глубины. Следовательно, соответствующая этой вершине строка *abb* — самая длинная повторяющаяся подстрока в строке *abbabb*. Вершины с метками 1 и 4 указывают, где начинаются два вхождения строки *abb*. В данном случае они не перекрываются. □

Рассмотрим подробно задачу построения позиционного дерева. В оставшейся части этой главы $x\$$ будет строкой $a_1...a_na_{n+1}$, где a_{n+1} — единственный правый концевой маркер \$. Через x_i, $1 < i < n + 1$, будем обозначать суффикс строки $a_1...a_na_{n+1}$, а через $s_i(j)$ — идентификатор позиции j в x_i. Все позиции нумеруются по исходной строке $a_1...a_na_{n+1}$.

Построим алгоритм построения позиционного дерева для строки $a_1...a_n a_{n+1}$ за время, пропорциональное количеству вершин окончательного дерева. Заметим, что позиционное дерево для строки $a_1...a_n a_{n+1}$ может содержать $O(n^2)$ вершин. Например, позиционное дерево для сстроки $a^n b^n a^n b^n\$$[8] содержит $n^2 + 6n + 2$ вершин, в чем легко убедиться. Однако при разумных предположениях о том, что такое "случайная" строка (например, символы выбираются из алфавита с равной вероятностью и независимо друг от друга), можно показать, что среднее позиционное дерево для строки длины n содержит $O(n)$ вершин. Мы не будем показывать это здесь. Отметим лишь, что существует алгоритм сложности $O(n)$ в худшем случае, который строит компактную форму позиционного дерева прямо по данной строке. Работы, обсуждающие этот алгоритм, указаны в библиографических замечаниях.

Рассмотрим различия между множествами S_i и S_{i+1} идентификаторов для строк x_i и x_{i+1} поскольку в алгоритме, описываемом ниже, множество S_i строится по S_{i+1}. Одно очевидное различие состоит в том, что множество S_i содержит идентификатор $s_i(i)$ первой позиции в строке x_i. Из-за того, что множество S_i содержит эту дополнительную строку, может оказаться, что идентификатор некоторой позиции k, содержащийся в S_{i+1}, не является идентификатором позиции k в S_i. Это происходит тогда и только тогда, когда идентификатор $s_{i+1}(k)$ позиции k в S_{i+1} является префиксом строки $s_i(i)$. В таком случае необходимо удлинить строку $s_{i+1}(k)$, чтобы сделать из нее строку $s_i(k)$, т.е. идентификатор позиции k в множестве S_i. Ситуация, в которой необходимо удлинить два идентификатора из множества S_{i+1}, невозможна. В самом деле, если бы необходимо было удлинять строки $s_{i+1}(k_1)$ и $s_{i+1}(k_2)$, то они обе были бы префиксами строки $s_i(i)$ и, значит, одна из них была бы префиксом другой вопреки лемме 9.3.2.

Пример 9.20. Пусть $a_1...a_n a_{n+1} = abbabb\$$. Идентификаторы для $x_4 = abb\$$, $x_3 = babb\$$, $x_2 = bbabb\$$ и $x_1 = abbabb\$$ приведены на рис. 9.21. Заметим, что множество S_3 получается из S_4 добавлением одной строки $s_3(3) = ba$. С другой стороны, для построения множества S_2 из S_3 потребовалось два изменения. Мы добавили $s_2(2) = bba$ во множество S_2 и дописали символ \$ к концу строки $s_3(5)$, чтобы получить $s_2(5) = bb\$$. Чтобы построить множество S_1, мы добавили строку $s_1(1) = abba$ во множество S_2 и дописали строку $bb\$$ к строке $s_2(4)$, чтобы получить $s_1(4) = abb\$$. □

---

[8] Выражение a^n означает n-кратную конкатенацию символа a.

Позиция	$s_4(p)$	$s_3(p)$	$s_2(p)$	$s_1(p)$
7	\$	\$	\$	\$
6	*b*\$	*b*\$	*b*\$	*b*\$
5	*bb*	*bb*	*bb*\$	*bb*\$
4	*a*	*a*	*a*	*abb*\$
3	–	*ba*	*ba*	*ba*
2	–	–	*bba*	*bba*
1	–	–	–	*abba*

Рис. 9.21. Идентификаторы некоторых суффиксов строки *abbabb*\$

Рассмотрим, что же необходимо для построения позиционного дерева T_i, представляющего множество S_i, по позиционному дереву T_{i+1}, представляющему множество S_{i+1}. Допустим, дерево T_{i+1} уже построено. Необходимо добавить к дереву T_{i+1} лист с меткой i, соответствующий строке $s_i(i)$. Если для некоторого k, $i < k \le n$, строка $s_{i+1}(k)$ служит префиксом строки $s_i(i)$, то необходимо также удлинить путь в дереве T_{i+1}, идущий из корня в лист с меткой k, чтобы новый лист с меткой k в дереве T_i соответствовал строке $s_i(k)$. Таким образом, ключом к эффективному построению позиционного дерева T_i для x_i по дереву T_{i+1} является умение быстро находить такую строку y, что строка $a_i y$ является самым длинным префиксом строки x_i, входящий также в x_{i+1}, кроме того, умение определять, служит ли строка $a_i y$ префиксом идентификатора из множества T_{i+1}.

Такое построение требует добавления к позиционному дереву трех новых структур. Первая из них — двоичный вектор (массив), который приписывается каждой вершине позиционного дерева T_i. Вектор, приписываемый вершине v, обозначим через B_v. Каждому символу из множества I в этом векторе будет соответствовать свой компонент. Если v — вершина в дереве T_i, соответствующая строке y, и $a \in I$, то $B_v[a] = 1$, если ay — подстрока строки x_i. В противном случае $B_v[a] = 0$.

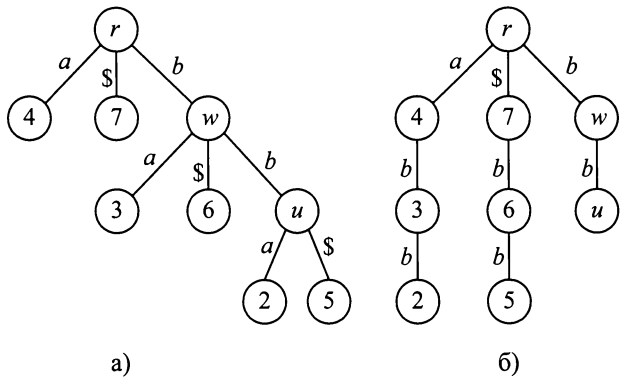
Далее каждой вершине приписывается ее глубина в позиционном дереве. Эту информацию легко корректировать по мере роста дерева, и впредь мы будем по умолчанию предполагать, что она вычисляется.

Последнее добавление к позиционному дереву — новая древовидная структура, расположенная на его вершинах. Мы будем называть ее "вспомогательным деревом", но на самом деле это просто другое множество ребер, соединяющих вершины нашего позиционного дерева.

Определение. Пусть T_i — позиционное дерево для строки $x_i = a_i a_{i+1} \ldots a_n\$$. *Вспомогательным деревом* для позиционного дерева T_i назовем $(I \cup \{\$\})$-дерево A_i, обладающее следующими свойствами.

1. Дерево A_i имеет то же множество вершин, что и дерево T_i.
2. Вершина w является a-сыном вершины v в дереве A_i, если ay — строка, соответствующая вершине w в дереве T_i, и строка y соответствует вершине v в дереве T_i. Таким образом, пути в дереве A_i, идущему из корня в вершину w, приписана строка $y^R a$, а пути в T_i, идущему из корня в w, приписана строка ay.

Пример 9.21. Позиционное и вспомогательное деревья для строки $x_2 = bbabb\$$ изображены на рис. 9.22. (Числа на листьях указывают позиции относительно строки $x = abbabb\$$.) Заметим, что листья вспомогательного дерева не обязательно являются листьями позиционного дерева, хотя множества вершин у обоих деревьев совпадают. □



а) б)

Рис. 9.22. Позиционное (*а*) и вспомогательное (*б*) деревья для строки $bbabb\$$

Из определения не видно, всегда ли у позиционного дерева существует вспомогательное. Действительно, произвольное I-дерево может не иметь вспомогательного. Условия, при которых I-дерево имеет вспомогательное дерево, сформулированы в следующей теореме.

Теорема 9.11. Для того чтобы у данного I-дерева T существовало вспомогательное дерево, необходимо и достаточно, чтобы T обладало таким свойством: если в T есть вершина, соответствующая строке ax, где $a \in I$ и $x \in I^*$, то в нем есть также вершина, соответствующая строке x.

Доказательство. Оставляем читателям в качестве упражнения. □

Следствие. Любое позиционное дерево имеет вспомогательное дерево.

Доказательство. Если ax — префикс идентификатора позиции i, то в силу леммы 9.3.1, строка x является префиксом идентификатора позиции $i+1$. □

Теперь мы готовы описать алгоритм, который строит позиционное дерево T_i для x_i из позиционного дерева T_{i+1} для x_{i+1}.

Алгоритм 9.5. Построение T_i из T_{i+1}

Вход. Строка $a_1...a_n a_{n+1}$, позиционное дерево T_{i+1} и вспомогательное дерево A_{i+1} для строки $x_{i+1} = a_{i+1}...a_n a_{n+1}$.

Выход. Позиционное T_i и вспомогательное A_i деревья для x_i.

1. Находим в дереве T_{i+1} лист с меткой $i+1$. (Это последний лист, добавленный к дереву T_{i+1}.)
2. Проходим по пути, ведущему из этого листа в корень дерева T_{i+1} до такой вершины u_y, что $B_{u_y}[a_i] = 1$. (Вершина u_y соответствует такой самой длинной строке y, что $a_i y$ — префикс строки x_i и подстрока строки x_{i+1}, начинающаяся в некоторой позиции k, $i < k \le n$.) Если такой вершины нет, доходим до корня.
3. Полагаем $B_v[a_i] = 1$ для каждой вершины v на пути, идущем из вершины с меткой $i+1$ к сыну вершины u_y, расположенному на том же пути, или в корень, если такого u_y нет. (Каждая вершина v на этом пути соответствует некоторому префиксу z строки x_{i+1}, поэтому очевидно, что $a_i z$ — подстрока строки $a_i x_{i+1}$.)
4. **а)** Если вершины u_y нет, переходим к варианту 1.

   б) Если вершина u_y существует, но у нее нет a_i-сына во вспомогательном дереве A_{i+1}, переходим к варианту 2.

   в) Если у вершины u_y есть a_i-сын v_y во вспомогательном дереве, переходим к варианту 3. Вершина v_y соответствует строке $a_i y$ в позиционном дереве T_{i+1}.

Вариант 1. Символ a_i не входит в строку x_{i+1}. Тогда идентификатором позиции i является сам символ a_i. Чтобы построить дерево T_i из T_{i+1} выполняем следующие действия.

1. Создаем новый лист с меткой i, являющийся a_i-сыном корня дерева T_{i+1}.
2. Полагаем $B_i[a] = 0$ для всех $a \in I$.

Чтобы построить дерево A_i из дерева A_{i+1}, делаем новую вершину с меткой i a_i-сыном корня дерева A_{i+1}.

Вариант 2. Символ a_i входит в строку x_{i+1}, но в дереве T_{i+1} есть лишь собственный префикс строки $a_i y$ (приписанный пути из корня дерева T_{i+1} в некоторый лист v_j). Эта ситуация возникает тогда, когда нужно удлинять идентификатор некоторой позиции k, $i < k \le n$, чтобы он стал идентификатором позиции k в дереве T_i. Допустим, $y = a_{i+1}a_{i+2}...a_j$ и вершина v_1 соответствует строке $a_i a_{i+1}...a_p$ для некоторого $p < j$. Тогда k — метка листа v_1 и $a_i a_{i+1}...a_p$ — идентификатор позиции k в деревае T_{i+1} (т.е. $a_i a_{i+1}...a_p = a_k a_{k+1}...a_l$).

Чтобы построить T_i из T_{i+1}, добавим к вершине v_1 поддерево с двумя новыми листьями, помеченными числами i и k. Пути, идущему из v_1 в k в этом добавленном поддереве, приписана строка $a_{i+1}a_{i+2}...a_{m+1}$, а пути из v_1 в i — строка $a_{p+1}a_{p+2}...a_{j+1}$, где $a_{i+1}a_{i+2}...a_m = a_{p+1}a_{p+2}...a_j$. Таким образом, пути из корня дерева T_i в лист k приписана строка $a_k a_{k+1}...a_m a_{m+1}$, являющаяся новым идентификатором позиции k в x_i, а пути из корня в лист i — строка $a_i a_{i+1}...a_j a_{j+1}$, являющаяся идентификатором позиции i в x_i.

Точнее, для построения дерева T_i из T_{i+1} выполняем следующие действия.

1. Двигаясь по пути в T_{i+1}, идущему из u_y в корень, находим вершину u_1, являющуюся первым предком вершины u_y, имеющим a_i-сына в дереве A_{i+1}. Допустим, что этот a_i-сын — вершина v_1. В дереве T_{i+1} вершина v_1 — это лист с меткой k для некоторого $i < k \le n$. Стираем метку k у вершины v_1.
2. Пусть u_1, u_2, ..., u_q, u_y — последовательность вершин на пути в T_{i+1}, идущем из u_1 в u_y. Предположим, что ребро из u_h в u_{h+1} помечено символом c_h, $1 \le h < q$, а ребро из u_q в u_y — символом c_q, как показано на рис. 9.23 ($c_1c_2...c_q = a_{p+1}a_{p+2}...a_j = a_{i+1}a_{i+2}...a_m$).
3. В дереве T_i образуем q новых вершин v_2, v_3, ..., v_q, v_y. Делаем вершину v_{h+1} c_h-сыном вершины v_h для $1 \le h < q$. Делаем вершину v_y c_q-сыном вершины v_q.
4. Пусть j равна сумме i и глубины вершины u_y, а m — сумма k и глубины вершины u_y. Добавим к дереву T_i две новые вершины с метками i и k. Лист i сделаем a_{j+1}-сыном вершины v_y, а лист k — a_{m+1}-сыном вершины v_y
5. Для всех $a \in I$ полагаем $B_v[a] = B_{v_1}[a]$ для каждого $v \in \{v_2, v_3, ..., v_q, v_y, k\}$.
6. Полагаем $B_i[a] = 0$ для всех $a \in I$.

Чтобы построить дерево A_i из дерева A_{i+1}, выполняем следующие действия.

7. Делаем вершину v_y a_i-сыном вершины u_y в дереве A_i.
8. Допустим, u' — a_{j+1}-сын вершины u_y в дереве T_{i+1}. Новый лист с меткой i делаем a_i-сыном вершины u' в дереве A_i.

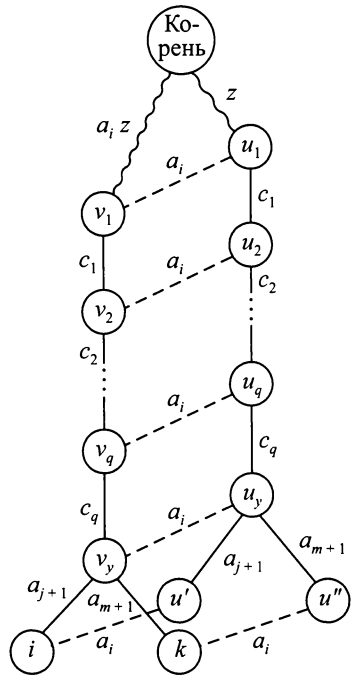


Рис. 9.23. Основные части дерева T_i в варианте 2.
Пунктирные линии обозначают ребра дерева A_i

9. Допустим, u'' — a_{m+1}-сын вершины u_y в дереве T_{i+1}. Новый лист с меткой k делаем a_i-сыном вершины u'' в дереве A_i.
10. Делаем вершину v_h a_i-сыном вершины u_h в дереве A_1 для $2 \le h < q$.

Вариант 3. Вершина u_y имеет a_i-сына v_y в дереве A_{i+1}. В этом случае необходимо рассмотреть две ситуации.

а) v_y — лист с меткой k в дереве T_{i+1}. Такая ситуация возникает, когда $s_i(i) = a_i a_{i+1} ... a_j a_{j+1}$ и $s_{i+1}(k) = a_k a_{k+1} ... a_m$, где $a_k a_{k+1} ... a_m = a_i a_{i+1} ... a_j$; это частный случай варианта 2, когда $v_1 = v_y$. Чтобы построить дерево T_i из дерева T_{i+1}, удалим метку k из вершины v_1 и создадим для вершины v_y a_{j+1}-сына с меткой i и a_{m+1}-сына с меткой k. Детали этого построения те же, что и в случае 2 (без шагов 3, 7 и 10).

б) v_y — внутренняя вершина дерева T_{i+1}. Такая ситуация возникает, когда $S_i = S_{i+1} \cup \{s_i(i)\}$. Чтобы построить дерево T_i из T_{i+1}, создадим для вершины v_y a_{j+1}-сына (которого у нее нет) и пометим этот лист меткой i. Подробный алгоритм имеет следующий вид.

1. Пусть метка j равна сумме метки i и глубины вершины u_y.
2. Новую вершину с меткой i делаем a_{j+1}-сыном вершины v_y в дереве T_i. (Заметим, что у вершины v_y не может быть a_{j+1}-сына в девере T_{i+1} в силу максимальности y.)
3. Полагаем $B_i[a] = 0$ для всех $a \in I$. Поскольку $a_i y a_{j+1}$ не является подстрокой строки x_{i+1}, то, разумеется, строка $aa_i y a_{j+1}$ не является подстрокой строки x_i ни для какого символа a.
4. Чтобы построить дерево A_i из дерева A_{i+1}, допустим, что вершина u' является a_{j+1}-сыном вершины u_y в дереве T_{i+1}. (В дереве T_{i+1} вершине u' соответствует вершина $y a_{j+1}$.) Сделаем новую вершину i a_i-сыном вершины u' в дереве A_{i+1}. (В дереве A_i вершина i будет соответствовать строке $a_{i+1} y^R a_i$.)

Связи между вершинами, упомянутыми в варианте 3, *б*, показаны на рис. 9.24.

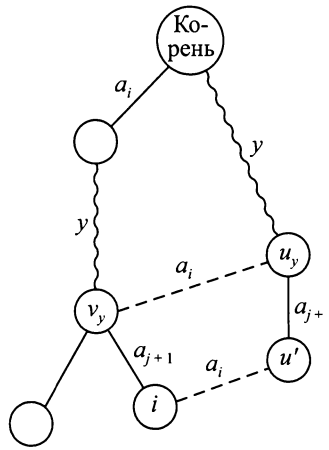


Рис. 9.24. Основные части дерева T_i в варианте 3, *б*.
Пунктирные линии с метками a_i обозначают ребра дерева A_i

Пример 9.22. Пусть $a_1 ... a_n a_{n+1} = abbabb\$$, а T_2 и A_2 — деревья с рис. 9.22. С помощью алгоритма 9.5 строим деревья T_1 и A_1 из деревьев T_2 и A_2. Здесь $i = 1$ и $a_1 = a$.

Сначала найдем лист с меткой 2 и, поскольку $B_2[a] = 0$, положим $B_2[a] = 1$, а затем поднимемся в вершину, обозначенную буквой u на рис. 9.22. В вершине u находим, что $B_u[a] = 1$, поскольку abb — подстрока строки $bbabb\$$. Таким образом, вершина u — это u_y. Теперь ищем вершину v_y, т.е. a-сына вершины u в дереве A_2. Такой вершины нет, так что здесь подходит вариант 2.

Вершина w, т.е. отец вершины u в дереве T_2, не имеет a-сына в дереве A_2. Однако вершина r, отец вершины w, имеет такого сына. Им является вершина 4. Поэтому на шаге 1 варианта 2 находим, что v_1 — это вершина с меткой 4 в деревьях T_2 и A_2. На шаге 2 находим, что $q = 2$, $u_1 = r$ и $u_2 = w$. Кроме того, $c_1 = c_2 = b$. Поэтому на шаге 3 создадим вершину v, являющуюся b-сыном вершины с меткой 4 в T_2 (в дереве T_1 она потеряла свою метку). Образуем также вершину v_y и сделаем ее b-сыном вершины v.

На шаге 4 получаем $j = 3$. Поскольку бывшая метка k вершины v_1 равна 4, заключаем, что $m = 6$. Находим $a_{j+1} = a$ и $a_{m+1} = \$$. Следовательно, новые вершины с метками 1 и 4 становятся соответственно a-сыном и \$-сыном вершины v_y. Дерево T_i изображено на рис. 9.25. Остальные шаги оставляем читателям в качестве упражнения. □

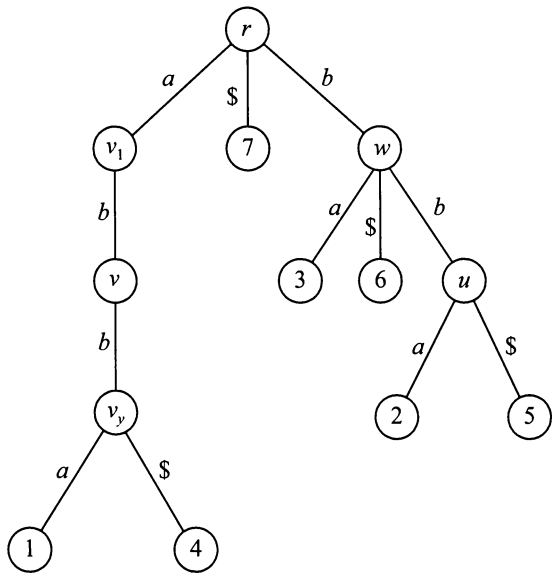


Рис. 9.25. Финальные позиции в дереве T_1

Лемма 9.4. Если T_{i+1} и A_{i+1} — позиционное и вспомогательное деревья для x_{i+1}, то деревья T_i и A_i, построенные с помощью алгоритма 9.5, — позиционное и вспомогательное деревья для x_i.

Доказательство. Допустим, T_{i+1} представляет множество S_{i+1} идентификаторов позиций в строке x_{i+1}, а A_{i+1} — вспомогательное дерево для T_{i+1}. Здесь могут быть две возможности:

1. $S_i = S_{i+1} \cup \{s_i(i)\}$.
2. $S_i = (S_{i+1} - \{s_{i+1}(k)\}) \cup \{s_i(k), s_i(i)\}$ для некоторого $i < k \le n$.

Первая возможность покрывается вариантами 1 и 3, *б*, алгоритма 9.5, вторая — вариантами 2 и 3, *а*. Рассуждения, необходимые для доказательства корректности алгоритма 9.5 в вариантах 1–3, включены в его описание. □

Таким образом, позиционное дерево для произвольной строки можно построить с помощью следующего алгоритма.

Алгоритм 9.6. Построение позиционного дерева

Вход. Строка $x\$ = a_1....a_n a_{n+1}$, где $a_i \in I$ для $1 \le i \le n$ и $a_{n+1} = \$$.

Выход. Позиционное дерево T_1 для $x\$$.

Метод.

1. Пусть T_{n+1} — позиционное дерево на рис. 9.26 и $B_r[a] = B_{n+1}[a] = 0$ для всех $a \in I$.
2. Пусть A_{n+1} — вспомогательное дерево, совпадающее с деревом, представленным на рис. 9.26.
3. **for** $i \leftarrow n$ **step** -1 **until** 1 **do** алгоритм 9.5 для построения деревьев T_i и A_i по деревьям T_{i+1} и A_{i+1}. □

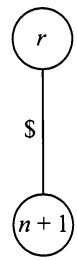


Рис. 9.26. Начальное позиционное дерево

Теорема 9.12. Алгоритм 9.6 строит позиционное дерево для строки $x\$$ за время, пропорциональное количеству вершин в окончательном позиционном дереве T_1.

Доказательство. На шаге 1 алгоритма 9.5 можно найти лист $i + 1$ за фиксированное время с помощью указателя на этот лист, установленного в момент добавления листа к дереву T_{i+1}. Когда к дереву T_i добавляется лист i, этот указатель переводится на лист i.

Объем работы, производимой при каждом выполнении шагов 2 и 3, очевидно, пропорционален длине пути из вершины $i + 1$ в u_y, а при каждом выполнении шага 1 он является постоянным. Легко проверить, что время, затрачиваемое на выполнение любого из вариантов 1–3 алгоритма 9.5, пропорционально количеству вершин, добавляемых к дереву. Таким образом, общее время, которое тратится всеми частями алгоритма 9.5, кроме шагов 2 и 3, пропорционально размеру дерева T_1.

Осталось показать, что сумма расстояний между вершинами $i + 1$ и u_y (или корнем, если вершины u_y не существует) в дереве T_{i+1} для $1 \le i \le n$ не превосходит размера дерева T_1. Обозначим эти расстояния через d_2, d_3, ..., d_{n+1}, и пусть e_i — глубина вершины i в дереве T_i, где $1 \le i \le n + 1$. Простой анализ вариантов 1–3 показывает, что

$$e_i \le e_{i+1} - d_{i+1} + 2. \tag{9.1}$$

Просуммируем обе части неравенства (9.1) по i от 1 до n:

$$\sum_{i=2}^{n+1} d_i \le 2n + e_{n+1} - e_1. \tag{9.2}$$

Из (9.2) непосредственно следует, что время, затрачиваемое шагами 2 и 3 алгоритма 9.5, составляет $O(n)$ и, значит, по порядку не превосходит размера дерева T_1. □

Как мы уже отмечали, позиционное дерево для строки длины n может содержать $O(n^2)$ вершин. Следовательно, любой алгоритм распознавания образцов, в котором строится такое дерево, должен иметь временную сложность $O(n^2)$. Однако позиционное и вспомогательное деревья можно уплотнить, сжав все цепи в позиционном дереве в одну вершину. *Цепью* называется путь, каждая вершина которого обладает в точности одним сыном. Нетрудно показать, что уплотненное позиционное дерево для строк длины n содержит не более $4n - 2$ вершин. Уплотненное дерево может давать ту же информацию, что и исходное позиционное дерево, и потому его можно использовать в тех же алгоритмах распознавания образцов.

Алгоритм 9.5 можно модифицировать так, чтобы он строил уплотненные позиционное и вспомогательное деревья за линейное время. Мы не приводим здесь этот модифицированный алгоритм потому, что он аналогичен алгоритму 9.5. Кроме того, во многих приложениях размер позиционного дерева пропорционален длине входной строки. Работы, в которых излагается алгоритм с уплотнением позиционного дерева и другие его приложения, указаны в библиографических замечаниях.

Упражнения

9.1. Постройте регулярные выражения и графы переходов конечных автоматов для следующих регулярных множеств строк в алфавите $I = \{a, b\}$.

а) Все строки, начинающиеся и кончающиеся символом a.

б) Все строки, не содержащие двух символов a подряд.

***в)** Все строки, содержащие нечетное число символов a и четное число символов b.

***г)** Все строки, не содержащие подстроки aba.

9.2. Докажите, что множество, допускаемое недетерминированным конечным автоматом, представленным на рис. 9.1, имеет вид $(a + b)^* aba$.

9.3. Постройте недетерминированный конечный автомат, допускающий следующие регулярные множества:

а) $(a + b)^* (aa + bb)$;

б) $a^*b^* + b^*a^*$;

в) $(a + \varepsilon)(b + \varepsilon)(c + \varepsilon)$.

9.4. Покажите, что дополнение регулярного множества регулярно.

***9.5.** Покажите, что следующие множества строк нерегулярны:

а) $\{a^n b^n \mid n \geq 1\}$;

б) $\{ww | w \in \{a, b\}^*\}$;

в) $\{w | w \in \{a, b\}^*$ и $w = w^R\}$ (т.е. множество палиндромов).

9.6. Пусть $x = a_1 a_2 ... a_n$ — заданная строка и α — регулярное выражение. Модифицируйте алгоритм 9.1 так, чтобы он находил наименьшее число k, а по нему (*а*) наименьшее j и (*б*) наибольшее j, такое, что $a_j a_{j+1} ... a_k$ принадлежит множеству, представленному выражением α. (Подсказка: каждому состоянию из S_i поставьте в соответствие целое число j.)

***9.7.** Пусть x и α — те же, что и в упражнении 9.6. Модифицируйте алгоритм 9.1 так, чтобы он находил такое наименьшее j и по нему наибольшее k, что $a_j a_{j+1} ... a_k$ принадлежит множеству, представленному регулярным выражением α.

9.8. Пусть x и α — те же, что и в упражнении 9.6. Постройте алгоритм, который находил бы все подстроки в строке x, принадлежащие множеству, представленному выражением α. Какова временная сложность вашего алгоритма?

***9.9.** Пусть I — алфавит и $\varnothing$ — символ, не принадлежащий I. *Строкой с несущественными символами* назовем строку в алфавите $I \cup \{\varnothing\}$. Будем гово-

рить, что строка с несущественными символами $b_1b_2...b_n$ совпадает со строкой с несущественными символами $a_1a_2...a_n$ в позиции i, если для всех j, $i \le j \le n$, либо $a_j = b_{j-i+1}$, либо один из символов a_j или b_{j-i+1} равен $\varnothing$. Покажите, что для фиксированного алфавита I проблема определения позиций, в которых одна строка с несущественными символами совпадает с другой, эквивалентна по асимптотической временной сложности *(и/или)-умножению*, т.е. операции вида $c_j = \bigvee_{i=1}^{n} d_i \wedge e_{j-i+1}$, где все c, d и e — булевы величины.

**9.10. Покажите, что можно определить позиции, в которых одна строка с несущественными символами совпадает с другой, за время $O_A\left(n \log^2 n \log\log n\right)$. (*Подсказка*: используйте упражнение 9.9 и алгоритм Шёнхаге–Штрассена для умножения целых чисел.)

9.11. Напишите алгоритм сложности $O(n)$, который по данным строкам $a_1a_2...a_n$ и $b_1b_2...b_n$ узнавал бы, существует ли такое k, что $a_i = b_{(k+i) \bmod n}$ для $1 \le i \le n$.

9.12. С помощью алгоритма 9.3 постройте машины для строк y, равных

а) *abbbbabbbabbaba*;

б) *abcabcab*.

9.13. Докажите теорему 9.8.

9.14. Пусть $S = \{y_1, y_2, ..., y_m\}$ — множество строк. Напишите алгоритм, который находил бы все вхождения строк из множества S в текстовую строку x. Какова временная сложность вашего алгоритма?

*9.15. Постройте двусторонний детерминированный магазинный автомат, допускающие языки

а) $\{xcy_1cy_2c...cy_m | m \ge 1, x \in \{a, b\}^*, y_i \in \{a, b\}^*$ для $1 \le i \le m$ и хотя бы одна из строк $y_1, y_2, ..., y_m$ входит в строку $x\}$;

б) $\{xyy^R | x, y \in I^*$ и $|y| \ge 1\}$;

в) $\{x_1cx_2c...cx_n | x_i \in \{a, b\}^*$ для $1 \le i \le n$ и все x_i разные$\}$;

г) $\{x_1cx_2c...cx_kdy_1cy_2c...cy_l$ | все x_i и y_i принадлежат множеству $\{a, b\}^*$ и $x_i = y_i^R$ для некоторых i и $j\}$.

9.16. Рассмотрим двусторонний детерминированный магазинный автомат P с правилами:

$\delta(s_0, a, Z_0) = \delta(s_0, a, A) = (s_0, +1, \textbf{push}\ A)$;
$\delta(s_0, \$, A) = (s_1, -1)$;
$\delta(s_1, a, A) = \delta(s_1, a, Z_0) = (s_1, -1, \textbf{pop})$;
$\delta(s_1, ¢, Z_0) = (s_f, 0)$.

а) Запишите все поверхностные конфигурации автомата P со входом *aa*.

б) Примените алгоритм 9.4 для вычисления массива TERM.

***9.17.** Предположим, что двусторонний детерминированный магазинный автомат может из позиции i (n обозначает входную длину) перейти за один шаг в любую из указанных ниже позиций:

а) $n - i$;

б) $i/2$;

в) $n/2$;

г) $\log n$.

Покажите, что ни одна из этих модификаций не увеличивает распознающей силы двустороннего детерминированного магазинного автомата. Какие операции вы можете добавить, чтобы не увеличить при этом допускающей силы?

****9.18.** Постройте двусторонний детерминированный магазинный автомат, допускающий язык

$$\left\{w_1 w_1^R w_2 w_2^R w_3 w_3^R u \mid w_1, w_2, w_3 \in \{a, b\}^+ \text{ и } u \in \{a, b\}^*\right\}.$$

(*Подсказка*: пусть x — входная строка. Напишите подпрограмму для нахождения такой кратчайшей строки w_1, что $w_1 w_1^R y = x$. Затем примените вашу подпрограмму к y и т.д.)

****9.19.** Постройте алгоритмы, распознающие за линейное время следующие языки:

а) $\{ww^R \mid w \in I^*\}^*$;

б) $\{ww^R u \mid w, u \in I^*, |w| \geq 1\}$;

в) $\{w_1 w_2 ... w_n \mid w_i \in \{a, b\}^*$ и каждая строка w_i является непустым палиндромом четной длины$\}$.

9.20. Напишите алгоритм, который по двум строкам, $a_1 a_2 ... a_n$ и $b_1 b_2 ... b_n$, отыскивает за линейное время такое наибольшее число l, что $a_1 a_2 ... a_l$ —подстрока строки $b_1 b_2 ... b_p$. Как с помощью этого алгоритма проверить, является ли данная строка палиндромом?

В следующих четырех упражнениях обсуждаются некоторые варианты магазинных автоматов.

***9.21.** *k-головчатым двусторонним детерминированным магазинным автоматом* называют двусторонний детерминированный магазинный автомат с k независимыми считывающими головками на входной ленте. Покажите, как распознать за время $O(n^k)$, допускается ли k-головчатым двусторонним детерминированным магазинным автоматом входная строка длины n.

9.22. *Односторонним магазинным автоматом* называют магазинный автомат, никогда не сдвигающий свою считывающую головку влево. Недетерминированным магазинным автоматом называют автомат, у которого есть нуль или более возможностей для выбора для очередного шага. Таким образом, различаются четыре семейства магазинных автоматов: односторонний

детерминированный, односторонний недетерминированный, двусторонний детерминированный и двусторонний недетерминированный. Язык, допускаемый односторонним недетерминированным магазинным автоматом, называется *бесконтекстным*.

а) Покажите, что $\{wcw^R | w \in \{a, b\}^*\}$ допускается некоторым односторонним детерминированным магазинным автоматом.

б) Покажите, что $\{ww^R | w \in \{a, b\}^*\}$ допускается некоторым односторонним недетерминированным магазинным автоматом. (Этот язык не может допускаться никаким односторонним детерминированным магазинным автоматом.)

в) Покажите, что $\{wwx | w \in \{a, b\}^*, |w| \geq 1$ допускается некоторым двусторонним недетерминированным магазинным автоматом. (Этот язык не может допускаться никаким односторонним недетерминированным магазинным автоматом.)

***9.23.** Покажите, что язык L порождается бесконтекстной грамматикой в нормальной форме Хомского[9] тогда и только тогда, когда он допускается некоторым односторонним недетерминированным магазинным автоматом.

***9.24.** Покажите, что за время $O(n^3)$ можно распознать, допускается ли входная строка длины n некоторым двусторонним недетерминированным магазинным автоматом.

9.25. Постройте позиционные деревья для строк

а) *baaaab*\$;

б) *abababa*\$.

9.26. Покажите, что позиционное дерево для $a^n b^n a^n b^n$\$ содержит $n^2 + 6n + 1$ вершин.

***9.27.** Покажите, что позиционное дерево для случайной входной строки x содержит $O(|x|)$ вершин при условии, что символы во всех позициях выбираются из фиксированного алфавита равновероятно и независимо друг от друга.

9.28. Для позиционных деревьев из упражнения 9.25 найдите следующее:

а) вспомогательные деревья;

б) двоичные векторы B_v для каждой вершины v.

9.29. Покажите, что для каждого позиционного дерева существует вспомогательное дерево.

9.30. Завершите доказательство леммы 9.4, показав, что деревья T_i и A_i правильно строятся по деревьям T_{i+1} и A_{i+1}.

---

[9] См. определение перед упражнением 2.34.

***9.31.** Покажите, что с помощью позиционного дерева для x можно проверить, является ли какая-то строка из $y_1, y_2, ..., y_m$ подстрокой в строке x, за время порядка

$$|y_1| + |y_2| + ... + |y_m|.$$

9.32. Напишите алгоритм, который находил бы длиннейшую общую подстроку двух данных строк x и y. Какова временная сложность вашего алгоритма?

9.33. Напишите эффективный алгоритм, который по строкам x и $b_1b_2...b_p$ находил бы для каждого i, $1 \le i \le p$, длиннейшую подстроку в строке x, являющуюся префиксом строки $b_ib_{i+1}...b_p$.

9.34. Напишите эффективный алгоритм, который по двум данным строкам, $x = a_1a_2...a_n$ и $y = b_1b_2...b_n$, находил бы в алфавите I кратчайшее представление для строки y в виде $c_1c_2...c_m$, где c_i — символ из алфавита I или символ, обозначающий подстроку строки x. Например, если $x = abbabb$ и $y = ababbaa$, то $[1 : 2][4 : 6]aa$ — представление строки y длины 4 (при этом $[i : j]$ обозначает подстроку $a_ia_{i+1}...a_j$ строки x). (*Подсказка*: воспользуйтесь упражнением 9.33.)

9.35. Докажите, что уплотненное позиционное дерево для строки длины n содержит не более $4n - 2$ вершин.

****9.36.** Напишите алгоритм, который за время $O(n)$ находил бы уплотненное позиционное дерево для строки длины n.

9.37. Строка $x = a_1a_2...a_n$ называется *подпоследовательностью* строки $y = b_1b_2...b_p$, если строка $a_1a_2...a_n = b_{i_1}b_{i_2}...b_{i_n}$ для некоторых $i_1 < i_2 < ... < i_n$ (т.е. строка x получается из строки y вычеркиванием нуля или более символов). Напишите алгоритм сложности $O(|x| \cdot |y|)$ для нахождения самой длинной общей подпоследовательности строк x и y.

9.38. Напишите алгоритм, который по двум заданным строкам x и y находил бы кратчайшую последовательность вставок и удалений одного символа, превращающую x в y.

Проблемы для исследования

9.39. Можно ли улучшить оценку времени в упражнении 9.10?

9.40. Необходимо ли время $O(|\alpha||x|)$ для распознавания вхождения в x строки из языка, представленного регулярным выражением α?

9.41. K-головчатый двусторонний детерминированный магазинный автомат можно промоделировать на машине RAM за время $O(n^k)$ (см. упражнение 9.21). Может ли каждый бесконтекстный язык допускаться некоторым двухголовчатым двусторонним детерминированным магазинным автома-

том? Если да, то каждый бесконтекстный язык можно было бы распознать на некоторой машине RAM за время $O(n^2)$.

9.42. Существует ли бесконтекстный язык, не допускаемый никаким двусторонним детерминированным магазинным автоматом?

9.43. Существует ли язык, допускаемый некоторым двусторонним недетерминированным магазинным автоматом, но не допускаемый никаким двусторонним детерминированным магазинным автоматом?

9.44. Можно ли улучшить оценку времени в упражнении 9.37? (См. Aho, Hirschberg, Ullman [1974], где для получения оценок использована модель дерева решений.)

Библиографические замечания

Эквивалентность конечных автоматов и регулярных выражений установлена Kleene [1956]. Недетерминированные конечные автоматы изучались Rabin and Scott [1959], которые доказали их эквивалентность детерминированным. Алгоритм сопоставления строк, задаваемых регулярным выражением (алгоритм 9.1), построен на основе алгоритма Thompson [1968]. Ehrenfeucht and Zeiger [1974] обсуждают сложность недетерминированного конечного автомата как устройства для классификации образцов. Распознавание вхождения одной строки в другую за линейное время (алгоритм 9.3) заимствован у Morris and Pratt [1970]. Моделирование двустороннего детерминированного магазинного автомата за линейное время, а также обобщение в упражнении 9.21 принадлежат Cook [1971a]. Свойства двустороннего детерминированного магазинного автомата изучали Gray, Harrison, Ibarra [1967]. Моделирование двустороннего недетерминированного магазинного автомата за время $O(n^3)$ (упражнение 9.24) описано в Aho, Hopkroft and Ullman [1968]. Упражнение 9.15, *б*, принадлежит D. Chester, а упражнение 9.19, *в*, — V. Pratt. Решение упражнения 9.19, *в*, приведено в работе Knuth and Pratt [1971].

Материал раздела 9.5, касающийся позиционных деревьев, а также понятие уплотненного позиционного дерева можно найти в Weiner [1973]. Там же изложены решения упражнений 9.20 и 9.31–9.36 вместе с некоторыми другими приложениями; см. также Knuth [1973b]. Karp, Miller and Rosenberg [1972] приводят несколько алгоритмов распознавания образцов, касающихся повторяющихся строк. Упражнения 9.9 и 9.10 о совпадении строк с несущественными символами принадлежат Fischer and Paterson [1974]. Fischer [1974] приводят решение упражнения 9.3. Алгоритмы для задачи об общей подпоследовательности (см. упражнение 9.37) обсуждаются в их статье и статье Hirschberg [1973].

ГЛАВА 10

NP-полные задачи

Сколько вычислений должно требовать решение задачи, чтобы мы отнесли ее к категории действительно трудных? Задача считается безусловно трудноразрешимой, если ее невозможно решить быстрее, чем за экспоненциальное время. Согласно этой схеме классификации задачи, решаемые алгоритмами полиномиальной сложности, являются легкоразрешимыми. Однако следует иметь в виду, что, хотя экспоненциальная функция, например 2^n, растет быстрее любой полиномиальной функции, зависящей от n, при небольших значениях n алгоритм, требующий $O(2^n)$ времени, может оказаться эффективнее многих алгоритмов с полиномиально ограниченным временем работы. Например, сама функция 2^n не превосходит n^{10} до значения n, равного 59. Тем не менее скорость роста экспоненциальной функции настолько высокая, что мы будем называть задачу *трудноразрешимой*, если у всех алгоритмов ее решения сложность, по меньшей мере, экспоненциальная.

В этой главе мы приведем свидетельства того, что некоторый класс задач, а именно полных задач, для которых существует недетерминированный алгоритм решения с полиномиальным временем работы (кратко, *NP*-полных), вероятно, содержит только трудноразрешимые задачи. К этому классу относятся многие классические комбинаторные задачи, например, задача о коммивояжере, о гамильтоновом цикле, задача целочисленного линейного программирования, причем все задачи из этого класса считаются эквивалентными в том смысле, что если хотя бы одна из них является легкоразрешимой, то и все остальные тоже. Поскольку многие из этих задач десятилетиями исследовались математиками и специалистами по компьютерным наукам и ни для одной из них не удалось найти алгоритм полиномиальной сложности, естественно предположить, что таких полиномиальных алгоритмов не существует в принципе, и, следовательно, все задачи этого класса можно рассматривать как трудноразрешимые.

Мы также рассмотрим еще один класс задач, называемых полными задачами с полиномиальной пространственной сложностью, которые по меньшей мере так же сложны, как *NP*-полные задачи, но еще не доказано, что они являются трудноразрешимыми. В главе 11 будут изучены некоторые задачи, относительно которых действительно можно доказать, что они являются трудноразрешимыми.

10.1. Недетерминированные машины Тьюринга

По причинам, которые вскоре станут ясными, ключевым понятием в теории *NP*-полных задач является недетерминированная машина Тьюринга.[1] Мы уже обсуждали недетерминированные конечные автоматы, которые на каждом шаге могут переходить в несколько разных состояний. По аналогии недетерминированная машина Тьюринга имеет конечное число возможных шагов, среди которых выбирается какой-то один. Входная строка x допускается, если по крайней мере одна последовательность шагов для входа x приводит к допускающему мгновенному описанию.

При данной входной строке x можно считать, что недетерминированная машина Тьюринга M параллельно выполняет все возможные последовательности шагов, пока не достигнет допускающего мгновенного описания или состояния, в котором дальнейшие шаги невозможны. Иначе говоря, после i шагов можно считать, что существует много экземпляров машины M. Каждый экземпляр представляет собой мгновенное описание, в котором машина M может оказаться после i шагов. На $(i+1)$-м шаге экземпляр C порождает j своих экземпляров, если машина Тьюринга, находясь в мгновенном описании C, может выбрать следующий шаг j способами.

Таким образом, возможные последовательности шагов машины M на входе x можно представить в виде дерева мгновенных описаний. Каждый путь из корня в лист этого дерева представляет некоторую последовательность возможных шагов. Если σ — кратчайшая последовательность шагов, которая заканчивается допускающим мгновенным описанием, то, как только машина M сделает $|\sigma|$ шагов, она остановится и допустит вход x. Время, затрачиваемое на обработку входа x, равно длине последовательности σ. Если на входе x никакая последовательность шагов не приводит к допускающему мгновенному описанию, то машина M отвергает x, а время, затрачиваемое на обработку x, остается неопределенным.

Часто удобно представлять себе, что машина M угадывает только шаги из последовательности σ и проверяет, что σ на самом деле завершается допускающим мгновенным описанием. Однако поскольку детерминированная машина обычно не может заранее угадать допускающую последовательность шагов, детерминированное моделирование машины M потребовало бы обхода дерева всех возможных последовательностей шагов на входе x в определенном порядке до обнаружения кратчайшей последовательности, заканчивающейся допускающим мгновенным описанием. Если никакая последовательность шагов не приводит к допуску входной строки, то детерминированное моделирование машины M могло бы продолжаться бесконечно долго, если только нет какой-нибудь априорной оценки длины кратчайшей допускающей последовательности. Следовательно, естествен-

[1] Читателям, не знакомым с машинами Тьюринга, рекомендуем перечитать раздел 1.6.

но предположить, что недетерминированные машины Тьюринга способны выполнять задания, которые не могут выполнить никакие детерминированные машины Тьюринга с эквивалентной временной или пространственной сложностью. Тем не менее все еще не решен важный вопрос о существовании языков, допускаемых недетерминированной машиной Тьюринга с заданной временной или емкостной сложностью, но не допускаемых никакой детерминированной машиной Тьюринга с той же сложностью.

Определение. *k-ленточной недетерминированной машиной Тьюринга* (Nondeterministic Turing Machine — NDTM) M называют семерку $(Q, T, I, \delta, b, q_0, q_f)$, у которой значения всех компонентов такие же, как и у обычной детерминированной машины Тьюринга, за исключением того, что здесь функция переходов δ представляет собой отображение множества $Q \times T^k$ во множество подмножеств в $Q \times (T \times \{L, R, S\})^k$. Иначе говоря, по данному состоянию и списку из k ленточных символов функция δ выдает конечное множество вариантов выбора следующего шага. Каждый вариант выбора состоит из нового состояния, k новых ленточных символов и k сдвигов головок. Заметим, что недетерминированная машина Тьюринга M может выбирать любой шаг, но не может выбрать следующее состояние из одного шага, а новые ленточные символы — из другого или выполнить какую-то другую комбинацию шагов.

Мгновенные описания для недетерминированной машины Тьюринга определяются точно так же, как и для детерминированной машины Тьюринга. Данная недетерминированная машина Тьюринга $M = (Q, T, I, \delta, b, q_0, q_f)$ делает *шаг*, исходя из своего текущего состояния, скажем, q, и символов, сканируемых каждой из k головок, скажем, $X_1, X_2, ..., X_k$. Затем машина выбирает некоторый элемент $(r, (Y_1, D_1), ..., (Y_k, D_k))$ из множества $\delta(q, X_1, X_2, ..., X_k)$. Этот элемент указывает, что новым состоянием должно стать r, на i-й ленте следует напечатать Y_i вместо X_i и сдвинуть головку в направлении, обозначенном знаком D_i, $1 \le i \le k$. Если при некотором выборе следующего шага мгновенное описание C переходит в мгновенное описание D, то пишут $C \vdash_M D$ (или $C \vdash D$, если ясно, о какой машине M идет речь). Заметим, что для данной недетерминированной машины Тьюринга M могут существовать несколько таких D, что $C \vdash D$, но если машина M является детерминированной, то для каждого C существует не более одного такого D.

Запись $C_1 \vdash^* C_k$ означает, что для некоторого k выполняется условие $C_1 \vdash C_2 \vdash ... \vdash C_k$ или $C_1 = C_k$. Недетерминированная машина Тьюринга M *допускает* строку w, если $(q_0w, q_0, q_0, ..., q_0) \vdash^* (\alpha_1, \alpha_2, ..., \alpha_k)$, где строка α_1 (и, следовательно, $\alpha_2, ..., \alpha_k$) содержит символ заключительного состояния q_f. *Языком* $L(M)$, *допускаемым машиной* M, называют множество всех строк, допускаемых M.

Пример 10.1. Построим недетерминированную машину Тьюринга, допускающую такие строки вида $10^{i_1}10^{i_2}...10^{i_k}$, что существует такое множество $I \subseteq \{1, 2, ..., k\}$, для которого выполняется условие $\sum_{j \in I} i_j = \sum_{j \notin I} i_j$. Иначе говоря, строка w должна допускаться, если список[2] целых чисел $\{i_1, i_2, ..., i_k\}$, представленный ею, можно разбить на два подсписка так, чтобы суммы чисел в них были равны. Эта задача известна как задача о разбиении. Доказано, что она является *NP*-полной, если целые числа представлены в двоичном виде и размером задачи считается длина списка этих двоичных целых чисел.[3]

Мы построим трехленточную недетерминированную машину Тьюринга M, распознающую этот язык. Она просматривает свою входную ленту слева направо, и каждый раз, когда доходит до блока вида 0^{i_j}, на вторую или третью ленту добавляются i_j нулей, причем выбор ленты не детерминирован. После того как машина дойдет до конца входной строки, она проверит, совпадают ли числа нулей на лентах 2 и 3, и, если они совпадают, то допустит входную строку. Следовательно, если какая-нибудь последовательность выборов, указывающая, в какое из множеств помещать очередное число i_j — в первое (на ленту 2) или во второе (на ленту 3), приводит к равным суммам, то эта недетерминированная машина Тьюринга допустит входную строку. Последовательности шагов, приводящие к неравным строкам на лентах 2 и 3, не учитываются, если хотя бы одна последовательность выборов приводит к результату. Формально эту машину можно представить следующим образом:

$$M = (\{q_0, q_1, ..., q_5\}, \{0, 1, \text{b}, \$\}, \{0, 1\}, \delta, \text{b}, q_0, q_5),$$

где функция переходов δ задана на рис. 10.1.

На рис. 10.2 показаны две из многих последовательностей шагов, которые может сделать на входе 1010010 эта недетерминированная машина Тьюринга. Первая приводит к допускающему состоянию, вторая нет. Поскольку по крайней мере одна последовательность шагов приводит к допускающему состоянию, эта машина допускает строку 1010010. □

---

[2]Это список, а не множество из-за возможных повторений.

[3]Как мы увидим, способ кодирования данных очень важен для формулировки задачи. Нетрудно показать, что распознавание языка, допускаемого недетерминированной машиной Тьюринга в примере 10.1, с помощью детерминированой машины Тьюринга, имеет временную сложность $O_{TM}(n^2)$, где n — длина входной строки. Однако если входная строка представлена в двоичном виде, то ее длина будет равна сумме логарифмов $i_1, i_2, ..., i_k$ и та же стратегия порождает алгоритм, имеющий сложность $O_{TM}(c^n)$, где n — это теперь длина двоичного входа и $c > 1$.

Состояние	Текущий символ на ленте			Новый символ, сдвиг головки на ленте			Новое состояние	Примечание
	1	2	3	1	2	3		
q_0	1	b	b	1, S	\$, R	\$, R	q_1	Машина помечает левые концы лент 2 и 3 знаком \$ и переходит в состояние q_1
q_1	1	b	b	$\begin{cases}1,R\\1,R\end{cases}$	b, S b, S	b, S b, S	q_2 q_3	Здесь мы выбираем, куда записать следующий блок — на ленту 2 (q_2) или ленту 3 (q_3)
q_2	0	b	b	0, R	0, R	b, S	q_2	Машина переписывает блок из нулей на ленту 2, затем по достижении единицы на ленте 1 возвращает состояние q_1. Если же на ленте 1 достигнут символ b, то она переходит в состояние q_4, чтобы сравнить длины лент 2 и 3.
	1	b	b	1, S	b, S	b, S	q_1	
	b	b	b	b, S	b, L	b, L	q_4	
q_3	0	b	b	0, R	b, S	0, R	q_3	То же, что и в состоянии q_2, но запись выполняется на ленту 3
	1	b	b	1, S	b, S	b, S	q_1	
	b	b	b	b, S	b, L	b, L	q_4	
q_4	b	0	0	b, S	0, L	0, L	q_4	Сравниваем длины лент 2 и 3
	b	\$	\$	b, S	\$, S	\$, S	q_5	
q_5								Допустить

Рис. 10.1. Переходы (шаги) недетерминированной машины Тьюринга. Каждая строка представляет один вариант выбора

Определение. Говорят, что недетерминированная машина Тьюринга M имеет *временную сложность* $T(n)$, если для всякой допускаемой входной строки длины n существует последовательность, состоящая не более чем из $T(n)$ шагов, которая приводит в допускающее состояние, и *пространственную сложность* $S(n)$, если для всякой допускаемой входной строки длины m существует последовательность шагов, приводящая в допускающее состояние, в которой число просмотренных головкой ячеек на каждой ленте не превосходит $S(n)$.

$(q_0 1010010, q_0, q_0)$	$(q_0 1010010, q_0, q_0)$
$\vdash (q_1 1010010, \$q_1, \$q_1)$	$\vdash (q_1 1010010, \$q_1, \$q_1)$
$\vdash (1q_2 010010, \$q_2, \$q_2)$	$\vdash (1q_3 010010. \$q_3, \$q_3)$
$\vdash (10q_2 10010, \$0q_2, \$q_2)$	$\vdash (10q_3 10010, \$q_3, \$0q_3)$
$\vdash (10q_1 10010, \$0q_1\ \$q_1)$	$\vdash (10q_1 10010, \$q_1, \$0q_1)$
$\vdash (101q_3 0010, \$0q_3, \$q_3)$	$\vdash (101q_3 0010, \$q_3, \$0q_3)$
$\vdash (1010q_3 010, \$0q_3, \$0q_3)$	$\vdash (1010q_3 010, \$q_3, \$00q_3)$
$\vdash (10100q_3 10, \$0q_3, \$00q_3)$	$\vdash (10100q_3 10, \$q_3, \$000q_3)$
$\vdash (10100q_1 10, \$0q_1\ \$00q_1)$	$\vdash (10100q_1 10, \$q_1, \$000q_1)$
$\vdash (101001q_2 0, \$0q_2, \$00q_2)$	$\vdash (101001q_3 0, \$q_3, \$000q_3)$
$\vdash (1010010q_2, \$00q_2, \$00q_2)$	$\vdash (1010010q_3, \$q_3, \$0000q_3)$
$\vdash (1010010q_4, \$0q_4 0, \$0q_4 0)$	$\vdash (1010010q_4, q_4\$, \$000q_4 0)$
$\vdash (1010010q_4, \$q_4 00, \$q_4 00)$	Остановка, следующего мгновенного описания нет.
$\vdash (1010010q_4, q_4\$00, q_4\$00)$	
$\vdash (1010010q_5, q_5\$00, q_5\$00)$	
Допустить	

Рис. 10.2. Две возможные последовательности шагов недетерминированной машины Тьюринга

Пример 10.2. Недетерминированная машина Тьюринга из примера 10.1 имеет временную сложность $2n + 2$ (худшим случаем является входная строка, состоящая из одних единиц) и пространственную сложность $n + 1$. Другие разумные схемы кодирования задачи о разбиении также дают эту сложность. Например, пусть $B(i)$ — двоичное представление целого числа i и

$$L_1 = \{\#B(i_1)\#B(i_2)...\#B(i_k) | \text{существует такое множество } I \subseteq \{1, 2, ..., k\}, \text{ что } \sum_{j \in I} i_j = \sum_{j \notin I} i_j \},$$

где # — разделительный маркер. Чтобы распознать язык L_1, можно построить новую недетерминированную машину Тьюринга M_1, которая работает так же, как недетерминированная машина Тьюринга, представленная на рис. 10.1. Отличие состоит в том, что вместо копирования нулей на ленту 2 или 3 машина M_1 запоминает на лентах двоичные числа. Каждое новое двоичное число, встречающееся во входной строке, прибавляется к числу на той или другой ленте.

Чтобы обработать список целых чисел $i_1, i_2, ..., i_k$, машина M_1 может просматривать входную строку $x = \#B(i_1)\#B(i_2)...\#B(i_k)$. Машина M при решении той же задачи просматривала бы входную строку $w = 10^{i_1}10^{i_2}...10^{i_k}$, которая может оказаться экспоненциально длиннее строки x. Таким образом, хотя машина M_1 в определенном смысле работает экспоненциально быстрее, чем недетерминированная машина Тьюринга, представленная на рис. 10.1, ее временная и пространственная

сложности все равно имеют порядок $O(n)$, где n — длина входа, поскольку входная строка соответственно укорочена. □

С помощью моделирования можно показать, что любой язык, допускаемый какой-нибудь недетерминированной машиной Тьюринга, допускается также и некоторой детерминированной машиной Тьюринга, но, по-видимому, это происходит за счет значительного увеличения временной сложности. Наименьшая верхняя граница, которую можно установить для такого моделирования, является экспоненциальной. Иначе говоря, если $T(n)$ — разумная функция, задающая временную сложность (разумная в том смысле, что она "конструируется по времени", этот термин определяется в главе 11), то для каждой недетерминированной машины Тьюринга M, время работы которой ограничено функцией $T(n)$, можно найти такие константу c и детерминированную машину Тьюринга M', что $L(M) = L(M')$ и время работы машины M' составляет $O_{\text{ТМ}}(c^{T(n)})$.

Этот результат можно доказать, построив детерминированную машину Тьюринга M', моделирующую машину M путем полного перебора возможных вычислений. Существует константа d, ограничивающая количество вариантов выбора очередного шага машины M во всех ситуациях. Следовательно, последовательность, содержащую вплоть до $T(n)$ шагов работы машины M, можно представить строкой в алфавите $\Sigma = \{0, 1, ..., d - 1\}$, длина которой не превосходит $T(n)$. Машина M' моделирует поведение машины M на входной строке x длины n следующим образом. Машина M' последовательно порождает в лексикографическом порядке все строки из алфавита Σ^*, длины которых не превосходят $T(n)$. Таких строк не более $(d + 1)^{T(n)}$. Когда порождается новая строка w, машина M' моделирует последовательность σ_w шагов машины M, представленную строкой w. Если последовательность σ_w приводит к тому, что машина M допускает строку x, то машина M' также допускает строку x. Если последовательность σ_w не является корректной последовательностью шагов машины M или σ_w не приводит к допуску строки x машиной M, то M' повторяет этот процесс со следующей строкой из алфвита Σ^*.

Машина M' может выполнить моделирование последовательности σ_w за время $O_{\text{ТМ}}(T(n))$. Порождение очередной строки w требует выполнения $O_{\text{ТМ}}(T(n))$ шагов. Следовательно, все моделирование работы недетерминированной машины Тьюринга M может занять время порядка $O_{\text{ТМ}}(T(n)(d+1)^{T(n)})$, которое не превосходит $O_{\text{ТМ}}(c^{T(n)})$ для некоторой константы c. Детали моделирования оставляем читателям в качестве упражнения.

Следует, однако, подчеркнуть, что не известно никаких нетривиальных нижних оценок, касающихся сложности моделирования недетерминированной машины Тьюринга с помощью детерминированной машины Тьюринга. Иначе говоря, не известен такой язык L, который может допустить какая-то недетерминирован-

ная машина Тьюринга с временной сложностью $T(n)$, но про который можно доказать, что он не допускается никакой детерминированной машиной Тьюринга с временной сложностью $T(n)$. Ситуация, связанная с пространственной сложностью, намного проще.

Определение. Функцию $S(n)$ назовем *конструируемой по пространству*, если некоторая детерминированная машина Тьюринга M, начав обработку заданной входной строки длины n, поместит специальный маркер на $S(n)$-ю ячейку одной из своих лент, просмотрев не более $S(n)$ ячеек на каждой ленте. Самые обычные функции, например полиномы с целыми коэффициентами, 2^k, $n!$, $\lceil n\log(n+1)\rceil$ являются конструируемыми по пространству.

Можно показать, что если $S(n)$ — конструируемая по пространству функция, а M — недетерминированная машина Тьюринга с пространственной сложностью, ограниченной функцией $S(n)$, то найдется такая детерминированная машина Тьюринга M' с пространственной сложностью $O(S^2(n))$, что $L(M) = L(M')$.

Одна из стратегий, с помощью которой машина M' может моделировать машину M, заключается в интересном приложении метода "разделяй и властвуй". Если пространственная сложность k-ленточной недетерминированной машины Тьюринга $M = (Q, T, I, \delta, b, q_0, q_f)$ ограничена функцией $S(n)$, конструируемой по пространству, то для некоторой константы c количество различных мгновенных описаний, в которые машина M может попасть, начав работу с входной строки длины n, не превосходит $c^{S(n)}$, а точнее, $\|Q\|(\|T\|+1)^{kS(n)}(S(n))^k$, где первый множитель представляет количество состояний, второй ограничивает количество возможных состояний лент, а последний — количество возможных положений головок. Следовательно, если $C_1 \vdash_M^* C_2$, то машина M может перейти из мгновенного описания C_1 в мгновенное описание C_2 по некоторому пути, состоящему не более чем из $c^{S(n)}$ шагов. Можно выяснить, осуществим ли переход $C_1 \vdash_M^* C_2$ за $2i$ шагов, проверив для всех C_3, осуществимы ли переходы $C_1 \vdash_M^* C_3$ и $C_3 \vdash_M^* C_2$ за i шагов.

Таким образом, стратегия, на которой основана работа M', состоит в том, чтобы установить с помощью описанного ниже алгоритма, приведет ли начальное мгновенное описание C_0 к какому-нибудь допускающему мгновенному описанию.

Алгоритм 10.1. Детерминированное моделирование недетерминированной машины Тьюринга

Вход. Недетерминированная машина Тьюринга M с границей $S(n)$ на пространственную сложность, где $S(n)$ — конструируемая по пространству функция, и входная строка w длины n.

Выход. "Да", если $w \in L(M)$, и "нет" в противном случае.

Метод. Рекурсивная процедура TEST(C_1, C_2, i), приведенная на рис. 10.3, распознает, осуществим ли переход $C_1 \overset{*}{\underset{M}{\vdash}} C_2$ за i шагов. Если да, то она принимает значение **true**, в противном случае **false**. В этом алгоритме C_1 и C_2 обозначают мгновенные описания, в которых на каждой ленте использовано не более $S(n)$ ячеек.

Полный алгоритм состоит в вызове процедуры TEST(C_0, C_f, $c^{S(n)}$)) для каждого допускающего мгновенного описания C_f, где C_0 — начальное мгновенное описание машины M для входа w. Если обнаружится, что один из таких вызовов выдал значение **true**, то ответом алгоритма будет "правда", в противном случае — "ложь". □

```
procedure TEST(C1, C2, i):
if i = 1 then
    if C1 ⊢ C2 или C1 = C2 then return true
        else return false
    else
        begin
            for каждого мгновенного описания C3, в котором
                  на каждой ленте использовано не более S(n) ячеек do
                if TEST(C1, C3, ⌈i/2⌉) и TEST(C3, C2, ⌊i/2⌋)
                    then return true;
            return false
        end
```

Рис. 10.3. Процедура TEST

Теорема 10.1. Если M — недетерминированная машина Тьюринга с конструируемой по пространству пространственной сложностью $S(n)$, то найдется такая детерминированная машина Тьюринга M' с пространственной сложностью $O(S^2(n))$, что $L(M) = L(M')$.

Доказательство. Доказательство сводится к реализации алгоритма 10.1 с помощью машины Тьюринга. Из раздела 2.5 мы знаем, как моделировать рекурсивную процедуру TEST на машине RAM. Ту же стратегию со стеком можно применить на машине Тьюринга. Поскольку аргументами процедуры TEST, занимающими значительный объем память, являются мгновенные описания длины $O(S(n))$, то на ленте необходимо расположить фреймы стека того же размера; это можно сделать потому, что функция $S(n)$ является конструируемой по пространству. Анализируя процедуру TEST, мы видим, что при каждом ее рекурсивном вызове третий аргумент уменьшается, по существу, вдвое. Следовательно, можно показать, что в любой момент число фреймов стека не

превосходит $1+\log\lceil c^{S(n)} \rceil$, т.е. $O(S(n))$. Поскольку на каждый фрейм приходится $O(S(n))$ ячеек, всего для стека наша машина Тьюринга использует $O(S^2(n))$ ячеек. Остальные детали конструкции машины Тьюринга оставляем читателям в качестве упражнения. □

Чтобы упростить доказательства, часто желательно ограничиться одноленточными машинами Тьюринга. Следующая лемма позволяет это сделать, если мы готовы платить увеличением времени вычислений.

Лемма 10.1. Если язык L допускается k-ленточной недетерминированной машиной Тьюринга $M = (Q, T, I, \delta, \text{b}, q_0, q_f)$ с временной сложностью $T(n)$, то он допускается одноленточной недетерминированной машиной Тьюринга с временной сложностью $O(T^2(n))$.

Доказательство. Построим одноленточную недетерминированную машину Тьюринга M_1, допускающую язык L с временной сложностью $O(T^2(n))$. Будем считать, что лента машины M_1 имеет $2k$ дорожек (рис. 10.4). Иначе говоря, ленточными символами машины M_1 являются $2k$-членные кортежи, в которых на нечетных местах стоят символы алфавита T, а на четных — либо символ пробела b, либо специальный маркер #. Дорожки с нечетными номерами соответствуют k лентам машины M, а каждая дорожка с четным номером содержит символ b во всех ячейках, кроме одной, в которой стоит символ #. Символ # на дорожке с номером $2j$ отмечает положение головки машины M на ленте j, которой соответствует дорожка с номером $2j-1$. На рис. 10.4 головка j-й ленты сканирует ячейку i_j на j-й ленте для всех $1 \le j \le k$.

Лента 1 машины M	X_{11}	X_{12}	...	X_{1i_1}	...	
Головка на ленте 1	b	b	...	#	...	
Лента 2 машины M	X_{21}	X_{22}	...		X_{2i_2}	...
Головка на ленте 2	b	b	...		#	...
			. . .			
Лента k машины M	X_{k1}	X_{k2}	...	X_{ki_k}	...	
Головка на ленте k	b	b	...	#	...	

Рис. 10.4. Лента машины M_1

Машина M_1 моделирует один шаг машины M следующим образом. Предположим, что вначале головка машины M_1 установлена на ячейке, содержащей самую левую головку машины M.

1. Головка машины M_1 движется вправо, пока не пройдет через все k маркеров положений головок на дорожках с четными номерами. Во время этого движения машина M_1 запоминает в своем состоянии символы, сканируемые каждой из головок машины M.
2. Действия машины M_1 на шаге 1 являются детерминированными. Теперь машина M_1 делает недетерминированное разветвление: исходя из состояния машины M, которое машина M_1 запомнила в своем состоянии, и символов на лентах, сканируемых машиной M, которые M_1 также нашла, машина M_1 недетерминированно выбирает возможный шаг для машины M. Для каждого шага, возможного для M в этой ситуации, машина M_1 имеет состояние, в которое может перейти на следующем шаге.
3. Выбрав для моделирования шаг машины M, машина M_1 изменяет в соответствии с ним состояние машины M, которое она помнит в своем состоянии. Затем машина M_1 сдвигает свою головку влево, пока не пройдет через все k маркеров головок. Всякий раз, когда машина M_1 находит такой маркер, она изменяет ленточный символ, стоящий на дорожке над маркером, и сдвигает этот маркер не более чем на одну ячейку влево или вправо в соответствии с выбранным шагом.

В этот момент машина M_1 смоделировала один шаг работы машины M. Ее головка находится правее левого маркера головок не больше чем на две ячейки, так что можно найти этот маркер и повторить цикл.

Машина M_1 может допустить вход, если его допускает машина M, поскольку M_1 помнит состояние машины M. Если M допускает строку w длины n, то делает это с помощью последовательности, состоящей не более чем из $T(n)$ шагов. Очевидно, в последовательности из $T(n)$ шагов головки машины M не могут разойтись больше, чем на $T(n)$ ячеек, и, следовательно, машина M_1 может смоделировать один шаг этой последовательности не более чем за $O(T(n))$ своих шагов. Таким образом, машина M_1 допускает строку w, выполняя не более $O(T^2(n))$ шагов. Отсюда следует, что машина M_1 допускает язык L и имеет временную сложность $O(T^2(n))$. □

Следствие 1. Если язык L допускается k-ленточной детерминированной машиной Тьюринга с временной сложностью $T(n)$, то он допускается одноленточной детерминированной машиной Тьюринга с временной сложностью $O(T^2(n))$. □

Доказательство. Если в приведенном выше доказательстве машина M является детерминированной, то машина M_1 тоже является детерминированной.

Следствие 2. Если язык L допускается k-ленточной недетерминированной машиной Тьюринга с пространственной сложностью $S(n)$, то он допускается одноленточной недетерминированной машиной Тьюринга с пространственной сложностью $S(n)$.

Следствие 3. Если язык L допускается k-ленточной детерминированной машиной Тьюринга с пространственной сложностью $S(n)$, то он допускается одноленточной детерминированной машиной Тьюринга с пространственной сложностью $S(n)$.

10.2. Классы $\mathcal{P}$ и $\mathcal{NP}$

Введем два важных класса языков.

Определение. $\mathcal{P}$-TIME — это множество всех языков, допускаемых детерминированной машиной Тьюринга с полиномиальной временной сложностью.

$\mathcal{P}$-TIME = $\{L|$существуют такие детерминированные машины Тьюринга M и полином $P(n)$, что временная сложность машины M равна $P(n)$ и $L(M) = L\}$.

Определим $\mathcal{NP}$-TIME как множество всех языков, допускаемых недетерминированной машиной Тьюринга с полиномиальной временной сложностью. Для краткости мы будем часто писать $\mathcal{P}$ и $\mathcal{NP}$ вместо $\mathcal{P}$-TIME и $\mathcal{NP}$-TIME соответственно.

Прежде всего заметим, что, хотя классы $\mathcal{P}$ и $\mathcal{NP}$ определены в терминах машин Тьюринга, можно было бы использовать любую из многих других моделей вычислений. Интуитивно можно интерпретировать $\mathcal{P}$ как класс языков, распознаваемых за полиномиальное время. Например, мы показали, что если язык L допускается машиной Тьюринга с временной сложностью $T(n)$, то временная сложность его распознавания на машинах RAM или RASP при логарифмическом весовом критерии будет лежать между $k_1T(n)$ и $k_2T^4(n)$, где k_1 и k_2 — некоторые положительные константы. Таким образом, язык L допускается машиной Тьюринга с полиномиальной временной сложностью тогда и только тогда, когда существует алгоритм его распознавания, реализуемый на машинах RAM или RASM с полиномиальной сложностью при логарифмическом весовом критерии.

Можно также определить *недетерминированную* машину RAM или RASP, добавив к системе команд инструкцию CHOICE(L_1, L_2, ..., L_k), означающую, что выбор и последующее выполнение одного из операторов L_1, L_2, ..., L_k являются недетерминированными. Таким образом, класс $\mathcal{NP}$ можно также определить с по-

мощью недетерминированных машин RAM или RASP с полиномиально ограниченной временной сложностью при логарифмическом весовом критерии.

Следовательно, можно представить себе недетерминированную вычислительную машину вроде RAM или RASP, способную выполнить много различных возможных последовательностей шагов, начинающихся с данного мгновенного описания. Такое устройство может распознать за полиномиальное время многие языки, вероятно, нераспознаваемые за полиномиальное время детерминированными алгоритмами. Конечно, любая попытка прямого моделирования недетерминированного устройства N детерминированным устройством D, выполняющим все возможные последовательности шагов, требует гораздо больше времени, чем любая отдельная реализация последовательности шагов устройства N, поскольку устройство D должно прослеживать работу огромного количества копий N. На основе результатов предыдущего раздела мы можем утверждать лишь, что если язык L принадлежит классу $\mathcal{NP}$, то он допускается некоторой детерминированной машиной Тьюринга с временной сложностью $k^{p(n)}$, где k — константа и p — полином, зависящие от L.

С другой стороны, еще никто не смог доказать, что существует язык из класса $\mathcal{NP}$, не принадлежащий классу $\mathcal{P}$. Таким образом, неизвестно, является ли класс $\mathcal{P}$ собственным подклассом класса $\mathcal{NP}$. Однако можно доказать, что некоторые языки не менее трудны, чем любой язык из $\mathcal{NP}$ в том смысле, что если бы у нас был детерминированный алгоритм, распознающий один из этих не менее трудных языков за полиномиальное время, то можно было бы для каждого языка из $\mathcal{NP}$ найти детерминированный алгоритм, распознающий его за полиномиальное время. Такие языки называются NP-полными.

Определение. Язык L_0 из $\mathcal{NP}$ называется *полным для недетерминированного полиномиального времени* (или NP-полным), если выполнено следующее условие: по заданному детерминированному алгоритму распознавания L_0 с временной сложностью $T(n) \geq n$ и любому языку L из класса $\mathcal{NP}$ можно эффективно найти детерминированный алгоритм, распознающий язык L за время $T(p_L(n))$, где p_L — некоторый полином, зависящий от L. Говорят, что язык L *сводится* к языку L_0.

NP-полноту языка L_0 можно доказать, показав, что L_0 принадлежит классу $\mathcal{NP}$ и каждый язык $L \in \mathcal{NP}$ можно полиномиально преобразовать в L_0.

Определение. Язык L называется *полиномиально преобразуемым* в L_0, если некоторая детерминированная машина Тьюринга M с полиномиально ограниченным временем работы преобразует каждую строку w в алфавите языка L в такую строку w_0 в алфавите языка L_0, что $w \in L$ тогда и только тогда, когда $w_0 \in L_0$.

Если язык L является преобразуемым в язык L_0, который распознается некоторым детерминированным алгоритмом A с временной сложностью $T(n) \geq n$, то можно выяснить принадлежность строки w языку L, преобразовав строку w в строку w_0 с помощью машины M и затем применив алгоритм A для выяснения принадлежности w_0 языку L_0. Если время работы машины M ограничено полиномом $p(n)$, то $|w_0| \leq p(|w|)$. Таким образом, существует алгоритм, выясняющий принадлежность строки w языку L за время $p(|w|) + T(p(|w|)) \leq T(2p(|w|))$. Если бы функция T была полиномом (т.е. язык L_0 принадлежал бы классу $\mathcal{P}$), то этот алгоритм, распознающий язык L, работал бы полиномиально ограниченное время, и язык L также принадлежал бы классу $\mathcal{P}$.

Некоторые авторы, действительно, определяют язык L_0 как $\mathcal{NP}$-полный, если он принадлежит классу $\mathcal{NP}$ и каждый язык из $\mathcal{NP}$ является полиномиально преобразуемым в язык L_0. Это определение представляется более узким, чем предыдущее, хотя не известно, приводят ли указанные два определения *NP*-полных языков к разным классам. Определение, основанное на сводимости, означает, что если M_0 — детерминированная машина Тьюринга, распознающая *NP*-полный язык L_0 за время $T(n)$, то любой язык из класса $\mathcal{NP}$ можно распознать за время $T(p(n))$, где p — некоторый полином, с помощью детерминированной машины Тьюринга, которая обращается к машине M_0 как к подпрограмме нуль или более раз. Определение, основанное на преобразуемости, означает, что к машине M_0 можно обратиться лишь один раз и затем использовать результат ее работы заранее фиксированным образом. Хотя мы примем более широкое определение, все наши доказательства *NP*-полноты походят и для узкого определения.

Независимо от этих определений, очевидно, что если некоторый детерминированный алгоритм распознает язык L_0 за полиномиальное время, то все языки из класса $\mathcal{NP}$ можно распознать за полиномиальное время. Таким образом, либо все *NP*-полные языки принадлежат классу $\mathcal{P}$, либо ни один из них не принадлежит ему. Первое утверждение верно тогда и только тогда, когда $\mathcal{P} = \mathcal{NP}$. К сожалению, в настоящее время мы можем лишь выдвинуть гипотезу о том, что $\mathcal{P}$ — собственный подкласс класса $\mathcal{NP}$.

10.3. Языки и задачи

Мы определили $\mathcal{P}$ и $\mathcal{NP}$ как классы языков. Это решение носит двойственный характер. Во-первых, оно упрощает систему обозначений. Во-вторых, задачи из разных областей, таких, как теория графов и теория чисел, часто можно формулировать как задачи распознавания языков. Например, рассмотрим задачу, которая при каждом значении входных данных имеет ответы “да” или “нет”. Можно закодировать все возможные значения входных данных такой задачи в виде строк и переформулировать ее как задачу о распознавании языка, состоящего из всех строк,

представляющих входные данные нашей задачи, которые соответствуют ответу "да". Такие способы кодирования уже встречались в главе 9, где задачи идентификации формулировались в терминах задач распознавания языков. Однако надо аккуратно выбирать эти кодирования, потому что от них может зависеть временная сложность задачи.

Чтобы связь задачи с языком стала более явной, зададим единообразные представления задач. В частности, примем следующие соглашения.

1. Целые числа будем представлять в десятичной системе счисления.
2. Вершины графа будем представлять целыми числами 1, 2, ..., n, закодированными как десятичные числа в соответствии с соглашением 1. Ребро будем представлять строкой (i_1, i_2), где i_1 и i_2 — десятичные представления пары вершин, определяющей это ребро.
3. Булевы выражения с n пропозициональными переменными будем представлять строками, в которых символ * обозначает операцию "и", символ + — операцию "или", $\neg$ — операцию отрицания,[4] а целые числа 1, 2, ..., n представляют пропозициональные переменные. Как правило, знак * опускается. При необходимости будем ставить скобки.

Используя эти соглашения, можно утверждать, что задача принадлежит классу $\mathcal{P}$ или $\mathcal{NP}$, если ее стандартный код принадлежит классу $\mathcal{P}$ или $\mathcal{NP}$ соответственно.

Пример 10.3. Рассмотрим задачу о клике для неориентированных графов. k-кликой в графе G называют k-вершинный полный подграф (каждая пара различных вершин в этом подграфе соединена ребром) графа G. Задача о клике формулируется так: содержит ли k-клику данный граф G, где k — заданное целое число?

Пример задачи о клике для графа G на рис. 10.5 при $k = 3$ можно закодировать строкой

$$3\ (1, 2)\ (1, 4)\ (2, 3)\ (2, 4)\ (3, 4)\ (3, 5)\ (4, 5).$$

Первое число представляет значение k. За ним идут пары вершин, соединенных ребрами, причем вершина v_i представлена числом i. Таким образом, ребра в порядке перечисления выглядят так: (v_1, v_2), (v_1, v_4), ..., (v_4, v_5).

Язык L, представляющий задачу о клике, — это такое множество строк вида

$$k(i_1, j_1)\ (i_2, j_2) \ldots (i_m, j_m),$$

---

[4] Часто часто вместо $\neg(\alpha)$ мы будем писать $\overline{\alpha}$. Если α состоит из единственного литерала (переменной или ее отрицания), то скобки можно опускать.

что граф с ребрами (i_r, j_r), $1 \le r \le m$, содержит k-клику. Задачу о клике могут представлять и другие языки. Например, можно было бы потребовать, чтобы константа k следовала за графом, а не предшествовала ему. Кроме того, можно было бы использовать двоичные числа вместо десятичных. Однако для любых двух таких языков должен существовать такой полином p, что строку w, представляющую частный случай задачи о клике в одном языке, можно за время $p(|w|)$ преобразовать в строку, представляющую тот же частный случай этой задачи в другом языке. Таким образом, когда речь идет о принадлежности классу $\mathcal{P}$ или $\mathcal{NP}$, точный выбор языка для представления задачи о клике неважен, если применяется стандартное кодирование. □

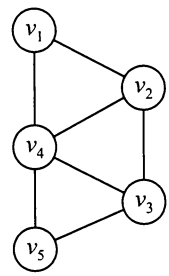


Рис. 10.5. Неориентированный граф

Задача о клике относится к классу $\mathcal{NP}$. Машина Тьюринга сначала может указать, какие k вершин образуют полный подграф, а затем проверить, что любая пара этих вершин соединена ребром, при этом для проверки достаточно $O(n^3)$ шагов, где n — длина кода задачи о клике. Этот факт демонстрирует силу недетерминированного подхода, поскольку все подмножества из k вершин проверяются параллельно независимыми экземплярами распознающего устройства. Граф на рис. 10.5 содержит три клики размера 3, а именно: $\{v_1, v_2, v_4\}$, $\{v_2, v_3, v_4\}$ и $\{v_3, v_4, v_5\}$.

Мы увидим позже, что задача о клике является NP-полной. В настоящее время не известны способы решения задачи о клике за детерминированное полиномиальное время.

Пример 10.4. Булево выражение $(p_1 + p_2) * p_3$ можно представить строкой $(1 + 2)3$, где число i представляет переменную p_i. Рассмотрим язык L, образованный всеми строками, представляющими *выполнимые* булевы выражения (т.е. выражения, принимающие значение 1 на некотором наборе значений 0 и 1). Мы утверждаем, что язык L принадлежит классу $\mathcal{NP}$. Недетерминированный алгоритм, распознающий язык L, начинает с того, что угадывает выполняющий набор значений 0 и 1 пропозициональных переменных во входной строке, если такой набор существует. Затем угаданное значение (0 или 1) каждой переменной подставляется вместо

всех вхождений этой переменной во входную строку. Чтобы закрыть пробелы, образующиеся в строке в результате подстановки одного символа 0 или 1 вместо десятичного представления пропозициональной переменной, потребуются некоторые сдвиги строки. Далее вычисляется значение полученного выражения, чтобы проверить, равно ли оно единице.

Это вычисление осуществляется за время, пропорциональное длине выражения, многими алгоритмами синтаксического анализа (см. Aho, Ullman [1972]). Но и без них нетрудно вычислить значение булевого выражения за $O(n^2)$ шагов. Следовательно, некоторая недетерминированная машина Тьюринга допускает выполнимые булевы выражения с полиномиальной временной сложностью, и потому задача распознавания выполнимости булевых выражений принадлежит классу $\mathcal{NP}$. Снова отметим, что недетерминированный подход позволил распараллелить задачу, поскольку угадывание правильного решения в действительности означает параллельную проверку всех возможных решений. Как будет показано позже, эта задача также является *NP*-полной. □

Часто нас интересуют задачи оптимизации, например поиск наибольшей клики в графе. Многие такие задачи можно преобразовать за полиномиальное время в задачи распознавания языка. Например, наибольшую клику в графе G можно найти следующим образом. Пусть n — длина представления графа G. Для каждого k от 1 до n выясняем, содержит ли граф клику размера k. Установив размер m наибольшей клики, удаляем по одной вершине до тех пор, пока удаление очередной вершины v не разрушит все оставшиеся клики размера m. Затем рассмотрим подграф G', состоящий из всех вершин графа G, смежных с вершиной v. Рекурсивно выполняя этот процесс, находим клику C размера $m - 1$ в графе G'. Наибольшей кликой для графа G будет граф $C \cup v$.

Мы предоставляем читателям самим убедиться в том, что время нахождения наибольшей клики таким методом полиномиально зависит от длины n представления графа G и от времени выяснения, содержит ли граф G клику размера k.

Часто решить задачи оптимизации вида "найти такое наибольшее k, что $P(k)$ истинно", где P — некоторое условие, а число допустимых k оценивается экспонентой от длины описания задачи n, можно с помощью бинарного поиска (см. раздел 4.3). Если из истинности условия $P(k)$ следует истинность условия $P(i)$ для $i < k$, а число k изменяется от 0 до c^n, где c — некоторая константа, то наибольшее число k, для которого условие $P(k)$ истинно, можно найти с помощью бинарного поиска, проведя $\log c^n = n \log c$ проверок условия $P(k)$. Мы снова предлагаем читателям самостоятельно убедиться в том, что оптимальное значение k можно найти за время, полиномиально зависящее от n и от наибольшего времени проверки $P(k)$.

Разработка методов подобного рода зависит от изобретательности читателя, и в дальнейшем мы будем заниматься только задачами, требующими ответа "да" или "нет".

10.4. NP-полнота и задача выполнимости

Можно показать, что задача, а точнее, ее представление в виде языка L_0, является NP-полной, доказав, что язык L_0 принадлежит классу $\mathcal{NP}$ и всякий язык из класса $\mathcal{NP}$ полиномиально преобразуется в L_0. Благодаря мощи недетерминированного подхода недетерминизма принадлежность данной задачи классу $\mathcal{NP}$ обычно доказывается без труда. Примеры 10.3 и 10.4 служат типичными иллюстрациями этого шага. Трудности вызывает доказательство того, что любая задача из класса $\mathcal{NP}$ полиномиально преобразуется в данную задачу. Однако, если установлена NP-полнота некоторой задачи L_0, можно доказать NP-полноту новой задачи L, показав, что задача L принадлежит классу $\mathcal{NP}$ и задача L_0 полиномиально преобразуется в задачу L.

Мы покажем, что задача выполнимости булевых выражений является $\mathcal{NP}$-полной. Затем докажем NP-полноту некоторых фундаментальных задач из пропозиционального исчисления и теории графов, показав, что они принадлежат классу $\mathcal{NP}$ и что задача выполнимости (или какая-нибудь задача, NP-полнота которой уже доказана) полиномиально преобразуется в них.

Для изучения важных NP-полных задач, касающихся неориентированных графов, нам понадобятся следующие определения.

Определение. Пусть $G = (V, E)$ — неориентированный граф.

1. *Вершинным покрытием* графа G называется такое подмножество $S \subseteq V$, что каждое ребро графа G инцидентно некоторой вершине из S.
2. *Гамильтоновым циклом* называется цикл в графе G, содержащий все вершины из V.
3. Граф G называется *k-раскрашиваемым*, если существует такое назначение целых чисел 1, 2, ..., k, называемых "цветами", вершинам графа G, что никаким двум смежным вершинам не присвоен один и тот же цвет. *Хроматическим числом* графа G называется такое наименьшее число k, что граф G является k-раскрашиваемым.

Для представления важных NP-полных задач об ориентированных графах понадобятся следующие определения.

Определение. Пусть $G = (V, E)$ — ориентированный граф.

1. *Множеством вершин, разрезающих циклы*, называется такое подмножество $S \subseteq V$, что каждый цикл в графе G содержит вершину из графа S.
2. *Множеством ребер, разрезающих циклы*, называется такое подмножество $F \subseteq E$, что каждый цикл в графе G содержит ребро из графа F.
3. *Ориентированным гамильтоновым циклом* называется цикл в графе G, содержащий все вершины из графа V.

Теорема 10.2. Следующие задачи принадлежат классу $\mathcal{NP}$:

1. (*Выполнимость*) является ли выполнимым данное булево выражение?
2. (*Клика*) Содержит ли данный неориентированный граф k-клику?
3. (*Вершинное покрытие*) Имеет ли данный неориентированный граф вершинное покрытие размера k?
4. (*Гамильтонов цикл*) Имеет ли данный неориентированный граф гамильтонов цикл?
5. (*Раскрашиваемость*) Является ли данный неориентированный граф k-раскрашиваемым?
6. (*Множество вершин, разрезающих циклы*) Имеет ли данный ориентированный граф k-элементное множество вершин, разрезающих циклы?
7. (*Множество ребер, разрезающих циклы*) Имеет ли данный ориентированный граф k-элементное множество ребер, разрезающих циклы?
8. (*Ориентированный гамильтонов цикл*) Имеет ли данный ориентированный граф ориентированный гамильтонов цикл?
9. (*Покрытие множествами*) Существует ли для данного семейства множеств $S_1, S_2, ..., S_n$ такое подсемейство из k множеств $S_{i_1}, S_{i_2}, ..., S_{i_k}$,[5] что
$$\bigcup_{j=1}^{k} S_{i_j} = \bigcup_{j=1}^{k} S_j ?$$
10. (*Точное покрытие*) Существует ли для данного семейства множеств $S_1, S_2, ..., S_n$ подсемейство попарно непересекающихся множеств, являющееся покрытием?

[5]Конечные множества мы кодируем строками вида $\{i_1, i_2, ..., i_m\}$. где i —десятичные (или двоичные) целые числа, представляющие элементы этих множеств. Если во всех множествах всего n разных элементов, мы представляем j-й элемент целым числом j.

Доказательство. Принадлежность задач 1 и 2 классу $\mathcal{NP}$ продемонстрирована соответственно в примерах 10.4 и 10.3. Аналогично для каждой из остальных задач надо построить недетерминированную машину Тьюринга (или недетерминированную машину RAM) полиномиальной сложности, которая угадывает решение и проверяет, что это действительно решение. Детали оставляем читателям в качестве упражнения. □

Докажем, что всякий язык из класса $\mathcal{NP}$ полиномиально преобразуется в задачу выполнимости. Тем самым будет установлено, что задача выполнимости булевых выражений является *NP*-полной.

Теорема 10.3. Задача выполнимости булевого выражения является *NP*-полной.

Доказательство. Мы уже знаем, что задача выполнимости принадлежит классу $\mathcal{NP}$. Необходимо лишь показать, что всякий язык L из класса $\mathcal{NP}$ полиномиально преобразуется в задачу выполнимости. Пусть M — недетерминированная машина Тьюринга, распознающая язык L за полиномиальное время, и пусть на ее вход подана строка w. На основе машины M и строки w можно построить булево выражение w_0, которое является выполнимым тогда и только тогда, когда машина M допускает строку w. Основная трудность доказательства — показать, что для каждой машины M найдется алгоритм, который построит булево выражение w_0 из строки w за время, ограниченное полиномом. Этот полином зависит от M.

В силу леммы 10.1 каждый язык из класса $\mathcal{NP}$ распознается недетерминированной одноленточной машиной Тьюринга за полиномиальное время. Следовательно, можно считать, что машина M имеет всего одну ленту. Пусть состояниями M являются $q_1, q_2, ..., q_s$, а символами на ленте — $X_1, X_2, ..., X_m$. Пусть $p(n)$ — временная сложность машины M. Предположим, что строка w, поданная на вход машины M, имеет длину n. Таким образом, если машина M допускает строку w, то делает это не более чем за $p(n)$ шагов. Если машина M допускает строку w, то найдется хотя бы одна такая последовательность мгновенных описаний $Q_0, Q_1, ..., Q_q$, что Q_0 — начальное мгновенное описание, $Q_{i-1} \vdash Q_i$ для $1 \le i \le q$; Q_q — допускающее мгновенное описание, $q \le p(n)$ и ни одно мгновенное описание не занимает более $p(n)$ ячеек на ленте.

Построим булево выражение w_0, которое моделирует последовательность мгновенных описаний машины M. Любое присвоение значений "истина" (число 1) и "ложь" (число 0) переменным выражения w_0 соответствует самое большее одной последовательности мгновенных описаний, которая может быть невозможной. Булево выражение w_0 примет значение 1 тогда и только тогда, когда это присваивание значений переменным соответствует последовательности мгновенных

описаний $Q_0, Q_1, ..., Q_q$, ведущей к допуску входной строки. Иначе говоря, булево выражение w_0 будет выполнимым тогда и только тогда, когда M допускает w. В качестве пропозициональных переменных в w_0 употребляются следующие переменные. Перечислим их вместе с их подразумеваемой интерпретацией.

1. $C\langle i, j, t\rangle = 1$ тогда и только тогда, когда i-я ячейка на входной ленте машины M содержит символ X_j в момент времени t. Здесь $1 \le i \le p(n)$, $1 \le j \le m$ и $0 \le t \le p(n)$.
2. $S\langle k, t\rangle = 1$ тогда и только тогда, когда машина M в момент времени t находится в состоянии q_k. Здесь $1 \le k \le s$ и $0 \le t \le p(n)$.
3. $H\langle i, t\rangle = 1$ тогда и только тогда, когда головка в момент времени t сканирует i-ю ячейку. Здесь $1 \le i \le p(n)$ и $0 \le t \le p(n)$.

Таким образом, количество пропозициональных переменных равно $O(p^2(n))$ и их можно представить двоичными числами, содержащими не более $c \log n$ разрядов, где c — константа, зависящая от p. В дальнейшем будет удобно интерпретировать пропозициональную переменную как один символ, а не $c \log n$ символов. Эта потеря множителя $c \log n$ не может отразиться на полиномиальной ограниченности времени, затрачиваемого на вычисление той или иной функции.

Из этих пропозициональных переменных построим булево выражение w_0, следуя структуре последовательности мгновенных описаний $Q_0, Q_1, ..., Q_q$. В этом построении мы воспользуемся предикатом $U(x_1, x_2, ..., x_r)$, принимающим значение 1, когда только один из аргументов $x_1, x_2, ..., x_r$ принимает значение 1. Предикат U можно выразить булевым выражением вида

$$U(x_1, x_2, ..., x_r) = (x_1 + x_2 + ... + x_r)\left(\prod_{\substack{i,j \\ i \ne j}} (\neg x_i + \neg x_j)\right). \qquad (10.1)$$

Первый множитель в (10.1) утверждает, что по крайней мере одна из переменных x_i является истинной. Остальные $r(r-1)/2$ множителей утверждают, что никакие две переменные не являются истинными одновременно. Заметим, что формальная запись выражения $U(x_1, x_2, ..., x_r)$ имеет длину $O(r^2)$.[6]

Если машина M допускает строку w, то найдется допускающая последовательность мгновенных описаний $Q_0, Q_1, ..., Q_q$, машины M в процессе обработки строки w. Для упрощения изложения, не уменьшая общности, будем считать, что машина M модифицирована так, что она делает ровно $p(n)$ шагов, а все ее мгновенные описания содержат $p(n)$ ячеек. Этого можно добиться, заставив машину M, если нужно, работать после перехода в допускающее состояние, не изменяя его

[6]Напомним, что мы считаем переменную одним символом. Строго говоря, потребуется $O(r^2 \log r)$ и, значит, не более $O(r^3)$ символов.

и не сдвигая головку, и дополнив все мгновенные описания нужным количеством пустых символов. Таким образом, будем строить выражение w_0 как произведение семи выражений A, B, ..., G, которые утверждают, что Q_0, Q_1, ..., Q_q — допускающая последовательность мгновенных описаний, причем каждое мгновенное описание Q_i имеет длину $p(n)$ и $q = p(n)$. Утверждение о том, что Q_0, Q_1, ..., $Q_{p(n)}$ — допускающая последовательность мгновенных описаний, равносильно совокупности следующих утверждений.

1. В каждом мгновенном описании головка сканирует ровно одну ячейку.
2. В каждом мгновенном описании в каждой ячейке ленты стоит только один символ.
3. Каждое мгновенное описание содержит только одно состояние.
4. При переходе от одного мгновенного описания к следующему изменяется содержимое только ячейки, сканируемой головкой.
5. Изменение состояния, положения головки и содержимого ячейки при переходе от мгновенного описания к следующему происходит в соответствии с функцией переходов машины M.
6. Первое мгновенное описание является начальным.
7. Состояние в последнем мгновенном описании является заключительным.

Построим булевы выражения A, ..., G, соответствующие утверждениям 1–7.

1. Выражение A утверждает, что в каждый момент времени машина M сканирует в точности одну ячейку. Пусть выражение A_t утверждает, что в момент t сканируется только одна ячейка. Тогда $A = A_0 A_1 ... A_{p(n)}$, где

$$A_t = U\big(H\langle 1, t\rangle, H\langle 2, t\rangle, ..., H\langle p(n), t\rangle\big).$$

   Заметим, что если раскрыть выражение U, то окажется, что выражение A имеет длину $O(p^3(n))$ и ее можно записать за такое же время.

2. Выражение B утверждает, что каждая ячейка ленты в каждый момент времени содержит только один символ. Пусть выражение B_{it} утверждает, что i-я ячейка ленты в момент времени t содержит ровно один символ. Тогда

$$B = \prod_{i,t} B_{it},$$

   где

$$B_{it} = U\big(C\langle i, 1, t\rangle, C\langle i, 2, t\rangle, ..., C\langle i, m, t\rangle\big).$$

Длина каждого выражения B_{it} не зависит от n, поскольку размер m ленточного алфавита зависит только от машины Тьюринга M. Таким образом, выражение B имеет длину $O(p^2(n))$.

3. Выражение C утверждает, что в каждый момент времени t машина M находится в одном и только одном состоянии:

$$C = \prod_{0 \le i \le p(n)} U\left(S\langle 1, t\rangle, S\langle 2, t\rangle, \ldots, S\langle s, t\rangle\right).$$

Поскольку число состояний s машины M постоянно, то выражение C имеет длину $O(p(n))$.

4. Выражение D утверждает, что в любой момент времени t можно изменить содержимое не более чем одной ячейки:[7]

$$D = \prod_{i,j,t}\left[\left(C\langle i, j, t\rangle \equiv C\langle i, j, t+1\rangle\right) + H\langle i, t\rangle\right].$$

Выражение $\left(C\langle i, j, t\rangle \equiv C\langle i, j, t+1\rangle\right) + H\langle i, t\rangle$ утверждает, что либо

а) в момент t считывающая головка сканирует ячейку i, либо

б) j-й символ записан в ячейке i в момент $t+1$ тогда и только тогда, когда он был записан там в момент t.

Поскольку выражения A и B утверждают, что в момент t считывающая головка сканирует только одну ячейку и ячейка i содержит лишь один символ, в момент времени t либо головка сканирует ячейку i, либо содержимое ячейки i не меняется. Если раскрыть значение знака $\equiv$, то длина D будет равна $O(p^2(n))$.

5. Выражение E утверждает, что очередное мгновенное описание машины M получается из предыдущего за один переход, разрешаемый функцией переходов δ машины M. Пусть E_{ijkt} означает одно из следующих утверждений:

а) в момент t ячейка i не содержит символ j;

б) в момент t считывающая головка не сканирует ячейку i;

в) в момент t машина M не находится в состоянии k;

г) очередное мгновенное описание машины M получается из предыдущего переходом, разрешаемым функцией переходов машины M.

Тогда

$$E = \prod_{i,j,k,t} E_{ijkt},$$

---

[7] $x \equiv y$ означает $xy + \overline{x}\,\overline{y}$ (x тогда и только тогда, когда y).

где

$$E_{ijkt} \equiv \neg C\langle i, j, k\rangle + \neg H\langle i, t\rangle + \neg S\langle k, t\rangle + \\ + \sum_l \left[C\langle i, j_l, i+1\rangle S\langle k_l, t+1\rangle H\langle i_l, t+1\rangle \right].$$

Здесь индекс пробегает по всем шагам машины M, возможным в случае, когда машина M сканирует символ X_j в состоянии q_k. Иначе говоря, для каждой тройки $\langle q, X, d\rangle$ из $\delta(q_k, X_j)$ существует одно значение l, для которого $X_{j_l} = X$, $q_{k_l} = q$ и число i_l равно $i-1$, i или $i+1$, в зависимости от значения d, которое может быть равным L, S или R соответственно. Здесь δ — функция переходов машины M. Поскольку машина M является недетерминированной, таких троек (q, X, d) может быть несколько, но во всяком случае, их число конечно. Таким образом, выражение E_{ijkt} имеет ограниченную длину, не зависящую от n. Следовательно, выражение E имеет длину $O(p^2(n))$.

6. Выражение F утверждает, что выполнены начальные условия:

$$F = S\langle 1, 0\rangle H\langle 1, 0\rangle \prod_{1\le i\le n} C\langle i, j_i, 0\rangle \prod_{n<i\le p(n)} C\langle i, 1, 0\rangle,$$

где выражение $S\langle 1, 0\rangle$ утверждает, что машина M в момент $t = 0$ находится в состоянии q_1, которое мы считаем начальным. Выражение $H\langle 1, 0\rangle$ утверждает, что машина M в момент $t = 0$ сканирует крайнюю левую ячейку ленты. Выражение $\prod_{1\le i\le n} C\langle i, j_i, 0\rangle$ утверждает, что первые n ячеек вначале содержат входную строку w. Выражение $\prod_{n<i\le p(n)} C\langle i, 1, 0\rangle$ утверждает, что остальные ячейки вначале пустые. Мы считаем, что пустым символом является X_1. Очевидно, что выражение F имеет длину $O(p(n))$.

7. Выражение G утверждает, что машина M в конце концов приходит в заключительное состояние. Поскольку мы потребовали, чтобы машина M оставалась в заключительном состоянии после того, как оно достигнуто, то $G = S\langle s, p(n)\rangle$. Мы считаем, что заключительным состоянием является q_s.

Булево выражение w_0 — это произведение $ABCDEFG$. Поскольку каждый из семи сомножителей состоит не более чем из $O(p^3(n))$ символов, само выражение w_0 также состоит не более чем из $O(p^3(n))$ символов. Поскольку мы считали каждую пропозициональную переменную отдельным символом, а в действительности переменные представляются строками длины $O(\log n)$, то на самом деле оценкой длины выражения $|w_0|$ будет $O(p^3(n)\log n)$, которая не превосходит $cnp^3(n)$, где c — некоторая константа. Таким образом, длина строки w_0 полиномиально зависит от длины строки w. Должно быть ясно, что если заданы строка w и полином p, то выражение w_0 можно записать за время, пропорциональное его длине.

По данной допускающей последовательности мгновенное описание $Q_1, Q_2, ..., Q_q$ можно, очевидно, найти такое присваивание значений 0 и 1 пропозициональным переменным $C\langle i, j, t\rangle$, $S\langle k, t\rangle$ и $H\langle i, t\rangle$, что выражение w_0 примет значение **true**. Обратно, если задан набор значений переменных выражения w_0, при котором она становится истинной, то легко найти допускающую последовательность мгновенных описаний. Таким образом, выражение w_0 выполнимо тогда и только тогда, когда машина M допускает строку w.

Мы не накладывали на язык L, допускаемый машиной M, никаких ограничений, кроме того, что он принадлежит классу $\mathcal{NP}$. Следовательно, мы показали, что любой язык из класса $\mathcal{NP}$ является полиномиально преобразуемым в задачу выполнимости булевых формул. Таким образом, задача выполнимости является *NP*-полной. □

На самом деле мы доказали больше, чем утверждается в теореме 10.3. Говорят, что булево выражение находится в *конъюнктивной нормальной форме*, если она представляет собой произведение сумм литералов, где каждый *литерал* имеет вид x или $\neg x$ для некоторой переменной x. Например, $(x_1 + x_2)(x_2 + \bar{x}_1 + x_3)$ находится в конъюнктивной нормальной форме, а $x_1x_2 + x_3$ — нет. Выражение w_0, построенное в доказательстве теоремы 10.3, практически находится в конъюнктивной нормальной форме, поскольку его можно представить в конъюнктивной нормальной форме, увеличив длину не более чем в постоянное число раз.

Следствие. Задача выполнимости булевых выражений, находящихся в конъюнктивной нормальной форме, является *NP*-полной.

Доказательство. Достаточно показать, что каждое из выражений A, ..., G, построенных в доказательстве теоремы 10.3, либо уже находится в конъюнктивной нормальной форме, либо может быть преобразовано в него с помощью правил булевой алгебры, причем ее длина увеличится не более чем в постоянное число раз. Выражение U, определенное равенством (10.1), уже находится в конъюнктивной нормальной форме. Отсюда следует, что выражения A, B и C тоже находятся в конъюнктивной нормальной форме. Тот факт, что выражения F и G находятся в конъюнктивной нормальной форме, является тривиальным, поскольку F и G — произведения литералов.

Выражение D представляет собой выражение вида $(x \equiv y) + z$. Если раскрыть знак $\equiv$, получится выражение

$$xy + \bar{x}\bar{y} + z, \tag{10.2}$$

эквивалентное выражению

$$(x + \bar{y} + z)(\bar{x} + y + z). \tag{10.3}$$

Подставив (10.3) вместо (10.2) в выражение D, получим выражение, эквивалентное исходному, но находящееся в конъюнктивной нормальной форме и превосходящее исходное выражение по длине не более чем в два раза.

Наконец, E — произведение выражений E_{ijkl}. Поскольку длины всех выражений E_{ijkl} ограничены независимо от n, каждое выражение E_{ijkl} можно преобразовать в выражение, находящееся в конъюнктивной нормальной форме, причем его длина не будет зависеть от n. Следовательно, преобразование выражения E в выражение, находящееся в конъюнктивной нормальной форме, увеличивает его длину не более чем в постоянное число раз.

Итак, выражение w_0 можно представить в конъюнктивной нормальной форме, увеличив его длину не более чем в постоянное число раз. □

Мы только что показали, что задача выполнимости формул в конъюнктивной нормальной форме является *NP*-полной. Можно показать, что даже при более жестких ограничениях на вид выражения задача выполнимости булевых выражений также является *NP*-полной. Говорят, что выражение находится в *k-конъюнктивной нормальной форме*, если оно представляет собой произведение сумм, состоящих не более чем из k литералов. *Задача k-выполнимости* состоит в выяснении выполнимости выражений, находящихся в k-конъюнктивной нормальной форме. Для $k = 1$ и 2 известны детерминированные алгоритмы полиномиальной сложности, проверяющие k-выполнимость. Ситуация (по-видимому) изменяется при $k = 3$, как видно из следующей теоремы.

Теорема 10.4. Задача 3-выполнимости является *NP*-полной.

Доказательство. Покажем, что выполнимость формул в конъюнктивной нормальной форме полиномиально преобразуется в 3-выполнимость. В данном произведении сумм заменим каждую сумму $(x_1 + x_2 + ... + x_k)$, $k \geq 4$, на

$$(x_1 + x_2 + y_1)(x_3 + \overline{y}_1 + y_2)(x_4 + \overline{y}_2 + y_3)...(x_{k-2} + \overline{y}_{k-4} + y_{k-3})(x_{k-1} + x_k + \overline{y}_{k-3}), \quad (10.4)$$

где y_1, y_2, ..., y_{k-3} — новые переменные. Например, для $k = 4$ выражение (10.4) принимает вид $(x_1 + x_2 + y_1)(x_3 + x_4 + \overline{y}_1)$. Присвоить значения новым переменным так, чтобы подставляемое выражение приняло значение 1, можно тогда и только тогда, когда один из литералов x_1, x_2, ..., x_k имеет значение 1, т.е. тогда и только тогда, когда исходное выражение принимает значение 1. Допустим, что $x_i = 1$. Тогда положим $y_j = 1$ для $j \leq i - 2$ и $y_j = 0$ для $j > i - 2$. Подставляемое выражение принимает значение 1. Обратно, допустим, что при некоторых значениях переменных y_i подставляемое выражение принимает значение 1. Если $y_1 = 0$, то одна из переменных x_1 и x_2 должна иметь значение 1. Если $y_{k-3} = 1$, то одна из переменных x_{k-1} и x_k должна иметь значение 1. Если $y_1 = 1$ и $y_{k-3} = 0$, то для некоторого i, $1 \leq i \leq k - 4$,

выполнены оба равенства, $y_i = 1$ и $y_{i+1} = 0$, откуда следует, что переменная x_{i+2} должна иметь значение 1. В любом случае некоторая переменная x_i должна иметь значение 1.

Длина выражения, получаемого в результате описанной выше замены, превосходит длину исходного выражения не более чем в постоянное число раз. Таким образом, по данной формуле E в конъюнктивной нормальной форме можно найти, применяя описанное выше преобразование к каждой сумме, выражение E' в 3-конъюнктивной нормальной форме, выполнимое тогда и только тогда, когда является выполнимым исходное выражение. Более того, легко показать, что выражение E' можно найти за время, пропорциональное длине выражения E, выполняя преобразования очевидным образом. □

10.5. Еще несколько *NP*-полных задач

Теперь приступим к доказательству того, что каждая из задач, упомянутых в теореме 10.2, является *NP*-полной, прямо или косвенно сводя выполнимость к этим задачам. Последовательность преобразований, которые мы в действительности будем применять, показана в виде дерева на рис. 10.6. Если на рис. 10.6 вершина P — отец вершины P', то мы покажем, что задача P полиномиально преобразуется в задачу P'.

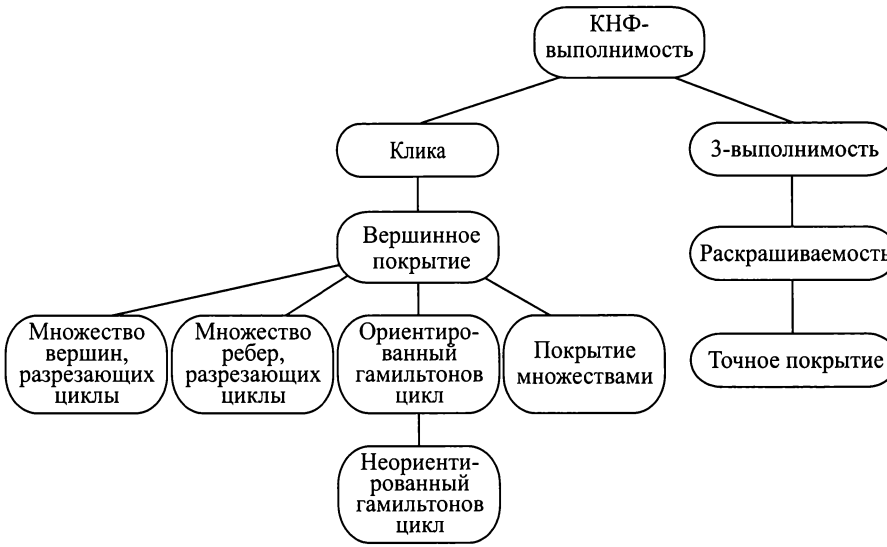


Рис. 10.6. Последовательность преобразований для различных трудных задач

Теорема 10.5. Задача КНФ-выполнимости полиномиально преобразуется в задачу о клике. Следовательно, задача о клике является *NP*-полной.

Доказательство. Пусть $F = F_1F_2...F_q$ — выражение в конъюнктивной нормальной форме, где F_i — сомножители; каждое выражение F_i имеет вид $\left(x_{i1} + x_{i2} + ... + x_{ik_i}\right)$, где x_{ij} — литерал. Построим неориентированный i граф $G = (V, E)$, вершинами которого служат пары целых чисел $[i, j]$ для $1 \le i \le q$ и $1 \le j \le k_i$. Первый компонент такой пары представляет сомножитель, а второй — литерал, входящий в него. Таким образом, каждая вершина графа естественным образом соответствует конкретному вхождению в конкретный сомножитель.

Ребрами графа G служат пары $([i, j], [k, l])$, для которых $i \ne k$ и $x_{ij} \ne \overline{x}_{kl}$. Неформально вершины $[i, j]$ и $[k, l]$ являются смежными в графе G, если они соответствуют различным сомножителям и можно так присвоить значения переменным из литералов x_{ij} и x_{kl}, чтобы оба литерала приняли значения 1 (присваивая значение 1 выражениям F_i и F_k). Иначе говоря, либо $x_{ij} = x_{kl}$, либо переменные, входящие в литералы x_{ij} и x_{kl}, различны.

Количество вершин в графе G, очевидно, меньше длины выражения F, а количество ребер не превосходит квадрата числа вершин. Следовательно, граф G можно закодировать в виде строки, длина которой ограничена полиномом, зависящим от длины выражения F. Более того, такой код можно найти за время, ограниченное полиномом от длины выражения F. Мы покажем, что граф G содержит клику размера q тогда и только тогда, когда выражение F является выполнимым. Отсюда будет следовать, что по данному алгоритму, решающему задачу о клике за полиномиально ограниченное время, можно построить алгоритм с полиномиально ограниченным временем работы для задачи КНФ-выполнимости, построив по выражению F граф G, содержащий клику размера q тогда и только тогда, когда выражение F является выполнимым. Итак, покажем, что, для того, чтобы граф G содержал клику размера q, необходимо и достаточно, чтобы выражение F было выполнимым.

Достаточность. Пусть выражение F является выполнимым. Тогда существует набор значений переменных, состоящий из нулей и единиц, при котором $F = 1$. При этом наборе каждый сомножитель выражения F принимает значение 1. Каждый сомножитель F_i содержит хотя бы один литерал, принимающий значение 1. Пусть таким литералом в выражении F_i будет x_{im_i}.

Мы утверждаем, что множество вершин $\{[i, m_i] \mid 1 \le i \le q\}$ образует клику размера q. Если бы это было не так, то нашлись бы такие i и j, что $i \ne j$ и вершины $[i, m_i]$ и $[j, m_j]$ не соединены ребром. Отсюда следовало бы, что $x_{im_i} = \overline{x}_{jm_j}$

(по определению множества ребер графа G). Однако это невозможно, поскольку $x_{im_i} = x_{jm_j} = 1$ вследствие выбора переменных x_{im_i}.

Необходимость. Допустим, что граф G содержит клику размера q. Первые компоненты вершин, составляющих такую клику, должны быть разными, поскольку вершины с одинаковыми первыми компонентами не соединяются ребрами. Поскольку в этой клике ровно q вершин, то вершины клики взаимно однозначно соответствуют сомножителям выражения F. Пусть вершины клики имеют вид $[i, m_i]$, $1 \le i \le q$. Пусть $S_1 = \{y \mid x_{im_i} = y$, где $1 \le i \le q$ и y — переменная$\}$ и $S_2 = \{y \mid x_{im_i} = \overline{y}$, где $1 \le i \le q$ и y — переменная$\}$. Иначе говоря, S_1 и S_2 — множества переменных и отрицаний переменных соответственно, представленных вершинами клики. Тогда $S_1 \cap S_2 = \varnothing$, поскольку в противном случае какие-то вершины $[s, m_s]$ и $[t, m_t]$, для которых $x_{sm_s} = \overline{x}_{tm_t}$ соединялись бы ребром. Если положить переменные из множества S_1 равными 1, а из множества S_2 — равными 0, то каждое выражение F_i примет значение 1. Следовательно, выражение F является выполнимым. □

Пример 10.5. Рассмотрим выражение $F = (y_1 + \overline{y}_2)(y_2 + \overline{y}_3)(y_3 + \overline{y}_1)$. Литералы таковы:

$$x_{11} = y_1, \quad x_{21} = y_2, \quad x_{31} = y_3,$$
$$x_{12} = \overline{y}_2, \quad x_{22} = \overline{y}_3, \quad x_{32} = \overline{y}_1.$$

Конструкция из теоремы 10.5 порождает граф, изображенный на рис. 10.7. Например, вершина [1, 1] не соединена с вершиной [1, 2], потому что их первые компоненты одинаковы, и с вершиной [3, 2], потому что $x_{11} = y_1$ и $x_{32} = \overline{y}_1$. В то же время, вершина [1, 1] соединена с остальными тремя вершинами.

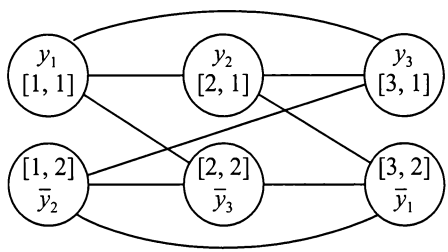


Рис. 10.7. Граф, построенный по теореме 10.5

В выражение F входят три сомножителя, а в графе, показанном на рис. 10.7, содержатся две клики размера 3, а именно: {[1, 1], [2, 1], [3, 1]} и {[1, 2], [2, 2], [3, 2]}. В первой клике представлены три литерала — y_1, y_2 и y_3. Это переменные без отрицаний, и первая клика соответствует присвоению значений $y_1 = y_2 = y_3 = 1$

в выражении F. Вторая клика соответствует другому набору значений, обращающему выражение F в истинное, а именно $y_1 = y_2 = y_3 = 0$. □

Теорема 10.6. Задача о клике полиномиально преобразуется в задачу о вершинном покрытии. Следовательно, задача о вершинном покрытии является *NP*-полной.

Доказательство. Рассмотрим неориентированный граф $G = (V, E)$ и его дополнение $\overline{G} = (V, \overline{E})$, где $\overline{E} = \{(v, w) | v, w \in V, v \neq w \text{ и } (v, w) \notin E\}$. Мы утверждаем, что множество $S \subseteq V$ является кликой в графе G тогда и только тогда, когда $V - S$ является вершинным покрытием графа $\overline{G}$. Действительно, если S — клика в графе G, то никакое ребро в графе $\overline{G}$ не соединяет никакие две вершины в клике S. Следовательно, любое ребро в графе $\overline{G}$ инцидентно хотя бы одной вершине из $V - S$, откуда следует, что $V - S$ — вершинное покрытие графа $\overline{G}$. Аналогично, если $V - S$ — вершинное покрытие графа $\overline{G}$, то каждое ребро из графа $\overline{G}$ инцидентно хотя бы одной вершине из $V - S$. Следовательно, никакое ребро из графа $\overline{G}$ не соединяет два вершины из множества S. Таким образом, каждая пара вершин из множества S соединена в графе G, и, значит, S — клика в графе G.

Чтобы узнать, существует ли клика размера k, построим граф $\overline{G}$ и выясним, содержит ли она вершинное покрытие размера $||V|| - k$. Разумеется, по заданному стандартному представлению графа $G = (V, E)$ и числам k можно найти представление графа $\overline{G}$ и числа $||V|| - k$ за время, полиномиально зависящее от длины представления графа G и k. □

Пример 10.6. Граф G на рис. 10.8, *а*, содержит две клики размера 3: {1, 2, 5} и {1, 4, 5}. Граф $\overline{G}$ на рис. 10.8, *б*, содержит соответствующие вершинные покрытия {3, 4} и {2, 3} размера 2. Граф G содержит (среди других) клики размера 2: {2, 3} и {3, 4}, а граф $\overline{G}$ — соответствующие вершинные покрытия {1, 4, 5} и {1, 2, 5} размера 3. □

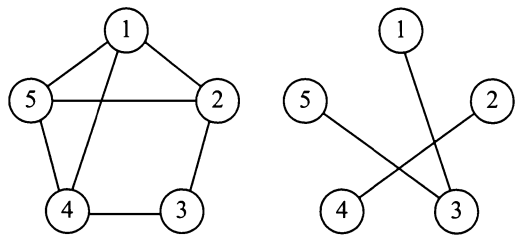


Рис. 10.8. Граф G (*а*) и его дополнение $\overline{G}$ (*б*)

Теорема 10.7. Задача о вершинном покрытии полиномально преобразуется в задачу о множестве узлов, разрезающих циклы. Поэтому задача о множестве узлов, разрезающих циклы, является *NP*-полной.

Доказательство. Пусть $G = (V, E)$ — неориентированный граф, а D — ориентированный граф, полученный заменой каждого ребра графа G двумя ориентированными ребрами. Точнее, пусть $D = (V, E')$, где $E' = \{(v, w), (w, v) | (v, w) \in E\}$. Поскольку каждое ребро в E заменено циклом графа D, то множество $S \subseteq V$ (т.е. каждый цикл графа D содержит вершину из S) тогда и только тогда, когда S — вершинное покрытие графа G. Кроме того, представление графа D легко найти по графу G за полиномиальное время. □

Теорема 10.8. Задача о вершинном покрытии полиномиально преобразуется в задачу о множестве ребер, разрезающих циклы. Следовательно, задача о множестве ребер, разрезающих циклы, является *NP*-полной.

Доказательство. Пусть $G = (V, E)$ — неориентированный граф, а $D = (V \times \{0, 1\}, E')$ — ориентированный граф, где E' состоит из следующих ребер (рис. 10.9):

1. $[v, 0] \to [v, 1]$[8] для каждого $v \in V$;
2. $[v, 1] \to [w, 0]$ и $[w, 1] \to [v, 0]$ для каждого неориентированного ребра $(v, w) \in E$.

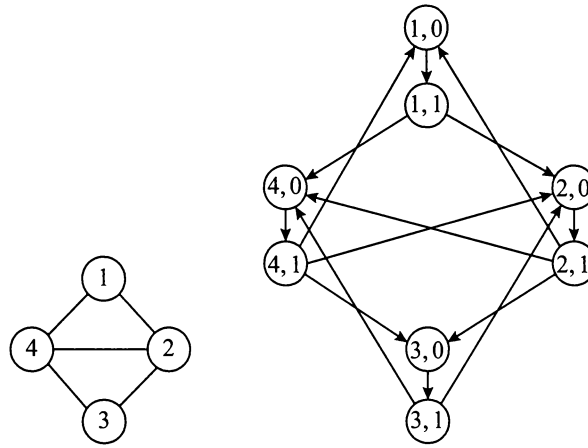
Пусть $F \subseteq E'$ — множество ребер графа D, содержащих хотя бы одно ребро из каждого цикла в графе D. Заменим каждое ребро из множества F, имеющее вид $[v, 1] \to [w, 0]$, ребром $[w, 0] \to [w, 1]$. Полученное множество обозначим через F'. Мы утверждаем, что $\|F'\| \leq \|F\|$ и F' содержит хотя бы одно ребро из каждого цикла. (Единственное ребро, выходящее из вершины $[w, 0]$, идет в вершину $[w, 1]$, так что $[w, 0] \to [w, 1]$ принадлежит любому циклу, содержащему $[v, 1] \to [w, 0]$.)

Не умаляя общности, будем считать, что

$$F' = \{[v_i, 0] \to [v_i, 1] | 1 \leq i \leq k\}$$

для некоторого k. Тогда каждый цикл в графе D содержит ребро $[v_i, 0] \to [v_i, 1]$ для некоторого i, $1 \leq i \leq k$. Однако заметим, что если (x, y) — произвольное ребро в графе G, то $[x, 1], [y, 0], [y, 1], [x, 0], [x, 1]$ образуют цикл в графе D. Следовательно, каждое ребро в графе G инцидентно некоторой вершине v_i, $1 \leq i \leq k$, и, значит, $\{v_1, ..., v_k\}$ — вершинное покрытие графа G.

[8] Здесь и далее в этой главе ориентированное ребро (x, y) обозначается через $x \to y$. Это соглашение облегчает чтение.



а) б)

Рис. 10.9. Неориентированный граф G (*а*) и соответствующий ориентированный граф D (*б*). Вершинное покрытие $\{2, 4\}$ соответствует множеству $\{[2, 0] \to [2, 1], [4, 0] \to [4, 1]\}$ ребер, разрезающих циклы

И наоборот, легко показать, что если S — вершинное покрытие размера k, то $\{[v, 0) \to [v, 1] \mid v \in S\}$ — множество ребер, разрезающих циклы в графе D. Чтобы узнать, содержит ли граф G вершинное покрытие размера k, необходимо построить за полиномиальное время граф D и выяснить, есть ли в нем k-элементное множество ребер, разрезающих циклы. □

Теорема 10.9. Задача о вершинном покрытии полиномиально преобразуется в задачу о гамильтоновом цикле для ориентированных графов. Следовательно, задача о гамильтоновом цикле в ориентированном графе является *NP*-полной.

Доказательство. Пусть дан неориентированный граф $G = (V, E)$ и целое число k. Покажем, как за время, полиномиальное по $||V||$. построить ориентированный граф $G_D = (V_D, E_D)$, содержащий гамильтонов цикл тогда и только тогда, когда некоторое k-элементное подмножество множества V покрывает ребра графа G.

Пусть $V = \{v_1, v_2, ..., v_n\}$. Обозначим ребро (v_i, v_j) через e_{ij}.[9] Пусть $a_1, a_2, ..., a_k$ — новые символы. Множество V_D будет включать в себя по одной вершине для каждого a_i и еще по четыре вершины для каждого ребра графа G. Точнее,

[9] Заметим, что e_{ij} и e_{ji} обозначают одно и то же ребро, и их следует рассматривать как один и тот же символ.

$V_D = \{a_1, a_2, ..., a_k\} \cup \{[v, e, b] | v \in V, e \in E, b \in \{0, 1\}$ и ребро e инцидентно $v\}$.

Прежде чем формально описывать ребра графа G_D, дадим интуитивное объяснение. Ребро (v_i, v_j) графа G представляется подграфом с четырьмя вершинами (рис. 10.10).

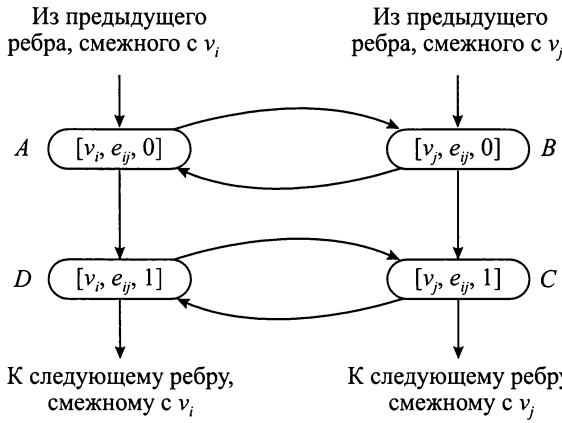


Рис. 10.10. Представление ребра (v_i, v_j)

Если гамильтонов цикл входит в подграф, представляющий ребро e_{ij}, в вершине A, то он должен выйти из него в вершине D, поскольку если он выйдет в вершине C, то вершины $[v_j, e_{ij}, 0]$ или $[v_i, e_{ij}, 1]$ не смогут оказаться в этом цикле. Идя из вершины A в вершину D, цикл может посетить либо все четыре вершины подграфа, либо только две самые левые. В последнем случае гамильтонов цикл должен в какой-то момент пройти из вершины B в вершину C, посетив две самые правые вершины.

Граф G_D можно представить состоящим из $||V||$ списков, по одному списку для каждой вершины. Список для вершины v содержит все ребра, инцидентные вершине v и расположенные в специальном порядке. Из каждого узла a_j, $1 \le j \le k$, идут ребра к таким вершинам $[v_i, e_{il}, 0]$, что e_{il} — первое ребро в списке для v_i. Далее, в каждую вершину a_j идут ребра из $[v_i, e_{im}, 1]$, если e_{il} —последнее ребро в списке для v_i. Из вершины $[v_i, e_{il}, 1]$ идет ребро в вершину $[v_i, e_{im}, 0]$, если ребро e_{im} непосредственно следует за e_{il} в списке для v_i. Эти ребра вместе с ребрами, необходимыми в силу рис. 10.10, образуют множество E_D. Покажем, что, для того, чтобы в графе G_D был гамильтонов цикл, необходимо и достаточно, чтобы в графе G существовало множество из k вершин, покрывающее все ребра.

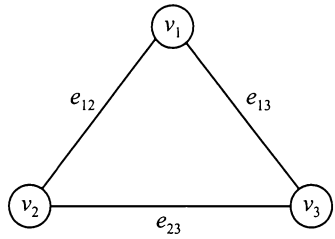
Достаточность. Допустим, вершины $v_1, v_2, ..., v_k$ покрывают все ребра графа G. Пусть $f_{i1}, f_{i2}, ..., f_{il}$ — список ребер, инцидентных v_i, $1 \le i \le k$. Рассмотрим цикл, идущий из вершины a_1 в $[v_1, f_{11}, 0]$, $[v_1, f_{11}, 1]$, $[v_1, f_{12}, 0]$, $[v_1, f_{12}, 1]$, ..., $\left[v_1, f_{1l_1}, 0\right]$, $\left[v_1, f_{1l_1}, 1\right]$ и далее в a_2, затем в $[v_2, f_{21}, 0]$, $[v_2, f_{21}, 1]$, ... и т.д. по ребрам списков для v_3, v_4, ..., v_k и, наконец, обратно в a_1. Этот цикл проходит через каждую вершину графа G_D, кроме вершин, содержащихся в списках ребер для вершин, не совпадающих с $v_1, ..., v_k$. Для каждой пары вершин графа G_D, имеющей вид $[v_j, e, 0]$, $[v_j, e, 1]$, $j > k$, можно расширить этот цикл, поскольку ребро e инцидентно некоторой вершине v_i, $1 \le i \le k$. Заменим ребро, идущее из вершины $[v_i, e, 0]$ в вершину $[v_i, e, 1]$ в этом цикле, путем

$$[v_i, e, 0], [v_j, e, 0], [v_j, e, 1], [v_i, e, 1].$$

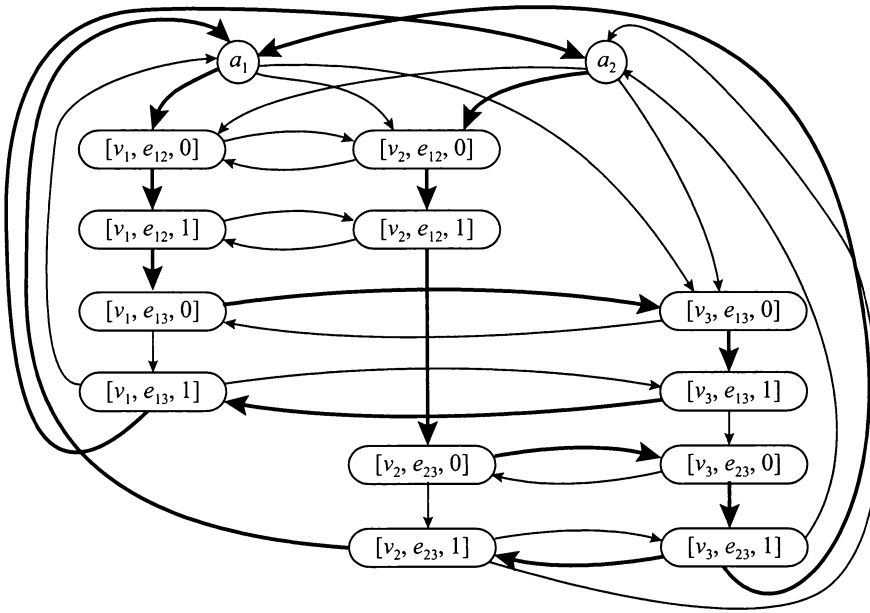
Цикл, исправленный описанным выше способом для каждой вершины v_j и ребра e, содержит каждую вершину графа G_D и, значит, является гамильтоновым.

Необходимость. Допустим, в графе G_D существует гамильтонов цикл. Этот цикл можно разделить на k путей, каждый из которых начинается в некоторой вершине a_i и завершается в какой-то вершине a_j. Из наших предыдущих рассмотрений возможных путей через подграф с четырьмя вершинами, представляющий какое-то ребро (как на рис. 10.10), следует, что существует такая вершина v, что каждая вершина на пути из a_i в a_j имеет вид либо $[v, e, 0]$ или $[v, e, 1]$, где e — ребро графа G, или $[w, e, 0]$ или $[w, e, 1]$, где e — ребро (v, w). Следовательно, первый компонент каждой вершины, лежащей на пути из a_i в a_j, представляет собой либо v, либо узел, смежный с v. Таким образом, пути из a_i в a_j можно поставить в соответствие некоторой вершине v. Для каждой вершины $[w, e, b]$ графа G_D ребро e должно быть инцидентно одной из k вершин графа G, поставленных в соответствие путям. Следовательно, эти k вершин образуют вершинное покрытие графа G. Отсюда заключаем, что граф G_D содержит гамильтонов цикл тогда и только тогда, когда в графе G существуют такие k вершин, что каждое ребро из графа G инцидентно одной из этих k вершин. □

Пример 10.7. Пусть G — граф с вершинами v_1, v_2, v_3 и ребрами e_{12}, e_{13}, e_{23} (рис. 10.11, *а*). Граф G_D, построенный в теореме 10.9, изображен на рис. 10.11, *б*. Гамильтонов цикл в графе G_D, основанный на вершинном покрытии $\{v_1, v_2\}$, указан жирными линиями. □



а)



б)

Рис. 10.11. Граф с гамильтоновым циклом: неориентированный граф G (*а*), ориентированный граф G_D (*б*)

Теорема 10.10. Задача о гамильтоновом цикле для ориентированных графов полиномиально преобразуется в задачу о гамильтоновом цикле для неориентированных графов. Следовательно, задача существования гамильтонова цикла в неориентированном графе является *NP*-полной.

Доказательство. Пусть $G = (V, E)$ — ориентированный граф, а $U = (V \times \{0, 1, 2\}, E')$ — неориентированный граф, где E' состоит из ребер

1. $([v, 0], [v, 1])$ для $v \in V$.
2. $([v, 1], [v, 2])$ для $v \in V$.
3. $([v, 2], [w, 0])$ тогда и только тогда, когда ребро $v \longrightarrow w$ принадлежит E.

Отметим, что каждой вершине из множества V можно поставить в соответствие путь, состоящий из трех вершин. Поскольку гамильтонов цикл в графе U должен включать в себя все вершины, последний компонент вершин, составляющих этот цикл, должен изменяться в порядке 0, 1, 2, 0, 1, 2, ... или обратном к нему. Будем считать, что реализуется первый из этих двух случаев; тогда ребра типа 3, у которых второй компонент меняется с 2 на 0, образуют ориентированный гамильтонов цикл в графе G. И наоборот, гамильтонов цикл в графе G можно превратить в неориентированный гамильтонов цикл в графе U, заменив каждое ребро v–w путем $[v, 0]$, $[v, 1]$, $[v, 2]$, $[w, 0]$ в графе U. □

Теорема 10.11. Задача о вершинном покрытии полиномиально преобразуется в задачу о покрытии множествами. Следовательно, задача о покрытии множествами является *NP*-полной.

Доказательство. Пусть $G = (V, E)$ — неориентированный граф. Для $1 \le i \le \|V\|$ обозначим через S_i множество всех ребер, инцидентных вершине v_i. Очевидно, что $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ образуют покрытие множествами для S_i тогда и только тогда, когда $\{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ — вершинное покрытие графа G. □

Теорема 10.12. Задача 3-выполнимости полиномиально преобразуется в задачу раскрашиваемости.

Доказательство. Пусть задано выражение F в виде 3-КНФ с n переменными и t сомножителями. Покажем, как построить за время, ограниченное полиномом от $\mathrm{MAX}(n, t)$, неориентированный граф $G = (V, E)$ с $3n + t$ вершинами, который можно раскрасить в $n + 1$ цветов тогда и только тогда, когда выражение F является выполнимым.

Пусть $x_1, x_2, \ldots, x_n$ и $F_1, F_2, \ldots, F_t$ — соответственно переменные и сомножители выражения F. Пусть $v_1, v_2, \ldots, v_n$ — новые символы. Без потери общности будем считать, что $n \ge 4$, поскольку любое выражение в виде 3-КНФ, число различных переменных которого не превосходит 3, можно проверить на выполнимость за время, линейно зависящее от его длины, не прибегая к раскрашиванию. Вершины графа G имеют следующий вид.

1. x_i, $\overline{x}_i$, v_i для $1 \le i \le n$.
2. F_i для $1 \le i \le t$.

Ребра графа G задаются следующим образом.

1. Все (v_i, v_j), для которых $i \ne j$.
2. Все (v_i, x_j) и $(v_i, \overline{x}_j)$, для которых $i \ne j$.
3. $(x_i, \overline{x}_i)$, для $1 \le i \le n$.
4. (x_i, F_j), если x_i не входит в F_j, и $(\overline{x}_i, F_j)$, если $\overline{x}_i$ не входит в F_j.

Узлы v_1, v_2, ..., v_n образуют полный подграф с n вершинами, так что для их раскраски требуется n различных цветов. Каждая из вершин x_j и $\overline{x}_j$ соединена с каждой из вершин v_i, $i \ne j$, и, значит, вершины x_j и $\overline{x}_j$ не могут иметь одинаковый цвет с вершиной v_i, если $i \ne j$. Поскольку вершины x_j и $\overline{x}_j$ являются смежными, то они не могут быть одинакового цвета, поэтому граф G можно раскрасить в $n + 1$ цветов только тогда, когда одна из вершин x_j и $\overline{x}_j$ имеет тот же цвет, что и v_j, а другая имеет новый цвет, который мы назовем *специальным*.

Допустим, что одной из вершин x_j и $\overline{x}_j$, окрашенной в специальный цвет, приписано значение 0. Теперь рассмотрим цвет, приписанный вершинам F_j. Вершина F_j является смеженной по крайней мере с $2n - 3$ из $2n$ вершин $x_1, ..., x_n, \overline{x}_1, ..., \overline{x}_n$. Поскольку мы предположили, что $n \ge 4$, то для каждого j найдется такое i, что вершина F_j является смежной как с вершиной x_i, так и с вершиной $\overline{x}_i$. Поскольку одна из вершин x_i или $\overline{x}_i$ окрашена в специальный цвет, вершина F_j не может быть окрашена в специальный цвет.

Если выражение F_j содержит такой литерал y, что вершине y приписан специальный цвет, то вершина F_j не является смежной ни с одной вершиной, окрашенной так же, как вершина y, и, значит, ей можно приписать тот же цвет, что и у вершины y. В противном случае нужен новый цвет. Следовательно, все вершины F_i можно раскрасить без дополнительных цветов тогда и только тогда, когда литералам можно так присвоить специальный цвет, чтобы каждый сомножитель содержал такой литерал y, что литералу $\overline{y}$ приписан специальный цвет, т.е. тогда и только тогда, когда переменным можно так присвоить значения, чтобы в каждом сомножителе оказался литерал y со значением 1 (литерал $\overline{y}$ в этом случае равен 0), т.е. тогда и только тогда, когда выражение F является выполнимым. □

Пример 10.8. Пусть $F = (x_1 + x_2)(\overline{x}_1 + x_3)$. Заметим, что здесь в каждом сомножителе по два, а не по три члена. Это изменение не влияет на конструкцию графа G в доказательстве теоремы 10.12, но позволяет рассматривать выражения, содержащие только три переменные, и графы, раскрашиваемые в четыре цвета. (Заметим, что аналог теоремы 10.2 для случая 2-КНФ также верен, но неинтере-

сен. Поскольку существует алгоритм для 2КНФ с полиномиально ограниченным временем работы, то выполнимость для 2-КНФ преобразуется за полиномиальное время в любую задачу.) Результирующий граф показан на рис. 10.12. В качестве цветов взяты A, B, C и S (специальный). Эта 4-раскраска соответствует решению $x_1 = \bar{x}_2 = x_3 = 0$. □

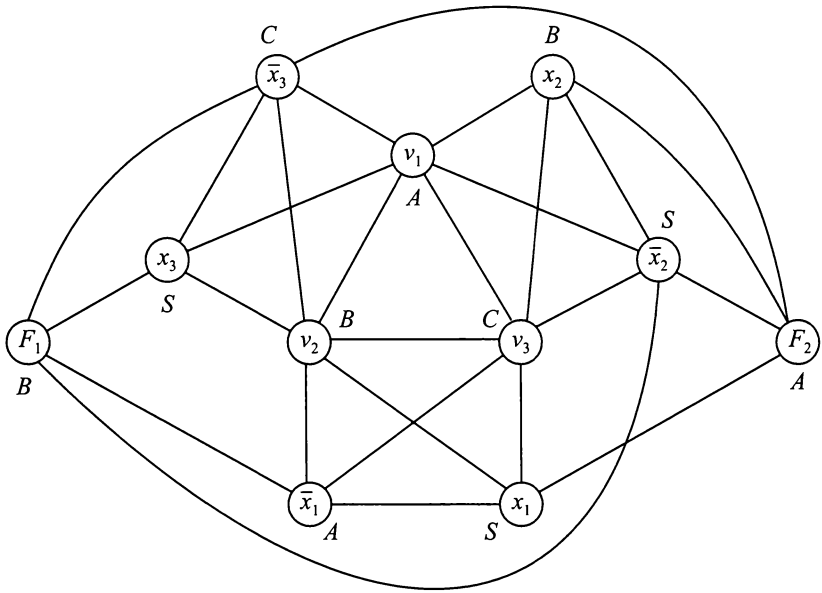


Рис. 10.12. Граф с хроматическим числом 4

Теорема 10.13. Задача раскрашиваемости полиномиально преобразуется в задачу о точном покрытии. Следовательно, задача о точном покрытии является *NP*-полной.

Доказательство. Пусть $G = (V, E)$ — неориентированный граф и k — целое число. Будем строить множества, элементы которых выбираются из множества

$$S = V \cup \{[e, i] | e \in E \text{ и } 1 \le i \le k\}.$$

Для каждой вершины $v \in V$ и $1 \le i \le k$ зададим множества

$$S_{vi} = \{v\} \cup \{[e, i] \mid \text{ребро } e \text{ инцидентно } v\}. \tag{10.5}$$

Для каждого ребра $e \in E$ и $1 \le i \le k$ зададим множества

$$T_{ei} = \{[e, i]\}. \tag{10.6}$$

Установим соответствие между k-раскрасками графа G и точными покрытиями набора множеств, определенных равенствами (10.5) и (10.6). Предположим, граф G имеет k-раскраску, в которой вершине приписан цвет c_v, где c_v — целое число от 1 до k. Тогда легко проверить, что набор множеств, состоящий из множеств S_{vc_v} для каждого v тех одноэлементных множеств T_{ei} для которых $[e,i] \notin S_{vc_v}$ при всех v образует точное покрытие. Для доказательства заметим, что если $e = (v, w)$ — ребро, то $c_v \neq c_w$, поэтому $S_{xc_x} \cap S_{yc_y} = \varnothing$. Ясно, что если x и y не являются смежными вершинами, то S_{xc_x} и S_{yc_y} не пересекаются, а множество T_{ei} выбирается, только если оно не пересекается ни с одним из выбранных множеств.

И наоборот, допустим, что набор множеств, определенных в (10.5) и (10.6), имеет точное покрытие Σ. Тогда для каждого v найдется такое единственное число c_v, что $S_{vc_v} \in \Sigma$. Это вытекает из того, что каждая вершина должна принадлежать точно одному множеству из Σ. Мы утверждаем, что для получения k-раскраски графа G можно раскрасить каждую вершину в цвет c_v. Допустим, что это не так. Тогда существует такое ребро $e = (v, w)$, что $c_v = c_w = c$. Тогда каждое из множеств S_{vc_v} и S_{wc_w} содержит $[e, c]$ и потому Σ не является точным покрытием, что противоречит предположению. □

10.6. Задачи с полиномиально ограниченной памятью

Класс $\mathcal{NP}$ содержится в другом естественном классе трудных задач. Этот класс, обозначаемый $\mathcal{P}$-SPACE, образован языками, допускаемыми детерминированными машинами Тьюринга с полиномиально ограниченной памятью.

Вспомним, что по теореме 10.1, если язык L допускается некоторой недетерминированной машиной Тьюринга с пространственной сложностью $P(n)$, где p — полином, то он допускается некоторой детерминированной машиной Тьюринга с пространственной сложностью $p^2(n)$. Следовательно, класс $\mathcal{P}$-SPACE включает в себя также все языки, допускаемые недетерминированной машиной Тьюринга с полиномиально ограниченной пространственной сложностью. Поскольку каждая недетерминированная машина Тьюринга с полиномиальной временной сложностью имеет полиномиально ограниченную пространственную сложность, выполняется включение $\mathcal{NP}$-TIME $\subseteq$ $\mathcal{P}$-SPACE (рис. 10.13). Кроме того, поскольку $\mathcal{P}$TIME $\subseteq$ $\mathcal{NP}$-TIME $\subseteq$ $\mathcal{P}$-SPACE, если $\mathcal{P}$-TIME = $\mathcal{NP}$-SPACE, то $\mathcal{P}$TIME = $\mathcal{NP}$-TIME, а потому распознавание детерминированным алгоритмом с полиномиально ограниченным временем работы любого языка из класса $\mathcal{P}$SPACE еще менее вероятно, чем распознавание таким алгоритмом любого языка из класса $\mathcal{NP}$-TIME.

Можно указать довольно естественный язык L, *полный для класса* $\mathcal{P}$-SPACE, т.е. язык, который принадлежит классу $\mathcal{P}$-SPACE и удовлетворяет следующему условию: если он допускается какой-то детерминированной машиной Тьюринга

с временной сложностью, ограниченной функцией $T(n)$, то для каждого языка L из $\mathcal{P}$-SPACE найдется детерминированная машина Тьюринга с временной сложностью, ограниченной функцией $T(p_L(n))$, где p_L — полином, зависящий от L. (Здесь слово "детерминированная" можно заменить на "недетерминированная".) Следовательно, если $L_r \in \mathcal{P}$-TIME, то $\mathcal{P}$TIME = $\mathcal{NP}$TIME = $\mathcal{P}$-SPACE. Если $L_r \in \mathcal{NP}$-TIME, то $\mathcal{NP}$-TIME = $\mathcal{P}$SPACE. Язык L_r — это множество регулярных выражений, дополнения которых представляют непустые множества.

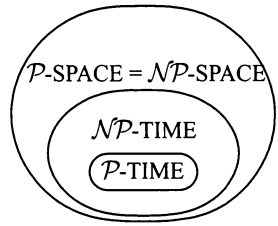


Рис. 10.13. Взаимоствязь пространства и времени

Лемма 10.2. Пусть $M = (Q, T, I, \delta, b, q_0, q_f)$ — одноленточная детерминированная машина Тьюринга с пространственной сложностью, ограниченной полиномом p. Тогда найдется такой полином p', что если $x \in I^*$ и $|x| = n$, то за время $p'(n)$ можно построить регулярное выражение R_x, дополнение которого представляет пустое множество тогда и только тогда, когда машина M не допускает x.

Доказательство. Очень похоже на доказательство теоремы 10.3. Там для описания вычислений машины Тьюринга использовались булевы функции. Здесь мы используем язык регулярных выражений. Регулярные выражения можно записывать в компактной форме, и с их помощью удается выразить факты о последовательностях мгновенных описаний, значительно более длинных, чем сами регулярные выражения.

Регулярное выражение R_x можно представлять себе как способ описания последовательностей мгновенных описаний машины M, которые не допускают строку x. Для наших целей понадобится алфавит $\Delta = T \cup \{[qX] | q \in Q \text{ и } X \in T\}$. Символ $[qX]$ из $Q \times T$ представляет ячейку входной ленты машины M, сканируемую считывающей головкой, причем ячейка содержит X, а машина M находится в состоянии q. Если машина M допускает строку x, то в соответствующей последовательности шагов на каждом шаге используется не более $P(|x|)$ ячеек на ленте. Следовательно, можно представлять мгновенное описание строкой w_i, состоящей ровно из $P(n)$ символов, причем все они, кроме одного, взяты из множества T;

оставшийся символ имеет вид $[qX]$. Последовательность шагов машины M можно представить строкой $\#w_1\#w_2\#...w_k\#$ для некоторого $k \geq 1$, где # — новый разделительный символ и каждая строка w_i из Δ^* представляет некоторое мгновенное описание, $|w_i| = P(n)$. При необходимости строка w_i дополняется пустыми символами, чтобы ее длина стала равной $P(n)$.

Мы хотим построить регулярное выражение R_x так, чтобы оно представляло те строки из $(\Delta \cup \{\#\})^*$, которые не описывают допускающие вычисления машины M для входа x. Следовательно, выражение R_x будет представлять все строки из $(\Delta \cup \{\#\})^*$ тогда и только тогда, когда машина M не допускает строку x. Строка $y \in (\Delta \cup \{\#\})^*$ не будет представлять допускающее вычисление машины M только тогда, когда выполнено одно или более из следующих четырех условий. Будем считать, что $|x| = n$.

1. Строка y ни для какого $k \geq 1$ не имеет вида $\#w_1\#w_2\#...w_k\#$, где $|w_i| = P(n)$ для каждого i, $1 \leq i \leq k$, и все символы строки w_i принадлежат T, за исключением одного, который имеет вид $[qX]$.
2. Начальное мгновенное описание w_1 является неправильным, т.е. строка y не начинается с $\#[q_0a_1]a_2...a_n$bb...b#, где $x = a_1...a_n$, а число пустых символов равно $P(n) - n$.
3. Машина M никогда не попадает в заключительное состояние, т.е. ни один символ вида $[q_fX]$ не входит в строку y.
4. Существуют такие два соседних мгновенных описания, что второе не получается из первого за один шаг работы машины M.

Выражение R_x имеет вид $A + B + C + D$, где A, B, C, D — регулярные выражения, представляющие множества, определяемые условиями 1–4 соответственно.

Для удобства введем сокращения для записи регулярных выражений. Если мы пользуемся алфавитом $S = \{c_1, c_2, ..., c_m\}$, то S будет обозначать регулярное выражение $c_1 + c_2 + ... + c_m$, а S^i — регулярное выражение $(S)(S) ... (S)$ (i раз), т.е. строки длины i в алфавите S. Заметим, что длина регулярного выражения S равна $2m - 1$, а длина S^i — $i(2m + 1)$. Аналогично, если $S_1 - S_2 = (d_1, d_2,..., d_r)$, то $S_1 - S_2$ будет обозначать регулярное выражение $d_1 + d_2 + ... + d_r$.

При таких соглашениях регулярное выражение A можно записать в виде

$$\begin{aligned} A = {} & \Delta^* + \Delta^* \#\Delta^* + \Delta(\Delta + \#)^* + (\Delta + \#)^*\Delta + \\ & + (\Delta + \#)^* \#T^* \#(\Delta + \#)^* + \\ & + (\Delta + \#)^* \#\Delta^* (Q \times T)\Delta^* (Q \times T)\Delta^* \#(\Delta + \#)^* + \\ & + (\Delta + \#)^* \#\Delta^{p(n)+1}\Delta^* \#(\Delta + \#)^* + \\ & + A_0 + A_1 + \ ... \ + A_{p(n)-1}, \end{aligned} \tag{10.7}$$

где A_i будут определены позже. Первые семь слагаемых в выражении (10.7) представляют соответственно строки

1) без #;

2) лишь с одним символом #;

3) не начинающиеся с #;

4) не кончающиеся на #;

5) не содержащие символа из $Q \times T$ между двумя #;

6) содержащие более одного символа из $Q \times T$ между двумя #;

7) содержащие более $p(n)$ символов из Δ между символами #.

Остальные слагаемые A_i определяются равенствами $A_i = (\Delta + \#)^*\#\Delta^i\#(\Delta + \#)^*$ и в совокупности представляют множество строк, содержащих менее $p(n)$ символов из Δ между двумя символами #.

Предлагаем читателям самостоятельно проверить, что регулярное выражение (10.7) на самом деле представляет строки, удовлетворяющие условию 1. Кроме того, первые шесть слагаемых имеют фиксированную длину, зависящую только от машины M. Длина седьмого, очевидно, имеет порядок $O(p(n))$. Слагаемое A_i имеет длину $O(i)$, так что сумма длин всех A_i и, значит, длина выражения (10.7) имеет порядок $O(p^2(n))$, причем мультипликативная константа зависит только от машины M, но не от строки x. Далее, не трудно проверить, что выражение (10.7) легко выписать за время, полиномиально зависящее от n.

Теперь заметим, что для B, C и D достаточно написать выражения, представляющие все строки, которые удовлетворяют условию 2, 3 или 4, но нарушают условие 1, т.е. нам необходимо генерировать только строки вида $\#w_1\#w_2...\#w_k$, где $|w_i| = p(n)$ и $w_i \in T^*(Q \times T)T^*$. Однако для простоты мы выпишем выражения, определеляющие множества строк, которое удовлетворяют условию 2, 3 или 4, но нарушают условие 1, а также некоторых строк, удовлетворяющих не только условию 2, 3 или 4, но и условию 1. Поскольку строки последнего вида уже представлены в выражении R_x в силу определения выражения A, их наличие или отсутствие в выражениях B, C и D не имеет значения.

Полагая, что входная строка имеет вид $x = a_1a_2...a_n$, представим выражение B в виде $B_1 + B_2 + ... + B_{P(n)}$, где

$$B_1 = \#(\Delta - [q_0a_1])\,\Delta^{p(n)-1}\#(\Delta + \#)^*,$$

$$B_i = \#\Delta^{i-1}(T - \{a_i\})\Delta^{p(n)-i}\#(\Delta + \#)^* \text{ для } 1 < i \le n,$$

и

$$B_i = \#\Delta^{i-1}(T - \{\mathfrak{b}\})\Delta^{p(n)-i}\#(\Delta + \#)^* \text{ для } n < i \le P(n).$$

Следовательно, выражения B_i отличаются от начального мгновенного описания по крайней мере i-м входным символом. Очевидно, что выражение B имеет длину $O(p^2(n))$, причем мультипликативная константа зависит только от машины M.

Положим $C = ((\Delta + \#) - (\{q_f\} \times T))^*$. Очевидно, что длина этого выражения зависит только от машины M.

Наконец, построим правильное выражение D. Допустим, нам дана строка y, представляющая правильное вычисление машины M. Тогда строка y имеет вид $\#w_1\#w_2\#...w_k\#$, где $|w_i| = p(n)$ для каждого i и w_{i+1} — мгновенное описание, которое получается из w_i за один шаг работы машины. Заметим, что по заданным трем последовательным символам $c_1c_2c_3$ строки y можно однозначно определить символ, назовем его $f(c_1c_2c_3)$, который должен появиться в строке на $p(n) + 1$ символов правее c_2. Например, если c_1, c_2 и c_3 все принадлежат множеству T, то c_2 не может измениться в следующем мгновенном описании, так что $f(c_1c_2c_3) = c_2$. Если c_1 и c_2 принадлежат множеству T, $c_3 = [qX]$ — символ из $Q \times T$ и $\delta(q, X) = (p, X, L)$, то $f(c_1c_2c_3) = [pc_2]$. Остальные правила для определения f, включая те, которые соответствуют случаю, когда c_1 или c_2 принадлежат $Q \times T$ или один из символов c_i равен #, очевидны, поэтому мы оставляем их читателям в качестве упражнения.

Выражение D представляет собой сумму членов

$$D_{c_1c_2c_3} = (\Delta + \#)^* c_1c_2c_3 (\Delta + \#)^{p(n)-1} \overline{f}(c_1c_2c_3)(\Delta + \#)^*$$

для всех c_1, c_2 и c_3 из множества $\Delta \cup \{\#\}$, где $\overline{f}(c_1c_2c_3) = \Delta \cup \{\#\} - f(c_1c_2c_3)$. Очевидно, что выражение D имеет длину $O(P(n))$ и его можно выписать за время $O(P(n))$, где мультипликативная константа зависит лишь от M.

Таким образом, регулярное выражение R_x можно построить за время $p'(n)$, где p' — полином порядка p^2. Более того, выражение R_x представляет множество $(\Delta \cup \{\#\})^*$ тогда и только тогда, когда строка x не допускается машиной M. □

Итак, в лемме 10.2 построено регулярное выражение над алфавитом $\Delta \cup \{\#\}$, размер которого зависит от машины M. Мы хотим обсудить язык регулярных выражений, дополнения которых (относительно их алфавитов) представляют непустые множества. Поскольку любой язык должен иметь конечный алфавит, мы будем кодировать регулярное выражение над конечным алфавитом Σ следующим образом.

1. +, *, ∅, ε и скобки обозначают сами себя.
2. i-й символ алфавита Σ (в произвольном порядке) обозначается через $\#w\#$, где w — десятичное представление числа i.

Таким образом, для кодирования любого регулярного выражения над любым алфавитом достаточно 17 символов. Выберем в качестве такого алфавита множество $\Gamma = \{+, *, \varnothing, \varepsilon, (,), \#, 0, 1, ..., 9\}$.

Определим язык $L_r \subseteq \Gamma^*$ как множество кодов регулярных выражений R, дополнения которых представляют непустые множества. Если R — выражение над алфавитом Σ, то дополнение берется относительно Σ. Заметим, что если выражение R имеет длину $n \geq 2$, то длина его кода не превосходит $3n\log n$.

Лемма 10.3. Язык L_r принадлежит классу $\mathcal{P}$-SPACE.

Доказательство. По теореме 10.1 достаточно построить недетерминированную машину Тьюринга M, допускающую язык L_r, с полиномиально ограниченной пространственной сложностью. Пусть R — регулярное выражение, код которого подан на вход машины M. В силу теоремы 9.2 по заданному регулярному выражению длины n можно построить недетерминированный конечный автомат A с не более чем $2n$ состояниями, распознающий язык, представленный этим регулярным выражением. Недетерминированная машина Тьюринга M сначала строит автомат A по коду выражения R, поданному машине M. Заметим, что по теореме 9.2 можно построить функцию переходов автомата A за время, не превосходящее $O(n^2)$.

Допустим, что дополнение множества, представленного выражением R, непусто. Тогда существует строка $x = a_1...a_m$, не принадлежащая языку, представленному выражением R. Машина M угадывает ее символ за символом. Для хранения множества S_i состояний, в которых автомат A может оказаться после прочтения строки $a_1...a_i$, $1 \leq i \leq m$, машина M использует массив из $2n$ битов. Она начинает с множества S_0 состояний, достижимых из начального состояния автомата A только с помощью ε-переходов. После того как машина угадает строку $a_1...a_i$, автомат A может находиться в одном из состояний, принадлежащих множеству S_i. Когда машина M угадывает следующий входной символ a_{i+1}, она прежде всего вычисляет множество $T_{i+1} = \{s' | s' \in \delta_A(s, a_{i+1})$ и $s \in S_i\}$, где δ_A — функция переходов автомата A, а затем строит множество S_{i+1}, добавляя к T_{i+1} все состояния из $\delta_A(s', \varepsilon)$ для s' из T_{i+1}. (Обратите внимание на аналогию с алгоритмом 9.1.)

Когда машина M угадает строку $a_1a_2...a_m$, она выяснит, что множество S_m не содержит заключительного состояния автомата A. Обнаружив это, машина M допустит свой вход, т.е. код регулярного выражения R. Таким образом, машина M допускает код выражения R тогда и только тогда, когда дополнение множества, представленного выражением R, непусто. □

Теперь покажем, что язык L_r состоящий из кодов тех регулярных выражений, которые представляют множества с непустыми дополнениями, является полным в смысле полиномиальной пространственной сложности.

Теорема 10.14. Если язык L_r допускается детерминированной машиной Тьюринга с временной сложностью $T(n) \geq n$, то для каждого $L \in \mathcal{P}$-SPACE найдется такой полином p_L, что язык L допускается за время $T(p_L(n))$.

Доказательство. Пусть M — детерминированная машина Тьюринга, допускающая язык L. В силу леммы 10.2 и рассуждений, проведенных выше, существует алгоритм с полиномиально ограниченным временем работы, скажем, с временной сложностью $p_1(n)$, который по входной строке x строит регулярное выражение R_x. Это выражение в фиксированном 17-символьном алфавите, и, разумеется, оно не длиннее $p_1(|x|)$. По лемме 10.2 его дополнение пусто тогда и только тогда, когда $x \in L$. По предположению можно проверить пустоту этого дополнения за $T(|R_x|) = T(p_1(|x|))$ шагов. Построение R_x занимает $p_1(|x|)$ шагов, а проверка пустоты — $T(p_1(|x|))$ шагов. Поскольку $T(n) \geq n$, то для завершения доказательства достаточно взять $p_L(n) = 2p_1(n)$. □

Следствие. Язык L_r принадлежит классу $\mathcal{P}$-TIME тогда и только тогда, когда $\mathcal{P}$-TIME = $\mathcal{P}$-SPACE.

Доказательство. Если $\mathcal{P}$-TIME = $\mathcal{P}$-SPACE, то язык L_r принадлежит классу $\mathcal{P}$-TIME по лемме 10.3. Обратное утверждение следует из теоремы 10.14 с полиномиальной функцией $T(n)$. □

Теорема 10.15. Если язык L_r допускается недетерминированной машиной Тьюринга с временной сложностью $T(n) \geq n$, то для каждого $L \in \mathcal{P}$-SPACE найдется такой полином p_L, что язык L допускается недетерминированной машиной Тьюринга за время $T(p_L(n))$.

Доказательство. Аналогично доказательству теоремы 10.14. □

Следствие. Язык L_r принадлежит классу $\mathcal{NP}$-TIME тогда и только тогда, когда $\mathcal{NP}$-TIME = $\mathcal{P}$-SPACE.

Упражнения

10.1. Выпишите все правильные последовательности шагов недетерминированной машины Тьюринга, изображенной на рис. 10.1, для входной строки 10101. Допускает ли эта недетерминированная машина Тьюринга указанную входную строку?

10.2. Неформально опишите недетерминированную машину Тьюринга или недетерминированную программу на псевдокоде Алгола, допускающую следующие множества.

а) Множество таких строк $10^{i_1}10^{i_2}...10^{i_k}$, что $i_r = i_s$ для некоторых $1 \le r \le s \le k$.

б) Можество таких строк xcy, что x и y принадлежат $\{a, b\}^*$ и x — подстрока в строке y.

в) Множество строк, аналогичное пункту (*б*), но при условии, что x —подпоследовательность в строке y.

10.3. Опишите RAM-программу, моделирующую недетерминированную RAM-программу.

10.4. Пусть M обозначает $(m \times n)$-матрицу из нулей и единиц. Напишите программу на псевдокода Алгола, которая находила бы такую наименьшую $(s \times n)$-подматрицу S матрицы M, что если $M[i, j] = 1$, то $S[k, j] = 1$ для некоторого $1 \le k \le s$. Какова временная сложность вашей программы?

10.5. Покажите, что следующие функции конструируются по пространству, неформально описав для этого детерминированную машину Тьюринга, помещающую маркер в нужную ячейку одной из своих лент:

а) n^2,

б) $n^3 - n^2 + 1$,

в) 2^n,

г) $n!$.

10.6. Покажите, что если одноленточная недетерминированная машина Тьюринга с пространственной сложностью $S(n)$, имеющая s состояний и t ленточных символов, допускает слово длины n, то она допускает его не более чем за $sS(n)t^{S(n)}$ шагов.

***10.7.** Покажите, как вычислить на машине RAM значение булевого выражения за время $O(n)$.

10.8. Покажите, что язык, состоящий из булевых функций, не являющихся тавтологиями,[10] является *NP*-полным.

10.9. Покажите, что *задача об изоморфизме подграфу* ("Изоморфен ли данный неориентированный граф G некоторому подграфу данного неориентированного графа G'?") является *NP*-полной. (*Подсказка*: воспользуйтесь *NP*-полнотой задачи о клике или о гамильтоновом цикле.)

[10]Тавтологией называется булево выражение, принимающее значение 1 на всех наборах значений ее переменных.

10.10. *Покажите, что задача о ранце* ("Существует ли для данной последовательности целых чисел $S = i_1, i_2, ..., i_n$ и целого числа k подпоследовательность в S, сумма членов которой равна k?") является *NP*-полной. (*Подсказка*: используйте задачу о точном покрытии.)

***10.11.** Покажите, что *задача о разбиении* (задача о ранце из упражнения 10.10 $\sum_{j=1}^{n} i_j = 2k$) является *NP*-полной.

10.12. Покажите, что *задача о коммивояжере* ("По данному графу G с целочисленными весами ребер найти цикл, который включает каждую вершину и сумма весов ребер которого не превосходит k") является *NP*-полной.

****10.13.** Покажите *NP*-полноту задачи распознавания следующего свойства: регулярное выражение без * (т.е. такое, в котором употребляются только операции + и ·) не представляет всех строк некоторой фиксированной длины. (*Подсказка*: преобразуйте эту задачу о регулярных выражениях в задачу о КНФ-выполнимости.)

****10.14.** Покажите *NP*-полноту задачи распознавания следующего свойства: регулярное выражение над алфавитом $\{0\}$ не представляет 0^*.

****10.15.** Пусть $G = (V, E)$ — неориентированный граф и v_1, v_2 — две разные вершины. Разрезом для v_1 и v_2 называется такое подмножество $S \subseteq E$, что любой путь между v_1 и v_2 содержит элемент из S. Покажите *NP*-полноту задачи распознавания того, есть ли в графе такой разрез размера k для v_1 и v_2, что обе части графа содержат равные количества вершин.

****10.16.** *Одномерная задача об упаковке состоит в следующем.* Пусть $G = (V, E)$ — неориентированный граф и k — положительное целое число. Существует ли такая нумерация $v_1, v_2, ..., v_n$ вершин из V, что

$$\sum_{(v_i, v_j) \in E} |i - j| \le k?$$

Покажите, что эта задача *NP*-полная.

****10.17.** Докажите, что задача о раскраске остается *NP*-полной, даже если ограничить k числом 3, а наибольшую степень каждой вершины — числом 4.

****10.18.** Выполните упражнение 10.17 для планарных графов. (*Подсказка*: покажите, что планарный граф на рис. 10.14 можно раскрасить в три цвета только тогда, когда v_1 и v_1', а также v_2 и v_2' раскрашены одинаково. Объедините этот результат с вашим ответом на упражнение 10.17.)

***10.19.** Докажите *NP*-полноту задачи о клике, непосредственно представляя вычисления недетерминированной машины Тьюринга вместо того, чтобы преобразовать ее из 3-КНФ-выполнимости.

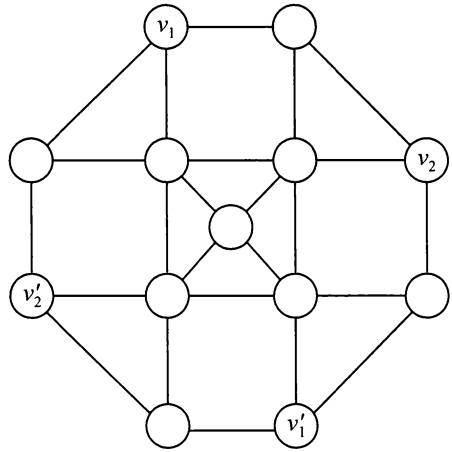


Рис. 10.14. Планарный граф

***10.20.** Рассмотрим ориентированный граф с двумя выделенными вершинами s и t. Припишем каждому ребру целочисленную пропускную способность. Можно построить максимальный поток из s в t, многократно находя путь из s в t и увеличивая поток вдоль него до максимума, разрешаемого пропускными способностями его ребер. Покажите, что задача нахождения наименьшего множества путей, на которых следует увеличивать поток, является *NP*-полной. *(Подсказка*: рассмотрите граф, аналогичный изображенному на рис. 10.15, и свяжите эту задачу с задачей о ранце.)

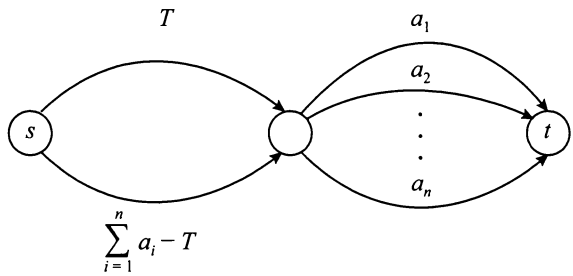


Рис. 10.15

****10.21.** *Задача о расписании работ с равными временами выполнения состоит в следующем.* Даны множество работ $S = (J_1, ..., J_n)$, лимит времени t, число процессоров p и частичный порядок $<$ на множестве S. Существует ли расписание работ из множества S, требующее не более t единиц времени, т.е. такое отображение f из S в $\{1, 2, ..., t\}$, что в каждое целое чис-

ло отображается не более p работ, и если $J < J'$, то $f(J) < f(J')$? Покажите, что эта задача является NP-полной.

10.22. Покажите, что каждая из задач, перечисленных в теореме 10.2, принадлежит классу $\mathcal{NP}$-TIME.

10.23. Пусть $p_1(x)$ и $p_2(x)$ — полиномы. Покажите, что некоторый полином для каждого x принимает значение, больше, чем $p_1(p_2(x))$.

10.24. Выпишите выражение E_{ijkl}, введенное в доказательстве теоремы 10.3, для $\delta(q_k, x_j) = \{(q_3, X_4, L), (q_9, X_2, R)\}$.

****10.25.** Постройте алгоритм полиномиальной сложности для проверки 2-выполнимости.

***10.26.** Известно, что некоторые частные случаи задачи об изоморфизме графов (например, изоморфизм планарных графов) являются легкими. Другие так же трудны, как и общая задача. Покажите, что задача об изоморфизме корневых ориентированных ациклических графов имеет ту же сложность, что и задача об изоморфизме произвольных графов.

***10.27.** *Контекстным* называется язык, допускаемый недетерминированной машиной Тьюринга с пространственной сложностью $n + 1$ (эта машина называется *линейно ограниченным автоматом*; см. Hopcroft, Ullman [1969]). Покажите, что задача распознавания, допускает ли данный линейно ограниченный автомат данную входную строку, является полной относительно полиномиальной пространственной сложности, т.е. задача распознавания принадлежности строки произвольному контекстному языку является полной в классе $\mathcal{P}$-SPACE.

10.28. Покажите, что задача об эквивалентности двух регулярных выражений является полной относительно полиномиальной пространственной сложности.

****10.29.** Опишите алгоритм, допускающий множество L_r кодов всех регулярных выражений R, представляющих непустое множество, который можно реализовать на детерминированной машине Тьюринга с линейно ограниченной памятью.

Проблемы для исследования

10.30. Одна из очевидных открытых проблем — выяснить, выполняются ли равенства $\mathcal{P}$-TIME = $\mathcal{NP}$-TIME и $\mathcal{NP}$-TIME = $\mathcal{P}$-SPACE. Если учесть объем работы, проделанный в поисках алгоритмов, решающих NP-полные задачи за полиномиальное время, то похоже, что эта проблема по меньшей мере такая же сложная, как некоторые классические проблемы математики, такие как гипотеза Ферма ("Имеет ли уравнение $x^n + y^n = z^n$ решение в целых числах для $n \geq 3$?") или проблема четырех красок.

10.31. В случае неудачи при решении упражнения 10.30 было бы интересно получить нетривиальную функцию $T(n)$, что всякий язык из класса $\mathcal{NP}$-TIME распознается детерминированной машиной с временной сложностью, не превышающей $T(n)$. Это не доказано даже для $T(n) = 2^n$.

Библиографические замечания

Дополнительную информацию о недетеминированных машинах Тьюринга можно найти в книге Hopcroft and Ullman [1969]. Теорему 10.1, устанавливающую связь между пространственной сложностью детерминированных и недетерминированных машин, доказал Savitch [1970] .

Ключевая теорема о *NP*-полноте задачи выполнимости принадлежит Cook [1971b]. Много классических *NP*-полных задач привел Karp [1972], ясно продемонстрировав важность понятия *NP*-полноты. С тех пор к известным *NP*полным задачам были добавлены новые, найденные Sahni [1972], [1973], Ullman [1973], Rounds [1973], Ibarra and Sahni [1973], Hunt and Rosencratz [1974], Garey, Johnson and Stockmeyer [1974], Bruno and Sethi [1974] и многими другими. Интересно, что до появления пионерской работы Cook [1971b] в нескольких работах была показана полиномиальная взаимосвязь между *NP*-полными задачами, но объем всего класса не был осознан. Например, Dantzig Blattnet, an Rao [1966] установили связь между задачами о коммивояжере и о кратчайшем пути с неотрицательными весами. Divetti and Graselli [1968] описали связь между задачами о покрытии множествами и о множестве ребер, разрезающих циклы.

Задачи, полные для $\mathcal{P}$-класса, впервые рассмотрели Meyer and Stockmeyer [1972]. Первый язык, полный в смысле пространственной сложности, неявно описал Savitch [1971]. Он описал язык (множество проходимых лабиринтов), полный относительно памяти размера $\log n$. Jones [1973], а также Stockmeyer and Meyer [1973] рассмотрели ограниченные формы сводимости задач за полиномиальное время. Book [1972, 1974] доказал несравнимость некоторых классов сложности.

Упражнения 10.8 и 10.9 взяты из Cook [1971b], упражнения 10.10 и 10.12 — из Karp [1972], упражнение 10.13 — из Stockmeyer and Meyer [1973] и Hunt [1973a], упражнения 10.14 и 10.28 — из Stockmeyer and Meyer [1973], а упражнения 10.15–10.18 — из Garey, Johnson and Stockmeyer [1974]. Рис. 10.14 представляет собой улучшенный вариант, предложенный M. Fisher. Доказательство *NP*-полноты задачи 3-раскрашиваемости планарных графов опубликовано в работе Stockmeyer [1973]. Упражнение 10.20 заимствовано из работы Even [1973], упражнение 10.21 — из Ullman [1973], упражнение 10.25 — из Cook [1971b], а редукция, необходимая для решения упражнения 10.27, позаимствована из Karp [1972]. Доказательство теоремы 10.13 предложил Even.

ГЛАВА 11

Некоторые доказуемо трудноразрешимые задачи

В этой главе мы приводим доказательства того, что задача пустоты дополнения для двух классов расширенных регулярных выражений является трудноразрешимой, т.е. для выполнения любого алгоритма, решающего ее для любого из двух классов, требуется по меньшей мере экспоненциальное время. Для одного из этих классов мы получим нижнюю оценку, которая значительно превышает экспоненциальную. В частности, будет показано, что эта задача требует времени, большего, чем

$$2^{n}$$

$$2^{2}$$

при любом конечном количестве двоек в показателе степени. Прежде чем получить нижние оценки, рассмотрим результаты о иерархиях, показывающие, что "чем больше времени или памяти можно использовать для вычислений, тем больше языков можно распознать".

11.1. Иерархии по сложности

В главе 10 мы показали, что некоторые задачи являются полными в смысле недетерминированного полиномиального времени или полиномиальной памяти. Чтобы доказать полноту конкретных задач, мы показали, как в терминах данной конкретной задачи формулируется произвольная задача из класса $\mathcal{P}$-TIME или $\mathcal{NP}$-SPACE. Метод доказательства представлял собой, по существу, разновидность моделирования. Например, мы показали, что задача выполнимости булевых выражений является NP-полной, а задача пустоты дополнения для регулярных выражений — полной в смысле полиномиальной памяти, причем в обоих случаях результат получался прямым вложением вычислений на машинах Тьюринга в частные случаи этих задач. Полноту других задач мы показали, сводя к ним ка-

кую-нибудь задачу, о которой уже известно, что она полна для соответствующего класса задач. Таким образом, мы показали, что оба класса, $\mathcal{NP}$-TIME и $\mathcal{P}$-SPACE, содержат самые трудные задачи, т.е. задачи, сложность которых по меньшей мере столь же велика, как и сложность любой задачи из этого класса.

Однако какими бы сильными ни были косвенные доводы, еще никому не удалось найти в классе $\mathcal{NP}$-TIME или $\mathcal{P}$-SPACE задачу, о которой можно было бы доказать, что она не принадлежит классу $\mathcal{P}$-TIME. Более того, похоже, что методов из главы 10 достаточно лишь для доказательства того, что данная задача по меньшей мере так же трудна, как и любая другая задача из некоторого класса. Чтобы действительно доказать, что данная задача не принадлежит классу $\mathcal{NP}$-TIME, требуется метод, которая позволил бы показать, что существует хотя бы один язык, не допускаемый никакой детерминированной машиной Тьюринга за полиномиальное время. Чаще всего для подобной цели применяется диагонализация. Хотя этот метод, по-видимому, не достаточно мощный, чтобы доказать, что $\mathcal{P}$-TIME $\neq$ $\mathcal{NP}$-TIME, с его помощью удалось получить результаты о иерархии как по пространственной, так и по временной сложности для обоих типов машин Тьюринга — детерминированных и недетерминированных. Каждая теорема о иерархии имеет следующий вид: для данных "хороших" функций $f(n)$ и $g(n)$, таких, что $f(n)$ растет "быстрее" $g(n)$, существует язык со сложностью $f(n)$, но не $g(n)$.

11.2. Иерархия детерминированных машин Тьюринга по пространственной сложности

В этом разделе мы докажем теорему об иерархии по пространственной сложности для детерминированных машин Тьюринга. Можно доказать аналогичные результаты для времени и для недетерминированных машин Тьюринга, но поскольку для наших целей они не нужны, мы оставляем их читателям в качестве упражнений. Вспомним, что в силу следствия 3 леммы 10.1, если язык L допускается k-ленточной детерминированной машиной Тьюринга с пространственной сложностью $S(n)$, он допускается и одноленточной детерминированной машиной Тьюринга с пространственной сложностью $S(n)$. Таким образом, можно ограничиться рассмотрением одноленточных детерминированных машин Тьюринга.

Чтобы получить результат об иерархии по пространственной сложности, необходимо *перечислить* детерминированные машины Тьюринга, т.е. каждому неотрицательному целому числу поставить в соответствие единственную детерминированную машину Тьюринга. Кроме того, каждая детерминированная машина Тьюринга должна появляться в этом перечислении бесконечно много раз. Мы будем перечислять только одноленточные детерминированные машины Тьюринга с входным алфавитом $\{0, 1\}$, поскольку этого вполне достаточно. Для перечисления всех машин Тьюринга достаточно простого обобщения этого метода.

Не умаляя общности, можно принять следующие соглашения о способе представления одноленточных детерминированных машин Тьюринга.

1. Состояния имеют имена $q_1, q_2, ..., q_s$, для некоторого s, причем q_1 — начальное и q_s — допускающее состояния.
2. Входным алфавитом является множество $\{0, 1\}$.
3. Ленточным алфавитом является множество $\{X_1, X_2, ..., X_t\}$ для некоторого t, где $X_1 = \text{b}$, $X_2 = 0$ и $X_3 = 1$.
4. Функция переходов δ представляет собой список пятерок вида $(q_l, X_j, q_k, X_l, D_m)$, означающих, что $\delta(q_l, X_j) = (q_k, X_l, D_m)$, где q_i и q_k — состояния, X_j и X_l — ленточные символы, D_m — направление сдвига головки: L (влево), R (вправо) и S (остается на месте) при $m = 0$, 1 и 2 соответственно. Считаем, что такая пятерка кодируется строкой $1\ 0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m\ 1$.
5. Сама машина Тьюринга кодируется строкой, получающейся конкатенацией в любом порядке кодов пятерок, представляющих ее функцию переходов. При необходимости строку можно дополнить ведущими единицами. Результатом такого кодирования будет начинающаяся с единицы строка из нулей и единиц, которую можно интерпретировать как целое число.

Любое целое число, которое нельзя декодировать, считается представлением тривиальной машины Тьюринга с пустой функцией переходов. Каждая одноленточная детерминированная машина Тьюринга будет появляться в этом перечислении бесконечно много раз, поскольку, имея какую-то детерминированную машину Тьюринга, можно приписывать к ее коду ведущие единицы и получать все бо́льшие и бо́льшие целые числа, представляющие одно и то же множество пятерок.

Построим четырехленточную детерминированную машину Тьюринга M_0, рассматривающую свою входную строку x одновременно как код одноленточной детерминированной машины Тьюринга M и как вход для машины M. Мы построим машину M так, чтобы для каждой детерминированной машины Тьюринга M с данной сложностью нашлась хотя бы одна входная строка, которую допускает машина M_0, но отвергает машина M, или наоборот. Одна из возможностей машины M_0 состоит в моделировании произвольной машины Тьюринга по ее описанию. Машина M_0 будет распознавать, допускает ли машина Тьюринга M входную строку x так, что при этом используется не более $S_1(|x|)$ ячеек, где S_1 — некоторая функция. Если машина M допускает x, используя не более $S_1(|x|)$ ячеек, то машина M_0 не допускает x. В противном случае машина M_0 допускает строку x. Таким образом, либо поведение машины M_0 отличается от поведения i-й детерминированной машины Тьюринга на входной строке x, являющейся двоичным представ-

лением числа i, либо i-я детерминированная машина Тьюринга на входной строке x использует более $S_1(|x|)$ ячеек.

Говорят, что машина M_0 является *диагонализацией* по всем детерминированным машинам Тьюринга с пространственной сложностью $S_1(n)$, поскольку если представить двумерную таблицу, (i, j)-й элемент которой показывает, допустила ли i-я машина Тьюринга входную строку j, то работа машины M_0 будет отличаться от работы некоторых машин Тьюринга на диагонали этой таблицы. В частности, машина M_0 будет отличаться от машин Тьюринга, допускающих свой вход с пространственной сложностью, не большей $S_1(n)$. В силу следствия 3 леммы 10.1 существует одноленточная детерминированная машина Тьюринга M_0', эквивалентная машине M_0 с такой же пространственной сложностью. Поскольку сама машина M_0' тоже представлена в таблице и она не может отличаться от себя, то можно сделать вывод, что пространственная сложность машин M и M_0' отличается от $S_1(n)$. Реальная конструкция машины M_0 специально сделана более сложной, потому что машина M_0 должна иметь такую пространственную сложность $S_2(n)$, что функции $S_1(n)$ и $S_2(n)$ почти одинаковы.

Определение. Пусть $f(n)$ — произвольная функция. Обозначим через $\inf\limits_{n\to\infty} f(n)$ предел при $n \to \infty$ наибольшей нижней грани последовательности $f(n)$, $f(n+1)$, ...

Пример 11.1. Поскольку последовательность $(n^2+1)/n^2$ монотонно убывает, то

$$\inf_{n\to\infty} \frac{n^2+1}{n^2} = \lim_{n\to\infty} \frac{n^2+1}{n^2} = 1.$$

В качестве второго примера рассмотрим функцию $f(n) = 1 - 1/n$, если n не является степенью числа 2, и $f(n) = n$ — в противном случае. Тогда $\inf\limits_{n\to\infty} f(n) = 1$, поскольку наибольшая нижняя грань последовательности $f(n), f(n+1)$, ... равна $1 - 1/n$, если n — не является степенью числа 2, и $1 - 1/(n+1)$ — в противном случае. □

Теорема 11.1. Пусть $S_1(n) \geq n$ и $S_2(n) \geq n$ — две функции, конструируемые по пространству, причем

$$\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0.$$

Тогда существует язык L, допускаемый некоторой детерминированной машиной Тьюринга с пространственной сложностью $S_2(n)$, но не допускаемый

никакой детерминированной машиной Тьюринга с пространственной сложностью $S_1(n)$.[1]

Доказательство. Пусть M_0 — четырехленточная детерминированная машина Тьюринга, которая работает на входной строке x длины n следующим образом.

1. Машина M_0 отмечает $S_2(n)$ ячеек на каждой ленте. После этого каждый раз, когда любая ее считывающая головка попытается выйти за отмеченные ячейки, машина M_0 останавливается, не допуская входную строку.
2. Если строка x не является кодом никакой одноленточной детерминированной машины Тьюринга, машина M_0 останавливается, не допуская входную строку.
3. В остальных случаях будем считать, что x — код детерминированной машины Тьюринга M. Машина M_0 определяет количество t используемых машиной M символов на ленте и количество s ее состояний. Третья лента машины M_0 может играть роль промежуточной памяти при вычислении t. Затем машина M_0 разбивает свою вторую ленту на $S_1(n)$ блоков по $\lceil \log t \rceil$ ячеек в каждом, причем эти блоки отделяются друг от друга ячейками, содержащими маркер #. Таким образом, если $(1+\lceil \log t \rceil)S_1(n) \le S_2(n)$, то количество ячеек равно $(1+\lceil \log t \rceil)S_1(n)$. Каждый символ на ленте машины M будет закодирован двоичным числом в соответствующем блоке на второй ленте машины M_0. Вначале машина M_0 записывает свою входную строку в виде двоичного кода в блоки ленты 2, а неиспользованные блоки заполняет кодом пустого символа.
4. На ленте 3 машина M_0 образует блок из $\lceil \log s \rceil + \lceil \log S_1(n) \rceil + \lceil \log t \rceil S_1(n)$ ячеек, инициализируя их нулями. Будем снова предполагать, что количество требуемых ячеек не превосходит $S_2(n)$. Лента 3 играет роль счетчика, в который можно помещать числа вплоть до $sS_1(n)t^{S_1(n)}$.
5. Машина M_0 моделирует машину M, используя ленту 1 (т.е. свою входную ленту) для определения шагов машины M и ленту 2 для моделирования ленты машины M. Номера шагов машины M записываются в виде двоичных чисел в блок на ленте 3, а на ленте 4 хранится состояние машины M. Если машина M допускает свой вход, то машина M_0 останавливается, не допу-

[1] В литературе часто приводится определение пространственной сложности машины Тьюринга, при котором не учитывается количество ячеек, просмотренных на входной ленте, поскольку на входной ленте нельзя изменять символы. При таком определении можно также рассматривать функции для пространственной сложности, меньшие n, и из условий теоремы убрать требование $S_1(n) \ge n$ и $S_2(n) \ge n$. Поскольку здесь нас интересуют лишь большая пространственная сложность, то результат для сложностей, меньших n, мы не приводим. В этой теореме можно ослабить ограничение на конструируемость $S_1(n)$ по памяти.

ская свою входную строку. Машина M_0 допускает входную строку, если M останавливается, не допуская свою входную строку, или если для моделирования работы M машина M_0 пытается занять больше ячеек на ленте 2, чем отведено, или если переполняется счетчик на ленте 3, т.е. количество шагов, сделанное машиной M, превосходит $sS_1(n)t^{S_1(n)}$.

Машина Тьюринга M_0, описанная выше, обладает пространственной сложностью $S_2(n)$ и допускает некоторый язык L. Допустим, что язык L допускается какой-то детерминированной машиной Тьюринга M_i с пространственной сложностью $S_1(n)$. В силу следствия 3 леммы 10.1 можно считать машину M_i одноленточной. Пусть машина M_i имеет s состояний и t символов на ленте. Записывая одну за другой пятерки, изображающие команды машины M_i и добавляя при необходимости единицы к началу полученной строки, можно построить строку w длины n, представляющую машину M_1, причем n настолько велико, что

$$S_2(n) > \text{MAX}\left[\left(1+\lceil \log t \rceil\right)S_1(n),\ \lceil \log s \rceil + \lceil \log S_1(n) \rceil + \lceil \log t \rceil S_1(n)\right].$$

Равенство $\inf_{n\to\infty}\left(S_1(n)/S_2(n)\right)=0$ гарантирует, что такое n существует. Теперь, когда на вход машины M_0 подается строка w, у машины M_0 есть достаточно памяти для моделирования работы M_i. Она допускает входную строку w тогда и только тогда, когда M_i не допускает его. Однако мы предположили, что машина M_i допускает язык L, т.е. ее выход совпадает с выходом M на всех входах. Следовательно, машина M_i невозможна, т.е. язык L не допускается никакой детерминированной машиной Тьюринга с пространственной сложностью $S_1(n)$. □

Типичное приложение теоремы 11.1 состоит, например, в доказательстве существования языка, допускаемого детерминированной машиной Тьюринга с пространственной сложностью $n^2 \log n$, но не допускаемого никакой детерминированной машиной Тьюринга с пространственной сложностью n^2. Некоторые другие приложения будут описаны в оставшейся части этой главы.

11.3. Задача, требующая экспоненциального объема времени и памяти

В разделе 10.6 мы изучили задачу пустоты дополнения для регулярных выражений и показали, что она является полной в смысле полиномиальной пространственной сложности. Поскольку класс регулярных множеств замкнут относительно пересечения и дополнения, то добавление знаков операций пересечения $\cap$ и дополнения $\neg$ к системе обозначений для регулярных выражений не увеличивает класс множеств, которые можно описать такими выражениями. Однако использование этих знаков значительно сокращает длину выражений, необходимых

для описания некоторых регулярных множеств. Из-за возможности такого сокращения для решения ряда задач с расширенными регулярными выражениями требуется еще больше времени, чем для решения задач с обычными регулярными выражениями (при условии, что время измеряется как функция длины данного выражения).

Определение. *Расширенное регулярное выражение* над алфавитом Σ определяется следующим образом.

1. ε, $\varnothing$ и a из Σ являются расширенными регулярными выражениями, представляющими соответственно $\{\varepsilon\}$, пустое множество и $\{a\}$.
2. Если R_1 и R_2 — расширенные регулярные выражения, представляющие соответственно языки L_1 и L_2, то $(R_1 + R_2)$, $(R_1 \cdot R_2)$, (R_1^*), $(R_1 \cap R_2)$ и $(\neg R_1)$ — тоже расширенные регулярные выражения, представляющие языки $L_1 \cup L_2$, L_1L_2, L_1^*, $L_1 \cap L_2$ и Σ^* — L_1 соответственно.

В расширенных регулярных выражениях можно опускать пары скобок, если принять соглашение, что приоритеты операций возрастают в следующем порядке:

$$+ \cap \neg \cdot *$$

Далее, знак операции $\cdot$ обычно не пишется. Например, выражение $a \neg b^* + c \cap d$ означает

$$((a \cdot (\neg(b^*))) + (c \cap d)).$$

Если расширенное регулярное выражение не содержит знаков дополнения, то его называют *полурасширенным*. *Задача о пустоте дополнения* для полурасширенных регулярных выражений состоит в выяснении, пусто ли дополнение множества, представленного данным выражением R (или, что эквивалентно, представляет ли R все строки во входном алфавите). Например, регулярное выражение

$$b^* + b^*aa^*(\varepsilon + b(a + b)(a + b)^*) + b^*aa^*b$$

представляет множество $(a + b)^*$, поскольку

$$b^* + b^*aa^*(\varepsilon + b(a + b)(a + b)^*) = \neg b^*aa^*b.$$

Приступим к доказательству того, что задача о пустоте дополнения для полурасширенных регулярных выражений не принадлежит классу $\mathcal{P}$-SPACE и, значит, не принадлежит классу $\mathcal{NP}$-TIME, поскольку $\mathcal{NP}$-TIME содержится в $\mathcal{P}$-SPACE. Фактически мы покажем, что существуют такие полурасширенные регулярные выражения длины n, для которых решение задачи о пустоте дополнения требует объема памяти (и, следовательно, времени) не менее $c'c^{\sqrt{n/\log n}}$, где $c' > 0$ и $c > 1$ — некоторые константы.

В разделе 11.4 мы увидим, что задача о пустоте дополнения для всего класса расширенных регулярных выражений гораздо труднее, чем для полурасширенных выражений. С учетом этого факта мы будем излагать многие результаты настоящего раздела в терминах всего класса расширенных регулярных выражений.

Кроме сокращений, введенных в разделе 10.6, будем обозначать расширенное регулярное выражение $R_1 + R_2 + ... + R_k$ через $\sum_{i=1}^{k} R_i$. Мы также будем писать R^+ вместо RR^*. Когда мы будем говорить о длине выражения, то будем подразумевать длину исходного выражения, записанного без этих сокращений.

Теперь введем понятие эталона. Пусть Σ — алфавит и x — произвольная строка в Σ^*. Введем множество $\texttt{CYCLE}(x) = \{zy | x = yz$, где $y \in \Sigma^*$ и $z \in \Sigma^*\}$. Пусть # — специальный маркер, не принадлежащий Σ. Множество $\texttt{CYCLE}(x\#)$ называется *эталоном* длины $|x\#|$. Таким образом, эталон — это множество всех циклических перестановок строки $x\#$. Когда это не будет вызывать недоразумений, мы будем называть эталоном саму строку $x\#$.

С помощью эталона мы будем определять длину мгновенного описания в правильном вычислении машины Тьюринга. Сначала покажем, как можно представить некоторые длинные эталоны относительно короткими полурасширенными регулярными выражениями.

Лемма 11.1. Для каждого $k \geq 1$ существует эталон с длиной, большей 2^k, который можно представить таким полурасширенным регулярным выражением R, что $|R| \leq ck^2$ для некоторой константы c, не зависящей от k.

Доказательство. Пусть $A = \{a_0, a_1, ..., a_k\}$ — алфавит из $k + 1$ различных символов. Положим $x_0 = a_0a_0$ и $x_i = x_{i-1}a_ix_{i-1}a_i$ для $1 \leq i < k$. Тогда

$$x_i = (...((a_0^2 a_1)^2 a_2)^2 ... a_i)^2.$$

Длина строки x_0 равна 2, а длина строки x_i больше удвоенной длины строки x_{i-1}. Следовательно, длина строки x_{k-1} не меньше 2^k для $k \geq 1$.

$\texttt{CYCLE}(x_{k-1}a_k)$ — искомый эталон. Осталось показать, что некоторое короткое полурасширенное регулярное выражение представляет множество $\texttt{CYCLE}(x_{k-1}a_k)$.

Для $0 \leq i \leq k$ пусть $A_i = (a_0 + a_1 + ... + a_i)$ и $A - A_i = (a_{i+1} + a_{i+2} + ... + a_k)$. Положим

$$\begin{aligned} R_0 = a_0a_0\left[(A - A_0)^+ a_0^2\right]^*(A - A_0)^+ + \\ + a_0\left[(A - A_0)^+ a_0^2\right]^*(A - A_0)^+ a_0 + \\ + \left[(A - A_0)^+ a_0^2\right]^*(A - A_0)^+ a_0a_0(A - A_0)^* \end{aligned}$$

и

$$R_i = A_{i-1}^+ a_i A_{i-1}^+ a_i \left[\left(A - A_i\right)^+ \left(A_{i-1}^+ a_i\right)^2 \right]^* \left(A - A_i\right)^+ A_{i-1}^* +$$
$$+ \quad a_i A_{i-1}^+ a_i \left[\left(A - A_i\right)^+ \left(A_{i-1}^+ a_i\right)^2 \right]^* \left(A - A_i\right)^+ A_{i-1}^+ +$$
$$+ \quad A_{i-1}^+ a_i \left[\left(A - A_i\right)^+ \left(A_{i-1}^+ a_i\right)^2 \right]^* \left(A - A_i\right)^+ A_{i-1}^+ a_i A_{i-1}^* +$$
$$+ \quad a_i \left[\left(A - A_i\right)^+ \left(A_{i-1}^+ a_i\right)^2 \right]^* \left(A - A_i\right)^+ A_{i-1}^+ a_i A_{i-1}^+ +$$
$$+ \quad \left[\left(A - A_i\right)^+ \left(A_{i-1}^+ a_i\right)^2 \right]^* \left(A - A_i\right)^+ A_{i-1}^+ a_i A_{i-1}^+ a_i \left(A - A_i\right)^*$$

для $1 \le i \le k-1$.

Мы утверждаем, что полурасширенное регулярное выражение

$$R = R_0 \cap R_1 \cap \ldots \cap R_{k-1} \cap A_{k-1}^* a_k A_{k-1}^*$$

представляет CYCLE($x_{k-1}a_k$). Для доказательства покажем индукцией по i, что любая строка из множества, представленного выражением $R_0 \cap R_1 \cap \ldots \cap R_i$ имеет вид

$$y_2 \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ y_1 +$$
$$+ \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ x_i \left(A - A_i\right)^*,$$

где $y_1 y_2 = x_i$.

База индукции. Для $i = 0$ видно, что R_0 представляет в точности строки вида

$$a_0^j \left[\left(A - A_0\right)^+ a_0^2 \right]^* \left(A - A_0\right)^+ a_0^{2-j} +$$
$$+ \left[\left(A - A_0\right)^+ a_0^2 \right]^* \left(A - A_0\right)^+ a_0 a_0 \left(A - A_0\right)^*$$

для некоторого $0 \le j \le 2$. Поскольку $x_0 = a_0^2$, утверждение верно.

Шаг индукции. Предположим, что $R_0 \cap R_1 \cap \ldots \cap R_{i-1}$ представляет все строки вида

$$y_2' \left[\left(A - A_{i-1}\right)^+ x_{i-1} \right]^* \left(A - A_{i-1}\right)^+ y_1' +$$
$$+ \left[\left(A - A_{i-1}\right)^+ x_{i-1} \right]^* \left(A - A_{i-1}\right)^+ x_{i-1} \left(A - A_{i-1}\right)^*, \qquad (11.1)$$

где $y_1' y_2' = x_{i-1}$.

Рассмотрим строку z из пересечения множества строк (11.1) и R_i. Каждая подстрока строки z, принадлежащая A_{i-1}^+ и имеющая максимальную длину, должна совпадать с x_{i-1}, кроме тех случаев, когда она является началом или концом этой строки (тогда она равна y_1' или y_2'). Следовательно, каждое вхождение A_{i-1}^+ в R_i

можно заменить на x_{i-1} (если только оно стоит не в начале и не в конце этого регулярного выражения), не изменяя пересечения множеств (11.1) и R_i. Вхождения A_{i-1}^+ и A_{i-1}^* в начале и в конце выражения R_i можно заменить на y_2' или y_1' соответственно. Замечая, что $x_{i-1}a_i x_{i-1}a_i = x_i$, получаем

$$\begin{aligned}
&R_0 \cap R_1 \cap \ldots \cap R_i = \\
&= y_2' a_i x_{i-1} a_i \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ y_1' + \\
&+ \quad a_i x_{i-1} a_i \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ x_{i-1} + \\
&+ \quad y_2' a_i \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ x_{i-1} a_i y_1' + \\
&+ \quad a_i \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ x_{i-1} a_i x_{i-1} + \\
&+ \quad \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ x_{i-1} a_i x_{i-1} a_i \left(A - A_i\right)^*.
\end{aligned} \tag{11.2}$$

Наконец, (11.2) можно переписать в виде

$$\begin{aligned}
&y_2 \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ y_1 + \\
&+ \left[\left(A - A_i\right)^+ x_i \right]^* \left(A - A_i\right)^+ x_i \left(A - A_i\right)^*,
\end{aligned}$$

где $y_1 y_2 = x_i$. Отсюда вытекает утверждение индукции. Полагая $i = k - 1$ и пересекая с $A_{k-1}^* a_k A_{k-1}^*$, заключаем, что R представляет $\mathtt{CYCLE}(x_{k-1}a_k)$.

Осталось показать, что длина расширенного регулярного выражения R не превосходит ck^2. Это непосредственно следует из существования такой константы c_1, что длина каждого регулярного выражения R_i не превосходит $c_1 k$. Иначе говоря, A_i и $A - A_i$ обозначают расширенные регулярные выражения длины $O(k)$, и поэтому каждое слагаемое суммы, определяющей R_i, обозначает расширенное регулярное выражение длины $O(k)$. Следовательно, длина полурасширенного регулярного выражения R ограничена сверху числом ck^2 для некоторой новой константы c. □

Конструкция, аналогичная конструкции из леммы 11.1, позволяет написать короткое регулярное выражение, представляющее множество всех строк из A^*, кроме $x = x_{k-1}$. Это регулярное выражение понадобится нам вместе с полурасширенным регулярным выражением R, только построенным для представления множества $\mathtt{CYCLE}(x\#)$.

Лемма 11.2. Пусть алфавит $A=\{a_0, a_1, \ldots, a_k\}$ и строки x_i — те же, что и в лемме 11.1. Тогда найдется регулярное выражение длины $O(k^2)$, представляющее $\neg x_{k-1}$.

Доказательство. Строка x_{k-1} обладает следующими свойствами.

1. Она не пустая и начинается с символа a_0 (символы a_0 входят в нее парами, и за каждым вхождением такой пары следует символ a_1).
2. Для $1 \le i \le k-2$ символы a_i также входят парами. В каждой паре два символа a_i отделены один от другого строкой из $a_0 A_{i-1}^*$, причем за вторым символом a_i этой пары следует a_{i+1}. Иначе говоря, за символом a_i, слева от которого стоят исключительно символы из A_{i-1} или непосредственно слева от которого стоит строка из A_{i-1}^*, имеющая перед собой символ из $A - A_i$, следует символ a_0.
3. Символ a_{k-1} входит ровно два раза. За первым экземпляром следует a_0, а второй является самым правым символом.

Еще важнее то, что x_{k-i} — единственная строка, обладающая этими свойствами. Можно доказать индукцией по j, что если $b_1 b_2 ... b_j$ — префикс строки, обладающей этими свойствами, то $b_1 b_2 ... b_j$ — префикс строки x_{k-1}. Например, в силу свойства 1 первым символом является a_0. Также в силу свойства 1 изолированный символ a_0 не может быть последним в этой строке и за ним может следовать только символ a_0, значит, строка начинается с $a_0 a_0$. Далее, опять по свойству 1 за символами $a_0 a_0$ должен следовать символ a_1. В силу свойства 2 при $i = 1$ за строкой $a_0 a_0 a_1$ должен следовать символ a_0 и т.д. Формальное доказательство по индукции оставляем читателям в качестве упражнения. Поскольку x_{k-1} на самом деле обладает этими свойствами, эта строка является единственной.

Легко написать регулярное выражение, представляющее все строки, не обладающие хотя бы одним из этих свойств. Строки, не обладающие свойством 1, представляются выражением[2]

$$S_1 = \varepsilon + (A - a_0)A^* + A^* a_0 a_0[(A - a_1)A^* + \varepsilon] + \\ + [\varepsilon + A^*(A - a_0)]a_0[(A - a_0)A^* + \varepsilon].$$

Первое слагаемое в S_1 обозначает пустую строку, второе — те строки, которые не начинаются с символа a_0, третье — те строки, в которых после пары символов a_0 не следует a_1, и последнее — те строки, которые содержат изолированный символ a_0. Строки, не обладающие свойством 2, представляются выражением

$$S_2 = \sum_{i=1}^{k-2} \Big[A^* a_i A_{i-1}^* a_i \Big[\big(A - a_{i+1}\big) A^* + \varepsilon \Big] + \\ + \Big[\varepsilon + A^* \big(A - A_i\big) \Big] A_{i-1}^* a_i \Big[\big(A - a_0\big) A^* + \varepsilon \Big].$$

Наконец, строки, не обладающие свойством 3, представляются выражением

---

[2] Через $A - a_i$ обозначена сумма $\sum_{\substack{i=0 \\ j \ne i}}^{k} a_i$.

$$S_3 = A^* a_{k-1} A^* a_{k-1} A^+ + (A - a_{k-1})^* a_{k-1} (A - a_{k-1})^* + \\ + (A - a_{k-1})^* + (A - a_{k-1})^* a_{k-1} (A - a_0) A^*.$$

В выражении S_3 первое слагаемое обозначает все строки, имеющие более двух вхождений символа a_{k-1}, а также строки с ровно двумя вхождениями символа a_{k-1}, второе из которых не приходится на правый конец строки. Таким образом, $S_1 \cup S_2 \cup S_3$ представляет $\neg x_{k-1}$. Легко видеть, что длина выражения $S_1 \cup S_2 \cup S_3$ имеет порядок $O(k^2)$. □

Прежде чем переходить к доказательству отсутствия алгоритма с полиномиальной памятью или с полиномиальным временем работы, который решал бы задачу пустоты дополнения для полурасширенных регулярных выражений, введем два определения.

Определение. *Гомоморфизмом* из Σ_1^* в Σ_2^* называется такая функция h, что $h(xy) = h(x)h(y)$ для любых строк x и y. Отсюда непосредственно следует, что $h(\varepsilon) = \varepsilon$ и $h(a_1 a_2 ... a_n) = h(a_1)h(a_2)...h(a_n)$. Таким образом, гомоморфизм h однозначно определяется значением $h(a)$ для каждого $a \in \Sigma_1$.

Говорят, что гомоморфизм h *сохраняет длину*, если $h(a)$ — однобуквенное слово из Σ_2 для каждого символа $a \in \Sigma_1$. Гомоморфизм, сохраняющий длину, просто переименовывает символы, возможно, отождествляя несколько символов путем переименования их в один символ. Если $w \in \Sigma_2^*$, то $h^{-1}(w) = \{x | h(x) = w\}$. Если $L \subseteq \Sigma_2^*$, то $h^{-1}(L) = \{x | h(x) \in L\}$.

Пример 11.2. Пусть $\Sigma_1 = \{a, b, c\}$ и $\Sigma_2 = \{0, 1\}$. Определим гомоморфизм h равенствами $h(a) = 010$, $h(b) = 1$ и $h(c) = \varepsilon$. Тогда $h(abc) = 0101$. Заметим, что h не сохраняет длину. Пусть L — язык, представляемый расширенным регулярным выражением $1 + \neg 1^*$, или, в эквивалентной форме, обычным регулярным выражением $1 + (0 + 1)^* 0 (0 + 1)^*$. Тогда $h^{-1}(L)$ представляется расширенным регулярным выражением $c^* bc^* + \neg(b + c)^*$ или обычным регулярным выражением $c^* bc^* + (a + b + c)^* a (a + b + c)^*$. □

Лемма 11.3. Пусть h — сохраняющий длину гомоморфизм из Σ_1^* в Σ_2^*, а R_2 — расширенное регулярное выражение, представляющее множество $S \subseteq \Sigma_2^*$. Можно построить такое расширенное регулярное выражение R_1, представляющее $h^{-1}(S)$, что его длина будет ограничена длиной выражения R_2, умноженной на константу (зависящую только от h), и оно будет содержать знак дополнения только тогда, когда его содержит R_2.

Доказательство. Заменим каждое вхождение в R_2 символа из Σ_2 регулярным выражением, представляющим множество символов, отображаемых в этот символ гомоморфизмом h. Например, если $\{a_1, a_2, ..., a_r\}$ — множество символов, отображаемых в a, заменим a на $(a_1 + a_2 + ... + a_r)$. Пусть R_1 — расширенное регулярное выражение, получающееся в результате такой замены. Доказательство того, что выражение R_1 представляет $h^{-1}(S)$, проводится индукцией по числу вхождений в R_2 знаков +, ·, *, $\cap$ и $\neg$. □

Теперь мы можем представить последовательности мгновенных описаний машин Тьюринга во многом так же, как делали это в лемме 10.2. Иначе говоря, для заданной машины Тьюринга $M = (Q, T, I, \delta, b, q_0, q_f)$ представляем ее мгновенное описание последовательностью символов из T и еще одним символом вида $[qX]$, где $q \in Q$ и $X \in T$, который указывает состояние и положение головки. При необходимости можно дополнить мгновенное описание пустыми символами, чтобы все мгновенные описания одного и того же вычисления были одинаковой длины.

Чтобы облегчить сравнение символов одного мгновенного описания с соответствующими символами в следующем мгновенном описании, применим к самим мгновенным описаниям эталон, длина которого равна длине этих мгновенных описаний. Формально мы используем "двухдорожечную" строку символов, в которой верхняя дорожка содержит последовательность мгновенных описаний, а нижняя — повторяющийся эталон. Здесь "двухдорожечными" символами служат пары $[a, b]$, где a — символ на верхней дорожке, а b — на нижней. Это изображено на рис. 11.1, где использован эталон CYCLE(x#) (x может быть любой строкой, не содержащей #).

Определение. Для заданной машины Тьюринга M положим $\Delta_1 = T \cup (Q \times T) \cup \{\#\}$ и Δ_2 — множество символов, входящих в x#, т.е. Δ_1 — множество символов, которые могут появляться на верхней дорожке, а Δ_2 — на нижней. "Двухдорожечным" алфавит имеет вид $\Delta_1 \times \Delta_2$. *Правильным вычислением* машины M с эталоном CYCLE(x#) называют строку из множества $(\Delta_1 \times \Delta_2)^*$, имеющую вид, как на рис. 11.1, где $C_i \vdash C_{i+1}$ за один шаг машины M для $0 \le s <$ f. C_0 — начальное мгновенное описание, и состоянием последнего мгновенного описания C_f является q_f. В мгновенное описание дописаны пустые символы, чтобы все мгновенные описания имели длину $|x|$.

Верхняя дорожка	#	C_0	#	C_1	#	...	#	C_f	#
Нижняя дорожка	#	x	#	x	#	...	#	x	#

Рис. 11.1. Последовательность мгновенных описаний с эталоном

Лемма 11.4. Пусть R_1 — расширенное регулярное выражение для множества CYCLE(x#) и R_1' — регулярное выражение над алфавитом строки x#, представляющее все строки, кроме x. Можно построить такое расширенное регулярное выражение R_2, представляющее все неправильные вычисления машины M с эталоном CYCLE(x#), что его длина будет линейной по $|R_1|+|R_1'|$, причем коэффициент пропорциональности будет зависеть только от машины M, и оно будет содержать знак дополнения только тогда, когда его содержит R_1 или R_1'.

Доказательство. Строка не является правильным вычислением с эталоном CYCLE(x#) тогда и только тогда, когда либо

1) нижняя дорожка не содержится в #(x#)*, либо
2) нижняя дорожка содержится в #(x#)*, но верхняя дорожка не является правильным вычислением.

Если Δ_1 и Δ_2 — алфавиты верхней и нижней дорожек соответственно, то обозначим через h_1 и h_2 гомоморфизмы, отображающие $(\Delta_1 \times \Delta_2)^*$ в Δ_1^* и Δ_2^* соответственно, так что $h_1([a, b] = a$ и $h_2([a, b] = b$. Тогда

$$h_2^{-1}\left[\Delta_2^*\#\left(R_1' \cap (\Delta_2 - \#)\right)^*\right)\#\Delta_2^* + (\Delta_2 - \#)\Delta_2^* + \Delta_2^*(\Delta_2 - \#)\Big] + \varepsilon$$

представляет те и только те строки, у которых нижняя дорожка не принадлежит #(x#)*. Первое слагаемое аргумента в h_2^{-1} представляет строки, у которых между двумя знаками # стоит символ, отличающийся от x, а остальные представляют строки, которые не начинаются или не кончаются символом #. По лемме 11.3 найдется расширенное регулярное выражение с длиной, пропорциональной $|R_1'|$, представляющее это множество.

Данная строка удовлетворяет условию 2, когда два символа, отделенные друг от друга символами в количестве, равном длине эталона, не соответствуют шагу машины M либо когда происходит одно или несколько из следующих нарушений структуры.

1. Строка на верхней дорожке не начинается или не кончается символом #.
2. В первом мгновенном описании или совсем нет состояния, или есть два или более состояния.
3. В первом мгновенном описании начальное состояние не является компонентом первого символа.
4. Допускающее состояние не появляется в качестве компонента никакого символа.

5. Первое мгновенное описание не имеет корректной длины, т.е. первые два символа # на верхней дорожке стоят не в тех же ячейках, что и первые два символа # на нижней дорожке.

Покажем, как написать расширенное регулярное выражение для последовательностей в $(\Delta_1 \times \Delta_2)^*$, по какой-то причине не отражающих правильные шаги машины M. Как и в лемме 10.2, замечаем, что в правильном вычислении три последовательных символа $c_1c_2c_3$ на верхней дорожке однозначно определяют символ, стоящий на $|x\#|$ позиций правее c_2. Пусть, как и раньше, этим символом будет $F(c_1c_2c_3)$. Положим

$$R_{c_1c_2c_3} = (\Delta_1 \times \Delta_2)^* \left[h_1^{-1}\left(c_1c_2c_3\Delta_1^*\right) \cap h_2^{-1}\left(R_1\right) \right] \left[h_1^{-1}\left(\Delta_1\left(\Delta_1 - f\left(c_1c_2c_3\right)\right)\Delta_1\right) \right] (\Delta_1 \times \Delta_2)^* \quad (11.3)$$

и

$$R = \sum_{c_1c_2c_3} R_{c_1c_2c_3}.$$

Заметим, что $h_1^{-1}\left(c_1c_2c_3\Delta_1^*\right)$ представляет все строки из $(\Delta_1 \times \Delta_2)^*$, у которых верхняя дорожка начинается с $c_1c_2c_3$, а $h_2^{-1}\left(R_1\right)$ — все строки длины $|x\#|$, у которых нижняя дорожка корректна. Поэтому их пересечение содержит все строки длины $|x\#|$, у которых на нижней дорожке стоит строка из CYCLE($x\#$), а верхняя начинается с $c_1c_2c_3$. Теперь должно быть понятно, что R включает все строки, удовлетворяющие условию 2, поскольку в них есть шаг, незаконный для M. Кроме них туда будут включены некоторые строки, у которых на нижней дорожке не стоит #($x\#$)*; они удовлетворяют и условию 1, и их присутствие или отсутствие в R несущественно. Далее, длина выражения R ограничена длиной выражения R_1, умноженной на константу (зависящую от M). Чтобы убедиться в истинности этого утверждения, достаточно заметить, что h_1^{-1} увеличивает длину регулярных выражений в постоянное число раз, зависящее от M. Аналогично, h_1^{-1} увеличивает длину выражений не более чем в $3\|\Delta_2\|$ раз, ибо если $h_1(b) = a$ ровно для k значений b, то $h_1^{-1}(a) = (b_1 + b_2 + \ldots + b_k)$ имеет длину $2k + 1$. Так как $1 \le k \le \|\Delta_2\|$, то $h_1^{-1}(a) \le 3\|\Delta_2\|$. Более того, должно быть ясно, что $\|\Delta_2\| \le |R_1|$, поскольку каждый символ из Δ_2 появляется в R_1. Следовательно, $h_1^{-1}\left(c_1c_2c_3\Delta_1^*\right)$ и $h_1^{-1}\left(\Delta_1\left(\Delta_1 - F\left(c_1c_2c_3\right)\right)\Delta_1\right)$ из (11.3) имеют длину $O(|R_1|)$. Наконец, $h_2^{-1}\left(R_1\right)$ и $(\Delta_1 \times \Delta_2)^*$, очевидно, имеют длину $O(|R_1|)$, где константа зависит только от M. Поэтому длина выражения (11.3), а следовательно, и расширенного регулярного выражения R, имеет порядок $O\left(|R_1| + |R_1'|\right)$.

Регулярные выражения, отражающие указанные нарушения структуры, строятся очень просто, поэтому оставляем это читателям. Изложенным способом можно построить такое расширенное регулярное выражение R_2, что его длина будет

линейной по $|R_1|+|R_1'|$ и оно будет содержать знак дополнения только тогда, когда его содержит R_1 или R_1'. □

Теорема 11.2. Любой алгоритм, выясняющий, представляет ли данное полурасширенное регулярное выражение[3] все строки в своем алфавите, имеет пространственную (и, следовательно, временную) сложность, которая для бесконечно многих n не меньше $c'c^{\sqrt{n/\log n}}$, где $c' > 0$ и $c > 1$ — константы.

Доказательство. Пусть L — произвольный язык с пространственной сложностью 2^n, но не $2^n/n$.[4] (Из теоремы 11.1 мы знаем, что такой язык существует.) Пусть M — детерминированная машина Тьюринга, допускающая язык L.

Предположим, у нас есть детерминированная машина Тьюринга M_0 с пространственной сложностью $f(n)$, выясняющая, пусто ли дополнение множества, представленного данным полурасширенным регулярным выражением. Тогда можно применить M_0 для распознавания языка L следующим образом. Пусть $w = a_1a_2...a_n$ — входная строка длины n.

1. Построим полурасширенное регулярное выражение R_1 для эталона длиной $2^n + 1$ или более. По лемме 11.1 (для $k = n$) выражение R_1 существует и имеет длину $O(n^2)$. Более того, чтобы найти выражение R_1, достаточно $O(n^2)$ памяти. Аналогично построим выражение R_1, представляющее $\neg x$, где x таково, что $R_1 = \texttt{CYCLE}(x\#)$. В силу леммы 11.2 выражение R_1' имеет длину $O(n^2)$, и, как легко видеть, его можно построить, заняв $O(n^2)$ ячеек памяти.
2. Построим полурасширенное регулярное выражение R_2, представляющее все неправильные вычисления машины M с эталоном R_1. В силу леммы 11.4 можно построить выражение R_2 так, чтобы его длина была не больше c_1n^2, где c_1 — константа, зависящая только от M.
3. Построим регулярное выражение R_3, представляющее все строки из $(\Delta_1 \times \Delta_2)^*$, которые не начинаются на верхней дорожке с $\#[q_0a_1]a_2...a_n\text{b...b}\#$, где q_0 — начальное состояние машины M. Очевидно, что существует такое выражение R_3 длины $O(n)$. Таким образом, $|R_2 + R_3| \le c_2n^2$ для некоторой константы c_2.
4. Применим M_0 к множеству, представленному выражением $R_2 + R_3$, чтобы выяснить, пусто ли его дополнение. Если оно не пустое, то существует пра-

---

[3] Предполагается кодировка, описанная в лемме 10.3.

[4] Выбор функции 2^n не имеет значения. Вместо 2^n можно было бы взять любую экспоненциальную функцию $f(n)$, а вместо $2^n/n$ — любую функцию, которая растет чуть медленнее, чем $f(n)$, при условии, что эти функции конструируемы по памяти.

вильное вычисление машины M с входной строкой w, так что w принадлежит языку L. В противном случае строка w не принадлежит языку L.

Можно построить детерминированную машину Тьюринга M' с пространственной сложностью $f(c_2 n^2 \log n)$, реализующую этот алгоритм распознавания L. Множитель $\log n$ в аргументе функции F появился из-за того, что регулярное выражение $R_2 + R_3$ над nсимвольным алфавитом необходимо закодировать строкой в фиксированном алфавите. Поскольку мы предположили, что алгоритм L имеет пространственную сложность 2^n, но не $2^n/n$, то неравенство $f(c_2 n^2 \log n) > 2^n/n$ должно выполняться по крайней мере для некоторых n. Однако если бы неравенство $f(c_2 n^2 \log n) > 2^n/n$ выполнялось лишь для конечного числа n, то можно было бы построить модифицированный вариант описанного выше алгоритма распознавания, который сначала проверял бы по конечной таблице, совпадает ли число $|w|$ с одним из тех n, для которых $f(c_2 n^2 \log n) > 2^n/n$, и если да, то выяснял бы, принадлежит ли строка w языку L. Таким образом, значение $f(c_2 n^2 \log n)$ должно превосходить $2^n/n$ для бесконечно многих n так, что $f(m) \geq 2^{c_3\sqrt{m/\log m}} \Big/ \sqrt{m/\log m} \geq c(c')^{\sqrt{m/\log m}}$ для бесконечно многих m и некоторых констант $c_3 > 0$, $c > 0$ и $c' > 1$. □

Следствие. Задача об эквивалентности полурасширенных регулярных выражений требует $c'c^{\sqrt{n/\log n}}$ памяти и времени при некоторых константах $c' > 0$ и $c > 1$ и бесконечно многих n.

Доказательство. Легко показать, что задача о пустоте дополнения полиномиально сводится к задаче эквивалентности, поскольку регулярное выражение, представляющее все строки, легко пишется и имеет небольшую длину. □

11.4. Неэлементарная задача

Рассмотрим весь класс расширенных регулярных выражений. Поскольку теперь у нас есть операция дополнения, достаточно рассматривать задачу о пустоте, а именно: пусто ли множество, представленное данным регулярным выражением? Мы увидим, что, располагая операцией дополнения, можно представлять регулярные множества еще более короткими выражениями и задача о пустоте для расширенных регулярных выражений значительно труднее задачи о пустоте дополнения для полурасширенных регулярных выражений.

Определим функцию $g(m, n)$ равенствами

1) $g(0, n) = n$,

2) $g(m, n) = 2^{g(m-1, n)}$ для $m > 0$.

Тогда $g(1, n) = 2^n$, $g(2, n) = 2^{2^n}$ и $g(m,n) = 2^{2^{\cdot^{\cdot^{2^n}}}}$,

где справа стоит стек из m двоек и последняя из них возводится в степень n. Функция $f(n)$ называется *элементарной*, если для некоторого m_0 она ограничена сверху функцией $g(m_0, n)$ для всех, кроме конечного числа, значений n.

С помощью методики, описанной в разделе 11.3, можно доказать, что задача о пустоте для всего класса расширенных регулярных выражений не может иметь пространственную сложность $S(n)$ ни для какой элементарной функции $S(n)$. Прежде чем приступить к доказательству, изменим немного определение правильного вычисления машины Тьюринга $M = (Q, T, I, \delta, \mathrm{b}, q_0, q_\mathrm{f})$ с эталоном CYCLE(x#). Удалим первый маркер # и заменим последний маркер новым символом \$ (рис. 11.2). Обозначим через C_i мгновенное описание (дополненное, если надо, пустыми символами, чтобы их длина стала равной длине строки x); как и ранее, $C_i \vdash C_{i+1}$ за один шаг машины M для $0 \le i < f$, C_0 — начальное мгновенное описание и C_f содержит состояние q_f.

Верхняя дорожка	C_0	#	C_1	#	...	#	C_f	\$
Нижняя дорожка	x	#	x	#	...	#	x	\$

Рис. 11.2. Новая форма правильных вычислений с эталоном

Будем использовать следующие обозначения (предполагаем, что $x \in \Sigma^*$):

1) $\Delta_1 = \mathrm{T} \cup (Q \times T) \cup \{\#, \$\}$ — алфавит верхней дорожки;
2) $\Delta_2 = \Sigma \cup \{\#, \$\}$ — алфавит нижней дорожки;
3) $h_1: \Delta_1 \times \Delta_2 \to \Delta_1$, причем $h_1([a, b]) = a$;
4) $h_2: \Delta_1 \times \Delta_2 \to \Delta_2$, причем $h_2([a, b]) = b$.

Лемма 11.5. Пусть R_1 — расширенное регулярное выражение для множества CYCLE(x#). Можно построить такое расширенное регулярное выражение R_2, представляющее множество циклических перестановок правильных вычислений машины Тьюринга M с эталоном CYCLE(x#), что $|R_2|$ есть $O(|R_1|)$, где константа зависит только от M.

Доказательство. Строка является циклической перестановкой правильного вычисления с эталоном CYCLE(x#) тогда и только тогда, когда

1) нижняя дорожка — циклическая перестановка строки вида $(x\#)^* x\$$;
2) верхняя дорожка — циклическая перестановка правильного вычисления машины M;
3) верхняя и нижняя дорожки циклически переставлены одинаково.

Пусть R_1' — выражение R_1, в котором вместо # стоит символ \$. Строки, удовлетворяющие условию 1, можно представить выражением $h_2^{-1}(U_1 \cap U_2 \cap U_3)$, где

$$U_1 = \Sigma^*(\#+\$)\left[(R_1 + R_1') \cap \left(\Sigma^*\#+\Sigma^*\$\right)\right]^* \Sigma^*,$$

$$U_2 = (\Sigma + \#)^* \ \$ \ (\Sigma + \#)^*,$$

$$U_3 = (R_1 + R_1')^*.$$

Подвыражение $\left[(R_1 + R_1') \cap \left(\Sigma^*\# + \Sigma^*\$\right)\right]$ в U_1 представляет две строки: $x\#$ и $x\$$. Поэтому выражение U_1 представляет $\Sigma^*(\# + \$)(x\# + x\$)^*\Sigma^*$. Выражение U_2 содержит строки с ровно одним вхождением \$. Поэтому всякая строка из $U_1 \cap U_2$ имеет вид

$$y_2\#x\#x\#...x\#x\$x\#...x\#y_1,$$

где y_1 и y_2 некоторые строки из Σ^*. Выражение U_3 приводит к тому, что $|y_2| + |y_1| = |x|$, откуда легко следует, что $y_1y_2 = x$ для строк из $U_1 \cap U_2 \cap U_3$. Таким образом, $S_1 = U_1 \cap U_2 \cap U_3$ представляет нижнюю дорожку всех строк, удовлетворяющих условию 1.

Что касается условия 2, то в лемме 11.4 было показано, как написать полурасширенное регулярное выражение для неправильных вычислений машины M с эталоном CYCLE($x\#$). Эта методика легко переносится на форму, показанную на рис. 11.2, и мы предлагаем читателям самим построить выражение E, представляющее циклические перестановки неправильных вычислений машины M с корректным эталоном на нижней дорожке. Применение операции дополнения к E дает расширенное регулярное выражение S_2. Это — единственное место, где применяется дополнение. Должно быть ясно, что все строки, представленные выражением $S_1 \cap S_2$, удовлетворяют условиям 1 и 2.

Выражение для условия 3 в предположении, что условия 1 и 2 выполнены, строится легко. Требуется лишь проверить, что один символ имеет символ \$ на обеих дорожках. Пересекая это выражение с $S_1 \cap S_2$, получаем искомое. □

Теперь покажем, как с помощью леммы 11.5 можно выразить все более и более длинные эталоны, не слишком увеличивая длины задающих их выражений.

Интуитивно мы строим регулярное выражение для какого-то эталона с небольшой длиной, скажем, длиной n. Затем используем его, чтобы построить новое регулярное выражение для всех циклических перестановок правильных вычислений с этим эталоном для некоторой машины Тьюринга, допускающей в точности одну строку длины n. Машина Тьюринга специально выбирается так, чтобы на входе длины n она делала по меньшей мере 2^{n+2} шагов, поэтому множество циклических перестановок ее правильных вычислений само будет эталоном дли-

ной 2^n или более. Повторяя этот процесс с новым эталоном, получаем эталоном все с бо́льшими и бо́льшими длинами, но не меньшими 2^n, 2^{2^n} и т.д. Пусть M_0 — конкретная одноленточная машина Тьюринга, которая ведет себя так:

1) проверяет, что ее входная лента начинается с последовательности символов a, для чего просматривает эту ленту до первого пустого символа;
2) если входная лента содержит строку из m символов a, за которыми стоит пустой символ, считает в двоичной системе от 0 до $2^{m+1}-1$ на части своей ленты, которая была занята символами a и следующим за ними пустым символом;
3) досчитав до $2^{m+1}-1$, останавливается и допускает свой вход.

Итак, машина M_0 обладает следующими свойствами. Для каждого n она допускает в точности одну строку длины n, а именно a_n. Она делает это, выполняя единственное правильное вычисление, состоящее не менее чем из 2^{n+2} шагов, поскольку половина ее шагов добавляет биты, а другая половина осуществляет переносы. Наконец, M_0 просматривает только $n+1$ ячеек ленты, когда ей подан вход длины n.

Исследуем теперь правильные вычисления машины M_0 с эталоном CYCLE(x#), имеющие вид, показанный на рис. 11.2. Если $|x| = n+1$, то существует единственное правильное вычисление машины M_0 с эталоном CYCLE(x#), у которого верхняя дорожка начинается с $[q_0a]a...a$b#. По лемме 11.5 для всех его циклических перестановок можно построить регулярное выражение R_2. Оно будет представлять множество, состоящее из циклических перестановок фиксированной строки $w \in (\Delta_1 \times \Delta_2)^*$. Строка $h_1(w)$ будет правильным вычислением машины M_0 на входе a^n, где, напомним, $|x| = n+1$. Так как M_0 делает на входе aa не менее 2^{n+2} шагов, R_2 можно взять в качестве эталона с длиной не менее 2^{n+2}.

Итак, пусть дан эталон CYCLE(x#), представленный выражением R_1, где $x \in \Sigma^*$. Тогда по выражению R_1 и описанной выше детерминированной машине Тьюринга M_0 можно построить эталон с длиной, не меньшей $2^{|x\#|}$, по лемме 11.5 для него найдется регулярное выражение R_2 длиной не более $c_1|R_1|$, где константа c_1 зависит только от M_0.

Лемма 11.6. Для всех $i \geq 1$ и $m \geq 2$ существует эталон с длиной, не меньшей $g(i, m)$[5], который можно представить расширенным регулярным выражением с длиной не более $c(c_1)^{i-1}m^2$, где c_1 — константа, зависящая от M_0 и введенная в предыдущем абзаце, а c — константа из леммы 11.1.

Доказательство. Докажем индукцией по i, что существует расширенное регулярное выражение R_i, представляющее CYCLE(x#) для некоторого x,

[5]Функция g определена в самом начале раздела.

где $|x\#| \geq g(i, m)$ и $|R_1| \leq c(c_1)^{i-1}m^2$. База ($i = 1$) следует из леммы 11.1 при $k = m$, поскольку можно построить расширенное регулярное выражение длины cm^2, представляющее CYCLE($x\#$).

Для шага индукции допустим, что построено расширенное регулярное выражение R_1 длины $|R_1| \leq c(c_1)^{i-2}m^2$, представляющее CYCLE($x\#$), где $|x\#| \geq g(i-1, m)$. Тогда на основе проведенного выше анализа, касающегося детерминированной машины Тьюринга M_0, заключаем, что можно построить регулярное выражение R длины не более $c_1|R_1| = c(c_1)^{i-1}m^2$, представляющее CYCLE($y\$$), где $|y\$| \geq 2^{|x\#|} \geq 2^{g(i-1, m)}g = (i, m)$. □

Теорема 11.3. Пусть $S(n)$ — произвольная элементарная функция. Тогда не существует детерминированная машина Тьюринга с пространственной (и, следовательно, временной) сложностью, ограниченной сверху функцией $S(n)$, которая могла бы установить, пусто ли множество, представленное произвольным расширенным регулярным выражением.

Доказательство. Допустим противное, т.е. что нашлась детерминированная машина Тьюринга M_1 с объемом памяти, ограниченной функцией $g(k_0, n)$, которая выясняет, пусто ли множество, представленное расширенным регулярным выражением. По теореме 11.1 существует язык L, допускаемый некоторой детерминированной машиной Тьюринга M с памятью, ограниченной функцией $g(k_0 + 1, n)$, но не допускаемый никакой детерминированной машиной Тьюринга с объемом памяти не больше $g(k_0 + 1, n)/n$. Предположив, что машина M_1 существует, можно построить детерминированную машину Тьюринга M_2, распознающую язык L следующим образом.

1. По заданной входной строке $w = a_1a_2...a_n$ длины n машина M_2 строит расширенное регулярное выражение R_1 с длиной d_1n^2 (где d_1 — константа, не зависящая от n), представляющее эталон с длиной, не меньшей $g(k_0 + 1, n)$. Для построения R_1 используется лемма 11.6.
2. Далее машина M_2 строит расширенное регулярное выражение R_2, представляющее правильные вычисления с эталоном R_1 для детерминированной машины Тьюринга M, которая допускает язык L и требует для этого не более $g(k_0 + 1, n)$ ячеек памяти. По лемме 11.5 можно добиться, чтобы было $|R_2| \leq d_2|R_1|$ для некоторой константы d_2.
3. Затем M_2 строит расширенное регулярное выражение R_3, представляющее правильные вычисления машины M с эталоном R_1 и начальным мгновенным описанием $C_0 = |q_0a_1]a_2...a_n$bb..., где q_0 — начальное состояние машины M, а именно

$$R_3 = R_2 \cap h^{-1}([q_0a_1]a_2...a_n\text{b}*\#)(\Delta_1 \times \Delta_2)^*,$$

где h_1 — гомоморфизм из леммы 11.5 и $\Delta_1 \times \Delta_2$ — алфавит выражения R_2. По лемме 11.3 $|R_3| \leq |R_2| + d_3 n$ для некоторой константы $d_3 > 0$. Поэтому

$$|R_3| \leq d_1 d_2 n^2 + d_3 n. \tag{11.4}$$

4. Наконец, M_2 кодирует выражение R_3 в фиксированном алфавите, как в теореме 11.2, и использует M_1 для проверки пустоты множества, представленного расширенным регулярным выражением R_3. Если оно пусто, то машина M_2 отвергает входную строку w, а если нет, то допускает ее. Таким образом, M_2 допускает язык L.

Теперь нетрудно видеть, что наибольшей памяти требует шаг 4. На этом шаге машине M_1 необходима память $g\left(k_0, |R_3|\log|R_3|\right)$, чтобы выяснить, пусто ли множество, представленное выражением R_3.

Следовательно, машине M_2 нужно столько же памяти. Учитывая границу (11.4) для $|R_3|$, заключаем, что M_2 имеет пространственную сложность $S(n) = d_4 g(k_0, d_5 n^2 \log n)$ для некоторых констант d_4 и d_5, поскольку для всех, кроме конечного числа, значений n первое слагаемое в правой части (11.4), а именно $d_1 d_2 n^2$, больше второго, т.е. $d_3 n$. Однако, рассуждая, как в приведенном доказательстве теоремы 11.2, можно показать, что пространственная сложность машины M_2 должна быть больше $g(k_0 + 1, n)/n$ для бесконечно многих n. Таким образом, если существует M_2, то

$$g(k_0 + 1, n)/n < d_4 g(k_0, d_5 n^2 \log n) \tag{11.5}$$

для бесконечно многих n. Однако независимо от выбора d_4 и d_5 лишь для конечного числа значений n справедливо (11.5) (это легко проверить). Итак, машины M_2 не существует, и, значит, не существует M_1. Поэтому проблема пустоты для расширенных регулярных выражений неразрешима никакой машиной Тьюринга с пространственной сложностью, ограниченной элементарной функцией. □

Упражнения

11.1. Функцию $T(n)$ называют *конструируемой по времени*, если некоторая детерминированная машина Тьюринга M при заданной входной строке длины n делает ровно $T(n)$ шагов до своей остановки. Покажите, что функции

а) n^2,

б) 2^n,

в) $n!$.

конструируемы по времени.

***11.2.** Покажите, что всякая функция, конструируемая по времени, конструируема по пространству, и если функция $S(n)$ конструируема по памяти, то $cS^{(n)}$ конструируема по времени для некоторого целого числа c.

***11.3.** Покажите, что если язык L допускается за время $T(n)$ какой-нибудь k-ленточной детерминированной (недетерминированной) машиной Тьюринга, то он допускается некоторой одноленточной детерминированной (недетерминированной) машиной Тьюринга с временной сложностью $O(T^2(n))$.

***11.4.** (*Иерархия по времени для детерминированной машины Тьюринга*) Покажите, что если функции $T_1(n)$ и $T_2(n)$ конструируемы по времени и

$$\inf_{n\to\infty} \frac{T_1^2(n)}{T_2(n)} = 0,$$

то существует язык, который допускается детерминированной машиной Тьюринга за время $T_2(n)$, но не $T_1(n)$. (*Подсказка*: В силу упражнения 11.3 достаточно выполнить диагонализацию по одноленточным детерминированным машинам Тьюринга с временной сложностью не выше $(T_1(n))^2$. Саму диагонализацию можно выполнить с помощью многоленточной детерминированной машиной Тьюринга.

В следующих двух упражнениях сформулированы слабые варианты результатов об иерархиях для недетерминированных машин Тьюринга.

****11.5.** Покажите, что для каждого целого числа $k \geq 1$ существует язык, допускаемый некоторой недетерминированной машиной Тьюринга с пространственной сложностью n^{k+1}, но не допускаемый никакой недетерминированной машиной Тьюринга с пространственной сложностью n^k.

****11.6.** Докажите тот же результат, что и в упражнении 11.5, для временной сложности.

В следующих двух упражнениях иерархия детерминированных машин Тьюринга по времени уплотняется.

****11.7.** Покажите, что всякий язык L, допускаемый некоторой k-ленточной детерминированной машиной Тьюринга с временной сложностью $T(n)$, допускается двухленточной детерминированной машиной Тьюринга с временной сложностью $O(T(n)\log T(n))$.

11.8. С помощью упражнения 11.7 докажите, что если функции $T_1(n)$ и $T_2(n)$ конструируемы по времени и

$$\inf_{n\to\infty} \frac{T_1(n)\log T_1(n)}{T_2(n)} = 0,$$

то некоторый язык допускается детерминированной машиной Тьюринга за время $T_2(n)$, но не $T_1(n)$.

***11.9.** Покажите, что если язык L допускается детерминированной (недетерминированной) машиной Тьюринга M с пространственной сложностью $S(n)$ и временной сложностью $T(n)$, а $c > 0$ — произвольная константа, то язык L допускается некоторой детерминированной (недетерминированной) машиной Тьюринга M' с пространственной сложностью $\mathrm{MAX}(cS(n), n + 1)$ и временной сложностью $\mathrm{MAX}(cT(n), 2n)$. (*Подсказка*: машина M' должна начинать свою работу со сжатия блоков ячеек машины M так, чтобы каждый такой блок помещался в одну клетку машины M'.)

11.10. Покажите, что если язык L допускается недетерминированной машиной Тьюринга с временной сложностью $T(n)$, то найдется такая константа c, что язык L допускается детерминированной машиной Тьюринга с временной сложностью $c^{T(n)}$.

***11.11.** Пусть T_1 и T_2 такие функции, что

$$\inf_{n\to\infty} \frac{T_1(n)}{T_2(n)} = 0.$$

Покажите, что существует язык, допускаемый машиной RAM за время $T_2(n)$, но не $T_1(n)$ (время определяется по логарифмическому весовому критерию).

11.12. Закончите доказательство леммы 11.2.

***11.13.** По данному расширенному регулярному выражению R_1 для множества $\mathrm{CYCLE}(x\#)$ постройте расширенное регулярное выражение R_2 для $\mathrm{CYCLE}((x\#)^*)$.[6] Длина выражения R_2 должна быть не больше длины выражения R_1, умноженной на константу.

****11.14.** Постройте недетерминированный алгоритм с пространственной сложностью $O(n)$, выясняющий, пусто ли множество, представленное произвольным полурасширенным регулярным выражением. Почему ваш алгоритм не срабатывает, когда вы пытаетесь узнать, представляет ли данное регулярное выражение все строки? Почему он должен не срабатывать?

11.15. Постройте детерминированный алгоритм, решающий за экспоненциальное время задачу пустоты дополнения для полурасширенных регулярных выражений.

---

[6] Результатом применения операции CYCLE к множеству строк служит объединение ее значений на каждой отдельной строке из этого множества.

11.16. Напишите регулярное выражение для нарушений структуры в доказательстве леммы 11.4.

11.17. Элементарна ли функция, определяемая равенствами $F(0) = 1$ и $F(n) = 2^{F(n-1)}$ для $n \geq 1$? (Эта функция введена в разделе 4.7.)

***11.18.** Постройте алгоритм, выясняющий, представляет ли произвольное расширенное регулярное выражение пустое множество. Какова временная и емкостная сложности вашего алгоритма?

****11.19.** Покажите, что задача о пустоте для расширенных регулярных выражений, использующих только знаки операций +, · и ¬, не является элементарной.

Проблемы для исследования

11.20. Естественная область исследований, подсказываемая материалом этой главы, — поиск практически важных задач, о которых можно доказать, что они трудноразрешимые. Работу в этом направлении проделали Fisher and Rabin [1974], исследовавшие сложность задач с элементарными арифметическими операциями; Cook and Reckhow [1974], изучившие процедуры доказательства теорем; Hunt [1974], рассмотревший ряд задач из теории языков, и некоторые другие авторы, упомянутые в библиографических замечаниях.

11.21. Интересно также, насколько плотнее могут быть иерархия детерминированных машин Тьюринга по времени и иерархия недетерминированных машин Тьюринга по времени и памяти по сравнению с иерархиями, указанными в упражнениях 11.5, 11.6 и 11.8. Например, существуют ли конструируемые по времени функции $T(n)$, для которых нет языков, распознаваемых за время $T(n)\log T(n)$, но не $T(n)$? Мы советуем читателям обратиться к работе Seiferas, Fisher, and Meyer [1973], где приводится самая плотная из известных иерархий для недетерминированной машины Тьюринга.

Библиографические замечания

Первое обширное исследование иерархий по памяти и времени для машин Тьюринга выполнили Hartmanis and Stearns [1965] и Hartmanis, Lewis, and Stearns [1965]. Статья Rabin [1963] была одной из первых работ о временной сложности, заслуживающей изучения. Улучшения иерархии по времени, данные в упражнениях 11.7 и 11.8, заимствованы у Hennie and Stearns [1966]. Иерархии недетерминированных машин Тьюринга по памяти взяты из Ibarra [1972], а по времени — из

Cook [1973]. Иерархия машин RAM в упражнении 11.11 принадлежит Cook and Reckhow [1973], упражнение 11.14 —Hopcroft, упражнение 11.19 — Meyer and Stockmeyer [1973].

Определение экспоненциальных нижних оценок для естественных задач началось с работ Meyer [1972] и Meyer and Stockmeyer [1972] . Теорема 11.2 о полурасширенных регулярных выражениях взята из Hunt [1973b], теорема 11.3 о расширенных регулярных выражениях — из Meyer and Stockmeyer [1973. Среди других работ о трудноразрешимых задачах заслуживают внимания Book [1972], Hunt [1973a, 1974], Stockmeyer and Meyer [1973], Meyer [1972], Hunt and and Rosenkrantz [1974], Rounds [1973], and Constable, Hunt, and Sahni [1974].

ГЛАВА 12

Нижние оценки числа арифметических операций

При разработке алгоритма для решения поставленной задачи возникает основной вопрос: какова ее истинная вычислительная сложность? Зная теоретическую нижнюю границу эффективности алгоритма, можно лучше оценить предлагаемые алгоритмы и объем усилий, которые следует затратить на поиск лучшего решения. Например, если известно, что задача является трудноразрешимой, то можно довольствоваться эвристическими способами нахождения приближенных решений.

К сожалению, выяснить истинную вычислительную сложность задачи обычно очень трудно. Для большинства практических задач качество алгоритма, как правило, оценивается на основе опыта. Однако иногда можно довольно точно оценить снизу количество арифметических операций, требуемых для выполнения вычислений. В этой главе мы изложим некоторые важные результаты такого рода. Например, покажем, что умножение матрицы $n \times p$ на p-вектор требует np умножений, а вычисление значения полинома n-й степени требует n умножений. Много дополнительных результатов о нижних оценках содержится в упражнениях. Читателям, интересующимся нижними оценками, советуем рассматривать эти упражнения как неотъемлемую часть данной главы.

12.1. Поля

Чтобы получить точные нижние границы, необходимо знать, какие элементарные операции разрешены. Для определенности будем предполагать, что все вычисления ведутся в некотором поле, таком, как поле действительных чисел, в котором основные операции — сложение и умножение элементов поля.

Определение. *Полем* называется такая алгебраическая система $(A, +, \cdot, 0, 1)$, что выполняются следующие условия.

1. Оно является кольцом с единицей относительно умножения.
2. Умножение коммутативно.
3. Каждый элемент a из $A - \{0\}$ обладает обратным элементом a^{-1} относительно умножения, т.е. $aa^{-1} = a^{-1}a = 1$.[1]

Пример 12.1. Действительные числа с обычными операциями сложения и умножения образуют поле. Целые числа образуют кольцо, но не поле, поскольку целые числа, не равные ±1, не имеют обратных, являющихся целыми числами. Однако для простого числа p целые числа по модулю p образуют (конечное) поле. □

Рассмотрим задачу вычисления произвольного полинома $p(x) = \sum_{i=0}^{n} a_i x^i$ в некоторой точке x. Мы хотим найти алгоритм, который по значениям a_i и x в качестве входных данных вычислял бы соответствующее значение $p(x)$ в качестве результата. Этот алгоритм должен работать на всех возможных значениях x, принадлежащих некоторому полю. Максимальное количество сложений, вычитаний и умножений, выполняемых алгоритмом, где максимум берется по всем допустимым входным значениям, называется *арифметической сложностью* этого алгоритма.

Заметим, что одни полиномы n-й степени вычислить оказывается проще, чем другие. Например, на вычисление $x^n + 2$ затрачивается порядка $\log n$ операций, хотя интуитивно мы чувствуем, что для вычисления произвольного полинома n-й степени должно требоваться линейное количество операций. Следовательно, алгоритм для вычисления значений, учитывающий свойства конкретного полинома, может работать намного быстрее, чем алгоритм, применимый ко всем полиномам. Чтобы сформулировать понятие алгоритма, пригодного для целого класса задач, введем понятие неопределенной переменной, которое будет использоваться для представления входных переменных.

Определение. *Неопределенной переменной* над алгебраической системой называется символ, не принадлежащий множеству элементов этой системы. Пусть $F = (A, +, \cdot, 0, 1)$ — поле и $x_1, x_2, ..., x_n$ — неопределенные переменные над F. *Расширением* $F[x_1, ..., x_n]$ поля F этими неопределенными переменными $x_1, x_2, ..., x_n$ называется такое наименьшее коммутативное[2] кольцо $(B, +, \cdot, 0, 1)$, что B содержит $A \cup \{x_1, x_2, ..., x_n\}$.

Заметим, что неопределенные переменные не связаны никакими неявными тождествами. Следовательно, любой полином с коэффициентами из F и неизвест-

[1] Как обычно, если это не приводит к недоразумениям, знак · опускается.

[2] Т.е. умножение в этом кольце является коммутативным.

ными $x_1, x_2, ..., x_n$ представляет некоторый элемент из кольца $F[x_1, x_2, ..., x_n]$. Два полинома обозначают один и тот же элемент из кольца $F[x_1, x_2, ..., x_n]$, если один из них можно преобразовать в другой с помощью законов коммутативного кольца. Единица в поле F является также единицей в кольце $F[x_1, x_2, ..., x_n]$.

Пример 12.2. Пусть F — поле действительных чисел. Тогда кольцо $F[x, y]$ включает в себя x, y и все действительные числа. Поскольку $F[x, y]$ замкнуто относительно сложения, то $x + y$ и $x + 4$ принадлежат $F[x, y]$. Поскольку $F[x, y]$ замкнуто относительно умножения, то это кольцо содержит элемент $(x + y)(x + 4)$, который в силу закона дистрибутивности для кольца эквивалентен элементу $x^2 + xy + 4x + 4y$. □

12.2. Еще раз о неветвящихся программах

Вернемся к вопросу: сколько арифметических операций требуется для вычисления значений произвольного полинома? Этот вопрос сводится к вопросу о количестве операций + и ·, необходимых для вычисления выражения $\sum_{i=0}^{n} a_i x^i$ (или эквивалентного ему) от неопределенных переменных $a_0, ..., a_n$ и x. Это мотивирует следующую модель вычислений, которая по существу является неветвящейся программой из раздела 1.5.

Определение. Пусть F — поле. *Вычисление* относительно F состоит из

1) множества формальных переменных, называемых *входами*;
2) множества *имен переменных*;
3) последовательности шагов вида $a \leftarrow b \theta c$, где θ — один из знаков +, − или *; a — переменная, не встречавшаяся ни на одном из предыдущих шагов; b и c — либо входы, либо элементы из поля F, либо имена переменных, появившиеся слева от стрелки на одном из предыдущих шагов.

Для краткости будем писать $a \leftarrow b$ вместо $a \leftarrow b + 0$ и $a \leftarrow -b$ вместо $a \leftarrow 0 - b$. Элемент поля F, участвующий в вычислении, называется *константой*.

Для определения результата вычисления нам нужно понятие *значения переменной* в вычислении. Говоря неформально, мы считаем, что вычисление выполняется шаг за шагом; на каждом шаге новой переменной присваивается элемент из кольца $F[x_1, ..., x_n]$, где $x_1, ..., x_n$ — входы. *Значение* $v(a)$ переменной или входа a определяется следующим образом. Если a — вход или элемент из F, то $v(a) = a$. Если a — переменная и $a \leftarrow b \theta c$ — шаг, в котором a стоит слева, то $v(a) = v(b) \theta v(c)$.

Говорят, что вычисление *вычисляет* множество E выражений из кольца $F[x_1, ..., x_n]$ относительно поля F, если для каждого выражения $e \in E$ в этом вычислении найдется такая переменная f, что $v(f) = e$.

Заметим, что вычисление рассматривается относительно заданного основного поля. Например, чтобы вычислить $x^2 + y^2$ относительно поля F действительных чисел, требуются два умножения, даже если не считать умножений на константу. Однако если F — поле комплексных чисел, то достаточно одного умножения (не считая умножений на константы), а именно $(x + iy)(x - iy)$. В качестве основного поля обычно берется поле действительных чисел, хотя можно было бы взять и поле комплексных или рациональных чисел или какое-нибудь конечное поле. Какое поле использовать, зависит от того, какие операции считаются элементарными. Если предполагается, что мы умеем представлять действительные числа и выполнять их сложение и умножение как основные операции, то в качестве F выбирается поле действительных чисел.

Пример 12.3. Вспомним, что вычисление произведения двух комплексных чисел $a + ib$ и $c + id$ относительно поля действительных чисел можно рассматривать как вычисление выражений $ac - bd$ и $ad + bc$. Очевидное вычисление выглядит так:

$$f_1 \leftarrow a * c$$
$$f_2 \leftarrow b * d$$
$$f_3 \leftarrow f_1 - f_2$$
$$f_4 \leftarrow a * d$$
$$f_5 \leftarrow b * c$$
$$f_6 \leftarrow f_4 + f_5$$

Значением f_1 является ac. Аналогично $v(f_2) = bd$ и $v(f_3) = ac - bd$. Таким образом, значение f_3 равно первому выражению. Значение f_6 равно $ad + bc$, т.е. второму выражению. Следовательно, это вычисление вычисляет произведение двух комплексных чисел.

Другое вычисление для комплексного умножения использует только три действительных умножения:

$$f_1 \leftarrow a + b$$
$$f_2 \leftarrow f_1 * c$$
$$f_3 \leftarrow d - c$$
$$f_4 \leftarrow a * f_3$$
$$f_6 \leftarrow d + c$$
$$f_7 \leftarrow b * f_6$$
$$f_8 \leftarrow f_2 - f_7$$

Легко показать, что $v(f_5) = ad + bc$ и $v(f_8) = ac - bd$. □

Должно быть очевидным, что данное вычисление вычисляет некоторое выражение тогда и только тогда, когда, подставляя вместо входов произвольные значения из поля F, мы получаем вычисление того частного случая выражения, который получается, если подставить эти входные значения вместо неопределенных переменных рассматриваемого выражения.

В оставшейся части главы нас будут интересовать прежде всего нижние оценки числа умножений, необходимых для вычисления данного множества выражений, поскольку в некоторых ситуациях умножения имеют гораздо большую сложность, чем сложения или вычитания.

Например, в главе 6 мы видели, что умножение двух матриц размером 2×2 за 7 арифметических умножений дает алгоритм умножения произвольных матриц за время $O_{\text{A}}\left(n^{\log 7}\right)$. Использование же 18 сложений в алгоритме Штрассена не влияет на асимптотику. При n, стремящемся к бесконечности, семь умножений матриц $(n/2) \times (n/2)$ на первом уровне рекурсии в алгоритме Штрассена становятся гораздо сложнее, чем 18 (или любое другое количество) сложений и вычитаний матриц того же размера.

На самом деле если бы можно было умножать две матрицы размером 2×2 с помощью только шести умножений в некоммутативном кольце, то существовал бы алгоритм умножения матриц со сложностью $O_{\text{A}}\left(n^{\log 6}\right) = O_{\text{A}}\left(n^{2,59}\right)$, независимо от того, сколько операций сложения и вычитания требуется для умножения матриц размером 2×2. (Можно, однако, показать, что для умножения матриц размером 2×2 с помощью такого алгоритма в произвольном кольце необходимы семь умножений; см. Hopcroft and Kerr [1971].)

Рассмотрим другой пример. В разделе 2.6 мы видели, что два n-разрядных двоичных целых числа можно умножить за $O_{\text{B}}(n^{1,59})$ шагов, потому что достаточно трех умножений, чтобы вычислить выражения ac, bd и $ad + bc$. Если бы можно было вычислять эти выражения с помощью только двух умножений, то выполнялось бы неравенство $M(n) \leq 2M(n/2) + cn$ для некоторой константы c. Такое вычисление, примененное рекурсивно, привело бы к алгоритму умножения целых чисел со сложностью $O_{\text{B}}\left(n \log n\right)$, который был бы лучше всех известных. К сожалению, как мы покажем, для вычисления этих выражений в любом поле требуется не менее трех умножений.

12.3. Матричная формулировка задач

Многие задачи можно сформулировать в терминах вычисления произведения матрицы на вектор-столбец. Элементы матрицы принадлежат $F[a_1, ..., a_n]$, где F — поле и $a_1, ..., a_n$ — неопределенные переменные. Компонентами вектора-столбца являются неопределенные переменные, отличные от $a_1, ..., a_n$.

Пример 12.4. Задачу умножения двух комплексных чисел $a + ib$ и $c + id$ можно записать так:

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac - bd \\ bc + ad \end{bmatrix}.$$

Здесь F — поле действительных чисел, а элементы матрицы принадлежат $F[a, b]$. Вектор состоит из формальных переменных c и d. □

Пример 12.5. Вычисление значений полинома $\sum_{i=0}^{n} a_i x^i$ можно выразить как

$$\begin{bmatrix} 1 & x & x^2 & \dots & x^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

Здесь F — поле действительных чисел, а элементы матрицы размером $1 \times (n + 1)$ принадлежат $F[x]$. □

12.4. Нижняя граница для количества умножений, ориентированная на строки

Определение. Пусть F — поле и $a_1, \dots, a_n$ — неопределенные переменные. Обозначим m-мерное пространство векторов с компонентами из $F[a_1, \dots, a_n]$ через $F^m[a_1, \dots, a_n]$, а через F^m — m-мерное пространство векторов с компонентами из F. Множество векторов $\{\mathbf{v}_1, \dots, \mathbf{v}_k\}$ из $F^m[a_1, \dots, a_n]$ называется *линейно независимым по модулю* F^m, если для любых $u_1, \dots, u_k$ из F соотношение $\sum_{i=1}^{k} u_i \mathbf{v}_i \in F^m$ влечет за собой, что все u_i равны нулю. Если векторы $\mathbf{v}_i$ не являются линейно независимыми по модулю F^m, то их называют *зависимыми по модулю* F^m.

Линейную независимость по модулю F^m можно интерпретировать иначе, рассмотрев фактор-пространство $F^m[a_1, \dots, a_n]/F^m$ классов эквивалентности векторов из $F^m[a_1, \dots, a_n]$. (Векторы $\mathbf{v}_1$ и $\mathbf{v}_2$ из $F^m[a_1, \dots, a_n]$ *эквивалентны* тогда и только тогда, когда разность $\mathbf{v}_1 - \mathbf{v}_2$ принадлежит F^m.) Линейная независимость по модулю F^m означает линейную независимость классов эквивалентности из $F^m[a_1, \dots, a_n]/F^m$.

Пример 12.6. Пусть F — поле действительных чисел, a и b — формальные переменные. Тогда три вектора

$$\mathbf{v}_1 = \begin{bmatrix} 2a \\ 4b \\ 2a-2b \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} -a \\ -b \\ b+3 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} \pi \\ b-1 \\ a \end{bmatrix}.$$

в $F^3[a, b]$ являются зависимыми по модулю F^3, поскольку

$$\frac{1}{2}\mathbf{v}_1 + \mathbf{v}_2 - \mathbf{v}_3 = \begin{bmatrix} -\pi \\ 1 \\ 3 \end{bmatrix}$$

принадлежат F^3.

С другой стороны, векторы

$$\mathbf{v}_1 = \begin{bmatrix} b \\ a \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} a \\ 0 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 0 \\ b \end{bmatrix},$$

являются линейно независимыми по модулю F^2. Допустим, что $\sum_{i=1}^{3} u_i \mathbf{v}_i$ принадлежит полю F^2. Тогда $u_1 b + u_2 a \in F$. Поскольку ни a, ни b не принадлежит F, то $u_1 = u_2 = 0$. Кроме того, соотношение $u_1 a + u_3 b \in F$ влечет за собой $u_1 = u_3 = 0$. Следовательно, векторы $\mathbf{v}_1$, $\mathbf{v}_2$ и $\mathbf{v}_3$ линейно независимы по модулю F^2. □

Пусть M обозначает матрицу размером $r \times p$ с элементами из поля F. Число линейно независимых строк в матрице M равно числу ее линейно независимых столбцов. Однако если элементы матрицы M принадлежат кольцу $F[a_1, ..., a_n]$, то число линейно независимых строк по модулю F^p, вообще говоря, не равно числу линейно независимых столбцов в матрице M по модулю F^r. Например, в матрице $[a_1\ a_2\ a_3]$ размером 1×3 одна строка линейно независима по модулю F^3 и три столбца линейно независимы по модулю F. *Рангом по строкам матрицы M по модулю F^p* называется число строк в матрице M, линейно независимых по модулю F^p. *Ранг по столбцам матрицы M по модулю F^r* определяется аналогично.

В оставшейся части раздела и в двух следующих мы будем употреблять такие обозначения:

1. F — поле.
2. $\{a_1, ..., a_n\}$ и $\{x_1, ..., x_p\}$ — непересекающиеся множества формальных переменных.
3. M — матрица размером $r \times p$ с элементами из $F[a_1, ..., a_n]$.
4. $\mathbf{x}$ — вектор-столбец $[x_1, ..., x_p]^T$.[3]

[3]Поскольку записывать векторы-столбцы в виде столбцов неудобно, мы будем часто записывать их в виде строк, как в главе 7, и обозначать транспонирование верхним индексом T.

Теорема 12.1. Пусть M — матрица с элементами из $F[a_1, ..., a_n]$ и $\mathbf{x}$ — вектор-столбец $[x_1 ... x_n]^T$. Предположим, что ранг по строкам матрицы M равен r. Тогда любое вычисление произведения $M\mathbf{x}$ требует не менее r умножений.

Доказательство. Предположим, не теряя общности, что матрица M содержит r строк. (Иначе можно вычеркнуть все строки матрицы M, кроме r линейно независимых, и получить некоторую матрицу M'. Любое вычисление произведения $M\mathbf{x}$ будет вычислением и для $M'\mathbf{x}$. Отсюда следует, что если $M'\mathbf{x}$ требует r умножений, то и $M\mathbf{x}$ требует r умножений.)

Допустим, что в данном вычислении произведения $M\mathbf{x}$ необходимы s умножений. Пусть $e_1, ..., e_s$ — выражения, вычисляемые на каждом из шагов с умножением. Тогда $M\mathbf{x}$ можно представить в виде линейной функции от $e_1, ..., e_s$ и неопределенных переменных, т.е.

$$M\mathbf{x} = N\mathbf{e} + \mathbf{f}, \tag{12.1}$$

где $\mathbf{e}$ — вектор $[e_1, ..., e_s]$, а компоненты вектора $\mathbf{f}$ представляют собой линейные функции только от $a_1, ..., a_n$ и $x_1, ..., x_p$. Все элементы матрицы N размером $r \times s$ принадлежат полю F.

Теперь допустим, что $r > s$. В силу элементарного факта линейной алгебры это означает, что строки матрицы N линейно зависимы.[4] Таким образом, существует такой вектор $\mathbf{y} \neq \mathbf{0}$[5] из F^r, что $\mathbf{y}^T N = \mathbf{0}^T$. Умножив (12.1) на $\mathbf{y}^T$, получим

$$\mathbf{y}^T M\mathbf{x} = \mathbf{y}^T N\mathbf{e} + \mathbf{y}^T\mathbf{f} = \mathbf{y}^T\mathbf{f}. \tag{12.2}$$

Из (12.2) видно, что произведение $(\mathbf{y}^T M)\mathbf{x}$ равно $\mathbf{y}^T\mathbf{f}$, т.е. является линейной функцией от формальных переменных, поскольку компоненты вектора $\mathbf{x}$ — различные неопределенные переменные, то вектор-строка $\mathbf{y}^T M$ должна состоять только из элементов поля F, а иначе в $\mathbf{y}^T M\mathbf{x}$ нашлось бы слагаемое, состоящее из произведения неопределенных переменных, и, значит, такое слагаемое было бы и в $\mathbf{y}^T\mathbf{f}$, а это противоречит тому, что $\mathbf{f}$ — линейная функция от неопределенных переменных. Поскольку $\mathbf{y} \neq \mathbf{0}$ и $\mathbf{y}^T M \in F^p$, то строки матрицы M зависимы по модулю F^p вопреки условию. Поэтому $s \geq r$ и теорема доказана. □

Пример 12.7. В разделе 12.2 мы рассмотрели вычисление трех выражений ac, bd и $ad + bc$ простым рекурсивным алгоритмом умножения целых чисел. Все три умножения можно записать в виде умножения матрицы на вектор, а именно

[4] Для читателей, незнакомых с этим фактом, укажем, что в основе доказательства этого результата лежит доказательство более трудной *основной теоремы* (леммы 12.1).

[5] $\mathbf{0}$ — это вектор подходящей размерности, состоящий из одних нулей.

$$\begin{bmatrix} a & 0 \\ 0 & b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}. \tag{12.3}$$

Пусть **R** — поле действительных чисел. В примере 12.6 было показано, что три строки матрицы в (12.3) линейно независимы по модулю $\mathbf{R}^2$, и, следовательно, любое вычисление этих трех выражений над вещественным полем требует трех умножений действительных чисел. □

12.5. Нижняя граница для количества умножений, ориентированная на столбцы

Теорема, аналогичная теореме 12.1, связанная с количеством линейно независимых столбцов, также верна, но она доказывается сложнее. Нам понадобится предварительный результат о множествах линейно независимых векторов.

Лемма 12.1. Пусть $\{\mathbf{v}_1, ..., \mathbf{v}_k\}$ — множество, состоящее из $k \geq 1$ m-мерных векторов с компонентами из $F[a_1, ..., a_n]$. Если в $\{\mathbf{v}_i | 1 \leq i \leq k\}$ существует подмножество из q векторов, линейно независимых по модулю F^m для любых элементов $b_2, ..., b_k$ из F, то $\left\{\mathbf{v}'_i \middle| \mathbf{v}'_i = \mathbf{v}_i + b_i\mathbf{v}_1, \quad 2 \leq i \leq k\right\}$ содержит подмножество, состоящее из $q - 1$ векторов, линейно независимых по модулю F^m.

Доказательство. Перенумеровав при необходимости векторы $\mathbf{v}_2, \mathbf{v}_3, ..., \mathbf{v}_k$, можно считать, что линейно независимы либо векторы из $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_q\}$, либо векторы из $\{\mathbf{v}_2, \mathbf{v}_3, ..., \mathbf{v}_{q+1}\}$.[6]

Вариант 1. Пусть множество $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_q\}$ состоит из линейно независимых векторов. Мы утверждаем, что векторы из множества $\left\{\mathbf{v}'_2, \mathbf{v}'_3, ..., \mathbf{v}'_q\right\}$ также линейно независимы. Чтобы доказать это, допустим, что $\sum_{i=2}^{q} c_i\mathbf{v}'_i \in F^m$ для некоторого множества элементов $c_i \in F$. Тогда $\sum_{i=1}^{q} c_i\mathbf{v}_i \in F^m$, где $c_1 = \sum_{i=2}^{q} c_i b_i$. Однако поскольку векторы $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_q\}$ линейно независимы, то $c_i = 0$ для $1 \leq i \leq q$. Следовательно, множество $\left\{\mathbf{v}'_2, \mathbf{v}'_3, ..., \mathbf{v}'_q\right\}$ также состоит из линейно независимых векторов.

Вариант 2. Пусть множество $\{\mathbf{v}_2, \mathbf{v}_3, ..., \mathbf{v}_{q+1}\}$ состоит из линейно независимых векторов. Если векторы из $\left\{\mathbf{v}'_2, \mathbf{v}'_3, ..., \mathbf{v}'_q\right\}$ линейно независимы, то

---

[6] На протяжении доказательства будет опускать выражение "по модулю F^m".

лемма верна, поэтому допустим, что это не так. Тогда найдутся такие элементы $c_2, ..., c_q \in F$, не все равные 0, что $\sum_{i=2}^{q} c_i \mathbf{v}'_i \in F^m$. Пусть $c_1 = \sum_{i=2}^{q} c_i b_i$. Тогда $\mathbf{w} = \sum_{i=1}^{q} c_i \mathbf{v}_i \in F^m$.

Будем считать, что $c_1 \neq 0$, поскольку в противном случае вектор $\sum_{i=2}^{q} c_i \mathbf{v}_i$ принадлежал бы F^m вопреки предположению, что векторы из $\{\mathbf{v}_2 ..., \mathbf{v}_{q+1}\}$ линейно независимы. Поэтому можно написать

$$\mathbf{v}_1 = c_1^{-1}\left(\mathbf{w} - \sum_{i=2}^{q} c_i \mathbf{v}_i\right).$$

Поскольку не все из $c_2, ..., c_q$ равны нулю, можно, не умаляя общности, считать, что $c_2 \neq 0$. Повторно применим наше рассуждение к множеству $\{\mathbf{v}'_3, \mathbf{v}'_4, ..., \mathbf{v}'_{q+1}\}$. Если это множество является линейно независимым, задача решена, поэтому предположим, что $\sum_{i=3}^{q+1} d_i \mathbf{v}'_i \in F^m$ — элементы из F^m, не все равные нулю. Пусть $d_1 = \sum_{i=3}^{q+1} d_i b_i$. Тогда $\mathbf{x} = d_1 \mathbf{v}_1 + \sum_{i=3}^{q+1} d_i \mathbf{v}_i \in F^m$. Если $d_1 = 0$, то получаем противоречие с линейной независимостью векторов из $\{\mathbf{v}_3, ..., \mathbf{v}_{q+1}\}$. Если $d_1 = 0$, можно написать

$$\mathbf{v}_1 = d_1^{-1}\left(\mathbf{x} - \sum_{i=3}^{q+1} d_i \mathbf{v}_i\right).$$

Приравняем два выражения для $\mathbf{v}_1$:

$$d_1^{-1}\left(\mathbf{x} - \sum_{i=3}^{q+1} d_i \mathbf{v}_i\right) = c_1^{-1}\left(\mathbf{w} - \sum_{i=2}^{q} c_i \mathbf{v}_i\right).$$

Умножая на $c_1 d_1$ и перегруппировывая члены, получаем

$$d_1 c_2 \mathbf{v}_2 + \sum_{i=3}^{q}\left(d_1 c_i - c_1 d_i\right)\mathbf{v}_i - c_1 d_{q+1}\mathbf{v}_{q+1} = d_1 \mathbf{w} - c_1 \mathbf{x}.$$

Поскольку $\mathbf{w}$ и $\mathbf{x}$ принадлежат F^m, то $d_1\mathbf{w} - c_1\mathbf{x}$ также принадлежит F^m, поскольку c_2 и d_1 не равны нулю, то векторы из $\{\mathbf{v}_2, \mathbf{v}_3, ..., \mathbf{v}_{q+1}\}$ являются линейно зависимыми. Полученное противоречие доказывает лемму. □

Пусть, как и ранее, M — матрица размером $r \times p$ с элементами из $F[a_1, ..., a_n]$. Покажем теперь, что если q столбцов матрицы M являются линейно независимыми по модулю F^r, $q \geq 1$, то любое вычисление вектора $M\mathbf{x}$, где $\mathbf{x} = [x_1, ..., x_p]^T$, требует не меньше q умножений.

Определение. Умножение называется *активным*, если в значение одного из его операндов входит одна из неопределенных переменных x_i, а значение другого не принадлежит полю F. Например, умножение $b * c$ активно, если $v(b) = 3 + a_2$ и $v(c) = x_1 + 2 * x_3$, но не активно, если $v(b) = 3$ и $v(c) = x_1 + 2 * x_3$ или $v(b) = 3 + a_2$ и $v(c) = a_1 + 2 * a_3$.

Теорема 12.2. Пусть $\mathbf{y}$ — r-мерный вектор с элементами из $F[a_1, ..., a_n]$. Если ранг по столбцам матрицы M равен q, то любое вычисление вектора $M\mathbf{x} + \mathbf{y}$ содержит не менее q активных умножений, где $q \geq 1$.

Доказательство. Проводится индукцией по q.

База. $q = 1$. Существует столбец матрицы M, не принадлежащий F^r, и, значит, элемент e матрицы M, принадлежащий $F[a_1, ..., a_n]$, но не F. Поэтому некоторый компонент вектора $M\mathbf{x}$ содержит произведение ex_j для некоторого j. Тогда то же верно и для $M\mathbf{x} + \mathbf{y}$. Вычисление без активных умножений может вычислять только выражения вида $P(a_1, ..., a_n) + L(x_1, ..., x_p)$, где P — полином и L — линейная функция, оба с коэффициентами из F. Таким образом, вычисление вектора $M\mathbf{x} + \mathbf{y}$ должно включать хотя бы одно активное умножение.

Шаг индукции. Допустим, что $q > 1$ и теорема верна для $q - 1$. Пусть C — вычисление вектора $M\mathbf{x} + \mathbf{y}$. По предположению индукции C содержит не менее $q - 1 \geq 1$ активных умножений. Пусть $f \leftarrow g * h$ — первое такое умножение. Тогда без потери общности можно считать, что

$$v(g) = P(a_1, ..., a_n) + \sum_{i=1}^{p} c_i x_i, \tag{12.4}$$

где P — полином с коэффициентами из F и все c_i также принадлежат F. Более того, не умаляя общности, можно считать, что $c_1 \neq 0$.

Наша стратегия состоит в том, чтобы построить из C и множества выражений $M\mathbf{x} + \mathbf{y}$ новое множество выражений $M'\mathbf{x}' + \mathbf{y}'$ с вычислением C', содержащим на одно активное умножение меньше, чем C. Кроме того, $q - 1$ столбцов матрицы M' будут линейно независимыми по модулю F^r. Тогда по предположению индукции C' будет содержать не менее $q - 1$ активных умножений, а значит, C — не менее q активных умножений.

Конкретно мы заменим x_1 в C некоторым выражением e, которое сделает g из (12.4) равным нулю. Более того, выражение e будет вычислимо без активных умножений. Вычисление C' начинается с вычисления выражения e, при этом значение e присваивается формальной переменной x_1 (она уже больше не будет входной переменной). Остальная часть вычисления C' состоит из C, где $f \leftarrow g * h$ заменено на $f \leftarrow 0$. Затем мы покажем, что множество выражений, вычисляемых с помо-

щью C', можно выразить в виде $M'\mathbf{x}' + \mathbf{y}'$, где $q - 1$ столбцов матрицы M' линейно независимы по модулю F^r.

Итак, приступим к изложению деталей доказательства. Из (12.4) и предположения $c_1 \neq 0$ заключаем, что $g = 0$, если

$$x_1 = -c_1^{-1}\left[P(a_1, \ldots, a_n) + \sum_{i=2}^{p} c_i x_i\right]. \tag{12.5}$$

Правая часть равенства (12.5) равна выражению e, упомянутому выше. Вычисление C', получаемое из C описанным выше способом, вычисляет $M\mathbf{x} + \mathbf{y}$, куда вместо x_1 подставлено выражение e. Поэтому $M'\mathbf{x}' + \mathbf{y}'$ можно записать как

$$M\begin{bmatrix} e \\ x_2 \\ x_3 \\ \vdots \\ x_p \end{bmatrix} + \mathbf{y},$$

а затем переписать в виде

$$M\begin{bmatrix} -c_1^{-1}\sum_{i=2}^{p} c_i x_i \\ x_2 \\ x_3 \\ \vdots \\ x_p \end{bmatrix} + M\begin{bmatrix} -c_1^{-1}P(a_1, \ldots, a_n) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \mathbf{y}.$$

Рассмотрим первое слагаемое в (12.6). Пусть $\mathbf{m}_i$ — это i-й столбец матрицы M. Для $2 \le i \le p$ положим $\mathbf{m}'_i = \mathbf{m}_i - (c_i/c_1)\mathbf{m}_i$. Пусть M' — матрица, состоящая из $p - 1$ столбцов, в которой i-й столбец равен $\mathbf{m}'_{i+1}$, и пусть $\mathbf{x}' = [x_2, \ldots, x_p]^T$. Таким образом, первое слагаемое в (12.6) равно $M'\mathbf{x}'$.

Второе слагаемое в (12.6) — это вектор с компонентами из $F[a_1, \ldots, a_n]$, поскольку элементы матрицы M принадлежат $F[a_1, \ldots, a_n]$. Поэтому второе и третье слагаемые в (12.6) можно объединить в новый вектор $\mathbf{y}'$ с компонентами из $F[a_1, \ldots, a_n]$. Таким образом, C' вычисляет $M'\mathbf{x}' + \mathbf{y}'$. Из леммы 12.1 следует, что не менее $q - 1$ столбцов матрицы M' линейно независимы по модулю F^r. Значит, C' содержит не менее $q - 1$ активных умножений, и поэтому C содержит не менее q активных умножений. □

Приведем два примера применения теоремы 12.2.

Пример 12.8. Мы утверждаем, что умножение матрицы A размером $n \times p$ на p-мерный вектор $\mathbf{v}$ требует np умножений элементов. Формально пусть a_{ij} и v_j —неопределенные переменные, $1 \le i \le n$, $1 \le j \le p$. Тогда произведение $A\mathbf{v}$ матрицы на вектор имеет вид, показанный на рис. 12.1.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{np} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \end{bmatrix}$$

Рис. 12.1. Общий вид произведения матрицы на вектор

Непосредственно применяя теоремы 12.1 и 12.2 к $A\mathbf{v}$, заключаем лишь, что требуется $\mathrm{MAX}(n, p)$ умножений. Однако произведение матрицы на вектор можно также выразить в виде $M\mathbf{x}$, где в i-й строке матрицы M в столбцах с $(i-1)p+1$ по ip стоят $v_1, \dots, v_p$, а в остальных столбцах — нули. Вектор $\mathbf{x}$ — это вектор-столбец, содержащий строки матрицы A, равный конкатенации строк матрицы A, записанной сверху вниз. Матрица M и вектор $\mathbf{x}$ изображены на рис. 12.2.

$$\begin{matrix} n \text{ строк} \end{matrix} \begin{bmatrix} v_1 & v_2 & \dots & v_p & 0 & 0 & & 0 & \dots & 0 & 0 & & 0 \\ 0 & 0 & & 0 & v_1 & v_2 & \dots & v_p & & 0 & 0 & & 0 \\ \vdots & \vdots & & & & & & & & & & & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & \dots & v_1 & v_2 & \dots & v_p \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{1p} \\ a_{21} \\ a_{22} \\ \vdots \\ a_{2p} \\ \vdots \\ a_{n1} \\ a_{n2} \\ \vdots \\ a_{np} \end{bmatrix}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{np \text{ столбцов}}$$

Рис. 12.2. Эквивалентная форма произведения матрицы на вектор

Легко показать, что столбцы матрицы M линейно независимы по модулю F^n. Поэтому в силу теоремы 12.2 любое вычисление для $A\mathbf{v}$ требует не менее np умножений.

Пример 12.9. Рассмотрим задачу вычисления полинома n-й степени $\sum_{i=0}^{n} a_i x^i$. Эту задачу можно выразить через умножение матрицы M на вектор $\mathbf{x}$:

$$\begin{bmatrix} 1 & x & x^2 & \dots & x^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

Элементы матрицы M принадлежат $F[x]$ и $\mathbf{x} = [a_0, a_1, \dots, a_n]^T$. Легко показать, что все столбцы матрицы M, кроме первого, являются линейно независимыми по модулю F. Следовательно, в M содержатся n линейно независимых столбцов, и вычисление произвольного полинома n-й степени требует не менее n умножений.

Правило Горнера, вычисляющее полином по схеме

$$(\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0,$$

требует ровно n умножений и, значит, оптимально в том смысле, что использует наименьшее возможное число умножений. Аналогично можно показать, что для вычисления $\sum_{i=0}^{n} a_i x^i$ требуется n сложений и вычитаний. Следовательно, правило Горнера использует наименьшее число арифметических операций, необходимых для вычисления значения полинома в одной точке. □

12.6. Граница для количества умножений, ориентированная на строки и столбцы

Объединим теоремы 12.1 и 12.2, чтобы доказать более сильный результат, чем тот, который получается, если рассматривать независимость строк и столбцов отдельно. Пусть M — матрица размером $r \times p$ с элементами из $F[a_1, \dots, a_n]$, как и ранее, и $\mathbf{x} = [x_1, \dots, x_p]^T$.

Теорема 12.3. Пусть матрица M содержит такую подматрицу S с q строками и c столбцами, что для любых векторов $\mathbf{u}$ и $\mathbf{v}$ из F^q и F^c соответственно элемент $\mathbf{u}^T S \mathbf{v}$ принадлежит полю F тогда и только тогда, когда $\mathbf{u} = \mathbf{0}$ или $\mathbf{v} = \mathbf{0}$. Тогда любое вычисление произведения $M\mathbf{x}$ требует не менее $q + c - 1$ умножений.

Доказательство. Не теряя общности, с самого начала будем считать, что в матрице M только q строк, а подматрица S состоит из ее первых c столбцов. Допустим, что $M\mathbf{x}$ можно вычислить за s умножений. Пусть $\mathbf{e}$ — вектор, компонентами которого являются s выражений, вычисляемых этими умножениями. Предположим, кроме того, что i-й компонент вектора $\mathbf{e}$ вычислен раньше j-го для $i < j$. Тогда, как в теореме 12.1, можно написать

$$M\mathbf{x} = N\mathbf{e} + \mathbf{f}, \tag{12.7}$$

где N — матрица размером $q \times s$ с элементами из F и $\mathbf{f}$ — вектор, компоненты которого представляют собой линейные комбинации векторов $a_1, ..., a_n$ и $x_1, ..., x_p$.

Как и в теореме 12.1, можно прийти к выводу, что $s \geq q$. Если бы это было не так, то нашелся бы такой вектор $\mathbf{y} \neq \mathbf{0}$ из F^q, что $\mathbf{y}^T N = \mathbf{0}^T$, откуда следовало бы, что компоненты вектора $\mathbf{y}^T M$ принадлежат F. Тогда компоненты вектора $\mathbf{y}^T S$ принадлежали бы F вопреки условию (возьмите $\mathbf{u} = \mathbf{v}$ и $\mathbf{v}^T = [1, ..., 1]$).

Поскольку выполняется условие $s \geq q$, матрицу N можно разбить на две матрицы N_1 и N_2, где N_1 состоит из первых $s - q + 1$ столбцов матрицы N, а N_2 — из ее последних $q - 1$ столбцов. Далее, пусть $\mathbf{e}_1$ и $\mathbf{e}_2$ — векторы, составленные из первых $s - q + 1$ и последних $q - 1$ элементов вектора $\mathbf{e}$ соответственно. Тогда (12.7) можно записать в виде

$$M\mathbf{x} = N_1\mathbf{e}_1 + N_2\mathbf{e}_2 + \mathbf{f}. \tag{12.8}$$

Так как матрица N_2 имеет размер $q \times (q - 1)$, то найдется такой вектор $\mathbf{y} \neq \mathbf{0}$ из F^q, что $\mathbf{y}^T N_2 = \mathbf{0}^T$. Умножим (12.8) на $\mathbf{y}^T$:

$$\mathbf{y}^T M\mathbf{x} = \mathbf{y}^T N_1\mathbf{e}_1 + \mathbf{y}^T\mathbf{f}. \tag{12.9}$$

Положим $M' = \mathbf{y}^T M$. Заметим, что M' — это матрица размером $1 \times p$, равная линейной комбинации строк матрицы M. Поскольку произведения, входящие в векторы $\mathbf{e}_1$, можно вычислять, не обращаясь к произведениям из $\mathbf{e}_2$ (мы предположили, что первые вычисляются раньше вторых), то очевидно, что выражение $M'\mathbf{x}$ можно вычислить с помощью (12.9) за $s - q + 1$ умножений. Если теперь мы сможем показать, что c столбцов матрицы M' линейно независимы по модулю F, то в силу теоремы 12.2 сможем утверждать, что $s - q + 1 \geq c$. Тогда $s \geq q + c - 1$, что и требовалось доказать.

Итак, покажем, что первые c столбцов матрицы $M' = \mathbf{y}^T M$ линейно независимы по модулю F. Пусть $\mathbf{y}^T = [y_1, ..., y_q]$. Первые c элементов матрицы M' имеют вид $\sum_{i=1}^{q} y_i M_{ij}$ для $1 \leq j \leq c$, где M_{ij} — это (i, j)-й элемент матрицы M. Допустим, что нашелся такой вектор $\mathbf{z} \neq \mathbf{0}$ с компонентами $z_1, ..., z_c$ из F, что $\sum_{j=1}^{c} z_j \sum_{i=1}^{q} y_i M_{ij}$ принадлежит F, т.е. первые c столбцов матрицы M' являются зависимыми по модулю F. Тогда элемент $\mathbf{y}^T S\mathbf{z}$ оказался бы в F вопреки условию теоремы относительно S.

Следовательно, матрица M' содержит c линейно независимых по модулю F столбцов, и теорема доказана. □

Применим теорему 12.3 к умножению двух комплексных чисел $a + ib$ и $c + id$, чтобы показать, что оно требует трех умножений действительных чисел. Заметим, что каждая из теорем 12.1 и 12.2 в отдельности не достаточно сильна для этого.

Пример 12.10. Рассмотрим задачу умножения двух комплексных чисел $a + ib$ и $c + id$, а именно вычисления

$$M\mathbf{x} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}.$$

Пусть матрица S совпадает с матрицей M, а F — поле действительных чисел и векторы

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \neq \mathbf{0} \quad \text{и} \quad \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \neq \mathbf{0}$$

принадлежат F^2. Допустим, что

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

является элементом поля F. Это произведение равно $y_1z_1a + y_2z_1b + y_2z_2a - y_1z_2b$. Если оно принадлежит полю F, то коэффициенты при a и b должны равняться нулю. Следовательно,

$$y_1z_1 + y_2z_2 = 0 \tag{12.10}$$

и

$$y_2z_1 - y_1z_2 = 0. \tag{12.11}$$

Допустим, что $y_1 \neq 0$. Тогда из (12.11) получаем $z_2 = y_2z_1/y_1$. Подставив это в (12.10) и умножив на y_1, находим, что $\left(y_1^2 + y_2^2\right)z_1 = 0$. Поскольку $y_1 \neq 0$, то $y_1^2 + y_2^2 \neq 0$, значит, $z_1 = 0$.[7] Тогда в силу (12.11) $z_2 = 0$. Однако равенства $z_1 = z_2 = 0$ противоречат условию

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \neq \mathbf{0}.$$

Теперь допустим, что $y_1 = 0$. Если $y_2 = 0$, то противоречие возникает сразу. Если $y_2 \neq 0$, то, поменяв местами y_1 и y_2, получаем, что $z_1 = z_2 = 0$, что снова приводит к противоречию.

Таким образом, теорема 12.3 применима к $M\mathbf{x}$ при $q = c = 2$, так что три действительных умножения необходимы. Программа в примере 12.3 показывает, что три умножения, к тому же, достаточны. □

[7] Здесь предположение, что F — поле действительных чисел, является существенным.

Теоремы 12.1–12.3 можно обобщить на случай, когда в вычислении допускаются деления. В этом случае необходимо допускать такие шаги в вычислении, которые при некоторых значениях входов приводят к делению на нуль. Следовательно, теория, рассматривающая мультипликативные операции (т.е. умножение и деление), предполагает, что поле F бесконечно. Если бы оно было конечным, то мы могли бы столкнуться с вычислениями, не работающими ни для каких входов из-за того, что любой возможный вход приводит к делению на нуль.

С учетом этих изменений каждую из теорем 12.1–12.3 можно перенести на случай обеих мультипликативных операций вместо одного умножения. В теореме 12.2 определение активной мультипликативной операции $f \leftarrow g * h$ или $f \leftarrow g/h$ должно быть следующим: или значение одной из переменных g и h включает в себя одну из формальных переменных x_i, а другой операнд не принадлежит F, или g принадлежит F, h включает в себя одну из формальных переменных x_i и операцией является деление.

12.7. Предобусловливание

В примере 12.9 мы показали, что вычисление значения полинома n-й степени в одной точке требует n умножений. Однако мы исходили из предположения, что полином представлен своими коэффициентами. В этом разделе мы изучим задачу минимизации числа умножений, необходимых для вычисления одного полинома, когда разрешается представлять полином любым множеством параметров, вычисляемых по коэффициентам. Если бы нам было нужно вычислять данный полином несколько раз, то разумно было бы потратить некоторое время на вычисление другого представления полинома, при условии, что новое представление даст более быстрое вычисление. Такое изменение представления называется предобусловливанием.[8] В примере 12.9 умножение было активным, если один из сомножителей включал в себя коэффициенты a_0, a_1, ..., a_n, а другой не принадлежал F. Таким образом, умножение, в котором оба сомножителя включали коэффициенты и не включали переменную x, учитывалось. В случае предобусловливания мы бесплатно получаем все умножения, в которых ни один сомножитель не содержит переменной.

Определение. *Степенью* полинома от нескольких переменных называют наибольшую степень переменных этого полинома. Например, степень полинома $P(x, y) = x^3 + x^2y^2$ равна 3.

[8]Следует отметить различие между этой ситуацией и ситуацией из раздела 8.5, где были известны все точки, в которых нужно вычислять значения полинома. Здесь же перед вычислением значений мы можем работать только с коэффициентами, а вычисления значений производятся по одному за раз, т.е. в онлайн-, а не офлайн-режиме.

Следующая лемма утверждает, что для любых $v + 1$ полиномов от v переменных существует нетривиальный полином от $v + 1$ переменных, тождественно равный нулю на данных полиномах. Например, пусть $p_1 = x^2$ и $p_2 = x + x^3$. Тогда $p_1 + 2p_1^2 + p_1^3 - p_2^2 = 0$ для всех x.

Лемма 12.2. Пусть $p_i(x_1, ..., x_r)$, $1 \le i \le n$ — полиномы. Если $n > r$, то существует полином $g(y_1, ..., y_n)$, не все коэффициенты которого равны нулю, тождественно равный нулю как функция от x_i.

Доказательство. Пусть S_d — множество всех полиномов вида $p_1^{i_1} p_2^{i_2} ... p_n^{i_n}$, где $0 \le i_j \le d$ для всех j. Заметим, что $||S_d|| = (d + 1)^n$. Пусть m — наибольшая из степеней полиномов p_i. Тогда каждый полином $q \in S_d$ — это полином от $x_1, x_2, ..., x_r$ степени, не большей dmn.

Любой полином степени dmn от r переменных можно представить вектором длины $(dmn + 1)^r$. Компонентами этого вектора являются коэффициенты при описанных выше членах, расположенных в фиксированном порядке. Следовательно, можно говорить о линейной независимости полиномов и, в частности, утверждать, что по основной теореме линейной алгебры любое множество, состоящее из $(dmn + 1)^r + 1$ полиномов из S_d, должно быть линейно зависимым, т.е. некоторая их нетривиальная сумма должна равняться нулю.

В частности, если m, n и r фиксированы и $n > r$, то нетрудно показать, что $(d + 1)^n \ge (dmn + 1)^r$ для достаточно большого d, поскольку $(d + 1)^n$ — полином более высокой степени по d, чем $(dmn + 1)^r$. Следовательно, найдется такое d, что нетривиальная сумма полиномов из S_d будет тождественно равна нулю как функция x. Эта сумма и будет искомым полиномом g. □

С помощью леммы 12.2 покажем, что любое множество параметров полинома, через которое его коэффициенты выражаются в виде полиномиальных функций, должно содержать $n + 1$ параметров, чтобы представлять произвольный полином n-й степени.

Лемма 12.3. Пусть M — взаимно однозначное отображение n-мерного векторного пространства C^n коэффициентов на r-мерное векторное пространство D^r параметров. Если M^{-1} таково, что каждый компонент вектора в C^n является полиномиальной функцией компонентов соответствующего вектора в D^r, то $r \ge n$.

Доказательство. Пусть $r < n$ и M^{-1} задается равенствами $c_i = p_i(d_1, ..., d_r)$, где p_i — полиномы, d_i — параметры и c_i — коэффициенты. В силу леммы 12.2 найдется такой нетривиальный полином g, что $g(c_1, ..., c_n)$ — тождественный нуль

для всех d_i.[9] Однако поскольку сам полином g не равен тождественно нулю, то найдутся такие полиномы с коэффициентами c_1, ..., c_n, что $g(c_1, ..., c_n) \neq 0$. Эти полиномы нельзя представить в терминах параметров d_i, что приводит к противоречию. □

Теперь покажем, что вычисление значения полинома n-й степени в одной точке даже при наличии предобусловливания требует не менее $n/2$ умножений.

Теорема 12.4. Вычисление значения произвольного полинома n-й степени в одной точке требует не менее $n/2$ умножений, даже если не считать входящие в него умножения, которые включают в себя только коэффициенты полинома.

Доказательство. Допустим, что существует вычисление с m умножениями, включающими в себя неизвестную переменную x. Пусть результаты этих умножений представлены выражениями $f_1, f_2, ..., f_m$. Тогда каждое выражение f_i можно записать в виде

$$f_i = [L_{2i-1}(f_1, ..., f_{i-1}, x) + \beta_{2i-1}] * [L_{2i}(f_1, ..., f_{i-1}, x) + \beta_{2i}], \; 1 \leq i \leq m,$$

где каждая функция L_i является линейной по f и x, а каждая функция β_i является полиномиальной по коэффициентам. Полином $P(x)$, который вычисляется, можно записать в виде

$$P(x) = L_{2m+1}(f_1, ..., f_m, x) + \beta_{2m+1}.$$

Следовательно, $P(x)$ можно представить новым множеством из $2m + 1$ параметров, а именно множеством функций β_i. Более того, коэффициенты полинома $P(x)$ — полиномиальные функции параметров β_i. По лемме 12.3 $2m + 1 \geq n + 1$. Таким образом, $m \geq n/2$. □

Теорема 12.4 дает нижнюю оценку числа умножений, необходимых для вычисления значения полинома в одной точке, когда проводится предобусловливание. Значительный интерес вызывает проблема представления полинома, при котором его можно вычислить за $n/2$ умножений. Однако необходимо не только находить параметры для такого представления полинома, при котором легко вычисляются его значения, но и уметь быстро вычислять эти параметры по коэффициентам полинома. Таким образом, хотелось бы, чтобы параметры были рациональными функциями коэффициентов. Методика нахождения таких параметров приведена в упражнениях.

---

[9] Заметим, что g — полином по c_i, а, значит, по d_i, поскольку c — полиномы по параметрам d_i.

Упражнения

12.1. Пусть F — поле и x — неопределенная переменная. Покажите, что множество полиномов от x с коэффициентами из F образует коммутативное кольцо $F[x]$ и единица в F является единицей в $F[x]$.

12.2. Покажите, что любое вычисление для выражений

$$ac + bd,$$
$$ac - bd,$$
$$bc + ad,$$
$$bc - ad$$

требует не менее четырех умножений, где a, b, c, d — элементы поля F.

12.3. Покажите, что любое вычисление произведения вектор-столбца на вектор-строку

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}$$

требует не менее mn умножений.

12.4. Полином n-й степени от двух переменных x и y имеет общий вид $\sum_{i=0}^{n}\sum_{j=0}^{n} a_{ij}x^i y^j$. Покажите, что для вычисления значения этого полинома с предобуславливанием необходимо и достаточно $(n + 1)(n + 2)/2 - 1$ умножений.

12.5. Покажите, что умножение тёплицевой матрицы размером 2×2 (см. упражнения в главе 6) на вектор, т.е.

$$\begin{bmatrix} b & c \\ a & d \end{bmatrix} \begin{bmatrix} d \\ e \end{bmatrix}$$

требует трех умножений.

12.6. Обобщите упражнение 12.5, показав, что для умножения тёплицевой матрицы размером $n \times n$ на вектор необходимо $2n - 1$ умножений. Как близко к этой границе можно подойти при $n = 3$?

***12.7.** Можно обобщить алгоритм умножения из раздела 2.6, разбивая каждый из сомножителей на три части и затем вычисляя произведение матрицы на вектор

$$\begin{bmatrix} a & 0 & 0 \\ b & a & 0 \\ c & b & a \\ 0 & c & b \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

за минимально возможное число умножений. Сколько умножений требуется для решения этой задачи? Приводит ли этот подход к улучшению алгоритма из раздела 2.6?

**12.8. Пусть F — поле и a_1, ..., a_n и x_1, ..., x_p — непересекающиеся множества неопределенных переменных. Пусть $\mathbf{x} = [x_1, ..., x_p]^T$, а M — матрица размером $r \times p$ с элементами из $F[a_1, ..., a_n]$, у которой q столбцов линейно независимы по модулю F^r. Покажите, что при предобусловливании вектора $\mathbf{x}$, т.е. когда не учитываются произведения, включающие только x_i, для вычисления произведения $M\mathbf{x}$ все равно требуется $q/2$ умножений.

**12.9. Пусть ровно k выражений из множества P, никакая линейная комбинация элементов которого не равна константе, представляют собой одиночные умножения (например, они имеют вид $a * b$, где a и b —неопределенные переменные). Допустим, что q умножений достаточно, чтобы вычислить все выражения из P. Покажите, что существует вычисление для P с q умножениями, k из которых совпадают с теми выражениями из P, которые можно вычислить с помощью одного умножения.

*12.10. Покажите, что для вычисления выражений ac и $bc + ad$ требуется не менее трех умножений. (*Подсказка*: воспользуйтесь упражнением 12.9.)

*12.11. Покажите, что для вычисления множества из $2k$ выражений a_ic_i и $b_ic_i + a_id_i$, $1 \le i \le k$, требуется не менее $3k$ умножений.

Упражнения 12.12–12.14 связаны с умножением двух матриц A и B размером 2×2. В частности, мы хотим вычислить четыре выражения, входящие в произведение матрицы AB, используя некоммутативные операции умножения:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}a_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

**12.12. Покажите, что если существует вычисление над произвольным кольцом, содержащее q операций умножения, то существует и другое вычисление над кольцом по модулю 2, использующее q операций умножения, в котором все умножения имеют вид

$$\left(a_{i_1j_1} + a_{i_2j_2} + ... + a_{i_kj_k}\right) * \left(b_{m_1n_1} + b_{m_2n_2} + ... + b_{m_rn_r}\right).$$

***12.13.** Покажите, что если левая часть любого умножения, описанного в предыдущем упражнении содержит только один элемент a_{ij} (т.е. $k = 1$), то это вычисление содержит не менее семи умножений. (*Подсказка*: положите $a_{ij} = 0$ и используйте упражнение 12.11 с $k = 2$.)

****12.14.** Покажите, что если левая часть не содержит только один элемент a_{ij}, то такое вычисление все равно требует не менее семи умножений. (*Подсказка*: покажите, что в каждом таком случае существует набор условий, например $a_{ij} = 0$ или $a_{kl} = a_{mn}$, при котором одно умножение тождественно равно нулю и получающееся вычисление решает упражнение 12.11 при $k = 2$.)

****12.15.** Покажите, что умножение матрицы размером 2×2 на матрицу размером $2 \times n$ требует ровно $\lceil 7n/2 \rceil$ умножений.

****12.16.** Постройте алгоритм умножения матрицы размером $n \times 2$ на матрицу размером $2 \times m$, используя $mn + m + n$ умножений, предполагая, что скаляры образуют коммутативное кольцо. Используя упражнение 12.15, покажите, что в некоммутативном варианте умножение матриц требует больше операций умножения, чем в коммутативном.

Определение. Пусть R — кольцо и $a_1, \ldots, a_n$ и $b_1, \ldots, b_n$ — непересекающиеся множества неопределенных переменных. *Билинейной формой* называется выражение вида

$$\sum_{i,j} r_{ij} a_i b_j,$$

где r_{ij} — элементы R.

****12.17.** Выполните следующие задания.

а) Покажите, что для любого множества билинейных форм существует вычисление с минимальным количеством умножений, в котором все умножения выполняются над линейными функциями, зависящими от a_i и b_j.

б) Покажите, что существует вычисление с минимальным количеством умножений, в котором все умножения выполняются над линейными функциями, зависящими от a_i и b_j. (Это не так, если кольцо является коммутативным.)

в) Допустим, что вычисление допускает деление. Покажите, что существует вычисление с минимальным количеством умножений, не использующее деления.

**12.18. Пусть R — некоммутативное кольцо, $\mathbf{a}$, $\mathbf{b}$ и $\mathbf{x}$ — векторы-столбцы неопределенных переменных $[a_1, ..., a_m]^T$, $[b_1, ..., b_n]^T$ и $[x_1, ..., x_p]^T$ соответственно. Пусть X — матрица линейных сумм x_i. Из упражнения 12.17 следует, что вычисления над множеством билинейных форм $(\mathbf{a}^T X)^T$ можно выразить как

$$M(P\mathbf{a} \cdot Q\mathbf{x}),$$

где M, P и Q — матрицы, элементы которых принадлежат F. *Левым двойственным множеством* по отношению к множеству выражений $(\mathbf{a}^T X)^T$ называется множество выражений $(\mathbf{b}^T X^T)^T$, а *P-двойственным* вычислением по отношению к вычислению $M(P\mathbf{a} \cdot Q\mathbf{x})$ — вычисления $P^T(M^T\mathbf{b} \cdot Q\mathbf{x})$. Докажите, что для любого P-двойственного вычисления $(\mathbf{a}^T X)^T$ рассчитывается левое двойственное множество по отношению к множеству выражений $(\mathbf{a}^T X)^T$.

*12.19. Докажите, что минимальное число умножений, необходимое для умножения матрицы размером $m \times n$ на матрицу размером $n \times p$ над некоммутативным кольцом, равно минимальному числу умножений, необходимых для умножения матрицы размером $n \times m$ на $(m \times p)$-матрицу. (Подсказка: воспользуйтесь упражнением 12.18.)

До сих пор в упражнениях содержался материал, относящийся к нижним границам для числа арифметических операций. В последующих упражнениях содержится материал более общего характера. Упражнения 12.20–12.23 касаются транспонирования матриц. Пусть a_{ij} — неопределенные переменные, $1 \le i, j \le n$. Рассмотрим модель вычислений, в которой каждая переменная представляет собой кортеж, состоящий из n неопределенных переменных. Шаг $a \leftarrow b\ \theta\ c$ такого вычисления присваивает кортежу из n элементов неопределенные переменные a, выбранные из неопределенных- переменных b или c. При этом кортежи b и c, состоящие из n элементов, не изменяются.

*12.20. Докажите, что любое вычисление множества кортежей из n элементов

$$\{(a_{1i}, ..., a_{ni}) | 1 \le i \le n\}$$

по множеству входов $\{(a_{i1}, ..., a_{in}) | 1 \le i \le n\}$ требует не менее $n \log n$ шагов. (*Подсказка*: пусть s_{ij} для каждых i и j обозначает максимальное количество неопределенных переменных с нижним индексом j, которые входят в один кортеж из n элементов, содержащий a_{ij}. Пусть

$$f = \sum_{i=1}^{n} \sum_{j=1}^{n} \log s_{ij}.$$

Рассмотрите изменение f по мере выполнения шагов вычисления.)

**12.21. Наложим следующее ограничение на вид шагов вычисления: в $a \leftarrow b\ \theta\ c$ операция θ состоит в выборе $n/2$ неопределенных переменных из циклического сдвига кортежа b, состоящего из n элементов, и $n/2$ неопределенных переменных из дополнительных позиций в c. Найдите вычисление для множества кортежей из n элементов, описанных в упражнении 12.20, состоящее из $O(n \log n)$ шагов.

**12.22. Пусть G — ориентированный ациклический граф с n выделенными *истоками* (узлами, в которые не входят никакие ребра) и n выделенными *выходами* (узлами, из которых не выходят никакие ребра). Пусть X и Y —подмножества истоков и выходов соответственно, а $G(X, Y)$ — подграф, состоящий из всех ориентированных ребер, лежащих на путях из узлов множества X в узлы множества Y. *Пропускной способностью* графа $G(X, Y)$ называется наименьшее число узлов, удаление которых (вместе с входящими и выходящими ребрами) отделяет каждый узел из X от каждого узла из Y. Допустим, что для любых X и Y граф $G(X, Y)$ обладает пропускной способностью $\mathtt{MIN}(||X||, ||Y||)$. Покажите, что для каждого n существует такой граф G с $cn \log n$ ребрами, где c — некоторая константа.

**12.23. *Сдвигающей схемой* называется такой ориентированный граф с n истоками, занумерованными числами от 0 до $n-1$, и n выходами, занумерованными числами от 0 до $n-1$, что для каждого s, $0 \le s \le n-1$, найдется множество путей, непересекающихся по узлам, которое для каждого i содержит путь из истока i в выход $(i+s)$ по модулю n.

а) Покажите, что для каждого n существует сдвигающая схема с $2n \log n$ ребрами.

б) Покажите, что для сдвигающей схемы асимптотически требуется $n \log n$ ребер.

в) Покажите, что с помощью сдвигающей схемы можно вычислить транспонированную матрицу.

Определение. Пусть F — поле и $x_1, \ldots, x_n$ — неопределенные переменные. *Линейным вычислением* называется последовательность шагов вида $a \leftarrow \lambda_1 b + \lambda_2 c$, где a — переменная, b и c — переменные или неопределенные переменные, а λ_1 и λ_2 — элементы из F (называемые константами).

**12.24. Пусть F — поле комплексных чисел, A — матрица с элементов из F и $\mathbf{x} = [x_1, \ldots, x_n]^T$. Покажите, что любое линейное вычисление для $A\mathbf{x}$ требует $\log(\mathbf{det}(A))/\log 2c$ шагов, где c — наибольший из модулей[10] констант, входящих в шаги этого вычисления.

---

[10] *Модуль* числа $a + bi$ равен $\sqrt{a^2 + b^2}$.

***12.25.** Покажите, что для преобразования Фурье вектора $[x_1, ..., x_n]$ линейное вычисление с константами, по модулю не превосходящими 1, требует $\frac{1}{2} n \log n$ шагов.

Следующие шесть упражнений касаются минимального числа логических элементов с двумя входами, необходимых для реализации булевой функции.

***12.26.** Покажите, что каждую булеву функцию от n переменных можно реализовать с помощью не более $n2^n$ логических элементов с двумя входами.

***12.27.** Покажите, что для каждого n существует булева функция от n переменных, наименьшая реализация которой требует примерно $2^n/n$ логических элементов с двумя входами.

****12.28.** О сложности реализации конкретных булевых функций известно мало. Однако, если ограничиться реализациями на схемах, являющихся деревьями, можно показать, что для некоторых функций требуется $n^2/\log n$ логических элементов. Докажите, что следующее условие достаточно для того, чтобы булева функция от n переменных требовала для своей реализации не менее $n^2/\log n$ логических элементов.

Условие. Пусть $F(x_1, ..., x_n)$ — булева функция от n переменных. Разобьем множество переменных x_i на $b = n/\log n$ блоков по $\log n$ переменных в каждом. Присвоим значения (0 или 1) всем переменным в $b - 1$ блоках. Это можно сделать $2^n/n$ способами. Результатом будет функция от $\log n$ переменных, совпадающая с одной из 2^n функций от $\log n$ переменных. Функция, получаемая в действительности, зависит от значений, присвоенных переменным в $b - 1$ блоках. Пусть B_i — блок переменных, которым значения не присвоены. Если присваивание подходящих значений переменным не из B_i дает порядка $2^n/n$ различных функций от переменных из B_i, причем это выполняется для каждого i, $1 \le i \le \log n$, то любая древовидная схема для f должна содержать $n^2/\log n$ логических элементов.

****12.29.** Пусть $m = 2 + \log n$ и $\{\mathbf{t}_{ij} | 1 \le i \le n/m,\ 1 \le j \le m\}$ — такое множество m-мерных векторов из 0 и 1, что каждый вектор $\mathbf{t}_{ij}$ имеет среди своих компонентов не менее двух единиц. Пусть

$$f\left(x_{11}, x_{12}, ..., x_{n/m,m}\right) = \bigoplus_{\substack{1 \le i \le n/m, \\ 1 \le j \le m}} x_{ij} \cdot \left[\bigoplus_{\substack{1 \le k \le n/2, \\ k \ne i}} \prod_{\substack{1 \le l \le m \\ \text{такие что} \\ t^l_{ij} = 1}} x_{kl} \right],$$

где t_{ij}^{l} — это l-й компонент вектора $\mathbf{t}_{ij}$. Докажите, что в любой реализации функции f с помощью древовидной схемы не менее $n^2/\log n$ логических элементов.

***12.30.** Постройте другие булевы функции, для реализации которых с помощью древовидных схем требуется (*а*) $n^{3/2}$ и (*б*) $n^2/\log n$ логических элементов

12.31. Пусть $S_i(x_1, ..., x_n)$ — симметрическая булева функция, принимающая значение 1 тогда и только тогда, когда ровно i переменных x имеют значение 1.

а) Найдите реализацию функции $S_3(x_1, ..., x_n)$ с минимально возможным числом логических элементов.

б) Найдите древовидную реализацию для $S_3(x_1, ..., x_n)$ с минимально возможным числом логических элементов.

****12.32.** В примере 12.9 мы доказали, что любая схема, пригодная для вычисления значений произвольного полинома n-й степени, требует n умножений. Постройте конкретный полином n-й степени с действительными коэффициентами, для вычисления значений которого требуется n умножений.

****12.33.** Докажите, что умножение матриц в полукольце положительных целых чисел с операциями `MIN` и + требует cn^3 операций, где c — константа, $0 < c < 1$.

****12.34.** Докажите, что умножение булевых матриц с операциями `И` и `ИЛИ` требует n^3 операций.

Упражнения 12.35 и 12.36 касаются такого представления полинома, при котором значение полинома в точке можно вычислить примерно за $n/2$ умножений.

****12.35.** Пусть $p(x)$ — полином n-й степени, где n — нечетное целое число, большее 1. Запишем его в виде $p(x) = (x^2 - \alpha)q(x) + bx + c$.

а) Покажите, что при подходящем α можно взять $b = 0$.

б) Покажите, что существуют такие α_i и β_i, что $p(x)$ можно представить в виде

$$p(x) = (x^2 - \alpha_1)[(x^2 - \alpha_2)[\ ...\ ] + \beta_2] + \beta_1,$$

и, значит, полином $p(x)$, представленный этими α_i и β_i, можно вычислить за $\lfloor n/2 \rfloor + 2$ умножений.

в) Параметры α_i в пункте (*б*) могут оказаться комплексными числами. Как преодолеть эту трудность? (*Подсказка*: подставьте $y + c$ вместо x в $p(x)$ при подходящем c и вместо вычисления значения $p(x)$ в точке $x = x_0$ вычислите некоторый полином $p'(y)$ в точке $y = x_0 - c$.)

Упражнение 12.35 не дает метода для вычисления α_i. Более того, параметры α_i не обязательно являются рациональными числами. В следующем упражнении делается попытка преодолеть эти трудности.

****12.36.** Пусть m — целое число. Для $1 \le l \le m$ положим

$$r_{l0} = \frac{(m-l)^2 + (l-1)^2 + 3m + 1}{2}$$

и $r_{lj} = m - l + j + 1$, $1 \le j \le l$. Определим цепь с параметрами α_{ij} и β_{kl} как вычисление вида

$$c_{i1}(x) \leftarrow \left(x^{r_{i0}} + \alpha_{i1}\right)\left(x^{r_{i1}} + \beta_{i1}\right),$$

$$c_{i2}(x) \leftarrow \left(c_{i1} + \alpha_{i2}\right)\left(x^{r_{i2}} + \beta_{i2}\right),$$

$$\ldots$$

$$c_{ii}(x) \leftarrow \left(c_{i,i-1} + \alpha_{ii}\right)\left(x^{r_{ii}} + \beta_{ii}\right).$$

Пусть $p(x) = \sum_{i=1}^{l} c_{ii}(x)$.

а) Докажите, что показатель наибольшей степени переменной x, коэффициент при которой зависит от α_{ij}, равен $\sum_{k=j}^{l} r_{ik}$, а показатель наибольшей степени переменной x, коэффициент при которой зависит от β_{ij}, равен $\sum_{k=0}^{l} r_{ik} - r_{ij}$.

б) Докажите, что полином степени, не превосходящей $m(m + 1) + 1$, со старшим коэффициентом, равным 1, можно представить $m(m + 1) + 1$ параметрами, так что (1) этот полином можно вычислить по параметрам за $m(m + 1)/2 + O(m)$ умножений и (2) параметры являются рациональными функциями от коэффициентов.

Проблемы для исследования

Каждой линейной программе можно следующим образом поставить в соответствие ориентированный ациклический граф. Узлы графа представляют входы и переменные. Если $f \leftarrow g * h$ — шаг, то из g в f и из h в f идут ориентированные ребра. Заметим, что число шагов программы ровно вдвое меньше числа ребер. Поэтому нижняя оценка числа ребер в любом ориентированном ациклическом графе, соответствующем вычислению для конкретной задачи, дает нижнюю оценку числа шагов в любой неветвящейся программе для этой задачи.

12.37. Докажите или опровергните, что для больших n всякий ориентированный ациклический граф с n истоками и n выходами удовлетворяющими условию на пропускные способности из упражнения 12.22, содержит не менее $n \log n$ ребер.

12.38. Удовлетворяет ли каждый граф, соответствующий линейной программе для умножения двух n-разрядных двоичных целых чисел (предполагается, что используются битовые операции), условию на пропускные способности из упражнения 12.22?

Библиографические замечания

Теорема 12.2 и общая формулировка задачи, изучаемой в этом разделе, принадлежат Winograd [1970a]. Теоремы 12.1 и 12.3 взяты из работы Fiduccia [1971]. Тот факт, что без использования делений для вычисления значения полинома n-й степени требуется n умножений, Knuth [1969] приписывает A. Garcia. Pan [1966] распространил этот результат на мультипликативные операции. Motzkin [1955] показал, что для вычисления значения полинома с предобуславливанием необходимо $\lfloor n/2 \rfloor + 1$ умножений. Одной из первых в этом направлении была работа Ostrowski [1954]. Winograd [1970b] показал, что для умножения комплексных чисел необходимы три умножения.

Упражнение 12.7 обсуждается в Winograd [1973]. Материал о нижних оценках для умножения матриц (упражнения 12.9–12.16) основан на работе Hopcroft and Kerr [1971]. Результаты упражнения 12.17 независимо получены несколькими авторами. Часть в) Winograd (частное сообщение) приписывает P. Ungar. Упражнения 12.18 и 12.19 заимствованы их Hopcroft and Muszinski [1973]. Упражнения 12.20–12.22, 12.37 и 12.38 основаны на беседах с R.Floyd. Упражнения 12.24 и 12.25 взяты из Morgenstern [1973], 12.28 и 12.29 — из Neciporuk [1966], 12.30, *a*, — из Harper and Savage[1972], 12.32 — из Strassen [1974], 12.33 — из Kerr [1970], 12.34 — из Pratt [1974], 12.35 — из Eve [1964] и 12.36 — из Rabin and Winograd [1971].

Некоторые попытки получения нижних оценок для числа арифметических операций были сделаны Borodin and Cook [1974] и Kedem [1974].

Список литературы

1. Adel'son–Vel'skii, G. M., and Y. M. Landis [1962]. "An algorithm for the organization of information," *Dokl. Akad. Nauk SSSR* **146**, 263–266 (in Russian). English translation in Soviet Math. Dokl. 3 (1962), 1259–1262.
2. Aho, A. V. (ed.) [1973]. *Currents in the Theory of Computing*, Prentice–Hall, Enghwood Cliffs, N.J.
3. Aho, A. V., M. R. Garey, and J. D. Ullman [1972]. "The transitive reduction of directed graph," SIAM J. Computing **1**:2. 131-137.
4. Aho, A. V., D. S. Hirschberg, and J. D. Ullman [1974]. "Bounds on the complexity of the maximal common subsequence problem," *Conference Record IEEE 15th Annual Symposium on Switching and Automata Theory*.
5. Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1968]. "Time and tape complexity of pushdown automaton languages," *Information and Control* **13**:3, 186–206.
6. Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1976]. "On finding lowest common ancestors in trees," SIAM J. Computing, **1**:1, 115-132.
7. Aho, A. V., K. Steiglitz, and J. D. Ullman [1974]. "Evaluating polynomials at fixed sets of points," Bell Laboratories, Murray Hill, N. J.
8. Aho, A. V., and J. D. Ullman [1972]. *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*, Prentice–Hall, Englewood Cliffs, N.J.
9. Aho, A. V., and J. D. Ullman [1973]. *The Theory of Parsing, Translation, and Compiling, Volume 2: Compiling*, Prentice–Hall, Englewood Cliffs, N.J.
10. Arlazarov, V. L., E. A. Dinic, M. A. Kronrod, and I. A. Faradzev [1970]. "On economical construction of the transitive closure of a directed graph," *Dokl. Akad. Nauk SSSR* 194, 487–488 (in Russian). English translation in *Soviet Math. Dokl.* 11:5, 1209–1210.
11. Belaga, E. C. [1961]. "On computing polynomials in one variable with initial preconditioning of the coefficients," *Problemi Kibernetiki* **5**, 7–15.
12. Bellman, R. E. [1957]. *Dynamic Programming*, Princeton University Press, Princeton, N.J.
13. Berge, C. [1958]. *The Theory of Graphs and its Applications*, Wiley, New York.
14. Birkhoff, G., and T. Bartee [1970]. *Modern Applied Algebra*, McGraw–Hill, New York.

15. Bluestein, L. I. [1970]. "A linear filtering approach to the computation of the discrete Fourier transform," *IEEE Trans, on Electroacoustics AU–***18**:4, 451–455. Also in Rabiner and Rader [1972], pp. 317–321.

16. Blum, M. [1967]. "A machine independent theory of the complexity of recursive functions," *J. ACM* **14**:2, 322–336.

17. Blum, M., R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan [1972]. "Time bounds for selection," *J. Computer and System Sciences* **7**:4, 448–461.

18. Book, R. V. [1972]. "On languages accepted in polynomial time," *SIAM J. Computing* **1**:4, 281–287.

19. Book, R. V. [1974]. "Comparing complexity classes," J. Computer and System Sciences, **9**, 213-229.

20. Borodin, A. B. [1973a]. "Computational complexity–theory and practice," in Aho [1973], pp. 35–89.

21. Borodin, A. B. [1973b]. "On the number of arithmetics required to compute certain functions — circa May 1973," in Traub [1973], pp. 149–180.

22. Borodin, A. B., and S. A. Cook [1974]. "On the number of additions to compute specific polynomials," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 342–347.

23. Borodin, A. B., and I. Munro [1971]. "Evaluating polynomials at many points," *Information Processing Letters* **1**:2, 66–68.

24. Borodin, A. B., and I. Munro [1975]. *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, N.Y.

25. Brown, W. S. [1971]. "On Euclid's algorithm and greatest common divisors," *J. ACM* **18**:4, 478–504.

26. Brown, W. S. [1973]. *ALTRAN User's Manual* (3rd edition), Bell Laboratories, Murray Hill, N.J.

27. Bruno, J. L., and R. Sethi [1974]. "Register allocation for a one–register machine," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems*.

28. Bunch, J., and J. E. Hopcroft [1974]. "Triangular factorization and inversion by fast matrix multiplication," *Math. Comp.* **28**:125, 231–236.

29. Burkhard, W. A. [1973]. "Nonrecursive tree traversal algorithms," *Proc. 7th Annual Princeton Conference on Information Sciences and Systems*, 403–405.

30. Collins, G. E. [1973]. "Computer algebra of polynomials and rational functions," *American Math. Monthly* **80**:7, 725–754.

31. Constable, R. L., H. B. Hunt III, and S. K. Sahni [1974]. "On the computational complexity of scheme equivalence," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems.*

32. Cook, S. A. [1971a]. "Linear time simulation of deterministic two–way pushdown automata," *Proc. IFIP Congress 71, TA–2.* North–Holland, Amsterdam, 172–179.

33. Cook, S. A. [1971b]. "The complexity of theorem proving procedures," *Proc. 3rd Annual ACM Symposium on Theory of Computing*, 151–158.

34. Cook, S. A. [1973]. "A hierarchy for nondeterministic time complexity," *J. Computer and System Sciences* 7:4, 343–353.

35. Cook, S. A., and S. O. Aanderaa [1969]. "On the minimum complexity of functions," *Trans. Amer. Math. Soc.* **142**, 291–314.

36. Cock, S. A., and R. A. Reckhow [1973]. "Time–bounded random access machines," *J. Computer and System Sciences* 7:4, 354–375.

37. Cook, S. A., and R. A. Reckhow [1974]. "On the lengths of proofs in the propositional calculus," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 135–148.

38. Cooley, J. M., P. A. Lewis, and P. D. Welch [1967]. "History of the fast Fourier transform," *Proc. IEEE* **55**, 1675–1677.

39. Cooley, J. M., and J. W. Tukey [1965]. "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.* **19**, 297–301.

40. Crane, C. A. [1972]. "Linear lists and priority queues as balanced binary trees," Ph.D. Thesis, Stanford University.

41. Danielson, G. C., and C. Lanczos [1942].. "Some improvements in practical Fourier analysis and their application to X–ray scattering from liquids," *J. Franklin Institute* **233**, 365–380 and 435–452.

42. Dantzig, G. B., W. O. Blattner, and M. R. Rao [1966]. "All shortest routes from a fixed origin in a graph," *Theory of Graphs*, International Symposium, Rome, July 1966. Proceedings published by Gordon and Breach, New York, 1967, pp. 85–90.

43. Davis, M. [1958]. *Computability and Unsolvability*, McGraw–Hill, New York.

44. Dijkstra, E. W. [1959]. "A note on two problems in connexion with graphs," *Numerische Mathematik* **1**, 269–271.

45. Divetti, L., and A. Grasselli [1968]. "On the determination of minimum feedback arc and vertex sets," *IEEE Trans. Circuit Theory* **CT–15**:1, 86–89.

46. Ehrenfeucht, A., and H. P. Zeiger [1974]. "Complexity measures for regular expressions," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 75–79.

47. Elgot, C. C., and A. Robinson [1964]. "Random access stored program machines," *J. ACM* **11**:4, 365–399.

48. Eve, J. [1964]. "The evaluation of polynomials," *Numerische Mathematik* **6**, 17–21.

49. Even. S. [1973]. "An algorithm for determining whether the connectivity of a graph is at least k," TR–73–184, Dept, of Computer Science, Cornell University, Ithaca, N.Y.

50. Fiduccia, C. M. [1971]. "Fast matrix multiplication," *Proc. 3rd Annual ACM Symposium on Theory of Computing*, 45–49.

51. Fiduccia, C. M. [1972]. "Polynomial evaluation via the division algorithm — the fast Fourier transform revisited," *Proc. 4th Annual ACM Symposium on Theory of Computing*, 88–93.

52. Fischer, M. J. [1972]. "Efficiency of equivalence algorithms," in Miller and Thatcher [1972], pp. 153–168.

53. Fischer, M. J., and A. R. Meyer [1971]. "Boolean matrix multiplication and transitive closure," *Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory*, 129–131.

54. Fischer, M. J., and M. S. Paterson [1974]. "String–matching and other products." Project MAC Technical Memorandum 41, MIT, Cambridge, Mass.

55. Fischer, M. J., and M. O. Rabin [1974]. "The complexity of theorem proving procedures," Project MAC Report, MIT, Cambridge, Mass.

56. Floyd, R. W. [1962]. "Algorithm 97: shortest path," *Comm. ACM* **5**:6, 345.

57. Floyd, R. W. [1964]. "Algorithm 245: treesort 3," *Comm. ACM* **7**:12, 701.

58. Floyd, R. W., and R. L. Rivest [1973]. "Expected time bounds for selection," Computer Science Dept., Stanford University.

59. Ford, L. R., and S. M. Johnson [1959]. "A tournament problem," *Amer. Math. Monthly* **66**, 387–389.

60. Frazer, W. D., and A. C. McKellar [1970]. "Samplesort: a sampling approach to minimal storage tree sorting," *J. ACM* **17**:3, 496–507.

61. Furman, M. E. [1970]. "Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph," *Dokl. Akad. Nauk SSSR* **194**, 524 (in Russian). English translation in *Soviet Math. Dokl.* **11**:5, 1252.

62. Gale, D., and R. M. Karp [1970]. "A phenomenon in the theory of sorting," *Conference Record, IEEE' 11th Annual Symposium on Switching and Automata Theory*, 51–59.

63. Galler, B. A., and M. J. Fischer [1964]. "An improved equivalence algorithm," *Comm. ACM* 7:5, 301–303.

64. Garey, M. R., D. S. Johnson, and L. Stockmeyer [1974]. "Some simplified polynomial complete problems," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 47–63.

65. Gentleman, W. M. [1972]. "Optimal multiplication chains for computing powers of polynomials," *Math. Comp.* **26**:120, 935–940.

66. Gentleman, W. M., and S. C. Johnson [1973]. "Analysis of algorithms, a case study: determinants of polynomials." *Proc. 5th Annual ACM Symposium on Theory of Computing*, 135–141.

67. Gentleman, W. M., and G. Sande [1966]. "Fast Fourier transforms for fun and profit," *Proc. AFIPS 1966 Fall Joint Computer Conference*, v. 29, Spartan, Washington, D.C., pp. 563–578.

68. Gilbert, E. N., and E. F. Moore [1959]. "Variable length encodings," *Bell System Technical J.* **38**:4, 933–968.

69. Godbole, S. S. [1973]. "On efficient computation of matrix chain products," *IEEE Transactions on Computers* **C–22**:9, 864–866.

70. Good, I. J. [1958]. "The interaction algorithm and practical Fourier series," *J. Royal Statistics Soc.*, Ser B, **20**, 361–372. Addendum 22 (1960), 372–375.

71. Graham, R. L. [1972]. "An efficient algorithm for determining the convex hull of a planar set," *Information Processing Letters* **1**:4, 132–133.

72. Gray, J. N., M. A. Harrison, and O. H. Ibarra [1967]. "Two way pushdown automata," *Information and Control* **11**:3, 30–70.

73. Hadian, A., and M. Sobel [1969]. "Selecting the /th largest using binary errorless comparisons," Tech. Rept. 121, Department of Statistics, University of Minnesota, Minneapolis.

74. Hall, A. D. [197 k]. "The ALTRAN system for rational manipulation —a survey," *Comm. ACM* **14**:8, 517–521.

75. Harary, F. [1969]. *Graph Theory*, Addison–Wesley, Reading, Mass.

76. Harper, L. H., and J. E. Savage [1972]. "On the complexity of the marriage problem," in *Advances in Mathematics* **9**:3, 299–312.

77. Hartmanis, J. [1971]. "Computational complexity of random access stored program machines," *Mathematical Systems Theory* **5**:3, 232–245.

78. Hartmanis, J., and J. E. Hopcroft [1971]. "An overview of the theory of computational complexity," *J. ACM* **18**:3, 444–475.

79. Hartmanis, J., P. M. Lewis II, and R. E. Stearns [1965]. "Classification of computations by time and memory requirements," *Proc. IFIP Congress 65*, Spartan, N.Y., 31–35.

80. Hartmanis, J., and R. E. Stearns [1965]. "Oh the computational complexity of algorithms," *Trans. Amer. Math. Soc.* **117**, 285–306.

81. Hecht, M. S. [1973]. "Global data flow analysis of computer programs," Ph.D. Thesis, Dept, of Electrical Engineering, Princeton University.

82. Heindel, L. E., and E. Horowitz [1971]. "On decreasing the computing time for modular arithmetic," *Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory*, 126–128.

83. Hennie, F. C., and R. E. Stearns [1966]. "Two tape simulation of multitape machines," *J. ACM* **13**:4, 533–546.

84. Hirschberg, D. S. [1973]. "A linear space algorithm for computing maximal common subsequences," TR–138, Computer Science Laboratory, Dept, of Electrical Engineering, Princeton University, Princeton, N.J.

85. Hoare, C. A. R. [1962]. "Quicksort," *Computer J.* **5**:1 10–15.

86. Hohn, F. E. [1958]. *Elementary Matrix Algebra*, Macmillan, New York.

87. Hopcroft, J. E. [1971]. "An $n\log n$ algorithm for minimizing states in a finite automaton," in Kohavi and Paz [1971], pp. 189–196.

88. Hopcroft, J. E., and R. M. Karp [1971]. "An algorithm for testing the equivalence of finite automata," TR–71–114, Dept, of Computer Science, Cornell University, Ithaca, N.Y.

89. Hopcroft, J. E., and L. R. Kerr [1971]. "On minimizing the number of multiplications necessary for matrix multiplication," *SIAM J. Applied Math.* **20**:1, 30–36.

90. Hopcroft, J. E., and J. Musinski [1973]. "Duality in determining the complexity of noncommutative matrix multiplication," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 73–87.

91. Hopcroft, J. E., and R. E. Tarjan [1973a]. "Efficient planarity testing," TR73–165, Dept, of Computer Science, Cornell University, Ithaca, N.Y., also *J. ACM*, 1974, № 21. — P. 549-568.

92. Hopcroft, J. E., and R. E. Tarjan [1973b]. "Dividing a graph into triconnected components," *SIAM J. Computing* 2:3, 135-157.

93. Hopcroft, J. E., and R. E. Tarjan [1973c], "Efficient algorithms for graph manipulation," *Comm. ACM* **16**:6, 372–378.

94. Hopcroft, J. E., and J. D. Ullman [1969], *Formal Languages and Their Relation to Automata*, Addison–Wesley, Reading, Mass.

95. Hopcroft, J. E., and J. D. Ullman [1973]. "Set merging algorithms," *SIAM J. Computing*, **2**:4,294–303.

96. Hopcroft, J. E., and J. K. Wong [1974]. "A linear time algorithm for isomorphism of planar graphs," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 172–184.

97. Horowitz, E. [1972]. "A fast method for interpolation using preconditioning," *Information Processing Letters* **1**:4, 157–163.

98. Horvath, E. C. [1974]. "Some efficient stable sorting algorithms," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 194–215.

99. Hu, T. C. [1968]. "A decomposition algorithm for shortest paths in a network," *Operations Res.* **16**, 91–102.

100. Hu, T. C., and A. C. Tucker [1971]. "Optimum binary search trees," *SIAM J. Applied Math.* **21**:4, 514–532.

101. Hunt, H. B., III [1973a]. "On the time and tape complexity of languages," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 10–19. See also TR–73–182, Dept, of Computer Science, Cornell University, Ithaca, N.Y.

102. Hunt, H. B., III [1973b]. "The equivalence problem for regular expressions with intersection is not polynomial in tape," TR 73–156, Dept, of Computer Science, Cornell University, Ithaca, N.Y.

103. Hunt, H. B., IIII [1974]. "Stack languages, Rice's theorem and a natural exponential complexity gap," TR–142, Computer Science Laboratory, Dept, of Electrical Engineering, Princeton University, Princeton, N.J.

104. Hunt, H. B., III, and D. J. Rosenkrantz [1974]. "Computational parallels between the regular and context–free languages," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 64–74.

105. Ibarra, O. H. [1972]. "A note concerning nondeterministic tape complexities," *J. ACM* **19**:4, 608–612.

106. Ibarra, O. H., and S. K. Sahni [1973]. "Polynomial complete fault detection problems," Technical Report 74–3, Computer, Information and Control Sciences, University of Minnesota, Minneapolis.

107. Jones, N. D. [1973]. "Reducibility among combinatorial problems in log n space," *Proc. 7th Annual Princeton Conference on Information Sciences and Systems*, 547–551.

108. Johnson, D. B. [1973]. "Algorithms for shortest paths," Ph.D. Thesis, Dept, of Computer Science, Cornell University, Ithaca, New York.

109. Johnson, S. C. [1974]. "Sparse polynomial arithmetic," Bell Laboratories, Murray Hill, N.J.

110. Karatsuba, A., and Y. Ofman [1962]. "Multiplication of multidigit numbers on automata," *Dokl. Akad. Nauk SSSR* **145**, 293–294 (in Russian).

111. Kasami, T. [1965]. "An efficient recognition and syntax algorithm for context–free languages," Scientific Report AFCRL–65–758, Air Force Cambridge Research Laboratory, Bedford, Mass.

112. Karp, R. M. [1972]. "Reducibility among combinatorial problems," in Miller and Thatcher [1972], pp. 83–104.

113. Karp, R. M., R. E. Miller, and A. L. Rosenberg [1972]. "Rapid identification of repeated patterns in strings, trees, and arrays," *Proc. 4th Annual ACM Symposium on Theory of Computing*, 125–136.

114. Kedem, Z. [1974]. "On the number of multiplications and divisions required to compute certain rational functions," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 334–341.

115. Kerr, L. R. [1970]. "The effect of algebraic structure on the computational complexity of matrix multiplications," Ph.D. Thesis, Cornell University, Ithaca, N.Y.

116. Kirkpatrick, D. [1972]. "On the additions necessary to compute certain functions," *Proc. 4th Annual ACM Symposium on Theory of Computing*, 94–101.

117. Kirkpatrick, D. [1974]. "Determining graph properties from matrix representations," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 84–90.

118. Kislitsyn, S. S. [1964]. "On the selection of the £th element of an ordered set by pairwise comparison," *Sibirsk. Mat. Zh.* **5**, 557–564.

119. Kleene, S. C. [1956]. "Representation of events in nerve nets and finite automata," in Automata Studies (Shannon and McCarthy, eds.), Princeton University Press, pp. 3–40.

120. Knuth, D. E. [1968]. *Fundamental Algorithms*, Addison–Wesley, Reading, Mass.

121. Knuth, D. E. [1969]. *Seminumerical Algorithms*, Addison–Wesley, Reading, Mass.

122. Knuth, D. E. [1971]. "Optimum binary search trees," *Acta Informatica* **1**, 14–25.

123. Knuth, D. E. [1973a]. *Sorting and Searching*, Addison–Wesley, Reading, Mass.

124. Knuth, D. E. [1973b]. "Notes on pattern matching," University of Trondheim, Norway.

125. Knuth, D. E., and V. R. Pratt [1971]. "Automata theory can be useful," Stanford University, Stanford, California.

126. Kohavi, Z., and A. Paz (eds.) [1971]. *Theory of Machines and Computations*, Academic Press, New York.

127. Kruskal, J. B., Jr. [1956]. "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.* **7**:1, 48–50.

128. Kung, H. T. [1973]. "Fast evaluation and interpolation," Dept. of Computer Science, Carnegie–Mellon University, Pittsburgh.

129. Lewis, P. M., II, R. E. Stearns, and J. Hartmanis [1965]. "Memory bounds for recognition of context–free and context–sensitive languages," *Conference Record, IEEE 6th Annual Symposium on Switching Circuit Theory and Logic Design*, 191–202.

130. Lipson, J. [1971]. "Chinese remainder and interpolation algorithms," Proc. 2nd Symposium on Symbolic and Algebraic Manipulation, 372–391.

131. Liu, C. L. [1968]. *Introduction to Combinatorial Mathematics*, McGraw–Hill, New York.

132. Liu, C. L. [1972]. "Analysis and synthesis of sorting algorithms," *SIAM J. Computing* **1**:4, 290–304.

133. MacLane, S., and G. Birkhoff [1967]. *Algebra*, Macmillan, New York.

134. MacNaughton, R,, and H. Yamada [I960]. "Regular expressions and state graphs for automata," *IRE Trans. on Electronic Computers* **9**:1, 39–47. Reprinted in Moore [1964], pp. 157–174.

135. Meyer, A. R. [1972]. "Weak monadic second order theory of successor is not elementary recursive," Project MAC Report, MIT, Cambridge, Mass.

136. Meyer, A. R., and L. Stockmeyer [1972]. "The equivalence problem for regular expressions with squaring requires exponential space," *Conference Record, IEEE 13th Annual Symposium on Switching and Automata Theory*, 125–129.

137. Meyer, A. R., and L. Stockmeyer [1973]. "Nonelementary word problems in automata and logic," *Proc. AMS Symposium on Complexity of Computation*, April 1973.

138. Miller, R. E., and J. W. Thatcher (eds.) [1972]. Complexity of Computer Computations, Plenum Press, New York.

139. Minsky, M. [1967]. *Computation: Finite and Infinite Machines*, Prentice–Hall, Englewood Cliffs, N.J.

140. Moenck, R. [1973]. "Fast Computation of GCD's," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 142–151.

141. Moenck, R., and A. B. Borodin [1972]. "Fast modular transforms via division," *Conference Record, IEEE 13th Annual Symposium on Switching and Automata Theory*, 90–96.

142. Moore, E. F. (ed.) [1964]. *Sequential Machines: Selected Papers*, Addison–Wesley, Reading, Mass.

143. Morgenstern, J. [1973]. "Note on a lower bound of the linear complexity of the fast Fourier transform," *J. ACM* **20**:2, 305–306.

144. Morris, R. [1968]. "Scatter storage techniques," *Comm. ACM* **11**:1, 35–44.

145. Morris, J. H., Jr., and V. R. Pratt [1970]. "A linear pattern matching algorithm," Technical Report No. 40, Computing Center, University of California, Berkeley.

146. Motzkin, T. S. [1955]. "Evaluation of polynomials and evaluation of rational functions," *Bull. Amer. Math. Soc.* **61**, 163.

147. Munro, I. [1971]. "Efficient determination of the transitive closure of a directed graph," *Information Processing Letters* **1**:2, 56–58.

148. Muraoka, Y., and D. J. Kuck [1973]. "On the time required for a sequence of matrix products," *Comm. ACM* **16**:1, 22–26.

149. Neciporuk, E. I. [1966]. "A Boolean function," Dokl. Akad. Nauk SSSR 169:4 (in Russian). English translation in *Soviet Math. Dokl.* 7 (1966), 999–1000.

150. Nicholson, P. J. [1971]. "Algebraic theory of finite Fourier transforms," *J. Computer and System Sciences* **5**:5, 524–547.

151. Nievergelt, J., and E. M. Reingold [1973]. "Binary search trees of bounded balance," *SIAM J. Computing* **2**:1, 33–43.

152. Ofman, Y. [1962]. "On the algorithmic complexity of discrete functions," *Dokl. Akad. Nauk SSSR* 145, 48–51 (in Russian). English translation in *Soviet Physics Dokl.* 7 (1963), 589–591.

153. Ostrowski, A. M. [1954]. "On two problems in abstract algebra connected with Horner's rule," *Studies Presented to R. von Mises*, Academic Press, N.Y.

154. Pan, V. Y. [1966]. "Methods of computing values of polynomials," *Russian Mathematical Surveys* **21**:1, 105–136.

155. Paterson, M. S., M. J. Fischer, and A. R. Meyer [1974]. "An improved overlap argument for on–line multiplication," Project MAC Technical Memorandum 40 MIT, Cambridge, Mass.

156. Pohl, I. [1972]. "A sorting problem and its complexity," *Comm. ACM* **15**:6 462–464.

157. Pratt, T. W. [1975]. *Programming Language Design and Implementation*, Prentice–Hall, Englewood Cliffs, N.J.

158. Pratt, V. R. [1974]. "The power of negative thinking in multiplying Boolean matrices," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 80–83.

159. Pratt, V. R., and F. F. Yao [1973]. "On lower bounds for computing the *i*th largest element," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 70–81.

160. Prim, R. C. [1957]. "Shortest connection networks and some generalizations," *Bell System Technical J.*, 1389–1401.

161. Rabin, M. O. [1963]. "Real–time computation," *Israel J. Math.* **1**, 203–211.

162. Rabin, M. O. [1972]. "Proving simultaneous positivfty of linear forms," *J. Computer and System Sciences* **6**:6, 639–650.

163. Rabin, M. O., and D. Scott [1959]. "Finite automata and their decision problems," *IBM J. Research and Development* **3**, 114–125. Reprinted in Moore [1964], pp. 63–91.

164. Rabin, M. O., and S. Winograd [1971]. "Fast evaluation of polynomials by rational preparation," IBM Technical Report RC 3645, Yorktown Heights, N.Y.

165. Rabiner, L. R., and C. M. Rader (eds.) [1972]. *Digital Signal Processing*, IEEE Press, New York.

166. Rabiner, L. R., R. W. Schafer, and C. M. Rader [1969]. "The chirp z–transform and its applications," *Bell System Technical J.* **48**:3, 1249–1292. Reprinted in Rabiner and Rader [1972], pp. 322–328.

167. Rader, C. M. [1968]. "Discrete Fourier transform when the number of data points is prime," *Proc. IEEE* **56**, 1107–1108.

168. Reingold, E. M. [1972]. "On the optimality of some set algorithms," *J. ACM* 19:4, 649–659.

169. Rogers, H., Jr. [1967]. *Theory of Recursive Functions and Effective Computability*, McGraw–Hill, New York.

170. Rounds, W. C. [1973]. "Complexity of recognition in intermediate level languages," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 145–158.

171. Runge, C., and H. Konig [1924], *Die Grundlehren der mathematischen Vissenschaften*, v. 11, Springer, Berlin.

172. Sahni, S. K. [1972]. "Some related problems from network flows, game theory and integer programming," *Conference Record, IEEE 13th Annual Symposium on Switching and Automata Theory*, 130–138.

173. Savitch, W. J. [1970], "Relationship between nondeterministic and deterministic tape complexities," *J. Computer and System Sciences* **4**:2, 177–192.

174. Savitch, W. J. [1971], "Maze recognizing automata." *Proc. 4th Annual ACM Symposium on Theory of Computing*, 15 l–156.

175. Schonhage, A. [1971]. "Schnelle Berechnung von Kettenbruchentwicklungen," *Acta Informatica* **1**, 139–144.

176. Schonhage, A., and V. Strassen [1971]. "Schnelle Multiplikation grosser Zahlen," *Computing* 7, 281–292.

177. Seiferas, J. J., M. J. Fischer, and A. R. Meyer [1973]. "Refinements of nondeterministic time and space hierarchies," *Conference Record, IEEE Nth Annual Symposium on Switching and Automata Theory*, 130–137.

178. Sethi, R. [1973]. "Complete register allocation problems," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 182–195.

179. Shepherdson, J. C., and H. E. Sturgis [1963]. "Computability of recursive functions," *J. ACM* **10**:2, 217–255.

180. Sieveking, M. [1972]. "An algorithm for division of power series," *Computing* **10**, 153–156.

181. Singleton, R. C. [1969]. "Algorithm 347: An algorithm for sorting with minimal storage." *Comm. ACM* **12**:3, 185–187.

182. Sloane, N. J. A. [1973]. *A Handbook of Integer Sequences*, Academic Press, N.Y.

183. Spira, P. M. [1973]. "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$," *SIAM J. Computing* **2**:1, 28–32.

184. Spira, P. M., and A. Pan [1973]. "On finding and updating shortest paths and spanning trees," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 82–84.

185. Stearns, R. E., and D. J. Rosenkrantz [1969]. "Table machine simulation," *Conference Record, IEEE 10th Annual Symposium on Switching and Automata Theory*, 118–128.

186. Stockmeyer, L. [1973]. "Planar 3–colorability is polynomial complete," *SIGACT News* **5**:3, 19–25.

187. Stockmeyer, L., and A. R. Meyer [1973]. "Word problems requiring exponential time," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 1–9.

188. Stone, H. S. [1972]. *Introduction to Computer Organization and Data Structures*, McGraw–Hill, New York.

189. Strassen, V. [1969]. "Gaussian elimination is not optimal," *Numerische Mathematik* **13**, 354–356.

190. Strassen, V. [1974]. "Schwer berechenbare Polynome mit rationalen Koeffizienten," *SIAM J. Computing*.

191. Tarjan, R. E. [1972]. "Depth first search and linear graph algorithms," *SIAM J. Computing* **1**:2, 146–160.

192. Tarjan, R. E. [1973a]. "Finding dominators in directed graphs," *Proc. 7th Annual Princeton Conference on Information Sciences and Systems*, 414–418.

193. Tarjan, R. E. [1973b]. "Testing flow graph reducibility," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 96–107.

194. Tarjan, R. E. [1975]. "On the efficiency of a good but not linear set merging algorithm," J. ACM, **22**, 215-225.

195. Thompson, K. [1968]. "Regular expression search algorithm," *Comm. ACM* **11**:6, 419–422.

196. Traub, J. F. (ed.) [1973]. *Complexity of Sequential and Parallel Numerical Algorithms*, Academic Press, New York.

197. Turing, A. M. [1936]. "On computable numbers, with an application to the Entscheidungsproblem," *Proc. London Mathematical Soc. Ser. 2*, **42**, 230–265. Corrections, ibid., 43 (1937), 544–546.

198. Ullman, J. D. [1973]. "Polynomial complete scheduling problems," *Proc. 4th Symposium on Operating System Principles*, 96–101.

199. Ullman, J. D. [1974]. "Fast algorithms for the elimination of common subexpressions," *Acta Informatica*, **2**:3, 191–213.

200. Valiant, L. G. [1974]. "General context–free recognition in less than cubic time." Dept, of Computer Science, Camegie–Mellon University, Pittsburg, Pa.

201. Vari, T. M. "Some complexity results for a class of Toeplitz matrices," unpublished memorandum, York Univ., Toronto, Ontario, Canada.

202. Wagner, R. A. [1974]. "A shortest path algorithm for edge–sparse graphs." Dept, of Systems and Information Science, Vanderbilt University, Nashville, Tennessee.

203. Wagner, R. A., and M. J. Fischer [1974]. "The string–to–string correction problem," *J. ACM* **21**:1, 168–173.

204. Warshall, S. [1962]. "A theorem on Boolean matrices," *J. ACM* **9**:1, 11–12.

205. Weiner, P. [1973]. "Linear pattern matching algorithms," *Conference Record, IEEE Nth Annual Symposium on Switching and Automata Theory*, 1–11.

206. Williams, J. W. J. [1964]. "Algorithm 232: Heapsort," *Comm. ACM* 7:6, 347–348.

207. Winograd, S. [1965]. "On the time required to perform addition," *J. ACM* **12**:2, 277–285.

208. Winograd, S. [1967]. "On the time required to perform multiplication," *J. ACM* **14**:4, 793–802.

209. Winograd, S. [1970a]. "On the number of multiplications necessary to compute certain functions," *Comm. Pure and Applied Math.* **23**, 165–179.

210. Winograd, S. [1970b]. "On the multiplication of 2 x 2 matrices," IBM Research Report RC267, January 1970.

211. Winograd, S. [1970c]. "The algebraic complexity of functions," *Proc. International Congress of Mathematicians* (1970), vol. **3**, 283–288.

212. Winograd, S. [1973]. "Some remarks on fast multiplication of polynomials," in Traub [1973], pp. 181–196.

213. Younger, D. H. [1967]. "Recognition of context–free languages in time n^3," *Information and Control* **10**:2, 189–208. New York.

Предметный указатель

А

Автомат
  конечный
    детерминированный, 367; 373
    недетерминированный, 367
  линейно ограниченный, 467
  магазинный
    односторонний, 415
Адрес, 20
  возврата, 77
  значения, 77
  символьный, 35
Алгоритм
  быстрого преобразования Фурье, 297
  возведения в квадрат, 326
  вычисления обратного числа, 323
  Дейкстры, 239
  Евклида, 345
    расширенный, 346
  китайский, 338
    предобусловленный, 339
  Крускала, 201
  обращения полинома, 329
  поиска в глубину, 204
  предобусловленный, 338
  четырех русских, 281
  Шёнхаге–Штрассена, 311
  Штрассена, 264
    альтернативный, 286
  GCD
    для полиномов, 353
    для целых чисел, 354
Алфавит, 22; 366
Аннулятор, 226

Б

Билинейная форма, 516

В

Вектор
  характеристический, 66
Векторы
  зависимые, 500
  линейно независимые, 500
  эквивалентные, 500
Вершина графа
  смежная, 67
Вызов
  по значению, 55
  по имени, 55
  по ссылке, 55
Выражение
  выполнимое, 434
  регулярное
    полурасширенное, 475
    эквивалентное, 367
Высота
  вершины, 70
  дерева, 70
Вычисление, 497
  битовое, 37
  линейное, 518

Г

Глубина вершины, 70
Гомоморфизм, 480
Грамматика
  контекстно-свободная, 94

Граф, 67
ациклический, 73
корневой, 241
неориентированный, 67
ориентированный, 67
раскрашиваемый, 436
связный, 73

Д

Дерево, 69
2-3, 171
бинарного поиска, 139
бинарное, 70
полное, 71
доминаторное, 242
корневое, 69; 73
неориентированное, 73
ориентированное, 69
основное, 133
помеченное, 107
решений, 39
упорядоченное, 70
Диаграмма
переходов, 368
Динамическое программирование, 86
Доминатор, 242
непосредственный, 242

З

Задача
об упаковке, 465
одномерная, 465
об эквивалентности, 485
о коммивояжере, 465
о кратчайшем пути, 225
о пустоте, 485
о разбиении, 465
о ранце, 465
о расписании работ, 466
сортировки, 98
Замыкание, 227
графа, 225
рефлексивное, 225
транзитивное, 225
Клини, 366
позитивное, 366
Значение
операнда, 21

И

Изоморфизм
деревьев, 106
Интерполяция, 293

К

Китайская теорема об остатках, 338
Кольцо, 260
коммутативное, 260
Компонент графа
двусвязный, 208
Конечный автомат, 168
Конкатенация языков, 366
Константа, 497
Конфигурация автомата, 392
терминальная, 392
Корень дерева, 69
Куча, 110
сливаемая, 173
сцепляемая, 173

Л

Лес, 70
остовный, 133
Лист, 70

М

Матрица
булева, 280
верхняя треугольная, 263
невырожденная, 263
нижняя треугольная, 263
нормированная, 263
обратная, 262
перестановки, 263
смежности, 67
теплицева, 289

Машина
с произвольным доступом к памяти, RAM, 18
Тьюринга
многоленточная, 41
недетерминированная, 421
хранящая программу в памяти с произвольным доступом, RASP, 28
Мгновенное описание, 44
Модульная арифметика, 332
полиномов, 336
Моноид, 226
Мультиграф, 252

Н

Наибольший общий делитель, 345
Наименьшее общее кратное, 361

О

Образец, 375
Обратный элемент, 260
Обход дерева, 71
обратный, 72
прямой, 72
симметричный, 72
Определитель матрицы, 262
Отец вершины, 70
Отношение
антисимметричное, 98
рефлексивное, 98
транзитивное, 98
Отображение памяти, 20
Очередь, 64
с приоритетами, 173

П

Параметр
фактический, 54
формальный, 54
Паросочетание, 90
Переменная, 35
выходная, 35
Поддерево, 70
левое, 70
правое, 70
Подматрица, 263
главная, 263
Поиск
бинарный, 137
Покрытие
вершинное, 436
Поле, 260; 495
Полукольцо, 226
замкнутов, 226
Полустепень исхода, 67
Порядок
лексикографический, 99
линейный, 98
полный, 98
Потомок вершины
прямой, 70
Предок вершины, 70
прямой, 70
Преобразование Фурье
быстрое, FFT, 297
дискретное, 292
обратное, 293
Программа
линейная, 34
Процедура
рекурсивная, 73
Путь
простой, 67

Р

Разбиение дерева, 183
Разложение матрицы
LU, 268
LUP, 269
Ранг
вершины, 157
матрицы, 263
по столбцам, 501
Расширение поля, 496
Расщепление дерева, 181

Ребро графа, 67
древесное, 218
обратное, 218
поперечное, 218
составное, 245
Режим
онлайн, 132
офлайн, 132

С

Свертка, 294
отрицательно обернутая, 296
положительно обернутая, 296
Связность
сильная, 218
Символ
нетерминальный, 94
терминальный, 94
Словарь, 132
Сложность алгоритма
арифметическая, 496
временная, 15
асимптотическая, 15
в худшем случае, 25
пространственная, 15
асимптотическая, 15
средняя, 25
Сложность машины Тьюринга
временная, 46
пространственная, 46
Сложность RAM-программы
временная
в худшем случае, 26
средняя, 26
пространственная
в худшем случае, 26
логарифмическая, 27
средняя, 26
Сложность RASP-программы
временная, 30
пространственная, 33
Содержимое регистра, 20
Сортировка
блочная, 99
быстрая, 116
внешняя, 98
внутренняя, 98
пирамидальная, 110
слиянием, 85
с помощью сравнений, 109
топологическая, 89
Состояние
мгновенное, 367
допускающее, 367
начальное, 367
Список, 62
двусвязный, 64
свободный, 63
Статистика
порядковая, 120
Стек, 64
Степень вершины, 67
Стоимость пути, 225
Строка
пустая, 366
Сумматор, 20
Счетчик
адресов, 20
Сын вершины, 70
левый, 70
правый, 70

Т

Терминатор, 392
Точка сочленения, 208
Транзитивная редукция, 253
Транзитивное замыкание, 123

У

Уровень вершины, 70

Ф

Функция
конструируемая по пространству, 426; 491

конструируемая по времени, 490
отказов, 380
переходов, 43; 367
полиномиально связанная, 41
стоимости, 26; 133
логарифмическая, 26
хеширования, 136
частично рекурсивная, 22
экспоненциальная, 41

Х

Хеширование, 135

Ц

Цепь, 412
Цикл, 67; 436
гамильтонов, 436
ориентированный, 437
Циклическая разность, 317
Циркулянт, 317

Ч

Число
хроматическое, 436

Э

Эквивалентные
автоматы, 169
состояния, 168

Я

Язык
воспринимаемый программой, 22
допускаемый автоматом, 168
допускаемый машиной Тьюринга, 43
класса *NP*-TIME, 430
класса *P*-TIME, 430
контекстный, 467
полиномиально преобразуемый, 431
полный, 431
порождаемый грамматикой, 94
регулярный, 367
рекурсивно перечислимый, 22
рекурсивный, 22
NP-полный, 431

У цій монографії, що стала класикою, викладаються результати теоретичних і прикладних досліджень з побудови та аналізу ефективних обчислювальних алгоритмів. Розглянуті задачі пошуку, сортування масивів, множення цілих чисел, множення матриць, алгоритми на графах, а також основи теорії складності. Книга призначені для фахівців з комп'ютерних наук і програмістів, а також буде корисна студентам і аспірантам, що спеціалізуються в галузі інформатики.

Науково-популярне видання

Ахо, Альфред В., Хопкрофт, Джон Е., **Ульман**, Джеффрі Д.

Розробка та аналіз комп'ютерних алгоритмів

(Рос. мовою)

Зав. редакцією *С. М. Тригуб*

Із загальних питань звертайтеся до видавництва "Діалектика" за адресою:
info@dialektika.com, http://www.dialektika.com

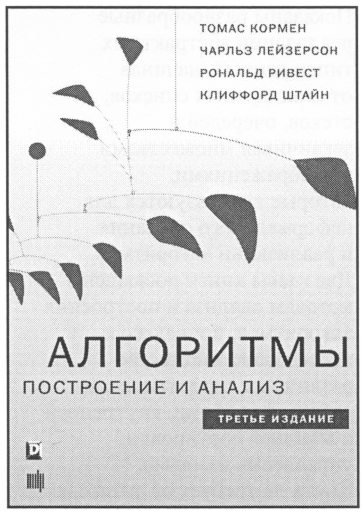
Підписано до друку 08.04.2021. Формат 60x90/16
Ум. друк. арк. 34,0. Обл.-вид. арк. 26,8
Зам. № 20-3100

Видавець ТОВ "Комп'ютерне видавництво "Діалектика"
03164, м. Київ, вул. Генерала Наумова, буд. 23-Б.
Свідоцтво суб'єкта видавничої справи ДК № 6758 від 16.05.2019.

Надруковано ТОВ "АЛЬФА ГРАФІК "
03067, м. Київ, вул. Машинобудівна, 42
Свідоцтво суб'єкта видавничої справи ДК № 6838 від 09.07.2019.

Алгоритмы: построение и анализ *3-е издание*

Томас Кормен,
Чарльз Лейзерсон,
Рональд Ривест,
Клиффорд Штайн



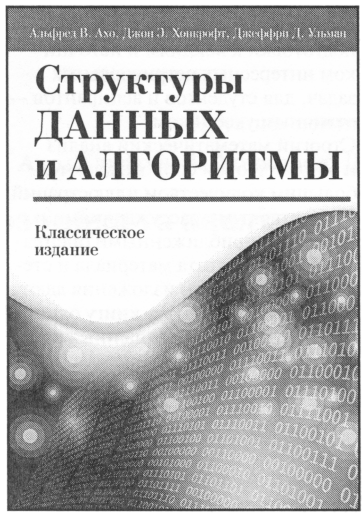
www.dialektika.com

Фундаментальный труд известных специалистов в области информатики достоин занять место на полке любого человека, чья деятельность так или иначе связана с вычислительной техникой и алгоритмами.
Для профессионала эта книга может служить настольным справочником, для преподавателя — пособием для подготовки к лекциям и источником интересных нетривиальных задач, для студентов и аспирантов — отличным учебником.
Строгий математический анализ и обилие теорем сопровождаются большим количеством иллюстраций, элементарными рассуждениями и простыми приближенными оценками. Широта охвата материала и степень строгости его изложения дают основания считать эту книгу одной из лучших книг, посвященных разработке и анализу алгоритмов.
Третье издание этого классического труда в большой степени доработано. В нем появились новые главы, в том числе посвященные такой важной в последнее время теме, как многопоточные алгоритмы, а старые подверглись переработке, местами весьма существенной, когда уже имевшийся во втором издании материал излагается с иных позиций, чем ранее.
Данная книга будет не лишней как на столе студента и аспиранта, так и на рабочей полке практикующего программиста.

ISBN 978-5-907114-11-1 **в продаже**

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

Альфред В. Ахо
Джон Э. Хопкрофт
Джерри Д. Ульман



В этой книге подробно рассмотрены структуры данных и алгоритмы, которые являются фундаментом современной методологии разработки программ. Показаны разнообразные реализации абстрактных типов данных, начиная от стандартных списков, стеков, очередей и заканчивая множествами и отображениями, которые используются для неформального описания и реализации алгоритмов. Две главы книги посвящены методам анализа и построения алгоритмов; приведено и исследовано множество различных алгоритмов для работы с графами, внутренней и внешней сортировки, управления памятью. Книга не требует от читателя специальной подготовки, только предполагает его знакомство с какими-либо языками программирования высокого уровня, такими как Pascal.16

www.williamspublishing.com

ISBN 978-5-6041393-6-3 **в продаже**

Разработка и анализ компьютерных алгоритмов

В книге описаны фундаментальные принципы построения алгоритмов, лежащих в основе всех компьютерных наук. В ней рассматриваются базовые структуры данных и методики программирования, применяемые при создании эффективных алгоритмов. В начале книги вы познакомитесь со списками, очередями, стеками, деревьями и графами. В последующих главах исследуются методы сортировки и поиска, а также алгоритмы на графах нахождения кратчайшего пути и алгоритмы Штрассена умножения матриц. В конце каждой главы приведены большое количество интересных упражнений разного уровня сложности.

Альфред В. Ахо — сотрудник компании Bell Telephone Laboratories в Мюррей Хилл, шт. Нью-Джерси, председатель программного комитета по компьютерным наукам в технологическом институте Стивенса и вице-президент специальной группы ACM по теории автоматов и вычислительным алгоритмам. Он — автор книг *The Theory of Parsing, Translation, and Computing, Volumes 1 and 2*, и *Theory of Computing*. Доктор Ахо получил степень бакалавра в университете Торонто, а магистерскую и докторскую степени — в Принстонском университете.

Джон Э. Хопкрофт — профессор факультета компьютерных наук Корнеллского университета, член Национального научного фонда в области компьютерных наук и ответственный редактор *SIAM Journal of Computing*. Он работал научным консультантом в компаниях Bell Telephone Laboratories и System Development Corporation. Доктор Хопрофт — один из авторов книги *Formal Languages and Their Relations to Automata* (Addison-Wesley, 1969). Он получил магистерскую и докторскую степени в Стэнфордском университете.

Джеффри Д. Ульман — профессор электротехники в Принстонском университете. Ранее работал в компании Bell Telephone Laboratories. Он один из авторов двухтомника *The Theory of Parsing, Translation, and Computing* вместе с Альфредом Ахо. Доктор Ульман получил степень бакалавра в Колумбийском университете, а докторскую степень — в Принстонском университете.

Изображение на обложке:
©Depositphotos.com/199945068
Автор: klss777

Категория: компьютерные науки/алгоритмы

ISBN 978-617-7874-04-0

Pearson
https://www.pearson.com/

Addison-Wesley
Addison-Wesley Publishing Company, Inc.

Комп'ютерне видавництво "ДІАЛЕКТИКА"
www.dialektika.com