Алгоритмы
Введение в разработку и анализ

Introduction to The Design & Analysis of Algorithms

Anany Levitin
Villanova University

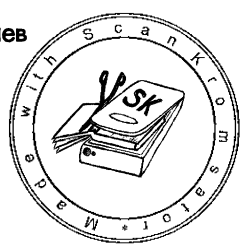
ADDISON-WESLEY

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Алгоритмы
Введение в разработку и анализ

Ананий Левитин
Университет Вилланова

Москва • Санкт-Петербург • Киев
2006



ББК 32.973.26–018.2.75
Л36
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. физ.-мат.наук *С.Г. Тригуб*,
канд.техн.наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Левитин, Ананий В.

Л36 Алгоритмы: введение в разработку и анализ. : Пер. с англ. — М. : Издательский дом "Вильямс", 2006. — 576 с. : ил. — Парал. тит. англ.

ISBN 5–8459–0987–2 (рус.)

Эта книга, автором которой является преподаватель информатики, представляет собой один из лучших учебников, посвященных алгоритмам. Делая основной упор на понимание идей, а не на механическое рассмотрение работы того или иного алгоритма, автор излагает принципы разработки алгоритмов так, что они могут быть применены как универсальный инструментарий для широкого диапазона задач, а не только для разработки алгоритмов.

Книга ориентирована в первую очередь на студентов и аспирантов соответствующих специальностей, поэтому для преподавателей она может стать хорошим пособием для подготовки к лекциям и источником интересных нетривиальных задач. Книга может оказаться полезной и профессионалам в области разработки алгоритмов благодаря использованному автором новому подходу к классификации методов проектирования. Описание алгоритмов на естественном языке дополняется псевдокодом, который позволяет каждому, кто имеет хотя бы начальные знания и опыт программирования, реализовать алгоритм на используемом им языке программирования.

ББК 32.973.26–018.2.75

ISBN 5–8459–0987–2 (рус.)
ISBN 0–201–74395–7 (англ.)

Оглавление

Содержание

С глубочайшей признательностью
Марии и Мириам

Предисловие

> Самое ценное в научном или техническом образовании — это развитие универсального мыслительного аппарата, который будет служить вам на протяжении всей жизни.
>
> — Джордж Форсайт (George Forsythe),
> "Что предпринять до прихода специалиста по вычислительной технике" ("What to do till the computer scientist comes") (1968)

Алгоритмы имеют первостепенное значение как в научной, так и в технической сфере. Осознание данного факта привело к появлению огромного количества книг, посвященных этому предмету. Вообще говоря, в плане представления алгоритмов все книги можно разделить на две большие группы. В одной из них алгоритмы классифицируются в соответствии с типом решаемой задачи. Как правило, в таких книгах алгоритмам сортировки, поиска, обработке графов и т.п. посвящены отдельные главы. Преимущество такого подхода заключается в том, что он позволяет непосредственно оценить эффективность применения различных алгоритмов для решения задачи. Недостаток же состоит в том, что при таком подходе акцент делается на решении самой задачи, а не на методологии проектирования алгоритма.

При использовании второго, альтернативного подхода основное внимание уделяется методике проектирования алгоритма. В таких книгах алгоритмы, относящиеся к различным областям вычислительной техники, группируются, если при их проектировании использованы одинаковые подходы. И здесь я также разделяю сложившееся мнение [11] по поводу того, что подобная организация книги больше всего подходит для основного курса, посвященного проектированию и анализу алгоритмов. Имеется несколько причин для того, чтобы сосредоточиться на методологии проектирования алгоритмов. Во-первых, учащиеся смогут применить ее при разработке алгоритмов для решения неизвестной задачи. Во-вторых, они смогут классифицировать все множество известных алгоритмов согласно лежащей в их основе идее проектирования. Основной целью образования в области информатики должно быть изучение общих идей проектирования алгоритмов,

относящихся к различным прикладным областям. В конце концов, при изучении любой научной дисциплины основное внимание уделяется рассмотрению системы ее основополагающих понятий. В-третьих, по моему мнению, изучение методологии проектирования алгоритмов имеет огромную важность, поскольку дает ключ к пониманию методики поиска общего решения задач в области информатики.

Существует несколько учебников, материал которых структурирован в соответствии с упомянутой выше методологией проектирования алгоритмов (в частности, [22, 54, 82]). На мой взгляд, все они имеют один недостаток: их авторы слепо следуют одной и той же классификации методик проектирования алгоритмов. Данную классификацию нельзя считать удачной, поскольку она имеет несколько серьезных недостатков как с теоретической, так и с образовательной точек зрения. Один из основных ее недостатков заключается в том, что она не позволяет классифицировать большое количество важных алгоритмов. Все это вынуждает авторов существующих учебников, отклоняясь от рассмотрения методологии проектирования, включать в них главы, посвященные решению конкретных задач. К сожалению, подобный уход от основной темы приводит к потере логики изложения курса и практически всегда запутывает учащихся.

Новая таксономия методологии проектирования алгоритмов

Описанные выше недостатки существующей системы классификации вынудили меня разработать новую таксономию[1] методологии проектирования алгоритмов [74], которая и была положена в основу этой книги. Позволю себе перечислить основные преимущества новой таксономии.

- Новая таксономия является более полной, чем существующая система классификации. Она включает ряд стратегий — решение задачи "в лоб", методом разделения, методом преобразования; компромисс между временем выполнения алгоритма и объемом оперативной памяти, — которые не так часто относят к важным примерам проектирования.
- Новая таксономия естественным образом охватывает большое количество классических алгоритмов, которые традиционная система не в состоянии классифицировать. Достаточно привести лишь несколько названий: алгоритм Евклида, пирамидальная сортировка, дерево поиска, хеширование, топологическая сортировка, схема Горнера. В результате,

[1] Теория классификации и систематизации сложноорганизованных областей действительности, имеющих обычно иерархическое строение (органический мир, объекты географии, геологии, языкознания, этнографии). *Современный словарь иностранных слов*, 3-е изд., М.: Рус. яз., 2000. — *Прим. ред.*

появляется возможность представить стандартный набор классических алгоритмов в единообразной и понятной форме.

- Она естественным образом приспособлена к существующему разнообразию основных методик проектирования.
- Она лучше всего подходит для анализа эффективности с помощью аналитических методов (см. приложение Б).

Методология проектирования как стратегия решения общих задач

Описанные в книге методики проектирования алгоритмов, как правило, применяются для решения классических задач вычислительной техники. Однако здесь есть одно новшество. В книгу включен материал, посвященный алгоритмам решения численных задач, описанный в рамках той же принятой структуры. (Включение этих алгоритмов было одобрено учебным планом по информатике за 2001 год *Computing Curricula 2001* [29] — разработанным с учетом современных требований.) Тем не менее описанные в книге методики проектирования можно считать универсальным средством решения задач; их применение не ограничивается только традиционными задачами вычислительной техники и математики. Важность этого подтверждается двумя факторами. Во-первых, все больше компьютерных приложений выходят за рамки традиционной области их применения, поэтому можно надеяться, что такая тенденция сохраниться и в будущем. Во-вторых, основной целью университетского образования считается выработка у студентов навыков самостоятельного решения поставленных задач. Поэтому среди всех курсов, читаемых в рамках программы по информатике, именно курс, посвященный разработке и анализу алгоритмов, как нельзя лучше подходит для этой цели, поскольку он развивает у студентов специфические навыки решения задач. Но это отнюдь не означает, что данный курс следует рассматривать как курс, посвященный решению общих задач. Тем не менее я считаю, что не следует упускать представившуюся вам уникальную возможность изучения методик разработки и анализа алгоритмов. Чтобы достичь поставленной цели, в книгу включены примеры головоломок и игр, построенных на их основе. Хотя идея изучения алгоритмов на основе головоломок, конечно, не нова, в этой книге сделана попытка систематизировать этот процесс, не ограничиваясь несколькими стандартными примерами.

Педагогика учебника

При написании книги я не ставил перед собой цель упростить излагаемый материал, но в то же время хотел сделать так, чтобы он был доступен для понимания большинству студентов во время самостоятельной работы. Поэтому ниже я приведу несколько отличительных особенностей книги, способствующих достижению моего замысла.

- Разделяя мнение Джорджа Форсайта (см. эпиграф), я постарался подчеркнуть основные идеи, лежащие в основе разработки и анализа алгоритмов. Поэтому, отбирая конкретные алгоритмы для иллюстрации этих идей, я ограничил их круг и привел в книге описание только тех из них, которые лучше всего проясняют основные подходы к проектированию или методы анализа алгоритмов. К счастью, большинство классических алгоритмов удовлетворяет этому критерию.
- В главе 2, посвященной анализу эффективности, мы будем различать методы анализа нерекурсивных алгоритмов и методы, которые обычно используются для анализа рекурсивных алгоритмов. В главу также включен раздел, посвященный эмпирическому анализу и алгоритмам визуализации.
- В текст всех глав включены вопросы к читателю. Часть из них носит риторический характер, и заданы они только потому, что имеют непосредственное отношение к излагаемой теме и могут рассеять сомнения читателей. Поэтому ответы на них даются сразу. Остальные вопросы предназначены для проверки полученных знаний. Правильный ответ на них является индикатором достаточного уровня знаний у читателя и сигналом к тому, что можно переходить к изучению следующих разделов книги.
- В конце каждой главы приведено краткое резюме, в котором подытожены основные понятия и результаты, рассмотренные в главе.
- В книгу включено порядка 600 упражнений. Часть из них — учебные. В остальных отрабатываются важные моменты, о которых шла речь в материале главы, или описываются алгоритмы, не упоминавшиеся ранее. В некоторых упражнениях используются ресурсы Internet. Ряд упражнений предназначен для подготовки читателя к восприятию материала, описанного в последующих разделах книги.

Упражнения, содержащие игры, головоломки и вопросы, построенные на их основе, отмечены специальной пиктограммой.

- В книге приведены советы и подсказки по решению всех упражнений. Дополнительную информацию читатели могут получить по адресу `http://www.aw.com/cssupport`.

Необходимые условия для чтения книги

Для того чтобы понимать материал книги, читатель должен прослушать начальный курс по программированию и один из стандартных курсов по дискретным структурам. Такого багажа знаний будет вполне достаточно для свободного освоения описанного здесь материала. Тем не менее основные структуры данных, необходимые формулы суммирования и рекуррентные соотношения описаны в разделе 1.4, приложениях А и Б, соответственно. Численные методы используются только в трех разделах (2.2, 10.4 и 11.4), и то в очень ограниченной степени. Поэтому если у читателя нет твердых знаний в области численных методов, он вполне может пропустить эти три раздела, и это не осложнит понимания остального материала книги.

Использование книги в учебном процессе

Материал этой книги можно использовать в качестве основы учебного курса, посвященного проектированию и анализу алгоритмов, с упором на методику проектирования алгоритмов. Его вполне достаточно для стандартного односеместрового курса. Вообще говоря, часть материала глав с 3 по 11 можно смело пропустить, и это никак не скажется на понимании читателями последующих глав книги. Любую из частей книги можно отдать для самостоятельной работы. В частности, изучение разделов 2.6 и 2.7, посвященных, соответственно, эмпирическому анализу и алгоритмам визуализации, можно совместить с выполнением программных проектов.

В табл. приведен примерный план 80-часового (40-лекционного) односеместрового учебного курса.

Примерный план 80-часового односеместрового учебного курса

Номер лекции	Тема	Разделы
1, 2	Введение	1.1–1.3
3, 4	Изучение основ; условные обозначения O, Θ и Ω	2.1, 2.2
5	Математический анализ нерекурсивных алгоритмов	2.3
6, 7	Математический анализ рекурсивных алгоритмов	2.4, 2.5 и приложение Б
8	Алгоритмы решения задач “в лоб”	3.1, 3.2 и раздел 3.3
9	Поиск методом перебора	3.4

Номер лекции	Тема	Разделы
10–12	Алгоритмы разбиения: сортировки слиянием, быстрой сортировки и двоичного поиска	4.1–4.3
13	Другие примеры алгоритмов декомпозиции	4.4 или 4.5 или 4.6
14–16	Алгоритмы уменьшения на единицу: сортировка методом вставки, поиск в глубину и ширину, топологическая сортировка	5.1–5.3
17	Алгоритмы уменьшения на постоянное значение	5.5
18	Алгоритмы уменьшения на переменное значение	5.6
19–21	Упрощение экземпляра, предварительная сортировка, исключение Гаусса, сбалансированные деревья поиска	6.1–6.3
22	Изменение представления: множества и пирамидальная сортировка	6.4
23	Изменение представления: схема Горнера и двоичный порядок	6.5
24	Приведение задачи	6.6
25–27	Компромисс между временем выполнения алгоритма и объемом оперативной памяти: поиск строк, хеширование, B-деревья	7.2–7.4
28–30	Алгоритмы динамического программирования	три из 8.1–8.4
31–33	Жадные алгоритмы: Прима, Крускала, Дейкстры, Хаффмана	9.1–9.4
34	Доказательство нижней границы	10.1
35	Деревья принятия решений	10.2
36	R, NP и NP-полные задачи	10.3
37	Алгоритмы численных методов	10.4 и 11.4
38	Поиск с возвратом	11.1
39	Метод ветвей и границ	11.2
40	Приближенные алгоритмы решения NP-сложных задач	11.3

Благодарности

Прежде всего хочу выразить признательность авторам других книг, посвященных алгоритмам, чья интуиция и идеи изложения материала прямо или косвенно помогли мне. Советы и критические замечания рецензентов, несомненно, существенно улучшили содержание книги. Выражаю благодарность Саймону Берковичу (Simon Berkovich) из университета Джорджа Вашингтона (George Washington University), Ричарду Бори (Richard Borie) из университета штата Алабама (University of Alabama), Дугласу М. Кэмпбеллу (Douglas M. Campbell) из университета Брайхем Янг (Brigham Young University), Бину Конгу (Bin Cong) из университета штата Калифорния в Фуллертоне (California State University, Fullerton), Стиву Хомеру (Steve Homer) из Бостонского университета (Boston University), Роланду Хабшеру (Roland Hubscher) из университета в Оберне (Auburn University), Сухамею Кунду (Sukhamay Kundu) из университета штата Луизиана (Louisiana State University), Ши-Донг Лангу (Sheau-Dong Lang) из университета центральной Флориды (University of Central Florida), Джону С. Ласту (John C. Lusth) из Арканзасского университета (University of Arkansas), Джону Ф. Мейеру (John F. Meyer) из Мичиганского университета (University of Michigan), Стивену Р. Зейделю (Steven R. Seidel) из Мичиганского технологического университета, Али Шокофандеху (Ali Shokoufandeh) из Дрексельского университета (Drexel University) и Джорджу Х. Вильямсу (George H. Williams) из колледжа в Юнионе.

Чрезвычайно признателен моей коллеге Мэри-Анджеле Папаласкари (Mary-Angela Papalaskari) за то, что она использовала текст рукописи этой книги при преподавании курса, посвященного алгоритмам, в университете г. Вилланова (Villanova University) и внесла существенные улучшения в текст книги и упражнения. Она с большим энтузиазмом поддержала идею широкого использования в книге различных головоломок. Еще один коллега, Джон Матулис (John Matulis), также использовал текст рукописи книги в учебном процессе и высказал ценные замечания. Помощь в подготовке рукописи оказывал мой бывший студент Эндисва Хейнегг (Andiswa Heinegg). Его критические замечания позволили сделать более понятным содержимое книги и более стройным стиль ее изложения.

На протяжении нескольких последних семестров рукопись этой книги служила в качестве основного учебника для студентов университета в Вилланове, что, несомненно, причиняло им неудобство. Я восхищаюсь их терпением и благодарю за ценные советы и найденные ошибки и опечатки в тексте рукописи. Наверняка в книге остались ошибки, которые я, конечно же, никак не могу отнести на их счет: ошибки вполне могли появиться по моей вине при внесении правок уже после того, как студенты прослушали курс.

Хочу выразить благодарность сотрудникам издательства Addison-Wesley и их коллегам — всем тем, кто работал над моей книгой. Особенно хочу поблагодарить моего редактора Майте Суарез-Риваз (Maite Suarez-Rivas) и ее бывшего заме-

стителя Лайзу Хог (Lisa Hogue) за то, что они разделяли и поддерживали мой энтузиазм в отношении этого проекта.

И в заключение мне хочется выразить глубокую признательность членам моей семьи. Ведь жить под одной крышей с супругом, который постоянно занят тем, что пишет книгу, гораздо тяжелее, чем писать саму книгу. Моя жена Мария пережила два таких года, посильно помогая мне. Помощь была огромной: более 250 иллюстраций к этой книге являются ее творением! Моя дочь Мириам — большой авторитет для меня в области английской прозы. Она не только прочла большие куски этой книги, но и подобрала остроумные эпиграфы к главам. Огромное им спасибо!

Ананий Левитин
anany.levitin@villanova.edu
Август 2002 г.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать любые ваши замечания, касающиеся книги.

Мы ждем ваших комментариев. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер 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.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Информация для писем из:

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

Глава 1

Введение

> В основе мироздания лежат две идеи: исчисление и алгоритм. Исчисление, базирующееся на мощном аппарате математического анализа, является основой современной науки. Однако наука невозможна без алгоритма, лежащего в основе современного мира.
>
> — Дэвид Берлински (David Berlinski),
> "Появление алгоритма" ("The Advent of the Algorithm") (2000)

Зачем изучать алгоритмы? Для настоящего компьютерного профессионала существует две причины: практическая и теоретическая. С практической точки зрения он должен иметь представление о стандартном наборе основных алгоритмов, относящихся к разным областям вычислительной техники. Кроме того, он должен уметь разрабатывать новые алгоритмы и анализировать их эффективность. С теоретической точки зрения процесс изучения алгоритмов, который иногда называют ***алгоритмикой*** (algorithmics), считается краеугольным камнем информатики (computer science). Дэвид Харел (David Harel) в своей великолепной книге, остроумно озаглавленной *Algorithmics: the Spirit of Computing*[1], пишет следующее:

> Алгоритмика — это нечто большее, чем просто раздел информатики. Она является основой информатики и, положа руку на сердце, можно сказать, что она существенно повлияла на современную науку, технику и бизнес [[48], с. 6].

И даже если вы не являетесь студентом вуза, специализирующимся в компьютерных науках, существуют достаточно веские причины для того, чтобы заняться изучением алгоритмов. Попросту говоря, без алгоритмов невозможно создать ни одну компьютерную программу. Поэтому по мере того, как компьютерные программы становятся все более значимыми практически во всех сферах профессиональной деятельности (да и личной жизни!), изучение алгоритмов становится все более и более важной задачей для широкого круга людей.

[1] Это название можно перевести на русский язык как *Алгоритмика: дух вычислений. — Прим. ред.*

Еще одна причина для изучения алгоритмов заключается в том, что этот процесс развивает у учащихся умение аналитически мыслить. В конце концов, алгоритмы можно рассматривать как особый подход к решению задач, когда важны не столько сами ответы, сколько точные инструкции для их получения. Следовательно, специальные методы проектирования алгоритмов можно считать стратегическим планом решения задачи, который пригодится в любом случае, независимо от того, используется компьютер или нет. Само собой разумеется, что круг задач, которые могут быть решены с помощью какого-либо алгоритма, по существу зависит от точности алгоритмического мышления. Например, невозможно сформулировать алгоритм, который позволит прожить счастливую жизнь или стать богатым и знаменитым. С другой стороны, выдвинутое требование точности имеет важное преимущество с точки зрения обучения. Дональд Кнут, один из выдающихся ученых в области информатики и истории алгоритмики, пишет следующее:

> Хорошо обученный в области информатики специалист обязан знать, как работать с алгоритмами: как их создавать, изменять, понимать и анализировать. Эти знания позволят не только писать хорошие компьютерные программы, но и станут основой универсального мыслительного аппарата, который окажет неоценимую помощь при постижении других наук, будь то химия, лингвистика, музыка и т.д. Причину этого можно объяснить следующим образом: часто говорят, что человек ничего не понимает, пока не объяснит это кому-то другому. Я бы перефразировал это так: человек **глубоко** не понимает предмет до тех пор, пока не научит этому **компьютер**, т.е. выразит что-либо в виде алгоритма... Попытка формализовать нечто в виде набора алгоритмов приводит к более глубокому пониманию сути вещей, чем при их осмыслении традиционным способом [[64], с. 9].

О том, что такое алгоритм речь пойдет в разделе 1.1. Там в качестве примеров мы рассмотрим три алгоритма решения одной и той же задачи — поиска наибольшего общего делителя (НОД). Я выбрал эту задачу по трем причинам. Во-первых, она знакома практически каждому со времени обучения в средней школе. Во-вторых, она позволяет продемонстрировать важный принцип: то, что одну и ту же задачу можно решить с помощью нескольких алгоритмов. Вполне естественно, что эти алгоритмы отличаются по своей сути, уровню сложности и эффективности. В-третьих, один из этих алгоритмов вполне достоин того, чтобы быть представленным первым, — как по "возрасту" (впервые он был описан в известном трактате Евклида более двух тысяч лет тому назад), так и по неисчерпаемым возможностям и важности. Наконец, метод определения НОД, которому учат в школе, позволит продемонстрировать одно из важных условий, которому должен удовлетворять любой алгоритм.

В разделе 1.2 рассматривается решение алгоритмической задачи. Там мы обсудим несколько важных моментов, связанных с разработкой и анализом алго-

ритмов. К различным аспектам решения алгоритмической задачи относятся как анализ задачи и средства выражения алгоритма, так и методы оценки его правильности и эффективности. В этом разделе вы не найдете описания магического средства для создания алгоритма решения *любой* задачи. Должно быть уже понятно, что такого средства попросту не существует. Тем не менее материал раздела 1.2 поможет вам систематизировать проектирование и анализ алгоритмов.

Раздел 1.3 посвящен некоторым типам задач, считающимся особенно важными при изучении алгоритмов, и их применению. На самом деле существует ряд учебников, материал которых структурирован по типу решаемых задач. Однако мы придерживаемся иного мнения, с которым согласны многие преподаватели: структурирование материала на основе метода проектирования алгоритма — гораздо важнее. Тем не менее очень важно разбираться в основных типах задач и не только потому, что они находят широкое применение в реальной жизни. В этой книге они используются для демонстрации особенностей метода проектирования алгоритмов.

В разделе 1.4 приведен обзор основных структур данных. Однако его следует рассматривать скорее как справочник, а не подробное руководство. Если же вам нужна детальная информация по этой теме, обратитесь к дополнительной литературе. Хороших книг, посвященных этой теме, написано довольно много, однако в большинстве из них акцент сделан на том или ином языке программирования.

1.1 Понятие алгоритма

Что же такое алгоритм? Универсального определения этого понятия нет, однако существует общее мнение по поводу того, что оно должно означать:

> **Алгоритм** — это последовательность четко определенных инструкций, предназначенных для решения некоторой задачи. Другими словами, это последовательность команд, позволяющих получить из корректных входных данных требующиеся выходные данные за ограниченный промежуток времени.

Это определение можно проиллюстрировать с помощью простой схемы (рис. 1.1).

Упомянув в приведенном выше определении слово "инструкции", мы подразумевали, что существует некоторое абстрактное устройство (или человек), способное распознать эти инструкции и выполнить предписываемые ими действия. На рис. 1.1 это устройство названо "вычислительным". Однако следует иметь в виду, что до появления компьютеров под термином "вычислительное устройство" понимался человек, выполняющий числовые расчеты. Естественно, если речь идет о современности, под словом "вычислительное устройство" понимается компьютер, т.е. популярное электронное устройство, которое практически повсеместно

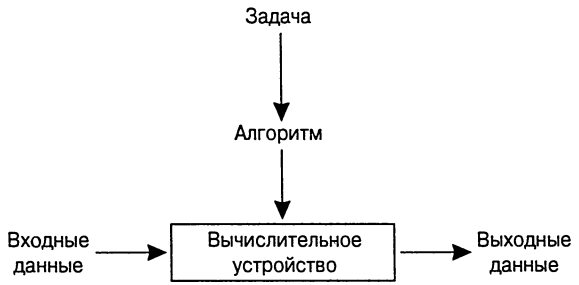


Рис. 1.1. Иллюстрация понятия алгоритма

вторглось в нашу жизнь. Тем не менее обратите внимание на то, что, хотя бóльшая часть алгоритмов в конечном счете предназначена для реализации на компьютере, само понятие алгоритма никак не связано с этим допущением.

В этом разделе в качестве примеров, иллюстрирующих понятие алгоритма, мы рассмотрим три способа решения одной и той же задачи — поиска НОД двух целых чисел. Эти примеры помогут нам проиллюстрировать перечисленные ниже важные моменты.

- Каждый шаг алгоритма должен быть четко и однозначно определен. Это требование является обязательным и не должно нарушаться ни при каких обстоятельствах.
- Должны быть точно указаны диапазоны допустимых значений входных данных, которые обрабатываются с помощью алгоритма.
- Один и тот же алгоритм можно представить несколькими разными способами.
- Для решения одной и той же задачи может существовать несколько разных алгоритмов.
- В основу алгоритмов для решения одной и той же задачи могут быть положены совершенно разные принципы, что может существенно повлиять на скорость решения этой задачи.

Обозначим функцию поиска НОД двух неотрицательных целых чисел m и n (причем m и n не могут одновременно равняться нулю) через $\gcd(m, n)$.[2] По определению эта функция должна найти наибольшее целое число, которое делится без остатка как на m, так и на n. Древнегреческий математик Евклид из Александрии (III в до н.э.), который прославился тем, что впервые систематически изложил курс геометрии, описал алгоритм решения этой задачи в одном из своих трудов под названием *Начала*. Выражаясь современным языком, ***алгоритм***

[2] От английского "greatest common divisor". — *Прим. ред.*

Евклида основан на рекуррентном вычислении следующего равенства:

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Здесь выражение $(m \bmod n)$ является остатком от деления m на n. Выполнение алгоритма заканчивается, когда выражение $(m \bmod n)$ становится равным нулю. Поскольку $\gcd(m, 0) = m$ (понятно, почему?), последнее полученное значение m будет также являться НОД исходных чисел m и n.

Например, вычисление НОД пары чисел (60, 24) можно выполнить следующим образом:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

(Если этот алгоритм не произвел на вас впечатления, попытайтесь определить НОД двух бóльших чисел, например таких, которые используются в задаче 4 упражнения 1.1.)

Ниже приводится более структурированное описание рассматриваемого нами алгоритма.

Вычисление НОД чисел m и n при помощи алгоритма Евклида

Шаг 1 Если $n = 0$, вернуть m в качестве ответа и закончить работу; иначе перейти к шагу 2.

Шаг 2 Поделить нацело m на n и присвоить значение остатка переменной r.

Шаг 3 Присвоить значение n переменной m, а значение r — переменной n. Перейти к шагу 1.

В качестве альтернативы запишем тот же алгоритм в виде псевдокода.

Алгоритм $Euclid(m, n)$

```
// Алгоритм Евклида вычисляет значение функции gcd(m, n)
// Входные данные:  два неотрицательных целых числа m и n,
//                  которые не могут одновременно быть равны нулю
// Выходные данные: наибольший общий делитель чисел m и n
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

Можем ли мы убедиться, что в конечном счёте выполнение алгоритма Евклида завершится? Это следует из констатации следующего факта: на каждом шаге итерации значение второго числа пары (n) будет уменьшаться, причем, по определению, оно не может быть меньше нуля. В самом деле, новое значение

числа n, получаемое на следующей итерации в результате вычисления выражения $(m \bmod n)$, будет всегда меньше, чем предыдущее значение числа n. Следовательно, рано или поздно значение второго числа пары станет равным 0, и выполнение алгоритма завершится.

Как и при решении большинства других задач, существует несколько алгоритмов вычисления НОД. Давайте рассмотрим два других способа решения этой задачи. Первый из них основан на подборе наибольшего целого числа — такого, чтобы числа m и n делились на него без остатка. Очевидно, что такой общий делитель не может быть больше наименьшего из чисел пары, которое можно записать как $t = \min\{m, n\}$. Поэтому выполнение алгоритма можно начать с проверки того, делятся ли оба числа, m и n, на t без остатка. Если это так, то число t является ответом; если нет, нужно уменьшить значение t на единицу и снова выполнить проверку. (Можем ли мы убедиться, что в конечном итоге этот процесс завершится?). Например, для рассмотренной выше пары чисел $(60, 24)$, выполнение алгоритма начинается с проверки числа 24, затем — 23 и т.д. до тех пор, пока значение числа t не станет равным 12, после чего алгоритм должен завершить свою работу.

Вычисление НОД чисел m и n методом последовательного перебора

Шаг 1 Присвоить значение функции $\min\{m, n\}$ переменной t.

Шаг 2 Разделить m на t. Если остаток равен нулю, перейти к шагу 3; иначе перейти к шагу 4.

Шаг 3 Разделить n на t. Если остаток равен нулю, вернуть t в качестве ответа и закончить работу; иначе перейти к шагу 4.

Шаг 4 Вычесть 1 из t. Перейти к шагу 2.

Обратите внимание, что в отличие от алгоритма Евклида, рассматриваемый нами алгоритм поиска НОД в описанной выше форме не будет корректно работать, если хотя бы один из его входных параметров равен нулю. Таким образом, этот пример позволяет показать, почему так важно явно определять диапазоны допустимых значений входных параметров алгоритма.

Третий способ поиска НОД должен быть вам знаком из курса средней школы.

Вычисление НОД чисел m и n "школьным" методом

Шаг 1 Разложить на простые множители число m.

Шаг 2 Разложить на простые множители число n.

Шаг 3 Для простых множителей чисел m и n, найденных на шаге 1 и 2, выделить их общие делители. (Если p является общим делителем чисел m и n и встречается в их разложении на простые множители, соответственно, p_m и p_n раз, то при выделении нужно повторить это $\min\{p_m, p_n\}$ раз.)

Шаг 4 Вычислить произведение всех выделенных общих делителей и вернуть его в качестве результата поиска НОД двух указанных чисел.

Таким образом, для рассмотренной выше пары чисел $(60, 24)$, получим:

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

Ностальгия по школьным годам не должна помешать нам сделать вывод о том, что последний из рассмотренных способов поиска НОД гораздо сложнее и медленнее, чем алгоритм Евклида. (Методы определения и сравнения времени выполнения алгоритмов будут описаны в следующей главе.) Но даже не принимая во внимание его низкую эффективность, метод определения НОД, изучаемый в средней школе, нельзя считать законным алгоритмом (по крайней мере в представленном здесь виде). Вы спросите, почему? Все дело в том, что этапы разложения на простые множители не определены однозначно. Для их выполнения требуется иметь список простых чисел, а я почему-то уверен, что школьный учитель не объяснил вам, как составить такой список. Вы, конечно же, можете возразить, что это не повод для подобных придирок. Однако до тех пор, пока этот момент не будет прояснен, мы не сможем, например, написать на каком-либо из языков программирования программу, реализующую этот алгоритм. (Кстати, с этой точки зрения шаг 3 также недостаточно четко определен. Однако его неопределенность гораздо легче прояснить, чем этапы разложения на простые множители. И еще, как вы собираетесь выделять общие элементы из двух списков?)

Итак, подошло время рассмотреть несложный алгоритм генерации последовательности простых чисел, не превышающих произвольно заданного целого числа n. Вероятнее всего он был придуман в древней Греции и поэтому назван ***решетом Эратосфена*** (прим. 200 год до н.э.). Для начала составим список, содержащий последовательность целых чисел от 2 до n, из которого затем мы должны будем выбрать простые числа. Далее, на первом проходе алгоритма нужно удалить из списка все числа, которые делятся на 2, т.е. 4, 6 и т.д. Затем необходимо выбрать из списка следующий элемент (в данном случае это 3) и удалить из списка все числа, которые делятся на него. (В описываемой простой версии алгоритма существует небольшая накладка, поскольку некоторые из чисел, например 6, должны удаляться из списка более одного раза.) Для числа 4 не нужно выполнять специальный проход по списку, так как число 4 кратно 2, поэтому оно уже будет удалено из списка на первом проходе. (Точно так же в этом алгоритме не нужно выполнять проход и для всех других чисел, которые были удалены из списка на предыдущих проходах.) Следующим числом, которое осталось в списке и используется на третьем проходе, является 5. Работа алгоритма продолжается так до тех пор, пока

в списке существуют числа, которые можно удалить. Числа, оставшиеся в списке после выполнения алгоритма, являются простыми.

В качестве примера использования описанного выше алгоритма рассмотрим процесс поиска простых чисел, не превышающих $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Для нашего примера не требуется большего количества проходов по списку, поскольку из него на рассмотренных проходах были удалены все составные числа. Таким образом, в списке остались только упорядоченные простые числа, меньшие или равные 25.

Возникает общий вопрос: при каком наибольшем значении числа p в списке еще остаются кратные ему числа? Прежде чем ответить на него, заметим, что, если p является числом, кратные которому числа должны быть удалены из списка на текущем проходе, то начать рассмотрение следует с числа $p \cdot p$, поскольку все меньшие кратные ему числа — $2p, \ldots, (p-1)\,p$ — уже были удалены из списка на предыдущих проходах. Это наблюдение позволит избежать удаления из списка одного и того же числа более одного раза. Очевидно, что $p \cdot p$ не должно быть больше, чем n, поэтому p не может превышать значения $\sqrt{n}$, округленного до целого числа в нижнюю сторону. На математическом языке это записывается как $\lfloor\sqrt{n}\rfloor$, т.е. с помощью так называемой функции "пол" (округления вниз, floor). В приведенном ниже псевдокоде алгоритма предполагается, что для вычисления используется готовая функция $\lfloor\sqrt{n}\rfloor$. Как альтернативный вариант можно в качестве условия продолжения вычисления цикла проверять значение неравенства $p \times p \leqslant n$.

АЛГОРИТМ *Sieve* (n) (РЕШЕТО ЭРАТОСФЕНА)

```
// Реализация решета Эратосфена
// Входные данные:  Положительное целое число n ⩾ 2
// Выходные данные: Массив L простых чисел, меньших или
//                  равных n
for p ← 2 to n do
    A[p] ← p
for p ← 2 to ⌊√n⌋ do          // См. абзац перед псевдокодом
    if A[p] ≠ 0               // Элементы, кратные p еще не были
        j ← p * p             // удалены на предыдущих проходах
        while j ⩽ n
            A[j] ← 0          // Пометим этот элемент и все
            j ← j + p         // последующие, кратные p, как удаленные
```

```
// Скопируем оставшиеся элементы массива A в массив
// простых чисел L
i ← 0
for p ← 2 to n do
  if A[p] ≠ 0
    L[i] ← A[p]
    i ← i + 1
return L
```

Итак, теперь можно объединить алгоритм решета Эратосфена с методом, который вы проходили в средней школе, получив совершенно законный алгоритм поиска НОД двух положительных чисел. Обратите внимание, что в этом алгоритме нужно предусмотреть частный случай, когда один или оба входных параметра равны 1. Поскольку математики не считают число 1 простым, то, строго говоря, этот метод и не должен работать в подобном случае.

Прежде чем закончить этот раздел, необходимо сделать еще одно важное замечание. Примеры, рассмотренные в этом разделе, не относятся к математическим задачам, несмотря на то, что большинство этих алгоритмов широко используются, а некоторые даже реализованы в виде компьютерных программ. Умение раскладывать задачи на алгоритмы пригодится в решении повседневных рутинных проблем, возникающих как на работе, так и дома. Вполне возможно, что, осознав важность алгоритмов в современном мире, вам захочется еще больше узнать об этом замечательном двигателе информационного века.

Упражнения 1.1

1. Ознакомьтесь с учениями Аль Хорезми — человека, от имени которого произошло слово "алгоритм". В частности, узнайте, что общего в словах "алгоритм" и "алгебра".

2. Известно, что основной целью деятельности патентной системы США является содействие "прикладным искусствам". Как вы думаете, можно ли запатентовать алгоритмы в этой стране?

3. а) Составьте подробную инструкцию (как это делается при описании алгоритма) вашего возвращения из института домой.

б) Составьте подробный рецепт приготовления вашего любимого блюда (как это делается при описании алгоритма).

4. а) Найдите значение функции $\gcd(31415, 14142)$ при помощи алгоритма Евклида.

б) Оцените, во сколько раз быстрее вычисляется значение функции $\gcd(31415, 14142)$ при помощи алгоритма Евклида по сравнению

с алгоритмом последовательного перебора чисел сверху вниз от значения $\min\{m, n\}$ до значения $\gcd(m, n)$.

5. Докажите, что для любых двух положительных чисел m и n выполняется равенство $\gcd(m, n) = \gcd(n, m \bmod n)$.

6. Что произойдет, если при использовании алгоритма Евклида первое из чисел будет меньше второго? Каково будет максимальное время выполнения алгоритма при подстановке таких входных данных?

7. **а)** При каких значениях входных параметров m и n в алгоритме Евклида выполняется минимальное количество операций деления, при условии, что $1 \leqslant m, n \leqslant 10$?

   б) При каких значениях входных параметров m и n в алгоритме Евклида выполняется максимальное количество операций деления, при условии, что $1 \leqslant m, n \leqslant 10$?

8. **а)** В своем трактате Евклид описал алгоритм поиска НОД, в котором вместо операций целочисленного деления используются операции вычитания. Запишите этот вариант алгоритма Евклида на псевдокоде.

   б) *Игра Евклида* (см. [20]). Напишите на доске два неравных положительных числа. В игре участвуют два игрока. Игроки должны по очереди писать на доске положительное число, равное разности двух чисел, уже написанных на доске. Причем это число должно быть новым, т.е. оно не должно уже находиться на доске. Проигравшим считается тот игрок, кто не сможет написать новое число. Какой игрок, по вашему мнению, имеет преимущество в этой игре: первый или второй?

9. Придумайте алгоритм вычисления функции $\lfloor \sqrt{n} \rfloor$.

10. В ***усовершенствованном алгоритме Евклида*** определяется не только НОД двух положительных чисел m и n (обозначим его через d), но и два числа x и y (не обязательно положительные), такие, что выполняется равенство $mx + ny = d$.

    а) Найдите описание усовершенствованного алгоритма Евклида (например, см. [[65], с. 28]) и реализуйте его в виде программы на своем любимом языке программирования.

    б) Видоизмените программу так, чтобы она могла находить целочисленное решение *диофантова уравнения* $ax + by = c$ для любых целых коэффициентов a, b и c. (Учтите, что для некоторых комбинаций коэффициентов уравнение может не иметь решения.)

1.2 Основы решения алгоритмической задачи

Начнем этот раздел с повторения важного вывода, сделанного во введении к этой главе:

Алгоритмы можно считать процедурным решением задач.

Причем эти решения являются не столько ответами, сколько точно определенными инструкциями для получения ответов. Именно этот акцент на точности определения конструктивных процедур отличает информатику от других дисциплин, в частности от теоретической математики, в которой обычно ограничиваются доказательством существования решения и (иногда) исследованиями характеристик решения.

А теперь перечислим и кратко опишем последовательность этапов проектирования и анализа алгоритмов (рис. 1.2).

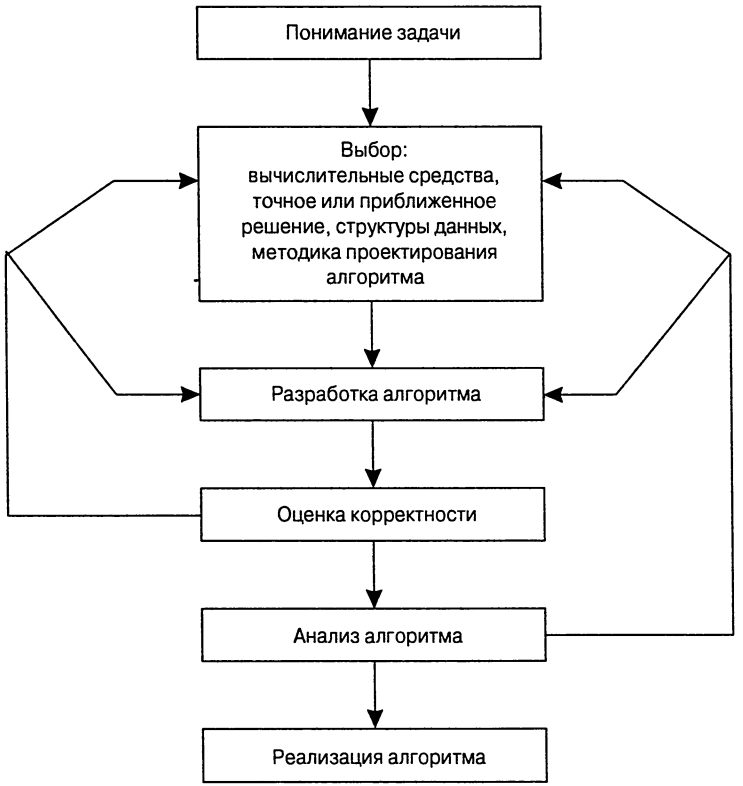


Рис. 1.2. Процесс проектирования и анализа алгоритмов

Понимание задачи

С практической точки зрения, прежде чем заняться проектированием алгоритма, необходимо полностью понять поставленную перед вами задачу. Прочтите внимательно условие задачи и задайте вопросы, если вы чего-то не поняли. Просчитайте вручную несколько небольших примеров, продумайте частные случаи и при необходимости снова задайте вопросы.

Существует несколько типов задач, которые при создании компьютерных приложений встречаются чаще всего. Их обзор будет сделан в следующем разделе. Если перед вами поставлена одна из этих задач, для ее решения можно воспользоваться известным алгоритмом. Конечно, в процессе решения необходимо понять, как работает алгоритм, и оценить все его достоинства и недостатки, особенно если существует возможность выбора среди нескольких алгоритмов. Однако чаще всего вы не сможете найти подходящий готовый алгоритм и должны будете создать собственный. Описанная в этом разделе последовательность этапов поможет вам в этом захватывающем, но не всегда легком деле.

Входные данные, обрабатываемые алгоритмом, определяют ***экземпляр задачи*** (problem's instance), решаемой с помощью выбранного алгоритма. При этом очень важно точно указать границы примеров, которые должны учитываться в алгоритме. (Вспомните, насколько отличаются границы примеров для трех алгоритмов поиска НОД, рассмотренных в предыдущем разделе.) Если этого не сделать, то алгоритм будет прекрасно работать для большинства значений входных параметров, однако при подстановке отдельных "граничных" значений вы получите некорректные результаты. Позвольте напомнить, что корректным считается такой алгоритм, который выдает абсолютно правильный результат для *всех* (а не только для большинства!) заранее оговоренных значений входных данных.

Не стоит недооценивать первый этап процесса решения алгоритмической задачи, поскольку в противном случае, как правило, приходится многое переделывать.

Определение возможностей вычислительного устройства

Полностью уяснив суть поставленной задачи, необходимо оценить возможности вычислительного устройства, для которого создается алгоритм. Подавляющее большинство современных алгоритмов предназначено для создания на их основе программы, работающей на компьютере, сильно напоминающем по структуре машину фон Неймана[3]. Суть этой архитектуры выражена в ее названии — ***машина с произвольным доступом*** (random-access machine, или ***RAM***). Предполагалось,

[3]Речь идет о структуре вычислительного устройства, описанного выдающимся американским математиком венгерского происхождения Джоном фон Нейманом (John von Neumann) (1903–1957) в сотрудничестве с А. Барксом (A. Burks) и Г. Голдстином (H. Goldstine) в 1946 году.

что команды должны последовательно выбираться из памяти и выполняться специальным устройством, называемым центральным процессором, причем в каждый момент времени в машине Неймана могла выполняться только одна команда. Поэтому алгоритмы, разработанные для выполнения на такой машине, назвали ***последовательными*** (sequential algorithms).

Появление машины Неймана не могло остановить прогресс в области вычислительной техники. Вскоре были придуманы компьютеры, которые могли одновременно (т.е. параллельно) выполнять несколько команд. Поэтому алгоритмы, разработанные для таких машин, назвали ***параллельными*** (parallel algorithms). Тем не менее изучение классических методов проектирования и анализа алгоритмов для машины Неймана еще долго будет оставаться краеугольным камнем алгоритмики.

Задумывались ли вы когда-нибудь о том, насколько быстро ваш компьютер выполняет команды и какой у него объем оперативной памяти? Наверняка вы ответите "нет", если до этого проектировали алгоритмы лишь с теоретической точки зрения. Как будет показано в разделе 2.1, большинство кибернетиков предпочитают изучать алгоритмы, не привязываясь к параметрам конкретной компьютерной системы. Ответ специалиста-практика наверняка будет зависеть от того, какую задачу перед ним поставили. Даже "медленные" по современным меркам компьютеры на деле оказываются невероятно быстродействующими. Следовательно, проблема "медленного компьютера" не должна возникать для большинства решаемых вами задач. Однако существует целый класс важных задач, которые по своей природе являются очень сложными, должны обрабатывать огромные массивы данных или работать в жестких временны́х рамках. В подобных ситуациях непременно следует учитывать быстродействие компьютерной системы, на которой будет работать реализация алгоритма, и доступный объем оперативной памяти.

Выбор между точным или приближенным методом решения задачи

Следующий принципиальный вопрос — выбор точного или приближенного метода решения задачи. В первом случае алгоритм называется ***точным*** (exact algorithm), а во втором — ***приближенным*** (approximation algorithm). Почему для решения задачи иногда выбираются приближенные алгоритмы? Во-первых, существуют задачи, которые нельзя решить точно. В качестве примера можно привести извлечение квадратного корня, решение нелинейных уравнений или вычисление определенных интегралов. Во-вторых, существующие алгоритмы для точного решения задачи могут быть недопустимо медленными, если ее сложность достаточно высока. Наиболее известной из таких задач является ***задача коммивояжера*** (traveling salesman problem), которая заключается в поиске кратчайшего маршрута

между n городами. В главах 3, 10 и 11 будут приведены и другие примеры подобных задач. В-третьих, приближенный алгоритм может являться частью другого, более сложного алгоритма, с помощью которого задача решается точно.

Выбор подходящих структур данных

В некоторых алгоритмах не требуется, чтобы входные данные были представлены в каком-то специфическом формате. Однако так бывает не всегда, более того, для работы многих алгоритмов требуются совершенно определенные структуры данных. Кроме того, некоторые из методов проектирования алгоритмов, о которых пойдет речь в главах 6 и 7, очень тесно связаны со структуризацией и реструктуризацией данных, определяющих экземпляры задачи. Много лет назад в уважаемой всеми книге провозглашалась чрезвычайная важность алгоритмов и структур данных и их влияние на процесс программирования. Причем об этой важности говорило само за себя название этой книги: *Algorithms + Data Structures = Programs* [123]. В современном мире, где властвует объектно-ориентированное программирование, структуры данных остаются чрезвычайно важным элементом процесса проектирования и анализа алгоритмов. Обзор основных структур данных будет сделан в разделе 1.4.

Методы проектирования алгоритмов

Теперь, когда все составляющие решения алгоритмической проблемы находятся на своих местах, остается выяснить, как нужно проектировать алгоритмы решения поставленных перед вами задач. Это основной вопрос данной книги. Для ответа на него вам предлагается изучить несколько общих методов проектирования алгоритмов. Что же такое метод проектирования алгоритма?

> **Метод проектирования алгоритма** (algorithm design technique) (или "стратегия", или "принцип") — это универсальный подход, применяемый для алгоритмического решения широкого круга задач, относящихся к различным областям вычислительной техники.

Взглянув на оглавление этой книги, вы поймете, что основная часть ее глав посвящена именно отдельным методам проектирования алгоритмов. В них излагается несколько проверенных временем ключевых идей, используемых при разработке алгоритмов. Изучение этих методов в высшей степени важно по следующим причинам.

Во-первых, они обеспечивают набор универсальных принципов, руководствуясь которыми можно разработать алгоритмы решения новых задач, т.е. таких задач, для решения которых еще не существует достаточно хороших алгоритмов. Поэтому, перефразируя известную поговорку, можно сказать, что изучение подобных методов сродни подаренной удочке, а не предложенной рыбе. Конечно, не

все из этих универсальных методов подойдут для решения практических задач, с которыми вам предстоит столкнуться. Однако они представляют собой мощный набор средств, которые пригодятся вам в учебе и работе.

Во-вторых, алгоритмы являются краеугольным камнем информатики. Классификация основополагающих понятий важна для любой науки, и информатика не является исключением. Изучение этих методов позволяет классифицировать алгоритмы согласно лежащему в их основе принципу проектирования. Поэтому они как нельзя лучше подходят и для классификации, и для изучения алгоритмов.

Методы представления алгоритмов

После того как алгоритм спроектирован, нужно представить его в каком-либо виде. В разделе 1.1 в качестве примера мы описали алгоритм Евклида — как словами (т.е. в свободной форме в виде последовательности выполняемых действий), так и в виде псевдокода. В наши дни для представления алгоритмов чаще всего используются эти две формы.

Безусловно, использование естественного языка для описания алгоритма имеет очевидное преимущество. Тем не менее присущая любому такому языку неопределенность иногда затрудняет лаконичное и понятное описание алгоритмов. Как бы то ни было, умение словесно описать алгоритм в процессе его изучения никогда не будет лишним.

Псевдокод (pseudocode) представляет собой смесь одного из естественных языков[4] и конструкций, характерных для языка программирования. Описание алгоритма на псевдокоде обычно является более точным по сравнению с естественным языком. Кроме того, в результате его использования часто получается более компактная запись алгоритма. Неожиданностью является то, что специалисты до сих пор так и не приняли какую-либо форму псевдокода в качестве стандарта. Поэтому в разных книгах можно встретить различные "диалекты" этого языка. К счастью, эти диалекты настолько близки, что любой человек, владеющий каким-либо современным языком программирования, сможет без особого труда в них разобраться.

Используемый в этой книге диалект псевдокода не должен представлять особых затруднений для читателя. Ради простоты изложения мы опустили операторы определения переменных, а для обозначения области действия таких операторов, как **for**, **if** и **while**, в тексте псевдокода использованы отступы. Как вы уже, наверное, заметили при чтении предыдущего раздела, для обозначения операции присваивания мы используем стрелку, ←, а комментариев — две косые черты подряд; //.

[4]Чаще всего английского, хотя встречаются случаи использования в псевдокоде русского языка. — *Прим. перев.*

На заре развития вычислительной техники основным способом представления алгоритмов были ***блок-схемы*** (flowchart), т.е. чертежи, состоящие из последовательности соединенных стрелками геометрических фигур, с помощью которых описывался каждый шаг выполнения алгоритма (примером может служить рис. 1.2). Однако подобный способ представления сочли неудобным, особенно в случае больших и сложных алгоритмов, поэтому в современной литературе он не используется.

Современные средства вычислительной техники еще не достигли такого уровня, чтобы описание алгоритма (будь-то его словесное описание или псевдокод) можно было непосредственно ввести в компьютер. Поэтому пока приходится вручную преобразовывать алгоритмы в компьютерные программы и записывать их на одном из доступных языков программирования. Таким образом, подобную программу можно считать еще одним средством представления алгоритма, хотя, строго говоря, программа является реализацией конкретного алгоритма.

Оценка корректности алгоритма

После представления алгоритма в какой-либо форме, необходимо оценить его ***корректность*** (correctness). Это означает, что вы должны доказать, что выбранный алгоритм за ограниченный промежуток времени выдает требуемый результат для любых корректных значений входных данных. Например, корректность алгоритма Евклида для вычисления НОД двух чисел, m и n, следует из правильности равенства $\gcd(m, n) = \gcd(n, m \bmod n)$, которое, в свою очередь, должно быть доказано (см. упражнение 1.1.5), а также из простого наблюдения, что второе число на каждой итерации будет все время уменьшаться, и по достижении им нуля, выполнение алгоритма прекращается.

Доказать корректность одних алгоритмов очень легко, других — невероятно сложно. Универсальным методом доказательства корректности алгоритма считается метод математической индукции. Дело в том, что алгоритмы по своей природе являются итеративными и описываются в виде последовательности пошаговых инструкций, которые как раз и используются при доказательстве методом индукции. Стоит отметить, что, хотя отслеживание быстродействия алгоритма с помощью нескольких наборов тщательно подобранных входных данных приводит к очень полезным результатам, подобную проверку нельзя считать убедительной при доказательстве корректности алгоритма. Однако доказательством некорректной работы алгоритма может служить лишь один набор входных данных, при обработке которых получается неправильный результат. Если некорректная работа алгоритма будет доказана, придется либо несколько видоизменить его, не выходя за рамки принятых структур данных, методов проектирования и т.п., либо решать проблему более кардинально, пересмотрев принятые ранее принципы и подходы (см. рис. 1.2).

Оценка корректности приближенных алгоритмов менее тривиальна, чем их точных аналогов. При этом нужно показать, что погрешность получаемых в результате работы алгоритма выходных данных не превышает заранее установленных пределов. Примеры подобных оценок приведены в главе 11.

Анализ алгоритма

Обычно создатели алгоритмов стараются, чтобы они удовлетворяли нескольким требованиям. После проверки корректности алгоритма одной из самых важных характеристик является оценка его эффективности. На практике существует два вида оценки эффективности алгоритма: временна́я и пространственная. ***Временная эффективность*** (time efficiency) является индикатором скорости работы алгоритма. Что касается ***пространственной эффективности*** (space efficiency), то эта оценка показывает количество дополнительной оперативной памяти, необходимой для работы алгоритма. Общая идея и конкретные методы анализа эффективности алгоритмов будут рассмотрены в главе 2.

Еще одной важной характеристикой алгоритма является его ***простота*** (simplicity). В отличие от эффективности, которую можно точно определить и оценить с помощью строгих математических выражений, простота алгоритма чем-то напоминает человеческую красоту, понятие которой настолько субъективно, что выработать объективные критерии ее оценки весьма непросто. Например, большинство людей считают, что алгоритм Евклида поиска НОД двух чисел проще, чем тот, которому нас учили в школе, но это вовсе не означает, что он понятнее. В то же время алгоритм Евклида проще алгоритма последовательного перебора целых чисел. Тем не менее простота является важной характеристикой алгоритма, и к ней нужно стремиться. Вы спросите, почему? Дело в том, что простые алгоритмы легче понять и запрограммировать. Следовательно, в полученной программе будет содержаться меньше ошибок. Кроме того, в простоте существует неоспоримая эстетическая привлекательность. Ну и наконец, простые алгоритмы зачастую более эффективны, чем их сложные аналоги. К сожалению, последнее утверждение не всегда выполняется. В подобных случаях необходимо руководствоваться здравым смыслом и принимать компромиссные решения.

Следующей важной характеристикой алгоритма является его ***общность***, или ***универсальность*** (generality). По сути, здесь можно выделить два момента: общность задачи, для решения которой разработан алгоритм, и диапазон допустимых значений его входных данных. По поводу первого замечания следует сказать, что иногда легче разработать алгоритм для решения общей задачи, чем заниматься поиском решения ее частного случая. В качестве примера рассмотрим такую задачу: определить, являются ли два целых числа взаимно простыми. Напомним, что два числа считаются взаимно простыми, если у них нет общих делителей, кроме 1. В данном случае легче разработать алгоритм для решения общей зада-

чи вычисления НОД двух целых чисел. Затем, чтобы решить исходную задачу, достаточно подставить заданные числа в функцию gcd и проверить, равно ли ее значение 1. Однако бывают случаи, когда разработка более общего алгоритма нежелательна, затруднена или даже невозможна. Например, излишне выполнять сортировку всего списка, содержащего n чисел, чтобы определить их медиану, поскольку достаточно просто найти значение m-го наименьшего элемента, где $m = \lceil n/2 \rceil$. Еще один пример: известно, что стандартную формулу для поиска корней квадратного уравнения нельзя обобщить для поиска корней полиномов более высоких степеней.

Что касается диапазона допустимых значений входных данных алгоритма, то здесь нужно отметить следующее. При разработке алгоритма нужно учитывать, что диапазон изменения значений входных параметров может колебаться в очень широких пределах и должен естественным образом соответствовать решаемой задаче. Например, неестественно в алгоритме поиска НОД исключать из рассмотрения значения входных параметров, равные 1. С другой стороны, хотя стандартную формулу корней квадратного уравнения можно использовать и в случае комплексных коэффициентов, при принятом уровне обобщения этот случай исключают из рассмотрения, поскольку он должен оговариваться отдельно.

Если вас не устраивает эффективность, простота или общность алгоритма, придется снова взять в руки карандаш и перепроектировать алгоритм. И даже если анализ алгоритма привел к положительным результатам, имеет смысл поискать другое алгоритмическое решение задачи. Достаточно вспомнить три алгоритма определения НОД, рассмотренных в предыдущем разделе. Вообще говоря, не стоит надеяться, что вы с первой попытки найдете оптимальное решение задачи. Самое меньшее, что можно сделать, попытаться оптимизировать созданный алгоритм. Например, мы внесли несколько улучшений в реализацию алгоритма решета Эратосфена, по сравнению с той, что была описана в разделе 1.1. (Можете ли вы назвать их не задумываясь?) Всегда помните высказывание Антуана де Сент–Экзюпери (Antoine de Saint–Exupery), известного французского писателя, летчика и авиаконструктора: "Конструктор достигает совершенства не тогда, когда ему больше нечего добавить к своему детищу, а тогда, когда больше ничего удалить".[5]

Кодирование алгоритма

Большинство алгоритмов рассчитано на то, что в конечном итоге они будут превращены в компьютерную программу. В процессе написания программы на

[5] Я обнаружил это высказывание по поводу простоты конструкции в сборнике рассказов Джона Бентли (Jon Bentley) [15]. Этот сборник посвящен особенностям проектирования и реализации алгоритмов и имеет совершенно логичное название *Programming Pearls* [15]. Настоятельно рекомендую почитать рассказы Бентли и Сент–Экзюпери. Не пожалеете!

основе алгоритма может возникнуть множество подводных камней. Главная опасность заключается в том, что при переводе алгоритма на машинный язык могут быть внесены ошибки, либо окажется, что сама программа написана крайне неэффективно. Поэтому некоторые авторитетные специалисты считают, что до тех пор, пока корректность компьютерной программы не будет доказана с помощью строгих математических выкладок, ее нельзя считать правильной. Они даже разработали специальные методы для выполнения подобных оценок (см. [47]). Однако пока эти методы формальной верификации удается применить только к очень маленьким программам. На практике корректность компьютерных программ все еще проверяется с помощью тестирования. Процесс тестирования является скорее искусством, чем наукой, но это отнюдь не означает, что здесь нечему поучиться. Тестированию и отладке посвящено довольно много хороших книг, однако их основную мысль можно сформулировать так: в процессе реализации алгоритма программа должна быть тщательно протестирована и отлажена.

Следует также заметить, что в этой книге мы всегда предполагаем, что диапазоны значений входных данных алгоритмов не выходят за заранее оговоренные рамки, поэтому их не нужно проверять. Однако при реализации алгоритмов в виде реально работающих прикладных программ такая проверка просто необходима.

Само собой разумеется, что корректная реализация алгоритма в виде программы является необходимым, но недостаточным условием, поскольку мощь алгоритма можно свести на нет неэффективной реализацией. Современные компиляторы до некоторой степени позволяют застраховаться от этого, особенно при использовании режима оптимизации кода. Тем не менее следует знать о таких стандартных вещах, как вынесение операторов вычисления инвариантного выражения (т.е. такого выражения, которое не изменяется в цикле) за пределы цикла, выделение общих подвыражений, замена медленных операторов их более быстрыми аналогами и т.п. (Хорошее описание методов оптимизации кода программы и других вопросов, связанных с кодированием алгоритмов, можно найти в [15] и [61].) Как правило, применение подобных методов оптимизации может ускорить выполнение программы только на небольшой постоянный множитель, тогда как использование более эффективного алгоритма иногда ускоряет выполнение программы на несколько порядков. Когда алгоритм окончательно выбран, повышение производительности реализующей его программы на 10–50% считается хорошим результатом.

После создания работающей версии программы можно выполнить дополнительный эмпирический анализ лежащего в ее основе алгоритма. Для этого нужно зафиксировать время выполнения программы при разных значениях входных данных, а затем проанализировать полученный результат. Достоинства и недостатки данного метода анализа алгоритмов будут описаны в разделе 2.6.

И в заключение позвольте еще раз подчеркнуть основную идею процесса проектирования и анализа алгоритмов, показанного на рис. 1.2.

> Хороший алгоритм получается, как правило, в результате кропотливой циклической работы, связанной с возможными переделками.

Даже если вам крупно повезет и вы придумаете алгоритм, который на первый взгляд кажется идеальным, все равно попытайтесь проанализировать, нельзя ли его еще в чем-то улучшить. Как ни странно, эта мысль не такая уж и плохая, поскольку позволяет получить истинное удовольствие от конечного результата. (Да, да! Кстати я собирался назвать эту книгу как *The Joy of Algorithms*.[6]) С другой стороны, нужно уметь вовремя останавливаться. Что касается реальной жизни, то здесь обычно критерий остановки один — срок сдачи проекта или долготерпение вашего начальника, в зависимости от того, что быстрее закончится. В общем вывод такой: достижение идеала — слишком дорогое удовольствие, которое, к тому же, не всегда нужно. Проектирование алгоритма является достаточно сложной инженерной задачей, связанной с принятием компромиссных решений в условиях ограниченного доступа к ресурсам, одним из которых является время работы проектировщика.

С академической точки зрения затронутый в этом разделе вопрос связан с проведением интересного, но, как правило, очень сложного исследования ***оптимальности*** (optimality) алгоритма. Как ни странно, этот вопрос не относится к эффективности самого алгоритма, а связан со сложностью решаемой с его помощью задачи. То есть необходимо выяснить, какое минимальное количество усилий нужно затратить, чтобы решить стоящую перед вами задачу с помощью *произвольного* алгоритма? Для некоторых задач ответ на этот вопрос найден. Например, в любом алгоритме сортировки элементов массива методом сравнения необходимо выполнить порядка $n \log_2 n$ операций сравнения, где n — размерность массива (см. раздел 10.2). Однако для многих кажущихся простыми задач, типа перемножения матриц, ученым до сих пор так и не удалось найти окончательного ответа.

Еще один важный вопрос, возникающий при решении алгоритмической проблемы, заключается в том, можно ли решить задачу вообще с помощью какого-либо алгоритма? В этой книге мы не будем обсуждать задачи, не имеющие решения, наподобие поиска вещественных корней квадратного уравнения с отрицательным дискриминантом. В подобных случаях алгоритм должен возвращать специальное значение, являющееся признаком того, что задача не имеет решения. Мы не будем также рассматривать неоднозначно определенные задачи. Речь идет о таких задачах, решение которых нельзя найти с помощью любого алгоритма, хотя у них может быть простой ответ — да или нет. Пример такой задачи будет приведен в разделе 10.3. К счастью, подавляющее большинство подобных задач *может* быть решено с помощью некоторого алгоритма.

И в заключение этого раздела хотелось бы развеять ваши сомнения по поводу того, что проектирование алгоритмов — довольно скучное занятие. Отчасти они

[6] Это название можно перевести на русский язык как *Удовольствие от алгоритмов. — Прим. ред.*

могли появиться при взгляде на рис. 1.2, где все четко разложено по полочкам и на первый взгляд нет никакой свободы для творчества. Это абсолютно не соответствует действительности! Изобретение (или поиск?) алгоритмов — творческий и необычайно захватывающий процесс, и я надеюсь, что книга убедит вас в этом.

Упражнения 1.2

1. *Древняя народная головоломка.* На берегу реки находятся крестьянин, волк, коза и кочан капусты. Крестьянин должен в своей лодке перевезти их на другой берег. Однако в лодке есть только два места — для крестьянина и еще одного объекта (т.е. либо волка, либо козы, либо капусты). В отсутствие крестьянина волк может съесть козу, а коза — капусту. Помогите крестьянину решить эту задачу или докажите, что она не имеет решения. (*Примечание.* Для определенности будем считать, что крестьянин является вегетарианцем, но терпеть не может капусту, поэтому в лодке он не может съесть ни козу, ни капусту. Кроме того, крестьянин должен учитывать, что он столкнулся с редкой породой волка, которая занесена в Красную книгу.)

2. *Современная народная головоломка.* Предположим, что четыре человека движутся по дороге в одном направлении и хотят перейти через мост. Ваша задача помочь им переправиться на другой берег за 17 минут. На дворе ночь, и у них только один фонарик. По мосту одновременно могут следовать не более двух человек (т.е. либо один, либо два), причем у одного из них обязательно должен быть фонарик. Фонарик нельзя перебросить с одного берега реки на другой, его можно только перенести по мосту обратно. Каждый человек затрачивает разное время на прохождение моста: первый — 1 минуту, второй — 2 минуты, третий — 5 минут и четвертый — 10 минут. Если по мосту передвигается пара людей, то они идут со скоростью более медлительного из них. Например, если по мосту передвигается первый и четвертый человек, то они достигнут противоположного берега через 10 минут. Если четвертый человек будет возвращать фонарь на другой берег, то с момента начала задачи пройдет 20 минут, и вы не решите задачу. (*Примечание.* В Internet бродят слухи, что одна из известных компаний по производству программного обеспечения, расположенная вблизи Сиэтла, предлагает решить эту задачу претендентам на собеседовании.)

3. Какую из перечисленных ниже формул можно использовать в качестве алгоритма для вычисления площади треугольника, длина сторон которого выражена положительными числами a, b и c?
   а) $S = \sqrt{p(p-a)(p-b)(p-c)}$, где $p = (a+b+c)/2$;
   б) $S = \frac{1}{2}bc\sin A$, где A — угол между сторонами b и c;
   в) $S = \frac{1}{2}ah_a$, где h_a — высота треугольника, опущенная на сторону a.
4. Запишите на псевдокоде алгоритм поиска вещественных корней квадратного уравнения $ax^2 + bx + c = 0$, где a, b и c — произвольные вещественные коэффициенты. (Подразумевается, что для поиска квадратного корня вы можете воспользоваться функцией $sqrt(x)$.)
5. Опишите стандартный алгоритм преобразования положительного десятичного числа в двоичное:
   а) словами;
   б) на псевдокоде.
6. Опишите алгоритм работы с банкоматом при получении денег с карточки (если, конечно, она у вас есть). (Опишите алгоритм либо словами, либо на псевдокоде — как вам больше нравится.)
7. **а)** Может ли быть точно решена задача вычисления числа π?
   б) Сколько экземпляров имеет данная задача?
   в) Поищите алгоритм решения этой задачи в World Wide Web.
8. Приведите пример задачи, отличной от поиска НОД, для решения которой существует несколько алгоритмов. Какой из этих алгоритмов проще? Какой эффективнее?
9. Проанализируйте приведенный ниже алгоритм поиска минимальной разницы между двумя элементами массива чисел.

```
Алгоритм MinDistance(A[0..n − 1])
  // Входные данные:  массив чисел A[0..n − 1]
  // Выходные данные: минимальная разница между двумя
  //                  элементами массива A
  dmin ← ∞
  for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
      if i ≠ j and |A[i] − A[j]| < dmin
        dmin ← |A[i] − A[j]|
  return dmin
```

Можете ли вы усовершенствовать алгоритм решения этой задачи, и если да, то какое количество изменений вы можете в него внести? (При

необходимости можно выбрать другой алгоритм. Если вы не можете улучшить предложенный алгоритм, то можете ли вы усовершенствовать его реализацию?)

10. Одна из самых известных книг, посвященная алгоритмическому решению задач, озаглавленная *How to Solve It*, написана американским математиком венгерского происхождения Джорждем Пойа (George Polya) (1887–1985) [89]. Пойа обобщил выдвинутые им идеи в виде резюме, состоящего из четырех пунктов. Найдите это резюме в Web (или, что еще лучше, прочтите саму книгу) и сравните его с предложенным планом решения алгоритмической задачи, который мы рассматривали в разделе 1.2. Что у них общего? В чем различия?

1.3 Важные типы задач

Среди огромного количества задач, встречающихся в вычислительной технике, можно выделить несколько типов, которым ученые всегда уделяли особое внимание. Подобный интерес вызван либо практическим значением задачи, либо какими-то ценными ее свойствами, представляющими особый интерес для исследования. К счастью, в большинстве случаев эти причины взаимосвязаны, что только усиливает интерес ученых.

В этом разделе кратко рассматриваются наиболее важные типы задач:

- сортировка;
- поиск;
- обработка строк;
- задачи из теории графов;
- комбинаторные задачи;
- геометрические задачи;
- численные задачи.

Эти задачи будут использоваться в последующих главах книги для иллюстрации различных методов проектирования и анализа алгоритмов.

Сортировка

Задача сортировки (sorting problem) заключается в упорядочении заданного списка каких-либо элементов в возрастающем порядке[7]. Само собой разумеется, что для определенности задачи структура этих элементов списка должна позволять их упорядочить. (В подобных случаях математики говорят, что между

[7]Очевидно, что для сортировки в порядке убывания достаточно заменить вид сравнения элементов последовательности на обратный. — *Прим. ред.*

элементами должны существовать отношения, допускающие полное упорядочение.) На практике обычно требуется отсортировать по возрастанию список чисел, расположить символы и строки в алфавитном порядке или, что наиболее важно, упорядочить набор записей, содержащих различную информацию, наподобие той, что хранится в папках со сведениями об учащихся школы, в читательских формулярах библиотеки, в отделе кадров о сотрудниках организации. В случае набора записей необходимо выбрать фрагмент записи, содержащий данные, по которым будет осуществляться сортировка. Например, набор записей о студентах можно отсортировать по фамилии, идентификационному номеру или по среднему баллу. Специально отобранный фрагмент данных называется ***ключом*** (key). Специалисты по информатике часто говорят о сортировке списка ключей, даже если элементы этого списка являются не записями, а, скажем, целыми числами.

Зачем может понадобиться отсортированный список? Ну, скажем затем, чтобы облегчить поиск ответов на ряд вопросов, связанных со списком. Наибольшую важность при этом имеет быстрота поиска информации. Вот почему словари, телефонные справочники, списки учащихся и т.п. всегда упорядочиваются по алфавиту. В разделе 6.1 будут приведены другие примеры, в которых используется предварительно отсортированный список. Аналогично сортировка используется также как вспомогательный этап в некоторых важных алгоритмах, относящихся к другим предметным областям, например к геометрии.

К настоящему моменту специалистами по вычислительной технике разработаны десятки алгоритмов сортировки. По сути, выдумывание нового алгоритма сортировки можно сравнить с изобретением пресловутого велосипеда. Тем не менее мы с гордостью хотим заявить, что поиск лучших вариантов велосипеда (т.е. алгоритмов сортировки) продолжается. Подобная настойчивость может быть оправданна, если принять во внимание следующие факты. С одной стороны, существует несколько хороших алгоритмов сортировки, в которых для сортировки произвольного массива из n элементов используется $n \log_2 n$ операций сравнения. С другой стороны, нет алгоритма, который бы выполнял сортировку методом сравнения всего значения ключа (а не, скажем, небольшой части ключа) за существенно меньшее количество операций, чем было указано выше.

Существует причина, сдерживающая разработку новых алгоритмов сортировки. Несмотря на то что часть алгоритмов оказывается лучше остальных, пока еще не придуман универсальный алгоритм сортировки, который бы наилучшим образом подходил для всех случаев. Одни из существующих алгоритмов являются простыми, но сравнительно медленными, другие работают быстрее, но более сложны. Некоторые из алгоритмов хорошо работают с неупорядоченными данными, тогда как для других нужно, чтобы списки были частично отсортированы. Часть алгоритмов пригодна только для сортировки списков данных, расположенных в оперативной памяти (т.е. в памяти с быстрым доступом), тогда как другие

можно применить для сортировки больших массивов данных, расположенных на внешних носителях данных (магнитном диске, ленте и т.д.).

Два свойства алгоритмов сортировки заслуживают особого внимания. Алгоритм сортировки называется ***устойчивым*** (stable), если в нем сохраняется относительный порядок любых двух равных элементов, находящихся во входном списке. Другими словами, если во входном списке есть два одинаковых элемента, номера которых равны i и j, причем $i < j$, то в отсортированном списке, где их номера будут, соответственно, равны i' и j', будет выполняться отношение $i' < j'$. Это свойство может пригодиться в случае, если, например, требуется отсортировать упорядоченный в алфавитном порядке список учащихся согласно их успеваемости (среднему баллу). В случае применения устойчивого алгоритма будет получен список, в котором учащиеся с одинаковым средним баллом будут упорядочены по алфавиту. Вообще говоря, алгоритмы сортировки методом обмена значениями ключей, расположенными на значительном расстоянии друг от друга, не являются устойчивыми, однако обычно они работают быстрее. Ниже в этой книге мы покажем, как это общее утверждение применяется к основным алгоритмам сортировки.

Второе важное свойство алгоритмов сортировки связано с количеством дополнительной оперативной памяти, необходимой для работы алгоритма. Алгоритм называют ***обменным*** (in place), если для его работы не требуется дополнительная оперативная память, кроме случаев возможного использования нескольких дополнительных ячеек памяти. Многие важные алгоритмы сортировки относятся к классу обменных, а другие, не менее важные — нет.

Поиск

Задача ***поиска*** связана с нахождением заданного значения, называемого ***ключом поиска*** (search key), среди заданного множества (или мультимножества[8]). Существует огромное количество алгоритмов поиска, так что есть из чего выбирать. Их сложность варьируется от самых простых алгоритмов поиска методом последовательного сравнения, до чрезвычайно эффективных, но ограниченных алгоритмов бинарного поиска, а также алгоритмов, основанных на представлении базового множества в иной, более подходящей для выполнения поиска форме. Последние из упомянутых здесь алгоритмов имеют особое практическое значение, поскольку применяются в реально действующих приложениях, выполняющих выборку и хранение массивов информации в огромных базах данных.

Для решения задачи поиска также не существует единого алгоритма, который бы наилучшим образом подходил для всех случаев. Некоторые из алгоритмов выполняются быстрее остальных, но для их работы требуется дополнительная

[8] ***Мультимножество*** — это множество, в котором несколько принадлежащих ему элементов могут иметь одинаковые значения.

оперативная память. Другие выполняются очень быстро, но их можно применять только для предварительно отсортированных массивов, и т.п. В отличие от алгоритмов сортировки в алгоритмах поиска нет проблемы устойчивости, но при их использовании могут возникать другие сложности. В частности, в тех приложениях, где обрабатываемые данные могут часто изменяться, причем количество изменений сравнимо с количеством операций поиска, поиск следует рассматривать в неразрывной связи с двумя другими операциями — добавления элемента в набор данных и удаления из него. В подобных ситуациях необходимо видоизменить структуры данных и алгоритмы так, чтобы достигалось равновесие между требованиями, выдвигаемыми к каждой операции. Кроме того, организация очень больших наборов данных с целью выполнения в них эффективного поиска (а также добавления и удаления элементов) представляет собой чрезвычайно сложную задачу, решение которой особенно важно с точки зрения практического применения.

Обработка строк

В связи с быстрым увеличением в последнее время количества приложений, связанных с обработкой нечисловых данных, интерес ученых и специалистов-практиков все больше привлекают алгоритмы обработки строк. ***Строкой*** (string) называется последовательность символов, взятых из заранее определенного алфавита. Практический интерес представляют, например, текстовые строки, состоящие из букв, цифр и специальных символов; битовые строки, состоящие из нулей и единиц; последовательности генов, которые могут быть смоделированы с помощью строк символов, взятых из четырехсимвольного алфавита {A, C, G, T}. Тем не менее стоит отметить, что алгоритмы обработки строк стали важны для вычислительной техники очень давно — с тех пор, как появились первые языки программирования и соответствующие им программы — компиляторы.

Существует одна специфическая задача, которая привлекла особое внимание специалистов по информатике. Речь идет о поиске заданного слова в строке текста. Ее назвали ***поиском подстрок*** (string matching). Для выполнения такого специфического поиска разработано несколько алгоритмов. В главе 3 мы опишем один очень простой алгоритм, а в главе 7 обсудим два алгоритма, созданных на основе замечательной идеи Р. Бойера (R. Boyer) и Дж. Мура (J. Moore).

Задачи из теории графов

Одной из самых старых и, пожалуй, наиболее интересных областей алгоритмики является обработка графов. Нестрого ***граф*** можно определить как набор точек, называемых вершинами, часть из которых соединена отрезками, называемыми ребрами. (Более строгое определение графа будет дано в следующем разделе.)

Графы являются довольно интересным объектом для изучения как с теоретической, так и с практической точек зрения. С помощью графов можно смоделировать довольно большое количество процессов, происходящих в реальной жизни, например функционирование транспортных и коммуникационных сетей, календарное планирование проекта и т.д. Одна из последних интересных задач — оценка "диаметра" Web, заключающаяся в определении максимального количества ссылок, которые нужно пройти от одной Web-страницы до другой по оптимальному маршруту[9].

К числу основных алгоритмов теории графов относят следующие:

- алгоритмы обхода графа (этот класс алгоритмов позволяет ответить на такие вопросы, как каким образом можно объехать все узлы железнодорожной сети);
- алгоритмы определения кратчайшего пути (этот класс алгоритмов позволяет ответить на вопросы наподобие следующего: каков кратчайший путь между двумя городами);
- алгоритмы топологической сортировки для ориентированных графов ребрами (этот класс алгоритмов позволяет выяснить, не является ли множество читаемых студентам курсов внутренне противоречивым).

К счастью, все перечисленные алгоритмы можно рассматривать как пример общих методов проектирования. Поэтому вы сможете найти их описание в соответствующих главах книги.

Некоторые из задач теории графов с вычислительной точки зрения очень трудны. Это означает, что только небольшое количество экземпляров подобных задач можно решить за приемлемое время с помощью самого быстродействующего компьютера, который только можно себе представить. К наиболее известным задачам теории графов этого типа вероятнее всего относятся задачи коммивояжера и раскраски графа. Напомним, что задача коммивояжера заключается в нахождении кратчайшего пути между n городами, каждый из которых он должен посетить только один раз. Задача ***раскраски графа*** (graph-coloring problem) заключается в том, что нужно с помощью минимального количества красок раскрасить вершины графа так, чтобы не было двух смежных вершин одинакового цвета. Эта задача появляется в процессе решения важных практических задач, таких как, например, календарное планирование. Если представить работы в виде вершин графа и соединить их между собой ребрами тогда и только тогда, когда соответствующие им работы не могут выполняться в одно и то же время, решение задачи раскраски такого графа даст оптимальный график выполнения работ.

[9]Согласно оценке группы исследователей из университета Нотр–Дама (University of Notre Dame), это количество ссылок составляет всего 19 [6].

Комбинаторные задачи

С точки зрения обобщения задачи коммивояжера и раскраски графа являются примерами ***комбинаторных задач***. Суть этих задач в конечном итоге сводится к нахождению такого комбинаторного объекта, как перестановка, сочетание или подмножество, который бы удовлетворял определенным ограничениям и обладал заданными свойствами (например, позволял максимизировать прибыль или минимизировать затраты.)

Вообще говоря, комбинаторные задачи относятся к классу самых сложных как с теоретической, так и с практической точек зрения. Их сложность обусловлена следующим. Во-первых, количество комбинаторных объектов обычно очень быстро растет при увеличении масштаба задачи и достигает невообразимых значений уже при весьма скромных ее масштабах. Во-вторых, пока не существует алгоритмов для поиска точного решения подобных задач за приемлемое время. Более того, большинство специалистов в области информатики считают, что таких алгоритмов попросту не существует. Это предположение не было ни доказано, ни опровергнуто. Оно до сих пор остается наиболее важным из нерешенных вопросов теоретической информатики. Более подробно этот вопрос мы обсудим в разделе 10.3.

Некоторые комбинаторные задачи можно решить с помощью эффективных алгоритмов, но это скорее счастливое исключение, чем правило.

Геометрические задачи

Геометрические алгоритмы связаны с такими геометрическими объектами, как точки, линии, многоугольники. Древние греки проявляли большой интерес к разработке процедур (естественно, они их еще не называли алгоритмами) решения различных геометрических задач, среди которых можно назвать построение простых геометрических форм (треугольников, окружностей и т.п.) с помощью негрaдуированной линейки и циркуля. С тех пор прошло более 2000 лет, и в век компьютеров интерес к геометрическим алгоритмам вспыхнул с новой силой. Для решения подобных задач линейки и циркули уже не нужны — только биты, байты и накопленный годами человеческий опыт. Естественно, сейчас человечество проявляет совершенно другой интерес к геометрическим алгоритмам. Они нужны для решения современных задач компьютерной графики, робототехники и томографии.

В этой книге мы рассмотрим алгоритмы решения только двух классических задач вычислительной геометрии: поиска пары ближайших точек и определения выпуклой оболочки. Название задачи поиска ***пары ближайших точек*** говорит само за себя: среди n точек на плоскости необходимо выбрать пару точек, расположенных наиболее близко друг к другу. При решении задачи построения ***выпуклой оболочки*** необходимо найти наименьший выпуклый многоугольник, который бы

охватывал все точки некоторого множества на плоскости. Чтобы больше узнать об этих и других геометрических задачах, обратитесь к специализированным монографиям (например, [90]) или к соответствующим главам учебников, построенных по принципу описания конкретных типов задач (например, [102]).

Численные задачи

Численные задачи относятся к еще одной довольно обширной области практического использования алгоритмов. Они имеют дело с математическими объектами, которые по своей сути являются непрерывными. Вот примеры типичных численных задач: решение уравнений и систем уравнений, вычисление определенных интегралов и значений функций и т.д. Подавляющее большинство таких задач может быть решено только приблизительно. Еще одна принципиальная трудность заключается в том, что при решении подобных задач обычно нужно выполнять операции с вещественными числами, которые в компьютере могут быть представлены только с определенной погрешностью. Более того, выполнение большого количества арифметических операций над представленными приближенно числами может привести к накоплению ошибок округления, что в свою очередь может кардинально повлиять на точность получаемых результатов.

За прошедшие годы было придумано большое количество довольно сложных алгоритмов решения численных задач, причем они продолжают играть ключевую роль при выполнении научных и инженерных расчетов. Однако в течение последних 25 лет интересы компьютерной индустрии сместились в сторону создания прикладных программ для деловой сферы. Для создания приложений этой категории потребовалось разработать базовые алгоритмы хранения и выборки данных, передачи их по сетям и отображения в удобном для пользователя виде. В результате таких революционных изменений численный анализ утратил былые позиции как в области промышленного, так и научного использования программ. Тем не менее для любого человека, изучающего компьютерную литературу, важно иметь хотя бы элементарные понятия в области численных алгоритмов. Несколько таких классических алгоритмов мы опишем в разделах 6.2, 10.4 и 11.4.

Упражнения 1.3

1. Проанализируйте приведенный ниже алгоритм сортировки массива методом подсчета. Вначале для каждого элемента массива подсчитывается количество элементов, меньших, чем он, и на основе этой информации текущий элемент помещается в соответствующее место отсортированного массива.

Алгоритм *ComparisonCountingSort* ($A[0..n-1]$)

```
// Сортировка массива методом подсчета сравнений
// Входные данные:  массив чисел A[0..n − 1], который нужно
//                  отсортировать
// Выходные данные: массив чисел S[0..n − 1], состоящий из
//                  элементов массива A, отсортированных
//                  в неубывающем порядке
for i ← 0 to n − 1 do
    Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else
            Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do
    S[Count[i]] ← A[i]
return S
```

а) Попробуйте с помощью этого алгоритма отсортировать числа: 60, 35, 81, 98,14, 47.

б) Является ли этот алгоритм устойчивым?

в) Относится ли он к обменным алгоритмам?

2. Назовите известные вам алгоритмы поиска. Кратко опишите словами каждый алгоритм. (Если вам еще не знаком ни один подобный алгоритм, воспользуйтесь удобной возможностью и разработайте его самостоятельно.)

3. Придумайте простой алгоритм поиска строк.

4. *Мосты Кенигсберга.* Считается, что решение этой головоломки дало толчок развитию теории графов. Задача была решена выдающимся российским математиком швейцарского происхождения Леонардом Эйлером (Leonard Euler) (1707–1783). В головоломке предлагается ответить, можно ли одну прогулку обойти все семь мостов Кенигсберга и вернуться в отправную точку? На рис. 1.3 изображен схематический план реки, посередине которой расположены два острова, к которым ведут семь мостов.

а) Сформулируйте эту задачу в терминах теории графов.

б) Имеет ли задача решение? Если вы уверены в том, что имеет, начертите схему обхода мостов. Если нет, объясните, почему, и оцени-

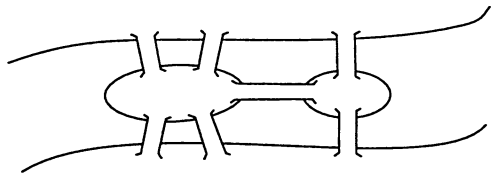


Рис. 1.3. Схематический план мостов Кенигсберга

те, какое минимальное количество новых мостов нужно построить, чтобы задача имела решение.

5. *Игра в икосаэдр*. Через сто лет после исследований Эйлера (см. задачу 1.3.4), известный ирландский математик сэр Вильям Гамильтон (Sir William Hamilton) (1805–1865) придумал еще одну головоломку — *Icosian Game*. На круглой деревянной доске вырезался граф, представленный на рис. 1.4.

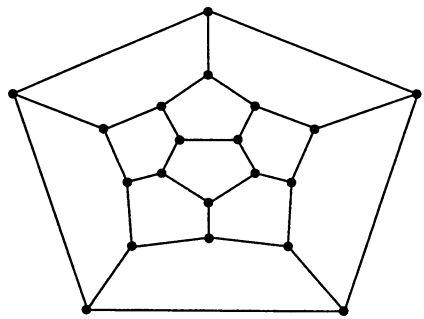


Рис. 1.4. Граф Гамильтона

Начертите ***гамильтонов цикл*** (Hamiltonian circuit) этого графа, т.е. путь однократного обхода всех вершин графа с возвратом в исходную точку.

6. Разработайте алгоритм поиска наилучшего маршрута для пассажира метрополитена, который бы позволял ему перемещаться от одной станции до другой по развитой системе подземных коммуникаций, наподобие вашингтонской или лондонской.
   - **а)** В условии этой задачи имеется небольшая неопределенность, что в общем-то типично для всех задач, находящих практическое применение. В частности, неясно, какой должен быть критерий оценки "наилучшего" маршрута? Можете ли вы его определить?
   - **б)** Можете ли вы смоделировать решение этой задачи с помощью графа?

7. **а)** Сформулируйте задачу коммивояжера в терминах комбинаторики.
   б) Сформулируйте задачу раскраски графа в терминах комбинаторики.
8. Проанализируйте карту, изображенную на рис. 1.5.

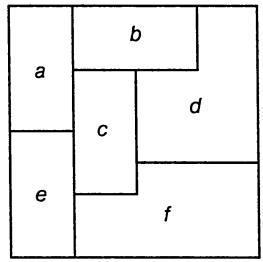


Рис. 1.5. Учебная контурная карта

   а) Поясните, как можно использовать решение задачи раскраски графа для окрашивания этой карты так, чтобы цвета двух соседних областей не совпадали.
   б) Воспользуйтесь решением пункта *а* упражнения и раскрасьте карту минимально возможным количеством цветов
9. Разработайте алгоритм решения следующей задачи. Предположим, на плоскости находится n точек, заданных их координатами (x, y). Определите, лежат ли все точки на одной и той же окружности?
10. Напишите программу, которая бы считывала из входного потока данных координаты (x, y) конечных точек двух отрезков P_1Q_1 и P_2Q_2 и определяла, пересекаются ли эти отрезки (т.е. программа должна найти координаты общей точки двух отрезков, если таковая существует).

1.4 Базовые структуры данных

Поскольку большинство рассматриваемых в этой книге алгоритмов выполняют обработку данных, то на процесс их проектирования и анализа существенно влияют конкретные способы организации данных. Понятие ***структуры данных*** (data structure) можно определить как особую систему организации взаимосвязанных элементов данных. Структура самих элементов данных зависит от задачи, в которой они используются. Они могут быть как очень простыми (например, представлять собой целые числа или строки символов), так и сложными структурами данных (например, для представления матриц часто используют одномерный массив, каждый элемент которого, в свою очередь, также является одномерным массивом). Тем не менее существует несколько чрезвычайно важных при проекти-

ровании алгоритмов для компьютеров структур данных. Поскольку вы наверняка знакомы с большинством из них (если не со всеми), ниже представлен лишь их краткий обзор.

Линейные структуры данных

Среди простых структур данных можно выделить две самых важных — одномерный массив и связанный список. ***Одномерный массив*** (array) — это последовательность из n однотипных элементов, расположенных подряд в оперативной памяти компьютера, доступ к которым выполняется по значению так называемого ***индекса*** массива (см. рис. 1.6).

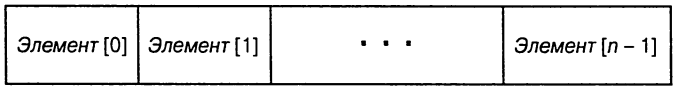


Рис. 1.6. Одномерный массив из n элементов

В большинстве случаев индекс массива, состоящего из n элементов, является целым числом и принимает значения от 0 до $n - 1$ (как показано на рис. 1.6) или от 1 до n. В некоторых языках программирования допускается, чтобы индекс массива принимал любые целочисленные значения (в том числе и отрицательные). Главное, чтобы он не выходил за пределы заранее установленного диапазона, определяющего *нижнюю* и *верхнюю* границы изменения индекса. Иногда в языках программирования для доступа к элементам массива могут использоваться нечисловые индексы. Подобные массивы называются *ассоциативными*. Например, можно создать ассоциативный массив из 12 элементов, соответствующий месяцам года, доступ к котором осуществляется по названиям месяцев.

Время обращения к любому элементу массива одинаково и не зависит от его положения в массиве. Эта особенность выгодно отличает массивы от связанных списков, о которых речь пойдет ниже. Однако не стоит забывать, что каждый элемент массива занимает одинаковое количество ячеек памяти компьютера.

На основе одномерных массивов создаются различные структуры данных. Среди них выделяются ***строки*** (string), т.е. последовательности символов заранее определенного алфавита, заканчивающиеся особым символом. Этот символ служит признаком конца строки. Строки, состоящие только из нулей или единиц, называются ***двоичными*** (***бинарными***) (binary strings) или ***битовыми*** (bit strings). Строки являются основным элементом, используемым при обработке текстовых данных, определении синтаксиса языка программирования, компилировании написанных на нем программ, а также при изучении абстрактных вычислительных моделей. Выполняемые над массивом операции в первую очередь зависят от его типа. Так, операции обычно выполняемые со строками, отличаются от операций, выполняемых над массивом чисел. В этот перечень входит:

- определение длины строки;
- сравнение двух строк, позволяющее определить, какая из них располагается раньше согласно так называемому лексикографическому порядку, т.е. как в словаре;
- объединение (конкатенация) двух строк, позволяющее сформировать из них одну строку, поместив вторую строку в конец первой.

Связанный список — это последовательность (которая может быть пустой) нескольких элементов данных, называемых ***узлами*** (nodes). В каждом узле хранится информация двух видов: собственно данные узла и одна или несколько ссылок на другие узлы связанного списка, называемых ***указателями*** (pointers). Если текущий узел связанного списка является последним, в него помещается т.н. нулевой указатель со специальным значением NULL; это говорит о том, что указатель ни на что не указывает. В ***однонаправленном связанном списке*** (singly linked list) каждый узел содержит только один указатель, в который помещается ссылка на следующий элемент списка, либо "ноль", если текущий элемент является последним (рис. 1.7).

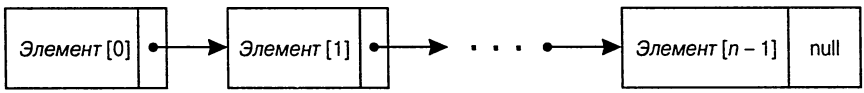


Рис. 1.7. Однонаправленный связанный список, состоящий из n элементов

Для обращения к заданному элементу связанного списка необходимо выбрать первый узел списка, а затем перемещаться по цепочке элементов до достижения нужного узла. Поэтому, в отличие от одномерных массивов, время, затрачиваемое на доступ к произвольному элементу однонаправленного списка зависит от его положения в списке. Это является существенным недостатком связанного списка. Однако у него есть и преимущество. Оно заключается в том, что для элементов связанного списка не требуется предварительное резервирование оперативной памяти компьютера. Кроме того, вставка и удаление элементов списка не представляют особого труда и выполняются достаточно быстро изменением значений нескольких указателей.

Гибкость структуры связанного списка позволяет использовать его различными способами во множестве приложений. Например, часто бывает очень удобно, чтобы в первом элементе связанного списка находился специальный узел, называемый ***заголовком*** (header). В него обычно помещают информацию о самом списке, такую как текущее количество элементов в списке и указатель на последний элемент списка.

Существует модификация структуры однонаправленного связанного списка, которая называется ***двунаправленным связанным списком*** (doubly linked list).

Отличие между ними в том, что каждый узел двунаправленного списка (кроме первого и последнего) содержит указатели на предыдущий и последующий узлы (рис. 1.8).

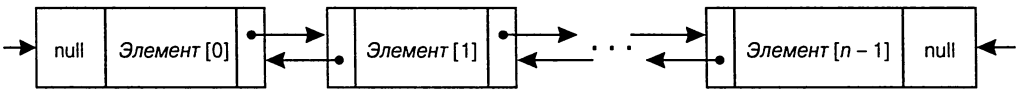


Рис. 1.8. Двунаправленный связанный список, состоящий из n элементов

Массив и связанный список чаще всего используются для представления еще одной абстрактной структуры данных, называемой линейным списком или просто списком. ***Список*** (list) представляет собой конечную последовательность элементов данных, т.е. набор элементов данных, организованных в заданном линейном порядке. К этой структуре данных могут применяться такие базовые операции, как поиск, добавление и удаление элемента.

Среди всего множества списков можно выделить два типа: стеки и очереди. ***Стеком*** (stack) называют такой список, в котором можно удалить только последний элемент, а новый элемент добавить только в его конец. Последний часто называют ***вершиной*** (top) стека, поскольку стеки обычно изображают на рисунках в виде вертикального прямоугольника (по аналогии со стопкой тарелок, которую он напоминает). Кратко сформулировать принцип работы стека можно так: "последним вошел, первым вышел" ("last-in-first-out", или LIFO), поскольку первым из стека всегда удаляется элемент, добавленный в него последним. Этим стек напоминает стопку тарелок, поскольку мы можем взять из нее только верхнюю тарелку, а новую тарелку — положить на верх стопки. Стеки широко применяются на практике, в частности, без них не обойтись при реализации рекурсивных алгоритмов.

Под ***очередью*** (queue) понимается такой список, элементы которого удаляются с одной стороны структуры (она называется ***головой*** (front)), а добавляются с другой (она называется ***хвостом*** (rear)). Первая операция называется ***удалением*** элемента из очереди (dequeue), а вторая — ***постановкой*** в очередь (enqueue). Таким образом, принцип работы очереди можно сформулировать так: "первым вошел, первым вышел" ("first-in-first-out", или FIFO). Здесь прослеживается полная аналогия с очередью в магазине, которая образуется, если продавец медленно отпускает товар или наплыв покупателей слишком велик. Очереди также находят широкое практическое применение, в частности, в некоторых алгоритмах решения задач теории графов.

Во многих важных приложениях требуется выбрать среди динамически изменяемого множества элемент с наивысшим приоритетом. Часто для решения подобной задачи применяют структуру данных, которая называется ***очередью с приоритетами***, или ***приоритетной очередью*** (priority queue). Эта очередь представ-

ляет собой совокупность элементов данных, относящихся к вполне упорядоченному универсуму[10] (чаще всего целых или вещественных чисел). К основным операциям над очередью с приоритетами относятся: поиск наибольшего элемента, извлечение наибольшего элемента и добавление нового элемента. Само собой разумеется, что последние две операции должны быть реализованы так, чтобы в результате их выполнения получалась новая очередь с приоритетами (либо переупорядочивалась старая). Проще всего реализовать рассматриваемую структуру данных на основе массива или упорядоченного массива, однако ни одно из этих решений не является оптимальным в плане достижения максимально возможной эффективности. В более удачных реализациях используется оригинальная структура данных, называемая ***пирамидой*** (heap)[11]. Эту структуру данных и основные алгоритмы сортировки на ее основе мы обсудим в разделе 6.4.

Графы

Как уже упоминалось, граф нестрого можно определить как совокупность точек на плоскости, называемых *вершинами*, или *узлами*, часть из которых соединена отрезками, называемыми *ребрами*, или *дугами*. Строго ***граф*** $G = \langle V, E \rangle$ определяется парой множеств: конечного множества элементов V, называемых ***вершинами***, и множества E, содержащего пары вершин, называемые ***ребрами***. Если пары вершин неупорядочены, т.е. пара вершин (u, v) означает то же самое, что и пара (v, u), то такой граф называют неориентированным. Если же пары упорядочены, то говорят, что ребро (u, v) ориентировано из вершины u к вершине v, при этом сам граф G называют ***ориентированным*** (directed). Ориентированные графы называют также ***диграфами*** (digraph).

Для удобства вершины графа обозначают латинскими буквами, целыми числами или даже символьными строками, если этого требуют условия задачи (рис. 1.9). Граф, показанный на рис. 1.9*а*, имеет 6 вершин и семь ребер:

$$V = \{a, b, c, d, e, f\}, \quad E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

Ориентированный граф, изображенный на рис. 1.9*б*, имеет шесть вершин и восемь ориентированных ребер (дуг):

$$V = \{a, b, c, d, e, f\}, \quad E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$

[10] Математический термин, означающий некоторое множество, фиксированное в рамках данной математической теории и содержащее в качестве элементов все объекты, рассматриваемые в этой теории. Употребляется как синоним термина ***универсальное множество***. *Математическая энциклопедия*, т. 5, М.: Сов. энц., 1985. — *Прим. ред.*

[11] Изначально термин "heap" использовался в контексте пирамидальной сортировки (heapsort), но в последнее время его основной смысл изменился, и он стал обозначать память со сборкой мусора (в частности, в языках программирования Lisp и Java) и переводиться как "куча". Однако в данной книге термину heap (который здесь переводится как "пирамида") возвращен его первоначальный смысл. — *Прим. ред.*

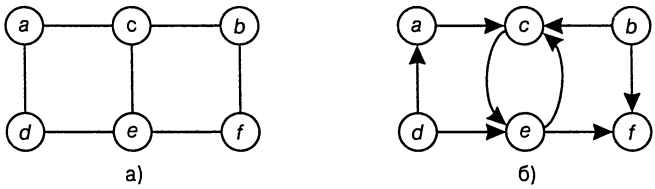


Рис. 1.9. а) Неориентированный граф; б) ориентированный граф

В приведенном выше определении графа не исключены ***петли*** (loops), или ребра, начинающиеся и заканчивающиеся в одной вершине. В книге мы будем считать, что у графа не может быть петель, если явно не указано иное. Поскольку по определению не разрешается, чтобы между двумя вершинами неориентированного графа существовало несколько ребер, то для такого графа, имеющего $|V|$ вершин и не содержащего петель, можно оценить количество возможных ребер $|E|$ с помощью следующего неравенства:

$$0 \leqslant |E| \leqslant |V|\,(|V| - 1)/2.$$

Если каждая из $|V|$ вершин графа соединяется ребрами с остальными $|V| - 1$ вершинами, то количество ребер в нем будет максимальным. Поскольку граф неориентированный, произведение $|V|\,(|V| - 1)$ необходимо разделить на 2, поскольку между двумя вершинами (u, v) имеется 2 ребра: от u к v и от v к u.

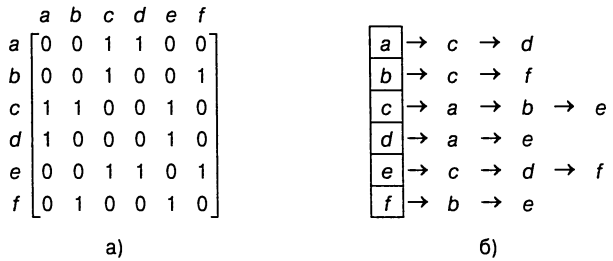
Граф, в котором каждая пара вершин соединяется ребром, называется ***полным*** (complete). Стандартное обозначение полного графа, состоящего из $|V|$ вершин, — $K_{|V|}$. Граф, в котором отсутствует относительно небольшое количество ребер, называется ***плотным*** (dense), и, напротив, граф, в котором присутствует относительно небольшое количество ребер[12], называется ***разреженным*** (sparse). От того, с каким графом мы имеем дело (плотным или разреженным), зависит способ его представления и, следовательно, время выполнения разрабатываемого или используемого алгоритма.

Представление графов

Графы, используемые в компьютерных алгоритмах, могут быть представлены двумя принципиально разными способами: в виде матрицы смежности и в виде связанных списков смежных вершин. ***Матрица смежности*** (adjacency matrix) графа, состоящего из n вершин, представляет собой булеву матрицу размером $n \times n$, в которой каждая строка и каждый столбец соответствуют одной из вершин графа. Элемент этой матрицы, находящийся на пересечении i-ой строки и j-го

[12] В этих определениях фразу "относительно небольшое" следует понимать как количество ребер, малое относительно количества ребер полного графа. — *Прим. ред.*

столбца, равен 1 в случае, если i-я и j-я вершины графа соединяются ребром, и 0 в противном случае[13]. Например, на рис. 1.10*а* показана матрица смежности для графа, изображенного на рис. 1.9*а*. Обратите внимание, что матрица смежности для неориентированного графа всегда симметрична (понятно, почему?), т.е. $A[i,j] = A[j,i]$ для всех $0 \leqslant i,j \leqslant n-1$.



а) б)

Рис. 1.10. Матрица смежности а) и связанные списки смежных вершин б) для графа, изображенного на рис. 1.9*а*

Связанные списки смежных вершин (adjacency linked lists) графа представляют собой совокупность связанных списков (по одному для каждой вершины), в которых содержится информация обо всех смежных вершинах текущей вершины (т.е. в каждом связанном списке содержится список всех вершин, соединенных ребром с текущей вершиной). Как правило, в начале каждого подобного списка располагается заголовок, содержащий информацию, идентифицирующую вершину, для которой этот список составлен. Например, на рис. 1.10*б* с помощью связанных списков смежных вершин представлен граф, изображенный на рис. 1.9*а*. Если взглянуть на все с другой стороны, то в связанных списках присутствуют вершины графа, которым в матрице смежности соответствуют единичные элементы.

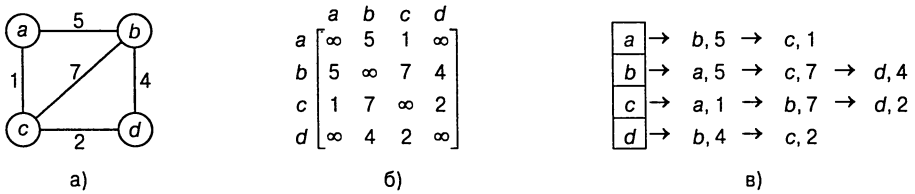
Связанные списки смежных вершин имеет смысл использовать для представления *разреженных* графов, поскольку при этом требуется меньше оперативной памяти по сравнению с матрицей смежности (которая, кстати, в этом случае будет состоять практически из одних нулей). Естественно, что дополнительную память, которую занимают указатели связанного списка, мы в расчет не принимаем. Однако ситуация в корне меняется в случае плотных графов. Вообще говоря, выбор способа представления графа зависит от типа задачи, алгоритма, используемого для ее решения и, возможно, от типа исходного графа (в зависимости от того, разреженный он или плотный).

[13] В общем случае значения элементов матрицы смежности равны не только 0 или 1 (т.е. матрица не является булевой). Каждый элемент матрицы, находящийся на пересечении i-й строки и j-го столбца, соответствует количеству ребер, соединяющих i-ю и j-ю вершины графа. — *Прим. ред.*

Взвешенные графы

Под ***взвешенным графом*** (weighted graph) подразумевается граф, ребрам которого поставлены в соответствие числа, как показано на рис. 1.11*а*. Эти числа называются ***весами*** (weights) или ***ценами***, или ***стоимостями*** (costs). Интерес к подобного типа графам был вызван большим количеством прикладных задач, таких как задача поиска кратчайшего маршрута между двумя пунктами или уже упоминавшаяся выше задача коммивояжера. Интересно, что определение кратчайшего пути может использоваться как при транспортировке грузов, так и в сетях передачи данных.

Оба рассмотренных выше способа представления графов легко приспособить и для случая взвешенных графов. Если взвешенный граф представляется с помощью матрицы смежности A, то ее элемент $A[i, j]$ должен равняться весу ребра, соединяющего i-ю и j-ю вершины. Если же эти вершины не соединены ребром, то вместо веса элемент $A[i, j]$ должен содержать какой-нибудь специальный знак, например знак бесконечности ∞. Такая матрица называется ***матрицей весов*** (weight matrix) или ***матрицей стоимости*** (cost matrix). Подобный подход проиллюстрирован на рис. 1.11*б*. (В некоторых случаях удобнее, чтобы на главной диагонали матрицы смежности располагались нули, а не знаки бесконечности.) Что касается представления взвешенного графа с помощью связанных списков смежных вершин, то каждый узел списка нужно немного расширить, включив в него не только имя смежной вершины, но и весовой коэффициент соответствующего ребра, как показано на рис. 1.11*в*.



	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

Рис. 1.11. *а*) Взвешенный граф и его представление *б*) с помощью матрицы весов и *в*) связанных списков смежных вершин

Пути и циклы

Среди множества интересных свойств графа можно выделить два особенно важных для решения огромного количества прикладных задач: ***связность*** (connectivity) и ***ациклличность*** (acyclicity). Оба этих свойства основаны на понятии пути. ***Путь***, или ***маршрут***, (path) от вершины u к вершине v можно определить как последовательность смежных (т.е. соединенных ребром) вершин, которая начинается в вершине u и заканчивается в вершине v. Если все ребра графа различны, то такой путь называют ***простым*** (simple). Под ***длиной*** (length) пути понимает-

ся общее количество вершин в последовательности, определяющей путь, минус единица. Другими словами, длина пути равна количеству ребер, через которые он проходит. Например, на рис. 1.9*а* от вершины a до вершины f можно проложить простой путь длиной 3: a, c, b, f. В то же время между этими вершинами существует путь (не простой) длиной 5: a, c, e, c, b, f.

В случае ориентированных графов нас обычно будут интересовать ориентированные пути. Под ***ориентированным маршрутом*** (directed path) понимается последовательность вершин, в которой каждая следующая друг за другом пара вершин (u, v) соединена дугой, начинающейся в вершине u и заканчивающейся в вершине v. Например, на рис. 1.9*б* существует следующий ориентированный путь из вершины a к вершине f: a, c, e, f.

Граф называется ***связным*** (connected), если для любой пары его вершин u и v существует путь от u к v. Объяснить это свойство графа, что называется, "на пальцах" можно так: если создать модель этого графа, связав между собой веревками (которые будут выполнять роль ребер) несколько мячей (они будут выполнять роль вершин), то все предметы можно будет удержать в одной руке. Если граф не является связным, то модель будет состоять из нескольких отдельных участков, которые называются связными компонентами. Строго ***связный компонент*** (connected component) графа можно определить как максимальный связный подграф[14] данного графа, который нельзя расширить за счет включения дополнительной вершины, смежной с одной из ее вершин. Например, графы, изображенные на рис. 1.9*а* и 1.11*а*, являются связными, тогда как граф, изображенный на рис. 1.12, не является связным, поскольку нельзя проложить путь, скажем, от вершины a к вершине f. Граф, изображенный на рис. 1.12, имеет два связанных компонента с вершинами $\{a, b, c, d, e\}$ и $\{f, g, h, i\}$ соответственно.

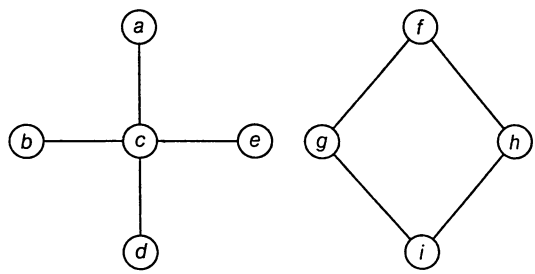


Рис. 1.12. Пример несвязного графа

Графы с несколькими компонентами связности успешно применяются при решении реальных прикладных задач. В качестве примера можно привести пред-

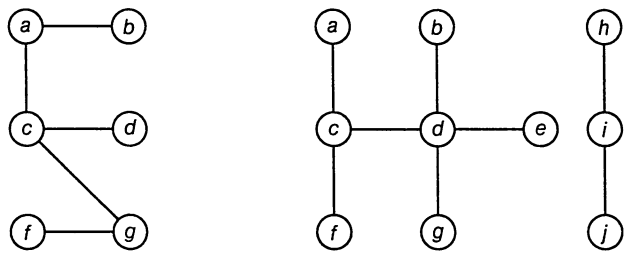
[14] ***Подграфом*** (subgraph) графа $G = \langle V, E \rangle$ является граф $G' = \langle V', E' \rangle$, такой, что $V' \subseteq V$ и $E' \subseteq E$.

ставление в виде графа системы магистральных шоссейных дорог (highway system), проложенных между штатами в США (можете объяснить, почему?).

Во многих прикладных задачах важно знать, содержит ли рассматриваемый граф циклы. Под ***циклом*** (cycle) понимается простой путь положительной длины, который начинается и заканчивается в одной и той же вершине. Например, в графе, изображенном на рис. 1.12, циклом является путь: f, h, i, g, f. Граф, не содержащий циклов, называют ***ациклическим*** (acyclic). Ациклические графы будут рассмотрены в следующем подразделе.

Деревья

Деревом (tree) (точнее, ***свободным деревом*** (free tree)) называется связный ациклический граф (рис. 1.13*а*). Граф, который не содержит циклов, но не обязательно является связным, называется ***лесом*** (forest). Каждая компонента связности леса является деревом (рис. 1.13*б*).



а) б)

Рис. 1.13. Пример *а*) дерева и *б*) леса

Деревья имеют несколько важных свойств, не присущих другим графам. В частности, число ребер в дереве всегда на единицу меньше, чем число его вершин:

$$|E| = |V| - 1.$$

На примере графа, показанного на рис. 1.12, можно сделать вывод, что это свойство является необходимым, но не достаточным, чтобы граф был деревом. Тем не менее для связных графов это свойство является достаточным, поэтому с его помощью удобно определять, содержит ли граф цикл.

Корневые деревья

Еще одним важным свойством деревьев является следующее: для любых двух вершин дерева существует только один простой путь от одной вершины к другой. Это свойство позволяет назначить произвольный узел в свободном дереве ***корнем*** (root) и преобразовать свободное дерево в так называемое ***корневое дерево*** (rooted

tree). Последнее обычно изображается так, чтобы его корень был вверху, т.е. располагался на так называемом нулевом уровне дерева. Ниже корня располагаются смежные с ним вершины. Они составляют первый уровень дерева. Вершины, соединенные с корнем двумя ребрами, составляют следующий (т.е. второй) уровень дерева, и т.д. На рис. 1.14 показано, как преобразовать свободное дерево в корневое.

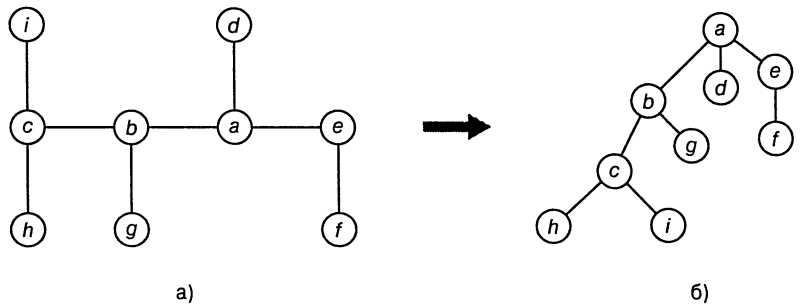


Рис. 1.14. Преобразование *а*) свободного дерева в *б*) корневое

Корневые деревья играют в информатике гораздо более важную роль, чем свободные деревья. Поэтому часто для краткости их называют просто "деревьями". Их основное назначение — описание иерархических структур, таких как структура каталогов файловой системы или организационная структура предприятия. Однако существуют и другие, менее очевидные области применения деревьев, например для организации словарей (об этом пойдет речь в следующем подразделе), для эффективного хранения очень больших массивов данных (см. раздел 7.4), для кодирования данных (см. раздел 9.4). Как вы увидите в главе 2, деревья также находят применение при анализе рекурсивных алгоритмов. И, чтобы закончить этот далеко не полный перечень возможных применений деревьев, следует упомянуть так называемые ***деревья пространств состояний*** (state-space trees). С их помощью можно объяснить два важных метода проектирования алгоритмов: поиск с возвратом (backtracking) и метод ветвей и границ (branch-and-bound). О них речь пойдет в разделах 11.1 и 11.2.

Для любой вершины v в дереве T верно следующее: все вершины, принадлежащие простому маршруту от корня дерева до этой вершины, называются ***предками*** (ancestors) v. Сама вершина v обычно считается собственным предком. Множество предков, в которое не включен собственный предок, называется множеством ***истинных предков*** (proper ancestors), или просто ***истинными предками***. Если (u, v) является последним ребром, принадлежащим простому маршруту, ведущему из корня дерева до вершины v (и $u \neq v$), то говорят, что u является ***родителем*** (parent) v, а v называют ***потомком***, или ***дочерней*** (child) вершиной u. Вершины, имеющие общего родителя, называются ***родственными***, или ***сестринскими***

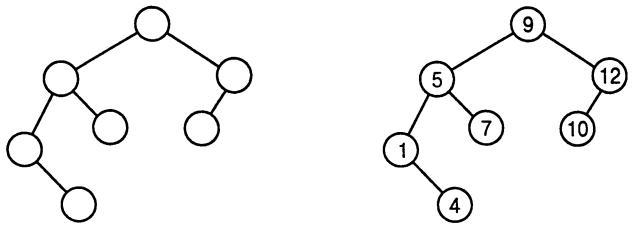
(sibling). Вершина, у которой нет потомков, называется ***листом*** (leaf). Вершина, у которой есть как минимум один потомок, называется ***родительской*** (parental). Совокупность всех вершин, для которых вершина v является предком, называют ***потомками*** (descendants) v. Вершина v вместе со всем своими потомками называется ***поддеревом*** (subtree) дерева T с корнем (поддерева) в вершине v. Таким образом, для дерева, изображенного на рис. 1.14*б*, корнем является вершина a; вершины d, g, f, h и i являются листьями; вершины a, b, e и c являются родительскими. Родительской вершиной для b является вершина a; потомками вершины b являются вершины c и g; родственными для b являются вершины d и e; вершинами поддерева с корнем в b являются: $\{b, c, g, h, i\}$.

Под ***глубиной*** (depth) вершины v понимается длина простого пути от корня до этой вершины. Под ***высотой*** (height) дерева понимается длина наибольшего простого пути от его корня до одного из листьев. Например, в дереве, изображенном на рис. 1.14*б*, глубина вершины c равна 2, а высота дерева равна 3. Таким образом, если сосчитать уровни дерева сверху вниз, начиная с корня (ему соответствует нулевой уровень), то глубина вершины будет равна уровню этой вершины в дереве, а высота дерева будет соответствовать максимальному уровню, на котором могут находиться вершины в дереве. (Обратите внимание, что в некоторых книгах высота дерева определяется как количество уровней в дереве. При подобном определении высота дерева будет на единицу больше, чем высота, которая соответствует длине наибольшего простого маршрута от корня до одного из листьев.)

Упорядоченные деревья

Упорядоченным (ordered) называется такое корневое дерево, в котором упорядочены все потомки каждой вершины. Принято считать, что на диаграмме деревья изображаются так, чтобы все потомки были упорядочены слева направо. ***Двоичное*** (***бинарное***) ***дерево*** (binary tree) можно определить как упорядоченное дерево, каждая вершина которого имеет не более двух потомков, причем каждый из потомков считается либо ***левым*** (left child) либо ***правым*** (right child) потомком своего родителя. Поддерево, корень которого находится в левом (правом) потомке вершины, называется ***левым*** (***правым***) поддеревом этой вершины. Пример бинарного дерева показан на рис. 1.15*а*.

На рис. 1.15*б* вершинам бинарного дерева, показанного на рис. 1.15*а*, назначены числа. Обратите внимание, что число, которое назначено каждой родительской вершине, больше, чем число, назначенное ее левому потомку, и меньше, чем число, назначенное ее правому потомку. Деревья подобного типа называют ***бинарным деревом поиска*** (binary search trees). Бинарные деревья и бинарные деревья поиска находят широкое применение в информатике. С некоторыми из них вы столкнетесь при чтении этой книги. Бинарные деревья поиска являются частным случаем более общего типа деревьев поиска, которые называются ***многоканальными де-***



а) б)

Рис. 1.15. Пример *а*) бинарного дерева и *б*) бинарного дерева поиска

ревьями поиска (multiway search trees). Последние незаменимы при организации эффективного хранения на дисках файлов очень больших размеров.

Ниже в этой книге мы покажем, что эффективность большинства основных алгоритмов, в которых используются бинарные деревья поиска и их вариации, непосредственно зависит от высоты дерева. Поэтому приведенное ниже неравенство для высоты бинарного дерева h, содержащего n вершин, особенно важно для анализа подобных алгоритмов:

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1.$$

При использовании в компьютерных программах бинарные деревья обычно реализуют в виде связанного списка, узлы которого соответствуют вершинам дерева. В каждый узел помещается информация о соответствующей ему вершине дерева (ее имя, присвоенное значение и т.д.), а также два указателя на его левого и правого потомка. На рис. 1.16 показана одна из подобных реализаций бинарного дерева поиска, показанного на рис. 1.15*б*.

Упорядоченное дерево в компьютере можно представить в виде родительской вершины с указателями на всех ее потомков. Однако такое представление не совсем удобно в том случае, когда количество потомков в процессе работы программы меняется в очень широких пределах. Чтобы избежать подобного неудобства, можно воспользоваться структурой данных, каждый узел которой будет иметь только два указателя (по аналогии с представлением бинарных деревьев). При этом левый указатель будет содержать ссылку на первого потомка текущей вершины, а правый — на следующую родственную ей вершину. Поэтому такое представление произвольного упорядоченного дерева называют "***первый потомок — следующий родственник***" (first child — next sibling). Таким образом, все родственные вершины будут связаны в один список с помощью правого указателя текущей вершины. При этом ссылка на первый элемент списка будет содержаться в левом указателе ее родителя. На рис. 1.17*а* показано подобное представление дерева, показанного на рис. 1.14*б*. Нетрудно заметить, что в результате подобного представления упорядоченное дерево оказалось преобразовано в бинарное. Поэтому

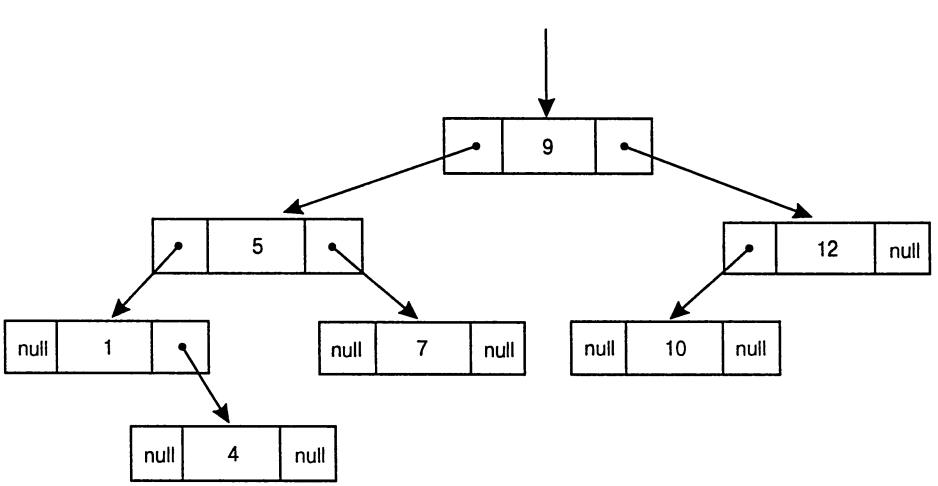


Рис. 1.16. Типовая реализация бинарного дерева поиска, показанного на рис. 1.15*б*

говорят, что данное бинарное дерево ассоциировано с упорядоченным деревом. Чтобы убедиться в этом, достаточно "развернуть" стрелки указателей по часовой стрелке примерно на 45°, как показано на рис 1.17*б*.

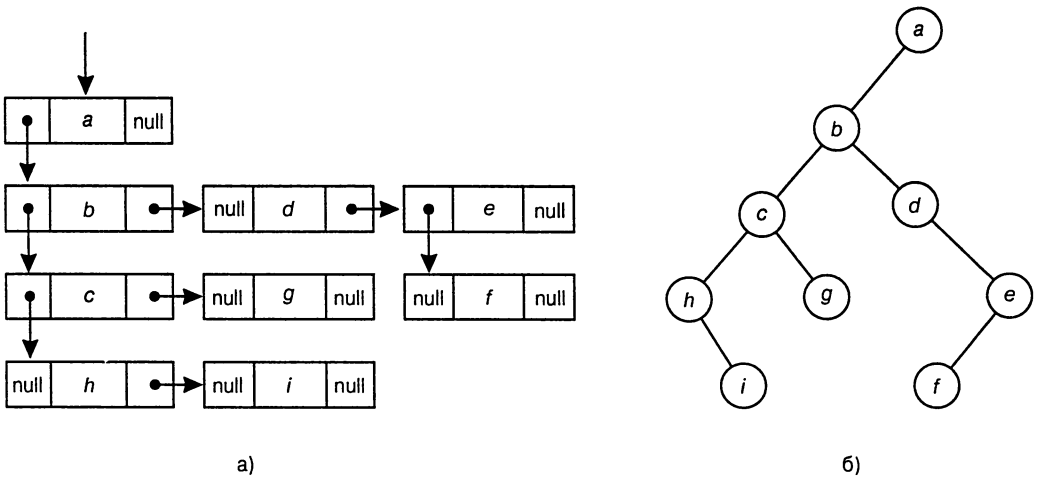


Рис. 1.17. *а*) Представление графа, показанного на рис. 1.14*б* в виде структуры "первый потомок — следующий родственник"; *б*) ассоциированное с ним бинарное дерево

Множества и словари

Понятие множества играет основную роль в математике. ***Множество*** (set) можно охарактеризовать как неупорядоченную совокупность (которая может быть

пуста) отдельных элементов, называемых элементами множества. Конкретное множество определяется либо в виде явного списка элементов (например, $S = \{2, 3, 5, 7\}$), либо в виде указания условия, которому должны удовлетворять все элементы множества и только они (например, $S = \{n$, где n — простое число, меньшее $10\}$). Основными операциями над множествами являются: проверка на членство в данном множестве (т.е. принадлежит ли элемент множеству; другими словами, нужно найти заданный элемент среди элементов множества), поиск объединения двух множеств (т.е. формирование нового множества, элементы которого принадлежат либо первому, либо второму множеству, либо обоим множествам одновременно), а также поиск пересечения двух множеств (т.е. формирование нового множества, элементы которого принадлежат только обоим множествам одновременно).

При разработке компьютерных приложений множества можно реализовать двумя способами. В первом случае рассматриваются только множества, которые являются подмножествами некоторого большого множества U, называемого ***универсальным множеством***, или ***универсумом*** (universal set). Если множество U состоит из n элементов, то любое его подмножество S можно представить в виде строки из n битов, называемой ***битовым вектором*** (bit vector), в которой i-й элемент равен 1 тогда и только тогда, когда i-й элемент множества U включен в подмножество S. Поэтому, возвращаясь к нашему примеру, если $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, то подмножество $S = \{2, 3, 5, 7\}$ можно представить в виде битовой строки 011010100. Описанный способ представления множеств позволяет реализовать стандартный набор операций над множеством в виде очень быстрых процедур. Однако платой за скорость является большой объем оперативной памяти, необходимой для представления элементов множества[15].

Чаще всего в компьютерных программах используется второй способ представления множества. Он заключается в представлении элементов множества в виде связанного списка. Само собой разумеется, речь идет о представлении только конечных множеств. К счастью, в отличие от математики, данный вид множеств наиболее широко используется в компьютерных приложениях. Тем не менее необходимо обратить внимание на два принципиальных различия между множествами и списками. Во-первых, в множестве не может быть одинаковых элементов, а в списке — может. Иногда это требование к уникальности элементов обходят путем введения так называемого ***мультимножества*** (multiset), или ***пакета*** (bag), т.е. неупорядоченной совокупности элементов, которые необязательно должны быть различными. Во-вторых, поскольку множество — это неупорядоченная совокупность элементов, изменение порядка его элементов не изменяет само множество. В свою очередь, список определяется как упорядоченная со-

[15] В нашей книге это первый пример компромисса между временем выполнения алгоритма и требуемым объемом оперативной памяти. Более подробно мы поговорим об этом в главе 7.

вокупность элементов, что в корне противоположно понятию множества. Выше мы перечислили важные теоретические различия между множеством и списком, однако, к счастью, они не так важны для большинства приложений. Кроме того, стоит упомянуть, что в том случае, когда множество представляется в виде списка, то для ряда приложений имеет смысл поддерживать его в отсортированном виде.

В прикладных программах над множеством или мультимножеством чаще всего выполняются следующие операции: поиск заданного элемента, добавление нового элемента и удаление элемента из совокупности. Структуру данных, задействованную в реализации трех этих операций, называют ***словарем*** (dictionary). Заметим, что о взаимосвязи этой структуры данных с задачами поиска говорилось выше в разделе 1.3. Очевидно, что здесь речь идет о выполнении поиска в динамически изменяемом контексте. Поэтому эффективная реализация словаря должна быть результатом достижения компромисса между эффективным выполнением поиска и двух других операций над множеством. В действительности существует несколько способов реализации словаря. Они могут быть как очень простыми, с использованием обычных массивов сортированных (или несортированных) элементов, так и довольно сложными. В последнем случае используются методы хеширования и сбалансированные деревья поиска, которые будут описаны в последующих главах этой книги.

При решении ряда прикладных задач из области вычислительной техники нужно динамически разбить множество, состоящее из n элементов, на ряд непересекающихся подмножеств. После инициализации множества как набора из n одноэлементных подмножеств задача сводится к последовательности операций поиска и объединения. Эта задача называется задачей ***объединения множества*** (set union). Эффективные алгоритмы решения этой задачи будут описаны в разделе 9.2 при рассмотрении одного из важных ее применений.

Вы, наверное, уже заметили, что в приведенном выше обзоре основных структур данных мы почти всегда упоминали об особенностях операций, которые обычно выполняются над рассматриваемыми структурами данных. Подобную тесную взаимосвязь между структурами данных и выполняемыми над ними операциями специалисты в области информатики заметили уже давно. Это навело их на мысль о существовании так называемых ***абстрактных типов данных*** (abstract data type, или ***ADT***), т.е. множества абстрактных объектов, представляющих элементы данных, и определенного на нем набора операций, которые могут быть выполнены над элементами этого множества. В качестве иллюстрации этого понятия перечитайте, например, приведенные выше определения приоритетной очереди и словаря. Несмотря на то что абстрактные типы данных можно реализовать с помощью старых процедурных языков программирования, таких как Pascal (в качестве примера см. [5]), все-таки намного удобнее делать это с помощью объектно-ориентированных языков программирования, таких как C++ или Java, в которых абстрактные типы данных поддерживаются с помощью ***классов*** (classes).

Упражнения 1.4

1. Поясните, как можно реализовать каждую из перечисленных ниже операций над массивом так, чтобы время ее выполнения не зависело от размера массива n.
   а) Удаление i-го элемента массива $1 \leqslant i \leqslant n$).
   б) Удаление i-го элемента отсортированного массива (разумеется, образовавшийся после удаления массив также должен остаться отсортированным).
2. Предположим, нужно найти число в списке, состоящем из n элементов. Насколько упростится решение, если список будет отсортирован? Рассмотрите отдельные решения для случаев, когда
   а) список представлен в виде массива;
   б) список представлен в виде связанного списка.
3. а) Каким будет содержимое стека после выполнения каждой команды приведенной ниже последовательности, считая, что в исходном состоянии стек пуст:

$$push\,(a),\ push\,(b),\ pop,\ push\,(c),\ push\,(d),\ pop.$$

   б) Изобразите содержимое очереди после выполнения каждой команды приведенной ниже последовательности, считая, что в исходном состоянии очередь пуста:

$$enqueue\,(a),\ enqueue\,(b),\ dequeue,\ enqueue\,(c),\ enqueue\,(d),\ dequeue.$$

4. а) Предположим, A является матрицей смежности неориентированного графа. Объясните, какое из свойств матрицы свидетельствует о том, что:
      1) граф является полным;
      2) граф содержит петли, т.е. какая-либо из вершин соединена ребром сама с собой;
      3) граф содержит изолированную вершину, т.е. в нем встречается вершина, не соединенная ребрами с другими вершинами.
   б) Дайте ответы на те же вопросы для представления в виде связанных списков смежности.
5. Приведите подробное описание алгоритма преобразования свободного дерева в корневое дерево, корень которого находится в заранее определенной вершине свободного дерева.

6. Докажите приведенное ниже неравенство для высоты бинарного дерева h, содержащего n вершин:

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1.$$

7. Покажите, как можно реализовать абстрактный тип данных очереди с приоритетами в виде:
   а) (несортированного) массива;
   б) отсортированного массива;
   в) бинарного дерева поиска.
8. Как бы вы реализовали словарь, содержащий достаточно малое количество элементов n, при условии, что все его элементы различны (например, представляют собой названия 50 штатов США)? Опишите реализацию каждой операции, выполняемой словарем.
9. Для каждой из перечисленных ниже задач укажите наиболее подходящую с вашей точки зрения структуру данных.
   а) Система автоматического ответа на телефонные звонки с учетом их приоритетности.
   б) Система отправки покупателям счетов за сделанные заказы, которая бы учитывала порядок поступления заказов.
   в) Реализация калькулятора для вычисления простых арифметических выражений.

10. Разработайте алгоритм проверки того, являются ли два заданных слова анаграммами, т.е. того, что одно из слов может быть получено путем перестановки букв другого слова. (Например, анаграммами являются слова *кот* и *ток*.)

Резюме

- *Алгоритм* — это последовательность четко определенных инструкций, предназначенных для решения некоторой задачи за ограниченный промежуток времени. Входные данные, обрабатываемые алгоритмом, определяют *экземпляры задачи*, решаемой с помощью выбранного алгоритма.
- Алгоритмы могут быть описаны на естественном языке либо на псевдокоде. Они также могут быть реализованы в виде компьютерных программ.
- Существует несколько способов классификации алгоритмов, но среди них можно выделить две принципиальные альтернативы:

  - группирование алгоритмов в соответствии с типом решаемой с их помощью задачи;
  - группирование алгоритмов в соответствии с лежащими в их основе методами проектирования.

- Важными типами задач являются: сортировка, поиск, обработка строк, задачи теории графов, комбинаторные задачи, геометрические задачи и численные задачи.
- *Метод проектирования* алгоритма — это универсальный подход, применяемый для алгоритмического решения широкого круга задач, относящихся к различным областям вычислительной техники.
- Несмотря на то что проектирование алгоритма, безусловно, — творческий процесс, в нем тем не менее можно выделить последовательность взаимосвязанных действий, которые необходимо выполнить, как показано на рис. 1.2.
- Хороший алгоритм получается, как правило, в результате кропотливой циклической работы, связанной с возможными переделками.
- Часто одну и ту же задачу можно решить с помощью нескольких алгоритмов. Например, для вычисления наибольшего общего делителя двух целых чисел в этой главе было описано три алгоритма: *алгоритм Евклида*, алгоритм последовательного перебора целых чисел и алгоритм, который вы изучали в средней школе. Последний алгоритм был нами немного усовершенствован: для получения упорядоченного списка простых чисел мы применили алгоритм под названием *решето Эратосфена*.
- Алгоритмы работают с данными. Поэтому для эффективного решения алгоритмической задачи необходимо правильно структурировать данные. Среди простых структур данных самыми важными являются *массив* и *связанный список*. Они используются для представления более абстрактных структур данных, таких как *список*, *стек*, *очередь*, *граф* (представляется с помощью *матрицы смежности* или *связанных списков смежных вершин*), *бинарное дерево* и *множество*.
- Множество абстрактных объектов, представляющих элементы данных и набор операций, которые могут быть выполнены над элементами этого множества, называются *абстрактным типом данных* (ADT). Важными примерами абстрактных типов данных являются *список*, *стек*, *очередь*, *очередь с приоритетами* и *словарь*. В современных языках программирования реализация абстрактных типов данных поддерживается с помощью классов.

Глава 2

Основы анализа эффективности алгоритмов

> Когда вы можете измерить и выразить в цифрах то, о чем говорите, вы что-то знаете об изучаемом предмете. Однако когда вы не можете выразить в цифрах то, о чем говорите, вы имеете весьма скудное представление об изучаемом предмете: возможно, это начало познания, но вряд ли вам удастся довести ваши идеи до состояния науки, о чем бы ни шла речь.
>
> — Лорд Кельвин (Lord Kelvin) (1824–1907)

> Не все, что можно подсчитать, принимают в расчет, и не все, что принимают в расчет, можно подсчитать.
>
> — Альберт Эйнштейн (Albert Einstein) (1824–1907)

Эта глава посвящена анализу алгоритмов. В американском энциклопедическом словаре *American Heritage Dictionary* слово "анализ" ("analysis") определяется как "разделение материального или духовного целого на составные части с целью их последующего изучения". Таким образом, все основные характеристики алгоритмов, упомянутые в разделе 1.2, должны быть изучены на вполне законных основаниях. Однако формально термин "анализ алгоритмов" обычно используется в более узком смысле и означает процесс исследования эффективности алгоритмов, которую можно оценить по двум параметрам: времени выполнения алгоритма и требуемому объему оперативной памяти. Важность эффективности очень легко объяснить. Во-первых, в отличие от таких характеристик алгоритма, как простота и универсальность, эффективность можно выразить количественно. Во-вторых, можно доказать, что эффективность является вопросом первостепенной важности с практической точки зрения. Однако сделать это почти всегда довольно сложно, учитывая быстродействие современных компьютеров и объем их оперативной памяти. Тем не менее в этой главе речь пойдет именно об эффективности алгоритмов.

Начнем обсуждение с описания в разделе 2.1 общих основ процесса анализа эффективности алгоритмов. Наверное, этот раздел является самым важным в главе. Более того, поскольку в нем рассмотрены фундаментальные понятия, можно сказать что этот раздел является самым важным во всей книге.

В разделе 2.2 мы познакомимся с тремя условными обозначениями: O (прописная "О"), Θ (прописная греческая "тета") и Ω (прописная греческая "омега"). Они были взяты из математики и являются *языком*, на котором происходит обсуждение эффективности алгоритмов.

В разделе 2.3 мы покажем, как полученные в разделе 2.1 базовые знания можно систематически применять к анализу эффективности нерекурсивных алгоритмов. Основным средством такого анализа является запись суммы, выражающая время выполнения алгоритма и последующее упрощение этой суммы с помощью стандартных правил суммирования.

В разделе 2.4 мы покажем, как полученные в разделе 2.1 базовые знания применять к анализу эффективности рекурсивных алгоритмов. Здесь основным средством анализа будет уже не сумма, а специальный тип уравнения, называемого рекуррентным. В этом разделе будет объяснено, как строить эти рекуррентные уравнения, а также приведен метод их решения.

Несмотря на то что в первых четырех разделах этой главы основы анализа эффективности алгоритмов и методы их применения проиллюстрированы с помощью разнообразных примеров, в разделе 2.5 будет рассмотрен еще один пример, посвященный числам Фибоначчи. Хотя эта знаменитая последовательность чисел была придумана более 800 лет назад, она по-прежнему широко применяется и в информатике, и в других прикладных науках. Благодаря обсуждению последовательности Фибоначчи нам удастся естественным образом познакомить читателя с важным классом рекуррентных соотношений, которые невозможно решить методами, рассмотренными в разделе 2.4. Кроме того, мы также обсудим несколько алгоритмов вычисления последовательности чисел Фибоначчи и на их основе выскажем несколько общих замечаний по поводу эффективности алгоритмов и методов ее анализа.

Методы, описанные в разделах 2.3. и 2.4, являются достаточно мощным средством анализа эффективности многих алгоритмов, выполняемого с математической точностью, однако они понятны далеко не всем. В последних двух разделах этой главы будут описаны два подхода (эмпирический анализ и визуализация алгоритма), которые дополняют чисто математические методики, изложенные в разделах 2.3 и 2.4. Хотя эти методики сравнительно новы и еще не так широко распространены, как их математические аналоги, тем не менее им отводится особое место среди средств анализа эффективности алгоритмов.

2.1 Основы анализа

В этом разделе мы познакомимся с основными понятиями, использующимся для анализа эффективности алгоритмов. Начнем с рассмотрения двух видов эффективности: временно́й и пространственной. ***Временная эффективность*** (***time efficiency***) является индикатором скорости работы алгоритма. ***Пространственная эффективность*** (***space efficiency***) показывает, сколько дополнительной оперативной памяти нужно для работы алгоритма. На заре развития электронных вычислительных машин (ЭВМ) особое значение имели два ресурса: быстродействие центрального процессора и объем оперативной памяти. Однако более чем полувековой процесс непрерывного технического прогресса привел к тому, что быстродействие и объем оперативной памяти вычислительных устройств увеличились во много раз. Теперь требования к дополнительному объему оперативной памяти, необходимой для работы алгоритма, стали не так важны, как раньше. Однако здесь следует сделать оговорку, что, конечно же, существует определенная разница между быстрой основной памятью, относительно медленной вторичной памятью и кэш-памятью. Что касается временны́х характеристик алгоритма и быстродействия современных компьютеров, то здесь, к сожалению, нельзя сказать, что эти вопросы совсем сняты с повестки дня. Более того, исходя из опыта научных исследований, можно сказать, что для большинства задач удается добиться существенного выигрыша именно в скорости работы алгоритма, а не в сокращении требуемого объема памяти (зачастую первое происходит за счет второго). Поэтому по сложившейся во многих учебниках по алгоритмам хорошей традиции, мы в основном сосредоточим внимание на временно́й эффективности. Тем не менее описанную в этом разделе аналитическую основу вы можете с успехом применить и для анализа пространственной эффективности алгоритмов.

Оценка размера входных данных

Начнем обсуждение с констатации очевидного факта, что время выполнения большинства алгоритмов напрямую зависит от размера вводимых данных (т.е. чем больше размер, тем дольше работает алгоритм). Например, довольно долго длится процесс сортировки больших массивов данных, перемножения больших матриц и т.п. Поэтому вполне логично описать эффективность алгоритма в виде функции от некоторого параметра n, связанного с размером входных данных[1]. В большинстве случаев выбрать такой параметр не представляет большого труда. Например, для задач, связанных с сортировкой, поиском, нахождением наименьшего элемента в списке и многих других, связанных с обработкой списков, таким параметром

[1] В некоторых алгоритмах для оценки размерности входных данных может использоваться сразу несколько параметров (например, количество вершин и ребер для тех алгоритмов, в которых граф представляется в виде связанных списков смежных вершин).

будет размер списка. Для задачи вычисления значения многочлена степени n $p(x) = a_n x^n + \cdots + a_0$ таким параметром может быть степень многочлена или количество его коэффициентов, которое на единицу больше степени многочлена. Ниже мы увидим, что подобное несущественное отличие не влияет на результаты анализа.

Конечно, бывают случаи, когда выбор параметра, показывающего размер входных данных, имеет особое значение. Один из примеров подобных случаев — перемножение двух матриц размером $n \times n$. Существует две подходящих единицы измерения размера данной задачи. Первая и наиболее часто используемая — это порядок матрицы n. Однако существует и другая, не менее подходящая единица измерения, — количество N перемножаемых элементов в матрицах. Последняя единица к тому же более универсальна, поскольку ее можно применять не только к квадратным матрицам. Поскольку существует простая формула, связывающая эти две единицы измерения, мы легко можем перейти от одной единицы к другой, однако искомая эффективность алгоритма будет в значительной степени отличаться в зависимости от того, какая из двух единиц измерения была использована при решении задачи (см. задачу 2 упражнения 2.1).

На выбор подходящей системы измерений размера задачи могут повлиять выполняемые рассматриваемым алгоритмом действия. Например, как оценить размер входных данных для алгоритма, выполняющего проверку орфографии? Если в алгоритме проверяется каждый вводимый символ, то оценить размер входных данных мы можем, подсчитав количество символов во входном потоке. Если же в алгоритме происходит обработка текста по словам, то нужно подсчитать количество слов во входном потоке.

Мы должны сделать специальное замечание по поводу оценки размера входных данных для алгоритмов, связанных с нахождением чисел, удовлетворяющих определенным условиям (например, проверяющих, является ли заданное целое число n простым). Для подобных алгоритмов кибернетики предпочитают оценивать размер входных данных по количеству битов b в двоичном представлении числа n:

$$b = \lfloor \log_2 n \rfloor + 1. \tag{2.1}$$

Подобная система измерений позволяет лучше оценить эффективность рассматриваемого алгоритма.

Единицы измерения времени выполнения алгоритма

Мы должны рассмотреть еще один вопрос, касающийся единиц измерения времени выполнения алгоритма. Безусловно, для этой цели можно просто воспользоваться общепринятыми единицами измерения времени — секундой, миллисекундой и т.д. и с их помощью оценить время выполнения программы, реали-

зующей рассматриваемый алгоритм. Однако у такого подхода существуют явные недостатки, поскольку результаты измерений будут зависеть от:

- быстродействия конкретного компьютера;
- тщательности реализации алгоритма в виде программы;
- типа компилятора, использованного для генерации машинного кода;
- точности хронометрирования реального времени выполнения программы.

Поскольку перед нами стоит задача измерения эффективности *алгоритма*, а не реализующей его программы, мы должны воспользоваться такой системой измерений, которая бы не зависела от приведенных выше посторонних факторов.

Один из возможных способов решения этой проблемы состоит в подсчете того, сколько раз выполняется каждая операция алгоритма. Однако подобный подход слишком сложен и, как мы увидим в дальнейшем, чаще всего не нужен. Поэтому мы должны составить список наиболее важных операций, выполняемых в алгоритме, называемых ***основными***, или ***базовыми операциями*** (***basic operation***), определить, какие из них вносят наибольший вклад в общее время выполнения алгоритма, и вычислить, сколько раз эти операции выполняются.

Как правило, составить список основных операций алгоритма совсем нетрудно. Обычно в него включают наиболее длительные по времени операции, выполняемые во внутреннем цикле алгоритма. Например, в большинстве алгоритмов сортировки используется метод сравнения двух элементов (ключей) списка, который сортируется. Для подобного типа алгоритмов основной является операция сравнения ключей. В качестве еще одного примера рассмотрим алгоритмы перемножения матриц и вычисления значения многочлена. В них используются две основные операции: умножение и сложение. На большинстве компьютеров команда умножения двух целых чисел выполняется намного дольше, чем сложение[2]. Поэтому она является безусловным кандидатом на включение в список основных операций.

Таким образом, закладывая основу для анализа временно́й эффективности алгоритмов, мы предполагаем, что этот показатель будет оцениваться по количеству основных операций, которые должен выполнить алгоритм при обработке входных данных размера n. О том, как определить этот показатель для нерекурсивных и рекурсивных алгоритмов, вы узнаете из разделов 2.3 и 2.4, соответственно.

Рассмотрим один важный пример. Предположим, что c_{op} — время выполнения основной операции алгоритма на конкретном компьютере, а $C(n)$ — количество раз, которые эта операция должна быть выполнена при работе алгоритма. Тогда время выполнения программной реализации этого алгоритма на данном компью-

[2]Кроме компьютеров с так называемой RISC-архитектурой. В качестве примера сравните временны́е характеристики команд, приведенные в [[61], с. 185–186].

тере $T(n)$ можно приблизительно определить по следующей формуле:

$$T(n) \approx c_{op} C(n).$$

Разумеется, эту формулу нужно использовать очень аккуратно. В значении счетчика $C(n)$ не учитывается количество выполняемых алгоритмом операций, не относящихся к основным. Кроме того, обычно значение этого счетчика также можно определить только приблизительно. Более того, значение константы c_{op} также можно определить лишь приблизительно и оценить ее точность весьма непросто. Тем не менее, эта формула дает приемлемую оценку времени выполнения алгоритма для не бесконечно больших и не бесконечно малых значений n. Она также позволяет ответить на вопросы наподобие: во сколько раз быстрее будет работать реализация данного алгоритма на компьютере, быстродействие которого больше нашего в 10 раз? Казалось бы, ответ очевиден — в 10 раз. Или пусть, например, $C(n) = n(n-1)/2$. Насколько дольше будет выполняться программа, если удвоить размер входных данных? Ответ будет такой: приблизительно в четыре раза медленнее. В самом деле, для достаточно больших n справедлива следующая формула:

$$C(n) = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2.$$

Поэтому

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op} C(2n)}{c_{op} C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2} n^2} = 4$$

Обратите внимание, что для ответа на второй вопрос не нужно знать реальное значение c_{op}, поскольку в приведенной выше дроби оно сокращается. Обратите также внимание, что постоянный множитель $1/2$ в формуле для $C(n)$ тоже сокращается. По этим причинам в процессе анализа эффективности при достаточно больших размерах входных данных не учитывают постоянные множители, а сосредотачиваются на оценке ***порядка роста*** (***order of growth***) количества основных операций с точностью до постоянного множителя.

Порядок роста

Почему выше мы сделали замечание по поводу вычисления порядка роста количества основных операций алгоритма для достаточно больших размеров входных данных? Дело в том, что при малых размерах входных данных невозможно заметить разницу во времени выполнения между эффективным и неэффективным алгоритмом. Например, при вычислении НОД двух небольших чисел, совершенно непонятно во сколько раз алгоритм Евклида работает быстрее двух других алгоритмов, рассмотренных в разделе 1.1. Непонятным остается также вопрос,

почему нас так волнует, какой из алгоритмов быстрее и во сколько раз. И только тогда, когда нужно вычислить НОД двух очень больших чисел, все эти вопросы, связанные с разной эффективностью алгоритмов, выходят на первый план и становятся понятными. Для больших значений n вычисляют порядок роста функции. В табл. 2.1 эти значения приведены для некоторых функций, играющих особую роль в процессе анализа алгоритмов.

Таблица 2.1. Приближенные значения важных для анализа алгоритмов функций

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Порядок чисел, приведенных в табл. 2.1, имеет чрезвычайное значение для анализа алгоритмов. Как видно из таблицы, самый малый порядок роста имеет логарифмическая функция. Причем его значение настолько мало, что программы, реализующие алгоритмы с логарифмическим количеством основных операций, будут выполняться практически мгновенно для всех диапазонов входных данных реального размера. Обратите также внимание, что, хотя некоторые значения при таких вычислениях, естественно, будут зависеть от основания логарифма, приведенная ниже формула позволяет легко переходить от одного основания логарифма к другому, сохраняя при этом логарифмическую зависимость вычислений (при этом используются новые постоянные множители):

$$\log_a n = \log_a b \ \log_b n.$$

Вот почему в случае, когда нужно только определить порядок роста количества основных операций алгоритма с точностью до постоянного множителя, мы будем опускать основание логарифма и записывать просто: $\log n$.

Существует и другая крайность: показательная функция 2^n и функция вычисления факториала $n!$. Обе эти функции имеют настолько высокий порядок роста, что его значение становится астрономически большим уже при умеренных значениях n. (По этой причине мы не стали приводить в табл. 2.1 значения порядка роста этих функций при $n > 10^2$.) Например, чтобы выполнить 2^{100} операций компьютеру, имеющему производительность в один триллион операций (10^{12}) в секунду, понадобиться без малого $4 \cdot 10^{10}$ лет! Однако это ничто по сравнению со временем, которое затратит тот же компьютер на выполнение 100! операций.

Его даже нельзя сравнить со временем жизни планеты Земля, которое, по приблизительным оценкам, составляет 4.5 миллиарда ($4.5 \cdot 10^9$) лет. Несмотря на существенную разницу между порядком роста показательной функции 2^n и функции $n!$, довольно часто говорят, что обе функции имеют экспоненциальный порядок роста. Однако, строго говоря, такое утверждение верно только для первой из этих функций. Подводя итог, можно сформулировать следующий вывод, который всегда важно помнить.

> С помощью алгоритмов, в которых количество выполняемых операций растет по экспоненциальному закону, можно решить лишь задачи очень малых размеров.

Существует еще один способ оценки качественного различия в порядке роста функций, приведенном в табл. 2.1. Необходимо рассмотреть реакцию функции на, скажем, двукратное увеличение значения параметра n. Для функции $\log_2 n$ это приведет к увеличению значения всего на 1, так как $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$. Линейная функция увеличит значение в два раза. Значение функции $n \log_2 n$ увеличится чуть больше, чем в два раза. Квадратичная n^2 и кубическая n^3 функции увеличатся в четыре и восемь раз, соответственно, поскольку $(2n)^2 = 4n^2$ и $(2n)^3 = 8n$. Значение функции 2^n увеличится в квадрате, поскольку $2^{2n} = (2^n)^2$. Что касается функции $n!$, то ее значение увеличится намного больше, чем значение любой из рассмотренных здесь функций.

Эффективность алгоритма в разных случаях

В начале этого раздела мы определили, что имеет смысл оценивать эффективность алгоритма как функцию от некоторого параметра n, связанного с размером входных данных. Однако существует большое количество алгоритмов, время выполнения которых зависит не только от размера входных данных, но также и от особенностей конкретных входных данных. В качестве примера рассмотрим задачу последовательного поиска. Она решается с помощью довольно простого алгоритма, который выполняет поиск заданного элемента (ключа поиска K) в списке, состоящем из n элементов, путем последовательного сравнения ключа K с каждым из элементов списка. Работа алгоритма завершается, либо когда заданный ключ найден, либо когда весь список исчерпан. Ниже этот алгоритм описан на псевдокоде. В нем для простоты полагается, что список задан в виде массива чисел. (Кроме того, предполагается, что второе условие $A[i] \neq K$ не будет проверяться в случае, если не выполняется первое условие, в котором проверяется, не выходит ли индекс массива за его верхнюю границу.)

АЛГОРИТМ $SequentialSearch\,(A\,[0..n-1]\,,K)$

```
// Входные данные:  массив чисел A[0..n − 1] и ключ поиска K
// Выходные данные: возвращается индекс первого элемента
//                  массива A, который равен K, либо −1,
//                  если заданный элемент не найден
i ← 0
while i < n and A[i] ≠ K do
  i ← i + 1
if i < n
  return i
else
  return −1
```

Совершенно очевидно, что время работы этого алгоритма может отличаться в очень широких пределах для одного и того же списка размера n. В наихудшем случае, т.е. когда в списке нет искомого элемента либо когда искомый элемент расположен в списке последним, в алгоритме будет выполнено наибольшее количество операций сравнения ключа со всеми n элементами списка: $C_{worst}\,(n) = n$.

Под ***эффективностью*** алгоритма ***в наихудшем случае*** (***worst-case efficiency***) подразумевают его эффективность для наихудшей совокупности входных данных размером n, т.е. для такой совокупности входных данных размером n среди всех возможных, для которой время работы алгоритма будет наибольшим. Способ определения эффективности алгоритма для наихудшего случая в принципе несложен: нужно проанализировать алгоритм и выяснить, для каких из возможных комбинаций входных данных размером n значение числа основных операций алгоритма $C\,(n)$ будет максимальным, а затем вычислить значение $C_{worst}\,(n)$ для наихудшего случая. (Что касается алгоритма последовательного поиска, то здесь ответ очевиден. Методы исследования более сложных ситуаций будут описаны в последующих разделах этой главы.) Очевидно, что анализ эффективности алгоритма для наихудшего случая позволяет получить очень важную информацию о быстродействии алгоритма в целом, поскольку в данном случае речь идет о максимально возможном времени его выполнения. Другими словами, он гарантирует, что для любых исходных данных размером n время выполнения алгоритма не будет превышать максимально возможного значения $C_{worst}\,(n)$, получаемого для наихудшей совокупности входных данных.

Под ***эффективностью*** алгоритма ***в наилучшем случае*** (***best-case efficiency***) подразумевают его эффективность для наилучшей совокупности входных данных размером n, т.е. для такой совокупности входных данных размером n среди всех возможных, для которой время работы алгоритма будет наименьшим. Соответственно, эффективность алгоритма для наилучшего случая можно проанализировать следующим образом. Сначала нужно определить, для каких из возможных

комбинаций входных данных размером n значение числа основных операций алгоритма $C(n)$ будет наименьшим. (Обратите внимание, что наилучший случай вовсе не означает случай, когда вводится минимальное количество данных. Он означает комбинацию входных данных размером n, при обработке которых алгоритм будет работать быстрее всего.). Затем мы должны определить само значение $C_{best}(n)$ для такого случая. Например, для алгоритма последовательного поиска наилучшим случаем входных данных будет такой список размером n, первый элемент которого равен ключу поиска K. Поэтому для него $C_{best}(n) = 1$.

Анализ эффективности алгоритма для наилучшего случая не так важен, как для наихудшего случая, хотя его нельзя назвать совсем уж бесполезным. При анализе алгоритма не стоит полагаться на то, что обрабатываемые им данные будут соответствовать наилучшему случаю. Тем не менее нужно учитывать тот факт, что для некоторых алгоритмов их высокое быстродействие для наилучшего случая сохраняется и для случаев близких к наилучшему. Например, существует алгоритм сортировки методом вставок, для которого наилучший случай входных данных соответствует заранее отсортированному массиву. При этом скорость работы такого алгоритма будет наибольшей. Причем она лишь ненамного ухудшается в случае, если входной массив почти отсортирован. Следовательно, такой алгоритм очень хорошо подойдет для случаев, когда приходится иметь дело с почти отсортированными массивами. Ну и конечно же, если эффективность алгоритма для наилучшего случая является неудовлетворительной, не имеет смысла продолжать его дальнейший анализ.

Как бы там ни было, из данного описания уже должно быть понятно, что на основании информации, полученной в результате анализа алгоритма для наилучшего и наихудшего случаев, нельзя сделать вывод о том, как поведет себя алгоритм при обработке типовых или случайно заданных входных данных. Чтобы получить подобную информацию, нужно выполнить анализ алгоритма для ***среднего случая*** (***average-case efficiency***). Для этого нужно сделать ряд предположений о совокупности входных данных размером n.

Давайте продолжим рассмотрение алгоритма поиска методом последовательного перебора. Сделаем два стандартных предположения:

1. вероятность успешного поиска равна p, где $0 \leqslant p \leqslant 1$;
2. вероятность первого совпадения ключа с i-м элементом списка одинакова для любого i.

Исходя из этих предположений (несмотря на всю очевидность, их справедливость обычно трудно доказать), можно вычислить среднее количество операций сравнения с ключом $C_{avg}(n)$, как описано ниже. Если совпадение с ключом найдено, вероятность того, что это произошло именно на i-м элементе списка, равна p/n для любого i. При этом количество операций сравнения, выполняемых в алгоритме в подобном случае, очевидно и равно i. Если совпадение с ключом не

найдено, количество операций сравнения равно n, а вероятность этого события равна $(1-p)$. Поэтому

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1-p) =$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots n] + n \cdot (1-p) =$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n \cdot (1-p) = \frac{p(n+1)}{2} + n \cdot (1-p).$$

Эта обобщенная формула позволяет ответить на несколько вполне уместных вопросов. Например, если $p = 1$ (т.е. искомый элемент, равный ключу, точно присутствует в списке и поэтому будет найден), среднее количество операций сравнения при поиске методом последовательного перебора будет равно $(n+1)/2$. Другими словами, в среднем, в алгоритме проверяется приблизительно половина элементов списка. Если $p = 0$ (т.е. искомого элемента в списке нет), среднее количество операций сравнения будет равно n, поскольку при этом в алгоритме проверяются все n элементов списка.

Как видно из этого очень простого примера, определение эффективности алгоритма для среднего случая куда более трудная задача, чем для наихудшего и наилучшего случаев. При использовании прямого метода для решения этой задачи, необходимо разбить всю совокупность экземпляров входных данных размером n на несколько групп так, чтобы для каждой группы число основных операций, выполняемых алгоритмом, было одинаково. (Сможете ли вы выделить эти группы для алгоритма поиска методом последовательного перебора?) Затем нужно найти или придумать функцию распределения вероятностей для входных данных и с ее помощью вычислить ожидаемое значение количества основных операций алгоритма. Технически реализовать этот план не составит большого труда, хотя проверить вероятностные предпосылки, лежащие в его основе, обычно очень тяжело. Поэтому для простоты, когда речь будет заходить об анализе эффективности для среднего случая, мы чаще всего будем приводить уже готовые результаты для обсуждаемого алгоритма. Если вас заинтересует источник этих результатов, рекомендуем обратиться к таким книгам, как [8, 65, 66, 67] и [105].

Нужны ли реально кому-нибудь данные об эффективности алгоритма для среднего случая? Ответ на этот вопрос не вызывает сомнений — конечно же да! Существует большое число важных алгоритмов, эффективность которых гораздо выше для среднего случая, чем для излишне пессимистичного наихудшего случая, что привлекает к ним практический интерес. Таким образом, информация об эффективности алгоритма для среднего случая позволяет принять правильное решение и не упустить много важных алгоритмов. Наконец, из приведенного выше описания уже должно быть понятно, что эффективность алгоритма для среднего случая нельзя получить, усреднив его эффективности для наихудшего и наилучшего случаев. Даже если иногда полученные таким образом результаты совпадут

с истинными данными для среднего случая, такой способ анализа не является законным.

Существует еще один вид эффективности, называемый ***амортизированной эффективностью*** (***amortized efficiency***). Она предназначена не столько для анализа общего времени выполнения алгоритма, сколько для анализа последовательности операций, выполняемых над одной и той же структурой данных. Бывает так, что одна из операций алгоритма может оказаться слишком продолжительной, но общее время выполнения последовательности из n таких операций будет значительно меньше, чем его оценка, выполненная для наихудшего случая, равная произведению максимального времени выполнения этой операции на число таких операций, n. Поэтому мы можем разделить (или "амортизировать") высокую стоимость выполнения алгоритма для наихудшего случая на всю последовательность выполняемых операций по аналогии с тем, как в экономике разносят стоимость дорогостоящего оборудования на весь период его эксплуатации. Этот нетривиальный подход был предложен американским кибернетиком Робертом Таржаном (Robert Tarjan). Он использовал его наряду с другими методами для исследования интересного варианта классического двоичного дерева поиска. (Очень интересное описание этого метода, лишенное технических деталей, приведено в статье Таржана [117], а в статье [116] этот же метод рассмотрен со всеми подробностями.) Полезный пример применения амортизированной эффективности будет приведен в разделе 9.2, где мы будем рассматривать алгоритмы объединения непересекающихся множеств.

Повторение пройденного

Давайте в заключение этого раздела еще раз отметим основные моменты, на которых мы останавливались ранее.

- И временна́я, и пространственная эффективности оцениваются в виде функции от некоторого параметра n, связанного с размером входных данных.
- Временна́я эффективность измеряется путем подсчета числа основных операций, выполняемых в алгоритме. Пространственная эффективность оценивается в виде количества дополнительных единиц памяти, требуемых для работы алгоритма.
- Эффективность отдельных алгоритмов может различаться в очень широких пределах для разных наборов входных данных одинакового размера. Поэтому эффективность таких алгоритмов оценивают для наихудшего, наилучшего и типичного набора входных данных.
- Основной целью анализа эффективности алгоритмов является определение порядка роста времени выполнения алгоритма (а также количе-

ства дополнительных единиц памяти) в случае, когда размер входных данных стремится к бесконечности.

В следующем разделе будут рассмотрены строгие методы определения порядка роста. В разделах 2.3 и 2.4 мы обсудим особенности методов анализа нерекурсивных и рекурсивных алгоритмов. Мы увидим, как применить на практике полученные в этом разделе базовые знания для анализа эффективности конкретных алгоритмов. Остальные примеры будут приведены в других главах книги.

Упражнения 2.1

1. Для каждого из приведенных ниже алгоритмов определите: 1) естественные единицы измерения размера входных данных; 2) основную операцию алгоритма; 3) будет ли изменяться количество основных операций, выполняемых алгоритмом, в зависимости от используемого набора входных данных одинакового размера.

а) Вычисление суммы n чисел.

б) Вычисление $n!$.

в) Поиск наибольшего элемента в списке из n чисел.

г) Алгоритм Евклида.

д) Решето Эратосфена.

е) Алгоритм умножения в столбик двух n-разрядных десятичных целых чисел.

2. а) Рассмотрите алгоритм сложения двух матриц размером $n \times n$, основанный на определении сложения матриц. Назовите его основную операцию. Определите зависимость количества основных операций как функцию от порядка матрицы n. Найдите зависимость количества основных операций как функцию от числа элементов в матрицах.

б) Выполните аналогичное упражнение для алгоритма умножения двух матриц размером $n \times n$, соответствующего определению этой операции.

3. Рассмотрите вариант алгоритма поиска методом последовательного перебора, который бы возвращал количество элементов в списке, совпадающих с заданным ключом поиска. Будет ли его эффективность отличаться от эффективности классического алгоритма поиска методом последовательного перебора?

4. а) В ящике хранится 22 перчатки: 5 пар красных, 4 пары желтых и 2 пары зеленых. Предположим, что вы выбираете их в темноте наугад

и можете проверить, что именно вы выбрали, только после того, как выбор сделан. Чему равно минимальное количество перчаток, которые надо взять из ящика, чтобы получить как минимум одну пару перчаток одинакового цвета? Дайте ответ на этот вопрос для наилучшего и наихудшего случая. (Задача взята из [80], №18.)

б) Предположим, у вас есть 5 разных пар носков, и вы обнаружили, что два носка потеряны. Естественно, вы заинтересованы в том, чтобы в результате осталось наибольшее количество полных пар, поскольку носить разные носки вы не будете. Поэтому в лучшем случае у вас останется 4 полных пары, а в худшем — только 3. Предположим, вы имеете равные шансы потерять любой из 10 носков. Определите вероятности наилучшего и наихудшего случаев, а также количество пар носков, которые остаются у вас в среднем случае. (Задача взята из [80], №48.)

5. а) Докажите формулу (2.1), определяющую количество битов в двоичном представлении положительного десятичного целого числа.

б) Приведите аналогичную формулу для количества десятичных цифр.

в) Объясните, почему в рамках принятой нами модели основных понятий, не имеет значения, какие цифры (двоичные или десятичные) мы используем для указания размера n входных данных.

6. Как, по вашему мнению, следует дополнить любой из алгоритмов сортировки, чтобы количество операций сравнения его ключей в лучшем случае составляло $n - 1$, где n, как обычно, размер списка. Как вы думаете, будет ли подобное дополнение уместно для любого алгоритма сортировки?

7. Для решения системы из n линейных уравнений, где n — произвольное целое число, используется классический алгоритм последовательного исключения Гаусса, в котором выполняется $n^3/3$ умножений, являющихся основной операцией алгоритма.

а) Определите, во сколько раз дольше будет работать алгоритм Гаусса для решения системы из 1000 уравнений по сравнению со временем решения системы из 500 уравнений.

б) Предположим, вы собираетесь приобрести компьютер, быстродействие которого превосходит в 1000 раз быстродействие того компьютера, на котором вы сейчас работаете. Определите, на сколько порядков большую систему уравнений сможет решить за одно и то же время новый компьютер по сравнению со старым.

8. Для каждой из приведенных ниже функций определите, во сколько раз изменится ее значение при увеличении значения аргумента в четыре раза.

   а) $\log_2 n$ **б)** $\sqrt{n}$ **в)** n **г)** n^2

   д) n^3 **е)** 2^n

9. Для каждой из приведенных ниже пар функций определите, соответствует ли первая функция порядку роста второй функции (с точностью до постоянного множителя), меньше его или больше.

   а) $n(n+1)$ и $2000n^2$ **б)** $100n^2$ и $0.01n^3$

   в) $\log_2 n$ и $\ln n$ **г)** $\log_2^2 n$ и $\log_2 n^2$

   д) 2^{n-1} и 2^n **е)** $(n-1)!$ и $n!$

10. Согласно древней легенде, игру в шахматы придумал много столетий тому назад мудрец, проживавший в северо-западной Индии. Он показал эту игру своему повелителю. Ему она понравилась настолько, что он предложил мудрецу любую награду, которую тот только пожелает. Мудрец попросил повелителя выдать ему в качестве награды несколько хлебных зерен, количество которых должно определяться по следующему правилу. На первую клетку шахматной доски кладется одно зернышко, на вторую — два, на третью — четыре, на четвертую — восемь, на пятую — 32 и так до тех пор, пока все 64 клетки доски не будут заполнены. Какое окончательное количество зерен должен был получить мудрец согласно описанному алгоритму?

2.2 Асимптотические обозначения и основные классы эффективности

Как отмечалось в предыдущем разделе, при изложении основ анализа эффективности алгоритмов основное внимание было сосредоточено на порядке роста количества базовых операций алгоритма, который мы использовали в качестве основного критерия эффективности алгоритма. Для того чтобы можно было сравнивать между собой эти порядки роста и классифицировать их, ученые ввели три условных обозначения: O (прописная "О"), Θ (прописная греческая "тета") и Ω (прописная греческая "омега"). Для начала дадим нестрогое описание этих понятий, а затем, рассмотрев несколько примеров, приведем их строгое определение. В приведенном ниже описании $t(n)$ и $g(n)$ могут быть любыми неотрицательными функциями, определенными на множестве натуральных чисел. Через $t(n)$ мы обозначили время выполнения алгоритма. Обычно оно выражается в виде числа основных операций алгоритма, обозначаемых через $C(n)$. Под $g(n)$ мы будем

понимать некоторую простую функцию, с которой будет проводиться сравнение количества операций.

Нестрогое введение

Говоря нестрого, обозначение $O(g(n))$ — это множество всех функций, порядок роста которых при достаточно больших n не превышает (т.е. меньше или равен) некоторую константу, умноженную на значение функции $g(n)$. Вот несколько примеров — все приведенные ниже утверждения справедливы:

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad n(n-1)/2 \in O(n^2).$$

В самом деле, первые две функции линейны, поэтому их порядок роста заведомо меньше, чем $g(n) = n^2$. Последняя функция является квадратичной, поэтому ее порядок роста такой же, как у функции n^2. С другой стороны,

$$n^3 \notin O(n^2), \quad 0.00001n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

В самом деле, обе функции n^3 и $0.00001n^3$ являются кубическими, поэтому имеют более высокий порядок роста, чем функция n^2. Те же рассуждения справедливы и для полинома четвертой степени $n^4 + n + 1$.

Второе обозначение, $\Omega(g(n))$, — это множество всех функций, порядок роста которых при достаточно больших n не меньше (т.е. больше или равен) некоторой константы, умноженной на значение функции $g(n)$. Например:

$$n^3 \in \Omega(n^2), \quad n(n-1)/2 \in \Omega(n^2), \quad \text{но } 100n + 5 \notin \Omega(n^2).$$

Наконец, обозначение $\Theta(g(n))$ — это множество всех функций, порядок роста которых при достаточно больших n равен некоторой константе, умноженной на значение функции $g(n)$. Таким образом, любая квадратичная функция $an^2 + bn + c$ при $a > 0$ принадлежит множеству $\Theta(n^2)$. Кроме того, среди бесконечного количества других функций множеству $\Theta(n^2)$ принадлежат также функции $n^2 + \sin n$ и $n^2 + \log n$ (можете объяснить, почему?).

Надеюсь, что приведенное выше нестрогое описание позволит вам глубже понять идею, лежащую в основе трех асимптотических обозначений. А теперь перейдем к строгим определениям.

O-обозначение

Определение 1. Говорят, что функция $t(n)$ принадлежит множеству $O(g(n))$, что записывается как $t(n) \in O(g(n))$, если для всех больших значений n функция $t(n)$ ограничена сверху некоторой константой, умноженной на значение функции $g(n)$, т.е. если существует положительная константа c и неотрицательное целое число n_0 такое, что $t(n) \leqslant cg(n)$ для всех $n \geqslant n_0$. ■

Это определение проиллюстрировано на рис. 2.1. Чтобы сделать рисунок нагляднее, мы использовали вещественный тип числа n.

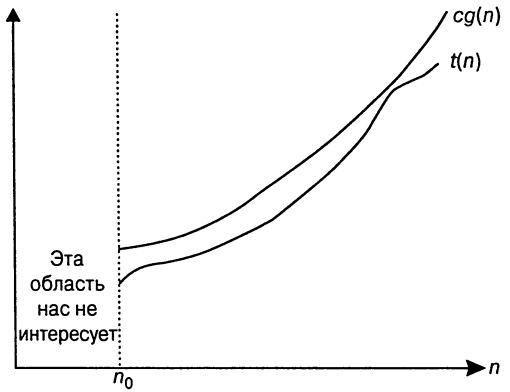


Рис. 2.1. Иллюстрация O-обозначения: $t(n) \in O(g(n))$

В качестве примера давайте приведем строгое доказательство одного из утверждений, приведенных во введении к этой главе: $100n + 5 \in O(n^2)$. В самом деле, для всех $n \geqslant 5$ справедливо следующее выражение:

$$100n + 5 \leqslant 100n + n = 101n \leqslant 101n^2.$$

Таким образом, для данного случая в качестве констант c и n_0, о которых говорится в приведенном выше определении, можно, соответственно, взять 101 и 5.

Заметьте, что в определении ничего не говорится о том, какие именно значения констант c и n_0 мы должны выбирать. Поэтому приведенный выше пример можно доказать и так: для всех $n \geqslant 1$ справедливо выражение

$$100n + 5 \leqslant 100n + 5n = 105n.$$

Из него следует, что в данном случае $c = 105$, а $n_0 = 1$.

Ω-обозначение

Определение 2. Говорят, что функция $t(n)$ принадлежит множеству $\Omega(g(n))$, что записывается как $t(n) \in \Omega(g(n))$, если для всех больших значений n функция $t(n)$ ограничена снизу некоторой константой, умноженной на значение функции $g(n)$, т.е. если существует положительная константа c и неотрицательное целое число n_0, такое, что $t(n) \geqslant cg(n)$ для всех $n \geqslant n_0$. ■

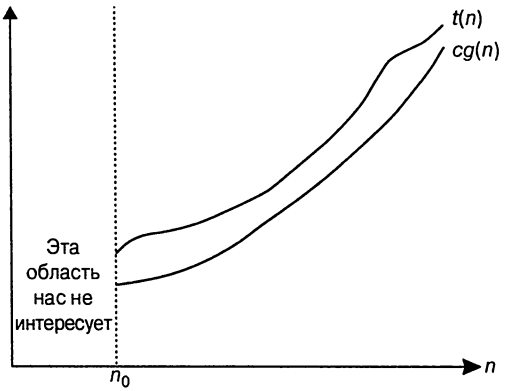


Рис. 2.2. Иллюстрация Ω-обозначения: $t(n) \in \Omega(g(n))$

Это определение проиллюстрировано на рис. 2.2.

В качестве примера приведем строгое доказательство того, что $n^3 \in \Omega(n^2)$. В самом деле, для всех $n \geqslant 0$ выполняется неравенство $n^3 \geqslant n^2$. Таким образом, для данного случая можно положить $c = 1$ и $n_0 = 0$.

Θ-обозначение

Определение 3. Говорят, что функция $t(n)$ принадлежит множеству $\Theta(g(n))$, что записывается как $t(n) \in \Theta(g(n))$, если для всех больших значений n функция $t(n)$ ограничена снизу и сверху некоторыми константами, умноженными на значения функции $g(n)$, т.е. если существуют положительные константы c_1 и c_2, а также неотрицательное целое число n_0 такое, что $c_2 g(n) \leqslant t(n) \leqslant c_1 g(n)$ для всех $n \geqslant n_0$. ■

Это определение проиллюстрировано на рис. 2.3.

В качестве примера приведем строгое доказательство того, что $n(n-1)/2 \in \in \Theta(n^2)$. Сначала, давайте докажем правую часть неравенства (т.е. определим верхнюю границу). Для всех $n \geqslant 0$ справедливо следующее:

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leqslant \frac{1}{2}n^2.$$

Затем докажем левую часть неравенства (т.е. определим нижнюю границу). Для всех $n \geqslant 2$ справедливо следующее:

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geqslant \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n = \frac{1}{4}n^2.$$

Таким образом, для данного случая можно положить $c_2 = 1/4$, $c_1 = 1/2$, $n_0 = 2$.

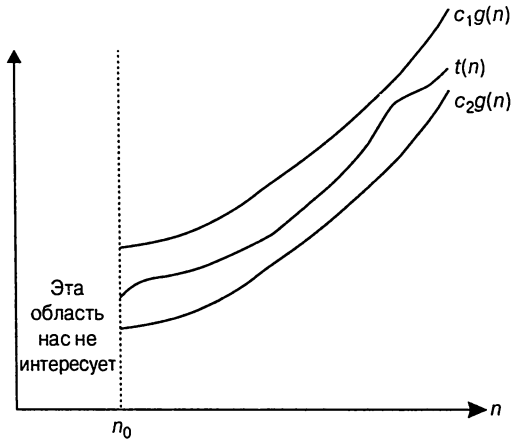


Рис. 2.3. Иллюстрация Θ-обозначения: $t(n) \in \Theta(g(n))$

Полезные свойства сделанных асимптотических обозначений

С помощью сделанных выше строгих определений асимптотических обозначений можно доказать ряд их основных свойств. (Несколько простых примеров были приведены в задаче 1 упражнения 2.1). В частности, приведенные ниже свойства пригодятся при анализе алгоритмов, состоящих из двух исполняемых частей.

Теорема 1. Если $t_1(n) \in O(g_1(n))$ и $t_2(n) \in O(g_2(n))$, то $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$. (Аналогичные утверждения справедливы также для множеств Ω и Θ).

Доказательство. (Как мы увидим, доказательство этой теоремы представляет собой расширение на порядок роста следующего простого соотношения для произвольных вещественных чисел a_1, b_1, a_2 и b_2: если $a_1 \leqslant b_1$ и $a_2 \leqslant b_2$, то $a_1 + a_2 \leqslant 2\max\{b_1, b_2\}$.) Поскольку $t_1(n) \in O(g_1(n))$, существует константа c_1 и неотрицательное целое число n_1 такие, что для всех $n \geqslant n_1$ справедливо $t_1(n) \leqslant c_1 g_1(n)$.

По аналогии, поскольку $t_2(n) \in O(g_2(n))$, существует константа c_2 и неотрицательное целое число n_2 такие, что для всех $n \geqslant n_2$ справедливо $t_2(n) \leqslant c_2 g_2(n)$.

Давайте обозначим $c_3 = \max\{c_1, c_2\}$ и рассмотрим $n \geqslant \max\{n_1, n_2\}$, т.е. когда верны оба неравенства. Сложив приведенные выше неравенства, получим

следующее:

$$t_1(n) + t_2(n) \leqslant c_1 g_1(n) + c_2 g_2(n) \leqslant$$
$$\leqslant c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)] \leqslant$$
$$\leqslant c_3 2 \max\{g_1(n), g_2(n)\}.$$

Отсюда следует, что $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$. Исходя из определения множества O, в качестве констант c и n_0 в данном случае нужно выбрать: $c = 2c_3 = 2\max\{c_1, c_2\}$ и $n_0 = \max\{n_1, n_2\}$. ■

Итак, что же означает это свойство с точки зрения анализа эффективности алгоритмов, состоящих из двух исполняемых частей? Оно означает, что общая эффективность алгоритма зависит от той части, которая имеет наибольший порядок роста, т.е. от наименее эффективной его части:

$$\left.\begin{array}{|l|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Например, проверить, содержит ли массив повторяющиеся элементы, можно с помощью описанного ниже двухэтапного алгоритма. Сначала нужно отсортировать массив, воспользовавшись одним из известных алгоритмов сортировки. Затем нужно пройтись по всем элементам отсортированного массива и проверить, не дублируют ли соседние элементы друг друга. Если, к примеру, в алгоритме сортировки, используемом на первом этапе, выполняется не более чем $n(n-1)/2$ операций сравнения (и поэтому он принадлежит множеству $O(n^2)$), а в алгоритме, используемом на втором этапе, выполняется не более чем $n-1$ операций сравнения (т.е. он принадлежит множеству $O(n)$), суммарная эффективность общего алгоритма будет принадлежать множеству $O(\max\{n, n^2\}) = O(n^2)$.

Использование пределов для сравнения порядка роста двух функций

Несмотря на то что без строгих определений множеств O, Ω, Θ нельзя обойтись при доказательстве их абстрактных свойств, они редко используются при сравнении порядков роста двух конкретных функций. Дело в том, что существует более удобный метод выполнения этой оценки, основанный на вычислении предела отношения двух рассматриваемых функций. Может существовать три

основных случая[3]:

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{если } t(n) \text{ имеет меньший порядок роста, чем } g(n)\\ c & \text{если } t(n) \text{ имеет тот же порядок роста, чем } g(n)\\ \infty & \text{если } t(n) \text{ имеет больший порядок роста, чем } g(n).\end{cases}$$

Обратите внимание, что для двух первых случаев $t(n)\in O(g(n))$, для двух последних случаев $t(n)\in\Omega(g(n))$, а для второго случая $t(n)\in\Theta(g(n))$.

Методы, основанные на вычислении пределов, зачастую более удобны для анализа алгоритмов, чем описанные выше методы, основанные на определении множеств. Дело в том, что в первом случае можно воспользоваться преимуществами мощного математического аппарата, специально созданного для вычисления пределов, в частности правилом Лопиталя (L'Hôpital)

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

и формулой Стирлинга (Stirling):

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ для достаточно больших } n.$$

Ниже приведены примеры использования пределов для сравнения порядка роста двух функций.

Пример 1. Сравните порядки роста функций $n(n-1)/2$ и n^2. (Это один из тех примеров, который мы использовали выше для иллюстрации определений множеств.)

$$\lim_{n\to\infty}\frac{n(n-1)/2}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)=\frac{1}{2}.$$

Поскольку предел равен положительной константе, обе функции имеют одинаковый порядок роста, что записывается как $n(n-1)/2\in\Theta(n^2)$. ■

Пример 2. Сравните порядки роста функций $\log_2 n$ и $\sqrt{n}$. (В отличие от примера 1 решение данной задачи не столь очевидно.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}=0.$$

[3]Есть и четвертый случай, когда предела попросту не существует, однако он встречается довольно редко в реальной практике анализа алгоритмов. Тем не менее эта особенность приводит к тому, что подход к оценке порядка роста, основанный на вычислении предела отношения двух рассматриваемых функций, является менее универсальным, чем тот, в котором используется определения множеств O, Ω, Θ.

Поскольку предел равен нулю, функция $\log_2 n$ имеет меньший порядок роста, чем $\sqrt{n}$. (Так как $\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = 0$, то для этого случая можно использовать так называемую форму записи "с маленькой о": $\log_2 n \in o(\sqrt{n})$. В отличие от прописной греческой "O" при анализе алгоритмов строчная "o" используется довольно редко.) ■

Пример 3. Сравните порядки роста функций $n!$ и 2^n. (С этим примером мы уже встречались в предыдущем разделе.) Воспользовавшись формулой Стирлинга, получим:

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty.$$

Следовательно, хотя функция 2^n растет очень быстро, функция $n!$ растет еще быстрее, что записывается так: $n! \in \Omega(2^n)$. Обратите внимание, что при использовании этой формы записи читатель, в отличие от случая использования пределов, не может сделать однозначный вывод о том, что порядок роста функции $n!$ выше, чем функции 2^n, а не, скажем, равен ей. ■

Основные классы эффективности

Хотя в основах анализа алгоритмов к одному классу относят все функции, чей порядок роста одинаков с точностью до постоянного множителя, существует бесконечное множество подобных классов. Например, порядок роста показательной функции a^n зависит от значения основания a. Поэтому для вас может оказаться неожиданностью, что временну́ю эффективность большого количества алгоритмов можно отнести всего к нескольким классам. Эти классы, их названия, а также некоторые пояснения, приведены в табл. 2.2 в соответствии с возрастанием их порядка роста.

Таблица 2.2. Основные асимптотические классы эффективности

Класс	Название	Примечание
1	*Константа*	Если не считать эффективности в наилучшем случае, в этот класс попадает очень небольшое количество алгоритмов. Обычно при бесконечном увеличении размера входных данных время выполнения алгоритма также стремится к бесконечности

Окончание табл. 2.2

Класс	Название	Примечание
$\log n$	*Логарифмическая*	Обычно такая зависимость появляется в результате сокращения размера задачи на постоянное значение на каждом шаге итерации алгоритма (см. раздел 5.5). Обратите внимание, что в логарифмическом алгоритме невозможна работа со всеми входными данными (и даже с их некоторой фиксированной частью). В этом случае время выполнения любого алгоритма будет находиться по меньшей мере в линейной зависимости от размера входных данных
n	*Линейная*	К этому классу относятся алгоритмы, выполняющие сканирование списка, состоящего из n элементов, например алгоритм поиска методом последовательного перебора
$n \log n$	*n-log-n*	К этому классу относятся большое количество алгоритмов декомпозиции (см. главу 4), таких как алгоритмы сортировки слиянием и быстрой сортировки
n^2	*Квадратичная*	Как правило, подобная зависимость характеризует эффективность алгоритмов, содержащих два встроенных цикла (см. следующий раздел). В качестве типичных примеров достаточно назвать простой алгоритм сортировки и целый ряд операций, выполняемых над матрицами размером $n \times n$
n^3	*Кубическая*	Как правило, подобная зависимость характеризует эффективность алгоритмов, содержащих три встроенных цикла (см. следующий раздел). К этому классу относятся несколько довольно сложных алгоритмов линейной алгебры
2^n	*Экспоненциальная*	Данная зависимость типична для алгоритмов, выполняющих обработку всех подмножеств некоторого множества, состоящего из n элементов. Часто термин "экспоненциальный" используется в широком смысле и означает очень высокие порядки роста, т.е. включает более быстрые по сравнению с экспонентой порядки роста
$n!$	*Факториальная*	Данная зависимость типична для алгоритмов, выполняющих обработку всех перестановок некоторого множества, состоящего из n элементов

Вы можете возразить, что классификация алгоритмов в соответствии с основными асимптотическими классами эффективности имеет небольшое практическое применение, поскольку обычно значения постоянных множителей никто не указывает. А это означает, что не исключена возможность, когда для входных дан-

ных реального размера, алгоритм, относящийся к худшему классу эффективности, будет работать быстрее, чем алгоритм, относящийся к лучшему классу эффективности. Например, если время выполнения одного алгоритма изменяется по закону n^3, а другого — по закону $10^6 n^2$, кубический алгоритм будет работать быстрее при условии, что n не превышает 10^6. Известно несколько подобных аномалий. Например, существуют алгоритмы перемножения матриц, имеющих лучшую асимптотическую эффективность, чем у алгоритма, построенного на определении этой операции (см. раздел 4.5) и имеющего кубическую зависимость эффективности. Однако поскольку для таких алгоритмов значения постоянных множителей намного большие, они могут представлять собой лишь теоретическую ценность.

К счастью, обычно значения постоянных множителей не отличаются настолько сильно. Поэтому, как правило, можно предполагать, что алгоритм, относящийся к лучшему асимптотическому классу, будет работать быстрее алгоритма, относящегося к худшему асимптотическому классу, даже при обработке входных данных среднего размера. Это замечание главным образом справедливо для тех алгоритмов, время выполнения которых растет не так быстро, как у экспоненциальных или факториальных алгоритмов.

Упражнения 2.2

1. Выберите подходящее обозначение O, Θ или Ω, которое бы наиболее точно указывало на принадлежность к конкретному классу временно́й эффективности алгоритма поиска методом последовательного перебора (см. раздел 2.1):
   а) для худшего случая;
   б) для лучшего случая;
   в) для среднего случая.
2. Используя нестрогие определения множеств O, Θ и Ω, определите, истинны или ложны перечисленные ниже утверждения.
   а) $\frac{n(n+1)}{2} \in O(n^3)$ **б)** $\frac{n(n+1)}{2} \in O(n^2)$
   в) $\frac{n(n+1)}{2} \in \Theta(n^3)$ **г)** $\frac{n(n+1)}{2} \in \Omega(n)$
3. Для каждой из приведенных ниже функций укажите класс $\Theta(g(n))$, к которому относится функция. (При ответе используйте максимально простую функцию $g(n)$). Докажите ваши утверждения.
   а) $(n^2+1)^{10}$ **б)** $\sqrt{10n^2+7n+3}$
   в) $2n \lg (n+2)^2 + (n+2)^2 \lg (n/2)$
   г) $2^{n+1} + 3^{n-1}$ **д)** $\lfloor \log_2 n \rfloor$

4. а) В табл. 2.1 указаны значения функций, играющих особую роль в процессе анализа алгоритмов. Их анализ делает очевидным, что функции $\log n$, n, $n \log n$, n^2, n^3, 2^n, $n!$ перечислены согласно возрастанию их порядка роста. Можно ли считать эти значения строгим математическим доказательством данного факта?

б) Докажите, что приведенный выше список функций в самом деле упорядочен в соответствии с их порядком роста.

5. Расположите перечисленные ниже функции в соответствии с их порядком роста (от самого меньшего к самому большему).

$$(n-2)!,\ 5 \lg (n+100)^{10},\ 2^{2^n},\ 0.001n^4 + 3n^3 + 1,\ \ln^2 n,\ \sqrt[3]{n},\ 3^n$$

6. а) Докажите, что любой многочлен вида $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, где $a_k > 0$, принадлежит множеству $\Theta(n^k)$.

б) Докажите, что порядок роста показательной функции a^n зависит от значения ее неотрицательного основания a.

7. Используя определения рассматриваемых множеств, докажите или опровергните, приведя соответствующий контрпример, следующие утверждения:

а) Если $t(n) \in O(g(n))$, то $g(n) \in \Omega(t(n))$

б) $\Omega(\alpha g(n)) = \Omega(g(n))$, где $\alpha > 0$

в) $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

г) Для любых двух неотрицательных функций $t(n)$ и $g(n)$, определенных на множестве неотрицательных целых чисел, справедливо либо выражение $t(n) \in O(g(n))$, либо выражение $t(n) \in \Omega(g(n))$, либо оба эти выражения одновременно.

8. Докажите рассматриваемую в этом разделе теорему для:

а) Ω-обозначений; **б)** Θ-обозначений.

9. В этом разделе было сказано, что проверить, содержит ли массив повторяющиеся элементы, можно с помощью двухэтапного алгоритма, основанного на предварительной сортировке самого массива.

а) Если известно, что сортировка массива выполняется с помощью алгоритма, временна́я эффективность которого $\Theta(n \log n)$, к какому классу временно́й эффективности будет относиться двухэтапный алгоритм в целом?

б) Если для работы алгоритма сортировки требуется дополнительный массив размером n, к какому классу пространственной эффективности будет относиться двухэтапный алгоритм?

10. Вы подошли вплотную к стене, не имеющей ни начала, ни конца. Известно, что в стене есть дверь, но вы не знаете в каком направлении и как долго нужно двигаться, чтобы найти эту дверь. Заметить дверь в стене можно, только подойдя непосредственно к ней и став напротив. Разработайте алгоритм, который бы позволял находить дверь в стене методом обхода не более чем за $O(n)$ шагов, где n — не известное заранее число шагов между вашим первоначальным положением и дверью. (Задача взята из [88], №652.)

2.3 Математический анализ нерекурсивных алгоритмов

В этом разделе мы систематически применим описанные в разделе 2.1 основы анализа эффективности алгоритмов для исследования нерекурсивных алгоритмов. Начнем с очень простого примера, на основе которого продемонстрируем все важные этапы, которые обычно выполняются при анализе подобных алгоритмов.

Пример 1. Рассмотрим задачу поиска наибольшего элемента в списке из n чисел. Для простоты предположим, что этот список реализован в виде массива. Ниже приведен псевдокод алгоритм решения этой задачи.

Алгоритм $MaxElement(A[0..n-1])$

```
// Входные данные:  массив вещественных чисел A[0..n − 1]
// Выходные данные: возвращается значение наибольшего
//                  элемента массива A
maxval ← A[0]
for i ← 1 to n − 1 do
  if A[i] > maxval
    maxval ← A[i]
return maxval
```

Очевидно, что в этом алгоритме размер входных данных нужно оценивать по количеству элементов в массиве, т.е. числом n. Операции, выполняемые чаще всего, находятся во внутреннем цикле **for** алгоритма. Таких операций две: сравнение ($A[i] > maxval$) и присваивание ($maxval \leftarrow A[i]$). Какую из них считать базовой? Поскольку сравнение выполняется на каждом шаге цикла, а присваивание — нет, логично считать, что основной операцией алгоритма является операция присваивания. (Обратите внимание, что для любого массива размером n количество операций сравнения в рассматриваемом алгоритме постоянно. Поэтому не имеет смысла отдельно рассматривать эффективность алгоритма для худшего, типичного и лучшего случаев.)

Обозначим через $C(n)$ количество выполняемых в алгоритме операций сравнения и попытаемся вывести формулу, выражающую их зависимость от размера входных данных n. Известно, что за один цикл в алгоритме выполняется одна операция сравнения. Этот процесс повторяется для каждого значения переменной цикла i, которое изменяется от 1 до $n-1$ включительно. Поэтому для $C(n)$ получаем следующую сумму:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

Значение этой суммы очень легко вычислить, поскольку в ней единица суммируется сама с собой $n-1$ раз. Поэтому

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

■

Ниже предлагается общий план анализа эффективности нерекурсивных алгоритмов.

Общий план анализа эффективности нерекурсивных алгоритмов

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма. (Как правило, она находится в наиболее глубоко вложенном внутреннем цикле алгоритма.)
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.
4. Запишите сумму, выражающую количество выполняемых основных операций алгоритма[4].
5. Используя стандартные формулы и правила суммирования, упростите полученную формулу для количества основных операций алгоритма. Если это невозможно, определите хотя бы их порядок роста.

Прежде чем рассмотреть следующие примеры, полезно обратиться к приложению А, в котором приведен список формул суммирования и правила вычисления,

[4]Иногда при анализе нерекурсивных алгоритмов применяют не суммы, а рекуррентные отношения, выражающие зависимость количества выполняемых основных операций алгоритма. Создание и решение рекуррентных отношений более типично для анализа рекурсивных алгоритмов (см. раздел. 2.4).

которые часто используются в процессе анализа алгоритмов. Особенно часто мы будем использовать два основных правила суммирования:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i \tag{R1}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i \tag{R2}$$

и две формулы суммирования:

$$\sum_{i=l}^{u} 1 = u - l + 1, \text{ где } l \leqslant u \text{ — целые числа, представляющие нижнюю и верхнюю границы суммы} \tag{S1}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2) \tag{S2}$$

Обратите внимание, что формула $\sum_{i=1}^{n-1} 1 = n-1$, которой мы воспользовались в примере 1, является частным случаем формулы (S1) при $l = 1$ и $u = n - 1$.

Пример 2. Рассмотрим задачу проверки единственности элементов. Другими словами, нужно убедиться, что все элементы массива различны. Эту задачу можно решить с помощью приведенного ниже несложного алгоритма.

Алгоритм *UniqueElements* $(A[0..n-1])$

```
// Входные данные:  массив вещественных чисел A[0..n − 1]
// Выходные данные: возвращается значение "true", если все
//                  элементы массива A различны, и "false"
//                  в противном случае
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j]
            return false
return true
```

В этом алгоритме, как и в предыдущем, размер входных данных вполне естественно оценивать по количеству элементов в массиве, т.е. числом n. Поскольку в наиболее глубоко вложенном внутреннем цикле алгоритма выполнятся только одна операция сравнения двух элементов, ее и будем считать основной операцией этого алгоритма. Обратите внимание, что количество операций сравнения будет зависеть не только от общего числа n элементов в массиве, но и от того, есть ли в массиве одинаковые элементы, и если есть, то на каких позициях они расположены. Ограничимся рассмотрением наихудшего случая.

По определению наихудший случай входных данных соответствует такому массиву элементов, при обработке которого количество операций сравнений $C_{worst}(n)$ будет максимальным среди всей совокупности входных массивов размером n. После анализа внутреннего цикла алгоритма приходим к выводу, что наихудший случай входных данных (т.е. когда цикл выполняется от начала до конца, а не завершается досрочно) может возникнуть при обработке массивов двух типов: а) в которых нет одинаковых элементов; б) в которых два одинаковых элемента находятся рядом и расположены в самом конце массива. В подобных случаях при каждом повторе внутреннего цикла в нашем алгоритме выполняется одна операция сравнения. При этом переменная цикла j последовательно принимает значения от $i+1$ до $n-1$. Внутренний цикл каждый раз повторяется заново при каждом выполнении внешнего цикла. При этом переменная внешнего цикла i последовательно принимает значения от 0 до $n-2$. Таким образом, получаем:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) =$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} =$$

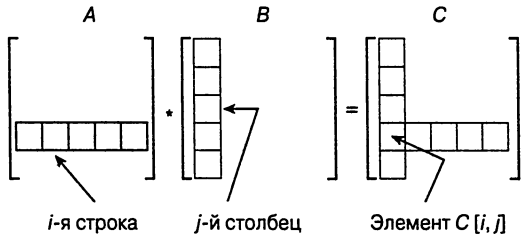
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

В приведенных выше выкладках сумму $\sum_{i=0}^{n-2} (n-1-i)$ можно вычислить проще, если воспользоваться формулой (S2):

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 \overset{(S2)}{=} \frac{(n-1)n}{2}.$$

Обратите внимание, что этот результат можно было легко предсказать: в рассматриваемом алгоритме в самом худшем случае для массива, состоящего из n элементов, нужно сравнить между собой все $(n-1)n/2$ различных пар элементов. ■

Пример 3. Для двух заданных матриц A и B размером $n \times n$ определите временну́ю эффективность алгоритма их умножения $C = AB$, основанного на определении этой операции. По определению C — это матрица размером $n \times n$, элементы которой являются скалярными произведениями соответствующих строки матрицы A ни столбца матрицы B, как показано ниже.



Здесь для каждой пары индексов $0 \leqslant i, j \leqslant n-1$ элемент

$$C[i,j] = A[i,0]\,B[0,j] + \cdots + A[i,k]\,B[k,j] + \cdots + A[i,n-1]\,B[n-1,j].$$

Алгоритм $MatrixMultiplication\,(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
// Выполняется умножение двух квадратных матриц размером
// n × n. Используется алгоритм, основанный на определении
// этой операции
// Входные данные:  две квадратные n × n матрицы A и B
// Выходные данные: матрица C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

В этом алгоритме размер входных данных соответствует размеру матрицы n. Во внутреннем цикле алгоритма выполняются две арифметические операции: умножение и сложение. В принципе, в качестве основной операции алгоритма можно выбрать как одну, так и другую операцию. Мы рассмотрим случай, когда в качестве основной выбрана операция умножения (см. раздел 2.1). Заметьте, что для рассматриваемого алгоритма не обязательно отдавать предпочтение одной из этих операций, поскольку на каждом шаге внутреннего цикла каждая из них выполняется только один раз. Поэтому, подсчитав, сколько раз выполняется одна из операций, мы автоматически подсчитываем и количество выполнения другой операции. А теперь давайте составим сумму для общего числа операций умножения $M(n)$, выполняемых в алгоритме. Поскольку это число зависит только от размера исходных матриц, не требуется отдельно рассматривать наихудший, типичный и наилучший случаи.

Вполне очевидно, что на каждом шаге внутреннего цикла алгоритма выполняется только одна операция умножения. При этом значение переменной цикла k последовательно изменяется от нижней границы 0 до верхней границы $n-1$. Поэтому количество операций умножения, выполняемых для каждой пары значений переменных i и j, можно записать как $\sum_{k=0}^{n-1} 1$. Соответственно, общее количество

операций умножения $M(n)$ выражается следующей тройной суммой:

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1.$$

Для вычисления значения этой суммы воспользуемся формулой (S1) и правилом (R1). Принимая во внимание, что значение внутренней суммы $\sum_{k=0}^{n-1} 1 = n$ (почему?), получим:

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Рассматриваемый алгоритм достаточно прост. Поэтому мы могли бы получить тот же самый результат и без всех этих манипуляций с суммами. Хотите знать, как? Для получения результирующей матрицы в алгоритме необходимо вычислить значения n^2 ее элементов. Каждый элемент этой матрицы является скалярным произведением n-элементной строки первой матрицы на n-элементный столбец второй матрицы. При этом выполняется n операций умножения. Таким образом, общее количество операций умножения в рассматриваемом алгоритме составляет $n \cdot n^2 = n^3$. Именно эту последовательность логических рассуждений вы должны были применить при выполнении упражнения 2.1.2 (по крайней мере я это подразумевал, когда ставил ее перед вами).

Время выполнения алгоритма на конкретном компьютере можно оценить с помощью следующего произведения:

$$T(n) \approx c_m M(n) = c_m n^3,$$

где c_m — время выполнения одной команды умножения на рассматриваемом компьютере. Чтобы получить более точную оценку, необходимо также учесть время выполнения команд сложения:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

где c_a — время выполнения одной команды сложения. Обратите внимание, что полученная оценка отличается от прежней только постоянным множителем, а не порядком роста. ■

У вас не должно сложиться ошибочное мнение по поводу того, что приведенным выше планом анализа нерекурсивных алгоритмов можно пользоваться всегда. Бессистемное изменение значения переменной цикла, слишком сложная для анализа сумма и трудности, присущие анализу эффективности алгоритма для среднего случая, — вот только малая часть тех проблем, которые могут оказаться

непреодолимой преградой для исследователя. Однако вопреки такому строгому предупреждению наш план можно с успехом применять для анализа большого количества простых нерекурсивных алгоритмов. В этом вы убедитесь сами, прочитав последующие главы этой книги.

В качестве последнего примера рассмотрим алгоритм, в котором значение переменной цикла изменяется несколько иначе, чем в приведенных выше примерах.

Пример 4. С помощью приведенного ниже алгоритма можно определить количество разрядов, которое будет иметь положительное десятичное число в двоичном представлении.

Алгоритм $Binary(n)$

```
// Входные данные:  целое положительное число n
// Выходные данные: количество разрядов в двоичном
//                  представлении числа n
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

Для начала заметим, что в этом алгоритме операция сравнения $n > 1$, которая выполняется чаще всего, не находится в его внутреннем цикле **while**. Однако она определяет, будет ли выполняться все тело цикла. Поскольку количество выполняемых операций сравнения всегда на единицу больше, чем общее число повторов выполнения цикла, выбор основной операции алгоритма не имеет большого значения.

Более важно в этом примере то, что в процессе вычислений переменная цикла принимает всего насколько значений в интервале от ее минимального до максимального значения. Поэтому для определения количества раз, которые будет выполняться цикл, мы должны использовать другой метод. Поскольку на каждом шаге цикла значение переменной n уменьшается приблизительно вдвое, то искомый результат должен примерно равняться $\log_2 n$. На самом деле точная формула для определения количества выполняемых операций сравнения $n > 1$ равна $\lfloor \log_2 n \rfloor + 1$, что совпадает с приведенной выше формулой (2.1) для количества бит в двоичном представлении числа n. Такой же ответ мы могли бы получить, применив методы, основанные на анализе рекуррентных соотношений. Их мы обсудим в следующем разделе, поскольку они очень удобны для анализа рекурсивных алгоритмов. ■

Упражнения 2.3

1. Вычислите значения приведенных ниже сумм.

   а) $1+3+5+7+\cdots+999$

   б) $2+4+8+16+\cdots+1024$

   в) $\sum_{i=3}^{n+1} 1$ **г)** $\sum_{i=3}^{n+1} i$ **д)** $\sum_{i=0}^{n-1} i(i+1)$

   е) $\sum_{j=1}^{n} 3^{j+1}$ **ж)** $\sum_{i=1}^{n}\sum_{j=1}^{n} ij$

2. Определите порядки роста приведенных ниже сумм.

   а) $\sum_{i=0}^{n-1} \left(i^2+1\right)^2$ **б)** $\sum_{i=2}^{n-1} \lg i^2$

   в) $\sum_{i=1}^{n} (i+1)\, 2^{i-1}$ **г)** $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} (i+j)$

   Используйте обозначения $\Theta(g(n))$ с простейшей функцией $g(n)$.

3. Эмпирическая дисперсия выборки, состоящей из n элементов $x_1, \ldots, x_n$ вычисляется по формуле

$$s^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}, \text{ где } \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n},$$

   или

$$s^2 = \frac{\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2 / n}{n-1}.$$

   Найдите и сравните количество операций деления, умножения, сложения и вычитания (последние две операции обычно объединяются и считаются как одна), которые необходимо выполнить для вычисления эмпирической дисперсии по каждой из приведенных выше формул.

4. Рассмотрим следующий алгоритм.

   Алгоритм $Mystery(n)$

   // **Входные данные:** целое положительное число n

   $S \leftarrow 0$

   for $i \leftarrow 1$ **to** n **do**

   $S \leftarrow S + i * i$

   return S

   а) Что вычисляется с помощью этого алгоритма?

   б) Назовите основную операцию алгоритма.

   в) Сколько раз выполняется основная операция?

г) К какому классу эффективности относится этот алгоритм?

д) Усовершенствуйте этот алгоритм или предложите другой алгоритм и оцените его класс эффективности. Если вам не удастся этого сделать, попытайтесь доказать, что это невозможно.

5. Рассмотрим следующий алгоритм.

Алгоритм *Secret* $(A[0..n-1])$

```
// Входные данные:  массив n вещественных чисел A[0..n − 1]
minval ← A[0]
maxval ← A[0]

for i ← 1 to n − 1 do
    if A[i] < minval
        minval ← A[i]
    if A[i] > maxval
        maxval ← A[i]
return maxval − minval
```

Ответьте на вопросы **а)–д)** задачи 4.

6. Рассмотрим следующий алгоритм.

Алгоритм *Enigma* $(A[0..n-1, 0..n-1])$

```
// Входные данные:  матрица действительных чисел
//                  A[0..n − 1, 0..n − 1]
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i, j] ≠ A[j, i]
            return false
return true
```

Ответьте на вопросы **а)–д)** задачи 4.

7. Улучшите реализацию алгоритма умножения матриц (см. пример 3) за счет уменьшения количества выполняемых операций сложения. Как это отразится на эффективности работы алгоритма?

8. К какому классу эффективности относится алгоритм, описанный в примере 4, как функция от количества битов в двоичном представлении числа n?

9. Докажите формулу

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

Для этого воспользуйтесь методом математической индукции либо постарайтесь применить логику 10-летнего школьника Карла Фридриха Гаусса (Karl Friedrich Gauss) (1777–1855), который впоследствии стал одним из величайших математиков.

10. Рассмотрите один из вариантов важного алгоритма, который мы будем изучать в этой книге позже.

```
Алгоритм GE(A[0..n − 1, 0..n])
  // Входные данные:  матрица вещественных чисел
  //                  A[0..n − 1, 0..n] размером n × n + 1
  for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
      for k ← i to n do
        A[j, k] ← A[j, k] − A[i, k] ∗ A[j, i]/A[i, i]
```

а) Определите класс временно́й эффективности этого алгоритма.

б) Какое место в этом псевдокоде сразу бросается в глаза из-за своей неэффективности? Как можно переписать этот алгоритм, чтобы повысить скорость работы алгоритма?

2.4 Математический анализ рекурсивных алгоритмов

В этом разделе мы увидим, как применить описанные в разделе 2.1 основы анализа эффективности алгоритмов для исследования рекурсивных алгоритмов. А начнем мы с рассмотрения примера, который часто используется в учебниках для иллюстрации идеи рекурсивных алгоритмов.

Пример 1. Вычислим значение функции факториала $F(n) = n!$ для произвольного целого неотрицательного значения n. Так как $n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$ для всех $n \geqslant 1$ и по определению $0! = 1$, мы можем вычислить $F(n) = F(n-1) \cdot n$ при помощи следующего рекурсивного алгоритма.

```
Алгоритм F(n)
  // Рекурсивное вычисление факториала
  // Входные данные:  Целое неотрицательное число n
  // Выходные данные: Значение n!
  if n = 0
    return 1
  else
    return F(n − 1) ∗ n
```

Для простоты будем считать, что размер входных данных алгоритма будет указывать само число n, а не количество битов в его двоичном представлении. Основной операцией этого алгоритма является умножение, количество выполнений которой мы обозначим через $M(n)$.[5] Так как для всех $n > 0$ значение функции $F(n)$ вычисляется по следующей формуле:

$$F(n) = F(n-1) \cdot n,$$

количество операций умножения $M(n)$, выполняемых при этом в алгоритме, должно удовлетворять следующему равенству для всех $n > 0$:

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{Для вычисления} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{Для умножения} \\ F(n-1) \text{ на } n}}$$

В самом деле, для всех $n > 0$, для вычисления значения функции $F(n-1)$ необходимо выполнить $M(n-1)$ операций умножения. Еще одна операция умножения тратится на вычисление значения функции $F(n)$, т.е. умножения полученного результата $F(n-1)$ на n.

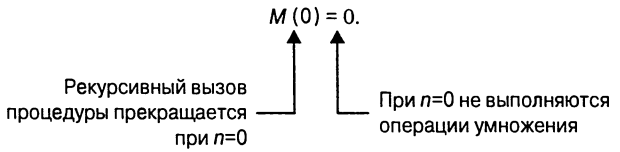
Последнее уравнение определяет последовательность $M(n)$, которую мы должны найти. Обратите внимание, что значение для $M(n)$ не задано явно, т.е. в виде функции от числа n. Оно выражено неявно в виде функции, значение которой зависит от значения этой же функции на предыдущем шаге алгоритма, т.е. при $n-1$. Подобные равенства называют ***рекуррентными уравнениями***, или ***рекуррентными соотношениями*** (***recurrence relations***), или, для краткости, просто ***рекуррентностями*** (***recurrences***). Рекуррентные соотношения играют важную роль не только при анализе алгоритмов, но и в некоторых других областях прикладной математики. Обычно их подробно изучают на лекциях по дискретной математике или дискретным структурам. В приложении Б приведен их очень краткий обзор. Сейчас же наша цель состоит в том, чтобы найти решение рекуррентного соотношения $M(n) = M(n-1) + 1$, т.е. определить явную формулу для последовательности $M(n)$, зависящую только от n.

Следует обратить ваше внимание, что существует бесконечное множество последовательностей, удовлетворяющих решению приведенного выше рекуррентного соотношения. (Можете ли вы привести примеры хотя бы двух из них?) Чтобы найти однозначное решение, нужно задать ***начальные условия*** (***initial condition***), т.е. указать значение, с которого начинается последовательность. Чтобы определить это значение, нужно проанализировать условие, при котором в алгоритме прекращается рекурсивный вызов процедуры:

[5] В качестве альтернативы мы могли бы подсчитать количество выполняемых операций сравнения $n = 0$ или, что то же самое, количество вызовов функции $F(n)$ из самой себя, выполняемых в алгоритме (см. задачу №2 упражнения 2.4).

if $n = 0$ **return** 1

Эта строка кода позволяет сделать два вывода. Во-первых, так как рекурсивный вызов процедуры в нашем алгоритме прекращается при $n = 0$, минимальное значение n, для которого будет выполняться алгоритм и, следовательно, должна быть определена функция $M(n)$, равно 0. Во-вторых, обратившись к строке кода алгоритма, в которой задано условие завершения работы, мы видим, что при $n = 0$ в алгоритме не выполняются операции умножения. Поэтому начальное условие, которое мы искали, будет выглядеть так:



Таким образом, рекуррентное соотношение и начальные условия для количества операций умножения $M(n)$ рассматриваемого нами алгоритма выглядят так:

$$M(n) = M(n-1) + 1 \quad \text{для } n > 0, \qquad (2.2)$$
$$M(0) = 0.$$

Перед тем как приступить к решению этого рекуррентного соотношения, нам необходимо сделать небольшую паузу и напомнить вам одну важную вещь. Выше мы обсуждали две рекурсивные функции. Первая из них — это, собственно, сама функция вычисления факториала $F(n)$, которая определена с помощью следующего рекуррентного уравнения:

$$F(n) = F(n-1) \cdot n \quad \text{для всех } n > 0,$$
$$F(0) = 1.$$

Вторая — это функция $M(n)$, определяющая количество операций умножения, выполняемых в рекурсивном алгоритме вычисления функции факториала $F(n)$, псевдокод которого приведен в начале этого раздела. Как было показано выше, функция $M(n)$ определяется с помощью рекуррентного соотношения (2.2). И именно его нам нужно решить.

Хотя в данном случае решение нетрудно "предсказать" (вспомните, какая из последовательностей начинается с 0 при $n = 0$ и увеличивается на 1 на каждом следующем шаге?), тем не менее будет крайне полезно получить его систематическим образом. Существует несколько методик решения рекуррентных отношений. Мы воспользуемся одной из них, которая называется ***методом обратной подстановки*** (***method of backward substitutions***). Идея метода и, собственно, его название сразу станут понятны после того, как мы применим его для решения

нашего рекуррентного уравнения:

$$M(n) = M(n-1) + 1 = \quad \text{подставляем } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 =$$
$$= M(n-2) + 2 = \quad \text{подставляем } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 =$$
$$= M(n-3) + 3.$$

Взглянув на первые три строки, легко заметить закономерность, которая позволит предсказать не только вид следующей строки решения (кстати, какой она должна быть?), но также и вывести для него общую формулу:

$$M(n) = M(n-i) + i.$$

Строго говоря, правильность этой формулы должна быть доказана с помощью метода математической индукции. Однако гораздо легче сначала получить решение, как показано ниже, а затем проверить его правильность.

Осталось только применить начальные условия. Поскольку они заданы для $n = 0$, то, чтобы получить последнюю строку (или конечный результат) решения методом обратных подстановок, нужно в приведенной выше общей формуле закономерности выполнить подстановку для $i = n$:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Вы не должны разочаровываться из-за того, что на получение этого столь "очевидного" и простого ответа было затрачено довольно много усилий. Преимущества описываемого метода, который был проиллюстрирован на этом простом примере, станут понятны чуть ниже, когда придется решать более сложные рекуррентные уравнения. Обратите также внимание, что для получения произведения n последовательных целых чисел можно было воспользоваться простым циклическим алгоритмом, в котором выполнялось бы ровно столько же операций умножения. При этом в нем не тратилось бы время на дополнительный вызов процедур и не расходовалась бы дополнительная память для размещения параметров рекурсивной процедуры в стеке.

Следует отметить, что для задачи вычисления $n!$ проблема временно́й эффективности алгоритма не стоит так уж остро. Как было показано в разделе 2.1, факториал растет настолько быстро, что реально мы можем вычислить ее значения только для очень малых n. Мы привели этот простой и понятный пример только для демонстрации стандартного подхода, используемого при анализе рекурсивных алгоритмов. ■

Обобщая опыт, полученный при исследовании рекурсивного алгоритма вычисления $n!$, можно составить следующий общий план исследования рекурсивных алгоритмов.

Общий план анализа эффективности рекурсивных алгоритмов

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма.
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.
4. Составьте рекуррентное уравнение, выражающее количество выполняемых основных операций алгоритма, и укажите соответствующие начальные условия.
5. Найдите решение рекуррентного уравнения или, если это невозможно, определите хотя бы его порядок роста.

Пример 2. В качестве следующего примера мы рассмотрим еще одну ставшую уже классической задачу, для решения которой используется рекурсивный алгоритм. Речь идет о головоломке под названием ***Ханойские башни***. Суть ее состоит в том, что у нас (ну или у некоторого мифического персонажа, если вам не хочется перемещать все эти диски) есть n дисков разного диаметра и три колышка. В исходном состоянии все диски надеты на первый колышек и упорядочены по диаметру, т.е. самый большой диск находится внизу стопки, а самый малый — вверху. Цель задачи — перенести все диски на третий колышек, используя при необходимости в качестве вспомогательного второй колышек. Нужно учесть, что за один раз можно перенести только один диск, и кроме того, нельзя помещать диск большего диаметра на диск меньшего диаметра.

Эта задача решается с помощью изящного рекурсивного алгоритма, как показано на рис. 2.4. Чтобы перенести $n > 1$ дисков с первого колышка на третий (при этом второй колышек используется в качестве вспомогательного), сначала нужно рекурсивно перенести $n - 1$ дисков с первого колышка на второй (используя при этом третий колышек в качестве вспомогательного). После этого мы должны перенести наибольший диск непосредственно с первого колышка на третий и, наконец, рекурсивно перенести $n - 1$ дисков со второго колышка на третий (используя при этом первый колышек в качестве вспомогательного). Естественно, что при $n = 1$, мы можем непосредственно перенести единственный диск с первого колышка на третий.

Давайте применим приведенный выше общий план анализа эффективности рекурсивных алгоритмов к задаче о Ханойских башнях. Очевидно, что в данном случае размер входных данных нужно оценивать по количеству дисков n, а основной операцией алгоритма будет перенос одного диска. Понятно, что количество переносов $M(n)$ должно зависеть только от числа n. Поэтому для любого $n > 1$

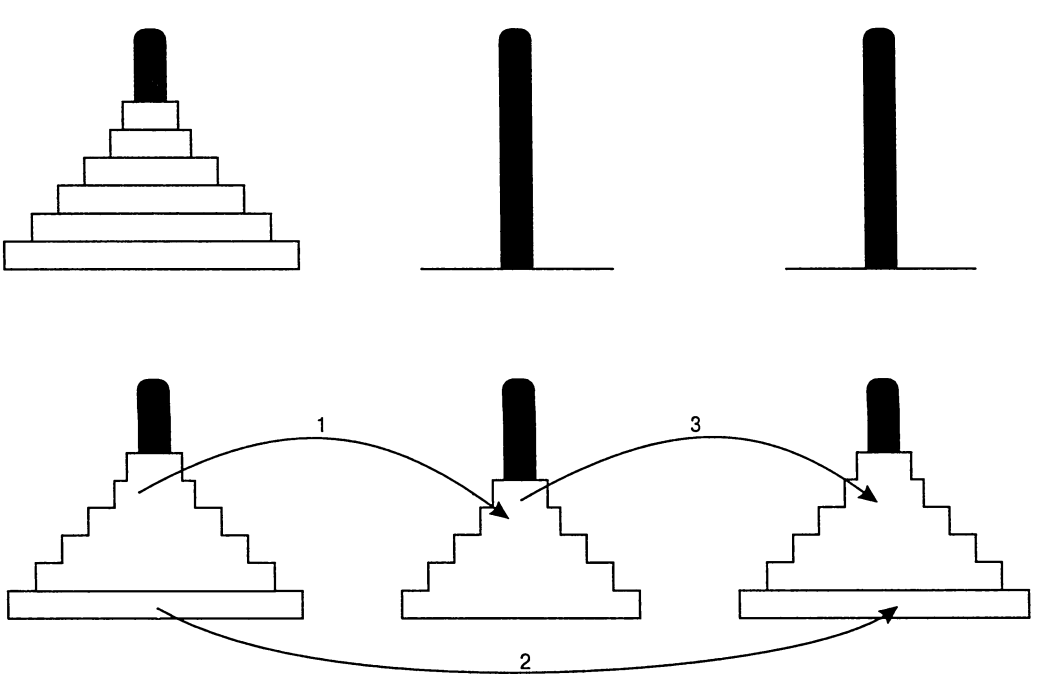


Рис. 2.4. Иллюстрация рекурсивного алгоритма решения головоломки о Ханойских башнях

будет верно следующее рекуррентное уравнение:

$$M(n) = M(n-1) + 1 + M(n-1).$$

Добавив к нему очевидное начальное условие $M(1) = 1$, получим следующее рекуррентное соотношение для количества переносов дисков $M(n)$:

$$\begin{aligned} &M(n) = 2M(n-1) + 1 \quad \text{для } n > 1, \\ &M(1) = 1. \end{aligned} \tag{2.3}$$

Для решения этого рекуррентного уравнения снова воспользуемся методом обратных подстановок:

$$\begin{aligned} &M(n) = 2M(n-1) + 1 = && \text{подставляем } M(n-1) = 2M(n-2) + 1 \\ &= 2[2M(n-2) + 1] + 1 = \\ &= 2^2 M(n-2) + 2 + 1 = && \text{подставляем } M(n-2) = 2M(n-3) + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = \\ &= 2^3 M(n-3) + 2^2 + 2 + 1 \end{aligned}$$

В первых трех строках этого решения прослеживается очевидная закономерность, так что четвертая строка будет выглядеть так:

$$2^4 M(n-4) + 2^3 + 2^2 + 2 + 1.$$

Чтобы получить обобщенную формулу решения, подставим вместо номера строки число i. В результате получим следующее:

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Поскольку начальные условия заданы для $n = 1$, то, чтобы получить соответствующую этому числу строку решения рекуррентного отношения, нужно подставить $i = n - 1$. В результате получим следующее решение рекуррентного уравнения (2.3):

$$\begin{aligned} M(n) &= 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1 = \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Мы выяснили, что рассматриваемый алгоритм относится к классу экспоненциальных. А это означает, что он будет работать невообразимо длительное время даже для умеренных значений n (см. упражнение 2.4.5). Причем это отнюдь не означает, что мы выбрали для решения головоломки о ханойских башнях плохой алгоритм. На самом деле несложно доказать, что данный алгоритм является самым эффективным среди всех возможных алгоритмов решения нашей задачи. Все дело заключается в сложности, присущей самой задаче, что существенно затрудняет ее решение численными методами. Тем не менее этот пример позволяет сделать один важный вывод:

> Рекурсивные алгоритмы следует использовать очень осторожно, поскольку часто за их внешней компактностью скрывается крайняя неэффективность.

Если в рекурсивном алгоритме выполняется более одного вызова его самого, то для анализа такого алгоритма полезно построить дерево его рекурсивных вызовов. Узлы такого дерева будут соответствовать рекурсивным вызовам. Их можно обозначить в соответствии со значением параметра (или, в более общем случае, нескольких параметров), который передается рекурсивной функции в момент вызова. Для нашего примера с ханойскими башнями дерево рекурсивных вызовов будет выглядеть так, как показано на рис. 2.5. Сосчитав количество узлов в дереве, мы получим общее число вызовов, выполняемых в этом алгоритме:

$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1,$$

где l — номер уровня в дереве, показанном на рис. 2.5.

Как видим, это число соответствует количеству переносов дисков, которое было получено нами ранее. ■

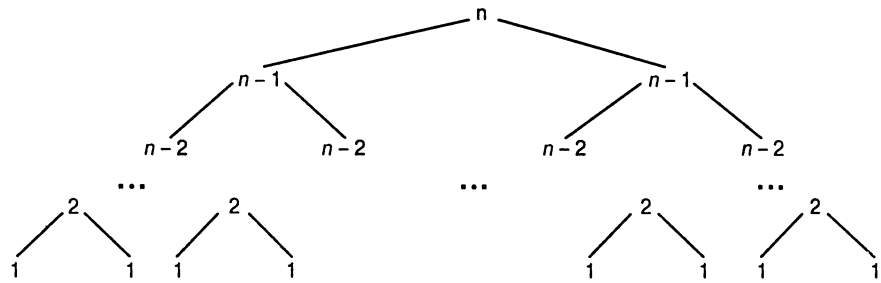


Рис. 2.5. Дерево рекурсивных вызовов, выполняемых в алгоритме решения головоломки о ханойских башнях.

Пример 3. В качестве следующего примера рассмотрим рекурсивный вариант алгоритма, описанного в конце раздела 2.3.

Алгоритм $BinRec(n)$

```
// Входные данные:  целое положительное число n
// Выходные данные: количество разрядов в двоичном
//                  представлении числа n
if n = 1
    return 1
else
    return BinRec(⌊n/2⌋) + 1
```

Давайте построим рекуррентное уравнение и зададим начальные условия для количества операций сложения $A(n)$, выполняемых в этом алгоритме. Количество операций сложения, выполняющихся при вычислении функции $BinRec(\lfloor n/2 \rfloor)$, равно $A(\lfloor n/2 \rfloor)$. К этому нужно прибавить еще одну операцию сложения, которая выполняется в алгоритме для увеличения на единицу возвращаемого функцией $BinRec$ значения. Поэтому для $n > 1$ можно записать следующее рекуррентное уравнение:

$$A(n) = A(\lfloor n/2 \rfloor) + 1. \tag{2.4}$$

Поскольку процесс рекурсивных вызовов завершается при $n = 1$ и при этом не выполняется никаких операций сложения, начальное условие для этого алгоритма будет выглядеть так:

$$A(1) = 0.$$

Наличие в параметре функции *BinRec* выражения $\lfloor n/2 \rfloor$ приводит к тому, что в методе обратных подстановок сложно использовать значения параметра n, которые не являются степенью 2. Поэтому стандартным подходом при решении такого

рекуррентного отношения является поиск решения только для $n = 2^k$ с последующим применением теоремы, называемой ***правилом сглаживания*** (***smoothness rule***), которая описана в приложении Б. В ней утверждается, что сделанную оценку порядка роста функции для $n = 2^k$ можно с очень высокой степень приближенности считать правильной для всех значений n. (В качестве альтернативы, найдя решение для $n = 2^k$, иногда можно немного видоизменить его так, чтобы получить корректную формулу для произвольного значения n.) Итак, давайте применим описанное выше правило к нашему рекуррентному уравнению, которое при $n = 2^k$ примет следующий вид:

$$A\left(2^k\right) = A\left(2^{k-1}\right) + 1 \quad \text{при } k > 0,$$
$$A\left(2^0\right) = 0.$$

Теперь можно без проблем применить метод обратной подстановки:

$$A\left(2^k\right) = A\left(2^{k-1}\right) + 1 = \qquad \text{подставляем } A\left(2^{k-1}\right) = A\left(2^{k-2}\right) + 1$$
$$= \left[A\left(2^{k-2}\right) + 1\right] + 1 =$$
$$= A\left(2^{k-2}\right) + 2 = \qquad \text{подставляем } A\left(2^{k-2}\right) = A\left(2^{k-3}\right) + 1$$
$$= \left[A\left(2^{k-3}\right) + 1\right] + 2 =$$
$$= A\left(2^{k-3}\right) + 3 = \qquad \cdots$$
$$\cdots$$
$$= A\left(2^{k-i}\right) + i =$$
$$\cdots$$
$$= A\left(2^{k-k}\right) + k.$$

В результате получаем, что

$$A\left(2^k\right) = A\left(1\right) + k = k.$$

Поскольку $n = 2^k$, получаем, что $k = \log_2 n$. Поэтому

$$A\left(n\right) = \log_2 n \in \Theta\left(\log n\right).$$

На самом деле можно доказать (см. упражнение 2.4.6), что точное решение для произвольного значения n получается, если переписать формулу как $A\left(n\right) = \lfloor \log_2 n \rfloor$. ■

В этом разделе мы рассмотрели только азы анализа рекурсивных алгоритмов. Приведенные в нем методики будут использоваться в последующих главах этой книги, и по мере необходимости мы их будем расширять. В частности, в следующем разделе мы рассмотрим числа Фибоначчи. При анализе алгоритма их вычисления мы применим более сложные рекуррентные отношения, для решения которых воспользуемся методом, отличающимся от метода обратных подстановок.

Упражнения 2.4

1. Найдите решение для следующих рекуррентных отношений:
   а) $x(n) = x(n-1) + 5$, для $n > 1$, $x(1) = 0$
   б) $x(n) = 3x(n-1)$, для $n > 1$, $x(1) = 4$
   в) $x(n) = x(n-1) + n$, для $n > 0$, $x(0) = 0$
   г) $x(n) = x(n/2) + n$, для $n > 1$, $x(1) = 1$ (найдите решение для $n = 2^k$)
   д) $x(n) = x(n/3) + 1$, для $n > 1$, $x(1) = 1$ (найдите решение для $n = 3^k$)
2. Постройте рекуррентное уравнение для количества вызовов функции $F(n)$ в рекурсивном алгоритме вычисления $n!$ и найдите его решение.
3. Рассмотрим приведенный ниже рекурсивный алгоритм вычисления суммы кубов первых n целых чисел $S(n) = 1^3 + 2^3 + \cdots + n^3$.

   Алгоритм $S(n)$
   // **Входные данные:** целое положительное число n
   // **Выходные данные:** сумма кубов первых n целых чисел
   if $n = 1$
   return 1
   else
   return $S(n-1) + n * n * n$

   а) Постройте рекуррентное уравнение для количества выполнений основной операции алгоритма и найдите его решение.
   б) Сравните этот алгоритм с простым нерекурсивным алгоритмом.
4. Рассмотрим следующий рекурсивный алгоритм.

   Алгоритм $Q(n)$
   // **Входные данные:** целое положительное число n
   if $n = 1$
   return 1

else
return $Q(n-1)+2*n-1$

а) Постройте рекуррентное уравнение для значения этой функции, найдите его решение и определите, что вычисляется с помощью этого алгоритма.

б) Постройте рекуррентное уравнение для количества умножений, выполняемых в алгоритме, и найдите его решение.

в) Постройте рекуррентное уравнение для количества сложений/вычитаний, выполняемых в алгоритме, и найдите его решение.

5. а) В оригинальном варианте головоломки о ханойских башнях, который был опубликован в 1890 году французским математиком Эдуардом Лукасом (Edouard Lucas), после перемещения 64 дисков из мифической Башни Брахмы должен был наступить конец света. Оцените, сколько лет потребуется служителю культа для перемещения всех дисков, считая, что он выполняет эту работу со скоростью один диск в минуту. (Для простоты предположим, что служитель не будет ни есть, ни спать, т.е. не будет отвлекаться от своей работы и жить бесконечно долго.)

б) Сколько раз в этом алгоритме будет перенесен i-ый наибольший диск ($1 \leqslant i \leqslant n$)?

в) Придумайте нерекурсивный алгоритм решения задачи о ханойских башнях.

6. а) Докажите, что точное количество операций сложения, выполняемых в рекурсивном алгоритме $BinRec(n)$ для произвольного положительного целого числа n, равно $\lfloor \log_2 n \rfloor$.

б) Постройте рекуррентное соотношение для количества сложений, выполняемых нерекурсивной версией этого алгоритма (см. раздел 2.3, пример 4), и решите его.

7. а) Разработайте рекурсивный алгоритм вычисления значения 2^n для произвольного неотрицательного целого числа n, который основан на формуле $2^n = 2^{n-1} + 2^{n-1}$.

б) Постройте рекуррентное уравнение для количества операций сложения, выполняемых в алгоритме, и найдите его решение.

в) Изобразите дерево рекурсивных вызовов для этого алгоритма и подсчитайте количество вызовов рекурсивной функции, выполняемых алгоритмом.

г) Можно ли сказать, что это хороший алгоритм для решения поставленной задачи?

8. Рассмотрим следующий рекурсивный алгоритм.

Алгоритм $Min1\,(A\,[0..n-1])$

```
// Входные данные: массив A[0..n − 1] вещественных чисел
if n = 1
    return A[0]
else
    temp ← Min1(A[0..n − 2])
    if temp ⩽ A[n − 1]
        return temp
    else
        return A[n − 1]
```

а) Что вычисляется с помощью этого алгоритма?

б) Постройте рекуррентное уравнение для количества выполнений основной операции алгоритма и найдите его решение.

9. Рассмотрите еще один алгоритм решения задачи 8, в котором рекурсивно выполняется разделение массива на две части. Первый раз функция вызывается так: $Min2\,(A\,[0..n-1])$.

Алгоритм $Min2\,(A\,[l..r])$

```
if l = r
    return A[l]
else
    temp1 ← Min2(A[l..⌊(l + r)/2⌋])
    temp2 ← Min2(A[⌊(l + r)/2⌋ + 1..r])
    if temp1 ⩽ temp2
        return temp1
    else
        return temp2
```

а) Постройте рекуррентное уравнение для количества выполнения основной операции алгоритма и найдите его решение.

б) Какой из алгоритмов — $Min1$ или $Min2$ — быстрее? Можете ли вы предложить алгоритм для решения этой же задачи, который был бы эффективнее двух рассмотренных выше алгоритмов?

10. Детерминант (определитель) матрицы размером $n \times n$

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

обозначаемый как $\det A$, может быть определен следующим образом. При $n = 1$ он равен a_{11}. При $n > 1$ его можно вычислить по приведенной ниже рекурсивной формуле

$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j.$$

В этой формуле элемент s_j равен $+1$, если j нечетное, и -1, если j четное; a_{1j} — это элемент первой строки и j-го столбца, а A_j — детерминант матрицы размером $(n-1) \times (n-1)$, полученной из исходной матрицы A путем вычеркивания первой строки и j-го столбца.

а) Постройте рекуррентное уравнение для количества операций умножения, выполняемых при вычислении детерминанта по описанной выше рекурсивной формуле.

б) Не решая это рекуррентное уравнение, что вы можете сказать о его порядке роста в сравнении с факториалом $n!$?

2.5 Пример: числа Фибоначчи

В этом разделе мы рассмотрим ***числа Фибоначчи***, т.е. знаменитую последовательность целых чисел:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \tag{2.5}$$

Ее можно определить с помощью простого рекуррентного соотношения

$$F(n) = F(n-1) + F(n-2) \quad \text{для } n > 1 \tag{2.6}$$

и двух начальных условий

$$F(0) = 0, \quad F(1) = 1. \tag{2.7}$$

Эта последовательность чисел впервые была получена в 1202 году итальянским математиком Леонардо Фибоначчи (Leonardo Fibonacci) при решении задачи о размножении кроликов. С тех времен в естественном мире было обнаружено множество примеров последовательностей чисел наподобие чисел Фибоначчи; они используются даже при решении таких задач, как предсказание биржевых цен на товары и цен на акции. Существуют и другие интересные применения чисел Фибоначчи, в том числе и в области информатики. Например, наихудший случай входных данных для алгоритма Евклида соответствует последовательным числам Фибоначчи (однако это уже выходит за рамки нашего изложения). В этом разделе мы сначала найдем явную формулу для определения n-го элемента последовательности чисел Фибоначчи $F(n)$, а затем кратко обсудим алгоритм для его вычисления.

Явная формула для определения n-го элемента последовательности чисел Фибоначчи

При попытке применить метод обратных подстановок для решения рекуррентного соотношения (2.6) мы столкнемся с тем, что выделить очевидную закономерность для получения его решения будет невозможно. Поэтому в данном случае мы применим результаты теоремы, в которой ищется решение для ***однородной линейной рекурсии второго порядка с постоянными коэффициентами***:

$$ax(n) + bx(n-1) + cx(n-2) = 0. \tag{2.8}$$

Здесь a, b и c — постоянные вещественные числа (причем $a \neq 0$), которые называются коэффициентами рекурсии, а $x(n)$ — это неизвестная последовательность, которую нужно найти. Согласно этой теореме (см. теорему 1 приложения Б), рекурсия (2.8) имеет бесконечное количество решений, которые можно получить с помощью одной из трех формул. Какую из трех формул следует применять в каждом конкретном случае, зависит от количества вещественных корней квадратного уравнения, коэффициенты которого совпадают с коэффициентами рекурсии (2.8):

$$ar^2 + br + c = 0. \tag{2.9}$$

Вполне логично, что выражение (2.9) называется ***характеристическим уравнением*** для рекурсии (2.8).

Применим указанную теорему к случаю вычисления последовательности чисел Фибоначчи. Для этого перепишем рекуррентное уравнение (2.6) следующим образом:

$$F(n) - F(n-1) - F(n-2) = 0. \tag{2.10}$$

Его характеристическое уравнение имеет вид:

$$r^2 - r - 1 = 0.$$

Корни этого уравнения вычисляются по следующей формуле:

$$r_{1,2} = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}.$$

Поскольку это характеристическое уравнение имеет два различных вещественных корня, для решения рекуррентного соотношения (2.10) мы должны использовать формулу, приведенную в первом случае теоремы 1:

$$F(n) = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Настало время учесть начальные условия (2.7). Воспользуемся ими, чтобы определить конкретные значения параметров α и β. Для этого подставим числа 0 и 1 (т.е. значения параметра n, для которого приведены начальные условия) в последнюю формулу и приравняем полученный результат, соответственно, к 0 и 1 (т.е. к значениям $F(0)$ и $F(1)$ согласно формуле (2.7)):

$$F(0) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^0 + \beta\left(\frac{1-\sqrt{5}}{2}\right)^0 = 0,$$

$$F(1) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^1 + \beta\left(\frac{1-\sqrt{5}}{2}\right)^1 = 1.$$

Проведя ряд несложных алгебраических преобразований, в конечном итоге мы получим приведенную ниже систему из двух линейных уравнений, содержащую неизвестные коэффициенты α и β:

$$\begin{aligned} \alpha &+ & \beta &= 0 \\ \left(\left(1+\sqrt{5}\right)/2\right)\alpha &+ & \left(\left(1-\sqrt{5}\right)/2\right)\beta &= 1 \end{aligned}$$

Найдя решение этой системы уравнений (например, подставив во второе уравнение $\beta = -\alpha$ и решив его для параметра α), получим следующие значения для искомых коэффициентов:

$$\alpha = \frac{1}{\sqrt{5}}, \quad \beta = -\frac{1}{\sqrt{5}}.$$

Таким образом,

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right) \qquad (2.11)$$

где

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803 \quad \text{и} \quad \hat{\phi} = \frac{1-\sqrt{5}}{2} = -\frac{1}{\phi} \approx -0.61803.^6$$

На первый взгляд трудно поверить в то, что формула (2.11), в которой в произвольную целую степень возводятся два иррациональных числа, определяет n-ый элемент последовательности целых чисел Фибоначчи (2.5), но тем не менее это так!

[6] Константу ϕ называют золотым сечением (golden ratio). В древности такое отношение длин сторон прямоугольника считали наиболее оптимальным и приятным для восприятия человеческим глазом. Поэтому его сознательно использовали древние архитекторы и скульпторы.

Формула (2.11) имеет одно явное преимущество: при взгляде на нее сразу понятно, что функция $F(n)$ имеет экспоненциальный порядок роста (помните задачу о размножении кроликов?), т.е. $F(n) \in \Theta(\phi^n)$. Это следует из констатации того факта, что значение числа $\hat{\phi}$ находится в интервале между -1 и 0. Поэтому при n, стремящемся к бесконечности, значение члена $\hat{\phi}^n$ становится бесконечно малым. На самом деле можно доказать, что влияние второго члена формулы (2.11) $\frac{1}{\sqrt{5}}\hat{\phi}^n$ на значение $F(n)$ можно учесть, если округлить значение первого члена этой формулы до ближайшего целого числа. Другим словами, для любого неотрицательного целого числа n будет справедлива следующая формула:

$$F(n) = \frac{1}{\sqrt{5}}\phi^n, \text{ округленное до ближайшего целого числа.} \qquad (2.12)$$

Алгоритмы вычисления чисел Фибоначчи

Несмотря на то что числа Фибоначчи имеют много замечательных свойств, в этом разделе мы ограничимся лишь несколькими замечаниями по поводу существующих алгоритмов для их вычисления. В самом деле, последовательность этих чисел растет настолько быстро, что здесь мы должны рассматривать не эффективный по времени метод для их вычисления, а способы работы с большими числами. Кроме того, для простоты в приведенных ниже алгоритмах мы будем учитывать только такие операции, как сложение и умножение. Поскольку последовательность чисел Фибоначчи очень быстро растет до огромных значений, она должна быть проанализирована более подробно, а не так, как это сделано ниже. Тем не менее, несмотря на сделанное предупреждение, рассмотренные алгоритмы и методы их анализа являются хорошими примерами для тех, кто учится проектировать и анализировать алгоритмы.

А начнем мы с использования рекурсии (2.6) и ее начальных условий (2.7) для построения очевидного рекурсивного алгоритма вычисления функции $F(n)$.

Алгоритм $F(n)$

```
// Вычисление n-того элемента последовательности чисел
// Фибоначчи с использованием рекурсивного алгоритма,
// соответствующего определению
// Входные данные:  целое неотрицательное число n
// Выходные данные: n-ый элемент последовательности чисел
//                  Фибоначчи
if n ⩽ 1
  return n
else
  return F(n − 1) + F(n − 2)
```

Прежде чем погрузиться в формальный анализ этого алгоритма, можете ли вы сказать, является ли он эффективным? Впрочем, в любом случае мы должны провести его формальный анализ. Очевидно, что основной операцией этого алгоритма является сложение. Поэтому давайте обозначим через $A(n)$ количество операций сложения, которые выполняются в алгоритме при вычислении функции $F(n)$. Тогда количество операций сложения, выполняемых для вычисления значений функции $F(n-1)$ и $F(n-2)$, будет, соответственно, равно $A(n-1)$ и $A(n-2)$. Кроме того, в алгоритме нужно выполнить еще одну операцию сложения для суммирования этих значений. Таким образом, для $A(n)$ мы получим следующую рекурсию:

$$\begin{aligned} &A(n) = A(n-1) + A(n-2) + 1 \quad \text{для } n > 1, \\ &A(0) = 0, \quad A(1) = 0. \end{aligned} \tag{2.13}$$

Рекурсия $A(n) - A(n-1) - A(n-2) = 1$ полностью совпадает с рекурсией (2.10), но ее правая часть не равна нулю. Подобные рекуррентные отношения называют ***неоднородными рекурсиями*** (***inhomogeneous recurrences***). Для их решения существует несколько общих методик (за подробностями обратитесь к приложению Б или к любому учебнику по дискретной математике), но для нашей конкретной рекурсии можно воспользоваться специальным приемом и тут же получить ее решение. Для этого нужно привести неоднородную рекурсию к однородной, переписав выражение (2.13) следующим образом:

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0.$$

Обозначив $B(n) = A(n) + 1$, получим:

$$\begin{aligned} B(n) - B(n-1) - B(n-2) = 0, \\ B(0) = 1, \quad B(1) = 1. \end{aligned}$$

Точное решение этой однородной рекурсии можно найти по аналогии с решением рекурсии (2.10), которое было приведено выше при поиске явной формулы для $F(n)$. Но можно этого не делать, поскольку легко заметить, что фактически $B(n)$ — это та же рекурсия, что и $F(n)$, за исключением того, что она начинается с двух единиц, т.е. опережает $F(n)$ на один шаг. Поэтому $B(n) = F(n+1)$ и

$$A(n) = B(n) - 1 = F(n+1) - 1 = \frac{1}{\sqrt{5}}\left(\phi^{n+1} - \hat{\phi}^{n+1}\right) - 1$$

Отсюда следует, что $A(n) \in \Theta(\phi^n)$. Если же оценивать размер задачи количеством битов в двоичном представлении числа n, $b = \lfloor \log_2 n \rfloor + 1$, то получится, что найденное решение будет относиться к еще худшему классу эффективности, поскольку представляет собой дважды экспоненту.

Низкую эффективность описанного выше алгоритма можно было предвидеть, взглянув на формулу рекурсии (2.13). В самом деле — в нем содержится два рекурсивных вызова, которым передаются ненамного меньшие значения параметров, чем само число n. (Вы не сталкивались с подобным случаем ранее?) Кроме того, чтобы понять причину низкой эффективности алгоритма, достаточно взглянуть на его дерево рекурсивных вызовов, построенное по результатам выполнения алгоритма. Пример подобного дерева, построенного для $n = 5$ приведен на рис. 2.6. Обратите внимание, что в этом алгоритме функция с одним и тем же значением аргумента вызывается по несколько раз. Поэтому очевидно, что такой алгоритм будет крайне неэффективен.

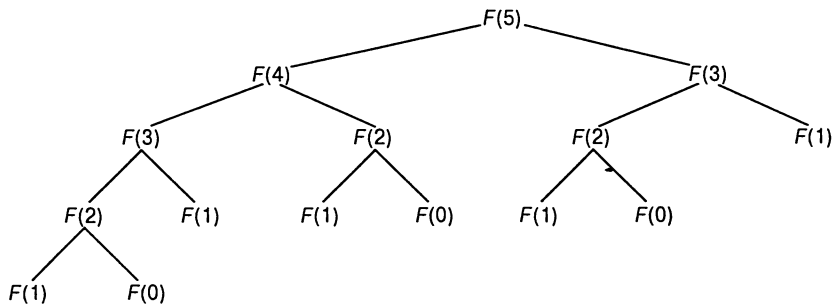


Рис. 2.6. Дерево рекурсивных вызовов, построенное для алгоритма вычисления последовательности чисел Фибоначчи при $n = 5$

Получится намного более эффективный алгоритм, если числа Фибоначчи вычислять последовательно одно за другим в цикле, как показано ниже.

Алгоритм $Fib\,(n)$

// Вычисление n-го элемента последовательности чисел
// Фибоначчи при помощи итеративного алгоритма
// с использованием определения чисел Фибоначчи
// **Входные данные:** целое неотрицательное число n
// **Выходные данные:** n-ый элемент последовательности чисел
// Фибоначчи
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

Очевидно, что в этом алгоритме выполняется $n-1$ операций сложения. Следовательно, алгоритм является линейным, если рассматривать зависимость времени его работы от n, и "всего лишь" экспоненциальным, если рассматривать функцию

от количества битов b в двоичном представлении числа n. Заметьте, что можно избежать использования дополнительного массива для хранения всех предыдущих элементов последовательности Фибоначчи: для выполнения задачи достаточно сохранять только два значения (см. упражнение 2.5.6).

Третий способ вычисления n-го элемента последовательности чисел Фибоначчи заключается в использовании формулы (2.12). Очевидно, что эффективность этого алгоритма будет определяться эффективностью экспоненциального алгоритма, используемого для вычисления ϕ^n. Если это делается путем простого перемножения ϕ на себя $n-1$ раз, то алгоритм будет принадлежать множеству $\Theta(n) = \Theta\left(2^b\right)$. Для решения подобных задач имеются и более быстрые алгоритмы, не относящиеся к экспоненциальному классу. Например, в главах 5 и 6 мы рассмотрим алгоритм решения данной задачи со временем работы $\Theta(\log n) = \Theta(b)$. Заметим, что при реализации последнего алгоритма вычисления n-го элемента последовательности чисел Фибоначчи следует быть особенно внимательным и осторожным. Поскольку все промежуточные результаты вычислений являются иррациональными числами, мы должны убедиться в том, что их приблизительные значения представлены в компьютере с достаточной точностью, чтобы это не повлияло на правильность конечного результата.

Наконец, для вычисления n-го элемента последовательности чисел Фибоначчи существует еще один алгоритм со временем работы $\Theta(\log n)$, в котором используются только целые числа. Он основан на равенстве

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n, \text{ для } n \geqslant 1,$$

и использует эффективные способы возведения матриц в степень.

Упражнения 2.5

1. Поищите Web-сервер, посвященный применению чисел Фибоначчи, и ознакомьтесь с его содержимым.
2. Проверьте методом непосредственной подстановки, что функция $\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ на самом деле удовлетворяет рекуррентному отношению (2.6) для любого $n > 1$ и начальным условиям (2.7) для $n = 0$ и 1.
3. Известно, что максимальное значение переменных простых типов языка Java `int` и `long` составляет, соответственно, $2^{31} - 1$ и $2^{63} - 1$. Найдите наименьшее n, для которого значение n-го элемента последовательности чисел Фибоначчи уже не будет помещаться в ячейку памяти, выделенную под переменную указанного ниже типа:

   а) `int` **б)** `long`

4. Определите количество различных способов подъема по лестнице, состоящей из n ступенек, при условии, что на каждом шаге можно подыматься либо на одну, либо сразу на две ступени. Например, по лестнице, состоящей из трех ступенек, можно подняться тремя способами: 1–1–1, 1–2 и 2–1 [118].

5. Рассмотрим рекурсивный алгоритм вычисления n-го элемента последовательности чисел Фибоначчи $F(n)$, основанный на их определении. Пусть $C(n)$ и $Z(n)$ — количество вычислений значений $F(1)$ и $F(0)$, соответственно. Докажите, что

   а) $C(n) = F(n)$ **б)** $Z(n) = F(n-1)$

6. Усовершенствуйте алгоритм $Fib(n)$, чтобы он требовал при работе только $\Theta(1)$ памяти.

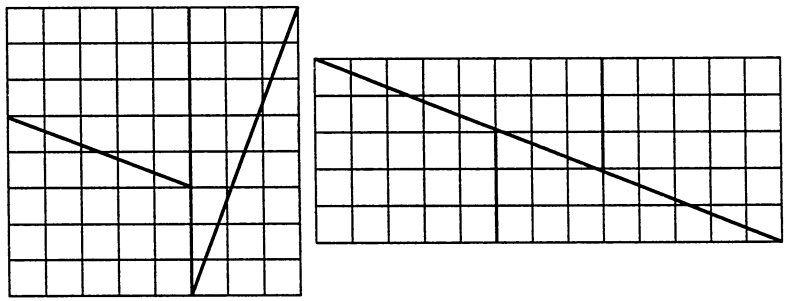
7. Докажите, что при $n \geqslant 1$

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n.$$

8. Сколько делений по модулю будет выполнено алгоритмом Евклида для входных данных, представляющих собой два последовательных числа Фибоначчи — $F(n)$ и $F(n-1)$?

9. **а)** Докажите тождество Кассини: $F(n+1)\,F(n-1) - [F(n)]^2 = (-1)^n$ для всех $n \geqslant 1$.

б) Рассмотрим следующий парадокс, основанный на тождестве Кассини. Возьмем шахматную доску размером 8×8 (или, в общем случае, доску размером $F(n) \times F(n)$, разделенную на $[F(n)]^2$ клеток). Разделим ее на две трапеции и два треугольника, как показано в левой части рисунка. Затем соберем эти трапеции и треугольники в новую фигуру, показанную в правой части рисунка. Площадь левого квадрата равна $8 \times 8 = 64$ клетки, а правого прямоугольника — $13 \times 5 = 65$ клеток. Поясните этот парадокс.



10. Реализуйте на языке программирования по собственному усмотрению три алгоритма вычисления пяти последних цифр n-го числа Фибоначчи, основанных на а) рекурсивном алгоритме $F(n)$; б) итеративном алгоритме $Fib(n)$ и в) версии $Fib(n)$ с экономным использованием памяти. Проведите эксперимент по поиску максимального значения n, для которого разработанная вами программа работает на вашем компьютере меньше 1 минуты.

2.6 Эмпирический анализ алгоритмов

В разделах 2.3 и 2.4 мы видели, как можно выполнить математический анализ рекурсивных и нерекурсивных алгоритмов. Несмотря на успешное применение описанных методов ко многим простым алгоритмам и множество усовершенствованных технологий математического анализа алгоритмов ([45, 46, 93] и [105]), считать возможности математики безграничными нельзя. Имеется масса алгоритмов, причем достаточно простых, которые с трудом поддаются математическому анализу. Как отмечалось в разделе 2.2, это в особенности относится к анализу среднего случая.

Принципиальной альтернативой математическому анализу эффективности алгоритмов является их эмпирический анализ. Вот примерный план проведения этого вида анализа.

Общий план эмпирического анализа эффективности алгоритма

1. Уяснение цели предстоящего эксперимента.
2. Определение измеряемой метрики M и единиц измерения (количество операций или время работы).
3. Определение характеристик входных данных (их диапазон, размер и т.д.).
4. Создание программы, реализующей алгоритм (или алгоритмы), для проведения эксперимента.
5. Генерация образца входных данных.
6. Выполнение алгоритма (или алгоритмов) над образцом входных данных и запись наблюдаемых данных.
7. Анализ полученных данных.

Рассмотрим по очереди указанные шаги. Имеется ряд целей, которые могут быть поставлены перед эмпирическим анализом алгоритмов. Они включают проверку точности теоретических выводов об эффективности алгоритма, сравнение эффективности нескольких алгоритмов, предназначенных для решения одной

и той же проблемы или различных реализаций одного и того же алгоритма, выдвижение гипотезы о классе эффективности алгоритма, выяснение эффективности программы, реализующей алгоритм, на данной конкретной машине. Очевидно, что разработка эксперимента должна зависеть от того, на какой именно вопрос он должен ответить.

В частности, цель эксперимента должна влиять, если не определять, на то, каким образом будет выполняться измерение эффективности алгоритма. Первый вариант заключается во вставке в программу, реализующую алгоритм, счетчика (или счетчиков), которые будут подсчитывать количество выполнений алгоритмом базовых операций. Обычно это достаточно просто, и вы должны только не забывать, что базовые операции могут выполняться не в одном месте программы, и учитывать все возможные их выполнения. Само собой, вы всегда должны проверять модифицированную таким образом программу, чтобы убедиться в ее корректности — как в смысле решения поставленной перед алгоритмом задачи, так и в смысле корректности работы внесенных в программу счетчиков.

Второй вариант заключается в определении времени работы программы, реализующей исследуемый алгоритм. Простейший путь выяснения времени работы — использование системной команды, такой как `time` в UNIX. Можно также определять время работы фрагмента кода, запрашивая системное время непосредственно перед началом выполнения фрагмента (t_{start}) и сразу после его завершения (t_{finish}), а затем просто вычислять разность полученных значений ($t_{finish} - t_{start}$).[7] В C и C++ для этой цели можно использовать функцию `clock`, а в Java — метод `currentTimeMillis()` из класса `System`.

Однако очень важно не забывать о некоторых вещах. Во-первых, системное время обычно не очень точное, и вы можете получить несколько отличающиеся друг от друга результаты при повторных запусках одной и той же программы с одними и теми же входными данными. Очевидным средством противодействия этому эффекту является запуск программы и выполнение измерений несколько раз, с последующим усреднением полученных результатов. Во-вторых, высокая скорость современных компьютеров может привести к тому, что время работы будет невозможно зарегистрировать (будут получаться нулевые значения). Обойти эту неприятность легко, запуская программу в цикле много раз, а затем поделив зарегистрированное время выполнения на количество итераций цикла. В-третьих, на компьютере, работающем под управлением многозадачной операционной системы (как, например, UNIX), регистрируемое время может включать время, затраченное процессором на работу над другими программами, что, понятно, мешает проведению эксперимента. Таким образом, вы должны запросить у операционной системы время, затраченное конкретно на выполнение вашей программы (в UNIX

[7] Если системное время возвращается в "тиках" (ticks), то надо не забыть разделить полученное число на константу, определяющую количество "тиков" в единице времени.

это время называется "пользовательским временем" (user time) и автоматически предоставляется командой `time`).

Таким образом, измерение физического времени работы имеет ряд недостатков как принципиального характера (наиболее важным их них является зависимость от конкретной машины, на которой проводится эксперимент), так и технических, которых нет у метода подсчета базовых операций. С другой стороны, измерение физического времени дает очень конкретную информацию о производительности алгоритма в данной вычислительной среде, что для экспериментатора может оказаться более важным, чем, скажем, класс асимптотической эффективности алгоритма. Кроме того, измерения времени, затраченного на различные части программы, могут помочь выявить узкие места в производительности программы, что не позволит сделать абстрактное рассмотрение базовых операций алгоритма. Получение таких данных, которое называется ***профилированием***, — важный ресурс для эмпирического анализа времени работы алгоритмов; обычно в большинстве сред имеются специальные системные инструменты для получения данной информации.

Независимо от того, решите ли вы измерять производительность алгоритма при помощи подсчета базовых операций или фиксируя время выполнения, перед вами встанет вопрос о выборе образца исходных данных для проведения эксперимента. Зачастую требуется использовать образец входных данных, представляющий собой "типичные" данные. Для некоторых классов алгоритмов — например, алгоритмов для решения задачи коммивояжера, которая будет рассмотрена позже в этой книге, — исследователи разработали набор экземпляров задач, которые используются в качестве тестовых образцов. Однако на практике гораздо чаще приходится сталкиваться с ситуациями, когда входные данные должен выбирать и готовить сам экспериментатор. Обычно приходится самостоятельно решать, каков должен быть размер входных данных (разумно начать с данных относительно небольшого размера и при необходимости постепенно увеличивать их), диапазон величин входных данных (чтобы он не был ни слишком малым, ни слишком большим), и разрабатывать процедуру для генерации входных данных в выбранном диапазоне. Обычно данные либо следуют некоторому шаблону (например, 1000, 2000, 3000, ..., 10 000 или 500, 1000, 2000, 4000, ..., 128 000), либо генерируются случайным образом (например, с равномерным распределением между наименьшим и наибольшим значениями).

Главное достоинство следования определенному шаблону в том, что в этом случае легче проанализировать влияние данных на работу и эффективность алгоритма. Например, если значения входных данных генерируются путем удвоения, можно вычислить отношение $M(2n)/M(n)$ наблюдаемой метрики M (количество операций или время) и увидеть, является ли поведение алгоритма типичным для одного из основных классов эффективности (см. раздел 2.2). Основной недостаток неслучайных величин — возможность того, что для конкретного набора

данных алгоритм продемонстрирует нетипичное поведение. Например, если все значения входных данных четны, а исследуемый алгоритм для нечетных данных работает существенно медленнее, результат эксперимента окажется далек от истины.

Еще один важный вопрос, связанный со входными данными: следует ли использовать несколько экземпляров данных одного и того же размера. Если вы ожидаете, что наблюдаемая метрика может существенно изменяться даже для данных одного и того же размера, вероятно, будет разумно включить во входные данные несколько разных экземпляров данных. (В математической статистике имеются хорошо разработанные методы, которые могут помочь в выработке верного решения в данной ситуации — стоит лишь немного покопаться в литературе на эту тему.) Само собой, при использовании нескольких экземпляров данных наблюдаемые значения для каждого размера данных должны быть усреднены.

Весьма часто эмпирический анализ эффективности требует генерации случайных чисел. Даже используя для входных данных шаблон, мы обычно хотим, чтобы сами экземпляры данных генерировались случайным образом. Генерация случайных чисел на цифровом компьютере, как известно, представляет собой сложную задачу, которая в принципе может быть решена только приближенно. В этом заключается причина того, что кибернетики предпочитают именовать такие числа ***псевдослучайными***. С практической точки зрения простейший и наиболее естественный способ получения таких чисел состоит в использовании генератора случайных чисел из библиотеки используемого вами языка программирования. Обычно такой генератор дает на выходе (псевдо)случайные значения, равномерно распределенные в интервале от 0 до 1. Если требуются некоторые другие (псевдо)случайные значения, можно применить соответствующие преобразования. Например, если x — непрерывная случайная величина, равномерно распределенная в диапазоне $0 \leqslant x < 1$, то величина $y = l + \lfloor x(r - l) \rfloor$ будет представлять собой целую величину, равномерно распределенную в диапазоне между целыми числами l и $r - 1$ ($l < r$).

Вы можете не пользоваться библиотечным генератором, а реализовать один из множества известных алгоритмов для генерации (псевдо)случайных чисел. Наиболее широко используемым и детально изученным среди таких алгоритмов является так называемый линейный конгруэнтный метод (*linear congruential method*).

АЛГОРИТМ $Random(n, m, seed, a, b)$

```
// Генерирует последовательность из n псевдослучайных чисел
// с использованием линейного конгруэнтного метода
// Входные данные:  Натуральное число n и натуральные
//                  параметры m, seed, a, b
// Выходные данные: последовательность r_1, ..., r_n
//                  псевдослучайных чисел, равномерно
```

```
//                    распределенных между 0 и m − 1
// Примечание:        Псевдослучайные числа между 0 и 1 могут
//                    быть получены путем рассмотрения
//                    генерируемых алгоритмом целых чисел как
//                    цифр после десятичной точки
r_0 ← seed
for i ← 1 to n do
    r_i ← (a * r_{i−1} + b) mod m
```

Простота приведенного алгоритма обманчива, поскольку все сложности скрыты в выборе параметров алгоритма. Вот неполный список рекомендаций, основанных на результатах сложного математического анализа (см. [66]): значение *seed* можно выбрать произвольным образом (зачастую для этого используются текущая дата и время); m должно быть большим (может оказаться удобным выбор для m величины 2^w, где w — размер слова компьютера); a должно быть целым числом от $0.01m$ до $0.99m$ без определенной закономерности в его цифрах, но такое, что $a \bmod 8 = 5$; значение b может быть выбрано равным 1.

Эмпирические данные, полученные в ходе эксперимента, должны быть записаны, а затем представлены для дальнейшего анализа. Данные могут быть представлены в таблице или графически, точками в декартовой системе координат. Неплохая мысль использовать одновременно оба способа, поскольку для каждого из них характерны свои сильные и слабые стороны.

Основное преимущество табулированных данных заключается в легкости доступа и работы с ними. Например, мы можем вычислить отношения $M(n)/g(n)$, где $g(n)$ — кандидат в представители класса эффективности исследуемого алгоритма. Если алгоритм действительно принадлежит $\Theta(g(n))$, то, вероятнее всего, эти отношения будут сходиться к некоторой положительной константе при возрастании n. (Заметим, что некоторые невнимательные экспериментаторы-новички полагают, что эта константа должна быть равна единице, что, конечно же, неверно в силу определения $\Theta(g(n))$). Мы можем также вычислить отношения $M(2n)/M(n)$ и посмотреть, как ведет себя время работы алгоритма при удвоении размера входных данных. Как было показано в разделе 2.2, в случае логарифмического алгоритма это отношение должно изменяться очень слабо, а для линейного, квадратичного и кубического алгоритмов — сходиться к 2, 4 и 8, соответственно.

С другой стороны, графическое представление данных также может помочь в выдвижении гипотезы о вероятном классе эффективности алгоритма. В случае логарифмического алгоритма график имеет выпуклый вверх вид (рис. 2.7*а*). Этот вид графика отличает логарифмические алгоритмы от всех прочих алгоритмов. В случае линейного алгоритма экспериментальные точки имеют тенденцию выстраиваться вдоль обычной прямой линии или, вообще говоря, располагаться между двумя прямыми линиями (рис. 2.7*б*). Графики функций из $\Theta(n \lg n)$

и $\Theta(n^2)$ имеют выпуклый вниз вид (рис. 2.7*в*), что делает их идентификацию более сложной. График для кубического алгоритма также имеет выпуклый вниз вид, но растет с существенно более высокой скоростью. В случае экспоненциального алгоритма вертикальная ось требует использования логарифмической шкалы, т.е. на график следует наносить точки $\log_a M(n)$ вместо $M(n)$ (наиболее часто применяемые основания логарифмов — 2 и 10). В такой системе координат график истинной экспоненциальной функции представляет собой прямую линию, поскольку из $M(n) \approx ca^n$ следует $\log_b M(n) \approx \log_b c + n \log_b a$.

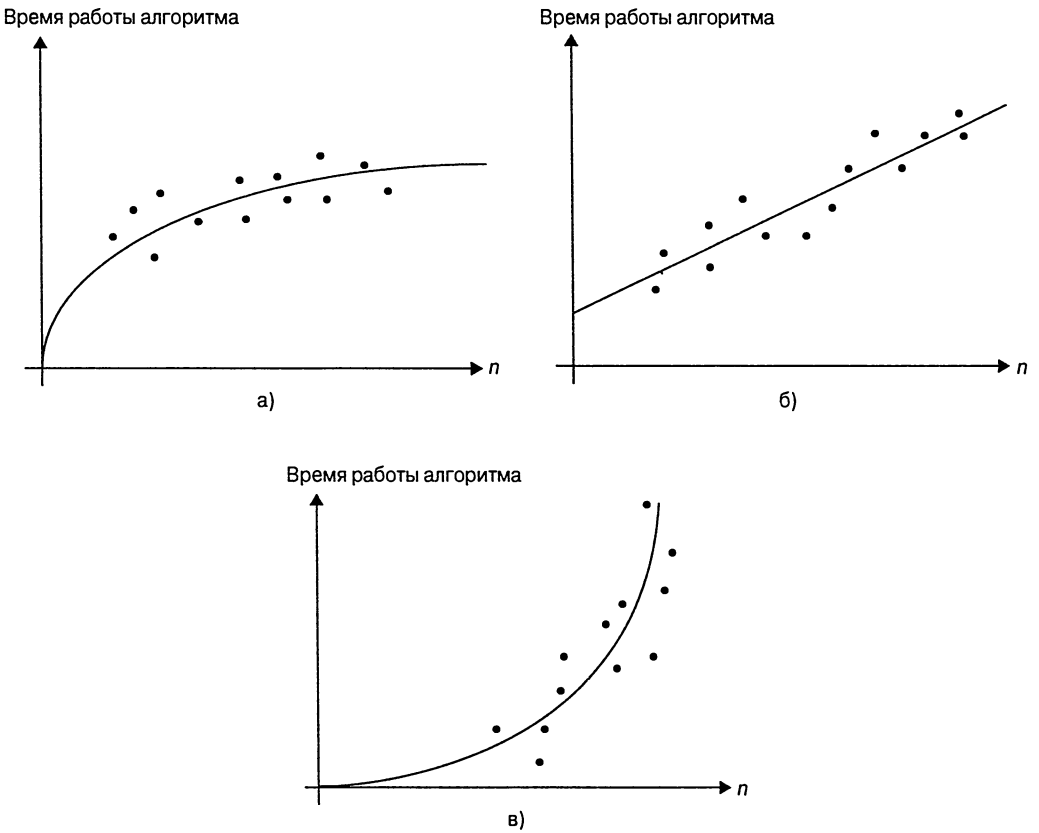


Рис. 2.7. Типичные графики функций: а) логарифмической, б) линейной, в) выпуклой вниз функции

Одно из возможных применений эмпирического анализа — попытка предсказать производительность алгоритма для экземпляра исходных данных, не включенного во множество экземпляров исходных данных эксперимента. Например, если мы заметим, что отношение $M(n)/g(n)$ близко к некоторой константе c для экземпляров, использованных в эксперименте, то мы можем аппроксимировать $M(n)$ произведением $cg(n)$ и для других значений n. Хотя этот подход вполне

разумен, его следует использовать с осторожностью, в особенности для значений n, которые находятся вне экспериментально исследованного диапазона. (Математики называют такое предсказание ***экстраполяцией*** (***extrapolation***) в отличие от ***интерполяции*** (***interpolation***), которая работает со значениями в пределах исследуемого диапазона.) В частности, нельзя ничего сказать о точности таких оценок. Конечно, можно попытаться применить стандартные методы статистического анализа данных, однако заметим, что они в основном базируются на определенных вероятностных предположениях, которые могут оказаться неверны для рассматриваемых экспериментальных данных.

Представляется уместным завершить данный подраздел перечислением основных отличий между математическим и эмпирическим анализом алгоритмов. Главное преимущество математического анализа — его независимость от конкретных входных данных, а недостаток — ограниченная применимость, в особенности для исследования эффективности в среднем случае. Эмпирический анализ, напротив, применим к любому алгоритму, но его результаты могут зависеть от конкретных входных данных и использованного для проведения эксперимента компьютера.

Упражнения 2.6

1. Рассмотрим хорошо известный алгоритм сортировки (чуть позже мы более детально познакомимся с ним), в который вставлен счетчик количества сравнений ключей.

```
Алгоритм SortAnalysis(A[0..n − 1])
  // Входные данные:  Массив A[0..n − 1] из n упорядочиваемых
  //                  элементов
  // Выходные данные: Общее количество выполненных сравнений
  //                  ключей
  count ← 0
  for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ⩾ 0 and A[j] > v do
      count ← count + 1
      A[j + 1] ← A[j]
      j ← j − 1
    A[j + 1] ← v
  return count
```

Правильно ли размещен счетчик? Если да, докажите это; если нет, внесите необходимые изменения в код.

2. **а)** Выполните программу из упражнения 1 с корректно вставленным счетчиком (или счетчиками) и определите количество сравнений для 20 случайных массивов размером 1000, 1500, 2000, 2500, ..., 9000, 9500.

   б) Проанализируйте полученные данные и выдвиньте гипотезу об эффективности рассматриваемого алгоритма в среднем случае.

   в) Оцените количество сравнений ключей, которые следует ожидать при работе рассматриваемого алгоритма со случайным образом сгенерированным массивом из 10 000 элементов.

3. Выполните упражнение 2, но на этот раз измеряйте не количество сравнений, а время работы программы в миллисекундах.

4. Выдвиньте гипотезу о классе эффективности алгоритма на основании следующих эмпирических подсчетов количества базовых операций:

Размер	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Количество	11966	24303	39992	53010	67272	78692	91274	113063	129799	140538

5. Какое масштабирующее преобразование сделает логарифмический график линейным?

6. Как можно отличить график алгоритма из $\Theta(\lg \lg n)$ от графика алгоритма $\Theta(\lg n)$?

7. **а)** Найдите эмпирически наибольшее количество делений, которые алгоритм Евклида выполняет при вычислении $\gcd(m, n)$ при $1 \leqslant n \leqslant m \leqslant 100$.

   б) Для каждого натурального k эмпирически найдите наименьшую пару целых чисел $1 \leqslant n \leqslant m \leqslant 100$, для которых алгоритм Евклида требует выполнения k делений при поиске $\gcd(m, n)$.

8. Эффективность алгоритма Евклида в среднем случае для входных данных размером n можно измерить как количество делений $D_{avg}(n)$, выполняемых алгоритмом при вычислении $\gcd(n, 1)$, $\gcd(n, 2)$, ..., $\gcd(n, n)$. Например,

$$D_{avg}(5) = \frac{1}{5}(1 + 2 + 3 + 2 + 1) = 1.8.$$

   Постройте график функции $D_{avg}(n)$ и укажите класс эффективности алгоритма в среднем случае.

9. Выполните эксперимент для выяснения класса эффективности решета Эратосфена (см. раздел 1.1).

10. Выполните эксперименты по исследованию времени работы трех алгоритмов вычисления $\gcd(m, n)$, представленных в разделе 1.1.

Визуализация алгоритмов

Помимо математического и эмпирического анализа имеется еще один путь изучения алгоритмов. Он называется ***визуализацией алгоритма*** и может быть определен как использование изображений для передачи некоторой полезной информации об алгоритмах. Эта информация может быть визуальной иллюстрацией действий, выполняемых алгоритмом, или его производительности для разных входных данных, либо его скорости выполнения по сравнению с другими алгоритмами для решения той же задачи. Для достижения данной цели визуализация алгоритма использует графические элементы (точки, отрезки, прямоугольники или параллелепипеды и т.д.) для представления некоторых "интересных событий" в работе алгоритма. Имеются два основных варианта визуализации алгоритма:

- статическая визуализация
- динамическая визуализация, именуемая также ***анимацией алгоритма***.

Статическая визуализация алгоритма представляет выполнение алгоритма посредством серии изображений. Анимация алгоритма, в свою очередь, использует непрерывную презентацию действий алгоритма в стиле мультфильма. Анимация — более привлекательный выбор, но, конечно, и существенно сложнее в реализации.

Первые попытки визуализации алгоритмов относятся к 1970-м годам. Переломным моментом стало появление в 1981 году классической визуализации алгоритма — тридцатиминутного цветного звукового фильма *Сортировка за сортировкой*. Фильм был создан в университете Торонто Рональдом Беккером (Ronald Baecker) при участии Д. Шермана (D. Sherman) [9, 10]. Фильм содержал визуализацию девяти популярных алгоритмов сортировки (более половины из них будут рассмотрены позже в данной книге) и предоставлял убедительную демонстрацию их относительных скоростей.

Успех фильма *Сортировка за сортировкой* сделал алгоритмы сортировки несомненными фаворитами анимации алгоритмов. В самом деле, проблема сортировки естественным образом визуализируется с использованием вертикальных или горизонтальных прямоугольников различной высоты или длины, которые расставляются в соответствии с их размерами (рис. 2.8). Такое представление, однако, удобно лишь для иллюстрации работы алгоритмов сортировки при небольших входных данных. Для входных данных большего размера в фильме *Сортировка за сортировкой* использована остроумная идея представить данные точками на плоскости, где первая координата точки представляет позицию элемента в массиве данных, а вторая — его значение. При использовании такого представления

данных сортировка представляет собой трансформацию случайного заполнения точками в точки, лежащие на диагонали кадра (рис. 2.9). Кроме того, большинство алгоритмов сортировки работает путем поочередного сравнения и обмена двух элементов — события, которое относительно просто визуализировать.

После появления *Сортировки за сортировкой* было создано множество различных анимаций алгоритмов. Анимации могут содержать как один алгоритм, так и группу алгоритмов для решения одной и той же задачи (например, сортировки) или из одной и той же предметной области (например, геометрические алгоритмы), а также представлять собой анимационные системы общего назначения. Наиболее популярные системы общего назначения включают BALSA [24], TANGO [112] и ZEUS [23]; сравнительный обзор возможностей этих (и еще девяти других) пакетов можно найти в [91]. Хорошая анимационная система общего назначения должна позволять пользователю не только просматривать и взаимодействовать с имеющимися анимациями, но и позволять создавать новые анимации. Практика показывает, что создание таких систем — сложная, но не невозможная задача.

Появление Java и World Wide Web дало новый толчок развитию анимации алгоритмов. Рекомендую обратиться к Internet и познакомиться с образцами анимации алгоритмов. Поскольку мир Web очень нестабилен, здесь не приводятся конкретные адреса — попробуйте осуществить поиск по словам "algorithm animation" или "algorithm visualization". При рассмотрении и оценке различных анимаций алгоритмов можно пользоваться приведенными ниже "десятью заповедями анимации алгоритмов" — списком желательных возможностей пользовательского интерфейса, предложенным Питером Глуром (Peter Gloor) [42], главным разработчиком Animated Algorithms — еще одной популярной системы визуализации алгоритмов.

1. Последовательность.
2. Интерактивность.
3. Ясность и краткость.
4. Снисходительность к пользователю и прощение его ошибок.
5. Адаптация к уровню знаний пользователя.
6. Упор на визуальную часть.
7. Удержание интереса пользователя.
8. Включение символьного и пиктограммного представления.
9. Включение анализа алгоритма (статистики выполнения) и сравнение с другими алгоритмами для решения той же задачи.
10. Включение истории выполнения.

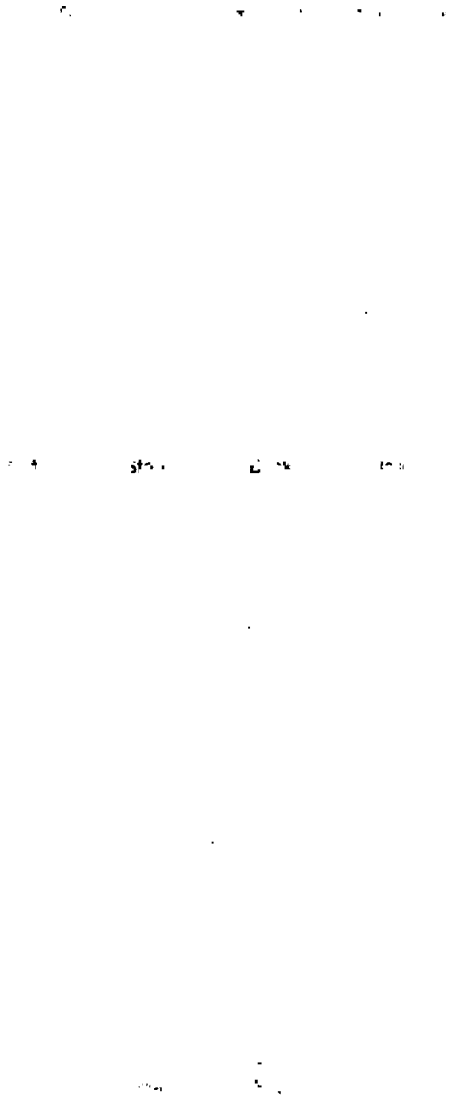


Рис. 2.8. Начальный и конечный экраны типичной визуализации алгоритма сортировки с использованием прямоугольников

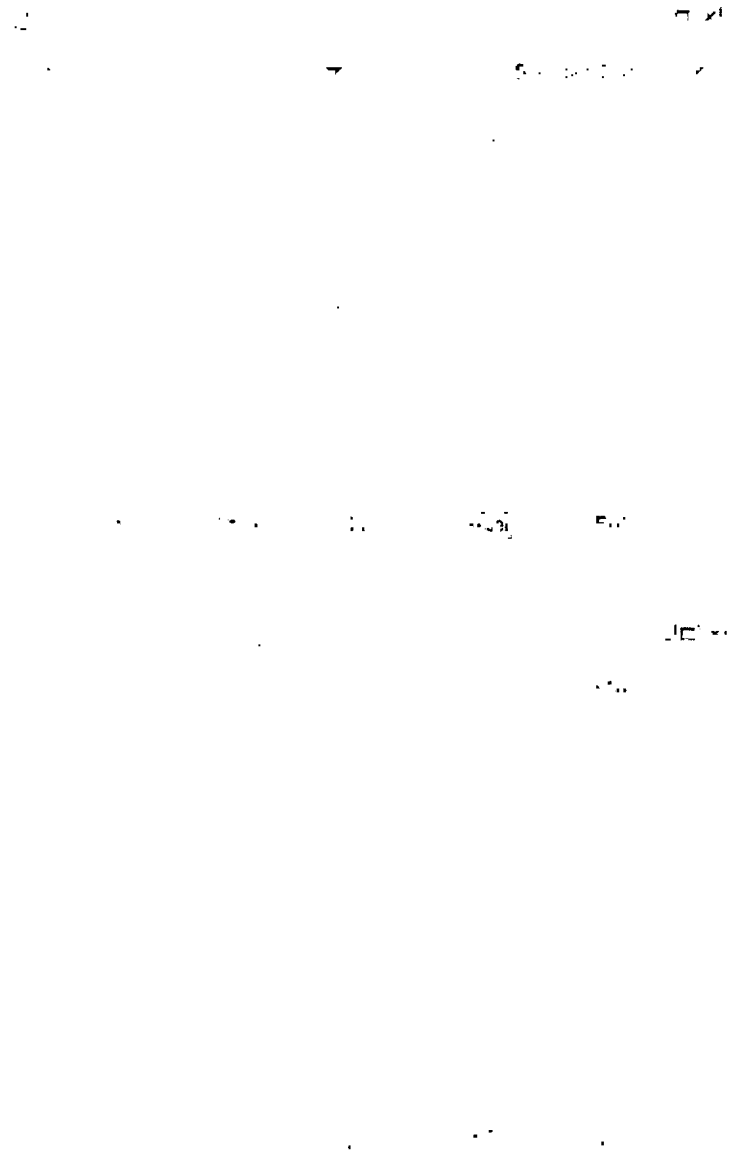


Рис. 2.9. Начальный и конечный экраны типичной визуализации алгоритма сортировки с использованием точек

Основные области применения визуализации алгоритмов — научные исследования и обучение. Применение визуализации алгоритмов в образовании призвано помочь студентам в изучении алгоритмов. Потенциальная польза визуализации в исследовательской работе заключается в возможности открытия некоторых пока что не известных свойств алгоритма. Например, один исследователь использовал визуализацию рекурсивного алгоритма для решения задачи о ханойских башнях, в которой четные и нечетные диски имели разные цвета. Он обратил внимание на то, что два диска одного цвета никогда не находятся в непосредственном контакте в процессе выполнения алгоритма. Это наблюдение помогло ему разработать лучшую нерекурсивную версию классического алгоритма. Однако, хотя и имеется масса сообщений об успешном применении визуализации в образовании и научных исследованиях, эти успехи не столь впечатляющи, как можно было бы ожидать. Опыт показывает, что создания сложной программной системы недостаточно — требуется глубокое понимание восприятия визуальной информации человеком, чтобы добиться полного раскрытия потенциала анимации алгоритма.

Резюме

- Различают два вида эффективности алгоритмов — временну́ю и пространственную. *Временна́я эффективность* указывает, насколько быстро выполняется алгоритм; *пространственная эффективность* показывает, как много дополнительной памяти требуется алгоритму для работы.
- Временна́я эффективность алгоритма в основном определяется как функция от размера входных данных, путем подсчета количества выполнения базовых операций. *Базовая операция* — это операция, которая вносит основной вклад в общее время выполнения алгоритма. Обычно это операция с наибольшим временем работы в наиболее глубоко вложенном цикле.
- Для ряда алгоритмов время работы существенно отличается для разных входных данных одного и того же размера, что приводит к эффективности для *наихудшего*, *наилучшего* и *среднего случаев*.
- Разработана схема анализа временно́й эффективности алгоритма, которая основывается на порядке роста времени работы алгоритма при росте размера его входных данных до бесконечности.
- Для указания и сравнения асимптотических порядков роста функций, выражающих эффективность алгоритмов, используются O-, Ω- и Θ-обозначения.

- Эффективность подавляющего большинства алгоритмов подразделяется на несколько классов — *константную*, *логарифмическую*, *линейную*, $n \log n$, *квадратичную*, *кубическую* и *экспоненциальную*.
- Главным инструментом анализа временно́й эффективности нерекурсивного алгоритма является построение выражения для суммы количества выполнений его основной операции и выяснение порядка ее роста.
- Главным инструментом анализа временно́й эффективности рекурсивного алгоритма является построение рекуррентного соотношения для количества выполнений его основной операции и выяснение порядка его роста.
- Лаконичность рекурсивного алгоритма может скрывать его неэффективность.
- *Числа Фибоначчи* представляют собой важную последовательность целых чисел, в которой каждый элемент равен сумме двух своих предшественников. Имеется несколько алгоритмов вычисления чисел Фибоначчи с существенно различной эффективностью.
- Эмпирический анализ алгоритма осуществляется путем выполнения программы, реализующей алгоритм, для некоторого образца входных данных и анализа полученных результатов (количества выполненных базовых операций или физического времени работы программы). Зачастую при этом используются псевдослучайные числа. Главным преимуществом этого подхода является его применимость к любым алгоритмам, а недостатком — зависимость результатов эксперимента от конкретного компьютера и образца исходных данных.
- *Визуализация алгоритма* представляет собой использование изображений для вывода полезной информации об алгоритме. Два основных варианта визуализации алгоритма — статическая визуализация и динамическая визуализация (именуемая также анимацией алгоритма).

Глава 3

Метод грубой силы

Наука так же далека от грубой силы, как этот меч от лома.

— Эдуард Литтон (Edward Lytton) (1803-1873)
Лейла (Leila), книга II, глава I

Делать что-либо хорошо — зачастую понапрасну терять время.

— Роберт Бирн (Robert Byrne),
игрок в бильярд и писатель

Обсудив схемы и методы анализа алгоритмов в предыдущих главах книги, мы готовы перейти к рассмотрению методов разработки алгоритмов. Каждая из последующих семи глав посвящена некоторой конкретной стратегии разработки. Тема данной главы — простейший метод грубой силы, именуемый также методом решения "в лоб". Этот метод можно описать следующим образом.

> **Метод грубой силы** представляет собой прямой подход к решению задачи, обычно основанный непосредственно на формулировке задачи и определениях используемых ею концепций.

"Сила" в определении стратегии — сила компьютера, а не сила интеллекта, т.е. сила из пословицы "Сила есть — ума не надо". Перефразировать определение данной стратегии можно проще: "Нечего думать, надо действовать!". Зачастую стратегия грубой силы оказывается наиболее простой в применении.

В качестве примера рассмотрим задачу возведения в степень: вычисление a^n для некоторого числа a и неотрицательного целого n. Хотя эта задача может показаться тривиальной, она позволяет проиллюстрировать несколько методов разработки алгоритмов, в том числе и подход грубой силы. (Заметим в скобках, что вычисление значения $a^n \bmod m$ для больших целых чисел является основным компонентом ведущих алгоритмов шифрования.) По определению возведения в степень

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ раз}}.$$

Отсюда сразу следует простейший алгоритм вычисления a^n — путем умножения на a начального значения, равного 1, n раз.

Мы уже встречались в этой книге как минимум с двумя алгоритмами с использованием грубой силы: последовательная проверка всех целых чисел при поиске $\gcd(m, n)$ (раздел 1.1) и алгоритм умножения матриц, основанный на определении данной операции (раздел 2.3). Множество других примеров вы найдете в этой главе. (Можете ли вы указать несколько известных вам алгоритмов, основанных на использовании грубой силы?)

Хотя метод грубой силы редко дает искусные или эффективные алгоритмы, его рассмотрение нельзя опустить, поскольку данный метод представляет собой важную стратегию разработки алгоритмов. Во-первых, в отличие от других стратегий, метод грубой силы применим к очень широкому диапазону задач. (Похоже, это единственный подход, для которого существенно сложнее указать задачу, для решения которой он *неприменим*.) В частности, метод грубой силы используется для многих элементарных, но важных алгоритмических задач, таких как вычисление суммы n чисел, поиск наибольшего элемента в списке и тому подобных. Во-вторых, для некоторых важных задач (например, сортировки, поиска, умножения матриц, поиска подстрок) метод грубой силы дает вполне рациональные алгоритмы. В-третьих, стоимость разработки более эффективного алгоритма может оказаться неприемлемой, если требуется решить только несколько экземпляров задачи, а алгоритм, основанный на грубой силе, позволяет решить их за приемлемое время. В-четвертых, даже будучи неэффективным в общем случае, метод грубой силы может оказаться полезен для решения небольших по размеру экземпляров задачи. Наконец, алгоритм, основанный на грубой силе, может служить для важных теоретических или дидактических целей, например мерилом для определения эффективности других алгоритмов для решения данной задачи.

3.1 Сортировка выбором и пузырьковая сортировка

В этом разделе мы рассмотрим применение метода грубой силы к задаче сортировки, которая заключается в следующем: дан список из n упорядочиваемых элементов (например, чисел, символов некоторого алфавита, символьных строк), которые надо разместить в неубывающем порядке. Как упоминалось в разделе 1.3, имеются десятки алгоритмов, разработанных для решения этой очень важной задачи. Некоторые из них вам уже, возможно, встречались. В таком случае попытайтесь на время забыть о них.

Итак, считая, что вы ничего не знаете о сортировке, задайте себе вопрос: какой метод решения задачи сортировки наиболее прост и непосредственен? Кое-кто может не согласиться, но наиболее простыми представляются два алгоритма, рассматриваемые далее, а именно сортировка выбором и пузырьковая сортировка.

Первый из этих алгоритмов кажется более предпочтительным, хотя бы потому, что более очевидно использует метод грубой силы.

Сортировка выбором

Мы начинаем сортировку выбором с поиска наименьшего элемента в списке и обмена его с первым элементом (таким образом, наименьший элемент помещается в окончательную позицию в отсортированном списке). Затем мы сканируем список, начиная со второго элемента, в поисках наименьшего среди оставшихся $n-1$ элементов и обмениваем найденный наименьший элемент со вторым, т.е. помещаем второй наименьший элемент в окончательную позицию в отсортированном списке. В общем случае, при i-ом проходе по списку ($0 \leqslant i \leqslant n-2$) алгоритм ищет наименьший элемент среди последних $n-i$ элементов и обменивает его с A_i:

$$\underbrace{A_0 \leq A_1 \leq \cdots \leq A_{i-1}}_{\text{Элементы в окончательных позициях}} \mid \overbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}^{\text{Последние } n-i \text{ элементов}}$$

После выполнения $n-1$ проходов список оказывается отсортирован.

Вот псевдокод данного алгоритма, в котором для простоты предполагается, что список реализован в виде массива.

Алгоритм *SelectionSort* $(A[0..n-1])$

```
// Сортировка массива методом выбора
// Входные данные:  Массив A[0..n − 1] упорядочиваемых
//                  элементов
// Выходные данные: Массив A[0..n − 1], отсортированный
//                  в неубывающем порядке
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]
            min ← j
    Обмен A[i] и A[min]
```

В качестве примера на рис. 3.1 приведена сортировка выбором следующего списка: 89, 45, 68, 90, 29, 34, 17.

Анализ сортировки выбором выполняется достаточно просто. Размер входных данных определяется количеством n элементов в списке. Базовой операцией алгоритма является сравнение ключей $A[j] < A[min]$. Общее количество сравнений зависит только от размера массива и определяется следующей суммой:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

89	45	68	90	29	34	**17**
17	| 45	68	90	**29**	34	89
17	29	| 68	90	45	**34**	89
17	29	34	| 90	**45**	68	89
17	29	34	45	| 90	**68**	89
17	29	34	45	68	| 90	**89**
17	29	34	45	68	89	| 90

Рис. 3.1. Сортировка выбором списка 89, 45, 68, 90, 29, 34, 17. Каждая строка соответствует одной итерации алгоритма, т.е. сканированию части списка справа от вертикальной черты. Полужирным шрифтом выделены наименьшие элементы, обнаруживаемые при сканировании. Элементы справа от вертикальной черты находятся в окончательных позициях и не сканируются

Мы уже встречались с последней суммой при анализе алгоритма из примера 2 в разделе 2.3 (так что вы можете вычислить ее самостоятельно). Каким бы способом вы ни вычисляли данную сумму, ответ, конечно, будет одним и тем же, а именно

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{(n-1)n}{2}.$$

Таким образом, для любых входных данных алгоритм сортировки выбором принадлежит $\Theta(n^2)$. Заметим, однако, что количество обменов элементов массива равно $\Theta(n)$, точнее, ровно $n-1$ — по одному для каждой итерации цикла i. Это свойство отличает сортировку выбором от многих других алгоритмов сортировки.

Пузырьковая сортировка

Еще одно применение метода грубой силы к задаче сортировки состоит в сравнении соседних элементов и их обмене, если они находятся не в надлежащем порядке. Неоднократно выполняя это действие, мы заставляем наибольший элемент "всплывать" к концу списка. Следующий проход приведет к всплыванию второго наибольшего элемента, и так до тех пор, пока после $n-1$ итерации список не будет полностью отсортирован. i-ый проход $0 \leqslant i \leqslant n-2$) пузырьковой сортировки можно представить следующей диаграммой:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leq \cdots \leq A_{n-1}$$

Элементы в окончательных позициях

Ниже приведен псевдокод данного алгоритма.

Алгоритм *BubbleSort* $(A[0..n-1])$
// Сортировка массива пузырьковой сортировкой
// **Входные данные:** Массив $A[0..n-1]$ упорядочиваемых
// элементов
// **Выходные данные:** Массив $A[0..n-1]$, отсортированный
// в неубывающем порядке
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$
      Обмен $A[j]$ и $A[j+1]$

Применение описываемого алгоритма к списку 89, 45, 68, 90, 29, 34, 17 показано на рис. 3.2.

89	? ↔	45		68		90		29		34		17	
45		89	? ↔	68		90		29		34		17	
45		68		89	? ↔	90	? ↔	29		34		17	
45		68		68		29		90	? ↔	34		17	
45		68		89		29		34		90	? ↔	17	
45		68		89		29		34		17		|90	
45	? ↔	68	? ↔	89	? ↔	29		34		17		|90	
45		68		29		89	? ↔	34		17		|90	
45		68		29		34		89	? ↔	17		|90	
45		68		29		34		17		89		|90	

Рис. 3.2. Два первых прохода пузырьковой сортировки списка 89, 45, 68, 90, 29, 34, 17. Каждая новая строка представляет собой результат обмена двух элементов. Элементы справа от вертикальной черты находятся в окончательных позициях и в последующих итерациях алгоритма не участвуют

Количество сравнений ключей в данной версии пузырьковой сортировки одинаково для всех массивов размером n. Оно представляется суммой, практически идентичной сумме для сортировки выбором:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] =$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)\,n}{2} \in \Theta\left(n^2\right).$$

Количество же обменов элементов зависит от входных данных. В наихудшем случае уменьшающегося массива оно равно количеству сравнений:

$$S_{worst}(n) = C(n) = \frac{(n-1)\,n}{2} \in \Theta\left(n^2\right).$$

Зачастую при использовании метода грубой силы первая версия алгоритма может быть усовершенствована ценой достаточно скромных усилий. В частности, приведенную сырую версию пузырьковой сортировки можно улучшить, воспользовавшись следующим наблюдением: если при проходе по списку не сделано ни одного обмена, значит, список отсортирован, и выполнение алгоритма можно прекратить (задача 3.1.9*а*). Однако, хотя новая версия и выполняется быстрее для некоторых входных данных, в наихудшем и среднем случае она все равно принадлежит $\Theta\left(n^2\right)$. В действительности даже среди элементарных методов сортировки пузырьковая сортировка — не лучший выбор, и если бы не броское название, вы бы могли никогда о ней не услышать. Тем не менее из ее рассмотрения мы вынесли важный урок:

> Первое применение метода грубой силы зачастую дает алгоритм, который можно улучшить ценой весьма скромных усилий.

Упражнения 3.1

1. **а)** Приведите пример алгоритма, который нельзя рассматривать как применение метода грубой силы.
   б) Приведите пример задачи, которую нельзя решить при помощи алгоритма, основанного на грубой силе.
2. **а)** Чему равна эффективность алгоритма вычисления a^n, основанного на методе грубой силы, как функция от n? А как функция количества бит в двоичном представлении числа n?
   б) Если вы вычисляете $a^n \bmod m$, где $a > 1$, а n — большое натуральное число, то как вы справитесь с проблемой очень больших значений a^n?
3. Использует ли каждый из алгоритмов задач 4–6 упражнений 2.3 метод грубой силы?
4. **а)** Разработайте алгоритм для вычисления значения полинома с помощью метода грубой силы

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

в заданной точке x_0 и определите его класс эффективности в наихудшем случае.

б) Если ваш алгоритм принадлежит множеству $\Theta(n^2)$, постарайтесь разработать алгоритм с линейным временем работы.

в) Можно ли разработать алгоритм с более высокой эффективностью, чем линейная?

5. Отсортируйте список букв E, X, A, M, P, L, E в алфавитном порядке при помощи сортировки выбором.

6. Устойчива ли сортировка выбором? (Определение устойчивого алгоритма сортировки было дано в разделе 1.3.)

7. Можно ли реализовать сортировку выбором для связанных списков с той же эффективностью $\Theta(n^2)$, что и для массивов?

8. Отсортируйте список букв E, X, A, M, P, L, E в алфавитном порядке при помощи пузырьковой сортировки.

9. а) Докажите, что если пузырьковая сортировка не делает ни одного обмена при проходе по списку, то список отсортирован и алгоритм можно завершать.

б) Напишите псевдокод улучшенного таким образом алгоритма.

в) Докажите, что в худшем случае алгоритм имеет квадратичную эффективность.

10. Устойчива ли пузырьковая сортировка?

3.2 Последовательный поиск и поиск подстрок методом грубой силы

В предыдущем разделе мы познакомились с двумя вариантами применения метода грубой силы к задаче сортировки. Теперь рассмотрим два возможных применения этой стратегии к задачам поиска. Первая — каноническая задача поиска элемента с данным значением, вторая — задача поиска подстроки.

Последовательный поиск

Мы уже сталкивались с алгоритмом, основанном на методе грубой силы, для решения общей задачи поиска — он называется последовательным поиском (см. раздел 2.1). Этот алгоритм просто по очереди сравнивает элементы заданного списка с ключом поиска до тех пор, пока не будет найден элемент с указанным значением ключа (успешный поиск) или весь список будет проверен, но требуемый элемент не найден (неудачный поиск). Зачастую применяется простой

дополнительный прием: если добавить ключ поиска в конец списка, то поиск обязательно будет успешным, следовательно, можно убрать проверку завершения списка в каждой итерации алгоритма. Далее приведен псевдокод такой улучшенной версии; предполагается, что входные данные имеют вид массива.

Алгоритм $SequentialSearch2\,(A\,[0..n]\,, K)$

```
// Алгоритм реализует последовательный поиск
// с использованием ключа поиска в качестве ограничителя
// Входные данные:  Массив A из n элементов и ключ поиска K
// Выходные данные: Позиция первого элемента массива
//                  A[0..n − 1], значение которого совпадает
//                  с K; если элемент не найден, возвращается
//                  значение −1
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n
    return i
else
    return −1
```

Если исходный список отсортирован, можно воспользоваться еще одним усовершенствованием: поиск в таком списке можно прекращать, как только встретится элемент, не меньший ключа поиска.

Последовательный поиск представляет превосходную иллюстрацию метода грубой силы, с его характерными сильными (простота) и слабыми (низкая эффективность) сторонами. Эффективность, вычисленная в разделе 2.1 для стандартной версии последовательного поиска, очень мало отличается от эффективности улучшенной версии последовательного поиска, так что алгоритм остается линейным как в наихудшем, так и в среднем случае. Позже мы познакомимся с некоторыми алгоритмами, имеющими более высокую эффективность в среднем случае.

Поиск подстроки

Вспомним описание задачи поиска подстроки из раздела 1.3: дана символьная строка длиной n, называющаяся ***текстом***, и строка длиной m ($m \leqslant n$), именуемая ***шаблоном*** (pattern); требуется найти в тексте подстроку, соответствующую шаблону. Говоря точнее, мы хотим определить i — индекс крайнего слева символа первой соответствующей шаблону подстроки в тексте — такой, что $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{ccccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & & \text{Текст } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & & & & \text{Шаблон } P \end{array}$$

Если требуется найти все такие подстроки, алгоритм поиска подстроки может просто продолжать работу до полной проверки текста.

Алгоритм, основанный на грубой силе, очевиден: выровняем шаблон с началом текста и начнем сравнивать соответствующие пары символов слева направо до тех пор, пока не убедимся, что символы во всех m парах равны (в этом случае алгоритм может прекращать работу), либо не встретим пару разных символов. В последнем случае шаблон смещается на одну позицию вправо, и сравнение символов продолжается, начиная с первого символа шаблона и символа в соответствующей позиции в тексте. Заметим, что последняя позиция в тексте, которая еще может выступать в роли начала искомой подстроки — $n - m$ (если позиции в тексте индексируются значениями от 0 до $n - 1$). После этой позиции в тексте остается слишком мало символов, чтобы они могли соответствовать шаблону. Следовательно, по достижении указанной позиции алгоритм не должен делать никаких сравнений.

Алгоритм $BruteForceStringMatch\,(T\,[0..n-1]\,, P\,[0..m-1])$

```
// Алгоритм реализует поиск подстроки методом грубой силы
// Входные данные:  массив T[0..n − 1] из n символов,
//                  представляющий текст;
//                  массив P[0..m − 1] из m символов,
//                  представляющий шаблон
// Выходные данные: позиция первого символа в тексте,
//                  с которой начинается первая искомая
//                  подстрока, соответствующая шаблону; если
//                  подстрока не найдена, возвращается −1
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m
        return i
return −1
```

Работа алгоритма проиллюстрирована на рис. 3.3.

Обратите внимание, что в данном примере алгоритм практически всегда смещает шаблон после первого же сравнения символа. Однако наихудший случай намного неприятнее: алгоритм может выполнять все m сравнений перед сдвигом шаблона, и это происходит в каждой из $n - m + 1$ попыток (в задаче 3.2.6

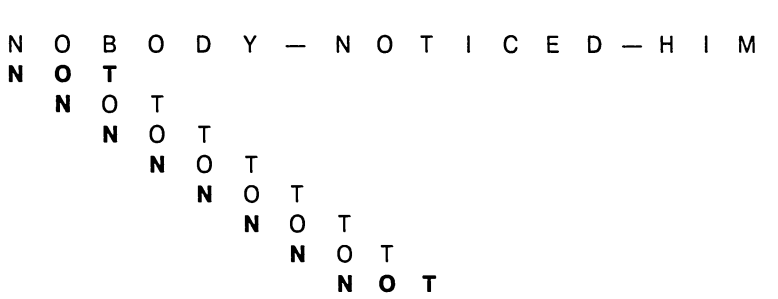


Рис. 3.3. Пример поиска подстроки с применением грубой силы. Символы шаблона, сравниваемые с символами текста, выделены полужирным шрифтом

требуется привести пример такой ситуации). Таким образом, в наихудшем случае алгоритм имеет время работы $\Theta(nm)$. Однако в случае типичного поиска слова в тексте на естественном языке можно ожидать, что большинство сдвигов будет выполняться после небольшого количества сравнений (взгляните на приведенный пример). Таким образом, эффективность в среднем случае должна быть существенно выше эффективности в худшем случае. Это так и есть на самом деле: можно показать, что при поиске в случайных текстах эффективность оказывается линейной, т.е. равной $\Theta(n+m) = \Theta(n)$. Для поиска подстрок имеется масса более интеллектуальных и эффективных алгоритмов. Наиболее широко известный из них — алгоритм Р. Бойера (R. Boyer) и Дж. Мура (J. Moore) — будет рассмотрен в разделе 7.2 вместе с упрощением, предложенным Р. Хорспулом (R. Horspool).

Упражнения 3.2

1. Найдите количество сравнений, выполняемых версией последовательного поиска с ограничителем
   а) в наихудшем случае;
   б) в среднем случае при вероятности успешного поиска p $(0 \leqslant p \leqslant 1)$.
2. Как было показано в разделе 2.1, среднее количество сравнений ключей, выполняемых при последовательном поиске (без ограничителя, при стандартных предположениях о входных данных), определяется формулой
$$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$$
где p — вероятность успешного поиска. Для фиксированного n определите значения p $0 \leqslant p \leqslant 1)$, для которых формула дает наибольшее и наименьшее значения $C_{avg}(n)$.

3. Разработайте и выполните эксперимент по сравнению времени работы двух версий последовательного поиска, описанных в разделах 2.1 и 3.2.

4. Определите количество сравнений символов, которые будут выполнены алгоритмом, основанном на грубой силе, при поиске шаблона `GANDHI` в тексте

   `THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED`

   (Будем считать, что длина текста — 47 символов — известна до начала поиска.)

5. Сколько сравнений (успешных и неудачных) сделает алгоритм поиска подстроки, основанный на грубой силе, для следующих шаблонов в тексте, состоящем из 1000 нулей?

   а) 00001 **б)** 10000 **в)** 01010

6. Приведите пример текста длиной n и шаблона длиной m, которые представляют собой входные данные для алгоритма поиска подстроки на основе грубой силы, приводящие к наихудшему случаю. Какое именно количество сравнений выполняется для таких входных данных?

7. Напишите псевдокод алгоритма поиска подстрок, основанного на грубой силе, который для заданного шаблона возвращает общее количество соответствующих ему подстрок в данном тексте.

8. Если вы ищете шаблон, содержащий редко встречающиеся символы (например, Q или Z в английском тексте), то как вы модифицируете алгоритм, основанный на грубой силе, чтобы воспользоваться этой информацией?

9. Популярная в Америке игра в слова состоит в том, что игрок должен найти каждое из заданного множества слов в квадратной таблице, заполненной отдельными буквами. Слова можно читать по горизонтали, вертикали или диагонали в любом направлении. Разработайте алгоритм для этой игры, основанный на использовании грубой силы, и реализуйте его в виде компьютерной программы.

10. Напишите программу для игры "морской бой", которая основана на версии поиска шаблона с использованием грубой силы. Правила игры очень просты. В игре участвуют два игрока (в нашем случае — человек и компьютер). Игра происходит на двух одинаковых полях размером 10×10 квадратов, где игроки размещают свои корабли, невидимые для противника. У каждого игрока по 5 кораблей, каждый из которых занимает определенное количество квадратов поля: эсминец (2 квадрата), подводная лодка (3 квадрата), крейсер (3 квадрата), линкор (4 квадра-

та) и авианосец (5 квадратов).[1] Корабли размещаются горизонтально и вертикально, причем не должны касаться друг друга. Игроки поочередно "стреляют", называя координаты выстрела. Результат каждого выстрела может быть "попал" или "промазал". При попадании попавший игрок делает внеочередной ход, и так до первого промаха. Цель игры — первым потопить все корабли противника (корабль считается потопленным, если попадания были во все составляющие его клетки).

3.3 Задачи поиска пары ближайших точек и вычисления выпуклой оболочки с использованием грубой силы

В этом разделе мы рассмотрим методы грубой силы для решения двух популярных задач, оперирующих с конечным множеством точек на плоскости. Эти задачи, помимо теоретического интереса, используются в таких важных прикладных областях, как вычислительная геометрия и анализ операций.

Поиск пары ближайших точек

Задача поиска пары ближайших точек заключается в том, чтобы во множестве из n точек найти две, расположенные друг к другу ближе других. Для простоты мы будем рассматривать только двумерный случай, хотя задача может быть поставлена и для большего числа измерений. Точки задаются стандартным способом с помощью декартовых координат (x, y), а расстояние между двумя точками $P_i = (x_i, y_i)$ и $P_j = (x_j, y_j)$ представляет собой стандартное расстояние геометрии Евклида (евклидово расстояние)

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Подход с применением грубой силы для решения этой задачи приводит нас к следующему очевидному алгоритму: вычислить расстояния между каждой парой точек и найти пару с наименьшим расстоянием. Естественно, мы хотим избежать повторного вычисления расстояния между одними и теми же точками, поэтому рассматриваем только пары точек (P_i, P_j), для которых $i < j$.

[1] Набор кораблей в отечественном "морском бое", по детским воспоминаниям, следующий. один корабль длиной 4 квадрата, два корабля длиной по 3 квадрата, три — по два квадрата и четыре — по одному. Специальных имен они не носили, а назывались по числу квадратов — от однопалубного до четырехпалубного. — *Прим. перев.*

АЛГОРИТМ *BruteForceClosestPoints* (P)

// **Входные данные:** список P, состоящий из $n \geqslant 2$ точек
// $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
// **Выходные данные:** индексы $index1$ и $index2$ пары
// ближайших точек
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow sqrt((x_i - x_j) * *2 +$ // $sqrt$ — функция вычисления
      $+(y_i - y_j) * *2)$ // квадратного корня,
    if $d < dmin$ // $**$ — возведение в степень
      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$;
return $index1$, $index2$

Базовой операцией алгоритма является вычисление расстояния между двумя точками. В век электронных калькуляторов с кнопкой вычисления квадратного корня может показаться, что это очень простая операция, как, скажем, сложение или умножение. Однако это не так. Начнем с того, что даже для большинства целых чисел квадратный корень является иррациональным числом, так что он может быть вычислен только приближенно. Более того, такие приближенные вычисления — отнюдь не тривиальная задача. Однако все оказывается гораздо интереснее: вычисления квадратного корня в алгоритме вполне можно избежать! (Можете ли вы придумать, каким образом?) Фокус в том, что можно просто проигнорировать функцию для вычисления квадратного корня и сравнивать значения $(x_i - x_j)^2 + +(y_i - y_j)^2$. Мы можем так поступить, поскольку чем меньше число, для которого мы вычисляем квадратный корень, тем меньше и вычисленный квадратный корень. Говоря строгим математическим языком, функция квадратного корня строго возрастающая.

Так что, если заменить

$d \leftarrow sqrt((x_i - x_j) * *2 + (y_i - y_j) * *2)$

на

$d \leftarrow (x_i - x_j) * *2 + (y_i - y_j) * *2$

то базовой операцией алгоритма станет возведение чисел в квадрат. Общее количество таких операций можно вычислить следующим образом:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i) = 2 [(n - 1) + (n - 2) + \cdots + 1] =$$
$$= (n - 1)\, n \in \Theta(n^2).$$

В главе 4 мы рассмотрим алгоритм для решения этой задачи, класс эффективности которого — $n \log n$.

Поиск выпуклой оболочки

Теперь рассмотри другую задачу — вычисление выпуклой оболочки. Начнем с определения того, что такое выпуклая оболочка.

Определение 1. Множество точек (конечное или бесконечное) на плоскости называется ***выпуклым***, если для любых двух точек P и Q, принадлежащих данному множеству, отрезок с конечными точками P и Q будет полностью принадлежать этому же множеству. ■

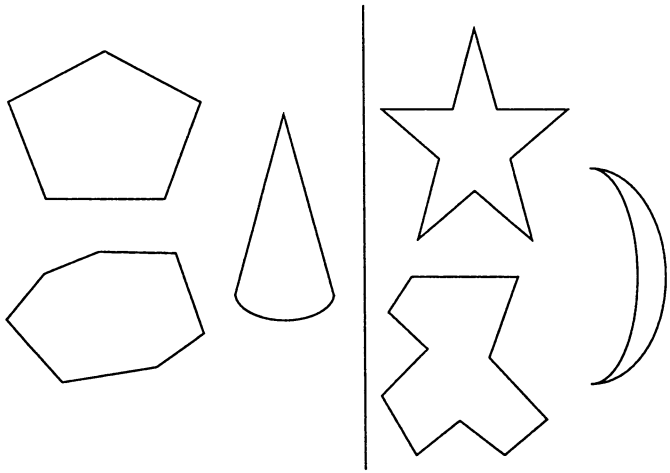


Рис. 3.4. *а*) Выпуклые множества. *б*) Множества, не являющиеся выпуклыми

Все множества на рис. 3.4*а* — выпуклые; выпуклыми являются обычная прямая, треугольник, прямоугольник (говоря более обобщенно — любой выпуклый многоугольник[2]), круг, вся плоскость. Множества, показанные на рис. 3.4*б*, любое множество, состоящее из двух различных точек, граница выпуклого многоугольника, окружность — все это примеры множеств, не являющихся выпуклыми.

Теперь можно перейти к понятию выпуклой оболочки. Интуитивно выпуклая оболочка множества из n точек на плоскости — это наименьший выпуклый многоугольник, который содержит все точки множества — внутри или на границе. Если такое определение не вызывает у вас энтузиазма, можно воспользоваться интерпретацией Д. Харела (D. Harel) [48]: как обнести n спящих тигров забором минимальной длины? Проблема в такой формулировке только в том, как вбить

[2]Под треугольником, прямоугольником и выпуклым многоугольником мы подразумеваем область — т.е. множество точек внутри и на границе указанной фигуры.

столб для забора в том месте, где спит тигр?... Есть и еще одна, не такая страшная интерпретация. Предположим, что каждая точка представлена гвоздем, вбитым в деревянную поверхность, представляющую плоскость. Возьмем резиновую ленту, растянем ее так, чтобы она охватывала все гвозди, и отпустим. Выпуклая оболочка представляет собой область, ограниченную отпущенной резиновой лентой (рис. 3.5).

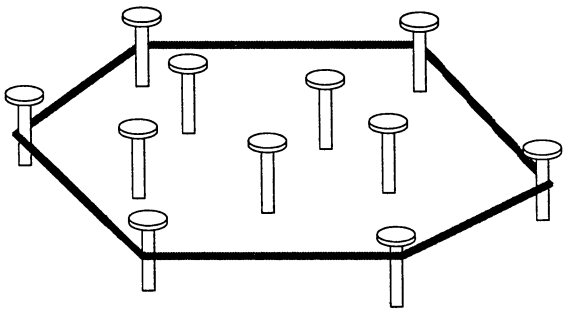


Рис. 3.5. Интерпретация выпуклой оболочки с помощью резиновой ленты

Формальное определение выпуклой оболочки применимо к произвольным множествам, включающим множества точек, лежащих на одной прямой, и другие подобные экстремальные случаи.

Определение 2. ***Выпуклая оболочка*** множества точек S представляет собой наименьшее выпуклое множество, содержащее S. ("Наименьшее" означает, что выпуклая оболочка S должна быть подмножеством любого выпуклого множества, содержащего S.) ■

Если множество S выпукло, то очевидно, что выпуклая оболочка S совпадает с S. Если S — множество, состоящее из двух точек, то его выпуклая оболочка представляет собой отрезок, соединяющий эти точки. Для множества из трех точек, не лежащих на одной прямой, выпуклая оболочка представляет собой треугольник с вершинами в этих трех точках; если же три точки лежат на одной прямой, то выпуклая оболочка является отрезком, соединяющим две, наиболее удаленные из них. На рис. 3.6 приведен пример выпуклой оболочки множества большего размера.

Изучение примеров позволяет сформулировать следующую теорему.

Теорема 1. Выпуклая оболочка произвольного множества S, состоящего из $n > 2$ точек (не лежащих на одной прямой), представляет собой выпуклый многоугольник с вершинами в некоторых точках S. (Если все точки лежат на одной прямой, то многоугольник вырождается в отрезок прямой, две конечные точки которого все равно принадлежат множеству S.) ■

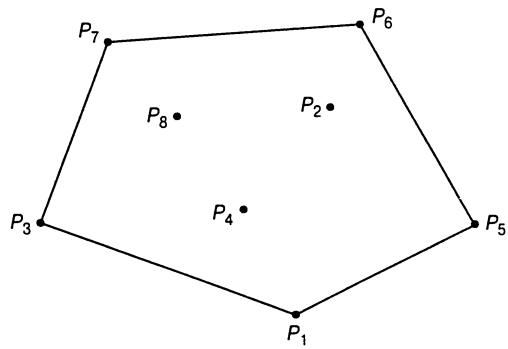


Рис. 3.6. Выпуклая оболочка для множества из 8 точек, представляющая собой выпуклый многоугольник

Вычисление выпуклой оболочки представляет собой задачу построения выпуклой оболочки для множества S, состоящего из n точек. Для ее решения необходимо найти точки, которые служат вершинами искомого многоугольника. Математики называют вершины такого многоугольника "угловыми точками" (extreme points). По определению, ***угловая точка*** выпуклого множества, которая не является срединной точкой никакого отрезка с конечными точками из данного множества. Например, угловыми точками треугольника являются три его вершины; угловыми точками круга — все точки его окружности; угловыми точками выпуклой оболочки восьми точек, показанной на рис. 3.6, — точки P_1, P_5, P_6, P_7 и P_3.

Угловые точки имеют ряд особых свойств, отсутствующих у других точек выпуклого множества. Одно из таких свойств используется очень важным алгоритмом — так называемым ***симплекс-методом*** (simplex method). Этот алгоритм предназначен для решения задач ***линейного программирования***, т.е. задач поиска минимума или максимума линейной функции от n переменных, на которые накладываются линейные ограничения (см. в качестве примера задачу 3.3.9 или обсуждение в разделе 6.6). Сейчас же угловые точки нас интересуют постольку, поскольку их идентификация решает задачу вычисления выпуклой оболочки. В действительности для решения этой задачи надо знать не только угловые точки, но и то, какие пары этих точек должны быть соединены для получения границы выпуклой оболочки. Заметим, что ответ на этот вопрос может быть получен при перечислении угловых точек в порядке обхода по или против часовой стрелки.

Так как же решить задачу вычисления выпуклой оболочки с применением грубой силы? Если у вас нет плана немедленной фронтальной атаки, не отчаивайтесь: задача вычисления выпуклой оболочки не имеет очевидного алгоритмического решения. Тем не менее имеется простой, но неэффективный алгоритм, основанный на следующем свойстве отрезков, образующих границу выпуклой оболочки: от-

резок, соединяющий точки P_i и P_j множества, состоящего из n точек, является частью границы выпуклой оболочки тогда и только тогда, когда все прочие точки лежат по одну сторону от прямой, проходящей через эти две точки.[3] (Проверьте это свойство для множества, показанного на рис. 3.6.) Проверка каждой пары точек дает список отрезков, которые составляют границу выпуклой оболочки.

Для реализации этого алгоритма требуется вспомнить несколько элементарных фактов из аналитической геометрии. Во-первых, прямая линия, проходящая через две точки (x_1, y_1) и (x_2, y_2) на координатной плоскости, описывается уравнением $ax + by = c$, где $a = y_2 - y_1$, $b = x_1 - x_2$ и $c = x_1y_2 - y_1x_2$.

Во-вторых, такая прямая делит плоскость на две полуплоскости: для всех точек одной из полуплоскостей $ax + by > c$, а для точек второй — $ax + by < c$ (естественно, для точек на прямой справедливо соотношение $ax + by = c$). Таким образом, для того, чтобы проверить, лежат ли точки по одну сторону прямой, достаточно вычислить для них значения $ax + by - c$ и выяснить, во всех ли точках эти значения имеют одинаковый знак. Реализация деталей этого алгоритма остается читателям в качестве самостоятельного упражнения.

Эффективность данного алгоритма равна $O\left(n^3\right)$: для каждой из $n\,(n-1)/2$ различных пар точек понадобится найти знаки выражений $ax + by - c$ для $n - 2$ точек. Для решения этой задачи существуют более эффективные алгоритмы; которые будут рассмотрены позже.

Упражнения 3.3

1. Можете ли вы разработать более быстрый алгоритм, основанный на методе грубой силы, с помощью которого выполнялся бы поиск пары ближайших точек для n точек $x_1, \ldots, x_n$ на действительной оси координат?

2. **а)** Имеется ряд альтернативных определений расстояний между двумя точками $P_1 = (x_1, y_1)$ и $P_2 = (x_2, y_2)$. В частности, так называемое ***манхэттенское расстояние*** определяется как

$$d_M\left(P_1, P_2\right) = |x_1 - x_2| + |y_1 - y_2|\,.$$

Докажите, что d_M удовлетворяет следующим аксиомам, которым должна удовлетворять любая функция расстояния:

1) $d_M\left(P_1, P_2\right) \geqslant 0$ для любых двух точек P_1 и P_2, и $d_M\left(P_1, P_2\right) = 0$ тогда и только тогда, когда $P_1 = P_2$;

2) $d_M\left(P_1, P_2\right) = d_M\left(P_2, P_1\right)$;

[3] Для простоты мы считаем, что никакие три точки не лежат на одной прямой. Модификация, необходимая для общего случая, остается читателям в качестве самостоятельного упражнения.

3) Для любых точек P_1, P_2 и P_3 справедливо неравенство $d_M(P_1, P_2) \leqslant d_M(P_1, P_3) + d_M(P_3, P_2)$.

б) Изобразите на координатной плоскости все точки, манхэттенское расстояние которых до начала координат $(0, 0)$ равно 1. Сделайте то же для евклидова расстояния.

в) Истинно или ложно следующее утверждение: результат поиска пары ближайших точек не зависит от того, какая из двух метрик используется — евклидова, d_E, или манхэттенская, d_M?

3. Задача поиска пары ближайших точек может быть поставлена в k-мерном пространстве, где евклидово расстояние между точками $P' = (x'_1, \ldots, x'_k)$ и $P'' = (x''_1, \ldots, x''_k)$ определяется как

$$d(P', P'') = \sqrt{\sum_{s=1}^{k} (x'_s - x''_s)^2}.$$

Каков класс эффективности алгоритма поиска пары ближайших точек в k-мерном пространстве, основанного на методе грубой силы?

4. Найдите выпуклые оболочки следующих множеств и укажите их угловые точки (если таковые имеются):

а) отрезок прямой;

б) квадрат;

в) граница квадрата;

г) прямая линия.

5. Какое минимальное и максимальное количество угловых точек может иметь выпуклая оболочка множества из n различных точек?

6. Разработайте алгоритм с линейным временем работы для поиска одной угловой точки выпуклой оболочки множества из n точек на плоскости.

7. Как изменить алгоритм, вычисляющий выпуклую оболочку с помощью грубой силы, чтобы он подходил для случая, когда на одной прямой могут располагаться больше двух точек?

8. Напишите программу, реализующую алгоритм для вычисления выпуклой оболочки с использованием метода грубой силы.

9. Рассмотрим следующий небольшой экземпляр задачи линейного программирования:

$$\begin{array}{ll} \text{максимизировать} & 3x + 5y \\ \text{при условиях} & x + y \leqslant 4 \\ & x + 3y \leqslant 6 \\ & x \geqslant 0, \quad y \geqslant 0. \end{array}$$

а) Изобразите на координатной плоскости ***допустимую область*** (feasible region) задачи, т.е. множество точек, удовлетворяющих всем ограничениям из условия задачи.

б) Определите угловые точки области.

в) Решите задачу оптимизации с использованием следующей теоремы: задача линейного программирования с непустой ограниченной допустимой областью всегда имеет решение, причем оно находится в одной из угловых точек допустимой области.

3.4 Исчерпывающий перебор

Многие важные задачи требуют поиска элемента со специальными свойствами в области, экспоненциально (или еще быстрее) растущей с увеличением размера экземпляра задачи. Обычно такие задачи возникают в случаях, когда присутствуют — явно или неявно — комбинаторные объекты, такие как перестановки, сочетания или подмножества данного множества. Многие подобные задачи являются задачами оптимизации: в них требуется найти элемент, который максимизирует или минимизирует некоторые интересующие нас характеристики, как, например, длина пути или назначенная стоимость.

Исчерпывающий перебор (exhaustive search) представляет собой подход к комбинаторным задачам с позиции грубой силы. Он предполагает генерацию всех возможных элементов из области определения задачи, выбор тех из них, которые удовлетворяют ограничениям, накладываемым условием задачи, и последующий поиск нужного элемента (например, оптимизирующего значение целевой функции задачи). Заметим, что, хотя идея исчерпывающего перебора весьма проста, ее реализация обычно требует алгоритма для генерации определенных комбинаторных объектов. Отложим подробное рассмотрение таких алгоритмов до главы 5, а пока что будем просто считать, что таковые существуют. Исчерпывающий перебор будет проиллюстрирован применением его к трем важным задачам: задаче коммивояжера, задаче о рюкзаке и задаче о назначениях.

Задача коммивояжера

Задача коммивояжера беспокоит умы исследователей уже более 100 лет простотой формулировки, важными приложениями и интересными связями с другими

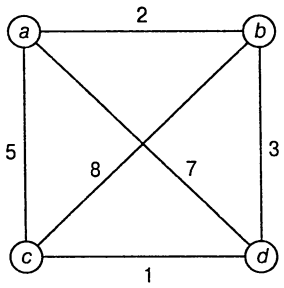
комбинаторными задачами. В переводе для любителя задача формулируется так: надо найти такой кратчайший путь по заданным n городам, чтобы каждый город посещался только один раз и конечным пунктом оказался город, с которого начиналось путешествие. Проблему удобно смоделировать при помощи взвешенного графа, вершины которого представляют города, а веса ребер определяют расстояния. После этого задача может быть сформулирована как задача поиска кратчайшего ***гамильтонового цикла*** (Hamiltonian circuit) неориентированного графа. (Гамильтоновым именуют цикл, который проходит по всем вершинам графа ровно по одному разу. Он назван так в честь ирландского математика Вильяма Ровена Гамильтона (William Rowan Hamilton) (1805–1865), который интересовался такими циклами в качестве применения своих алгебраических открытий.)

Легко видеть, что гамильтонов цикл можно также определить как последовательность $n + 1$ смежных вершин $v_{i_0}, v_{i_1}, \ldots, v_{i_{n-1}}, v_{i_0}$, где первая вершина в последовательности совпадает с последней, в то время как все остальные $n - 1$ вершин различны. Далее, без потери общности можно предположить, что все циклы начинаются и заканчиваются в одной конкретной вершине (в конце концов, все они циклы, верно?) Значит, можно получить все возможные маршруты, генерируя все перестановки $n - 1$ промежуточных городов, вычисляя длину соответствующих путей и находя кратчайший из них. На рис. 3.7 показан небольшой экземпляр данной задачи и его решение описанным методом.

При внимательном рассмотрении рис. 3.7 можно выявить три пары обходов, которые отличаются друг от друга только направлением. Таким образом, можно снизить количество перестановок вершин вдвое. Можно, например, выбрать две промежуточные вершины, скажем, B и C, и рассматривать только те перестановки, в которых B предшествует C (такой трюк неявно определяет направление обхода). Однако это улучшение алгоритма не дает заметного эффекта. Общее количество перестановок остается равным $(n - 1)!/2$, что делает исчерпывающий перебор неприемлемым для всех значений n, кроме самых малых. С другой стороны, если вы — оптимист и видите стакан наполовину полным, а не наполовину пустым, то скажете, что нельзя недооценивать снижения количества работы вдвое, в особенности если она делается вручную. Заметим также, что если бы не поставленное ограничение, чтобы все пути начинались в одной и той же вершине, то количество перестановок было бы в n раз больше.

Задача о рюкзаке

Вот еще одна известная алгоритмическая задача. Дано n предметов весом $w_1, \ldots, w_n$ и ценой $v_1, \ldots, v_n$, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке, и которое имеет при этом максимальную стоимость. Если вам не нравится представлять себя вором, пытающимся запихнуть в свой рюкзак самые ценные вещи, представьте



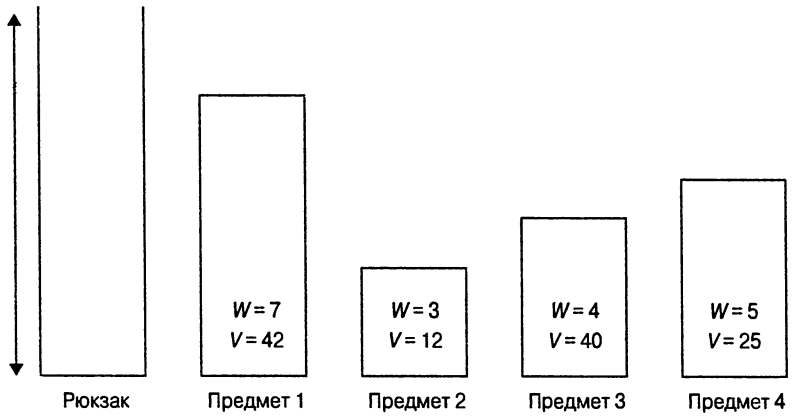
Путь	Длина	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	Оптимален
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	Оптимален
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Рис. 3.7. Решение небольшого экземпляра задачи коммивояжера методом исчерпывающего перебора

себя транспортным чиновником, которому надо перевезти максимально ценный груз, не превышающий определенного веса. На рис. 3.8*а* приведен небольшой экземпляр задачи о рюкзаке.

Исчерпывающий перебор в этой задаче приводит к рассмотрению всех подмножеств данного множества из n предметов, вычислению общего веса каждого из них для того, чтобы выяснить, допустим ли такой набор предметов (т.е. не превосходит ли его общий вес возможности рюкзака), и выбору из допустимых подмножества с максимальным весом. В качестве примера решение экземпляра задачи, представленного на рис. 3.8*а*, показано на рис. 3.8*б*. Поскольку общее количество подмножеств n-элементного множества равно 2^n, исчерпывающий перебор приводит к алгоритму со временем работы $\Omega(2^n)$, вне зависимости от того, насколько эффективным методом генерируются рассматриваемые подмножества.

Таким образом, применение метода исчерпывающего перебора к задачам коммивояжера и о рюкзаке приводит к исключительно неэффективным алгоритмам для любых входных данных. На самом деле эти две задачи представляют собой наиболее известные примеры так называемых ***NP-сложных задач*** (NP-hard problems). Ни для одной из NP-сложных задач не известен алгоритм, решающий их за полиномиальное время. Более того, большинство ученых-кибернетиков сходятся



а)

Подмножество	Общий вес	Общая стоимость
$\varnothing$	0	0
$\{1\}$	7	42
$\{2\}$	3	12
$\{3\}$	4	40
$\{4\}$	5	25
$\{1,2\}$	10	36
$\{1,3\}$	11	Недопустим
$\{1,4\}$	12	Недопустим
$\{2,3\}$	7	52
$\{2,4\}$	8	37
$\{3,4\}$	**9**	**65**
$\{1,2,3\}$	14	Недопустим
$\{1,2,4\}$	15	Недопустим
$\{1,3,4\}$	16	Недопустим
$\{2,3,4\}$	12	Недопустим
$\{1,2,3,4\}$	19	Недопустим

б)

Рис. 3.8. *а*) Экземпляр задачи о рюкзаке. *б*) Решение путем исчерпывающего перебора (оптимальное решение выделено полужирным шрифтом)

во мнении, что такие алгоритмы не существуют вообще, хотя это важное предположение никем не доказано. Более интеллектуальные подходы, рассматриваемые в разделах 11.2 и 11.2, позволяют решить некоторые (но не все) экземпляры

этих (и подобных) задач за время, меньшее экспоненциального. Можно также воспользоваться одним из приближенных алгоритмов, подобных рассмотренным в разделе 11.3.

Задача о назначениях

В третьем примере задачи, которая может быть решена исчерпывающим перебором, имеется n работников, которые должны выполнить n заданий, по одному заданию каждый (т.е. каждому работнику назначается только одно задание, и каждое задание назначается лишь одному человеку). Стоимость выполнения i-ым работником j-го задания известна и равна $C[i, j]$ для всех пар $i, j = 1, \ldots n$. Задача заключается в следующем: надо распределить задания между работниками таким образом, чтобы они были выполнены с наименьшей общей стоимостью.

Небольшой экземпляр данной задачи приведен в таблице стоимости выполнения заданий $C[i, j]$:

	Задание 1	**Задание 2**	**Задание 3**	**Задание 4**
Работник 1	9	2	7	8
Работник 2	6	4	3	7
Работник 3	5	8	1	8
Работник 4	7	6	9	4

Легко видеть, что экземпляр задачи с назначением заданий полностью определяется матрицей стоимости C. В терминах данной матрицы в задаче требуется выбрать по одному элементу из каждой строки матрицы так, чтобы выбранные элементы находились в разных столбцах, а их общая сумма имела наименьшее возможное значение. Заметим, что очевидной стратегии решения данной задачи не существует. Например, нельзя выбирать наименьшие элементы в каждой строке, поскольку они могут оказаться в одном и том же столбце. В действительности, наименьшие элементы матрицы могут вообще не входить в оптимальное решение. Так что исчерпывающий перебор для решения данной задачи может оказаться неизбежен.

Допустимое решение задачи о назначениях можно описать в виде кортежа из n значений $\langle j_1, \ldots, j_k \rangle$, в котором i-ый компонент $i = 1, \ldots, n)$ указывает столбец, где находится выбранный в i-ой строке компонент (т.е. номер задания, назначенного i-му работнику). Например, для приведенной выше матрицы стоимости $\langle 2, 3, 4, 1 \rangle$ указывает допустимое назначение второго задания первому работнику, третьего задания — второму работнику, четвертого задания — третьему работнику, и первого задания — четвертому работнику. Из условий задачи вытекает, что имеется однозначное соответствие между допустимыми назначениями и перестановками первых n натуральных чисел. Следовательно, исчерпывающий перебор для

задачи о назначениях может потребовать генерации всех перестановок натуральных чисел $1, 2, \ldots, n$, вычисления общей стоимости каждого назначения путем суммирования соответствующих элементов матрицы стоимости и окончательного выбора назначения с минимальной стоимостью. Несколько первых шагов применения данного алгоритма к приведенному ранее экземпляру задачи показано на рис. 3.9. Читателям предлагается самостоятельно завершить это упражнение.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> стоимость = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> стоимость = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> стоимость = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> стоимость = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> стоимость = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> стоимость = 9 + 7 + 1 + 6 = 23
и т.д.

Рис. 3.9. Несколько первых итераций решения экземпляра задачи о назначениях путем исчерпывающего перебора

Поскольку в задаче о назначениях в общем случае количество рассматриваемых перестановок равно $n!$, исчерпывающий перебор непрактичен для всех, кроме очень небольших, значений n. К счастью, для данной задачи имеется существенно более эффективный алгоритм, именуемый ***венгерским методом*** в честь открывших его венгерских математиков Кёнига (Konig) и Эгервари (Egervary) (см., например, [69]).

Это хорошая новость. Оказывается, экспоненциальный (или более быстрый) рост области определения задачи не обязательно влечет отсутствие эффективного алгоритма для ее решения. Позже в этой книге вы познакомитесь с другими подобными задачами. Однако такие примеры — скорее исключение, чем правило. Чаще всего для задач, область определения которых растет экспоненциально (при условии, что мы хотим получить точное решение), полиномиальный алгоритм решения неизвестен. И, как было сказано, очень может быть, что такие алгоритмы просто не существуют.

Упражнения 3.4

1. а) К какому классу эффективности относится описанный в тексте алгоритм исчерпывающего перебора для решения задачи коммивояжера, если считать, что каждый путь может быть сгенерирован за постоянное время?

б) Пусть такой алгоритм, реализованный в виде компьютерной программы, делает 1 миллиард сложений в секунду. Оцените максимальное количество городов, для которых задача коммивояжера может быть решена за

1) час;
2) сутки;
3) год;
4) столетие.

2. Набросайте схему алгоритма для поиска гамильтонова цикла методом исчерпывающего перебора.
3. Опишите алгоритм, с помощью которого можно определить, содержит ли граф, представленный в виде матрицы смежности, Эйлеров цикл. К какому классу эффективности относится данный алгоритм?
4. Продолжите применение алгоритма исчерпывающего перебора к экземпляру задачи о назначениях, приведенному в тексте раздела.
5. Приведите пример задачи о назначениях, оптимальное решение которой не включает наименьшие элементы матрицы стоимости.
6. Напишите реферат о венгерском методе.
7. Рассмотрим ***задачу разбиения*** (partition problem): даны n натуральных чисел, которые надо разбить на два непересекающихся подмножества с одинаковой суммой их элементов (понятно, что задача не всегда имеет решение). Разработайте алгоритм решения данной задачи на основе исчерпывающего перебора. Постарайтесь минимизировать количество подмножеств, которые должен сгенерировать алгоритм.
8. Рассмотрим ***задачу о клике*** (clique problem): для графа G и натурального числа k надо определить, содержит ли граф ***клику*** размера k, т.е. полный подграф из k вершин. Разработайте алгоритм на основе исчерпывающего перебора для решения данной задачи.
9. Поясните, как применить метод исчерпывающего перебора к задаче сортировки и определите класс эффективности такого алгоритма.

10. Магический квадрат порядка n представляет собой набор чисел от 1 до n^2, размещенных в матрице размером $n \times n$, причем каждое число встречается в матрице ровно один раз, при этом суммы чисел в каждой строке, каждом столбце и каждой диагонали одинаковы.
    а) Докажите, что если магический квадрат порядка n существует, то интересующая нас сумма должна быть равна $n\left(n^2+1\right)/2$.
    б) Разработайте алгоритм исчерпывающего перебора для генерации всех магических квадратов порядка n.
    в) Поищите в Internet или библиотеке более эффективный алгоритм генерации магических квадратов.

г) Реализуйте алгоритм исчерпывающего перебора и найденный вами более эффективный алгоритм и экспериментальным путем определите максимальный размер магического квадрата, который может быть сгенерирован каждой из реализаций за одну минуту.

Резюме

- Подход к решению задачи с использованием *грубой силы* обычно основан непосредственно на формулировке задачи и определениях применяемых в ней концепций.
- Главные достоинства подхода с использованием грубой силы — широкая применимость и простота; главный недостаток — очень низкая эффективность таких алгоритмов.
- Первое применение подхода с использованием грубой силы для решения задачи обычно дает алгоритм, который можно улучшить ценой достаточно скромных усилий.
- Примером подхода с использованием грубой силы могут служить следующие известные алгоритмы:
  - алгоритм умножения матриц, основанный на определении этой операции;
  - *сортировка выбором*;
  - *последовательный поиск*;
  - простой алгоритм поиска подстрок.
- *Исчерпывающий перебор* представляет собой подход с использованием грубой силы для решения комбинаторных задач. Он предполагает генерацию всех комбинаторных объектов задачи, выбор среди них тех, которые удовлетворяют ограничениям из условий задачи, и последующий поиск нужного объекта.
- *Задача коммивояжера*, *задача о рюкзаке* и *задача о назначениях* представляют собой типичные примеры задач, которые могут быть решены — по крайней мере теоретически — при помощи алгоритма исчерпывающего перебора.
- Исчерпывающий перебор непрактичен для всех, кроме очень небольших, экземпляров задач, к которым он может быть применен.

Глава 4

Метод декомпозиции

> О чем бы не молился человек, он молится о чуде. Каждый молящийся приходит к этому — Господи Боже, сделай, чтобы дважды два не было равно четырем.
>
> — И. Тургенев (1818–1883)

Метод декомпозиции (он же метод "разделяй и властвуй"), вероятно, наиболее популярный метод разработки алгоритмов. Ряд очень эффективных алгоритмов представляют собой реализации этой общей стратегии. Алгоритмы, основанные на методе декомпозиции, работают в соответствии со следующим планом.

1. Экземпляр задачи разбивается на несколько меньших экземпляров той же задачи, в идеале — одинакового размера.
2. Решаются меньшие экземпляры задачи (обычно рекурсивно, хотя иногда для небольших экземпляров применяется какой-нибудь другой алгоритм).
3. При необходимости решение исходной задачи находится путем комбинации решений меньших экземпляров.

Схема метода декомпозиции показана на рис. 4.1, где приведен случай разделения задачи на две равные по размеру подзадачи, что является, пожалуй, самой распространенной ситуацией (по крайней мере для алгоритмов декомпозиции, разработанных для выполнения на однопроцессорном компьютере).

В качестве примера рассмотрим задачу вычисления суммы чисел $a_0, \ldots, a_{n-1}$. Если $n > 1$, задачу можно разделить на два экземпляра той же задачи: вычисление суммы первых $\lfloor n/2 \rfloor$ чисел и вычисление суммы оставшихся $\lceil n/2 \rceil$ чисел (понятно, что при $n = 1$ в качестве ответа просто возвращается значение a_0). Как только каждая из этих двух сумм будет вычислена (с применением описанного метода, т.е. рекурсивно), мы сможем сложить их значения, чтобы получить окончательный ответ:

$$a_0 + \cdots + a_{n-1} = \left(a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}\right) + \left(a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}\right).$$

Рис. 4.1. Метод декомпозиции

Является ли описанный алгоритм эффективным способом суммирования n чисел? Небольшие размышления по поводу сравнения этого алгоритма с алгоритмом на основе грубой силы, конкретный пример суммирования четырех чисел, формальный анализ (приведенный далее), да и просто здравый смысл — все приводит к отрицательному ответу на этот вопрос.

Таким образом, не каждый алгоритм на основе декомпозиции эффективнее алгоритма, основанного на грубой силе. Но очень часто Бог алгоритмов — вспомните эпиграф — прислушивается к молитвам, и время, затраченное на выполнение алгоритма на основе декомпозиции, оказывается меньше, чем время решения задачи каким-то другим методом. Метод декомпозиции дал кибернетике ряд очень важных и эффективных алгоритмов. Мы рассмотрим несколько классических примеров таких алгоритмов в данной главе. Хотя здесь рассматриваются только последовательные алгоритмы, стоит помнить о том, что метод декомпозиции идеально подходит для параллельных вычислений, когда каждая подзадача может одновременно решаться собственным процессором.

Пример суммирования иллюстрирует наиболее типичный случай декомпозиции: экземпляр задачи размером n делится на два экземпляра размером $n/2$. В общем случае экземпляр задачи размера n может быть разделен на несколько экземпляров размером n/b, a из которых требуется решить (здесь a и b — кон-

станты; $a \geqslant 1$ и $b > 1$). Полагая для упрощения анализа, что размер n равен степени b, получаем следующее рекуррентное соотношение для времени работы алгоритма $T(n)$:

$$T(n) = aT(n/b) + f(n), \tag{4.1}$$

где $f(n)$ — функция, учитывающая затраты времени на разделение задачи на меньшие подзадачи и комбинирование решений подзадач. (В примере с суммированием чисел $a = b = 2$ и $f(n) = 1$.) Рекуррентное соотношение (4.1) называется ***обобщенным рекуррентным уравнением декомпозиции*** (general divide-and-conquer recurrence). Очевидно, что порядок роста его решения $T(n)$ зависит от значений констант a и b и порядка роста функции $f(n)$. Анализ эффективности множества алгоритмов, основанных на декомпозиции, существенно упрощается следующей теоремой (см. приложение Б).

ОСНОВНАЯ ТЕОРЕМА. Если в рекуррентном уравнении (4.1) $f(n) \in \Theta(n^d)$, где $d \geqslant 0$, то

$$T(n) \in \begin{cases} \Theta(n^d) & \text{если } a < b^d \\ \Theta(n^d \log n) & \text{если } a = b^d \\ \Theta(n^{\log_b a}) & \text{если } a > b^d \end{cases}$$

(Аналогичные результаты выполняются и для обозначений O и Ω.) ■

Например, рекуррентное соотношение для количества сложений $A(n)$ в описанном выше алгоритме суммирования декомпозицией для входных данных размера $n = 2^k$ равно

$$A(n) = 2A(n/2) + 1.$$

Таким образом, в данном примере $a = 2$, $b = 2$ и $d = 0$. Следовательно, поскольку $a > b^d$,

$$A(n) \in \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n).$$

Обратите внимание, что мы можем указать класс эффективности алгоритма, не решая само рекуррентное уравнение. Конечно же, этот подход позволяет установить порядок роста решения, не определяя неизвестные множители, которые можно найти, только решив рекуррентное уравнение.

4.1 Сортировка слиянием

Сортировка слиянием (mergesort) представляет собой превосходный пример успешного применения метода декомпозиции. Она сортирует заданный массив $A[0..n-1]$ путем разделения его на две половины, $A[0..\lfloor n/2 \rfloor - 1]$ и $A[\lfloor n/2 \rfloor .. n-1]$, рекурсивной сортировки каждой половины и слияния двух отсортированных половин в один отсортированный массив.

АЛГОРИТМ $Mergesort\,(A\,[0..n-1])$
// Рекурсивно сортирует массив $A[0..n-1]$
// **Входные данные:** Массив $A[0..n-1]$ упорядочиваемых
// элементов
// **Выходные данные:** Отсортированный в неубывающем порядке
// массив $A[0..n-1]$
if $n > 1$
Копировать $A[0..\lfloor n/2 \rfloor - 1]$ в $B[0..\lfloor n/2 \rfloor - 1]$
Копировать $A[\lfloor n/2 \rfloor..n-1]$ в $C[0..\lfloor n/2 \rfloor - 1]$
$Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
$Mergesort(C[0..\lfloor n/2 \rfloor - 1])$
$Merge(B, C, A)$

Слияние двух отсортированных массивов можно выполнить следующим образом. Два указателя (индекса массивов) после инициализации указывают на первые элементы сливаемых массивов. Затем элементы, на которые указывают указатели, сравниваются, и меньший из них добавляется в новый массив. После этого индекс меньшего элемента увеличивается, и он указывает на элемент, непосредственно следующий за только что скопированным. Эта операция повторяется до тех пор, пока не будет исчерпан один из сливаемых массивов, после чего оставшиеся элементы второго массива просто добавляются в конец нового массива.

АЛГОРИТМ $Merge\,(B\,[0..p-1]\,.\,C\,[0..q-1]\,,\,A\,[0..p+q-1])$
// Слияние двух отсортированных массивов в один
// **Входные данные:** Сортированные массивы $B[0..p-1]$
// и $C[0..q-1]$
// **Выходные данные:** Сортированный массив $A[0..p+q-1]$,
// состоящий из элементов B и C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leqslant C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else
    $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
  $k \leftarrow k + 1$
if $i = p$
  Копировать $C[j..q-1]$ в $A[k..p+q-1]$
else
  Копировать $B[i..p-1]$ в $A[k..p+q-1]$

Работа алгоритма со списком 8,3,2,9,7,1,5,4 показана на рис. 4.2.

Насколько эффективна сортировка слиянием? Положим для простоты, что n является степенью 2. Рекуррентное соотношение для количества выполняемых

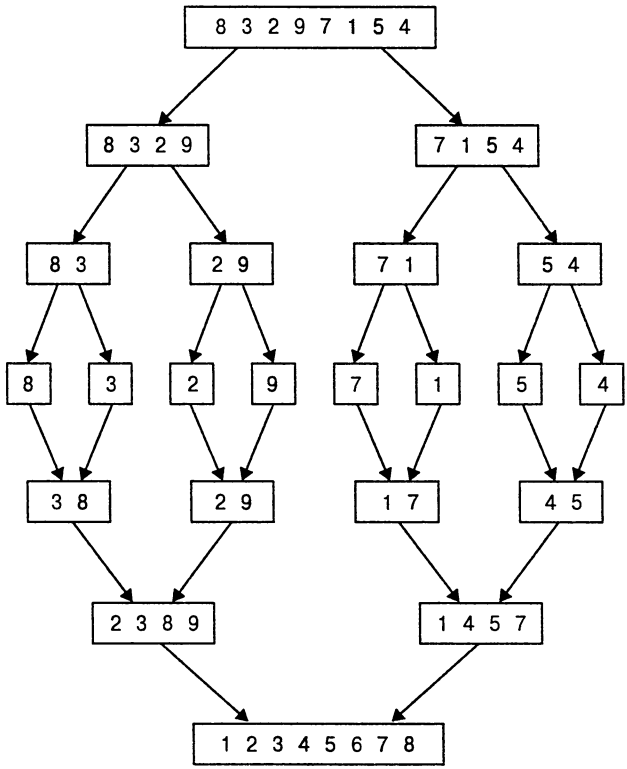


Рис. 4.2. Пример работы алгоритма сортировки слиянием

сравнений ключей $C(n)$ выглядит следующим образом:

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{для } n > 1, C(1) = 0.$$

Давайте проанализируем величину $C_{merge}(n)$, равную количеству сравнений ключей в процессе слияния. На каждом шаге выполняется в точности одно сравнение, после которого общее количество элементов в двух сливаемых массивах уменьшается на 1. В худшем случае ни один из массивов не исчерпывается до самого конца, пока в другом массиве не останется только один элемент (например, наименьшие элементы поступают из двух массивов поочередно). Следовательно, в наихудшем случае $C_{merge}(n) = n-1$, и мы получаем рекуррентное соотношение

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{для } n > 1, C_{worst}(1) = 0.$$

Согласно основной теореме, $C_{worst}(n) \in \Theta(n \log n)$ (почему?). Несложно найти и точное решение приведенного рекуррентного соотношения для $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1.$$

Количество сравнений ключей, выполняемых сортировкой слиянием, в худшем случае весьма близко к теоретическому минимуму[1] количества сравнений для любого алгоритма сортировки, основанного на сравнениях. Основной недостаток сортировки слиянием — необходимость дополнительной памяти, количество которой линейно пропорционально размеру входных данных. Сортировка слиянием возможна и без привлечения дополнительной памяти, но такая экономия памяти существенно ее усложняет и дает в результате значительно больший постоянный множитель в формуле для времени работы, так что эта версия сортировки слиянием представляет сугубо теоретический интерес.

Упражнения 4.1

1. **а)** Напишите псевдокод декомпозиционного алгоритма для поиска позиции наибольшего элемента в массиве из n чисел.
   б) Какими будут выходные данные алгоритма, если наибольшее значение имеют несколько элементов?
   в) Напишите и решите рекуррентное соотношение для количества сравнений ключей, выполняемых алгоритмом.
   г) Сравните созданный вами алгоритм с алгоритмом для решения указанной задачи, основанным на грубой силе.
2. **а)** Напишите псевдокод декомпозиционного алгоритма для поиска наибольшего и наименьшего элементов в массиве из n чисел.
   б) Напишите и решите рекуррентное соотношение для количества сравнений ключей, выполняемых этим алгоритмом.
   в) Сравните созданный вами алгоритм с алгоритмом для решения указанной задачи, основанным на грубой силе.
3. **а)** Напишите псевдокод декомпозиционного алгоритма для вычисления a^n, где $a > 0$, а n — натуральное число.
   б) Напишите и решите (для $n = 2^k$) рекуррентное соотношение для количества умножений, выполняемых алгоритмом.
   в) Сравните созданный вами алгоритм с алгоритмом для решения указанной задачи, основанным на грубой силе.
4. Рассматривая схему разработки и анализа алгоритмов в главе 2, мы упоминали, что в большинстве задач, возникающих при анализе алгоритмов, основание логарифма не играет роли. Насколько это верно для основной теоремы, в которой имеются логарифмы?

[1] Как мы увидим в разделе 10.2, этот теоретический минимум равен $\lceil \log_2 n! \rceil \approx \approx \lceil n \log_2 n - 1.44n \rceil$.

5. Найдите порядок роста решений следующих рекуррентных соотношений.
   а) $T(n) = 4T(n/2) + n$, $T(1) = 1$.
   б) $T(n) = 4T(n/2) + n^2$, $T(1) = 1$.
   в) $T(n) = 4T(n/2) + n^3$, $T(1) = 1$.
6. Примените сортировку слиянием для упорядочения букв E, X, A, M, P, L, E в алфавитном порядке.
7. Устойчива ли сортировка слиянием?
8. **а)** Решите рекуррентное соотношение для количества сравнений ключей, выполняемых сортировкой слиянием в наихудшем случае. (Можно считать, что $n = 2^k$.)
   б) Напишите рекуррентное соотношение для количества сравнений ключей, выполняемых алгоритмом сортировки слиянием в наилучшем случае, и решите его при $n = 2^k$.
   в) Напишите рекуррентное соотношение для количества перемещений ключей, выполняемых описанной в разделе 4.1 версией алгоритма сортировки слиянием. Изменится ли класс эффективности алгоритма, если учесть количество перемещений ключей?
9. Можно реализовать сортировку слиянием без рекурсии, начав со слияния соседних элементов данного массива, затем — отсортированных пар и т.д. Реализуйте такую восходящую версию алгоритма на своем любимом языке программирования.

10. *Триомино*. Триомино — элемент мозаичного заполнения в форме L, образованный тремя квадратами шахматной доски. Задача состоит в покрытии триомино шахматной доски размером $2^n \times 2^n$ с одной вырезанной в произвольном месте клеткой. Триомино должны покрывать все клетки, за исключением вырезанной, без пропусков и перекрытий.

Разработайте декомпозиционный алгоритм для решения этой задачи.

4.2 Быстрая сортировка

Быстрая сортировка (quicksort) — еще один важный алгоритм сортировки, основанный на методе декомпозиции. В отличие от сортировки слиянием, которая разделяет элементы массива в соответствии с их положением в массиве, быстрая сортировка разделяет элементы массива в соответствии с их значениями. Конкретно она выполняет перестановку элементов данного массива $A[0..n-1]$ для получения ***разбиения*** (partition), когда все элементы до некоторой позиции s не превышают элемента $A[s]$, а элементы после позиции s не меньше $A[s]$:

$$\underbrace{A[0]\dots A[s-1]}_{\text{Все элементы } \leqslant A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{Все элементы } \geqslant A[s]}.$$

Очевидно, что после разбиения $A[s]$ находится в окончательной позиции в отсортированном массиве, и мы можем сортировать два подмассива элементов до и после $A[s]$ независимо (например, тем же самым методом).

Алгоритм $Quicksort(A[l..r])$

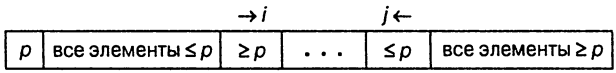
```
// Сортирует массив методом быстрой сортировки
// Входные данные:  Подмассив A[l..r] массива A[0..n − 1],
//                  определяемый начальным и конечным
//                  индексами l и r
// Выходные данные: Подмассив A[l..r], отсортированный
//                  в неубывающем порядке
if l < r
    s ← Partition(A[l..r])   // s — позиция разбиения
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

Разбиение массива $A[0..n-1]$ и, в общем случае, его подмассива $A[l..r]$ ($0 \leqslant l < r \leqslant n-1$) можно выполнить следующим образом. Сначала выбирается элемент, относительно которого будет выполняться разбиение. В силу важности роли этого элемента он называется ***опорным*** (pivot). Имеется ряд различных стратегий для выбора опорного элемента (мы вернемся к этому вопросу при рассмотрении эффективности алгоритма). А пока что воспользуемся простейшей стратегией — выберем в качестве опорного первый элемент подмассива: $p = A[l]$.

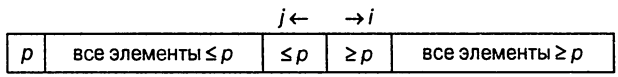
Имеется также ряд различных процедур перестановки элементов для разбиения. Здесь нами будет использоваться эффективный метод, основанный на двух проходах подмассива — слева направо и справа налево, и при каждом проходе элементы массива будут сравниваться с опорным элементом. Проход слева направо начинается со второго элемента. Поскольку мы хотим, чтобы элементы, меньшие опорного, находились в первой части подмассива, первый проход пропускает элементы, меньшие опорного, и останавливается, встретив первый элемент, который

не меньше опорного. Проход справа налево начинается с последнего элемента подмассива. Поскольку надо, чтобы все элементы, большие опорного, находились во второй части подмассива, при этом проходе опускаются все элементы, большие опорного, и проход останавливается, встретив первый элемент, не превышающий опорный.

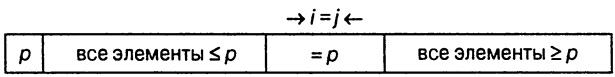
В зависимости от того, пересеклись ли индексы сканирования, возможны три ситуации. Если индексы сканирования i и j не пересеклись, т.е. $i < j$, то мы просто обмениваем элементы $A[i]$ и $A[j]$ и продолжаем сканирование путем увеличения i и уменьшения j:



Если при сканировании произошло пересечение индексов, т.е. $i > j$, то мы выполняем разбиение, обменивая опорный элемент с $A[j]$:



И, наконец, если при сканировании индексы остановились на одном элементе, т.е. $i = j$, то значение этого элемента должно быть равно p (почему?). Следовательно, мы имеем следующее разбиение массива:



Последний случай можно объединить со случаем пересечения индексов ($i > j$), обменивая опорный элемент с $A[j]$ при $i \geqslant j$.

Вот псевдокод, реализующий описанную процедуру разбиения.

АЛГОРИТМ *Partition* $(A[l..r])$

```
// Разбивает подмассив с использованием первого элемента
// в качестве опорного
// Входные данные:  подмассив A[l..r] массива A[0..n − 1],
//                  определяемый левым и правым
//                  индексами l и r (l < r)
// Выходные данные: разбиение A[l..r]; при этом позиция
//                  разбиения возвращается как значение
//                  функции
p ← A[l]
i ← l; j ← r + 1
repeat
  repeat
```

```
    i ← i + 1
  until A[i] ⩾ p
  repeat
    j ← j − 1
  until A[j] ⩽ p
  swap(A[i], A[j])
until i ⩾ j
swap(A[i], A[j])   // Отмена последнего обмена при i ⩾ j
swap(A[l], A[j])
return j
```

Заметим, что в таком псевдокоде возможен выход индекса i за пределы границ подмассива. Вместо того чтобы при каждом увеличении i проверять, не вышел ли индекс за пределы границы, можно добавить к массиву $A[0..n-1]$ "ограничитель", который не позволит индексу i выйти за пределы n. Заметим также, что выбор опорного элемента в конце подмассива делает ограничитель ненужным.

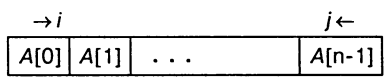
Пример сортировки массива при помощи алгоритма быстрой сортировки показан на рис. 4.3.

Обсуждение эффективности быстрой сортировки мы начнем с замечания о том, что количество сравнений ключей, выполненных до разбиения, достигает величины $n+1$, если индексы пересекаются, и n, если совпадают (почему?). Если все разбиения оказываются посредине соответствующих подмассивов, реализуется наилучший случай. Количество сравнений ключей в наилучшем случае, $C_{best}(n)$, удовлетворяет следующему рекуррентному соотношению:

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{при } n > 1, C_{best}(1) = 0.$$

Согласно основной теореме, $C_{best}(n) \in \Theta(n \log_2 n)$. Точное решение для $n = 2^k$ дает $C_{best}(n) = n \log_2 n$.

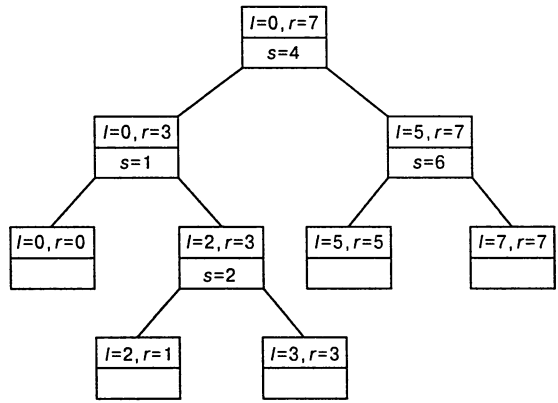
В наихудшем случае все разбиения оказываются такими, что один из подмассивов пуст, а размер второго на 1 меньше размера разбиваемого массива. Такая ситуация возникает, в частности, в возрастающем массиве, т.е. для входных данных, для которых задача сортировки уже решена! В самом деле, если $A[0..n-1]$ — строго возрастающий массив и мы используем в качестве опорного элемента $A[0]$, то сканирование слева направо остановится на элементе $A[1]$, в то время как сканирование справа налево дойдет до элемента $A[0]$, что указывает на разбиение в позиции 0:



Итак, после выполнения $n+1$ сравнений и обмена опорного элемента $A[0]$ с самим собой, выясняется, что теперь следует сортировать строго возрастающий массив $A[1..n-1]$. Такая сортировка строго возрастающих массивов будет

0	1	2	3	4	5	6	7
5	*i* 3	1	9	8	2	4	*j* 7
5	3	1	*i* 9	8	2	*j* 4	7
5	3	1	*i* 4	8	2	*j* 9	7
5	3	1	4	*i* 8	*j* 2	9	7
5	3	1	4	*i* 2	*j* 8	9	7
5	3	1	4	*j* 2	*i* 8	9	7
2	3	1	4	**5**	8	9	7
2	*i* 3	1	*j* 4				
2	*i* 3	*j* 1	4				
2	*i* 1	*j* 3	4				
2	*j* 1	*i* 3	4				
1	**2**	3	4				
1							
		3	*ij* 4				
		j **3**	*i* 4				
			4				
				8	*i* 9		*j* 7
				8	*i* 7		*j* 9
				8	*j* 7		*i* 9
				7	**8**		9
				7			
							9

а)



б)

Рис. 4.3. Пример работы алгоритма *Quicksort*. *а*) Преобразования массива (опорный элемент выделен полужирным шрифтом). *б*) Дерево рекурсивных вызовов *Quicksort* с указанием входных значений l и r и полученной позиции s

продолжаться до последнего массива $A[n-2..n-1]$. Общее количество выполненных сравнений ключей составляет

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Таким образом, встает вопрос об эффективности алгоритма в среднем случае. Обозначим среднее количество сравнений ключей, выполняемых при быстрой сортировке случайного массива размером n, через $C_{avg}(n)$. Считая, что разбиение

может выполняться в каждой позиции s ($0 \leqslant s \leqslant n-1$) с одинаковой вероятностью $1/n$, получаем следующее рекуррентное соотношение:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{при } n > 1,$$
$$C_{avg}(0) = 0,$$
$$C_{avg}(1) = 0.$$

Хотя решение этого соотношения оказывается проще, чем можно было бы ожидать, тем не менее оно существенно сложнее, чем в наихудшем и наилучшем случае. Оставляем решение этого соотношения читателям в качестве упражнения, а здесь приведем только готовый ответ:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38 n \log_2 n.$$

Таким образом, алгоритм быстрой сортировки в среднем случае выполняет сравнений ключей всего на 38% больше, чем в наилучшем случае. Кроме того, внутренний цикл данного алгоритма настолько эффективен, что для случайных массивов он работает быстрее, чем сортировка слиянием (и пирамидальная сортировка — еще один алгоритм класса $n \log n$, который будет рассмотрен в главе 6), оправдывая тем самым название, данное его разработчиком — выдающимся британским кибернетиком Хоаром (C. A. R. Hoare)[2].

Исходя из важности быстрой сортировки, многие годы делались попытки улучшить базовый алгоритм. Среди прочих усовершенствований, открытых различными исследователями, — улучшенные методы выбора опорного элемента (например, ***разбиение на основе медианы трех элементов***, когда в качестве опорного элемента используется медиана крайнего слева, справа и среднего элементов массива), переключение на более простую сортировку для малых подмассивов, удаление рекурсии (так называемая нерекурсивная быстрая сортировка). Согласно ведущему эксперту в области быстрой сортировки Седжвику (R. Sedgewick) [103], все вместе эти улучшения могут снизить время работы алгоритма на 20–25%.

Следует также отметить, что идея разбиения может оказаться полезной и в других приложениях, не только для сортировки. В частности, она лежит в основе быстрого алгоритма для важной ***задачи выбора***, рассматриваемой в разделе 5.6.

[2]Молодой Хоар изобрел этот алгоритм при попытке отсортировать слова в словаре русского языка для проекта машинного перевода с русского языка на английский. Дадим слово Хоару: "Моя первая мысль была — как бы выполнить задание при помощи пузырьковой сортировки, но тут меня настигло счастливое озарение и второй мыслью стала мысль о быстрой сортировке". Трудно не согласиться с его оценкой происшедшего: "Я был очень счастлив. Какое везение — начать карьеру в кибернетике с открытия нового алгоритма сортировки!" [51]

Упражнения 4.2

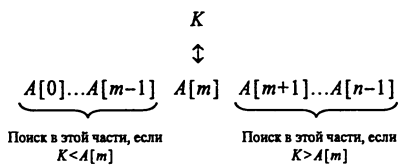
1. Примените быструю сортировку для сортировки списка E, X, A, M, P, L, E в алфавитном порядке. Изобразите дерево выполненных рекурсивных вызовов.
2. Для процедуры разбиения, описанной в разделе 4.2:
   а) Докажите, что если при сканировании индексы остановились на одном элементе, т.е. $i = j$, то значение этого элемента должно быть равно p.
   б) Докажите, что при прекращении сканирования индексов j не может указывать на элемент, находящийся более чем в одной позиции слева от элемента, на который указывает i.
   в) Почему наихудшим случаем является прекращение сканирования после того, как встретится элемент, равный опорному?
3. Устойчива ли быстрая сортировка?
4. Приведите пример массива из n элементов, для которого необходим ограничитель, упомянутый в разделе. Каким должно быть его значение? Поясните также, почему одного ограничителя достаточно для любых входных данных.
5. Для описанной версии быстрой сортировки:
   а) что собой представляет массив из одинаковых элементов — наихудший случай, наилучший случай или ни то, ни другое?
   б) что собой представляет массив из строго убывающих элементов — наихудший случай, наилучший случай или ни то, ни другое?
6. Предположим, что в алгоритме используется упомянутое в тексте разбиение на основе медианы трех элементов.
   а) Что в этом случае представляет собой массив из возрастающих элементов — наихудший случай, наилучший случай или ни то, ни другое?
   б) Дайте ответ на тот же вопрос для массива из убывающих элементов.
7. Решите рекуррентное соотношение для среднего случая алгоритма быстрой сортировки.
8. Разработайте такой алгоритм перестановки элементов в заданном массиве из n действительных чисел, чтобы все отрицательные числа предшествовали положительным. Алгоритм должен быть эффективен как в смысле времени работы, так и в смысле используемой памяти.
9. Реализуйте быструю сортировку на любом языке программирования, по вашему выбору. Выполните программу для различных входных дан-

ных, чтобы убедиться в справедливости теоретических предположений об эффективности алгоритма.

10. *Задача о гайках и болтах.* У вас имеется n болтов различного размера и n соответствующих гаек. Вы можете сравнивать гайку и болт и определять, подходят они друг к другу или гайка больше (или меньше) болта, но вы лишены возможности выполнить сравнение двух болтов или двух гаек между собой. Ваша задача состоит в том, чтобы разделить все болты и гайки по парам, в которых гайка по размеру будет соответствовать болту. Разработайте алгоритм для решения этой задачи за время $\Theta(n \log n)$ [94].

4.3 Бинарный поиск

Бинарный поиск представляет собой в высшей степени эффективный алгоритм для поиска в отсортированном массиве. Он работает путем сравнения искомого ключа K со средним элементом массива $A[m]$. Если они равны, алгоритм прекращает работу. В противном случае та же операция рекурсивно повторяется для первой половины массива, если $K < A[m]$, и для второй, если $K > A[m]$:



В качестве примера найдем ключ $K = 70$, применяя алгоритм бинарного поиска к массиву

3	14	27	31	39	42	55	70	74	81	85	93	98

Итерации алгоритма показаны в следующей таблице:

Индекс	0	1	2	3	4	5	6	7	8	9	10	11	12
Значение	3	14	27	31	39	42	55	70	74	81	85	93	98
Итерация 1	*l*						*m*						*r*
Итерация 2								*l*		*m*			*r*
Итерация 3								*l*, *m*	*r*				

Хотя ясно, что бинарный поиск основан на рекурсии, его очень легко реализовать как нерекурсивный алгоритм. Вот псевдокод нерекурсивной версии.

АЛГОРИТМ $BinarySearch\,(A\,[0..n-1]\,,K)$
// Реализует нерекурсивный бинарный поиск
// **Входные данные:** Массив $A[0..n-1]$, отсортированный
// в возрастающем порядке, и искомый ключ K
// **Выходные данные:** Индекс элемента массива, равного K,
// или -1, если такого элемента нет
$l \leftarrow 0$; $r \leftarrow n-1$;
while $l \leqslant r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$
    return m
  else if $K < A[m]$
    $r \leftarrow m-1$
  else
    $l \leftarrow m+1$
return -1

Стандартный способ анализа эффективности бинарного поиска состоит в подсчете количества сравнений искомого ключа с элементами массива. Кроме того, из соображений простоты, мы будем считать, что используются тройные сравнения, т.е. что одно сравнение K и $A\,[m]$ позволяет определить, меньше ли K, больше или равно значению $A\,[m]$.

Сколько же сравнений должен выполнить алгоритм при поиске в массиве из n элементов? Ответ, очевидно, зависит не только от n, но и от конкретного экземпляра задачи. Найдем количество сравнений $C_w\,(n)$ в наихудшем случае. В наихудший случай входят все массивы, которые не содержат искомого ключа (кроме них в этот разряд попадают и некоторые массивы, поиск в которых завершается успешно). Поскольку после одного сравнения алгоритм попадает в ту же ситуацию, что и до сравнения, только массив, в котором ведется поиск, становится вдвое меньше, можно записать следующее рекуррентное соотношение для $C_w\,(n)$:

$$C_w\,(n) = C_w\,(\lfloor n/2 \rfloor) + 1 \quad \text{для } n > 1, \quad C_w\,(1) = 1. \tag{4.2}$$

(Подумайте немного над тем, что значение $n/2$ следует округлять в меньшую сторону и что начальное условие именно таково, как указано в (4.2)).

Как уже говорилось в разделе 2.4, стандартный способ решения рекуррентных соотношений наподобие соотношения (4.2) состоит в том, что мы полагаем $n = 2^k$ и решаем получившееся соотношение путем обратной подстановки или при помощи какого-то другого метода. Читателю в качестве достаточно простого упражнения предлагается самостоятельно найти решение

$$C_w\left(2^k\right) = k + 1 = \log_2 n + 1. \tag{4.3}$$

Можно доказать, что решение (4.3) для $n = 2^k$ можно применить для получения корректного решения для любого положительного целого n:

$$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil . \tag{4.4}$$

Давайте убедимся с помощью непосредственной подстановки, что функция $C_w(n) = \lfloor \log_2 n \rfloor + 1$ действительно удовлетворяет уравнению (4.2) для любого положительного четного числа n. (Сделать то же самое для нечетных значений n предлагается в упражнении 4.3.3.) Если n положительно и четно, то $n = 2i$, где $i > 0$. Левая часть уравнения (4.2) при $n = 2i$ имеет вид

$$\begin{aligned} C_w(n) &= \lfloor \log_2 n \rfloor + 1 = \lfloor \log_2 2i \rfloor + 1 = \lfloor \log_2 2 + \log_2 i \rfloor + 1 = \\ &= (1 + \lfloor \log_2 i \rfloor) + 1 = \lfloor \log_2 i \rfloor + 2. \end{aligned}$$

Правая часть уравнения (4.2) при $n = 2i$ имеет вид

$$C_w(\lfloor n/2 \rfloor) + 1 = C_w(\lfloor 2i/2 \rfloor) + 1 = C_w(i) + 1 = (\lfloor \log_2 i \rfloor + 1) + 1 = \lfloor \log_2 i \rfloor + 2.$$

Оба выражения одинаковы, что и доказывает наше утверждение.

Формула (4.4) заслуживает особого внимания. Во-первых, из нее следует, что эффективность бинарного поиска в наихудшем случае равна $\Theta(\log n)$ (кстати говоря, этот вывод можно сделать, воспользовавшись Основной теоремой, но в этом случае мы не получили бы значения постоянного множителя). Во-вторых, полученный ответ именно таков, как и следовало ожидать: поскольку алгоритм на каждом шаге просто уменьшает размер оставшейся части массива вдвое, количество таких итераций для сведения массива из n элементов к единственному элементу должно быть примерно равно $\log_2 n$. В-третьих, как уже говорилось в разделе 2.1, логарифмическая функция растет очень медленно, так что ее значение остается небольшим даже при очень больших значениях n. В частности, согласно формуле (4.4), потребуется не более чем $\lfloor \log_2 10^3 \rfloor + 1 = 10$ тройных сравнений для поиска элемента с заданным значением (или выяснения того, что такой элемент отсутствует) в массиве из 1000 элементов, и не более $\lfloor \log_2 10^6 \rfloor + 1 = 20$ сравнений для поиска в массиве в миллион элементов!

Что можно сказать об эффективности бинарного поиска в среднем? Сложный анализ показывает, что среднее количество сравнений ключей в случае бинарного поиска только немного меньше такового в наихудшем случае:

$$C_{avg}(n) \approx \log_2 n.$$

(Более точные формулы для среднего количества сравнений в случае успешного и неудачного поиска — $C_{avg}^{yes}(n) \approx \log_2 n - 1$ и $C_{avg}^{no}(n) \approx \log_2 (n+1)$, соответственно.)

Если ограничиться только операцией сравнения ключей (см. раздел 10.2), то бинарный поиск является оптимальным алгоритмом, однако имеются алгоритмы поиска (см. интерполяционный поиск в разделе 5.6 и хеширование в разделе 7.3) с лучшим временем работы в среднем, а один из этих алгоритмов (хеширование) даже не требует, чтобы массив был отсортирован! Однако эти алгоритмы, кроме сравнения ключей, требуют некоторых специальных вычислений. И последнее: идея, лежащая в основе бинарного поиска, имеет ряд применений помимо поиска (см., например, [15]). Кроме того, она может быть использована для решения нелинейных уравнений (об этом мы поговорим в разделе 11.4).

Перед тем как завершить раздел, следует сделать еще одно замечание по поводу бинарного поиска. Иногда бинарный поиск представляют квинтэссенцией алгоритма, основанного на декомпозиции. Такая интерпретация некорректна, поскольку бинарный поиск на самом деле — весьма нетипичный случай метода декомпозиции. В самом деле, в соответствии с определением, данным в начале главы, метод декомпозиции делит задачу на *несколько* подзадач, каждая из которых требует решения. В случае же бинарного поиска необходимо решить только одну из двух подзадач. Таким образом, если рассматривать бинарный поиск как алгоритм декомпозиции, то его следует трактовать как вырожденный случай этого метода. В действительности бинарный поиск в большей степени соответствует классу алгоритмов деления пополам, которые мы рассмотрим в разделе 5.5. Почему же в таком случае мы решили описать бинарный поиск в этой главе? Частично — по традиции, частично — потому что плохой пример иногда в состоянии показать то, что не в состоянии показать хороший пример.

Упражнения 4.3

1. Дан массив

3	14	27	31	39	42	55	70	74	81	85	93	98

.

а) Чему равно максимальное количество сравнений при бинарном поиске в этом массиве?

б) Перечислите ключи массива, при поиске которых будет выполняться максимальное количество сравнений.

в) Найдите среднее количество сравнений, выполняемых при успешном бинарном поиске в этом массиве; предполагается, что вероятности поиска каждого ключа одинаковы.

г) Найдите среднее количество сравнений, выполняемых при неудачном бинарном поиске в этом массиве; предполагается, что вероятность поиска для каждого из 14 интервалов, образуемых элементами массива, одинакова.

2. Решите рекуррентное уравнение $C_w(n) = C_w(\lfloor n/2 \rfloor) + 1$ при $n > 1$, $C_w(1) = 1$ для $n = 2^k$ методом обратной подстановки.
3. **а)** Докажите равенство

$$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil \quad \text{при } n \geqslant 1.$$

   б) Докажите, что функция $C_w(n) = \lfloor \log_2 n \rfloor + 1$ является решением рекуррентного уравнения (4.2) для положительных нечетных n.
4. Оцените, во сколько раз быстрее в среднем будет выполняться успешный бинарный поиск в отсортированном массиве из 100 000 элементов по сравнению с последовательным поиском.
5. Последовательный поиск дает практически одинаковую эффективность при поиске в массиве и в связанном списке. Справедливо ли аналогичное утверждение для бинарного поиска? (Естественно, мы предполагаем, что список отсортирован.)
6. Как с помощью бинарного поиска найти диапазон, т.е. все элементы отсортированного массива, которые находятся между заданными значениями L и U включительно ($L \leqslant U$)? Какова эффективность этого алгоритма в наихудшем случае?
7. Напишите псевдокод рекурсивной версии бинарного поиска.
8. Разработайте версию бинарного поиска, где бы использовались обычные сравнения, такие как $\leqslant$ и $=$. Реализуйте алгоритм на любом языке программирования и тщательно отладьте его (обычно при написании таких программ образуется повышенное количество ошибок).
9. Проанализируйте временну́ю эффективность версии бинарного поиска с обычными сравнениями, предложенной в упражнении 8.

10. Перед вами разложены 42 рисунка в семь рядов, по 6 рисунков в каждом. Необходимо определить загаданный партнером рисунок, задавая вопросы, на которые можно получить ответ "да" или "нет". Ваша задача — предложить максимально эффективный алгоритм решения этой задачи и указать максимальное количество вопросов, которое потребуется задать, чтобы гарантированно определить рисунок.

4.4 Обход бинарного дерева

В этом разделе мы рассмотрим, каким образом метод декомпозиции может быть применен к бинарным деревьям. ***Бинарное дерево*** T определяется как конечное множество узлов, которое может быть либо пустым, либо состоит из корня

и двух непересекающихся бинарных деревьев T_L и T_R, именуемых, соответственно, левым и правым поддеревьями корня. Обычно мы говорим о бинарном дереве как о частном случае упорядоченного дерева (рис. 4.4). (Соответствующее определение бинарного дерева было дано в разделе 1.4.)

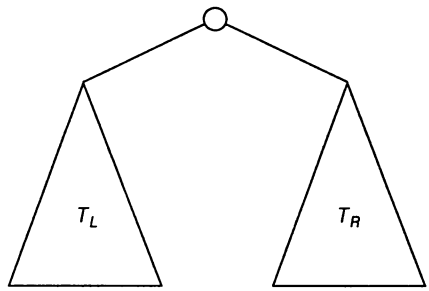


Рис. 4.4. Стандартное представление бинарного дерева

Поскольку по определению бинарное дерево делится на две меньшие структуры такого же типа — левое поддерево и правое поддерево, многие задачи, связанные с бинарными деревьями, могут решаться при помощи метода декомпозиции. В качестве примера рассмотрим рекурсивный алгоритм определения высоты бинарного дерева. Вспомним, что высота дерева определяется как длина самого длинного пути от корня к листу. Следовательно, ее можно вычислить, выбрав максимальное значение среди высот левого и правого поддеревьев корня и прибавив к нему 1 (чтобы учесть дополнительный уровень корня). Заметим также, что удобно определить высоту пустого дерева равной -1. Итак, мы получили следующий рекурсивный алгоритм:

Алгоритм $Height\,(T)$

```
// Рекурсивное вычисление высоты бинарного дерева
// Входные данные:  Бинарное дерево T
// Выходные данные: Высота бинарного дерева T
if T = ∅
   return −1
else
   return max{Height(T_L), Height(T_R)} + 1
```

Размер экземпляра данной задачи измеряется количеством узлов $n\,(T)$ бинарного дерева T. Очевидно, что количество сравнений, выполняемых для вычисления максимального из двух чисел, и количество сложений $A\,(n\,(T))$, выполняемое алгоритмом, одинаково. Для $A\,(n\,(T))$ можно записать следующее рекуррентное уравнение:

$$A\,(n\,(T)) = A\,(n\,(T_L)) + A\,(n\,(T_R)) + 1 \quad \text{при } n\,(T) > 0, A\,(0) = 0.$$

Прежде чем решать уравнение (можете ли вы сказать, каким будет его решение?), заметим, что сложение — не самая часто выполняемая операция в данном алгоритме. А какая же? Проверка, не является ли бинарное дерево пустым (кстати, это весьма типично для алгоритмов, работающих с бинарными деревьями). Например, для пустого дерева сравнение $T = \varnothing$ выполняется один раз, в то время как сложение в этом случае не выполняется вовсе. Для дерева из одного узла количество сравнений и сложений равны, соответственно, трем и одному.

При анализе алгоритмов, работающих с бинарными деревьями, зачастую помогает изображение расширения дерева, в котором пустые поддеревья заменены специальными узлами. Такие дополнительные узлы, изображенные на рис. 4.5 квадратами, называются ***внешними*** (external); исходные узлы (изображенные в виде кругов) называются ***внутренними*** (internal). По определению расширение пустого бинарного дерева представляет собой единственный внешний узел.

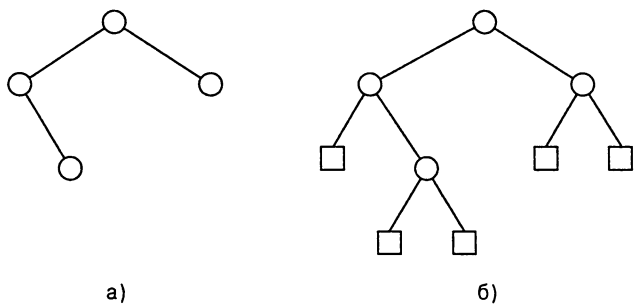


Рис. 4.5. *а*) Бинарное дерево. *б*) Его расширение. Внутренние узлы представлены кругами, внешние — квадратами

Легко увидеть, что алгоритм поиска высоты бинарного дерева выполняет ровно по одному сложению для каждого внутреннего узла расширенного дерева, и по одной проверке на пустоту дерева для каждого внутреннего и внешнего узла. Таким образом, для определения эффективности алгоритма надо знать, сколько внешних узлов может иметь расширенное дерево с n внутренними узлами. Рассмотрев рис. 4.5 и несколько аналогичных примеров расширенных деревьев, легко выдвинуть гипотезу (которую несложно доказать), что количество внешних узлов x всегда на один больше количества внутренних узлов n:

$$x = n + 1. \tag{4.5}$$

Докажем это равенство по индукции по количеству внутренних узлов $n \geqslant 0$. Базис индукции справедлив, поскольку при $n = 0$ мы имеем пустое дерево с одним внешним узлом по определению. Положим в общем случае, что $x = k + 1$ для любого расширенного дерева с $0 \leqslant k < n$ внутренними узлами. Пусть T –

расширенное бинарное дерево с n внутренними и x внешними узлами, n_L и x_L — количество, соответственно, внутренних и внешних узлов в левом поддереве T, а n_R и x_R — количество внутренних и внешних узлов в правом поддереве T, соответственно. Поскольку $n > 0$, бинарное дерево T имеет корень, который является внутренним узлом, и, следовательно, $n = n_L + n_R + 1$. Полагая гипотезу индукции справедливой для левого и правого поддеревьев, находим количество внешних узлов дерева T:

$$x = x_L + x_R = (n_L + 1) + (n_R + 1) = (n_L + n_R + 1) + 1 = n + 1,$$

что и завершает доказательство.

Возвращаясь к алгоритму $Height$, находим, что количество сравнений, выполняемых для проверки пустоты дерева, равно

$$C(n) = n + x = 2n + 1,$$

а количество сложений равно

$$A(n) = n.$$

Наиболее важные алгоритмы бинарных деревьев — это три классических обхода бинарных деревьев — в прямом порядке (preorder traversal), симметричный, или центрированный (inorder traversal), и в обратном порядке (postorder traversal). При всех указанных обходах узлы дерева посещаются рекурсивно, т.е. посещаются корень и его левое и правое поддеревья, и отличие обходов только в последовательности посещений.

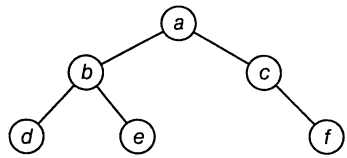
- При обходе в ***прямом порядке*** сначала посещается корень дерева, а затем левое и правое поддеревья (в указанном порядке).
- При ***симметричном обходе*** корень посещается после левого поддерева, но перед посещением правого.
- При обходе в ***обратном порядке*** корень посещается после левого и правого поддеревьев (в указанном порядке).

Псевдокод всех описанных обходов бинарного дерева очень прост (эти обходы к тому же рассматриваются в любом учебнике по структурам данных). Что же касается анализа эффективности, то он идентичен только что проведенному, поскольку рекурсивный вызов выполняется для каждого узла расширенного бинарного дерева.

Напоследок следует заметить, что не все задачи, связанные с бинарными деревьями, требуют обхода обоих поддеревьев. Например, поиск и вставка в бинарное дерево поиска требуют обработки только одного из двух поддеревьев. Следовательно, они должны рассматриваться не как приложения метода декомпозиции, а как примеры метода уменьшения на переменную величину, рассматриваемого в разделе 5.6.

Упражнения 4.4

1. Разработайте декомпозиционный алгоритм для вычисления количества уровней в бинарном дереве (в частности, алгоритм должен возвращать 0 для пустого дерева и 1 для дерева с одним узлом). К какому классу эффективности относится алгоритм?
2. Обойдите приведенное ниже дерево в прямом, центрированном и обратном порядке. Приведите содержимое стека обхода в процессе работы алгоритмов.



3. Напишите псевдокод для одного из классических алгоритмов обхода бинарного дерева (в прямом, центрированном или обратном порядке). Считая, что рассматриваемый алгоритм рекурсивен, найдите количество выполняемых рекурсивных вызовов.
4. Чему равно наибольшее количество узлов, которые могут одновременно находиться в стеке при симметричном обходе бинарного дерева? Опишите вид деревьев, для которых этот максимум достигается на практике.
5. В каком порядке — прямом, центрированном, обратном или никаком из перечисленных — алгоритм поиска высоты дерева, описанный в данном разделе, вычисляет высоты всех поддеревьев входного дерева?
6. Какой из трех классических алгоритмов обхода дерева приводит к отсортированному списку при применении к бинарному дереву поиска? Докажите это свойство указанного вами алгоритма обхода.
7. В корневом упорядоченном дереве каждый внутренний узел имеет положительное количество упорядоченных потомков, каждый из которых служит корнем упорядоченного поддерева. Разработайте алгоритм для вычисления высоты корневого упорядоченного дерева. (Предполагается, что наименьшее корневое упорядоченное дерево состоит из одного узла, а его высота равна 0.)
8. Приведенный далее алгоритм предназначен для вычисления количества листьев бинарного дерева.

АЛГОРИТМ $LeafCounter\,(T)$
// Рекурсивно вычисляет количество листьев бинарного дерева
// **Входные данные:** Бинарное дерево T
// **Выходные данные:** Количество листьев в T
if $T = \varnothing$
  return 0
else
  return $LeafCounter(T_L) + LeafCounter(T_R)$

Корректен ли приведенный алгоритм? Если да, докажите это; если нет, внесите необходимые изменения.

9. ***Длина внутреннего пути*** (internal path length) I расширенного бинарного дерева определяется как сумма длин путей от корня к внутреннему узлу, взятая по всем внутренним узлам. Аналогично, ***длина внешнего пути*** (external path length) E расширенного бинарного дерева определяется как сумма длин путей от корня к внешнему узлу, взятая по всем внешним узлам. Докажите, что $E = I + 2n$, где n — количество внутренних узлов в дереве.

10. Напишите программу для вычисления длины внутреннего пути бинарного дерева поиска. Воспользуйтесь ею для эмпирического исследования среднего количества сравнения ключей при поиске в случайно сгенерированном бинарном дереве поиска.

4.5 Умножение больших целых чисел и алгоритм умножения матриц Штрассена

В этом разделе мы рассмотрим два удивительных алгоритма для кажущихся простыми задач: умножения двух чисел и умножения двух квадратных матриц. Оба они пытаются уменьшить количество выполняемых умножений ценой небольшого увеличения количества сложений; оба используют метод декомпозиции.

Умножение больших целых чисел

В некоторых приложениях, особенно в современной криптологии, требуется работать с целыми числами, длина которых превышает 100 десятичных цифр. Понятно, что такое число слишком велико, чтобы разместить его в одном слове

современного компьютера, так что для работы с ним требуются специальные подходы. В этом разделе мы рассмотрим интересный алгоритм для перемножения таких больших чисел. Очевидно, что если мы используем классический алгоритм умножения двух n-значных чисел в столбик, то каждая из n цифр одного числа будет умножаться на каждую из n цифр второго числа, что в результате дает нам n^2 умножений цифр. (Если у одного числа количество цифр меньше, чем у второго, впереди к нему можно дописать некоторое количество нулей, чтобы длина чисел стала одинаковой.) Казалось бы, невозможно разработать алгоритм, выполняющий менее n^2 умножений цифр, но это не так. Метод декомпозиции способен творить чудеса.

Для демонстрации идеи, лежащей в основе алгоритма, рассмотрим умножение двух двузначных чисел, скажем, 23 и 14. Эти числа можно представить следующим образом:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \text{ и } 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Перемножим их:

$$\begin{aligned} 23 \cdot 14 &= \left(2 \cdot 10^1 + 3 \cdot 10^0\right) \cdot \left(1 \cdot 10^1 + 4 \cdot 10^0\right) = \\ &= (2 \cdot 1) \cdot 10^2 + (3 \cdot 1 + 2 \cdot 4) \cdot 10^1 + (3 \cdot 4) \cdot 10^0. \end{aligned}$$

Приведенная формула, конечно, дает верный ответ — 322, но в ней используются те же четыре умножения цифр, что и при умножении в столбик. К счастью, можно вычислить средний член при помощи только одного умножения, воспользовавшись тем, что нам известны произведения $(2 \cdot 1)$ и $(3 \cdot 4)$, которые все равно будут вычислены:

$$3 \cdot 1 + 2 \cdot 4 = (2 + 3) \cdot (1 + 4) - (2 \cdot 1) - (3 \cdot 4).$$

Конечно, в числах, которые мы использовали, нет ничего особенного. Для любой пары двузначных чисел $a = a_1a_0$ и $b = b_1b_0$ их произведение c можно вычислить по формуле

$$c = a \cdot b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0,$$

где

$c_2 = a_1 \cdot b_1$ — произведение первых цифр;

$c_0 = a_0 \cdot b_0$ — произведение вторых цифр;

$c_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0)$ — произведение суммы цифр a на сумму цифр b минус сумма c_2 и c_0.

Теперь применим этот трюк к произведению двух n-значных чисел a и b, где n — положительное четное число. Разделим числа пополам — в конце концов, мы же предупреждали, что будем использовать метод декомпозиции. Используем для первой половины цифр a запись a_1, а для второй — a_0. Для половин числа b будут использованы соответственно обозначения b_1 и b_0. При использовании такой записи из $a = a_1a_0$ вытекает $a = a_1 \cdot 10^{n/2} + a_0$, а из $b = b_1b_0 - b = b_1 \cdot 10^{n/2} + b_0$. Таким образом, воспользовавшись описанным ранее методом, для произведения чисел a и b получим

$$\begin{aligned} c = a \cdot b &= \left(a_1 \cdot 10^{n/2} + a_0\right) \cdot \left(b_1 \cdot 10^{n/2} + b_0\right) = \\ &= (a_1 \cdot b_1) \cdot 10^n + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 10^{n/2} + (a_0 \cdot b_0) = \\ &= c_2 \cdot 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

где

$c_2 = a_1 \cdot b_1$ — произведение первых половин;

$c_0 = a_0 \cdot b_0$ — произведение вторых половин;

$c_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0)$ — произведение суммы половин a на сумму половин b минус сумма c_2 и c_0.

Если $n/2$ четно, тот же способ можно применить и для вычисления произведений c_2, c_1 и c_0. Следовательно, если n представляет собой степень 2, получаем рекурсивный алгоритм для вычисления произведения двух n-значных целых чисел. В чистом виде рекурсия завершается, когда n становится равным единице, но ее можно прекратить и раньше — когда n станет достаточно малым для непосредственного перемножения чисел такого размера.

Сколько умножений цифр выполняется в данном алгоритме? Поскольку перемножение n-значных чисел требует трех умножений $n/2$-значных чисел, рекуррентное соотношение для количества умножений $M(n)$ имеет вид

$$M(n) = 3M(n/2) \quad \text{при } n > 1, M(1) = 1.$$

Решение этого уравнения методом обратной подстановки для $n = 2^k$ дает

$$\begin{aligned} M\left(2^k\right) &= 3M\left(2^{k-1}\right) = 3\left[3M\left(2^{k-2}\right)\right] = 3^2M\left(2^{k-2}\right) = \\ &= \cdots = 3^iM\left(2^{k-i}\right) = \cdots = 3^kM\left(2^{k-k}\right) = 3^k. \end{aligned}$$

Поскольку $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(На последнем шаге преобразований мы воспользовались тем свойством логарифмов, что $a^{\log_b c} = c^{\log_b a}$.)

Не следует забывать, что для чисел средней длины этот алгоритм, вероятно, будет работать дольше, чем классический. Брассард (Brassard) и Брейтли (Bratley) ([22], стр.70–71) пишут, что в их экспериментах алгоритм декомпозиции превосходил метод умножения в столбик только для чисел длиной более 600 цифр. Если вы работаете на Java, C++ или Smalltalk, то учтите, что для этих языков имеются специальные классы для работы с большими целыми числами.

Алгоритм Штрассена для умножения матриц

Теперь, когда мы продемонстрировали, как применение метода декомпозиции позволяет снизить количество умножений цифр при перемножении больших целых чисел, вас не должен удивлять подобный эффект при перемножении матриц. Такой алгоритм был опубликован в 1969 году Штрассеном (V. Strassen) [113]. Основная идея, лежащая в основе этого алгоритма, заключается в открытии, что произведение C двух матриц A и B размером 2×2 можно вычислить с помощью только 7, а не 8 умножений, которые необходимы при использовании алгоритма грубой силы, описанного в примере 3 раздела 2.3. Этого можно достичь, используя следующие формулы:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} =$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

где

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) \cdot (b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11}) \cdot b_{00} \\ m_3 &= a_{00} \cdot (b_{01} - b_{11}) \\ m_4 &= a_{11} \cdot (b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01}) \cdot b_{11} \\ m_6 &= (a_{10} - a_{00}) \cdot (b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11}) \cdot (b_{10} + b_{11}) . \end{aligned}$$

Таким образом, для умножения двух матриц размером 2×2 алгоритм Штрассена выполняет семь умножений и 18 сложений/вычитаний, в то время как алгоритм на основе грубой силы требует восьми умножений и четырех сложений. Приведенное выше вовсе не означает, что вы должны использовать алгоритм Штрассена для умножения матриц размером 2×2 — его важность в *асимптотическом* превосходстве над алгоритмом грубой силы, когда порядок перемножаемых матриц стремится к бесконечности.

Пусть A и B — две матрицы размером $n \times n$, где n — степень двойки. (Если n не является степенью двойки, можно добавить к матрицам необходимое количество строк и столбцов, заполненных нулями.) Матрицы A и B и их произведение C можно разделить на четыре подматрицы размером $n/2 \times n/2$ каждую следующим образом:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].$$

Нетрудно убедиться, что для получения корректного произведения эти подматрицы можно рассматривать как числа. Например, C_{00} можно вычислить как $A_{00} \cdot B_{00} + A_{01} \cdot B_{10}$ или как $M_1 + M_4 - M_5 + M_7$, где M_1, M_4, M_5 и M_7 вычисляются по формулам Штрассена, в которых числа заменены соответствующими подматрицами. Алгоритм Штрассена состоит в том, что требуемые семь произведений подматриц размером $n/2 \times n/2$ вычисляются рекурсивно с использованием описанного метода.

Давайте определим асимптотическую эффективность данного алгоритма. Если $M(n)$ — количество умножений, выполняемых алгоритмом Штрассена для умножения двух матриц размером $n \times n$ (где n — степень двойки), то мы получим следующее рекуррентное соотношение для $M(n)$:

$$M(n) = 7M(n/2) \quad \text{при } n > 1, M(1) = 1.$$

Поскольку $n = 2^k$,

$$M\left(2^k\right) = 7M\left(2^{k-1}\right) = 7\left[7M\left(2^{k-2}\right)\right] = 7^2 M\left(2^{k-2}\right) = \cdots =$$
$$= 7^i M\left(2^{k-i}\right) = \cdots = 7^k M\left(2^{k-k}\right) = 7^k.$$

Подставляя $k = \log_2 n$, получаем:

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

что меньше, чем n^3, необходимое для алгоритма на основе грубой силы.

Поскольку экономия количества умножений достигается ценой большего количества сложений, следует рассмотреть количество сложений $A(n)$, выполняемых алгоритмом Штрассена. Для умножения двух матриц порядка $n > 1$ алгоритму требуется семь умножений и 18 сложений матриц размером $n/2 \times n/2$; при $n = 1$ сложений вообще не требуется, поскольку в этом случае задача вырождается в перемножение двух чисел. Все это приводит к следующему рекуррентному уравнению:

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{при } n > 1, A(1) = 0.$$

Хотя можно получить точное решение этого уравнения (см. упражнение 4.5.8), в настоящий момент нас интересует только порядок роста решения этого соотношения. Согласно Основной теореме, приведенной в начале главы, $A(n) \in \in \Theta\left(n^{\log_2 7}\right)$. Другими словами, количество сложений имеет тот же порядок роста, что и количество умножений. Таким образом, можно сделать вывод о том, что эффективность алгоритма Штрассена — $\Theta\left(n^{\log_2 7}\right)$, что лучше, чем эффективность алгоритма на основе грубой силы $\Theta\left(n^3\right)$.

После Штрассена открыт ряд других алгоритмов для перемножения матриц действительных чисел размером $n \times n$ за время $O\left(n^\alpha\right)$ с постоянно уменьшающимся значением показателя α. Наиболее быстрый алгоритм был предложен Куперсмитом (Coopersmith) и Виноградом (Winograd) [31] — его эффективность равна $\Theta\left(n^{2.376}\right)$. Значение показателя уменьшается за счет увеличения сложности алгоритмов. Из-за большой величины постоянного множителя в формуле времени работы алгоритма все предложенные алгоритмы умножения матриц не имеют практической ценности, но представляют большой теоретический интерес. С одной стороны, новые алгоритмы приближаются к теоретическому пределу, который равен n^2 умножений, но точная величина "зазора" между теоретическим пределом и наилучшим из возможных алгоритмом остается неизвестна. С другой стороны, умножение матриц представляет собой вычислительный эквивалент некоторых других важных задач, например, таких, как решение систем линейных уравнений.

Упражнения 4.5

1. Чему равно наименьшее и наибольшее возможное количество цифр в произведении двух n-значных чисел?
2. Вычислите $2101 \cdot 1130$ с применением описанного в тексте алгоритма декомпозиции.
3. **а)** Докажите тождество $a^{\log_b c} = c^{\log_b a}$, которое дважды было использовано в разделе 4.5.
   б) Почему запись $n^{\log_2 3}$ лучше записи $3^{\log_2 n}$ в решении уравнения для $M(n)$?
4. **а)** Почему умножение на 10^n не включается в общее количество умножений $M(n)$ в алгоритме перемножения больших целых чисел?
   б) Кроме предположения о том, что n является степенью 2, при написании рекуррентного соотношения для $M(n)$ мы делаем для простоты еще одно, более тонкое предположение, которое не всегда истинно (но это не влияет на конечный ответ). Что это за предположение?
5. Сколько сложений цифр делается при умножении двух n-значных чисел в столбик? (Переносами можно пренебречь.)

6. Проверьте справедливость формул, лежащих в основе алгоритма Штрассена для перемножения матриц размером 2×2.
7. Примените алгоритм Штрассена для вычисления

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}.$$

   Рекурсию при $m = 2$ следует прекратить, т.е. перемножение матриц размером 2×2 выполняется с помощью алгоритма на основе грубой силы.
8. Решите рекуррентное уравнение для количества сложений, выполняемых алгоритмом Штрассена (в предположении, что n является степенью 2).
9. Пан (V. Pan) [85] открыл метод декомпозиции для умножения матриц, основанный на перемножении матриц размером 70×70 при помощи 143 640 умножений чисел. Найдите асимптотическую эффективность алгоритма Пана (сложения можете игнорировать) и сравните ее с эффективностью алгоритма Штрассена.
10. Практические реализации алгоритма Штрассена обычно переключаются на использование алгоритма, основанного на грубой силе, когда размеры матриц становятся меньше некоторой "точки пересечения". Проведите вычислительный эксперимент и определите точку пересечения для вашего компьютера.

4.6 Решение задач о паре ближайших точек и о выпуклой оболочке методом декомпозиции

В разделе 3.3 мы рассматривали основанные на грубой силе алгоритмы для решения двух классических задач вычислительной геометрии: задачу о паре ближайших точек и о выпуклой оболочке. Вы видели, что двумерные версии этих задач могут быть решены методом грубой силы за время $\Theta\left(n^2\right)$ и $O\left(n^3\right)$ соответственно. Здесь мы рассмотрим более сложные и асимптотически более эффективные алгоритмы решения указанных задач, основанные на методе декомпозиции.

Задача о паре ближайших точек

Пусть $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$ — множество S, состоящее из n точек на плоскости (для простоты будем считать, что n является степенью двойки). Без потери общности можно считать, что точки упорядочены в соответствии с возрастанием их координаты x (если это не так, то их можно упорядочить за время $O(n \log n)$, например, при помощи сортировки слиянием). Точки можно разделить на два подмножества S_1 и S_2, состоящие из $n/2$ точек каждое, проведя вертикальную линию $x = c$ так, чтобы слева и справа от нее оказалось по $n/2$ точек (один из способов поиска значения константы c состоит в использовании медианы μ координат x точек).

Следуя методу декомпозиции можно рекурсивно найти пары ближайших точек в левом подмножестве S_1 и в правом подмножестве S_2. Пусть d_1 и d_2 — наименьшие расстояния между парами точек в подмножествах S_1 и S_2, соответственно. Обозначим $d = \min\{d_1, d_2\}$. К сожалению, d не обязательно представляет собой наименьшее расстояние между парами точек в S_1 и S_2, поскольку пара точек с минимальным расстоянием между ними может лежать по разные стороны от разделяющей линии. Таким образом, этап комбинирования должен включать рассмотрение таких точек. Очевидно, что можно ограничиться рассмотрением точек, попадающих в вертикальную полосу шириной $2d$, поскольку расстояние между любыми другими парами точек по разные стороны от разделяющей линии заведомо превосходит расстояние d (рис. 4.6*а*). Пусть C_1 и C_2 — подмножества точек в левой и правой части полосы, соответственно.

Теперь для каждой точки $P(x, y)$ из C_1 мы должны рассмотреть точки в C_2, которые могут оказаться от точки P на расстоянии, меньшем d. Совершенно очевидно, что координаты y таких точек не могут выходить за пределы интервала $[y - d, y + d]$. Наиболее важным является то, что таких точек может быть не более шести, поскольку любая пара точек в C_2 находится как минимум на расстоянии d друг от друга. (Вспомним, что $d \leqslant d_2$, где d_2 — наименьшее расстояние между точками справа от разделяющей линии.) Наихудший случай проиллюстрирован на рис. 4.6*б*.

Второе важное наблюдение заключается в том, что можно поддерживать списки точек в C_1 и C_2 отсортированными в порядке возрастания их координат y (эти списки можно рассматривать как проекции точек на разделяющую линию). Кроме того, такой порядок можно поддерживать не путем пересортировки точек при каждой итерации, а слиянием двух ранее отсортированных списков (см. алгоритм *Merge* из раздела 4.1). Можно рассматривать точки C_1, в то время как указатель в список C_2 будет выполнять осцилляции в пределах отрезка $2d$, выбирая шесть кандидатов, для которых будут вычисляться расстояния до текущей точки P из списка C_1. Время $M(n)$, необходимое для "слияния" решений меньших подзадач, равно $O(n)$.

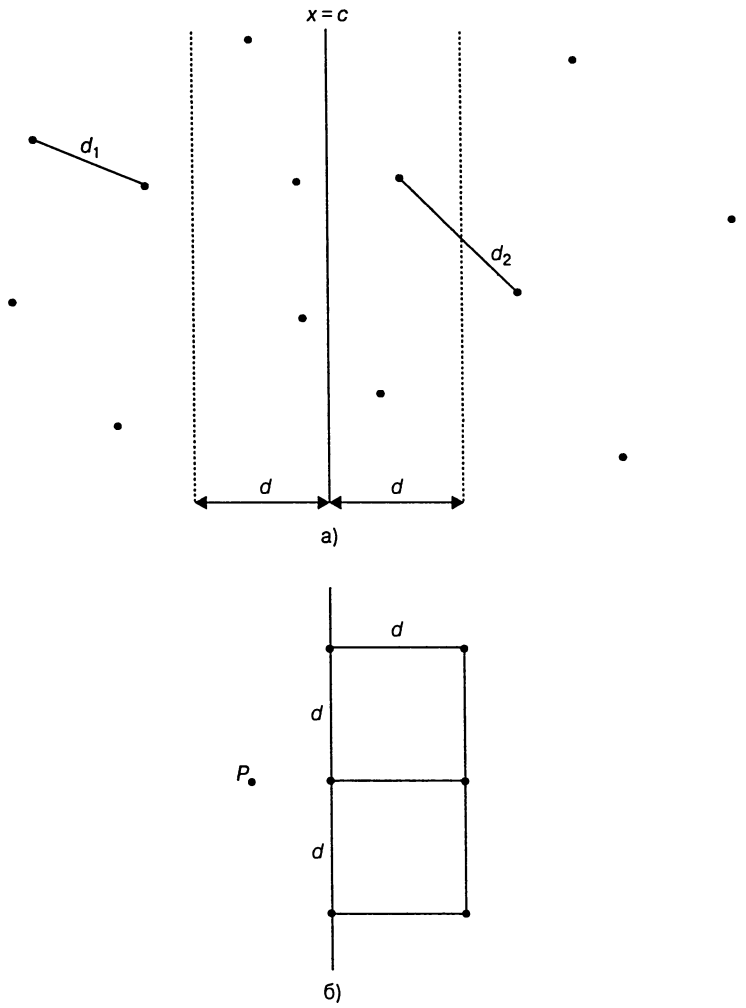


Рис. 4.6. *а*) Идея метода декомпозиции для решения задачи о паре ближайших точек. *б*) Шесть точек, которые может потребоваться рассмотреть для точки P

В результате рекуррентное соотношение для времени работы $T(n)$ описанного алгоритма над n предварительно отсортированными точками имеет вид:

$$T(n) = 2T(n/2) + M(n).$$

Применяя O-версию Основной теоремы (для $a = 2$, $b = 2$ и $d = 1$, получаем, что $T(n) \in O(n \log n)$. Возможная необходимость предварительной сортировки точек не меняет класс эффективности алгоритма, поскольку может быть выполнена за время $O(n \log n)$. Мы получили алгоритм с наивысшим классом эффек-

тивности для данной задачи, так как было доказано, что любой алгоритм для ее решения принадлежит к $\Omega(n \log n)$ (см. [90], стр. 188).

Задача о выпуклой оболочке

Для решения задачи о выпуклой оболочке, в которой требуется найти минимальный выпуклый многоугольник, содержащий n точек на плоскости, на самом деле имеется несколько алгоритмов, основанных на методе декомпозиции. Мы рассмотрим простейший из них, который иногда называют ***быстрой оболочкой*** (quickhull) по аналогии с быстрой сортировкой, работу которой он напоминает.

Пусть $P_1 = (x_1, y_1), \ldots, P_n(x_n, y_n)$ — множество из n точек на плоскости. Мы считаем, что эти точки отсортированы в возрастающем порядке по координате x, а в случае совпадения координаты x — в возрастающем порядке по координате y. Нетрудно доказать геометрически очевидный факт, что точки, крайние слева P_1) и справа P_n), должны принадлежать множеству вершин выпуклой оболочки (рис. 4.7). Пусть $\overrightarrow{P_1P_n}$ — прямая линия, проведенная от точки P_1 к точке P_n. Эта линия делит множество точек на два подмножества: S_1 — множество точек, лежащих слева от линии или на ней, и S_2 — множество точек, которые лежат справа от линии. (Мы говорим, что точка p_3 находится слева от линии $\overrightarrow{p_1p_2}$, проведенной от точки p_1 к точке p_2, если точки $p_1p_2p_3$ образуют цикл против часовой стрелки. Позже мы укажем аналитический способ проверки этого условия, основанный на определении знака детерминанта, образуемого координатами этих трех точек.)

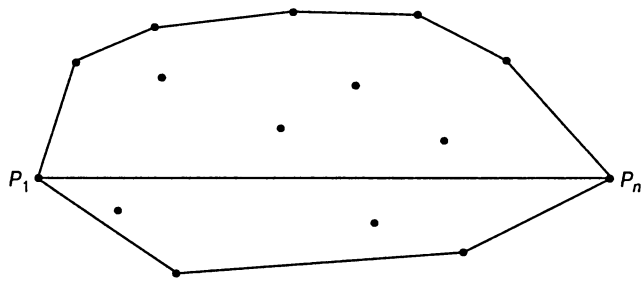


Рис. 4.7. Верхняя и нижняя оболочки множества точек

Выпуклая оболочка множества S_1 состоит из отрезка с конечными точками P_1 и P_n и верхней границы, представляющей собой ломаную линию, состоящую из сторон многоугольника, т.е. из последовательности отрезков, соединяющих некоторые точки S_1. Верхняя граница называется ***верхней оболочкой*** (upper hull). Аналогично, ломаная линия, служащая нижней границей выпуклой оболочки S_2, называется ***нижней оболочкой*** (lower hull). Тот факт, что выпуклая оболочка всего множества S состоит из верхней и нижней оболочек, которые могут быть

построены независимо одним и тем же способом, оказывается очень важным и используется в ряде алгоритмов решения данной задачи.

Для конкретности рассмотрим, как алгоритм быстрой оболочки строит верхнюю оболочку; нижняя оболочка строится аналогично. Во-первых, алгоритм определяет точку $P_{\max}$ в S_1, наиболее удаленную от линии $\overrightarrow{P_1P_n}$ (рис. 4.8). Если таких точек несколько, в качестве $P_{\max}$ выбирается та, для которой угол $\angle P_{\max}P_1P_n$ максимален. (Заметим, что если мы будем рассматривать все треугольники, две вершины которых — P_1 и P_n, а третья — некоторая точка из множества S_1, то при выборе в качестве третьей вершины $P_{\max}$ площадь треугольника будет максимальна.) После этого алгоритм определяет, какие точки множества S_1 находятся слева от линии $\overrightarrow{P_1P_{\max}}$; эти точки вместе с точками P_1 и $P_{\max}$ образуют множество $S_{1,1}$. Точки S_1, лежащие слева от линии $\overrightarrow{P_{\max}P_n}$, вместе с точками $P_{\max}$ и P_n образуют множество $S_{1,2}$. Нетрудно доказать, что не существует точек, лежащих слева от обеих линий. Точки внутри треугольника $\triangle P_1P_{\max}P_n$ можно в дальнейшем не рассматривать. Таким образом, алгоритм может рекурсивно продолжать построение верхней оболочки $S_{1,1}$ и $S_{1,2}$, а затем просто соединить их для получения верхней оболочки множества S_1.

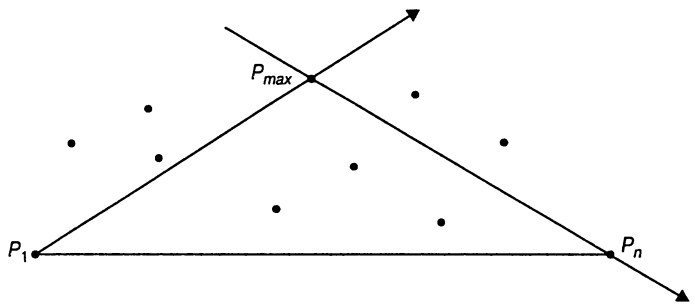


Рис. 4.8. Идея алгоритма быстрой оболочки

Теперь выясним, как реализуются геометрические операции алгоритма. К счастью, мы можем воспользоваться следующим фактом из аналитической геометрии: если $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ и $p_3 = (x_3, y_3)$ — три произвольные точки на плоскости, то площадь треугольника $\triangle p_1p_2p_3$ равна половине абсолютного значения детерминанта

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3,$$

а само это выражение положительно тогда и только тогда, когда точка $p_3 = (x_3, y_3)$ находится слева от линии $\overrightarrow{p_1p_2}$. Воспользовавшись этой формулой, можно за фик-

сированное время определить, лежит ли точка слева от линии, образованной двумя другими точками, и найти расстояние от точки до этой линии.

Алгоритм быстрой оболочки имеет ту же эффективность, что и алгоритм быстрой сортировки: $\Theta(n \log n)$ в среднем, и $\Theta(n^2)$ — в наихудшем случае. Хотя это и выше эффективности алгоритма, основанного на грубой силе (который был рассмотрен нами в разделе 3.3), имеются, как уже упоминалось, и более сложные алгоритмы для решения задачи о выпуклой оболочке, время работы которых в наихудшем случае составляет $\Theta(n \log n)$ (см., например, [90]).

Упражнения 4.6

1. **а)** Разработайте на основе метода декомпозиции алгоритм решения одномерной версии задачи о паре ближайших точек, т.е. задачи поиска пары ближайших чисел во множестве из n действительных чисел. Определите класс эффективности предложенного вами алгоритма.
   б) Насколько хорош этот алгоритм для решения поставленной задачи?
2. Рассмотрим еще одну версию алгоритма декомпозиции для решения двумерной задачи о паре ближайших точек. В нем мы просто заново сортируем каждое из двух множеств C_1 и C_2 в возрастающем порядке по координате y при каждом рекурсивном вызове. Полагая, что сортировка выполняется при помощи алгоритма сортировки слиянием, запишите приблизительное рекуррентное соотношение для времени работы такого алгоритма в наихудшем случае и решите его для $n = 2^k$.
3. Реализуйте описанный в данном разделе алгоритм декомпозиции для поиска пары ближайших точек на своем любимом языке программирования.
4. Найдите в Web визуализацию алгоритма решения задачи поиска пары ближайших точек. Какой алгоритм представляет найденная вами визуализация?
5. ***Многоугольник Вороного*** для точки P из множества точек S на плоскости определяется как множество всех точек плоскости, расстояние от которых до точки P меньше, чем до любой другой точки множества S. Объединение всех многоугольников Вороного для точек множества S образует ***диаграмму Вороного*** для множества S.
   а) Как выглядит диаграмма Вороного для множества из трех точек?
   б) Найдите в Web визуализацию алгоритма для генерации диаграммы Вороного и рассмотрите несколько примеров диаграмм. Можете ли вы указать, каким образом обобщить решение предыдущего вопроса?

6. Поясните, как найти точку $P_{\max}$ в алгоритме быстрой оболочки аналитически.
7. Опишите с использованием геометрических представлений входные данные, для которых множество S_1 будет пустым. Что будет верхней оболочкой в этой ситуации?
8. Какова эффективность алгоритма быстрой оболочки в наилучшем случае?
9. Приведите пример входных данных, для которых алгоритм быстрой оболочки выполняется за квадратичное время.
10. Реализуйте алгоритм быстрой оболочки на своем любимом языке программирования.

Резюме

- *Метод декомпозиции* ("разделяй и властвуй") представляет собой общий метод разработки алгоритмов, когда задача решается путем разделения ее на несколько меньших подзадач (в идеале — одинакового размера), рекурсивного решения каждой из них с последующим объединением полученных решений подзадач в решение исходной задачи. На основе этого метода построены многие эффективные алгоритмы, хотя к некоторым задачам он может быть неприменим, а в других случаях давать худшие результаты, чем иное, более простое алгоритмическое решение.
- Временна́я эффективность $T(n)$ многих алгоритмов декомпозиции удовлетворяет рекуррентному соотношению $T(n) = aT(n/b) + f(n)$. *Основная теорема* позволяет определить порядок роста решения такого рекуррентного соотношения.
- *Сортировка слиянием* представляет собой алгоритм сортировки, основанный на методе декомпозиции. Она выполняется путем разделения входного массива на две половины, рекурсивной сортировки этих половин с последующим их *слиянием* для получения отсортированного исходного массива. Временна́я эффективность алгоритма — $\Theta(n \log n)$ для любых входных данных; при этом количество сравнений ключей очень близко к теоретическому минимуму. Основным недостатком данного алгоритма является необходимость большого количества дополнительной памяти.
- *Быстрая сортировка* представляет собой алгоритм сортировки, основанный на методе декомпозиции. Она работает путем разбиения входных элементов в соответствии с их значениями относительно некото-

рого предварительно выбранного элемента. Среди прочих алгоритмов класса $n \log n$ быстрая сортировка известна своей высокой эффективностью при сортировке случайных массивов, но в наихудшем случае ее эффективность — квадратична.

- *Бинарный поиск* представляет собой алгоритм поиска в отсортированном массиве со временем работы $O(\log n)$. Это нетипичный пример применения метода декомпозиции, поскольку требует решения только одной подзадачи половинного размера на каждой итерации.
- Классические алгоритмы обхода бинарного дерева — в *прямом*, *центрированном* и *обратном* порядке — и аналогичные алгоритмы, требующие рекурсивной обработки левого и правого поддеревьев, могут рассматриваться как примеры метода декомпозиции. Их анализ упрощается при замене всех пустых поддеревьев рассматриваемого дерева специальными *внешними узлами*.
- Имеется алгоритм декомпозиции, который позволяет выполнить умножение двух n-значных чисел с использованием $n^{1.585}$ умножений цифр.
- *Алгоритму Штрассена* для умножения двух матриц размером 2×2 требуется всего 7 умножений, хотя при этом количество сложений у него выше, чем у алгоритма, основанного на определении умножения матриц. Метод декомпозиции позволяет перемножать матрицы размером $n \times n$ с помощью около $n^{2.807}$ умножений.
- Метод декомпозиции с успехом применяется для решения таких важных задач вычислительной геометрии, как задача о паре ближайших точек и задача о выпуклой оболочке.

Глава 5

Метод уменьшения размера задачи

> Плутарх говорит, что Серториус, желая показать своим солдатам, что настойчивость и остроумность лучше грубой силы, поставил перед ними двух лошадей и приказал двум людям вырвать у них хвосты. Один из них — сущий Геркулес — безуспешно тянул и рвал хвост целиком, в то время как другой — хитроумный худышка-портной, выдергивал хвост по одному волоску, и довольно быстро и без усилий оставил лошадь бесхвостой.
>
> — Кобхэм Брювер (E. Cobham Brewer),
> Словарь идиом и аллегорий (Dictionary of Phrase and Fable), 1898.

Метод уменьшения размера задачи ("уменьшай и властвуй") основан на использовании связи между решением данного экземпляра задачи и решением меньшего экземпляра той же задачи. Если такая связь установлена, ее можно использовать либо сверху вниз (рекурсивно), либо снизу вверх (без использования рекурсии). Имеется три основных варианта метода уменьшения размера:

- уменьшение на постоянную величину;
- уменьшение на постоянный множитель;
- уменьшение переменного размера.

При ***уменьшении на постоянную величину*** размер экземпляра задачи снижается на одну и ту же постоянную величину при каждой итерации алгоритма. Обычно эта величина равна единице (рис. 5.1), хотя встречается и снижение размера на два — в алгоритмах, которые по-разному работают для экземпляров задач четного и нечетного размера.

Рассмотрим в качестве примера задачу вычисления a^n для положительных целых показателей степени. Связь между решением экземпляра размером n и экземпляром размером $n-1$ выражается очевидной формулой $a^n = a^{n-1} \cdot a$. Таким образом, функция $f(n) = a^n$ может быть вычислена либо "сверху вниз" с исполь-

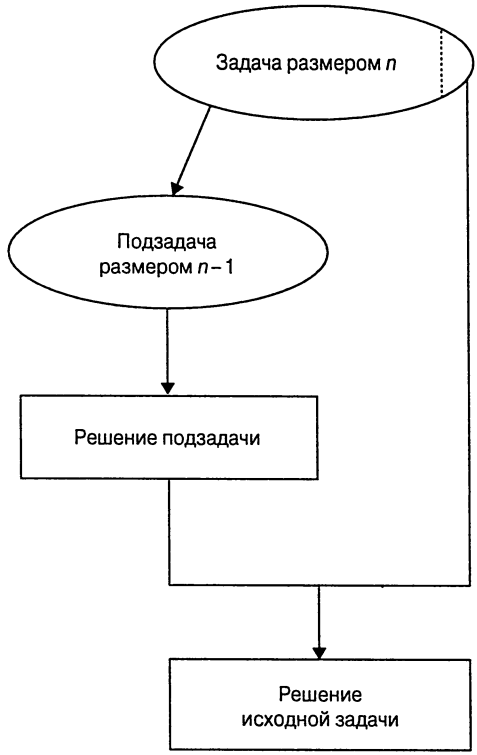


Рис. 5.1. Метод уменьшения размера (на единицу)

зованием рекурсивного определения

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{при } n > 1 \\ a & \text{при } n = 1 \end{cases} \tag{5.1}$$

либо "снизу вверх" путем умножения a на себя $n-1$ раз (да, это тот же алгоритм, что и при использовании грубой силы, но мы пришли к нему в результате другого подхода). С более интересными примерами алгоритмов уменьшения размера на единицу вы встретитесь в разделах 5.1–5.4.

Уменьшение на постоянный множитель предполагает уменьшение размера экземпляра задачи при каждой итерации алгоритма на один и тот же множитель. В большинстве приложений этот множитель равен двум. (Можете ли вы привести пример подобного алгоритма?) Идея метода уменьшения на постоянный множитель показана на рис. 5.2.

В качестве примера вновь обратимся к задаче возведения в степень. Для решения экземпляра задачи размером n — вычисления величины a^n — решается задача половинного размера, т.е. вычисляется значение $a^{n/2}$, которое связано с решением

Рис. 5.2. Метод уменьшения размера (вдвое)

исходной задачи очевидным соотношением $a^n = \left(a^{n/2}\right)^2$. Однако поскольку мы рассматриваем задачу только для целых показателей степени, описанный метод годится только для четных n. Если же n нечетно, то мы должны вычислить a^{n-1} с использованием правила для четных показателей степени, а затем умножить результат на a. Итак, необходимо следовать такой формуле:

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{при } n \text{ положительном четном,} \\ \left(a^{(n-1)/2}\right)^2 \cdot a & \text{при нечетном } n, \text{ большем 1,} \\ a & \text{при } n = 1. \end{cases} \qquad (5.2)$$

При рекурсивном вычислении a^n в соответствии с формулой (5.2) и измерении эффективности алгоритма посредством количества выполняемых умножений, следует ожидать, что он принадлежит классу $O(\log n)$, потому что при каждой итерации размер задачи снижается как минимум вдвое, ценой не более двух умножений.

Обратите внимание на отличие этого алгоритма от алгоритма, основанного на методе декомпозиции и решающего две задачи возведения в степень с разме-

ром $n/2$:

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil} & \text{при } n > 1, \\ a & \text{при } n = 1. \end{cases} \tag{5.3}$$

Алгоритм, основанный на формуле (5.3), неэффективен (почему?), так что алгоритм на основе формулы (5.2) работает гораздо быстрее.

Несколько других примеров алгоритмов с использованием метода уменьшения на постоянный множитель приведены в разделе 5.5 и упражнениях к нему. Такие алгоритмы весьма эффективны, однако мало распространены.

Наконец, при ***уменьшении переменного размера*** величина снижения размера задачи изменяется от итерации к итерации. Примером такого алгоритма может служить алгоритм Евклида для вычисления наибольшего общего делителя. Вспомним, что этот алгоритм основан на формуле

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Хотя аргументы в правой части уравнения всегда меньше аргументов в левой части (как минимум, начиная со второй итерации алгоритма), они не меньше ни на постоянное значение, ни на постоянный множитель. Несколько других примеров алгоритмов такого вида описано в разделе 5.6.

5.1 Сортировка вставкой

В этом разделе мы рассмотрим применение метода уменьшения на единицу к сортировке массива $A[0..n-1]$. Следуя идее метода, полагаем, что задача меньшего размера, а именно сортировка массива $A[0..n-2]$, уже решена и мы имеем отсортированный массив размером $n-1$: $A[0] \leqslant \cdots \leqslant A[n-2]$. Каким образом воспользоваться этим решением задачи меньшего размера для получения решения исходной задачи, учитывающей наличие элемента $A[n-1]$? Очевидно, все, что необходимо, — это найти нужную позицию среди отсортированных элементов и вставить туда элемент $A[n-1]$.

Имеется три подходящих способа сделать это. Во-первых, мы можем сканировать отсортированный подмассив слева направо, пока не достигнем первого элемента, большего или равного $A[n-1]$, и после этого осуществить вставку непосредственно перед найденным элементом. Во-вторых, мы можем сканировать подмассив справа налево, пока не найдем первый элемент, меньший или равный $A[n-1]$, и затем вставить $A[n-1]$ непосредственно за ним. По сути, оба варианта эквивалентны; на практике обычно используется второй способ, поскольку он лучше работает с отсортированными или почти отсортированными массивами (почему?). Получившийся в результате алгоритм сортировки называется ***простой сортировкой вставкой*** (straight insertion sort), или просто ***сортировкой вставкой***. Третий способ заключается в применении бинарного поиска для

определения позиции вставки элемента $A[n-1]$ в отсортированную часть массива. Получающийся при этом алгоритм носит название ***бинарной сортировки вставкой*** (binary insertion sort). В упражнениях к данному разделу вам предлагается самостоятельно реализовать эту идею и исследовать эффективность бинарной сортировки вставкой.

Хотя сортировка вставкой основана на рекурсивном подходе, более эффективной будет ее реализация снизу вверх, т.е. итеративная. Как показано на рис. 5.3, элемент $A[i]$ (начиная с элемента $A[1]$ и заканчивая $A[n-1]$) вставляется в соответствующее место среди первых i элементов массива (которые к этому моменту уже отсортированы). Однако в отличие от сортировки выбором элемент в общем случае вставляется не в окончательную позицию, которую он будет занимать в полностью отсортированном массиве.

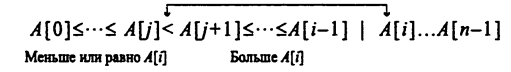


Рис. 5.3. Итерация сортировки вставкой: $A[i]$ вставляется в корректную позицию среди уже отсортированных элементов

Ниже приведен псевдокод данного алгоритма.

АЛГОРИТМ $InsertionSort(A[0..n-1])$
// Сортировка массива методом сортировки вставкой
// **Входные данные:** Массив $A[0..n-1]$ из n упорядочиваемых
// элементов
// **Выходные данные:** Массив $A[0..n-1]$, отсортированный
// в неубывающем порядке
for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i-1$
  while $j \geqslant 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

Работа алгоритма проиллюстрирована на рис. 5.4.

Базовой операцией алгоритма является сравнение ключей $A[j] > v$. (Почему не $j \geqslant 0$? Потому что это сравнение почти всегда быстрее предыдущего в реальных реализациях. Кроме того, это сравнение не является неотъемлемой частью алгоритма: реализация с ограничителем (см. упражнение 5.1.5) позволяет полностью его устранить.)

89 | **45** 68 90 29 34 17
45 89 | **68** 90 29 34 17
45 68 89 | **90** 29 34 17
45 68 89 90 | **29** 34 17
29 45 68 89 90 | **34** 17
29 34 45 68 89 90 | **17**
17 29 34 45 68 89 90

Рис. 5.4. Пример сортировки вставкой. Вертикальная черта отделяет отсортированную часть массива от остальных элементов; вставляемый элемент выделен полужирным шрифтом

Количество сравнений ключей в этом алгоритме, очевидно, зависит от природы входных данных. В худшем случае сравнение $A[j] > v$ выполняется наибольшее количество раз, т.е. для всех $j = i-1, \ldots, 0$. Поскольку $v = A[i]$, это происходит тогда и только тогда, когда $A[j] > A[i]$ для $j = i-1, \ldots, 0$. (Заметим, что мы используем тот факт, что на i-ой итерации сортировки вставкой все элементы, предшествующие $A[i]$, — первые i элементов входных данных, хотя и расположенные в отсортированном порядке.) Таким образом, для входных данных для наихудшего случая мы получаем $A[0] > A[1]$ (для $i = 1$), $A[1] > A[2]$ (для $i = 2$), $\ldots$, $A[n-2] > A[n-1]$ (для $i = n-1$). Другими словами, входные данные в наихудшем случае представляют собой массив строго уменьшающихся значений. Количество сравнений ключей для таких входных данных равно

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} \in \Theta\left(n^2\right).$$

Итак, в наихудшем случае сортировка вставкой выполняет столько же сравнений, сколько и сортировка выбором (см. раздел 3.1).

В лучшем случае сравнение $A[j] > v$ выполняется только один раз на каждой итерации внешнего цикла. Это происходит тогда и только тогда, когда $A[i-1] \leqslant A[i]$ для всех $i = 1, \ldots, n-1$, т.е. если входной массив уже отсортирован в возрастающем порядке. (Хотя "разумно", что наилучший случай алгоритма соответствует уже решенной задаче, это далеко не всегда так; вспомните, например, обсуждение быстрой сортировки в главе 4.) Итак, для отсортированного входного массива количество сравнений ключей равно

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n).$$

Эта очень хорошая производительность, которая достигается в наилучшем случае для отсортированного входного массива, не слишком полезна сама по себе, поскольку нет никаких оснований ожидать именно таких приятных входных данных. Однако в ряде приложений приходится сталкиваться с почти отсортированными файлами, и в этом случае сортировка вставкой имеет возможность проявить себя во всей красе. Например, при быстрой сортировке массива можно прекратить итерации алгоритма, когда подмассивы достигнут некоторых предопределенных размеров (например, будут содержать меньше 10 элементов). В этот момент весь исходный массив почти отсортирован, и мы можем завершить нашу работу, применив сортировку вставкой. Обычно такое видоизменение алгоритма позволяет уменьшить общее время сортировки примерно на 10%.

Строгий анализ эффективности алгоритма в среднем случае основан на исследовании количества пар с нарушенным условием сортировки (см. упражнение 5.1.8). Он показывает, что при работе со случайными массивами сортировка вставкой выполняет примерно в два раза меньше сравнений, чем в случае убывающего массива, т.е.

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta\left(n^2\right).$$

Такая удвоенная по сравнению с наихудшим случаем производительность алгоритма в среднем случае в совокупности с превосходной производительностью для почти отсортированных массивов выделяет сортировку вставкой из числа других элементарных алгоритмов сортировки — сортировки выбором и пузырьковой сортировки. Кроме того, расширение сортировки вставкой — ***сортировка Шелла*** (shellsort), открытая Д. Шеллом (D.L. Shell) [107], дает еще лучший алгоритм для сортировки достаточно больших файлов (см. упражнение 5.1.10).

Упражнения 5.1

1. Отряд из n солдат должен переправиться через широкую и глубокую реку, через которую нет моста. Солдаты заметили лодку с двумя мальчишками, но лодка так мала, что в ней могут поместиться либо два мальчишки, либо один солдат. Как солдатам переправиться через реку и после переправы возвратить лодку мальчишкам? Сколько раз при этом лодка проплывет от берега к берегу?

2. **а)** Разработайте рекурсивный алгоритм на основе уменьшения размера на единицу для поиска позиции наименьшего элемента в массиве из n элементов.

   б) Определите временну́ю эффективность этого алгоритма и сравните ее с эффективностью алгоритма грубой силы для решения той же задачи.

3. Разработайте алгоритм на основе уменьшения размера на единицу для генерации показательного множества для множества из n элементов. (Показательное множество (power set) для множества S — это множество всех подмножеств S, включая пустое множество и само множество S.)
4. С помощью сортировки вставкой отсортируйте список E, X, A, M, P, L, E в алфавитном порядке.
5. **а)** Какой ограничитель должен быть помещен перед первым элементом сортируемого массива для того, чтобы избежать проверки выхода за границы массива $j \geqslant 0$ на каждой итерации внутреннего цикла сортировки вставкой?

   б) Будет ли версия алгоритма с ограничителем принадлежать тому же классу эффективности, что и исходная версия?
6. Возможно ли реализовать сортировку вставкой для сортировки связанных списков? Будет ли она иметь ту же эффективность $O\left(n^2\right)$, что и версия для массивов?
7. Сравните реализацию сортировки вставкой в тексте раздела и следующую версию алгоритма.

   Алгоритм $InsertSort2\,(A\,[0..n-1])$

   for $i \leftarrow 1$ **to** $n-1$ **do**

   $j \leftarrow i-1$

   while $j \geqslant 0$ **and** $A[j] > A[j+1]$ **do**

   $swap(A[j], A[j+1])$ // Обмен $A[j]$ и $A[j+1]$

   $j \leftarrow j-1$

   Какова временна́я эффективность данного алгоритма? Сравните ее с эффективностью алгоритма, приведенного в тексте раздела.
8. Пусть $A\,[0..n-1]$ — массив из n сортируемых элементов. (Для простоты можете считать, что все элементы различны.) Пара индексов (i, j) называется ***инверсией*** (inversion), если $i < j$ и $A\,[i] > A\,[j]$.

   а) Какой массив размером n имеет наибольшее количество инверсий, и чему оно равно? Ответьте на тот же вопрос для наименьшего числа инверсий.

   б) Покажите, что количество сравнений ключей в среднем случае сортировки вставкой дается формулой

$$C_{avg}\,(n) \approx \frac{n^2}{4}.$$

9. Бинарная сортировка вставкой использует для поиска позиции вставки элемента $A[i]$ среди предварительно отсортированных элементов $A[0] \leqslant \cdots \leqslant A[i-1]$ бинарный поиск. Определите класс эффективности этого алгоритма в наихудшем случае.

10. Сортировка Шелла представляет собой важный алгоритм сортировки, который работает путем применения сортировки вставкой к каждому из нескольких чередующихся подфайлов данного файла. При каждом проходе по файлу подфайлы образуются путем смещения на величину h_i, взятую из предопределенной убывающей последовательности $h_1 > \cdots > h_i > \cdots > 1$; при последнем проходе смещение должно быть равно 1. (Алгоритм работает при использовании любой такой последовательности, хотя некоторые из них с точки зрения эффективности лучше других. Например, последовательность 1, 4, 13, 40, 121, ... — конечно, используемая в обратном порядке, известна как одна из лучших для данного алгоритма.)[1]

    а) Примените сортировку вставкой к списку S, H, E, L, L, S, O, R, T, I, S, U, S, E, F, U, L.

    б) Устойчива ли сортировка Шелла?

    в) Реализуйте сортировку Шелла, простую сортировку вставкой, бинарную сортировку вставкой, сортировку слиянием и быструю сортировку на вашем любимом языке программирования и сравните их производительность для случайных массивов размером 10^2, 10^3, 10^4 и 10^5 элементов, а также для возрастающих и убывающих файлов этих размеров.

5.2 Поиск в глубину и поиск в ширину

В двух следующих разделах главы мы поговорим об очень важных алгоритмах для работы с графами, которые можно рассматривать как применение метода уменьшения размера. Предполагается, что читатель знаком с концепцией графа и его основными разновидностями (неориентированный, ориентированный, взвешенный графы) и двумя основными представлениями (матрица смежности

[1] Чтобы было понятнее, опишем возможный процесс сортировки массива из 16 элементов с использованием последовательности 8,4,2,1. При первом проходе они делятся на 8 групп по 2 элемента, индексы которых отличаются на 8: $(a_1, a_9), \ldots, (a_8, a_{16})$, и выполняется сортировка в пределах каждой группы. При втором проходе получаем 4 группы по 4 элемента — $(a_1, a_5, a_9, a_{13}), \ldots, (a_4, a_8, a_{12}, a_{16})$, и сортируем каждую группу. на третьем проходе получаются и сортируются две группы по 8 записей $(a_1, a_3, \ldots, a_{15}), (a_2, a_4, \ldots, a_{16})$, а на последнем проходе весь массив рассматривается как одна группа. — *Прим. ред.*

и связанные списки), а также такими понятиями, как связность и ацикличность графа. Краткий обзор этого материала можно найти в разделе 1.4.

Как указывалось в разделе 1.3, графы представляют собой интересные структуры с широкой областью применения. Многие алгоритмы, работающие с графами, требуют определенной систематической обработки вершин или ребер графа. Для таких обходов графа имеется два основных алгоритма — поиск в глубину (depth-first search, DFS) и поиск в ширину (breadth-first search, BFS). Кроме своей основной работы (посещения всех вершин и обхода ребер графа), эти алгоритмы весьма полезны при исследовании ряда важных свойств графов.

Поиск в глубину

Поиск в глубину начинает посещение вершин графа с произвольной вершины, помечая ее как посещенную. На каждой итерации алгоритм обрабатывает непосещенные вершины, смежные с текущей. (Если таких вершин несколько, они обрабатываются в произвольном порядке. С практической точки зрения выбор кандидатов для посещения определяется структурой данных, представляющей граф. В примерах мы всегда будем использовать алфавитный порядок.) Этот процесс продолжается до достижения тупика — вершины, у которой нет непосещенных смежных вершин. По достижении тупика алгоритм возвращается на одно ребро назад (в вершину, из которой он попал в тупик) и пытается продолжить посещения смежных непосещенных вершин из этого места. В конце концов алгоритм прекращает работу, когда возвращается в начальную вершину, которая к этому моменту становится тупиком. В этот момент все вершины данного связного компонента графа оказываются посещенными. Если в графе после этого имеются непосещенные вершины, процесс должен повториться, при этом в качестве стартовой используется любая из непосещенных вершин.

Для выполнения поиска в глубину удобно использовать стек. Мы помещаем вершину в стек, когда впервые посещаем ее (т.е. когда начинается обход вершины), и снимаем ее со стека, когда она становится тупиком (т.е. обход вершины завершается).

Очень полезно одновременно с обходом графа строить так называемый ***лес поиска в глубину*** (depth-first search forest). Стартовая вершина поиска служит корнем первого дерева в таком лесу. Когда впервые достигается новая непосещенная ранее вершина, она добавляется в качестве дочерней к вершине, из которой была достигнута. Соответствующее ребро называется ***ребром дерева*** (tree edge), поскольку множество таких ребер образует лес. Алгоритм может встретиться с ребром, которое ведет в уже посещенную вершину, не являющуюся непосредственным предшественником данной (т.е. не являющейся родительским узлом в дереве). Такое ребро именуют ***обратным ребром*** (back edge), поскольку оно соединяет вершину с ее предком, не являющимся родительским узлом в дереве

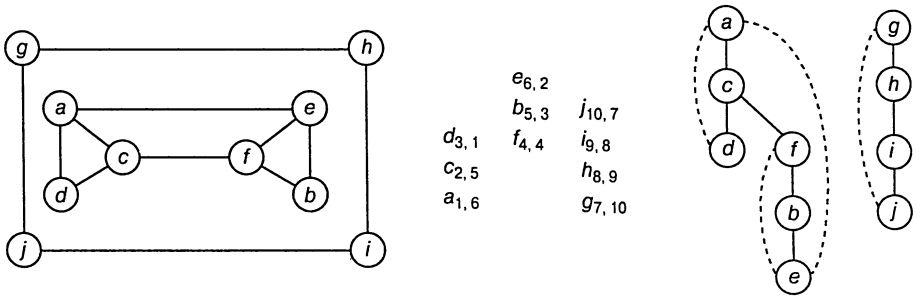


Рис. 5.5. Пример поиска в глубину. *а*) Граф. *б*) Стек обхода (первый индекс указывает порядок, в котором вершины посещались (вносились в стек), а второй — в каком они становились тупиками (снимались со стека)). *в*) Лес поиска в глубину (ребра деревьев показаны сплошными линиями, а обратные ребра — пунктиром)

поиска в глубину. На рис. 5.5 приведен пример поиска в глубину, со стеком обхода и соответствующим лесом поиска в глубину.

Вот как выглядит псевдокод поиска в глубину.

Алгоритм $DFS\,(G)$

// Реализует поиск в глубину для данного графа
// **Входные данные:** Граф $G = \langle V, E \rangle$
// **Выходные данные:** Граф G, вершины которого пронумерованы
// последовательными числами в порядке их
// первого посещения в процессе поиска
Помечаем все вершины в V числом 0 (непосещенная вершина)
$count \leftarrow 0$
for (для) каждой вершины v из V **do**
  if вершина v помечена 0
    $dfs(v)$

$dfs\,(v)$
// Рекурсивно посещает все непосещенные вершины, связанные
// с вершиной v, и присваивает им номера в порядке посещения
// при помощи глобальной переменной $count$
$count \leftarrow count + 1$; Помечаем v числом $count$
for (для) каждой вершины w из V, смежной с v **do**
  if w помечена 0
    $dfs(w)$

Краткость псевдокода процедуры DFS и простота, с какой ее можно выполнить вручную, создает ложное впечатление об уровне сложности данного алгоритма. Чтобы оценить истинную мощность и глубину алгоритма, вы должны пошагово пройти его, пользуясь не графическим представлением графа, а матрицей смежности или связанными списками смежных вершин (попробуйте сделать это для графа, изображенного на рис. 5.5, или даже для графа меньшего размера).

Насколько эффективен алгоритм поиска в глубину? Нетрудно увидеть, что этот алгоритм достаточно эффективен, поскольку время его работы пропорционально размеру используемой для представления графа структуры данных. Так, для представления с использованием матрицы смежности временна́я эффективность обхода равна $\Theta(|V|^2)$, а для представления с использованием связанных списков — $\Theta(|V| + |E|)$, где $|V|$ и $|E|$ — соответственно, количество вершин и ребер графа.

Лес поиска в глубину, который получается как побочный результат обхода, также заслуживает внимания. Начнем с того, что в действительности это не лес. Как мы уже видели в рассмотренном примере, ребра графа делятся при поиске в глубину на два непересекающихся класса — ребра дерева и обратные ребра (в лесу поиска в глубину для неориентированного графа не могут быть другие типы ребер). Ребра дерева — это те ребра, которые использовались поиском в глубину для достижения ранее непосещенных узлов. Рассматривая только эти ребра, мы в действительности получим лес. Обратные ребра соединяют вершины с ранее посещенными вершинами, которые не являются непосредственными предшественниками рассматриваемых при обходе. Обратные ребра соединяют вершины с предками, не являющимися непосредственными родителями.

Поиск в глубину и соответствующее представление графа в виде леса оказываются очень полезными инструментами при разработке эффективных алгоритмов для проверки многих важных свойств графов.[2] Заметим, что поиск в глубину дает два упорядочения вершин: порядок, в котором вершины впервые посещаются при обходе (вносятся в стек), и порядок, в каком они становятся тупиками (снимаются со стека). Это качественно различные порядки, и они применяются в разных типах приложений.

Важным применением поиска в глубину являются проверка связности графа и его ацикличности. Поскольку поиск в глубину завершается после посещения всех вершин, соединенных некоторым путем со стартовой, проверку связности графа можно выполнить следующим образом. Начнем поиск в глубину с произвольной вершины и после завершения проверим, все ли вершины графа посещены. Если все, то граф связный, в противном случае — нет. Вообще говоря,

[2]Честь открытия ряда таких применений в 1970-х годах принадлежит Джону Хопкрофту (John Hopcroft) и Роберту Таржану (Robert Tarjan) [53, 117]; за эти (и другие) работы они удостоились премии Тьюринга — наивысшей награды в области теоретической кибернетики.

поиск в глубину можно использовать для определения связных компонентов графа (как?).

Для проверки наличия циклов в графе можно воспользоваться представлением графа в виде леса поиска в глубину. Если в нем не окажется обратных ребер, очевидно, что граф ацикличен. Если же в лесу имеется обратное ребро от некоторой вершины u к вершине v (например, обратное ребро от d к a на рис. 5.5*в*), то граф имеет цикл, который включает в себя путь от v к u в виде последовательности ребер дерева, и обратное ребро от u к v.

Далее в книге вы найдете некоторые другие применения поиска в глубину, хотя ряд более сложных приложений, например, поиск точек сочленения графа, в ней отсутствуют. (Вершина связного графа называется ***точкой сочленения*** (articulation point), если при ее удалении вместе со всеми инцидентными ей ребрами граф распадается на непересекающиеся части.)

Поиск в ширину

Если поиск в глубину можно назвать обходом для смелых (алгоритм идет как можно дальше от "дома"), то поиск в ширину — это обход для осторожных. Он работает "кругами", сперва посещая все вершины, смежные со стартовой, затем — вершины, которые лежат на расстоянии двух ребер от стартовой, и т.д., пока не будут посещены все вершины связного компонента, которому принадлежит стартовая точка.

Для выполнения поиска в ширину удобно использовать очередь (обратите внимание на это отличие от поиска в глубину!). Очередь инициализируется стартовой вершиной поиска, помеченной как посещенная. На каждой итерации алгоритм находит все непосещенные вершины, смежные с вершиной в начале очереди, помечает их как посещенные и вносит в очередь; после этого вершина в начале очереди изымается из нее.

Как и в случае поиска в глубину, поиск в ширину полезно сопровождать построением так называемого леса поиска в ширину (breadth-first search forest). Стартовая вершина поиска в глубину служит корнем первого дерева в таком лесу. Когда при поиске впервые достигается новая непосещенная вершина, она присоединяется в качестве дочерней к вершине, из которой достигнута, и соединяющее их ребро называется ***ребром дерева*** (tree edge). Если ребро ведет к уже посещенной вершине, отличной от непосредственного предшественника (т.е. не являющейся родительской вершиной в дереве), такое ребро помечается как ***поперечное*** (cross edge). На рис. 5.6 приведен пример поиска в ширину, с очередью поиска и соответствующим лесом поиска в ширину.

Далее представлен псевдокод поиска в ширину.

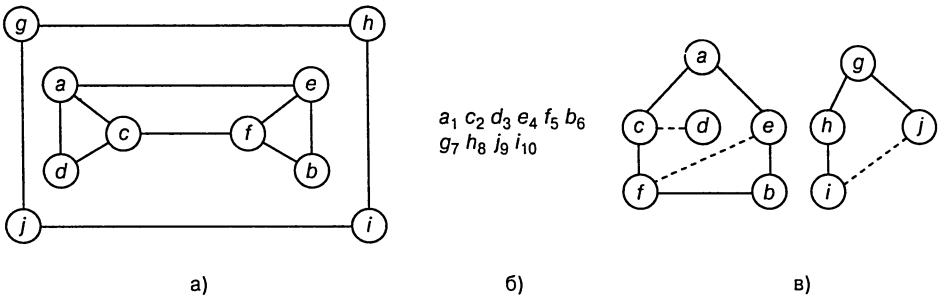


Рис. 5.6. Пример поиска в ширину. *а*) Граф. *б*) Очередь поиска, в которой индексы указывают очередность посещения вершин (т.е. последовательность их внесения в очередь (или извлечения из нее)). *в*) Лес поиска в ширину (ребра дерева показаны сплошными линиями, поперечные ребра — пунктиром)

Алгоритм $BFS\,(G)$

// Реализует поиск в ширину в данном графе
// **Входные данные:** Граф $G = \langle V, E \rangle$
// **Выходные данные:** Граф G, вершины которого маркированы
// последовательными целыми числами в том
// порядке, в каком они посещались при
// поиске в ширину
Помечаем каждую вершину в V числом 0, как непосещенную
$count \leftarrow 0$
for (для) каждой вершины v из V **do**
  if v помечена значением 0
    $bfs(v)$

$bfs\,(v)$
// Обход всех вершин, связанных с v и назначение им
// номеров в порядке посещения при помощи глобальной
// переменной $count$
$count \leftarrow count + 1$
Присваиваем v значение $count$ и инициализируем очередь вершиной v
while (пока) очередь не пуста **do**
  for (для) каждой вершины w из V, смежной с вершиной,
    находящейся в начале очереди **do**
    if w помечена значением 0
      $count \leftarrow count + 1$
      Помечаем w значением $count$
      Добавляем w в очередь
  Убираем из очереди первый элемент

Поиск в ширину имеет ту же эффективность, что и поиск в глубину: $\Theta(|V|^2)$ для представления графа с использованием матрицы смежности и $\Theta(|V| + |E|)$ для представления с использованием связанных списков. В отличие от поиска в глубину поиск в ширину дает единственное упорядочение вершин, поскольку очередь подчиняется принципу FIFO (first-in-first-out, первым вошел — первым вышел), следовательно, порядок добавления вершин в очередь совпадает с порядком их извлечения из очереди. Что касается структуры дерева поиска в ширину, то в ней так же, как и в дереве поиска в глубину, имеется два вида ребер — ребра дерева и поперечные ребра. Ребра дерева — это те ребра, которые используются для достижения ранее непосещенных вершин; поперечные ребра соединяют уже посещенные ранее вершины, но, в отличие от дерева поиска в глубину, соединяют либо сестринские узлы, либо "двоюродные" на том же уровне дерева поиска в ширину или на соседних уровнях.

Наконец, поиск в ширину может использоваться для проверки связности и ацикличности графа точно так же, как и поиск в глубину. Однако для некоторых не столь простых алгоритмов, например, таких, как поиск точек сочленения, поиск в ширину неприменим. С другой стороны, имеются ситуации, где можно использовать поиск в ширину, но не в глубину — например, при поиске пути между двумя вершинами, состоящего из минимального количества ребер. В этом случае мы начинаем с поиска в ширину в одной из двух заданных вершин и завершаем его, когда нужная вершина достигнута. Простой путь от корня дерева поиска в ширину ко второй вершине и есть искомый кратчайший путь. Например, в графе, представленном на рис. 5.7, путь $a - b - c - g$ имеет минимальное количество ребер из всех путей между вершинами a и g. Хотя корректность такого применения поиска в ширину, как представляется, следует непосредственно из принципа работы поиска в ширину, математическое доказательство этого факта не назовешь элементарным (см., например, [32]).

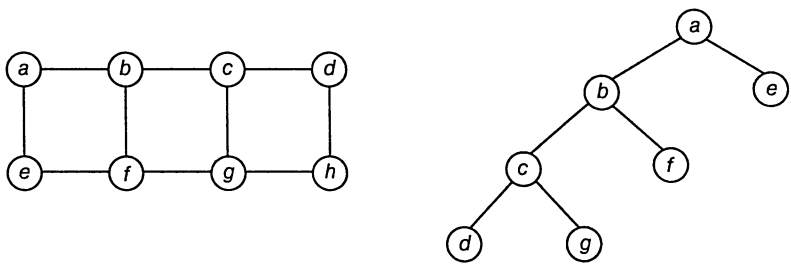


Рис. 5.7. Иллюстрация применения поиска в ширину для определения кратчайшего пути между двумя вершинами. *а*) Граф. *б*) Часть дерева поиска в ширину, позволяющая определить кратчайший путь от вершины a к вершине g

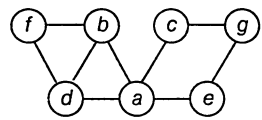
В табл. 5.1 приведены основные сведения о поиске в глубину и в ширину.

Таблица 5.1. Основные сведения о поиске в глубину (DFS) и в ширину (BFS)

	DFS	BFS
Структура данных	Стек	Очередь
Количество упорядочений вершин	2	1
Типы ребер (в неориентированном графе)	Ребра дерева и обратные ребра	Ребра дерева и поперечные ребра
Приложения	Связность, ацикличность, поиск точек сочленения	Связность, ацикличность, поиск кратчайшего пути
Эффективность при использовании матриц смежности	$\Theta\left(\lvert V\rvert^2\right)$	$\Theta\left(\lvert V\rvert^2\right)$
Эффективность при использовании связанных списков смежности	$\Theta\left(\lvert V\rvert + \lvert E\rvert\right)$	$\Theta\left(\lvert V\rvert + \lvert E\rvert\right)$

Упражнения 5.2

1. Рассмотрим граф

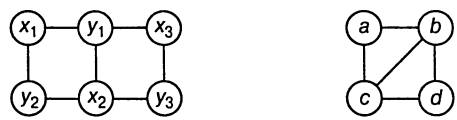


   а) Запишите матрицу смежности и связанные списки, представляющие данный граф (считаем, что строки и столбцы матрицы, а также вершины в связанных списках следуют алфавитному порядку меток вершин).

   б) Начиная с вершины a в качестве стартовой и работая в алфавитном порядке, обойдите граф при помощи поиска в глубину и постройте соответствующее дерево. Укажите порядок, в котором вершины впервые посещались при обходе (и вносились в стек обхода), и порядок, в котором они становились тупиками (и снимались со стека).

2. Если определить разреженный граф как такой, у которого $|E| \in O(|V|)$, то какая из реализаций поиска в глубину для него будет более эффективна — с использованием матрицы смежности или связанных списков?

3. Пусть G — граф с n вершинами и m ребрами. Верны или нет следующие утверждения.
   а) Все леса поиска в глубину (получающиеся при обходах, которые начинаются в разных вершинах) будут состоять из одинакового количества деревьев?
   б) Все леса поиска в глубину будут иметь одно и то же количество ребер деревьев и одинаковое количество обратных ребер.
4. Начиная с вершины a в качестве стартовой и работая в алфавитном порядке, выполните поиск в ширину для графа из упражнения 1 и постройте соответствующее дерево.
5. Докажите, что поперечные ребра в дереве поиска в ширину могут соединять вершины только на одном уровне или на соседних уровнях дерева поиска в ширину.
6. **а)** Объясните, как убедиться в ацикличности графа при помощи поиска в ширину.
   б) Верно ли утверждение, что один из поисков — в глубину или в ширину — всегда находит циклы быстрее другого? Если да, то какой из них более эффективен в этом смысле и почему. Если нет, приведите два примера, подтверждающие это.
7. Поясните, как можно идентифицировать связные компоненты графа
   а) при использовании поиска в глубину;
   б) при использовании поиска в ширину.
8. Граф называется ***двудольным*** (bipartite), если все его вершины можно разделить на два непересекающихся подмножества X и Y такие, что любое ребро соединяет вершину из X с вершиной из Y. (Мы можем также сказать, что граф двудольный, если его вершины могут быть окрашены в два цвета так, что каждое ребро соединяет вершины разных цветов; такие графы называются ***2-раскрашиваемыми*** (2-colorable). Так, на представленном ниже рисунке граф *а* — двудольный, а граф *б* — нет.



а) б)

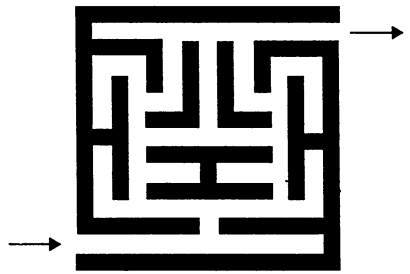
   а) Разработайте алгоритм на основе поиска в глубину, который бы определял, является ли граф двудольным.

б) Разработайте алгоритм на основе поиска в ширину, который бы определял, является ли граф двудольным.

9. Напишите программу, которая для данного графа выводила бы 1) вершины всех связных компонентов; 2) его цикл или сообщение, что граф ацикличен.

10. Лабиринт можно смоделировать при помощи вершин, соответствующих входу в лабиринт, выходу, тупикам и всем точкам лабиринта, в которых есть возможность выбора пути, и затем соединить эти вершины ребрами, соответствующими путям в лабиринте.

а) Постройте такой граф для показанного лабиринта.



б) Какой обход — в глубину или в ширину — вы предпочтете, попав в лабиринт, и почему?

5.3 Топологическая сортировка

В этом разделе мы рассмотрим важную задачу, связанную с ориентированными графами. Перед тем как приступить к ней, познакомимся с ориентированными графами поближе. ***Ориентированный граф*** (directed graph, digraph) — это граф, для каждого ребра которого указано направление (см., например, рис. 5.8*а*). Имеется два основных вида представления ориентированных графов: при помощи матриц смежности и при помощи связанных списков. Имеются два основных различия в представлении неориентированных и ориентированных графов: 1) матрица смежности ориентированного графа не обязана быть симметричной и 2) ребро ориентированного графа имеет только один (а не два) соответствующий узел в связанных списках смежности.

Поиск в глубину и поиск в ширину являются основными алгоритмами обхода ориентированных графов, но структура образующихся при этом лесов более сложна, чем в случае неориентированных графов. Так, даже для простого графа, показанного на рис. 5.8*а*, лес поиска в глубину, представленный на рис. 5.8*б*, демонстрирует все четыре типа различных ребер, которые могут присутствовать в лесу поиска в глубину для ориентированного графа. Это ***ребра дерева*** (tree edge)

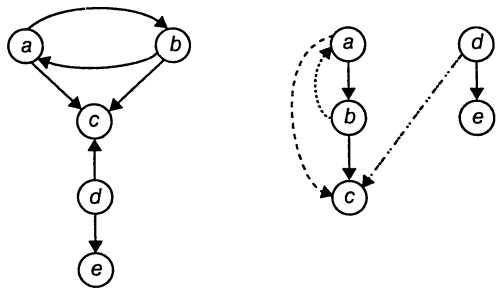


Рис. 5.8. *а*) Ориентированный граф. *б*) Лес поиска в глубину для данного ориентированного графа; начальная вершина поиска — *a*

(*ab*, *bc*, *de*), ***обратные ребра*** (back edge) (*ba*) от вершин к их предшественникам, ***прямые ребра*** (forward edge) (*ac*) от вершин к их потомкам, не являющимся непосредственными дочерними узлами, и ***поперечные ребра*** (cross edge) (*dc*), которые не принадлежат ни к одному из перечисленных ранее типов.

Заметим, что обратное ребро в лесу поиска в глубину может соединять вершину с родительским узлом. Так ли это или нет — в любом случае наличие обратных ребер указывает на то, что ориентированный граф имеет цикл. (***Ориентированный цикл*** в ориентированном графе представляет собой последовательность его вершин, которая начинается и заканчивается в одной и той же вершине и в которой каждая вершина соединена со своим непосредственным предшественником ребром, направленным от предшественника к данной вершине.) Если же лес поиска в глубину для ориентированного графа не имеет обратных ребер, то такой ориентированный граф является ***ориентированным ациклическим графом*** (directed acyclic graph, dag).

Наличие ориентации у ребер графа приводит к новым вопросам, которые для неориентированного графа бессмысленны или тривиальны. В этом разделе будет рассмотрена одна такая задача. В качестве "затравки" рассмотрим следующий пример. Имеется множество из пяти курсов $\{C1, C2, C3, C4, C5\}$, которые должны пройти студенты. Курсы могут быть прочитаны в любом порядке с учетом следующих требований: для прослушивания курса $C3$ следует сначала прослушать курсы $C1$ и $C2$, курс $C4$ следует прослушать после курса $C3$, а $C5$ — после $C3$ и $C4$. Курсы читаются строго последовательно. В каком порядке должны быть прочитаны упомянутые курсы?

Ситуация может быть смоделирована при помощи графа, вершины которого представляют собой курсы, а ориентированные ребра указывают предъявляемые к ним требования (рис. 5.9). В терминах данного ориентированного графа во-

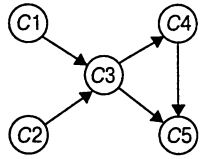


Рис. 5.9. Ориентированный граф, представляющий требования к курсам

прос в том, как перечислить его вершины в таком порядке, чтобы для каждого ребра ориентированного графа вершина, в которой оно начинается, при перечислении оказывалась до вершины, в которой ребро оканчивается (можете ли вы указать такой порядок для приведенного ориентированного графа?). Эта задача называется задачей ***топологической сортировки*** (topological sorting). Она может быть поставлена для любого ориентированного графа, но легко понять, что в случае графа с ориентированным циклом задача неразрешима. Чтобы можно было выполнить топологическую сортировку вершин графа, он должен быть ориентированным ациклическим графом. Оказывается, ориентированный ациклический граф — не только необходимое, но и достаточное условие для возможности топологической сортировки; т.е. если ориентированный граф не имеет циклов, то задача топологической сортировки его вершин разрешима. Имеется два эффективных алгоритма, которые наряду с проверкой того, является ли ориентированный граф ациклическим, выполняют (если это так) топологическую сортировку его вершин.

Первый алгоритм представляет собой простое применение поиска в глубину: выполним поиск в глубину и обратим внимание на порядок, в котором вершины становятся тупиками (т.е. снимаются со стека обхода). Обращение этого порядка дает нам решение задачи топологической сортировки — конечно, если в процессе обхода не встречаются обратные ребра. Наличие обратных ребер говорит о том, что ориентированный граф не является ациклическим, и топологическая сортировка его вершин невозможна.

Почему работает данный алгоритм? Когда вершина v снимается со стека, среди уже снятых со стека к этому моменту вершин нет ни одной вершины u, для которой имеется ребро от u к v (в противном случае ребро (u, v) — обратное). Следовательно, любая такая вершина u будет находиться в списке снятия со стека после v, и перед ней — в обращенном списке.

На рис. 5.10 показано применение этого алгоритма к ориентированному графу, изображенному на рис. 5.9. Обратите внимание на рис. 5.10*в*, где показаны ребра ориентированного графа: все они направлены слева направо, как и требуется в условии задачи. Это — удобный способ визуальной проверки корректности решения экземпляра задачи топологической сортировки.

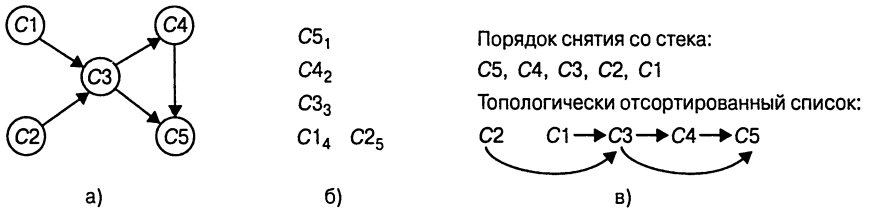


Рис. 5.10. *а*) Ориентированный граф, для которого выполняется топологическая сортировка. *б*) Стек обхода в глубину; индекс указывает порядок снятия со стека. *в*) Решение задачи топологической сортировки

Второй алгоритм основан на непосредственной реализации метода уменьшения размера на единицу: в нем выполняется итеративное определение ***источника*** (source) оставшегося ориентированного графа, представляющего собой вершину, в которую не входит ни одно ребро, и удаление его вместе со всеми исходящими из него ребрами. Если таких источников несколько, произвольно выбирается один из них. Если же источников нет, задача топологической сортировки неразрешима (см. упражнение 5.3.6*а*). Порядок удаления вершин дает решение задачи топологической сортировки. Применение этого алгоритма к уже рассматривавшемуся ориентированному графу пяти курсов показано на рис. 5.11.

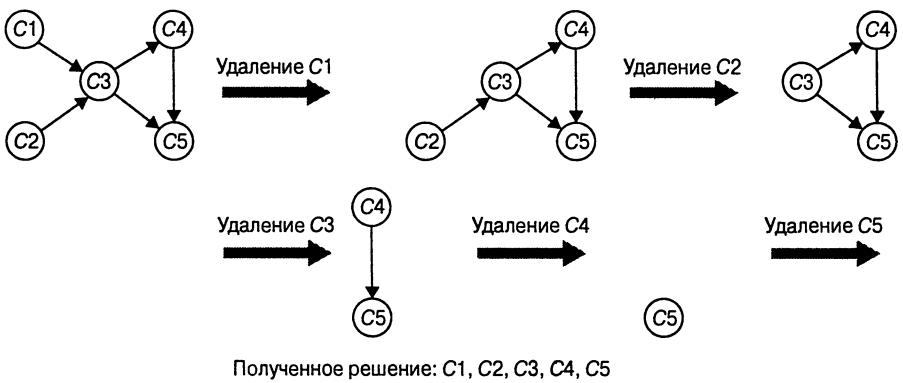


Рис. 5.11. Применение алгоритма удаления источника для решения задачи топологической сортировки. На каждой итерации из ориентированного графа удаляется вершина, в которую не входит ни одно ребро

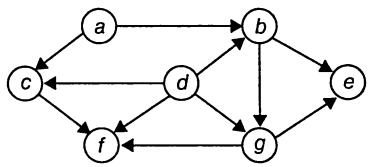
Заметим, что решение, полученное при использовании алгоритма удаления источника, отличается от решения, которое дает алгоритм на основе поиска в глубину. Конечно, оба они корректны, просто задача топологической сортировки может иметь несколько различных решений.

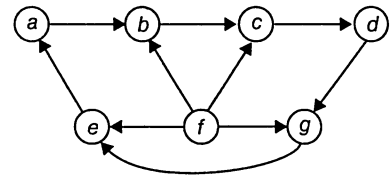
Небольшой размер использованного нами примера ориентированного графа может создать ложное впечатление о задаче топологической сортировки. Но пред-

ставьте сложный проект, например, конструкторский или исследовательский, в который включены тысячи взаимосвязанных заданий с определенными требованиями. Первое, что вы должны сделать в такой ситуации, — убедиться в непротиворечивости требований. Удобным способом сделать это может послужить решение задачи топологической сортировки ориентированного графа проекта. И только после этого вы можете думать о расписании, которое позволит, например, минимизировать полное время работы над проектом. Само собой, для этого потребуется другой алгоритм, который вы сможете найти в книгах по исследованию операций или в специализированной литературе, посвященной методу критического пути (Critical Path Method, CPM) или т.н. системе ПЕРТ (планирования и руководства разработками) (Program Evaluation and Review Technique, PERT).

Упражнения 5.3

1. Примените алгоритм на основе поиска в глубину для решения задачи топологической сортировки следующих ориентированных графов.





2. **а)** Докажите, что задача топологической сортировки ориентированного графа имеет решение тогда и только тогда, когда он ациклический.
   б) Чему равно максимально возможное количество различных решений задачи топологической сортировки ориентированного графа с n вершинами?
3. **а)** Какова временна́я эффективность алгоритма топологической сортировки на основе поиска в глубину?
   б) Как можно модифицировать алгоритм топологической сортировки на основе поиска в глубину, чтобы избежать обращения последовательности вершин, генерируемой при поиске в глубину?
4. Можно ли использовать для решения задачи топологической сортировки последовательность, в которой вершины вносятся в стек поиска в глубину?
5. Примените алгоритм удаления источника к ориентированным графам из упражнения 1.

6. **а)** Докажите, что ориентированный ациклический граф должен иметь как минимум один источник.

   б) Каким образом можно найти вершину, в которую не входит ни одно ребро (или убедиться, что такой вершины не существует), в ориентированном графе, представленном матрицей смежности? Какова временна́я эффективность данной операции?

   в) Каким образом можно найти источник (или убедиться, что он не существует) ориентированного графа, представленного связанными списками смежности? Какова временна́я эффективность данной операции?

7. Можете ли вы реализовать алгоритм удаления источника для ориентированного графа, представленного связанными списками смежности, так, чтобы время его работы составляло $O(|V| + |E|)$?

8. Реализуйте два алгоритма топологической сортировки на вашем любимом языке программирования. Проведите эксперимент по сравнению времени их работы.

9. Ориентированный граф называется ***сильно связным*** (strongly connected), если для любой пары двух различных вершин существует ориентированный путь от u к v и ориентированный путь от v к u. В общем случае вершины ориентированного графа могут быть разбиты на непересекающиеся максимальные подмножества вершин, взаимнодостижимых посредством ориентированных путей ориентированного графа. Такие подмножества называются ***сильно связными компонентами*** (strongly connected components). Есть два алгоритма, основанных на поиске в глубину, позволяющих найти строго связанные компоненты. Вот как выглядит более простой (и менее эффективный) из них.

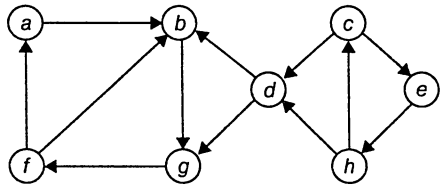
   Шаг 1. Выполнить поиск в глубину в данном ориентированном графе и пронумеровать его вершины в порядке снятия со стека (в котором они становятся тупиками).

   Шаг 2. Обратить направления всех ребер графа.

   Шаг 3. Выполнить поиск в глубину в новом ориентированном графе, начиная (и, при необходимости, повторно начиная) обход с вершины с максимальным номером среди еще непосещенных вершин.

   Сильно связные компоненты представляют собой подмножества вершин в каждом дереве леса поиска в глубину, полученного при последнем обходе.

   а) Примените описанный алгоритм к приведенному ориентированному графу для поиска его сильно связных компонентов.



б) К какому классу временно́й эффективности относится данный алгоритм? Дайте отдельные ответы для представления с использованием матрицы смежности и с применением связанных списков смежности.

в) Сколько строго связных компонентов имеет ориентированный ациклический граф?

10. ***Транзитивное замыкание*** (transitive closure) ориентированного графа с n вершинами можно определить как булеву матрицу T размером $n \times n$ такую, что $T[i,j] = 1$, если имеется путь положительной длины от вершины i к вершине j, и $T[i,j] = 0$ в противном случае $(1 \leqslant i,j \leqslant n)$. Разработайте алгоритм для вычисления транзитивного замыкания и определите его временну́ю эффективность.

5.4 Алгоритмы генерации комбинаторных объектов

В этом разделе мы сдержим данное ранее обещание рассмотреть алгоритмы для генерации комбинаторных объектов. Наиболее важными типами комбинаторных объектов являются перестановки, сочетания и подмножества данного множества. Обычно они возникают в задачах, требующих рассмотрения различных вариантов выбора. В частности, мы встречались с ними в главе 3 при рассмотрении исчерпывающего перебора. Комбинаторные объекты изучаются в разделе дискретной математики, именуемом комбинаторикой. Математиков, конечно, в первую очередь интересуют различные формулы для подсчета таких объектов, и мы должны быть признательны им за эти формулы, которые говорят нам, сколько элементов мы должны сгенерировать. (В частности, эти формулы предупреждают нас о том, что обычно количество комбинаторных объектов с увеличением размера задачи растет экспоненциально или даже быстрее.) Однако сейчас нас больше всего интересуют алгоритмы для генерации комбинаторных объектов, а не для их подсчета.

Генерация перестановок

Начнем с перестановок. Для простоты будем считать, что множество переставляемых элементов — это просто множество целых чисел от 1 до n. В общем случае их можно интерпретировать как индексы элементов n-элементного множества $\{a_1, \ldots, a_n\}$. Как будет выглядеть применение метода уменьшения размера к задаче о генерации всех $n!$ перестановок множества $\{1, \ldots, n\}$? Задача меньшего на единицу размера состоит в генерации всех $(n-1)!$ перестановок. Полагая, что задача меньшего размера успешно решена, мы можем получить решение большей задачи путем вставки n в каждую из n возможных позиций среди элементов каждой из перестановок $n-1$ элементов. Все получаемые таким образом перестановки будут различны (почему?), а их общее количество будет равно $n(n-1)! = n!$. Следовательно, таким образом мы получим все перестановки множества $\{1, \ldots, n\}$.

Мы можем вставлять n в ранее сгенерированные перестановки слева направо или справа налево. Как оказывается, выгодно начинать вставку n в последовательность $1\,2\,\ldots\,(n-1)$ справа налево и изменять направление всякий раз при переходе к новой перестановке множества $\{1, \ldots, n-1\}$. Пример применения такого подхода для $n = 3$ показан на рис. 5.12.

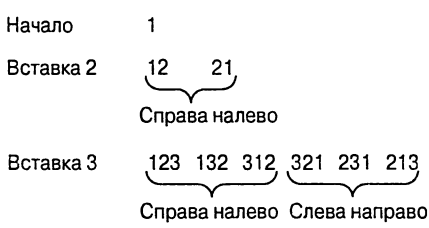


Рис. 5.12. Восходящая генерация перестановок (снизу вверх)

Преимущество такого порядка генерации перестановок связано с тем фактом, что этот порядок удовлетворяет так называемому требованию ***минимальных изменений*** (minimal-change requirement): каждая перестановка получается из свей непосредственной предшественницы при помощи обмена местами только двух элементов (проверьте выполнение этого свойства для перестановок, показанных на рис. 5.12). Требование минимальных изменений выгодно как с точки зрения скорости работы алгоритма, так и для приложения, которое будет использовать сгенерированные перестановки. Например, в разделе 3.4 нам были нужны перестановки городов для решения задачи коммивояжера методом исчерпывающего перебора. Если такие перестановки генерируются алгоритмом, удовлетворяющим требованию минимальных изменений, то мы можем вычислить длину нового

маршрута на основе длины старого маршрута за постоянное, а не линейное время (как?).

Можно получить тот же порядок перестановок n элементов и без явной генерации перестановок для меньших значений n. Это можно сделать, связав с каждым компонентом k перестановки направление. Будем указывать это направление при помощи стрелки над рассматриваемым элементом, например:

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}.$$

Компонент k в такой перестановке с использованием стрелок называется ***мобильным*** (mobile), если стрелка указывает на меньшее соседнее число. Например, в перестановке $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$ числа 3 и 4 мобильны, а 2 и 1 — нет. Воспользовавшись понятием мобильного элемента, мы получаем следующее описание так называемого ***алгоритма Джонсона–Троттера*** (Johnson–Trotter algorithm) для генерации перестановок.

Алгоритм $JohnsonTrotter\,(n)$

// Реализация алгоритма Джонсона–Троттера
// для генерации перестановок
// **Входные данные:** Натуральное число n
// **Выходные данные:** Список перестановок множества $\{1, \ldots, n\}$
Инициализируем первую перестановку значением $\overleftarrow{1}\ \overleftarrow{2}\ \ldots\ \overleftarrow{n}$
while (пока) имеется мобильное число k **do**
  Находим наибольшее мобильное число k
  Меняем местами k и соседнее целое число,
  на которое указывает стрелка у k
  Меняем направление стрелок у всех целых
  чисел, больших k

Вот пример использования этого алгоритма для $n = 3$ (наибольшее мобильное число показано полужирным шрифтом):

$$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{\mathbf{3}}\quad \overleftarrow{1}\ \overleftarrow{\mathbf{3}}\ \overleftarrow{2}\quad \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{\mathbf{2}}\quad \overrightarrow{\mathbf{3}}\ \overleftarrow{2}\ \overleftarrow{1}\quad \overleftarrow{2}\ \overrightarrow{\mathbf{3}}\ \overleftarrow{1}\quad \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}.$$

Этот алгоритм — один из наиболее эффективных для генерации перестановок и может быть реализован со временем работы, пропорциональным количеству перестановок, т.е. $\Theta\,(n!)$. Конечно, он очень медленный для всех значений n, кроме самых малых, но это беда не алгоритма, а задачи, которая требует от алгоритма генерации слишком большого количества элементов.

Можно показать, что порядок перестановок, генерируемых алгоритмом Джонсона–Троттера, не совсем естественный; например, было бы естественным ожидать, что последняя перестановка в списке будет иметь вид $n\,(n-1)\ldots 1$. Именно такая перестановка окажется последней, если перестановки будут упорядочены

в соответствии с ***лексикографическим порядком*** (lexicographic order), т.е. порядком, в котором они были бы перечислены в словаре, если рассматривать цифры как буквы алфавита:

$$1\,2\,3 \quad 1\,3\,2 \quad 2\,1\,3 \quad 2\,3\,1 \quad 3\,1\,2 \quad 3\,2\,1.$$

Каким образом мы можем сгенерировать перестановку, следующую за $a_1 a_2 \ldots a_{n-1} a_n$ в лексикографическом порядке? Если $a_{n-1} < a_n$, мы просто меняем местами два последних элемента (например, за 1 2 3 следует 1 3 2). Если $a_{n-1} > a_n$, мы должны обратиться к элементу a_{n-2}. Если $a_{n-2} < a_{n-1}$, мы должны переставить последние три элемента, минимально увеличивая $(n-2)$-ой элемент , т.е. помещая на это место следующий больший a_{n-2} элемент, выбранный из a_{n-1} и a_n, и заполняя позиции $(n-1)$ и n оставшимися двумя из трех элементов a_{n-2}, a_{n-1} и a_n в возрастающем порядке. Например, за 1 3 2 следует 2 1 3, а за 2 3 1 — 3 1 2. В общем случае мы сканируем текущую перестановку справа налево в поисках первой пары соседних элементов a_i и a_{i+1} таких, что $a_i < a_{i+1}$ (и, следовательно, $a_{i+1} > \cdots > a_n$). Затем мы находим наименьший элемент из "хвоста", больший a_i, т.е. $\min\{a_j \mid a_j > a_i, j > i\}$, и помещаем его в позицию i; позиции с $i+1$-ой по n-ую заполняются элементами $a_i, a_{i+1}, \ldots, a_n$, из которых изъят элемент для вставки в позицию i, в возрастающем порядке. Написание полного псевдокода этого алгоритма остается читателю в качестве самостоятельного упражнения.

Генерация подмножеств

Вспомним, как в разделе 3.5 мы рассматривали задачу о рюкзаке, в которой требовалось найти наиболее ценное подмножество элементов, размещающихся в рюкзаке данного объема. Мы рассматривали подход к решению этой задачи с использованием исчерпывающего перебора, который основан на генерации всех подмножеств данного множества элементов. В этом разделе мы рассмотрим алгоритм для генерации всех 2^n подмножеств абстрактного множества $A = \{a_1, \ldots, a_n\}$ (математики называют множество всех подмножеств данного множества ***показательным множеством*** (power set)).

К этой задаче также непосредственно применим метод уменьшения размера на единицу. Все подмножества множества $A = \{a_1, \ldots, a_n\}$ можно разделить на две группы — те, которые содержат элемент a_n, и те, которые не содержат его. Первая группа представляет собой не что иное, как все подмножества множества $\{a_1, \ldots, a_{n-1}\}$, в то время как все элементы второй группы можно получить путем добавления элемента a_n к подмножествам множества $\{a_1, \ldots, a_{n-1}\}$. Следовательно, как только мы получим список всех подмножеств множества $\{a_1, \ldots, a_{n-1}\}$, мы можем получить все подмножества множества $\{a_1, \ldots, a_n\}$, просто добавляя к списку все его элементы с добавленным к каждому из них

элементом a_n. Применение этого алгоритма для генерации всех подмножеств множества $\{a_1, a_2, a_3\}$ показано в табл. 5.2.

Таблица 5.2. Восходящая генерация подмножеств

n	**Подмножества**							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Как и в случае перестановок, мы не обязаны генерировать все показательные множества для множеств меньших размеров. Удобный способ решения поставленной задачи основан на взаимно однозначном соответствии между всеми 2^n подмножествами n-элементного множества $A = \{a_1, \ldots, a_n\}$ и всеми 2^n битовыми строками $b_1, \ldots, b_n$ длиной n. Простейший способ установить такое соответствие — назначить подмножеству битовую строку, в которой $b_i = 1$, если a_i принадлежит данному подмножеству, и $b_i = 0$ в противном случае (эта идея уже упоминалась в разделе 1.4, когда мы говорили о битовых векторах). Например, битовая строка 000 соответствует пустому подмножеству трехэлементного множества, а 111 — самому исходному трехэлементному множеству, т.е. $\{a_1, a_2, a_3\}$. Понятно, что строка 110 представляет подмножество $\{a_1, a_2\}$. Используя такое соответствие, мы можем сгенерировать все битовые строки длиной n, просто генерируя последовательно двоичные числа от 0 до $2^n - 1$, добавляя при необходимости соответствующее количество ведущих нулей. Например, при $n = 3$ мы получим:

Битовые строки	000	001	010	011	100	101	110	111
Подмножества	$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Заметим, что в то время как битовые строки, сгенерированные данным алгоритмом, находятся в лексикографическом порядке (с использованием двухсимвольного алфавита $\{0, 1\}$), порядок подмножеств выглядит далеко не естественно. Например, мы можем захотеть получить так называемый ***плотный порядок*** (squashed order), когда подмножество, включающее a_j, может находиться в списке только после всех подмножеств, включающих элементы $a_1, \ldots, a_{j-1}$ ($j = 1, \ldots, n$), как, например, в случае трехэлементного исходного множества, показанного в табл. 5.2. Описанный алгоритм, основанный на использовании битовых строк, весьма легко модифицировать для получения желаемого плотного порядка (см. упражнение 5.4.6).

Более интересный вопрос — о существовании алгоритма генерации битовых строк, соответствующего требованию минимальных изменений, так чтобы каждая

строка отличалась от непосредственно предшествующей ей только одним битом (в переводе на язык подмножеств это означает, что каждое подмножество отличается от своего предшественника в списке либо добавлением, либо удалением одного элемента, но не более). Ответ на этот вопрос положительный; например, для $n = 3$ мы можем получить строки

$$000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100.$$

Это — пример ***кода Грея*** (Gray code). Коды Грея обладают многими интересными свойствами и рядом полезных приложений, о чем вы можете прочесть в [25].

Упражнения 5.4

1. Реально ли реализовать алгоритм, который требует генерации всех перестановок 25-элементного множества, на вашем компьютере? А в случае генерации всех подмножеств данного множества?
2. Сгенерируйте все перестановки множества $\{1, 2, 3, 4\}$ с использованием
   а) восходящего алгоритма, соответствующего требованию минимальных изменений;
   б) алгоритма Джонсона–Троттера;
   в) алгоритма лексикографического упорядочения.
3. Напишите компьютерную программу для генерации перестановок в лексикографическом порядке.
4. Рассмотрим следующую реализацию алгоритма для генерации перестановок, открытого Хипом (B. Heap) [50].

```
АЛГОРИТМ HeapPermute(n)
  // Реализация алгоритма Хипа для генерации перестановок
  // Входные данные:  Натуральное n и глобальный массив A[1..n]
  // Выходные данные: Все перестановки элементов множества A
  if n = 1
    write A
  else
    for i ← 1 to n do
      HeapPermute(n − 1)
      if n нечетно
        Обмениваем A[1] и A[n]
```

else
Обмениваем $A[i]$ и $A[n]$

а) Пошагово пройдите алгоритм для $n = 2$, 3 и 4.

б) Докажите корректность алгоритма Хипа.

в) Определите временну́ю эффективность алгоритма $HeapPermute$.

5. Сгенерируйте все подмножества четырехэлементного множества $A = \{a_1, a_2, a_3, a_4\}$ при помощи обоих описанных в тексте раздела алгоритмов.

6. Какой простой трюк заставляет алгоритм с использованием битовых строк генерировать подмножества в плотном порядке.

7. Напишите псевдокод для рекурсивного алгоритма генерации всех 2^n битовых строк длиной n.

8. Напишите нерекурсивный алгоритм для генерации 2^n битовых строк длиной n, который реализует битовые строки в виде массивов и не использует бинарного сложения.

9. а) Используйте метод уменьшения размера на единицу для генерации кода Грея для $n = 4$.

б) Разработайте алгоритм на основе метода уменьшения размера на единицу для генерации кода Грея произвольного порядка n.

10. Разработайте алгоритм на основе метода уменьшения размера для генерации всех комбинаций из k элементов, выбранных из n-элементного множества (т.е. всех k-элементных подмножеств данного n-элементного множества). Удовлетворяет ли ваш алгоритм требованию минимальных изменений?

5.5 Алгоритмы с использованием уменьшения на постоянный множитель

Во введении к этой главе упоминалось, что алгоритмы, основанные на уменьшении размера задачи на постоянный множитель — вторая важная разновидность алгоритмов, основанных на уменьшении размера задачи. Мы уже встречались в книге с примерами этого метода — это бинарный поиск (раздел 4.3) и возведение в степень посредством возведения в квадрат (введение к данной главе). В этом разделе мы познакомимся с другими примерами алгоритмов, основанных на уменьшении размера задачи на постоянный множитель. Однако мы не должны ожидать наличия большого количества таких алгоритмов. Обычно такие алго-

ритмы — логарифмические и, будучи очень быстрыми, встречаются достаточно редко. Особенно большая редкость — множитель, не равный двум.

Задача поиска фальшивой монеты

Из целого ряда различных задач поиска фальшивой монеты мы рассмотрим одну, которая наилучшим образом иллюстрирует интересующий нас метод уменьшения размера задачи на постоянный множитель. Среди n одинаково выглядящих монет одна — фальшивая. На рычажных весах вы можете сравнить любые два множества монет и получить ответ, какое из множеств тяжелее (или что они равны по весу), но не на какую именно величину. Задача заключается в том, чтобы разработать эффективный алгоритм для поиска фальшивой монеты. В простейшей версии задачи, рассматриваемой в этом разделе, считается, что вы знаете, легче или тяжелее фальшивая монета по сравнению с подлинной.[3] (Здесь мы будем считать, что фальшивая монета легче настоящей.)

Наиболее естественный подход к решению данной версии задачи — разделить n монет на две кучки по $\lfloor n/2 \rfloor$ монет в каждой, отложив одну монету, если n — нечетное число. Если кучки имеют одинаковый вес, то отложенная монета и является фальшивой; если нет — мы выбираем более легкую кучку и выполняем описанные действия с ней. Заметим, что, хотя мы и разделили все монеты на две кучки, после взвешивания надо решить только одну задачу половинного размера. Следовательно, в соответствии с нашей классификацией методов проектирования алгоритмов, этот алгоритм относится к алгоритмам уменьшения размера задачи (вдвое), а не к алгоритмам на основе декомпозиции.

Легко записать рекуррентное уравнение для количества взвешиваний $W(n)$, необходимых алгоритму в наихудшем случае:

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \quad \text{при } n > 1, W(1) = 0.$$

Это рекуррентное соотношение должно быть вам знакомо. В самом деле, оно практически идентично соотношению для количества сравнений в бинарном поиске в наихудшем случае (отличие только в начальном условии). Эта схожесть неудивительна, поскольку оба алгоритма основаны на одном и том же методе — уменьшения размера экземпляра задачи в два раза. Решение данного рекуррентного соотношения также очень похоже на решение соотношения для бинарного поиска: $W(n) = \lfloor \log_2 n \rfloor$.

Все это выглядит элементарно, если не просто надоедливо. Но подождите немного: самое интересное в том, что это решение — не самое эффективное. Мы можем поступить разумнее, если будем делить монеты на *три* кучки, примерно по

[3] В более сложной постановке задачи вам ничего не известно о том, как именно соотносятся вес фальшивой и подлинной монеты, и даже неизвестно, имеется ли она среди n монет. Такая, более сложная версия задачи будет рассмотрена в разделе 10.2.

$n/3$ монет в каждой. (Детали точной формулировки остаются в качестве упражнения. Не пропустите его! Даже если преподаватель забудет об упражнении 5.5.3, напомните ему о том, что это упражнение должно быть обязательно выполнено.) После взвешивания двух кучек мы можем уменьшить размер экземпляра задачи в три раза, так что следует ожидать, что необходимое количество взвешиваний будет примерно равно $\log_3 n$, что меньше, чем $\log_2 n$ (можете ли вы сказать, во сколько раз?).

Умножение по-русски

Сейчас мы рассмотрим один нестандартный алгоритм для умножения двух положительных чисел, который называется ***умножением по-русски*** (multiplication à la russe), или ***русским крестьянским методом*** (Russian peasant method). Пусть n и m — натуральные числа, произведение которых мы хотим вычислить, а размером экземпляра задачи будем считать величину n. Теперь, если n четно, экземпляр задачи уменьшается вдвое, до $n/2$, и для решения большего экземпляра задачи мы используем решение меньшего экземпляра при помощи очевидной формулы

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

Если n нечетно, требуется внесение в формулу небольшой поправки:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

Используя приведенные формулы и тривиальный частный случай $1 \cdot m = m$ для остановки, мы можем рекурсивно или итеративно вычислить произведение $n \cdot m$. Пример вычисления $50 \cdot 65$ приведен на рис. 5.13. Обратите внимание, что дополнительные слагаемые (показанные в скобках) имеются в тех строках, где в первом столбце стоит нечетное значение. Таким образом, при умножении вручную никакие скобки (которые для ясности показаны на рис. 5.13*а*) не требуются, и для получения результата надо просто просуммировать значения из столбца m, соответствующие нечетным значениям в столбце n, что и сделано на рис. 5.13*б*.

Заметим также, что в алгоритме используются только простые операции удвоения, деления пополам и сложения — это должно показаться привлекательным тем, у кого проблемы с таблицей умножения. По крайней мере, как утверждают западные путешественники, русские крестьяне (именем которых и назван данный метод) еще в XIX веке использовали итеративные алгоритмы, основанные на методе уменьшения размера задачи вдвое. (Впрочем, идея этого алгоритма была известна еще примерно в 1650 г. до н.э. египетским математикам — см. [26], стр. 16.) Этот алгоритм приводит к очень быстрой аппаратной реализации умножения, поскольку удвоение и деление пополам двоичных чисел реализуются при помощи сдвига, который представляет собой одну из базовых операций на аппаратном уровне.

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	$+ (130 + 1040) = 3250$

а)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

б)

Рис. 5.13. Вычисление $50 \cdot 65$ умножением по-русски

Задача Иосифа

Наш последний пример — ***задача Иосифа*** (Josephus problem), названная так по имени Флавия Иосифа (Flavius Josephus), известного еврейского историка, участника и летописца восстания евреев против римлян в 66–70 гг. Иосиф в течение 47 дней командовал обороной крепости Иотапата (Jotapata), и после того, как она пала, укрылся с 40 фанатиками в пещере неподалеку. Здесь повстанцы решили покончить с собой, чтобы не сдаться римлянам. Иосиф предложил, чтобы все встали в круг и по очереди каждый убивал своего соседа (т.е. каждого второго)[4], пока не останется один человек, который должен совершить самоубийство. Нечего и говорить, что решивший сдаться Иосиф поставил себя и своего единомышленника на такие места в круге, что они остались последними.

Итак, пусть n человек, пронумерованных от 1 до n, встали в круг. Начнем счет с человека под номером 1 и будем убирать из круга каждого второго, пока в круге не останется только один человек. Задача состоит в том, чтобы определить номер этого уцелевшего $J(n)$. Например, если $n = 6$ (см. рис. 5.14), то при первом проходе из круга будут удалены номера 2, 4 и 6, а на втором проходе — номера 3 и 1, так что решение задачи равно $J(6) = 5$. Если $n = 7$, то при первом проходе будут удалены номера 2, 4, 6 и 1 (оказывается удобным включить удаление 1 в первый проход), а на втором — 5 и 3 (удаление 3 удобнее отнести ко второму проходу), так что решением данного экземпляра задачи является $J(7) = 7$.

Удобно рассматривать четные и нечетные значения n отдельно. Если n четно, т.е. $n = 2k$, то после первого прохода мы получаем экземпляр той же задачи, но половинного размера. Единственное отличие — в нумерации: например, чело-

[4] По другим сведениям, — каждого третьего. О задаче Иосифа и ее разновидностях можно прочесть в книге У. Болл, Г. Коксетер. *Математические эссе и развлечения*. — М.: Мир, 1986 (стр. 43–47). — *Прим. ред.*

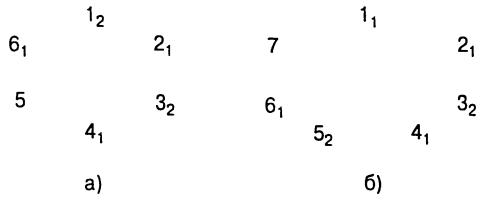


Рис. 5.14. Экземпляры задачи Иосифа при *а*) $n = 6$ и *б*) $n = 7$. Индексы указывают номер прохода, во время которого устраняется данный человек. Решения показанных экземпляров равны $J(6) = 5$ и $J(7) = 7$

век номер 3 на втором проходе находится в позиции 2, номер 5 — в позиции 3 и т.д. Легко увидеть, что для того, чтобы получить начальную позицию человека, надо просто умножить его новую позицию на 2 и вычесть 1. Это соотношение выполняется, в частности, для уцелевшего человека, так что

$$J(2k) = 2J(k) - 1.$$

Давайте теперь рассмотрим случай нечетного $n > 1$, т.е. $n = 2k + 1$. При первом проходе удаляются все люди в четных позициях. Если мы добавим сюда и человека номер 1, то получим экземпляр задачи размера k. В этот раз, чтобы получить начальную позицию человека по его новой позиции, мы должны умножить новую позицию на 2 и прибавить 1 (см. рис. 5.14*б*). Итак, для нечетных значения n получаем

$$J(2k + 1) = 2J(k) + 1.$$

Можем ли мы получить решение этих двух рекуррентных соотношений в явном виде (с учетом начального условия $J(1) = 1$)? Ответ положителен, хотя решение и требует более сложных приемов, чем простая обратная подстановка. Один из способов решения — применение прямой подстановки для получения, скажем, пятнадцати первых значений $J(n)$, выявление закономерности и ее доказательство методом математической индукции. Выполнение этих действий оставлено читателю в качестве самостоятельного упражнения. Вы можете также обратиться к [45] и найти там все интересующие вас сведения (данный раздел изложен по материалам этой книги). Интересно, что наиболее элегантный вид решения рекуррентных соотношений использует бинарное представление числа n: $J(n)$ можно получить путем циклического сдвига n на один бит влево! Например, $J(6) = J(110_2) = 101_2 = 5$ и $J(7) = J(111_2) = 111_2 = 7$.

Упражнения 5.5

1. Разработайте алгоритм вычисления $\lfloor \log_2 n \rfloor$, основанный на методе уменьшения размера задачи вдвое, и определите его временну́ю эффективность.
2. Рассмотрим ***тернарный поиск*** (ternary search) — следующий алгоритм для поиска в отсортированном массиве $A[0..n-1]$: если $n = 1$, мы просто сравниваем ключ K с единственным элементом массива; в противном случае поиск рекурсивно сравнивает K с $A[\lfloor n/3 \rfloor]$ и, если K больше, сравнивает его с $A[\lfloor 2n/3 \rfloor]$ для определения того, в какой трети массива следует продолжить поиск.
   а) На каком методе основан данный алгоритм?
   б) Запишите рекуррентное соотношение для количества сравнений ключей в наихудшем случае (можно считать, что $n = 3^k$).
   в) Решите это рекуррентное соотношение при $n = 3^k$.
   г) Сравните эффективность данного алгоритма с эффективностью бинарного поиска.

3. **а)** Напишите псевдокод алгоритма поиска фальшивой монеты путем деления на три кучи. (Убедитесь, что ваш алгоритм корректно работает для любого значения n, а не только для кратного 3.)
   б) Запишите рекуррентное соотношение для количества взвешиваний в алгоритме поиска фальшивой монеты путем деления на три кучи и решите его для $n = 3^k$.
   в) Во сколько раз этот алгоритм быстрее алгоритма поиска фальшивой монеты путем деления на две кучи при больших n? (Ответ не должен зависеть от n.)
4. Примените метод умножения по-русски для вычисления $26 \cdot 47$.
5. **а)** Имеет ли значение с точки зрения эффективности, выполняем ли мы умножение по-русски $n \cdot m$ или $m \cdot n$?
   б) Определите класс эффективности умножения по-русски.
6. Напишите псевдокод алгоритма умножения по-русски.
7. Найдите $J(40)$ — решения задачи Иосифа для $n = 40$.
8. Докажите, что решение задачи Иосифа равно 1 для всех n, являющихся степенью 2.
9. Для задачи Иосифа
   а) Вычислите $J(n)$ для $n = 1, 2, \ldots, 15$.
   б) Найдите закономерность в решениях задачи Иосифа для первых пятнадцати значений n и докажите ее справедливость в общем случае.

в) Докажите корректность получения значения $J(n)$ путем циклического сдвига бинарного представления n.

5.6 Алгоритмы с переменным уменьшением размера

Как упоминалось во введении к данной главе, третьим основным вариантом метода уменьшения размера задачи является уменьшение, меняющееся от итерации к итерации. Хорошим примером такого алгоритма является алгоритм Евклида для вычисления наибольшего общего делителя. В этом разделе мы познакомимся и с другими представителями этого класса алгоритмов.

Вычисление медианы и задача выбора

Задача выбора (selection problem) заключается в поиске k-го наименьшего элемента в списке из n чисел. Это число называется k-ой ***порядковой статистикой*** (order statistic). Естественно, при $k = 1$ или $k = n$ можно просто сканировать весь список в поисках наименьшего или наибольшего элемента. Более интересен случай, когда $k = \lceil n/2 \rceil$, т.е. когда надо найти элемент, больший одной половины элементов списка, и меньший — другой половины. Такое среднее значение, называемое ***медианой*** (median), является одной из наиболее важных величин в математической статистике. Очевидно, мы можем найти k-ый по величине элемент, отсортировав весь список и выбрав k-ый по порядку элемент из отсортированного списка. Время работы такого алгоритма определяется эффективностью используемого алгоритма сортировки. При использовании хорошего алгоритма типа сортировки слиянием эффективность такого алгоритма поиска порядковой статистики равна $O(n \log n)$.

Закрадывается подозрение, что сортировка всего списка — выполнение лишней работы, поскольку в задаче не требуется сортировать список, а надо только указать один-единственный элемент. К счастью, у нас имеется очень эффективный (в среднем) алгоритм для выполнения похожей задачи — разбиения элементов массива на два подмножества: одно из них содержит элементы, не превышающее некоторое опорное значение p, а второе — элементы, которые не меньше p:

$$\underbrace{a_{i_1} \dots a_{i_{s-1}}}_{\leqslant p} \; p \; \underbrace{a_{i_{s+1}} \dots a_{i_n}}_{\geqslant p}.$$

Такое разбиение является важной частью алгоритма быстрой сортировки, рассматривавшейся в главе 4.

Как воспользоваться преимуществами такого разбиения? Пусть s — позиция разбиения. Если $s = k$, то опорный элемент p и есть решением задачи выбора. (Ес-

ли элементы в списке нумеруются начиная с 0, то, конечно, решение получается при $s = k - 1$.) Если $s > k$, то k-ый наименьший элемент находится в левой части списка, а если $s < k$, то надо найти $(k - s)$-ый наименьший элемент из правой части списка. Итак, если мы не получаем решение задачи на данной итерации, то по крайней мере уменьшаем ее размер и получаем задачу, которая решается таким же методом, т.е. рекурсивно. На самом деле реализовать эту же идею можно и без применения рекурсии. При использовании нерекурсивной версии даже не надо изменять значение k — мы просто продолжаем выполнение вычислений, пока не получим $s = k$.

Пример 1. Найдем медиану списка из девяти элементов: 4, 1, 10, 9, 7, 12, 8, 2, 15. В этом случае $k = \lceil 9/2 \rceil = 5$, и наша задача состоит в поиске пятого по величине элемента массива. Как и ранее, мы предполагаем, что элементы в списке проиндексированы от 1 до 9.

Мы можем воспользоваться той же версией разбиения, которой пользовались при рассмотрении алгоритма быстрой сортировки в главе 4, т.е. выбирая в качестве опорного первый элемент и переставляя элементы в процессе двух противоположно направленных сканирований массива:

$$\begin{array}{cccccccccc} \mathbf{4} & 1 & 10 & 9 & 7 & 12 & 8 & 2 & 15 \\ 2 & 1 & \mathbf{4} & 9 & 7 & 12 & 8 & 10 & 15 \end{array}$$

Поскольку $s = 3 < k = 5$, продолжаем работу с правой частью списка:

$$\begin{array}{cccccc} \mathbf{9} & 7 & 12 & 8 & 10 & 15 \\ 8 & 7 & \mathbf{9} & 12 & 10 & 15 \end{array}$$

Поскольку $s = 6 > k = 5$, мы работаем с левой частью списка:

$$\begin{array}{cc} \mathbf{8} & 7 \\ 7 & \mathbf{8} \end{array}$$

Теперь $s = k = 5$, так что можно остановиться: мы нашли медиану, равную 8 — которая больше 2, 1, 4 и 7, но меньше 9, 12, 10 и 15. ■

Насколько эффективен этот алгоритм? Следует ожидать, что в среднем он более эффективен, чем быстрая сортировка, так как работает только с одной частью исходного массива после разбиения, в то время как быстрая сортировка должна обработать обе части. Если считать, что разбиение всегда выполняется посредине остающейся части массива, рекуррентное соотношение для количества сравнений будет иметь следующий вид:

$$C(n) = C(n/2) + (n + 1),$$

решение которого, согласно Основной теореме (см. главу 4), равно $\Theta(n)$. Хотя на самом деле размер массива уменьшается от одной итерации к другой непредсказуемым образом (иногда это уменьшение меньше, чем в два раза, иногда — больше), тщательный математический анализ показывает, что эффективность в среднем случае будет такой же, как если бы уменьшение размера задачи всякий раз было ровно в два раза. Другими словами, в среднем мы получаем линейный алгоритм. В худшем же случае эффективность алгоритма падает до $\Theta(n^2)$. Хотя кибернетики и открыли алгоритм, который решает задачу выбора за линейное время даже в наихудшем случае [19], он слишком сложен для практического применения.

Заметим также, что на самом деле алгоритм на основе разбиения решает немного более общую задачу, а именно: определение k наименьших и $n-k$ наибольших элементов данного списка, а не только поиск k-го элемента в порядке возрастания.

Интерполяционный поиск

В качестве следующего примера алгоритма с переменным уменьшением размера задачи мы рассмотрим еще один алгоритм поиска в отсортированном массиве, который называется ***интерполяционным поиском*** (interpolation search). В отличие от бинарного поиска, который всегда сравнивает ключ поиска со средним значением отсортированного массива (а следовательно, всегда уменьшает размер задачи вдвое), интерполяционный поиск учитывает значение ключа поиска при определении элемента массива, который будет сравниваться с ключом. В определенном смысле алгоритм имитирует поиск имени в телефонной книге. Если мы ищем в телефонной книге, например, Горбенко — вряд ли мы будем открывать ее в средине или ближе к концу, как поступили бы при поиске Подгорного.

Говоря более строго. при выполнении итерации поиска между элементами $A[l]$ (крайним слева) и $A[r]$ (крайним справа), алгоритм предполагает, что значения в массиве растут линейно (отличие от линейности может влиять на эффективность, но не на корректность данного алгоритма). В соответствии с этим предположением, значение v ключа поиска сравнивается с элементом, индекс которого вычисляется (с округлением) как координата x точки на прямой, проходящей через $(l, A[l])$ и $(r, A[r])$, координата y которой равна значению v (см. рис. 5.15).

Записав стандартное уравнение для прямой, проходящей через две точки $(l, A[l])$ и $(r, A[r])$, заменив в нем y на v и решая его относительно x, получим формулу

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor. \tag{5.4}$$

Логика, лежащая в основе этого метода, проста. Мы знаем, что значения массива возрастают (точнее говоря, не убывают) от $A[l]$ до $A[r]$, но не знаем, как именно. Пусть это возрастание — линейное (простейшая из возможных функци-

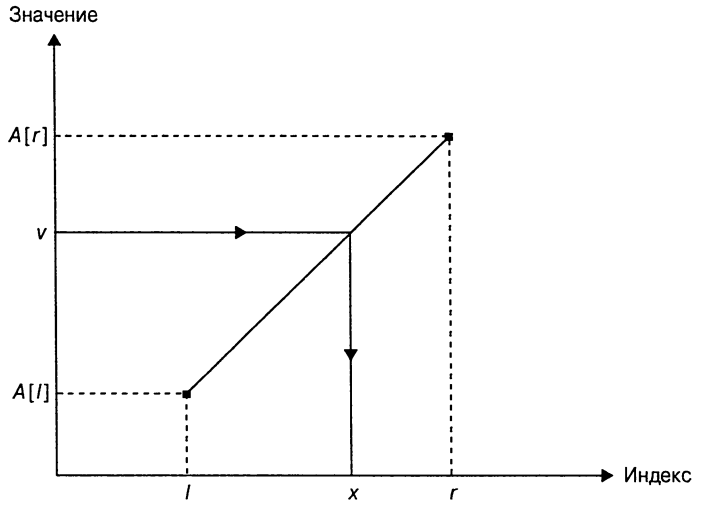


Рис. 5.15. Вычисление индекса при интерполяционном поиске

ональных зависимостей); в таком случае вычисленное по формуле (5.4) значение индекса — ожидаемая позиция элемента со значением, равным v. Конечно, если v не находится между $A[l]$ и $A[r]$, формулу (5.4) применять не следует (почему?).

После сравнения v с $A[x]$ алгоритм либо прекращает работу (если они равны), либо продолжает поиск тем же способом среди элементов с индексами либо от l до $x-1$, либо от $x+1$ до r, в зависимости от того, меньше ли v значения $A[x]$ или больше. Таким образом, на каждой итерации размер задачи уменьшается, но априори мы не знаем, насколько именно.

Анализ эффективности данного алгоритма показывает, что интерполяционный поиск в среднем требует менее $\log_2 \log_2 n+1$ сравнений ключей при поиске в списке из n случайных значений. Эта функция растет настолько медленно, что для всех реальных практических значений n ее можно считать константой (см. упражнение 5.6.6). Однако в наихудшем случае интерполяционный поиск вырождается в линейный, который рассматривается как наихудший из возможных (почему?). В качестве последнего замечания по поводу сравнения интерполяционного поиска с бинарным приведем мнение Седжвика (R. Sedgewick) [102], считающего, что бинарный поиск, вероятно, более выгоден для небольших входных данных, но для файлов большого размера и для приложений, в которых сравнение или обращение к данным — дорогостоящая операция, лучше использовать интерполяционный поиск. Упомянем, что в разделе 11.4 будет рассмотрена непрерывная версия интерполяционного поиска, которую также можно рассматривать как еще один пример алгоритма, основанного на переменном уменьшении размера задачи.

Поиск и вставка в бинарное дерево поиска

В качестве последнего примера в данном разделе вернемся к бинарным деревьям поиска. Вспомним, что бинарное дерево поиска — это бинарное дерево, узлы которого содержат упорядочиваемые элементы, по одному в каждом узле, так что для каждого узла все элементы в левом поддереве меньше, а все элементы в правом поддереве больше элемента в корне поддерева. Когда надо найти элемент с заданным значением (скажем, v) в таком дереве, мы рекурсивно повторяем следующие действия. Если дерево пустое — поиск завершается неудачно. Если дерево не пустое, мы сравниваем значение v со значением в корне дерева $K(r)$. Если они равны, искомый элемент найден, и поиск завершается успешно. Если же они не равны, то мы продолжаем поиск в левом поддереве, если $v < K(r)$, и в правом, если $v > K(r)$. Таким образом, на каждой итерации алгоритма задача поиска в бинарном дереве поиска сводится к задаче поиска в бинарном дереве меньшего размера. Наиболее разумной мерой размера бинарного дерева поиска является его высота; очевидно, что уменьшение высоты дерева может изменяться от итерации к итерации — это дает нам пример алгоритма с переменным уменьшением размера задачи.

В худшем случае бинарного дерева поиска оно строго линейное. Такое бывает, в частности, если дерево строится путем вставки возрастающей или убывающей последовательности ключей (рис. 5.16).

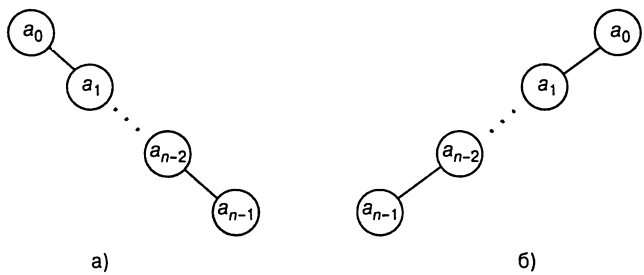


Рис. 5.16. Бинарные деревья поиска для *а*) возрастающей последовательности ключей и *б*) убывающей последовательности ключей

Очевидно, что поиск a_{n-1} в таком дереве требует n сравнений, что приводит к эффективности $\Theta(n)$ в наихудшем случае. К счастью, эффективность поиска в среднем случае — $\Theta(\log n)$. Говоря более точно, количество сравнений ключей, необходимое для поиска в бинарном дереве поиска, построенном из n случайных ключей, приблизительно равно $2 \ln n \approx 1.39 \log_2 n$. Поскольку вставка нового ключа в бинарное дерево поиска практически идентична поиску в нем, задача вставки в бинарное дерево поиска также является примером алгоритма

с переменным уменьшением размера задачи и имеет ту же эффективность, что и операция поиска.

Упражнения 5.6

1. **а)** Если измерять размер экземпляра задачи по вычислению наибольшего общего делителя m и n величиной второго параметра n, то насколько может уменьшиться размер экземпляра задачи при вычислении $\gcd(m, n)$ при помощи алгоритма Евклида?
   б) Докажите, что размер задачи поиска наибольшего общего делителя после двух последовательных итераций алгоритма Евклида уменьшается как минимум в два раза.
2. **а)** Напишите псевдокод нерекурсивной реализации алгоритма решения задачи выбора путем разбиения.
   б) Напишите псевдокод рекурсивной реализации этого алгоритма.
3. Покажите, что эффективность наихудшего случая алгоритма для решения задачи выбора путем разбиения квадратична.
4. Выведите формулу, лежащую в основе интерполяционного поиска.
5. Приведите пример входных данных для наихудшего случая интерполяционного поиска и покажите линейность данного алгоритма в наихудшем случае.
6. **а)** Найдите наименьшее значение n, для которого $\log_2 \log_2 n + 1$ превышает 6.
   б) Определите, какие из приведенных утверждений истинны:
   1) $\log \log n \in o(\log n)$ **2)** $\log \log n \in \Theta(\log n)$
   3) $\log \log n \in \Omega(\log n)$
7. **а)** Набросайте алгоритм для поиска наибольшего ключа в бинарном дереве поиска. Можно ли классифицировать ваш алгоритм как алгоритм с переменным уменьшением размера задачи?
   б) Какова временна́я эффективность вашего алгоритма в наихудшем случае?
8. **а)** Набросайте алгоритм для удаления ключа из бинарного дерева поиска. Можно ли классифицировать ваш алгоритм как алгоритм с переменным уменьшением размера задачи?
   б) Какова временна́я эффективность вашего алгоритма в наихудшем случае?

9. *Игра Ним с одной кучкой камней.* Рассмотрим следующую игру. Имеется кучка из n камней. Два игрока по очереди берут из нее от 1 до 4 камней за ход. Побеждает тот, кто возьмет последний камень. Разработайте выигрышную стратегию для игрока, делающего первый ход (если таковая существует).

10. *Переворачивающиеся блинчики.* Имеется стопка из n блинчиков разного размера. Вы можете вставить лопатку под любой блинчик и перевернуть всю стопку, оказавшуюся на лопатке, "вверх ногами". Ваша задача — при помощи последовательности таких операций разложить блинчики по размерам так, чтобы самый больший из них был в самом низу. (Визуализацию этой головоломки можно найти на узле *Interactive Mathematics Miscellany and Puzzles* [20]). Разработайте алгоритм для решения этой головоломки.

Резюме

- Метод уменьшения размера задачи для разработки алгоритмов основан на использовании соотношения между решением данного экземпляра задачи и решением меньшего экземпляра той же задачи. Если такое соотношение установлено, оно может использоваться либо сверху вниз (рекурсивно), либо снизу вверх (без рекурсии).
- Имеются три основные разновидности метода уменьшения размера задачи:
  - *уменьшение на постоянную величину*, чаще всего на единицу (например, сортировка вставкой);
  - *уменьшение на постоянный множитель*, чаще всего на два (например, бинарный поиск);
  - *переменное уменьшение размера* (например, алгоритм Евклида).
- *Сортировка вставкой* представляет собой непосредственное применение метода уменьшения размера на единицу к задаче сортировки. Этот алгоритм имеет эффективность $\Theta(n^2)$ как в среднем, так и в наихудшем случае, но в среднем он примерно вдвое быстрее, чем в наихудшем случае. Главное преимущество алгоритма — высокая производительность для почти отсортированных массивов.
- *Поиск в глубину* и *поиск в ширину* — два основных алгоритма обхода графов. Представление графов в виде леса поиска в глубину или леса поиска в ширину позволяет изучить многие важные свойства графов. Оба алгоритма имеют одинаковую эффективность: $\Theta(|V|^2)$ для пред-

ставления с использованием матрицы смежности и $\Theta(|V| + |E|)$ для представления с использованием связанных списков смежности.

- *Ориентированный граф* — это граф, ребра которого имеют ориентацию. *Задача топологической сортировки* состоит в перечислении вершин графа в таком порядке, что для каждого ребра графа его начальная вершина находится в списке до конечной вершины. Эта задача имеет решение тогда и только тогда, когда ориентированный граф является *ориентированным ациклическим графом*, т.е. не содержит ориентированных циклов.
- Имеется два алгоритма для решения задачи топологической сортировки. Первый основан на поиске в глубину; второй представляет собой непосредственное применение метода уменьшения размера задачи на единицу.
- Метод уменьшения размера задачи на единицу является естественным подходом к разработке алгоритмов для генерации элементарных комбинаторных объектов. Наиболее эффективный класс таких алгоритмов — алгоритмы, удовлетворяющие требованию минимальных изменений. Однако количество комбинаторных объектов растет настолько быстро, что даже лучшие алгоритмы представляют практический интерес только для очень маленьких экземпляров таких задач.
- Поиск фальшивой монеты на рычажных весах, *умножение по-русски* и *задача Иосифа* представляют собой примеры задач, решаемых с использованием алгоритмов уменьшения размера задачи на постоянный множитель. Два других, более важных примера таких алгоритмов — бинарный поиск и возведение в степень посредством возведения в квадрат.
- Для некоторых алгоритмов уменьшения размера задачи величина уменьшения варьируется от итерации к итерации. Примерами таких алгоритмов с *переменным уменьшением размера задачи* служат алгоритм Евклида, алгоритм решения *задачи выбора* путем разбиения, *интерполяционный поиск*, а также поиск и вставка в бинарном дереве поиска.

Глава 6

Метод преобразования

Секрет жизни ... в замене одних беспокойств другими.

— Чарльз Шульц (Charles M. Schulz) (1922–2000),
американский карикатурист

Эта глава посвящена группе методов, основанных на идее преобразования. Мы называем эту общую технологию "преобразуй и властвуй", поскольку такие методы работают в две стадии. Сначала, на стадии преобразования, экземпляр задачи преобразуется в другой, по той или иной причине легче поддающийся решению, после чего на стадии "властвования" решается полученный в результате преобразования экземпляр задачи.

Имеется три основных варианта этого метода, отличающихся способом преобразования (рис. 6.1).

- Преобразование в более простой или более удобный для решения экземпляр той же задачи — ***упрощение экземпляра***.
- ***Изменение представления*** имеющегося экземпляра задачи.
- ***Приведение задачи***, т.е. преобразование к экземпляру другой задачи, для которой имеется алгоритм решения.

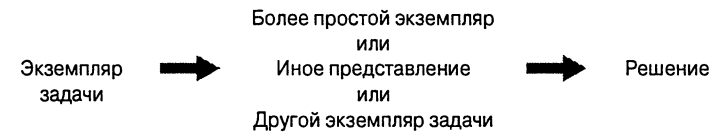


Рис. 6.1. Стратегия "преобразуй и властвуй"

В первых трех разделах данной главы мы встретимся с разными примерами упрощения экземпляра задачи. В разделе 6.1 мы познакомимся с простой, но плодотворной идеей предварительной сортировки. Многие задачи, связанные со списками, гораздо проще решить, если списки отсортированы. Естественно, преимущества от сортировки списков должны более чем компенсировать затраты времени на их сортировку; в противном случае лучше работать непосред-

ственно с несортированными списками. В разделе 6.2 мы познакомимся с одним из важных алгоритмов прикладной математики — методом исключения Гаусса. Этот алгоритм предназначен для решения системы линейных уравнений путем их предварительного преобразования к другой системе линейных уравнений, обладающей специальными свойствами, позволяющими легко находить ее решение. В разделе 6.3 идея упрощения экземпляра и изменения представления применена к деревьям поиска. В результате мы получим AVL-деревья и многопутевые сбалансированные деревья поиска; позже мы рассмотрим их простейший случай — 2-3-деревья.

В разделе 6.4 представлены пирамиды (кучи) и пирамидальная сортировка. Даже если вы уже знакомы с этой важной структурой данных и ее применением для сортировки, все равно стоит взглянуть на нее в новом свете метода преобразования. В разделе 6.5 мы обсудим схему Горнера — замечательный алгоритм для вычисления полиномов. Если бы существовал Зал славы алгоритмов, то схема Горнера была бы одним из главных претендентов быть представленным там за свою элегантность и эффективность. Мы также рассмотрим два алгоритма для решения задачи возведения в степень, которые основаны на идее изменения представления.

Завершается глава обзором ряда применений третьего варианта метода преобразования — приведения задачи. Это наиболее радикальное преобразование, так как задача преобразуется в совершенно иную задачу. Это очень мощный метод, который активно используется в теории сложности (глава 10). Однако применение этого метода для разработки практичных алгоритмов далеко не тривиально. Во-первых, мы должны определить новую задачу, в которую будет преобразована исходная. Затем мы должны убедиться, что алгоритм преобразования, за которым следует алгоритм решения новой задачи, является более эффективным методом решения с точки зрения времени работы, чем другие алгоритмические альтернативы. Среди нескольких примеров мы рассмотрим важный частный случай ***математического моделирования***, т.е. выражения задачи в терминах чисто математических объектов — таких как переменные, функции и уравнения.

6.1 Предварительная сортировка

Предварительная сортировка — старая идея в кибернетике. На самом деле интерес к алгоритмам сортировки в значительной степени обусловлен именно тем, что ряд задач с участием списков решаются существенно проще, если списки отсортированы. Понятно, что временна́я эффективность алгоритма, включающего в качестве этапа сортировку, может зависеть от эффективности использованного алгоритма сортировки. Для простоты в этом разделе мы полагаем, что все

списки реализованы в виде массивов, поскольку многие алгоритмы сортировки реализуются проще при использовании этого представления.

Мы уже рассматривали три элементарных алгоритма сортировки — сортировку выбором, пузырьковую сортировку и сортировку вставкой, — которые квадратичны как в наихудшем, так и в среднем случае, и два более эффективных алгоритма — сортировку слиянием, эффективность которой в любом случае равна $\Theta(n \log n)$, и быструю сортировку, эффективность которой в среднем случае также равна $\Theta(n \log n)$, но в худшем — квадратична. Имеются ли более быстрые алгоритмы сортировки? Как мы уже указывали в разделе 1.3 (см. также раздел 10.2), в общем случае ни один алгоритм сортировки, основанный на сравнении, не может иметь эффективность, превышающую $n \log n$ в наихудшем или в среднем случае.[1]

Далее приведены три примера использования предварительной сортировки. Дополнительные примеры можно найти в упражнениях к данному разделу.

Пример 1 (Проверка единственности элементов массива). Сама задача вам уже должна быть знакома — мы рассматривали алгоритм ее решения с применением грубой силы в разделе 2.3 (пример 2). Алгоритм на основе грубой силы для проверки того, что все элементы массива различны, попарно сравнивает все элементы этого массива, пока не будут найдены два одинаковых либо пока не будут пересмотрены все возможные пары. В наихудшем случае эффективность такого алгоритма равна $\Theta(n^2)$.

К решению задачи можно подойти и по-другому — сначала отсортировать массив, а затем сравнивать только последовательные элементы: если в массиве есть одинаковые элементы, то они должны следовать в отсортированном массиве один за другим.

```
Алгоритм PresortElementUniqueness (A[0..n − 1])
  // Проверка единственности элементов массива
  // Входные данные:  Массив A[0..n − 1] упорядочиваемых элементов
  // Выходные данные: true, если в A нет одинаковых элементов,
  //                  и false, если есть
  Сортировка массива A
  for i ← 0 to n − 2 do
    if A[i] = A[i + 1]
      return false
  return true
```

[1] Алгоритмы ***поразрядной сортировки*** (radix sort) линейны при рассмотрении зависимости от общего количества входных битов. Эти алгоритмы работают путем сравнения отдельных битов или частей ключей, а не сравнения ключей целиком. Хотя время работы таких алгоритмов пропорционально количеству входных битов, по сути они остаются алгоритмами класса $n \log n$, так как для наличия n различных входных ключей количество битов в одном ключе должно составлять как минимум $\log_2 n$.

Время работы данного алгоритма представляет собой сумму времени, затраченного на сортировку, и времени на проверку соседних элементов. Поскольку для сортировки требуется как минимум $n \log n$ сравнений, а для проверки соседних элементов — не более $n - 1$, именно сортировка и определяет общую эффективность алгоритма. Так, если мы используем здесь квадратичный алгоритм сортировки, то алгоритм в целом окажется не эффективнее метода грубой силы. Но если воспользоваться хорошим алгоритмом сортировки, таким как сортировка слиянием, эффективность которого в худшем случае составляет $\Theta(n \log n)$, то весь алгоритм проверки единственности элементов массива также будет иметь эффективность $\Theta(n \log n)$:

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n). \quad \blacksquare$$

Пример 2 (Вычисление моды). ***Модой*** (mode) называется значение, которое встречается в данном списке чаще других. Например, в случае значений 5, 1, 5, 7, 6, 5, 7 модой является значение 5 (если одинаково часто встречается несколько значений, модой может быть выбрано любое из них). Алгоритм на основе грубой силы сканирует весь список и вычисляет количество появлений в списке каждого из различных значений, после чего ищется наибольшее из найденных количеств появлений в списке. При реализации такого подхода встреченные значения и количество их появлений можно хранить в отдельном списке. При каждой итерации i-ый элемент исходного списка сравнивается со значениями уже встречавшихся элементов путем сканирования вспомогательного списка. Если значение i-го элемента имеется во вспомогательном списке, увеличивается счетчик количества элементов; если — нет, элемент добавляется во вспомогательный список, а его счетчику присваивается значение 1.

Нетрудно увидеть, что в наихудшем случае входные данные представляют собой список из неповторяющихся элементов. В таком списке его i-ый элемент сравнивается с $i - 1$ различными элементами вспомогательного списка, перед тем как быть добавленным к этому списку. В результате количество сравнений, выполняемых алгоритмом в наихудшем случае, при создании вспомогательного списка составляет

$$C(n) = \sum_{i=1}^{n} (i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)\,n}{2} \in \Theta(n^2).$$

Кроме того, для выявления элемента с наибольшим значением счетчика во вспомогательном списке требуется выполнить $n - 1$ сравнений, но они не влияют на квадратичность рассмотренного алгоритма в наихудшем случае.

Рассмотрим альтернативный вариант, начинающийся с сортировки списка. В таком случае все равные значения будут соседствовать друг с другом, и для

вычисления моды надо только найти наибольшую подпоследовательность одинаковых соседних значений в отсортированном списке.

Алгоритм $PresortMode(A[0..n-1])$
// Вычисляет моду массива с использованием
// предварительной сортировки
// **Входные данные:** Массив $A[0..n-1]$ упорядочиваемых элементов
// **Выходные данные:** Мода массива
Сортировка массива A
$i \leftarrow 0$ // Текущее сканирование начинается с позиции i
$modefrequency \leftarrow 0$ // Максимальное количество одинаковых элементов
while $i \leqslant n-1$ **do**
  $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
  while $i + runlength \leqslant n-1$ **and** $A[i+runlength] = runvalue$ **do**
    $runlength = runlength + 1$
  if $runlength > modefrequency$
    $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
  $i \leftarrow i + runlength$
return $modevalue$

Анализ этого алгоритма аналогичен анализу алгоритма из примера 1: время работы алгоритма определяется временем сортировки, поскольку время выполнения остальной части алгоритма — линейное (почему?). Следовательно, при использовании сортировки, принадлежащей классу эффективности $n \log n$, эффективность описанного алгоритма в наихудшем случае будет выше эффективности в наихудшем случае алгоритма с использованием грубой силы. ■

Пример 3 (Поиск). Рассмотрим поиск данного значения v в массиве из n упорядочиваемых элементов. Решение с использованием грубой силы — последовательный поиск (см. раздел 3.1) — требует в наихудшем случае n сравнений. Если массив предварительно отсортировать, то применение бинарного поиска приведет к $\lfloor \log_2 n \rfloor + 1$ сравнений в наихудшем случае. Даже при использовании максимально эффективной сортировки класса $n \log n$ общее время работы алгоритма в наихудшем случае составит

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

что хуже, чем в случае последовательного поиска. То же самое можно сказать и об эффективности в среднем случае. Конечно, если поиск требуется выполнять неоднократно, то затраты времени на сортировку могут оказаться оправданными (в упражнении 6.1.6 требуется оценить наименьшее количество поисков, при котором окупается предварительная сортировка). ■

Перед тем как завершить обсуждение предварительной сортировки, мы должны упомянуть, что многие (если не большинство) геометрические алгоритмы работают с множествами точек, отсортированных тем или иным образом. Точки могут быть отсортированы по одной из координат, их расстоянию от некоторой линии, некоторому углу и т.д. Например, предварительная сортировка используется в алгоритме декомпозиции для задачи пар ближайших точек и в алгоритме вычисления выпуклой оболочки, рассматривавшихся в разделе 4.6.

Упражнения 6.1

1. Вспомним, что ***медианой*** множества из n чисел называется его $\lceil n/2 \rceil$ в порядке возрастания элемент (т.е. медиана больше одной половины элементов множества и меньше другой). Разработайте алгоритм для поиска медианы с использованием предварительной сортировки и определите класс его эффективности.
2. Рассмотрим задачу поиска расстояния между двумя ближайшими числами в массиве из n чисел (расстояние между двумя числами x и y вычисляется как $|x - y|$).
   а) Разработайте алгоритм с использованием предварительной сортировки для решения данной задачи и определите его класс эффективности.
   б) Сравните эффективность разработанного вами алгоритма с эффективностью алгоритма грубой силы (см. упражнение 1.2.9).
3. Пусть $A = \{a_1, a_2, \ldots, a_n\}$ и $B = \{b_1, b_2, \ldots, b_m\}$ — два множества чисел. Рассмотрим задачу поиска их пересечения, т.е. множества C всех чисел, которые входят как в A, так и в B.
   а) Разработайте алгоритм грубой силы для решения данной задачи и определите класс его эффективности.
   б) Разработайте алгоритм с использованием предварительной сортировки для решения данной задачи и определите класс его эффективности.
4. Рассмотрим задачу поиска наибольшего и наименьшего элементов в массиве их n чисел.
   а) Разработайте алгоритм с использованием предварительной сортировки для решения данной задачи и определите класс его эффективности.
   б) Сравните эффективность трех алгоритмов: 1) алгоритма грубой силы, 2) алгоритма с использованием предварительной сортировки и 3) алгоритма декомпозиции (см. упражнение 4.1.2).

5. Покажите, что эффективность в среднем случае при однократном поиске при помощи алгоритма, состоящего из наиболее эффективного алгоритма сортировки на основе сравнений, за которым следует бинарный поиск, оказывается ниже эффективности последовательного поиска в среднем случае.

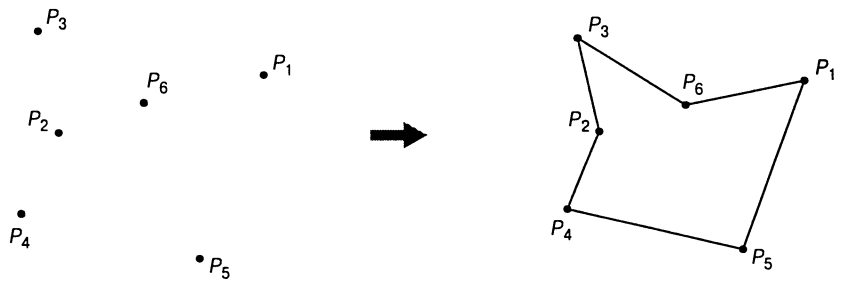
6. Оцените, какое количество поисков следует выполнить, чтобы оправдать время, затраченное на предварительную сортировку массива из 10^3 элементов, если она выполняется при помощи сортировки слиянием, а поиск — с использованием бинарного поиска (для простоты считаем, что выполняется поиск элементов, о которых заведомо известно их наличие в массиве). А в случае массива из 10^6 элементов?

7. Сортировать или не сортировать? Разработайте эффективные алгоритмы для решения следующих задач и определите их классы эффективности.

   а) У вас имеется n телефонных счетов и m чеков для их оплаты $n \geqslant m$). Считая, что номера телефонов указаны на чеках, надо найти всех должников (для простоты считаем, что для каждого счета выписано не более одного чека и что выписанный чек полностью оплачивает счет.)

   б) Имеется файл с n записями о студентах, в которых указаны номер, имя, адрес и дата рождения студента. Требуется найти количество студентов из каждого штата.

8. Дано множество из $n \geqslant 3$ точек на плоскости $x - y$. Требуется соединить их замкнутой ломаной линией без самопересечений так, чтобы получился многоугольник, например:



   а) Всегда ли поставленная задача имеет решение? Всегда ли это решение единственно?

   б) Разработайте эффективный алгоритм для решения поставленной задачи и определите его класс эффективности.

9. У вас есть массив из n чисел и целое число s. Определите, имеются ли в массиве два числа, сумма которых равна s. (Например, в случае массива 5, 9, 1, 3 и $s = 6$ ответ — "да", но если $s = 7$, ответ — "нет".) Разработайте алгоритм для решения поставленной задачи, эффективность которого превышает квадратичную.

10. **а)** Разработайте эффективный алгоритм для поиска всех множеств анаграмм в большом файле, как, например, словарь английских слов [15]. Например, *eat*, *ate* и *tea* принадлежат к одному такому множеству.

    б) Напишите программу, реализующую ваш алгоритм.

6.2 Метод исключения Гаусса

Вы наверняка знакомы с системой из двух линейных уравнений с двумя неизвестными:

$$a_{11}x + a_{12}y = b_1$$
$$a_{21}x + a_{22}y = b_2$$

Вспомним, что если только коэффициенты одного уравнения не пропорциональны коэффициентам другого, то система имеет единственное решение. Стандартный метод поиска этого решения состоит в использовании одного из уравнений для того, чтобы выразить одну переменную как функцию другой, а затем подставить результат в другое уравнение. Это дает линейное уравнение относительно одной переменной, решение которого позволяет найти значение второй переменной.

Во многих приложениях требуется решить систему из n уравнений с n неизвестными, где n — большое число:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

Теоретически такую систему уравнений можно решить, обобщив метод подстановки, примененный для решения системы из двух линейных уравнений (на каком методе разработки алгоритмов основан этот способ?), но полученный в результате алгоритм будет слишком громоздким.

К счастью, имеется гораздо более элегантный алгоритм решения систем линейных уравнений, который называется ***методом исключения Гаусса*** (Gauss eli-

mination)[2]. Идея метода заключается в преобразовании системы n линейных уравнений с n неизвестными в эквивалентную систему (т.е. систему с тем же решением, что и у исходной) с верхнетреугольной матрицей коэффициентов, т.е. такой, у которой все элементы ниже главной диагонали равны нулю:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \quad \Rightarrow \quad \begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{nn}x_n &= b'_n \end{aligned}$$

Используя матричные обозначения, можно записать это как

$$Ax = b \quad \Rightarrow \quad A'x = b'$$

где

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix} \quad b' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

(Мы добавили штрихи к элементам матрицы и свободным членам новой системы линейных уравнений для того, чтобы подчеркнуть отличие этих значений от значений их аналогов в исходной системе линейных уравнений.)

Чем же система линейных уравнений с верхнетреугольной матрицей коэффициентов лучше системы линейных уравнений с произвольной матрицей? Дело в том, что систему линейных уравнений с верхнетреугольной матрицей легко решить методом обратной подстановки следующим образом. Сначала мы вычисляем значение x_n из последнего уравнения; затем подставляем полученное значение в предпоследнее уравнение и получаем значение x_{n-1}. Продолжая выполнять подстановки вычисленных значений переменных в очередные уравнения, мы получим значения всех n переменных — от x_n до x_1.

Итак, каким же образом можно получить из системы линейных уравнений с произвольной матрицей коэффициентов A эквивалентную систему линейных уравнений с верхнетреугольной матрицей A'? Это можно сделать при помощи последовательности так называемых элементарных операций:

[2]Метод назван по имени Карла Фридриха Гаусса (Carl Friedrich Gauss) (1777–1855), который, как и другие гиганты в истории математики, например Исаак Ньютон (Isaac Newton) и Леонард Эйлер (Leonard Euler), выполнил ряд фундаментальных работ как в области теоретической, так и вычислительной математики.

- обмена двух уравнений системы линейных уравнений;
- умножения уравнения на ненулевую величину;
- замены уравнения на сумму или разность этого уравнения и другого уравнения, умноженного на некоторую величину.

Поскольку ни одна из элементарных операций не изменяет решение системы линейных уравнений, любая система линейных уравнений, полученная из исходной при помощи серии элементарных операций, будет иметь то же решение, что и исходная система линейных уравнений. Теперь посмотрим, как получить систему линейных уравнений с верхнетреугольной матрицей. Для начала используем в качестве опорного элемента a_{11} для того, чтобы сделать все коэффициенты при x_1 в строках ниже первой нулевыми. В частности, заменим второе уравнение разностью между ним и первым уравнением, умноженным на a_{21}/a_{11} для того, чтобы получить нулевой коэффициент при x_1. Выполняя то же для третьей, четвертой и далее строк и умножая первое уравнение, соответственно, на a_{31}/a_{11}, a_{41}/a_{11}, $\ldots$, a_{n1}/a_{11}, сделаем все коэффициенты при x_1 в уравнениях ниже первого равными 0. Затем обнулим все коэффициенты при x_2 в уравнениях ниже второго, вычитая из каждого из этих уравнений второе, умноженное на соответствующий коэффициент. Повторяя эти действия для каждой из первых $n-1$ строк, получим систему линейных уравнений с верхнетреугольной матрицей коэффициентов.

Перед тем как рассмотреть конкретный пример использования метода Гаусса, заметим, что можно работать только с матрицей коэффициентов, к которой в качестве $n+1$-го столбца добавлены свободные члены системы линейных уравнений. Другими словами, нет необходимости явно использовать имена переменных системы линейных уравнений или знаки $+$ и $=$.

Пример 1. (Решение системы линейных уравнений методом исключения Гаусса).

$$
\begin{aligned}
2x_1 - x_2 + x_3 &= 1\\
4x_1 + x_2 - x_3 &= 5\\
x_1 + x_2 + x_3 &= 0
\end{aligned}
$$

$$
\begin{bmatrix}
2 & -1 & 1 & 1\\
4 & 1 & -1 & 5\\
1 & 1 & 1 & 0
\end{bmatrix}
\quad
\begin{aligned}
&\text{Строка 2} - \tfrac{4}{2} \cdot \text{Строка 1}\\
&\text{Строка 3} - \tfrac{1}{2} \cdot \text{Строка 1}
\end{aligned}
$$

$$
\begin{bmatrix}
2 & -1 & 1 & 1\\
0 & 3 & -3 & 3\\
0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\quad
\text{Строка 3} - \tfrac{1}{2} \cdot \text{Строка 2}
$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Теперь при помощи обратной подстановки легко получить $x_3 = (-2)/2 = -1$, $x_2 = (3 - (-3)\,x_3)/3 = 0$ и $x_1 = (1 - x_3 - (-1)\,x_2)/2 = 1$. ■

Далее приведен псевдокод этапа исключения алгоритма решения систем линейных уравнений методом исключения Гаусса.

АЛГОРИТМ *GaussElimination* ($A\,[1..n, 1..n]\,, b\,[1..n]$)

```
// Применение метода исключения Гаусса к матрице
// коэффициентов системы линейных уравнений A, объединяемой
// со столбцом свободных членов b
// Входные данные:  Матрица A[1..n, 1..n] и вектор b[1..n]
// Выходные данные: Эквивалентная верхнетреугольная матрица
//                  на месте матрицы A со значениями
//                  в n + 1-ом столбце, соответствующим
//                  свободным членам новой системы линейных
//                  уравнений
for i ← 1 to n do
  A[i, n + 1] ← b[i]   // Расширение матрицы
for i ← 1 to n − 1 do
  for j ← i + 1 to n do
    for k ← i to n + 1 do
      A[j, k] ← A[j, k] − A[i, k] * A[j, i]/A[i, i]
```

По поводу приведенного псевдокода следует сделать два важных замечания. Во-первых, он не всегда корректен: если $A\,[i, i] = 0$, нельзя выполнить деление на этот элемент и, следовательно, использовать i-ую строку в качестве опорной на i-ой итерации алгоритма. В этом случае мы должны воспользоваться первой из элементарных операций и обменять i-ую строку с одной из строк ниже ее, у которой в i-ом столбце находится ненулевой элемент (если система линейных уравнений имеет единственное решение (что является нормальной ситуацией при рассмотрении систем линейных уравнений), то такая строка должна существовать).

Поскольку мы все равно должны быть готовы к возможному обмену строк, следует позаботиться и о другой потенциальной сложности: возможности того, что величина $A\,[i, i]$ будет столь мала (и, соответственно, столь велик коэффициент $A\,[j, i]/A\,[i, i]$), что новое значение $A\,[j, k]$ может оказаться искаженным ошибкой округления, связанной с вычитанием двух сильно отличающихся чисел.[3] Чтобы

[3] Подробнее об ошибках округления мы поговорим в разделе 10.4.

избежать этой проблемы, можно всегда выбирать строку с наибольшим абсолютным значением коэффициента в i-ом столбце для обмена с i-ой строкой, а затем использовать ее в качестве опорной на i-ой итерации. Такая модификация алгоритма, называющаяся ***выбором ведущего элемента*** (partial pivoting), гарантирует, что значение масштабирующего множителя никогда не превысит 1.

Во-вторых, заметим, что внутренний цикл написан с вопиющей неэффективностью. Можете ли вы, не обращаясь к приведенному далее псевдокоду, сказать, в чем именно заключается эта неэффективность и как ее избежать?

```
Алгоритм BetterGaussElimination (A[1..n, 1..n], b[1..n])
  // Реализует метод исключения Гаусса с выбором ведущего
  // элемента
  // Входные данные:  Матрица A[1..n, 1..n] и вектор b[1..n]
  // Выходные данные: Эквивалентная верхнетреугольная матрица
  //                  на месте матрицы A со значениями
  //                  в n + 1-ом столбце, соответствующим
  //                  свободным членам новой системы линейных
  //                  уравнений
  for i ← 1 to n do
    A[i, n + 1] ← b[i]   // Расширение матрицы
  for i ← 1 to n − 1 do
    pivotrow ← i
    for j ← i + 1 to n do
      if |A[j, i]| > |A[pivotrow, i]|
        pivotrow ← j
    for k ← i to n + 1 do
      swap(A[i, k], A[pivotrow, k])
    for j ← i + 1 to n do
      temp ← A[j, i]/A[i, i]
      for k ← i to n + 1 do
        A[j, k] ← A[j, k] − A[i, k] * temp
```

Давайте определим временну́ю эффективность этого алгоритма. Наиболее глубоко вложенный цикл состоит из одной строки

$$A[j, k] \leftarrow A[j, k] - A[i, k] * temp$$

которая содержит одну операцию умножения и одну — вычитания. На большинстве компьютеров умножение, несомненно, более дорогостоящая операция, чем сложение и вычитание, так что именно умножение рассматривается как базовая операция данного алгоритма.[4] Напомним, что в разделе 2.3 (см. также прило-

[4]Как упоминалось в разделе 2.1, на некоторых компьютерах умножение не обязательно более дорогостоящее по сравнению со сложением и вычитанием. Для данного алгоритма это не имеет

жение А) приведены стандартные формулы суммирования, которые будут очень полезны для понимания приведенных далее выкладок:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+2-i) =$$

$$= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i) =$$

$$= (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1 = \sum_{j=1}^{n-1} (j+2) j =$$

$$= \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} =$$

$$= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta\left(n^3\right).$$

Поскольку временна́я эффективность второй стадии (обратной подстановки) алгоритма исключения Гаусса равна $\Theta\left(n^2\right)$ (что требуется самостоятельно показать в упражнении 6.2.5), общее время работы алгоритма определяется доминирующим кубическим временем стадии исключения, так что алгоритм исключения Гаусса — кубический.

Теоретически метод исключения Гаусса всегда либо дает точное решение системы линейных уравнений (если она имеет единственное решение), либо выясняет, что такого решения не существует. В последнем случае система линейных уравнений может либо не иметь решения вовсе, либо иметь бесконечно много решений. На практике решение систем большого размера данным методом наталкивается на трудности, в первую очередь связанные с накоплением ошибок округления (см. раздел 10.4). Обратитесь к учебникам по численному анализу, где этот вопрос рассматривается более подробно, как для данного метода решения систем линейных уравнений, так и для других реализаций.

LU-разложение и другие приложения

Метод исключения Гаусса имеет интересный и очень полезный побочный результат, именуемый ***LU-разложением***, или ***LU-декомпозицией*** (LU-decomposition) матрицы коэффициентов. На деле современные коммерческие реализации метода исключения Гаусса основаны именно на этом разложении, а не на описанном ранее алгоритме.

значения, поскольку нас интересует количество выполнений внутреннего цикла (которое, конечно же, в данном случае совпадает с количеством выполняемых умножений и вычитаний).

Пример 2. Вернемся к примеру в начале этого раздела, когда мы применили метод исключения Гаусса к матрице

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

Рассмотрим нижнетреугольную матрицу L, образованную единицами на главной диагонали и множителями, вычисляемыми в процессе исключения Гаусса для обнуления соответствующих коэффициентов в строках матрицы:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 1/2 & 1 \end{bmatrix}$$

и верхнетреугольную матрицу U, представляющую собой результат исключения

$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}.$$

Оказывается, произведение LU этих матриц равно исходной матрице A (для рассматриваемых матриц L и U это можно проверить непосредственным умножением, но в общем случае этот факт, естественно, требует доказательства, которое мы опускаем).

Следовательно, решение системы линейных уравнений $Ax = b$ эквивалентно решению системы $LUx = b$. Решить ее можно следующим образом. Обозначим $y = Ux$, тогда $Ly = b$. Сначала решим систему $Ly = b$, что очень просто сделать, поскольку L — нижнетреугольная матрица. Затем решим систему $Ux = y$, что опять же несложно, поскольку U — верхнетреугольная матрица. Так, для системы в начале данного раздела мы сначала решаем уравнение $Ly = b$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}.$$

Ее решение —

$$y_1 = 1, \quad y_2 = 5 - 2y_1 = 3, \quad y_3 = 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2.$$

Затем надо решить уравнение $Uy = x$, т.е. матричное уравнение

$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix}.$$

Его решение —

$$x_3 = (-2)/2 = -1,$$
$$x_2 = (3 - (-3)\,x_3)/3 = 0,$$
$$x_1 = (1 - x_3 - (-1)\,x_2)/2 = 1.$$

■

Заметим, что получив LU-разложение матрицы A, мы можем решать системы линейных уравнений $Ax = b$ для разных векторов свободных членов b. Это главное преимущество метода LU-разложения по сравнению с классическим методом исключения Гаусса, описанным ранее. Заметим также, что LU-разложение не требует дополнительной памяти, поскольку ненулевую часть матрицы U мы можем хранить в верхнетреугольной части матрицы A (включая главную диагональ), а нетривиальную часть матрицы L — ниже главной диагонали A.

Вычисление обратной матрицы

Метод исключения Гаусса — очень полезный алгоритм, с помощью которого можно решить одну из наиболее важных задач прикладной математики — системы линейных уравнений. Метод исключения Гаусса может быть также применен и к некоторым другим задачам линейной алгебры, таким как вычисление ***обратной матрицы*** (matrix inverse). Обратная к матрице A размером $n \times n$ матрица также имеет размер $n \times n$ и обозначается как A^{-1}:

$$AA^{-1} = I,$$

где I — единичная матрица размером $n \times n$ (т.е. матрица, все элементы которой равны 0, за исключением элементов на главной диагонали, которые равны 1). Не каждая квадратная матрица имеет обратную, но если у данной матрицы есть обратная, то она — единственная. Матрица A, не имеющая обратной матрицы, называется сингулярной (singular). Можно доказать, что матрица сингулярна тогда и только тогда, когда одна из ее строк представляет собой линейную комбинацию (сумму умноженных на некоторые величины) других строк. Удобным способом проверить, является ли данная матрица не сингулярной, — применить к ней метод исключения Гаусса: если он даст верхнетреугольную матрицу с ненулевыми элементами на главной диагонали, матрица не сингулярна; в противном случае исходная матрица сингулярна. Сингулярные матрицы — очень специфичный частный случай, и большинство квадратных матриц имеют обратные.

Теоретически обратные матрицы очень важны, поскольку играют роль обратных величин в матричной алгебре, тем самым преодолевая отсутствие явной операции деления матриц. Например, аналогично линейному уравнению с одним неизвестным $ax = b$, решение которого $x = a^{-1}b$ (если a не равно 0), мы можем записать решение системы n линейных уравнений с n неизвестными $Ax = b$ как $x = A^{-1}b$ (если A не сингулярная матрица), где b, разумеется, — не число, а вектор.

В соответствии с определением обратной матрицы, для того, чтобы найти ее для не сингулярной матрицы A размером $n \times n$, требуется найти n^2 чисел x_{ij}, $1 \leqslant i, j \leqslant n$ таких, что

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & 1 \end{bmatrix}.$$

Мы можем найти неизвестные числа, решая n систем линейных уравнений с одной и той же матрицей коэффициентов A, у которых вектор неизвестных x^j представляет собой j-ый столбец обратной матрицы, а вектор свободных членов e^j — j-ый столбец единичной матрицы ($1 \leqslant j \leqslant n$):

$$Ax^j = e^j.$$

Эти системы линейных уравнений можно решить, применяя метод исключения Гаусса к матрице A, расширенной добавлением к ней единичной матрицы размером $n \times n$. Еще лучший способ — использовать метод исключения Гаусса для поиска LU-разложения матрицы A и решать системы линейных уравнений $LUx^j = e^j$, $j = 1, 2, \dots, n$, как описывалось ранее.

Вычисление определителя

Еще одна задача, которая может быть решена при помощи метода исключения Гаусса, — это вычисление ***определителя***, или ***детерминанта*** (determinant) матрицы. Определителем матрицы A размером $n \times n$, обозначаемым $\det A$ или $|A|$, является число, которое можно рекурсивно определить следующим образом. Если $n = 1$, т.е. A состоит из единственного элемента a_{11}, то $\det A = a_{11}$. Если $n > 1$, то $\det A$ вычисляется по рекурсивной формуле

$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j,$$

где s_j равно $+1$, если j нечетно, и -1, если j четно (т.е. $s_j = (-1)^{j+1}$), a_{1j} — элемент на пересечении первой строки и j-го столбца матрицы, а A_j — матрица

размером $(n-1)\times(n-1)$, полученная из матрицы A удалением первой строки и j-го столбца.

В частности, для матрицы размером 2×2 из определения вытекает следующая легко запоминаемая формула:

$$\det\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11}\det[a_{22}] - a_{12}\det[a_{21}] = a_{11}a_{22} - a_{12}a_{21}.$$

Другими словами, определитель матрицы размером 2×2 равен разности произведений ее диагональных элементов.

В случае матрицы размером 3×3 получаем

$$\det\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}\det\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12}\det\begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13}\det\begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} =$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{31}a_{22}a_{13} -$$
$$- a_{21}a_{12}a_{33} - a_{32}a_{23}a_{11}.$$

Кстати, эта формула очень удобна для применения в ряде приложений; в частности, мы уже воспользовались ею в разделе 4.6 в алгоритме быстрой оболочки.

Но что если требуется вычислить определитель большой матрицы? (Хотя на практике это требуется не так часто, тем не менее эта задача стоит того, чтобы ее рассмотреть.) Рекурсивное определение мало чем может помочь, поскольку оно приводит к вычислению суммы $n!$ членов. И здесь опять нас спасает метод исключения Гаусса. Основная идея заключается в том, что определитель верхнетреугольной матрицы равен произведению ее элементов на главной диагонали, и легко понять, как именно влияют на значение определителя элементарные операции, выполняемые алгоритмом (они либо оставляют его значение неизменным, либо меняют знак, либо приводят к умножению его на константу, использующуюся в алгоритме исключения Гаусса). В результате мы можем вычислить определитель матрицы размером $n\times n$ за кубическое время.

Определители играют важную роль в теории систем линейных уравнений. В частности, система из n линейных уравнений с n неизвестными $Ax = b$ имеет единственное решение тогда и только тогда, когда определитель матрицы коэффициентов $\det A$ не равен 0. Более того, решение можно найти по формуле, носящей название правила Крамера (Cramer's rule):

$$x_1 = \frac{\det A_1}{\det A}, \ldots, x_j = \frac{\det A_j}{\det A}, \ldots, x_n = \frac{\det A_n}{\det A},$$

где $\det A_j$ — определитель матрицы, полученной путем замены j-го столбца матрицы A столбцом b. (В упражнении 6.2.10 требуется выяснить, насколько хорошим алгоритмом для решения систем линейных уравнений является правило Крамера.)

Упражнения 6.2

1. Решите методом исключения Гаусса следующую систему линейных уравнений:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

2. **а)** Решите систему линейных уравнений из упражнения 1 методом LU-разложения.
   б) Как можно классифицировать метод LU-разложения с точки зрения методов разработки алгоритмов?
3. Решите систему линейных уравнений из упражнения 1 путем обращения матрицы коэффициентов с последующим умножением на вектор свободных членов.
4. Насколько корректен следующий вывод класса эффективности стадии исключения метода исключения Гаусса?

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} (n+2-i)(n-i) = \\ &= \sum_{i=1}^{n-1} \left((n+2)n - i(2n+2) + i^2 \right) = \\ &= \sum_{i=1}^{n-1} (n+2)n - \sum_{i=1}^{n-1} (2n+2)i + \sum_{i=1}^{n-1} i^2. \end{aligned}$$

   Поскольку $s_1(n) = \sum_{i=1}^{n-1} (n+2)n \in \Theta\left(n^3\right)$, $s_2(n) = \sum_{i=1}^{n-1} (2n+2)i \in \Theta\left(n^3\right)$ и $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta\left(n^3\right)$, то $s_1(n) - s_2(n) + s_3(n) \in \Theta\left(n^3\right)$.
5. Напишите псевдокод стадии обратной подстановки метода исключения Гаусса и покажите, что время его работы равно $\Theta\left(n^2\right)$.
6. Предположим, деление двух чисел выполняется в три раза дольше их умножения. Оцените, насколько в таком случае алгоритм *BetterGaussElimination* работает быстрее алгоритма *GaussElimination* (естественно, мы считаем, что компилятор не слишком интеллектуален и не устраняет неэффективность в *GaussElimination* самостоятельно).

7. **а)** Приведите пример системы двух линейных уравнений с двумя неизвестными, которая имеет единственное решение, и решите ее методом исключения Гаусса.
   б) Приведите пример системы двух линейных уравнений с двумя неизвестными, которая не имеет решения, и примените к ней метод исключения Гаусса.
   в) Приведите пример системы двух линейных уравнений с двумя неизвестными, которая имеет бесконечное количество решений, и примените к ней метод исключения Гаусса.
8. ***Метод исключения Гаусса–Джордана*** (Gauss–Jordan elimination) отличается от метода исключения Гаусса тем, что все элементы над главной диагональю делаются нулевыми в то же время и с использованием той же опорной строки, что и элементы под главной диагональю.
   а) Примените метод исключения Гаусса–Джордана к системе линейных уравнений из упражнения 1.
   б) На какой общей стратегии разработки основан этот алгоритм?
   в) Сколько умножений в общем случае выполняет данный алгоритм при решении системы n линейных уравнений с n неизвестными? Сравните это количество с числом умножений, выполняющихся при использовании метода исключения Гаусса (как на стадии исключения, так и на стадии обратной подстановки).
9. Система n линейных уравнений с n неизвестными $Ax = b$ имеет единственное решение тогда и только тогда, когда $\det A \neq 0$. Имеет ли смысл проверять выполнимость этого условия перед применением метода исключения Гаусса?
10. **а)** Примените правило Крамера для решения системы линейных уравнений из упражнения 1.
    б) Оцените, во сколько раз дольше решается система из n линейных уравнений с n неизвестными по правилу Крамера по сравнению с методом исключения Гаусса (считаем, что все определители в формуле Крамера вычисляются независимо с применением метода исключения Гаусса).

6.3 Сбалансированные деревья поиска

В разделах 1.4 и 4.4 мы рассматривали бинарные деревья поиска — одну из важнейших структур данных для реализации словарей. Бинарное дерево поиска — это бинарное дерево, узлы которого содержат элементы множества упорядочиваемых элементов, по одному элементу в узле, причем все элементы в левом поддере-

ве меньше элемента в корне поддерева, а элементы в правом поддереве — больше него. Отметим, что такое преобразование множества в бинарное дерево поиска представляет собой пример метода изменения представления. Чего мы достигаем таким преобразованием по сравнению с простой реализацией словаря, например, при помощи массива? Мы получаем более высокую эффективность поиска, вставки и удаления — время выполнения всех этих операций равно $\Theta(\log n)$, но только в среднем случае. В наихудшем случае эти операции выполняются за время $\Theta(n)$, поскольку дерево может выродиться в полностью несбалансированное, с высотой, равной $n-1$.

Ученые в области кибернетики затратили массу усилий в попытках найти структуру, которая сохраняет важные свойства классических бинарных деревьев поиска, — в первую очередь логарифмическую эффективность словарных операций и отсортированность элементов, — но при этом избегает вырожденности в наихудшем случае. Для этого используются два подхода.

- Первый подход представляет собой вариант упрощения экземпляра задачи — несбалансированное бинарное дерево поиска преобразуется в сбалансированное. Конкретные реализации этой идеи различаются по их определениям того, что такое сбалансированность. ***AVL-дерево*** (AVL tree) требует, чтобы разница высот левого и правого поддеревьев каждого узла не превышала 1. ***Красно-черное дерево*** (red-black tree) допускает, чтобы высота одного поддерева была в два раза больше высоты другого поддерева того же самого узла. Если вставка нового узла или удаление имеющегося приводит к тому, что нарушается условие сбалансированности, такое дерево перестраивается при помощи одного из семейства специальных преобразований, которые называются ***поворотами*** (rotation) и которые восстанавливают условия сбалансированности. (В этом разделе мы рассмотрим только AVL-деревья. Информацию о других типах бинарных деревьев поиска, которые используют идею балансировки посредством поворотов, включая красно-черные деревья и так называемые ***косые деревья*** (splay tree), можно найти в соответствующей литературе.)
- Второй подход представляет собой вариант изменения представления: допускается наличие более чем одного элемента в узле дерева поиска. Частными случаями таких деревьев являются ***2-3-деревья***, ***2-3-4-деревья*** и более общий и важный случай — ***B-деревья***. Они различаются количеством элементов, которые допустимы в одном узле дерева поиска, но все они являются идеально сбалансированными. (Здесь мы рассмотрим простейший вид таких деревьев, а именно 2-3-деревья, отложив рассмотрение B-деревьев до главы 7.)

AVL-деревья

AVL-деревья были открыты в 1962 г. двумя советскими математиками — Г.М. Адельсон-Вельским и Е.М. Ландисом [1]; изобретенная структура получила название по первым буквам их фамилий.

Определение 1. ***AVL-дерево*** представляет собой бинарное дерево поиска, в котором ***показатель сбалансированности*** (balance factor) каждого узла, определяемый как разность высот левого и правого поддеревьев узла, равен 0, +1 или −1 (высота пустого дерева считается равной −1). ■

Например, бинарное дерево поиска на рис. 6.2*а* — AVL-дерево, в то время как бинарное дерево поиска на рис. 6.2*б* таковым не является.

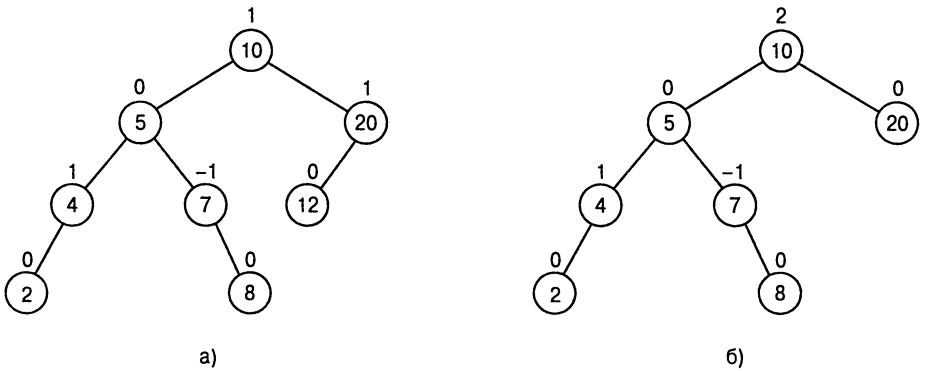


Рис. 6.2. а) AVL-дерево. б) Бинарное дерево поиска, не являющееся AVL-деревом. Показатели сбалансированности показаны над узлами деревьев

Если вставка нового узла делает AVL-дерево несбалансированным, оно преобразуется при помощи поворота. ***Поворот*** в AVL-дереве представляет собой локальное преобразование поддерева, корень которого имеет показатель сбалансированности, равный +2 или −2; если таких узлов несколько, мы поворачиваем дерево с несбалансированным корнем, который наиболее близок к вновь вставленному листу. Всего имеется только четыре типа поворотов, причем два из них представляют собой зеркальное отражение двух других. В простейшей форме четыре возможных поворота показаны на рис. 6.3.

Первый тип поворота — ***одиночный правый поворот*** (single right rotation), или ***R-поворот*** (*R*-rotation) (представьте поворот ребра, связывающего корень и его левый дочерний узел в бинарном дереве на рис. 6.3*а*, вправо). На рис. 6.4 одиночный *R*-поворот показан в наиболее общем виде. Обратите внимание, что такой поворот выполняется после вставки нового ключа в левое поддерево левого дочернего узла корня, который перед вставкой имел показатель сбалансированности +1.

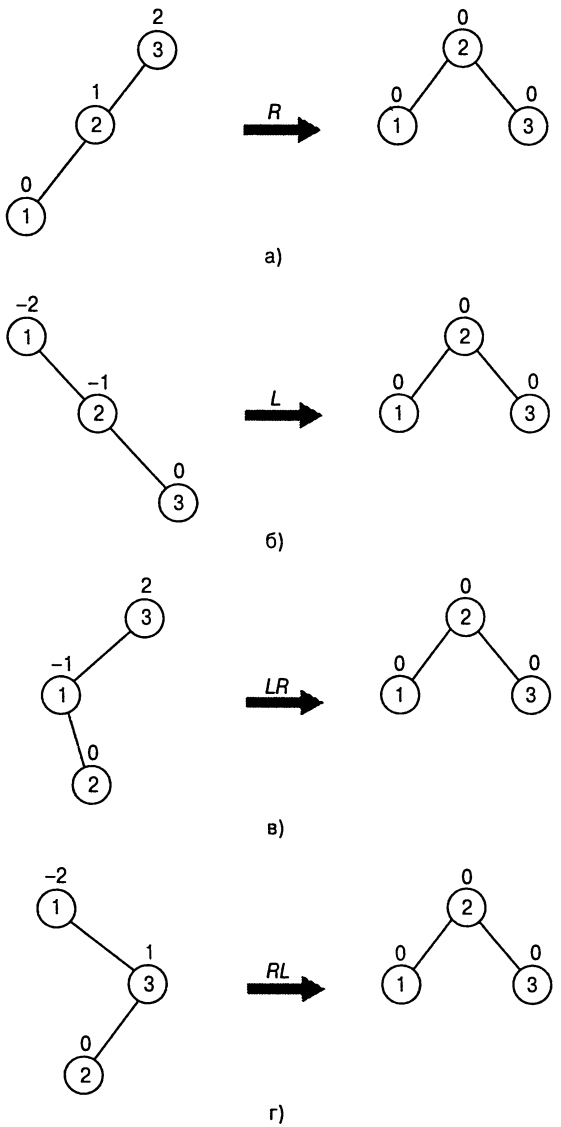


Рис. 6.3. Четыре типа поворотов AVL-деревьев с тремя узлами. а) Одиночный R-поворот. б) Одиночный L-поворот. в) Двойной LR-поворот. г) Двойной RL-поворот

Симметричный ему ***одиночный левый поворот*** (single left rotation), или ***L-поворот*** (L-rotation), представляет собой зеркальное отражение одиночного R-поворота. Он выполняется после вставки нового ключа в правое поддерево правого дочернего узла корня, который перед вставкой имел показатель сбаланси-

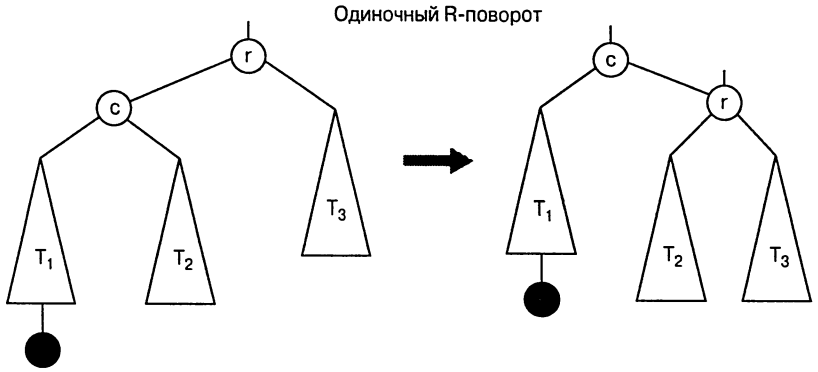


Рис. 6.4. Общий вид R-поворота в AVL-дереве. Последний вставленный узел выделен штриховкой

рованности -1 (в упражнениях к данному разделу имеется задание изобразить диаграмму одиночного L-поворота в общем виде).

Второй тип поворота — ***двойной лево-правый поворот*** (double left-right rotation), или ***LR-поворот*** (LR-rotation). Он представляет собой объединение двух поворотов: выполняется L-поворот левого поддерева корня r, за которым следует R-поворот нового поддерева, корнем которого является r (рис. 6.5). Он выполняется после вставки нового ключа в правое поддерево левого дочернего узла дерева, корень которого перед вставкой имеет показатель сбалансированности $+1$.

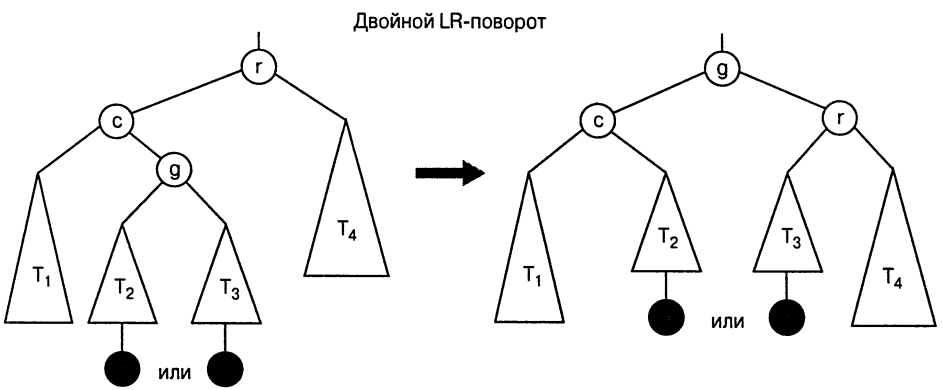


Рис. 6.5. Общий вид двойного LR-поворота в AVL-дереве. Последний вставленный узел выделен штриховкой. Он может быть либо в левом, либо в правом поддереве "внука" корня

Двойной право-левый поворот (double right-left rotation), или ***RL-поворот*** (RL-rotation), представляет собой зеркальное отражение двойного LR-поворота и оставлен читателю в качестве самостоятельного упражнения.

Заметим, что повороты не являются тривиальными преобразованиями, хотя, к счастью, могут быть выполнены за постоянное время. Они должны не только гарантировать сбалансированность получающихся в результате поворотов деревьев, но и сохранять базовые требования к бинарным деревьям поиска. Например, в исходном дереве на рис. 6.4 все ключи поддерева T_1 меньше c, который меньше всех ключей поддерева T_2, которые, в свою очередь, меньше r, а тот — меньше всех ключей в поддереве T_3. То же отношение значений ключей сохраняется, как и требуется, и в сбалансированном дереве после выполнения поворота.

В качестве примера на рис. 6.6 показано построение AVL-дерева для заданного списка чисел. При отслеживании операций, выполняемых алгоритмом, не забывайте о том, что, если имеется несколько узлов с показателем баланса ± 2, поворот выполняется для дерева, корнем которого является ближайший к вновь вставленному листу несбалансированный узел.

Насколько эффективны AVL-деревья? Как и для любого дерева поиска, критической характеристикой является его высота. Можно вывести, что высота AVL-дерева и сверху, и снизу ограничена логарифмической функцией. В частности, высота h любого AVL-дерева с n узлами удовлетворяет неравенствам

$$\lfloor \log_2 n \rfloor \leqslant h < 1.4405 \log_2 (n+2) - 1.3277.$$

(Использованные в приведенной формуле константы — округления иррациональных значений, связанных с числами Фибоначчи и золотым сечением — см. раздел 2.5.)

Непосредственно из приведенных неравенств следует, что операции поиска и вставки в худшем случае выполняются за время $\Theta(\log n)$. Получить точную формулу для средней высоты AVL-дерева, построенного для случайного набора ключей, достаточно сложно, но из многочисленных экспериментов известно, что средняя высота AVL-дерева составляет примерно $1.011 \log_2 n + 0.1$, за исключением малых значений n [67]. Таким образом, поиск в AVL-дереве требует в среднем того же количества сравнений, что и поиск в отсортированном массиве при бинарном поиске. Операция удаления ключа из AVL-дерева значительно сложнее, чем вставка, но, к счастью, она принадлежит к тому же классу эффективности, что и вставка, — т.е. к логарифмическому.

Однако эти впечатляющие характеристики эффективности достаются не задаром. Недостатками AVL-деревьев являются частые повороты, необходимость поддержания сбалансированности узлов дерева и сложность, в особенности операции удаления. Эти недостатки не позволили стать AVL-деревьям стандартной структурой данных для реализации словарей. В то же время лежащая в их основе идея о балансировке бинарного дерева поиска при помощи поворотов оказалась очень плодотворной и привела к открытию других интересных вариаций классических бинарных деревьев поиска.

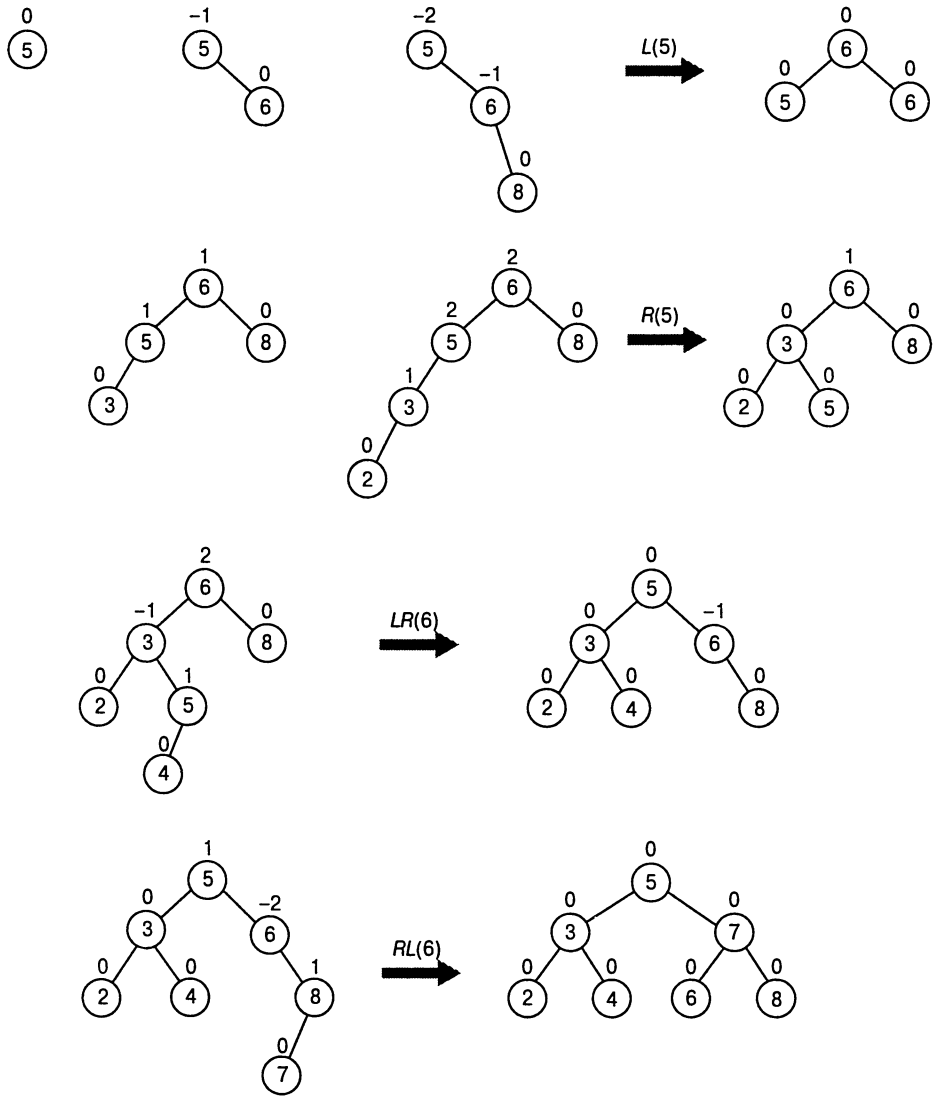


Рис. 6.6. Построение AVL-дерева путем последовательной вставки чисел из списка 5, 6, 8, 3, 2, 4, 7. Число в скобках после указания типа поворота — корень перестраиваемого дерева

2-3-деревья

Как мы упоминали в начале этого раздела, вторая идея балансировки деревьев поиска заключается в том, чтобы позволить узлу одновременно содержать несколько ключей. Простейшей реализацией этой идеи являются 2-3-деревья, разработанные в 1970 г. американским кибернетиком Дж. Хопкрофтом (John Hop-

croft) [4]. ***2-3-дерево*** представляет собой дерево, которое может иметь узлы двух видов — 2-узлы и 3-узлы. ***2-узел*** содержит единственный ключ K и имеет два потомка: левый дочерний узел служит корнем поддерева, все ключи в котором меньше K, а правый — корнем поддерева, все ключи в котором больше K. (Другими словами, 2-узел точно такой же, как и узел в классическом бинарном дереве поиска.) ***3-узел*** содержит два упорядоченных ключа K_1 и K_2 ($K_1 < K_2$) и имеет три дочерних узла. Левый дочерний узел служит корнем поддерева, ключи в котором меньше K_1, средний — корнем поддерева, ключи в котором больше K_1 и меньше K_2, а правый — корнем поддерева, все ключи в котором больше K_2 (рис. 6.7).

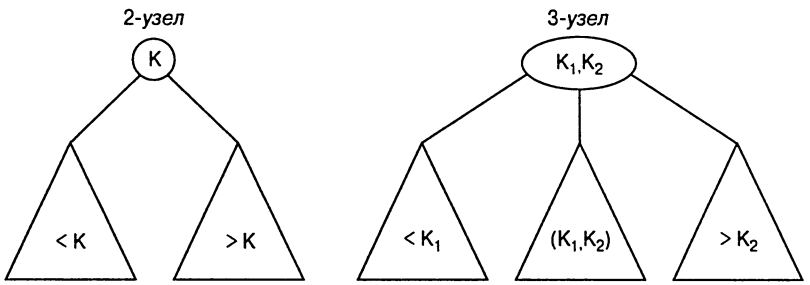


Рис. 6.7. Два вида узлов в 2-3-дереве

Последнее требование к 2-3-дереву заключается в том, что все его листья должны находиться на одном уровне, т.е. 2-3-дерево всегда ***сбалансировано по высоте*** (height-balanced): длина пути от корня дерева к листу должна быть одинакова для всех листьев дерева. Это свойство достигается ценой разрешения иметь узлы с тремя дочерними узлами.

Поиск заданного ключа K в 2-3-дереве достаточно прост. Он начинается с корня. Если корень представляет собой 2-узел, то мы действуем так же, как и в случае бинарного дерева поиска, — либо прекращаем поиск, если значение K равно значению ключа корня, либо продолжаем поиск в левом или правом поддереве, в зависимости от того, меньше ли значение K, чем ключ корня, или больше. Если же корень представляет собой 3-узел, то после не более чем двух сравнений мы знаем, следует ли прекратить поиск (если K равно одному из ключей 3-узла) или в каком из трех поддеревьев он должен быть продолжен.

Вставка нового ключа в 2-3-дерево выполняется следующим образом. Новый ключ K всегда вставляется в лист, за исключением случая пустого дерева. Соответствующий лист мы находим, выполняя поиск ключа K. Если искомый лист — 2-узел, мы вставляем K либо как первый, либо как второй ключ — в зависимости от того, меньше ли K, чем старый ключ, или больше. Если же искомый лист — 3-узел, то мы разделяем его на два: наименьший из трех ключей (двух старых и нового) помещается в первый лист, наибольший — во второй лист, а средний ключ

переносится в узел, родительский по отношению к старому листу (если лист — корень дерева, то для вставки нового ключа создается новый корень). Заметим, что перемещение среднего ключа в родительский узел может привести к переполнению родительского узла (если он был 3-узлом) и, следовательно, вызвать ряд разделений узлов вдоль цепочки предков листа.

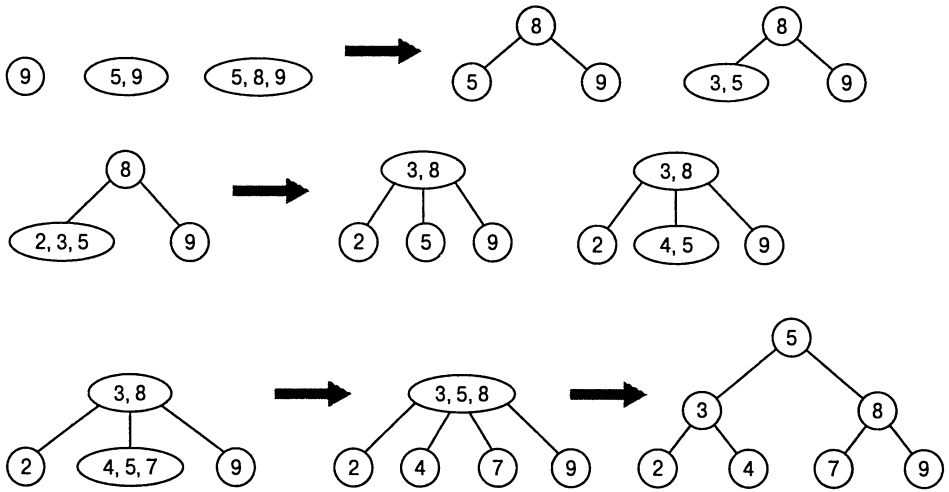


Рис. 6.8. Построение 2-3-дерева путем внесения ключей 9, 5, 8, 3, 2, 4, 7

Пример 2-3-дерева приведен на рис. 6.8.

Как и в любом дереве поиска эффективность словарных операций зависит от его высоты. Давайте сперва найдем ее верхнюю границу. 2-3-дерево высотой h с минимальным количеством ключей представляет собой полное дерево с 2-узлами (такое, как последнее дерево на рис. 6.8 для $h = 2$). Следовательно, для любого 2-3-дерева высотой h с n узлами мы получаем неравенство

$$n \geqslant 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$

следовательно,

$$h \leqslant \log_2 (n + 1) - 1.$$

С другой стороны, 2-3-дерево высотой h с максимальным количеством узлов представляет собой полное дерево, все узлы которого — 3-узлы, с двумя ключами и тремя потомками. Следовательно, для любого 2-3-дерева с n узлами

$$n \leqslant 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2\left(1 + 3 + \cdots + 3^h\right) = 3^{h+1} - 1,$$

следовательно,

$$h \geqslant \log_3 (n + 1) - 1.$$

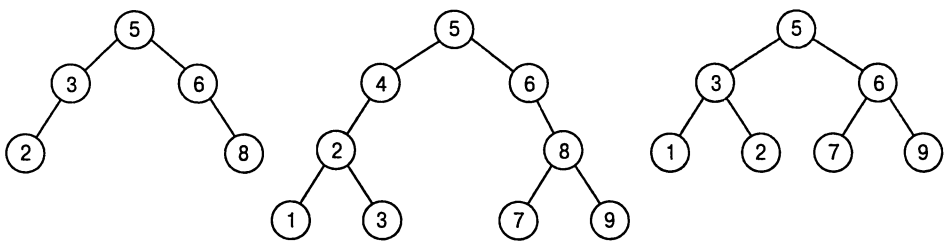
Из полученных таким образом верхней и нижней границ высоты h

$$\log_3 (n+1) - 1 \leqslant h \leqslant \log_2 (n+1) - 1$$

вытекает, что временна́я эффективность поиска, вставки и удаления как в наихудшем, так и в среднем случае — $\Theta(\log n)$. В разделе 7.4 мы рассмотрим очень важное обобщение 2-3-деревьев — B-деревья.

Упражнения 6.3

1. Какие из представленных на рисунке деревьев являются AVL-деревьями?



а) б) в)

2. **а)** Нарисуйте все бинарные деревья с n узлами (где $n = 1, 2, 3, 4, 5$), которые удовлетворяют требованию сбалансированности AVL-деревьев.

   б) Изобразите бинарное дерево высотой 4, которое представляет собой AVL-дерево и содержит наименьшее количество узлов среди всех таких деревьев.

3. Изобразите диаграмму одиночного L-поворота и двойного RL-поворота в общем виде.

4. Для каждого из представленных наборов чисел постройте AVL-дерево путем последовательного внесения чисел в изначально пустое дерево.

   а) 1, 2, 3, 4, 5, 6

   б) 6, 5, 4, 3, 2, 1

   в) 3, 6, 5, 1, 2, 4

5. **а)** Разработайте алгоритм вычисления диапазона AVL-дерева, содержащего действительные числа (т.е. разность между наибольшим и наименьшим числами в дереве) и определите его эффективность в наихудшем случае.

б) Истинно или ложно следующее утверждение: наименьший и наибольший ключи в AVL-дереве всегда находятся на последнем или предпоследнем уровне?

6. Напишите программу для построения AVL-дерева для списка из n различных целых чисел.

7. а) Постройте 2-3-дерево для следующего множества символов: C, O, M, P, U, T, I, N, G (используйте при этом алфавитное упорядочение букв и их последовательную вставку в изначально пустое дерево).

б) Найдите наибольшее и среднее количество сравнений ключей при успешном поиске в получившемся дереве в предположении равновероятного поиска ключей.

8. Пусть T_B и T_{2-3} представляют собой, соответственно, классическое бинарное дерево поиска и 2-3-дерево, построенные для одного и того же набора ключей, вставляемых в деревья в одном и том же порядке. Истинно или ложно следующее утверждение: поиск одного и того же ключа в T_{2-3} всегда требует меньшего или такого же количества сравнений ключей, что и поиск в T_B?

9. Разработайте для 2-3-дерева, содержащего действительные числа, алгоритм для вычисления диапазона (т.е. разности между наибольшим и наименьшим числом) дерева и определите его эффективность в наихудшем случае.

10. Напишите программу для построения 2-3-дерева для данного списка n различных целых чисел.

6.4 Пирамиды и пирамидальная сортировка

Структура данных под названием ***пирамида*** (heap) представляет собой хитроумную частично упорядоченную структуру данных, которая в особенности хорошо подходит для реализации очередей с приоритетами. Вспомним, что ***очередь с приоритетами*** (priority queue) представляет собой множество элементов с упорядочиваемой характеристикой, называющейся ***приоритетом*** (priority) элемента, и обеспечивающее выполнение следующих операций:

- поиск элемента с наивысшим (т.е. с наибольшим) приоритетом;
- удаление элемента с наибольшим приоритетом;
- добавление нового элемента в множество.

Пирамиды представляют особый интерес в первую очередь благодаря эффективной реализации перечисленных операций. Пирамида также является структурой данных, которая служит краеугольным камнем теоретически важного алгорит-

ма — пирамидальной сортировки (heapsort). Мы рассмотрим этот алгоритм позже, после того как дадим определение пирамиды и изучим ее основные свойства.

Понятие пирамиды

Определение 1. ***Пирамида*** (heap) может быть определена как бинарное дерево с ключами, назначенными ее узлам (по одному ключу на узел), для которого выполняются два следующих условия.

1. *Требование к форме дерева.* Бинарное дерево ***практически полное*** (essentially complete) или просто ***полное*** (complete), т.е. все его уровни заполнены, за исключением, возможно, последнего уровня, в котором могут отсутствовать некоторые крайние справа листья.
2. *Требование доминирования родительских узлов.* Ключ в каждом узле не меньше ключей в его дочерних узлах (условие считается автоматически выполняющимся для всех листьев).[5] ■

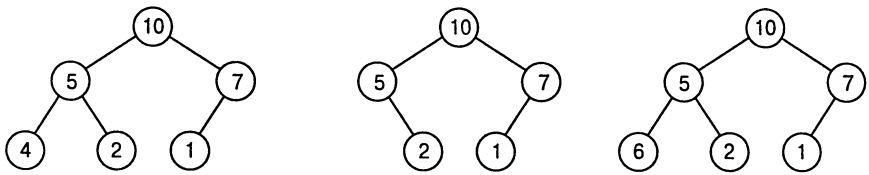


Рис. 6.9. Иллюстрация к определению пирамиды: таковой является только крайнее слева дерево

Рассмотрим, например, деревья на рис. 6.9. Первое дерево является пирамидой. Второе дерево — не пирамида, поскольку нарушено требование к форме дерева. Третье дерево также не является пирамидой, поскольку в нем нарушено требование доминирования родительских узлов для узла с ключом 5.

Обратите внимание на упорядоченность значений в пирамиде сверху вниз — т.е. последовательность значений на любом пути от корня к листу убывающая (невозрастающая, если допускается наличие одинаковых ключей). Однако упорядоченности ключей слева направо нет, т.е. нет никаких соотношений между значениями ключей в узлах на одном уровне дерева или, в общем случае, в левом и правом поддеревьях одного узла.

Вот список важных свойств пирамид, которые несложно доказать (в качестве примера проверьте их выполнение для пирамиды, показанной на рис. 6.10.

1. Имеется ровно одно практически полное бинарное дерево с n узлами. Его высота равна $\lfloor \log_2 n \rfloor$.
2. Корень пирамиды всегда содержит ее наибольший элемент.

[5] Некоторые авторы требуют, чтобы ключ в каждом узле *не превышал* ключи в дочерних узлах.

Рис. 6.10. Пирамида и ее представление в виде массива

3. Любой узел пирамиды со всеми его потомками также является пирамидой.
4. Пирамида может быть реализована в виде массива путем записи ее элементов сверху вниз слева направо. Удобно хранить элементы пирамиды в позициях такого массива с 1 по n, оставляя $H[0]$ либо неиспользуемым, либо размещая в нем ограничитель, значение которого превышает значение любого элемента пирамиды. При использовании такого представления
   а) ключи родительских узлов занимают первые $\lfloor n/2 \rfloor$ позиций в массиве, а ключи листьев — последние $\lceil n/2 \rceil$ позиций.
   б) дочерние ключи по отношению к родительскому в позиции i ($1 \leqslant i \leqslant \lfloor n/2 \rfloor$) находятся в позициях $2i$ и $2i+1$, соответственно; родительский ключ для ключа в позиции i ($2 \leqslant i \leqslant n$) находится в позиции $\lfloor i/2 \rfloor$.

Таким образом, мы можем определить пирамиду как массив $H[1..n]$, в котором каждый элемент в позиции i в первой половине массива больше или равен элементам в позициях $2i$ и $2i+1$, т.е.

$$H[i] \geqslant \max\{H[2i], H[2i+1]\} \quad \text{для } i = 1, 2, \ldots, \lfloor n/2 \rfloor .$$

(Конечно, если $2i+1 > n$, то выполняться должно только неравенство $H[i] \geqslant H[2i]$.) В то время как идеи, лежащие в основе алгоритмов с использованием пирамид, проще для понимания при представлении пирамид в виде бинарных деревьев, реальные реализации эти алгоритмов обычно существенно проще и эффективнее при использовании представления в виде массива.

Как же построить пирамиду для заданного множества ключей? Имеется два основных метода выполнить эту работу. Первый называется ***восходящим построением пирамиды*** (bottom-up heap construction) и проиллюстрировано на рис. 6.11. При этом практически полное бинарное дерево инициализируется путем размещения n ключей в заданном порядке, а затем дерево "пирамидизируется" следующим образом. Начиная с последнего родительского узла и заканчивая корнем алгоритм проверяет, выполняется ли для рассматриваемого узла требование доминирования родительского узла. Если нет, то алгоритм обменивает ключ узла

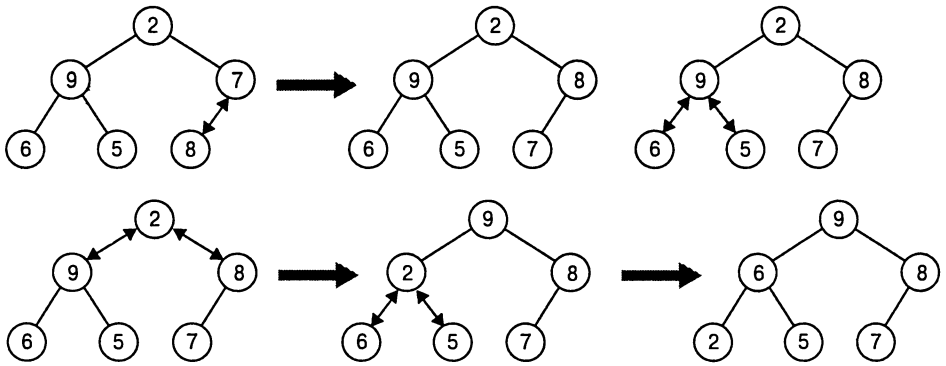


Рис. 6.11. Восходящее построение пирамиды для множества ключей 2, 9, 7, 6, 5, 8

K с наибольшим ключом среди его дочерних узлов и проверяет выполнение требования доминирования родительского узла для ключа K в новой позиции. Этот процесс продолжается до тех пор, пока для ключа K не будет выполнено требование доминирования родительского узла (в конечном итоге это требование будет выполнено, так как оно всегда выполняется для ключей в листьях). После завершения "пирамидизации" поддерева, корнем которого является данный родительский узел, алгоритм выполняет те же действия с непосредственным предком этого узла. Алгоритм завершает свою работу после обработки корня дерева.

Перед тем как будет приведен псевдокод восходящего построения пирамиду, следует сделать одно замечание. Поскольку значение ключа не изменяется при его перемещении вниз по дереву, нет необходимости выполнять промежуточные обмены. Такое улучшение алгоритма можно представить как обмен пустого узла с бóльшими ключами среди потомков (или как перемещение пустого узла вниз по дереву — *прим. перев.*) до достижения конечной позиции, куда и вставляется ранее сохраненный ключ.

АЛГОРИТМ $HeapBottomUp(H[1..n])$
// Построение пирамиды из элементов заданного массива при
// помощи восходящего алгоритма
// **Входные данные:** Массив $H[1..n]$ упорядочиваемых элементов
// **Выходные данные:** Пирамида $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
  $k \leftarrow i$; $v \leftarrow H[k]$
  $heap =$ **false**
  while not $heap$ **and** $2 * k \leqslant n$ **do**
    $j \leftarrow 2 * k$
    if $j < n$ // Имеется два дочерних узла
      if $H[j] < H[j+1]$

$j \leftarrow j + 1$
if $v \geqslant H[j]$
$heap \leftarrow$ **true**
else
$H[k] \leftarrow H[j]$; $k \leftarrow j$
$H[k] \leftarrow v$

Насколько эффективен данный алгоритм в наихудшем случае? Предположим для простоты, что $n = 2^k - 1$, так что дерево пирамиды заполнено, т.е. на каждом уровне находится максимально возможное количество узлов. Пусть h — высота пирамиды; согласно первому свойству пирамид из списка в начале этого раздела, $h = \lfloor \log_2 n \rfloor$ (или просто $\lceil \log_2 (n+1) \rceil - 1 = k - 1$ для рассматриваемого нами значения n). В наихудшем случае при выполнении алгоритма построения пирамиды каждый ключ на уровне i дерева будет перемещаться до уровня листа h. Поскольку перемещение на один уровень вниз требует двух сравнений (одного для поиска дочернего узла с наибольшим ключом, и второго для выяснения, требуется ли обмен ключами), общее количество сравнений ключей, выполняемых при перемещении ключа на уровне i, равно $2(h-i)$. Таким образом, общее количество сравнений ключей в наихудшем случае составляет

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\substack{\text{Ключи на} \\ \text{уровне } i}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)\,2^i = 2(n - \log_2(n+1)),$$

где корректность последнего равенства может быть доказана либо с использованием формулы для суммы $\sum_{i=1}^{h} i2^i$ (см. приложение А), либо при помощи математической индукции по h. Таким образом, при использовании восходящего алгоритма пирамида размером n может быть построена с выполнением менее чем $2n$ сравнений.

Альтернативный (и менее эффективный) алгоритм строит пирамиду путем последовательных вставок нового ключа в ранее построенную; некоторые авторы называют этот алгоритм ***нисходящим построением пирамиды*** (top-down heap construction). Каким же образом можно добавить новый ключ в пирамиду? Начнем с добавления нового узла с ключом K после последнего листа имеющейся пирамиды, а затем переместим K в соответствующее его значению место в новой пирамиде следующим образом. Сравним K с родительским ключом: если он не меньше K, алгоритм прекращает работу (полученная структура является пирамидой). В противном случае обменяем эти два ключа и будем сравнивать K с новым родителем. Этот процесс продолжается до тех пор, пока K не перестанет превышать значение ключа в родительском узле или не достигнет корня (этот процесс проиллюстрирован на рис. 6.12). В этом алгоритме также можно переме-

щать пустой узел до достижения им корректной позиции, а затем присвоить ему значение K.

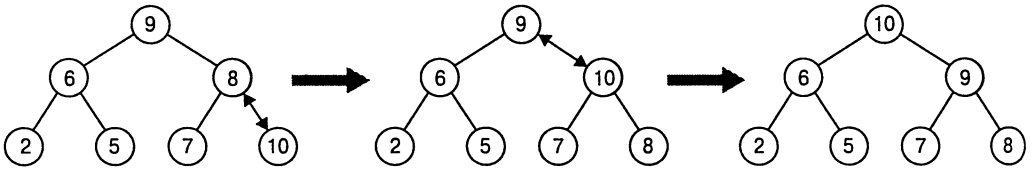


Рис. 6.12. Вставка ключа 10 в пирамиду, построенную на рис. 6.11

Очевидно, что такая вставка не может требовать большего количества сравнений ключей, чем высота пирамиды. Поскольку высота пирамиды с n узлами около $\log_2 n$, временна́я эффективность вставки составляет $O(\log n)$.

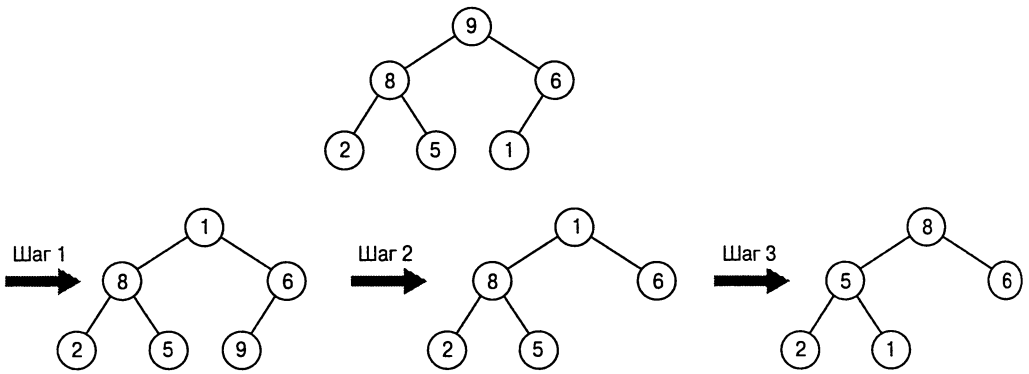


Рис. 6.13. Удаление корневого ключа из пирамиды

Как выполняется удаление узла из пирамиды? Мы рассмотрим здесь только наиболее важный случай удаления ключа из корня, оставляя вопрос об удалении произвольного ключа в качестве самостоятельного упражнения для читателя (авторы учебников вообще склонны перекладывать массу работы на читателей, правда?). Итак, удаление корневого ключа из пирамиды можно выполнить при помощи следующего алгоритма (проиллюстрированного на рис. 6.13).

Шаг 1. Обменять ключ в корне с последним ключом пирамиды.

Шаг 2. Уменьшить размер пирамиды на 1.

Шаг 3. "Пирамидизировать" уменьшенное дерево путем перемещения K вниз по дереву так же, как мы делали это в восходящем алгоритме построения пирамиды, — т.е. проверяя выполнение требования доминирования родительских узлов: если оно выполняется, алгоритм завершает работу, если нет — обмениваем K с наибольшим из дочерних узлов и повторя-

ем данную операцию до тех пор, пока в очередной позиции K требования доминирования родительских узлов не окажется выполненным.

Эффективность операции удаления определяется количеством выполняемых сравнений ключей, необходимых для "пирамидизации" дерева после того, как был сделан обмен и размер пирамиды был уменьшен на 1. Поскольку не может потребоваться сравнений больше, чем удвоенная высота пирамиды, временна́я эффективность удаления из пирамиды — $O(\log n)$.

Пирамидальная сортировка

Теперь мы можем описать ***пирамидальную сортировку*** (heapsort) — интересный алгоритм сортировки, открытый Дж. Вильямсом (J. W. J. Williams) [122]. Этот двухэтапный алгоритм работает следующим образом.

Этап 1 (построение пирамиды). Строим пирамиду для заданного массива.

Этап 2 (удаление наибольших элементов). Применяем операцию удаления корня $n-1$ раз.

В результате элементы массива удаляются в порядке уменьшения. Но поскольку при реализации пирамиды с использованием массива удаляемый элемент располагается последним, массив, получающийся в результате пирамидальной сортировки, оказывается отсортирован в порядке возрастания. Пример пошагового выполнения пирамидальной сортировки приведен на рис. 6.14 (на рис. 6.11 намеренно использовались те же входные данные, для того чтобы вы могли сравнить восходящее построение пирамиды при ее реализации в виде дерева и с использованием массива).

Поскольку мы уже знаем, что этап построения пирамиды алгоритма пирамидальной сортировки занимает время $O(n)$, осталось выяснить временну́ю эффективность второго этапа. Для количества сравнений ключей $C(n)$, необходимого для удаления корневых ключей из пирамид уменьшающегося от n до 2 размера, получаем следующее неравенство:

$$C(n) \leqslant 2\lfloor \log_2 (n-1) \rfloor + 2\lfloor \log_2 (n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \leqslant 2\sum_{i=1}^{n-1} \log_2 i \leqslant$$

$$\leqslant 2\sum_{i=1}^{n-1} \log_2 (n-1) = 2(n-1)\log_2 (n-1) \leqslant 2n \log_2 n.$$

Это означает, что для второго этапа пирамидальной сортировки $C(n) \in \in O(n \log n)$. Более подробный анализ показывает, что в действительности временна́я эффективность пирамидальной сортировки равна $\Theta(n \log n)$ как в среднем, так и в наихудшем случаях. Таким образом, временна́я эффективность пирамидальной сортировки попадает в тот же класс, что и сортировка слиянием, но,

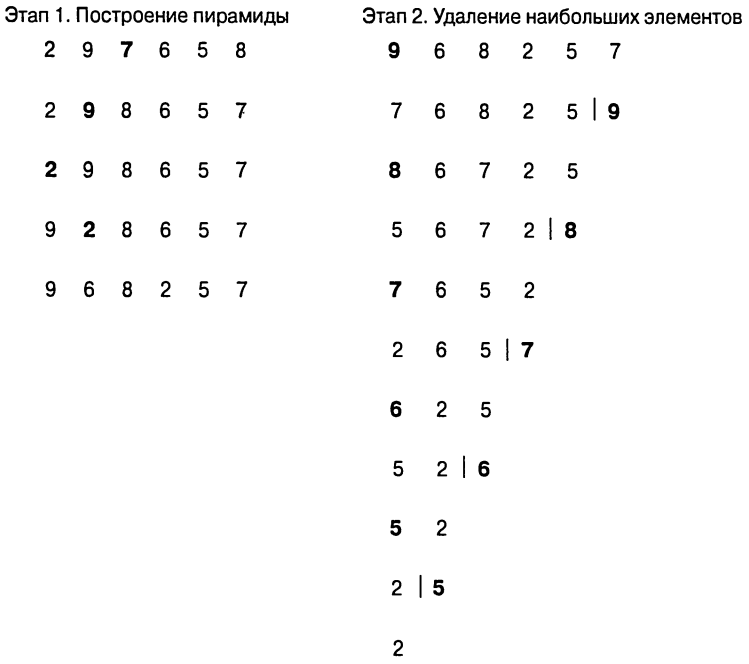


Рис. 6.14. Пирамидальная сортировка массива 2, 9, 7, 6, 5, 8

в отличие от последней, выполняется "на месте", без привлечения дополнительной памяти. Эксперименты, проведенные над случайными файлами, показывают, что пирамидальная сортировка работает медленнее быстрой, однако вполне может соперничать с сортировкой слиянием.

Упражнения 6.4

1. **а)** Постройте пирамиду для чисел 1, 8, 6, 5, 3, 7, 4 при помощи восходящего алгоритма.
   б) Постройте пирамиду для чисел 1, 8, 6, 5, 3, 7, 4 при помощи нисходящего алгоритма.
   в) Всегда ли восходящий и нисходящий алгоритмы построения пирамиды дают одну и ту же пирамиду?
2. Набросайте схему алгоритма для проверки того, является ли пирамидой массив $H[1..n]$, и определите его временну́ю эффективность.
3. **а)** Найдите наименьшее и наибольшее количество ключей, которые может содержать пирамида высотой h.
   б) Докажите, что высота пирамиды с n узлами равна $\lfloor \log_2 n \rfloor$.

4. Докажите следующее неравенство, использованное в тексте раздела:

$$\sum_{i=0}^{h-1} 2\,(h-i)\,2^i = 2\,(n - \log_2\,(n+1))\,, \quad \text{где } n = 2^{h+1} - 1.$$

5. а) Разработайте эффективный алгоритм для поиска и удаления из пирамиды элемента с наименьшим значением и определите его временну́ю эффективность.

б) Разработайте эффективный алгоритм для поиска и удаления из пирамиды элемента с заданным значением v и определите его временну́ю эффективность.

6. Отсортируйте следующие списки с помощью пирамидальной сортировки с использованием представления пирамид в виде массивов:

а) 1, 2, 3, 4, 5 (в возрастающем порядке)

б) 5, 4, 3, 2, 1 (в убывающем порядке)

в) S, O, R, T, I, N, G (в алфавитном порядке)

7. Является ли пирамидальная сортировка устойчивой?

8. Какую разновидность метода "преобразуй и властвуй" представляет собой пирамидальная сортировка?

9. Реализуйте три алгоритма сортировки — слиянием, быструю и пирамидальную — на языке программирования по вашему выбору и исследуйте их производительность для массивов размером $n = 10^2$, 10^3, 10^4, 10^5, 10^6. Для каждого размера рассмотрите

а) случайно сгенерированные числа в диапазоне $[1..n]$;

б) последовательность чисел $1, 2, \ldots, n$;

в) последовательность чисел $n, n-1, \ldots, 1$.

10. Представьте горку спагетти, длины которых представляют числа, которые требуется отсортировать.

а) Разработайте "сортировку спагетти" — алгоритм сортировки, использующий преимущества такого неортодоксального представления.

б) Какое отношение этот пример компьютерного фольклора (см. [34]) имеет к теме данной главы вообще и к пирамидальной сортировке в частности?

6.5 Схема Горнера и возведение в степень

В этом разделе мы рассмотрим задачу вычисления значения полинома

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \tag{6.1}$$

в заданной точке x и ее важный частный случай — вычисление x^n. Полиномы образуют наиболее важный класс функций благодаря множеству хороших свойств, с одной стороны, и возможности их применения для аппроксимации других типов функций, — с другой. Задача эффективной работы с полиномами актуальна несколько столетий, и за последние 50 лет были сделаны новые открытия в этой области. Несомненно, наиболее важным из них было быстрое преобразование Фурье (fast Fourier transform, FFT). Практическая важность этого замечательного алгоритма, основанного на представлении алгоритмов при помощи их значений в специально выбранных точках, настолько велика, что некоторые ученые считают его наиболее важным алгоритмическим открытием всех времен. Вследствие его относительной сложности мы не станем рассматривать быстрое преобразование Фурье и порекомендуем читателю обратиться к соответствующим учебникам, например к [102] или [32].

Схема Горнера

Схема Горнера (Horner's rule) — один из старых, но очень элегантных и эффективных алгоритмов для вычисления полиномов. Он назван по имени британского математика В. Горнера (W. G. Horner), который опубликовал этот алгоритм в начале 19 века. Однако, как утверждает Д. Кнут (D. Knuth) ([66]), еще за 150 лет до Горнера данный метод использовался И. Ньютоном (I. Newton). Вы оцените этот алгоритм еще больше, если сначала самостоятельно разработаете и исследуете эффективность алгоритма вычисления значения полинома (см. упражнения 1 и 2 к данному разделу).

Схема Горнера — хороший пример использования метода изменения представления, поскольку он основан на представлении $p(x)$ при помощи формулы, отличающейся от (6.1). Эта новая формула получается из (6.1) путем последовательного вынесения x за скобки с образованием полиномов с уменьшающимися степенями:

$$p(x) = (\ldots(a_n x + a_{n-1})\,x + \ldots)\,x + a_0. \tag{6.2}$$

Например, для полинома $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ получим:

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 = \\ &= x\left(2x^3 - x^2 + 3x + 1\right) - 5 = \\ &= x\left(x\left(2x^2 - x + 3\right) + 1\right) - 5 = \\ &= x\left(x\left(x\left(2x - 1\right) + 3\right) + 1\right) - 5. \end{aligned} \tag{6.3}$$

Для вычисления значения полинома в точке x это значение надо просто подставить в формулу (6.2). Глядя на нее, трудно поверить, что это и есть эффективный алгоритм вычисления значения полинома, однако ее неприглядность — не более чем внешний вид, и, как вы увидите, нет необходимости в такой записи полинома: все, что нам надо, — это список коэффициентов полинома.

Вычисления вручную проще организовать при помощи таблицы, состоящей из двух строк. Первая содержит коэффициенты полинома (включая те из них, которые равны 0, — если таковые имеются), перечисленные в порядке от старшего a_n к младшему a_0. Вторая строка используется для хранения промежуточных результатов (за исключением первой записи, которая просто равна a_n). После такой инициализации очередная запись в таблице вычисляется как последнее значение, умноженное на x, плюс коэффициент из первой строки. Последняя запись таблицы, вычисленная таким способом, и есть искомое значение полинома.

Пример 1. Вычисление $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ в точке $x = 3$.

Коэффициенты	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

Итак, $p(3) = 160$. (Сравнивая записи таблицы с формулой (6.3), можно видеть, что $3 \cdot 2 + (-1) = 5$ — это значение $2x - 1$ в точке $x = 3$; $3 \cdot 5 + 3 = 18$ — значение $x(2x - 1) + 3$ в точке $x = 3$; $3 \cdot 18 + 1 = 55$ — значение $x(x(2x - 1) + 3) + 1$ в точке $x = 3$ и, наконец, $3 \cdot 55 + (-5) = 160$ — значение $x(x(x(2x - 1) + 3) + 1) - 5 = p(x)$ в точке $x = 3$.) ■

Псевдокод схемы Горнера короче, чем мы себе представляем псевдокод для нетривиального алгоритма.

```
АЛГОРИТМ Horner (P[0..n], x)
  // Вычисление значения полинома в данной точке
  // по схеме Горнера
  // Входные данные:  Массив P[0..n] коэффициентов полинома
  //                  степени n (хранятся от младшего
  //                  к старшему) и число x
  // Выходные данные: Значение полинома в точке x
  p ← P[n]
  for i ← n − 1 downto 0 do
    p ← x * p + P[i]
  return p
```

Количество умножений и сложений определяется одной и той же формулой:

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$$

Чтобы оценить, насколько эффективен алгоритм схемы Горнера, рассмотрим только первый член полинома степени n: $a_n x^n$. Вычисление только одного этого члена методом грубой силы потребует n умножений, в то время как схема Горнера, кроме этого члена, вычисляет еще $n-1$ других членов и при этом использует то же количество умножений! Неудивительно, что схема Горнера — оптимальный алгоритм для вычисления полиномов без предварительной обработки полиномиальных коэффициентов.

Схема Горнера имеет несколько полезных побочных результатов. Промежуточные числа, генерируемые алгоритмом в процессе вычисления $p(x)$ в точке x_0, представляют собой коэффициенты частного от деления $p(x)$ на $x - x_0$, а конечный результат, равный значению $p(x_0)$, представляет собой остаток от деления $p(x)$ на $x - x_0$. Так, в соответствии с рассмотренным примером, частное и остаток от деления $2x^4 - x^3 + 3x^2 + x - 5$ на $x - 3$ равны, соответственно, $2x^3 + 5x^2 + 18x + 55$ и 160. Такой алгоритм деления, известный как синтетическое деление (synthetic division), более удобен, чем так называемое "длинное деление" (но в отличие от длинного деления он применим только для деления на $x - c$, где c — константа).

Бинарное возведение в степень

Поразительная эффективность схемы Горнера сводится на нет, будучи примененной к вычислению a^n, т.е. значению x^n при $x = a$. Фактически в этом частном случае алгоритм вырождается в алгоритм, основанный на применении грубой силы, — в умножение значения a само на себя, с бессмысленными прибавлениями 0 между умножениями. Поскольку вычисление a^n (на самом деле $a^n \bmod m$) является основной операцией в ряде важных алгоритмов проверки чисел на простоту и алгоритмов шифрования, мы рассмотрим два важных алгоритма вычисления a^n, которые основаны на идее изменения представления. Оба они используют бинарное представление показателя степени n, но один из них обрабатывает бинарную строку слева направо, а другой — справа налево.

Пусть $n = b_I \ldots b_i \ldots b_0$ — битовая строка, представляющая положительное целое n в двоичной системе счисления. Это означает, что значение n может быть вычислено как значение полинома

$$p(x) = b_I x^I + \cdots + b_i x^i + \cdots + b_0 \tag{6.4}$$

при $x = 2$. Например, если $n = 13$, его бинарное представление имеет вид 1101 и

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

Давайте вычислим значение этого полинома с использованием схемы Горнера — мы увидим, как из него вытекает способ вычисления степени

$$a^n = a^{p(2)} = a^{b_I x^I + \cdots + b_i x^i + \cdots + b_0}.$$

Схема Горнера для вычисления бинарного полинома $p(2)$	Следствие для вычисления $a^n = a^{p(2)}$

$p \leftarrow 1$ // Старший разряд всегда 1 при $n \geqslant 1$
for $i \leftarrow I-1$ **downto** 0
  $p \leftarrow 2p + b_i$
	$a^p \leftarrow a^1$
	for $i \leftarrow I-1$ **downto** 0
	  $a^p \leftarrow a^p \leftarrow a^{2p+b_i}$

Однако

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{если } b_i = 0, \\ (a^p)^2 \cdot a & \text{если } b_i = 1. \end{cases}.$$

Таким образом, после инициализации переменной-аккумулятора значением a можно сканировать битовую строку, представляющую показатель степени, при каждом сканировании нового бита возводя значение аккумулятора в квадрат и, если сканируемый бит равен 1, дополнительно умножая полученный квадрат на величину a. Это наблюдение приводит нас к следующему методу вычисления a^n — бинарному возведению в степень слева направо (left-to-right binary exponentiation).

Алгоритм $LeftRightBinaryExponentiation\,(a, b\,(n))$

```
// Вычисление a^n при помощи алгоритма бинарного возведения
// в степень слева направо
// Входные данные:  Число a и список b(n) битов b_I, ..., b_0
//                  в бинарном представлении натурального n
// Выходные данные: Значение a^n
product ← a
for i ← I − 1 downto 0 do
  product ← product * product
  if b_i = 1
    product ← product * a
return product
```

Пример 2. Вычислим a^{13} при помощи алгоритма бинарного возведения в степень слева направо. Здесь $n = 13 = 1101_2$. Таким образом, мы имеем

Биты n	1	1	0	1
Аккумулятор	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

■

Поскольку алгоритм выполняет при каждом повторении своего единственного цикла одно или два умножения, общее количество умножений $M(n)$ при вычислении a^n составляет

$$(b-1) \leqslant M(n) \leqslant 2(b-1),$$

где b — длина битовой строки, представляющей показатель степени n. Учитывая, что $b-1 = \lfloor \log_2 n \rfloor$, можно заключить, что эффективность бинарного возведения в степень слева направо — логарифмическая, так что этот алгоритм принадлежит лучшему классу эффективности, чем возведение в степень, основанное на грубой силе (которое всегда требует $n-1$ умножений).

При бинарном возведении в степень справа налево (right-to-left binary exponentiation) используется тот же полином $p(2)$ (см. (6.4)), дающий значение n, но вместо применения схемы Горнера полином используется иначе:

$$a^n = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0} = a^{b_I 2^I} \cdot \ldots \cdot a^{b_i 2^i} \cdot \ldots \cdot a^{b_0}.$$

Таким образом, a^n можно вычислить как произведение членов

$$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{если } b_i = 1, \\ 1 & \text{если } b_i = 0, \end{cases}$$

т.е. произведение последовательных членов a^{2^i}, с опусканием тех из них, для которых бит b_i равен 0. Кроме того, a^{2^i} можно вычислять путем возведения в квадрат члена, вычисленного для предыдущего значения i, поскольку $a^{2^i} = \left(a^{2^{i-1}}\right)^2$. Мы вычисляем все такого вида степени a, от наименьшей к наибольшей (справа налево), но в результат включаются только те из них, для которых соответствующий бит равен 1. Вот как выглядит псевдокод данного алгоритма.

Алгоритм *RightLeftBinaryExponentiation* $(a, b(n))$

```
// Вычисление a^n при помощи алгоритма бинарного возведения
// в степень справа налево
// Входные данные:  Число a и список b(n) битов b_I, ..., b_0
//                  в бинарном представлении натурального n
// Выходные данные: Значение a^n
term ← a   // Инициализация a^(2^i)
if b_0 = 1
  product ← a
else
  product ← 1
for i ← 1 to I do
  term ← term * term
  if b_i = 1
```

$product \leftarrow product * term$
return $product$

Пример 3. Вычислим a^{13} при помощи алгоритма бинарного возведения в степень справа налево. Здесь $n = 13 = 1101_2$. Таким образом, имеем

1	1	0	1	Биты n
a^8	a^4	a^2	a	Члены a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	Аккумулятор произведения

■

Очевидно, что эффективность данного алгоритма также логарифмическая — по тем же причинам, что и для алгоритма бинарного возведения в степень слева направо. Полезность обоих алгоритмов несколько снижается из-за того, что требуется точное бинарное разложение показателя степени n. В упражнении 8 к данному разделу требуется разработать алгоритм, который лишен этого недостатка.

Упражнения 6.5

1. Рассмотрим следующий алгоритм вычисления полинома, основанный на применении грубой силы.

   АЛГОРИТМ $BruteForcePolynomialEvaluation\,(P\,[0..n]\,, x)$
   // Алгоритм вычисляет значение полинома P в точке x
   // с использованием алгоритма на основе грубой силы,
   // вычисляющего члены от большей степени к меньшей
   // **Входные данные:** Массив $P[0..n]$ коэффициентов полинома
   // степени n, хранящиеся начиная от
   // меньшего к большему, и число x
   // **Выходные данные:** Значение полинома в точке x
   $p \leftarrow 0.0$
   for $i \leftarrow n$ **downto** 0 **do**
     $power \leftarrow 1$
     for $j \leftarrow 1$ **to** i **do**
       $power \leftarrow power * x$
     $p \leftarrow p + P[i] * power$
   return p

   Найдите общее количество умножений и сложений, выполняемых этим алгоритмом.

2. Напишите псевдокод алгоритма вычисления полинома, основанного на применении грубой силы, который подставляет значение переменной в формулу и вычисляет ее от меньшего члена к большему. Найдите общее количество умножений и сложений, выполняемых таким алгоритмом.
3. **а)** Оцените, насколько схема Горнера быстрее алгоритма грубой силы из упражнения 2, если 1) время одного умножения значительно превышает время одного сложения; 2) время одного умножения примерно равно времени одного сложения.

   б) Является ли схема Горнера быстрее алгоритма грубой силы ценой использования повышенного количества памяти?
4. **а)** Примените схему Горнера для вычисления полинома

$$p(x) = 3x^4 - x^3 + 2x + 5 \text{ в точке } x = -2.$$

   б) Используйте результаты работы схемы Горнера для поиска частного и остатка от деления $p(x)$ на $x + 2$.
5. Сравните количество умножений и сложений/вычитаний, необходимых для "длинного деления" полинома $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ на $x - c$, где c — константа, с количеством этих операций при "синтетическом делении".
6. **а)** Примените бинарное возведение в степень слева направо для вычисления a^{17}.

   б) Можно ли расширить бинарное возведение в степень слева направо так, чтобы оно работало для всех неотрицательных целых показателей степени?
7. Примените бинарное возведение в степень справа налево для вычисления a^{17}.
8. Разработайте нерекурсивный алгоритм для вычисления a^n, который имитирует бинарное возведение в степень справа налево, но не использует явное бинарное представление n.
9. Стоит ли использовать такой алгоритм общего назначения, как схема Горнера, для вычисления полинома $p(x) = x^n + x^{n-1} + \cdots + x + 1$?
10. В соответствии со следствием из Основной теоремы алгебры, каждый полином

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

    может быть представлен в виде

$$p(x) = a_n (x - x_1)(x - x_2) \ldots (x - x_n),$$

где $x_1, x_2, \ldots, x_n$ — корни полинома (в общем случае комплексные и не обязательно различные). Подумайте, какое из двух представлений более удобно для каждой из следующих операций:

а) вычисления полинома в данной точке;

б) сложения двух полиномов;

в) умножения двух полиномов.

6.6 Приведение задачи

Знаете ли вы историю о мальчике, которого мама учила, как сварить яйцо на завтрак в ее отсутствие? Он должен был снять с полки кастрюльку, налить в нее воду, поставить на плиту, включить плиту, положить яйцо в воду и через 5 минут после закипания достать вареное яйцо, выключить плиту, помыть и поставить на место кастрюльку... Но однажды мама не очень торопилась на работу, так что пришедший на кухню мальчик увидел, что кастрюлька с водой уже стоит на плите... Не растерявшись, он вылил воду, поставил кастрюльку на полку и вышел из кухни. А потом зашел и выполнил все действия, которым его научила мама.

Способ, которым мальчик приготовил яйцо на завтрак, — пример важной стратегии решения задач, именуемой ***приведением***, или ***сведением задачи*** (problem reduction). Если вам надо решить задачу, ее можно привести к другой задаче, решение которой вам известно (рис. 6.15).

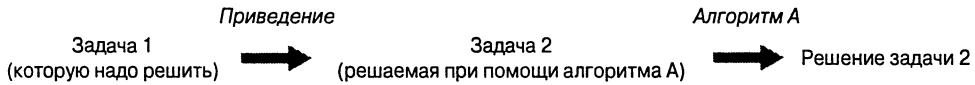


Рис. 6.15. Стратегия приведения задачи

Идея приведения задачи играет центральную роль в теоретической кибернетике, где она используется для классификации задач в соответствии с их сложностью (об этом мы поговорим в главе 10). Однако эта же стратегия может использоваться и для решения практических задач. Сложность заключается в том, чтобы найти задачу, к которой можно привести исходную. Кроме того, если мы хотим получить практический результат, необходимо, чтобы алгоритм приведения был более эффективным, чем алгоритм непосредственного решения исходной задачи.

Заметим, что раньше вы уже встречались с этой методикой. Например, в разделе 6.5 упоминалось о так называемом синтетическом делении, выполняемом при использовании схемы Горнера для вычисления значения полинома. В разделе 4.6 мы использовали следующий факт из аналитической геометрии: если $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ и $p_3 = (x_3, y_3)$ — три произвольные точки на плоско-

сти, то определитель

$$\det \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

положителен тогда и только тогда, когда точка p_3 находится слева от ориентированной прямой $\overrightarrow{p_1p_2}$, проведенной через точки p_1 и p_2. Другими словами, мы свели геометрическую задачу об относительном расположении трех точек к задаче о знаке определителя матрицы. Более того, вся аналитическая геометрия основана на идее приведения геометрических задач к алгебраическим, и основная заслуга в этом принадлежит Рене Декарту (René Descartes) (1596–1650). В этом разделе мы приведем еще несколько примеров алгоритмов, основанных на стратегии приведения задач.

Вычисление наименьшего общего кратного

Вспомним, что ***наименьшее общее кратное*** (least common multiple) двух натуральных чисел m и n (обозначаемое как $\mathrm{lcm}\,(m,n)$) определяется как наименьшее натуральное число, делящееся и на m, и на n. Например, $\mathrm{lcm}\,(24,60) = 120$ и $\mathrm{lcm}\,(11,5) = 55$. Наименьшее общее кратное — одно из наиболее важных понятий в арифметике и алгебре; возможно, вы вспомните метод его поиска, которому вас учили в школе. Если имеется разложение чисел m и n на простые множители, то $\mathrm{lcm}\,(m,n)$ можно вычислить как произведение всех общих простых множителей m и n, умноженное на произведение простых множителей m, не являющихся множителями n, и на произведение простых множителей n, не являющихся множителями m. Например,

$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$\mathrm{lcm}\,(24,60) = (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120$$

В качестве вычислительной процедуры данный алгоритм имеет те же недостатки, что и упоминавшийся в разделе 1.1 алгоритм поиска наибольшего общего делителя путем разложения чисел на простые множители — неэффективность и требование наличия списка последовательных простых чисел.

Существенно более эффективный алгоритм вычисления наименьшего общего кратного можно разработать при помощи приведения задачи. У нас есть очень эффективный алгоритм Евклида для поиска наибольшего общего делителя, который представляет собой произведение всех общих простых множителей m и n. Можем ли мы написать формулу, связывающую $\mathrm{lcm}\,(m,n)$ и $\gcd\,(m,n)$? Нетрудно увидеть, что произведение $\mathrm{lcm}\,(m,n)$ и $\gcd\,(m,n)$ включает по одному разу

каждый из простых множителей m и n, следовательно, это произведение просто равно произведению m и n. Это наблюдение приводит к формуле

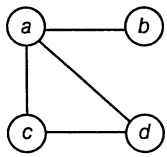
$$\operatorname{lcm}(m,n) = \frac{m \cdot n}{\gcd(m,n)},$$

где $\gcd(m,n)$ можно вычислить при помощи эффективного алгоритма Евклида.

Подсчет путей в графе

В качестве следующего примера рассмотрим задачу подсчета путей между двумя вершинами графа. Методом математической индукции нетрудно доказать, что количество различных путей длиной $k > 0$ от i-ой к j-ой вершине графа (ориентированного или неориентированного) равно (i,j)-ому элементу A^k, где A — матрица смежности графа (кстати, рассмотренные в предыдущем разделе алгоритмы возведения чисел в степень применимы и для возведения в степень матриц). Таким образом, задача подсчета путей в графе может быть решена при помощи алгоритма для вычисления соответствующей степени его матрицы смежности.

В качестве конкретного примера рассмотрим граф на рис. 6.16. Его матрица смежности A и ее квадрат A^2 указывают количество путей (длиной, соответственно, 1 и 2) между вершинами графа. В частности, имеется три пути длиной 2, начинающиеся и заканчивающиеся в вершине a: $a-b-a$, $a-c-a$ и $a-d-a$, и только один путь длиной 2 от a до c: $a-d-c$.



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

Рис. 6.16. Граф, его матрица смежности и ее квадрат A^2. Элементы матриц A и A^2 указывают количество путей длиной 1 и 2, соответственно

Приведение задач оптимизации

Очередной пример — решение задач оптимизации. Если в задаче требуется найти максимум некоторой функции, говорят, что это ***задача максимизации*** (maximization problem); если же задача состоит в поиске минимума — то это ***задача минимизации*** (minimization problem). Предположим, требуется найти минимум некоторой функции $f(x)$, и у нас есть алгоритм, позволяющий найти максимум функции. Как можно им воспользоваться? Ответ находится в простейшей формуле:

$$\min f(x) = -\max\left[-f(x)\right].$$

Другими словами, чтобы минимизировать функцию, можно максимизировать функцию с обратным знаком, а чтобы получить корректное значение минимума, надо изменить знак у найденного максимума. Это свойство проиллюстрировано на рис. 6.17.

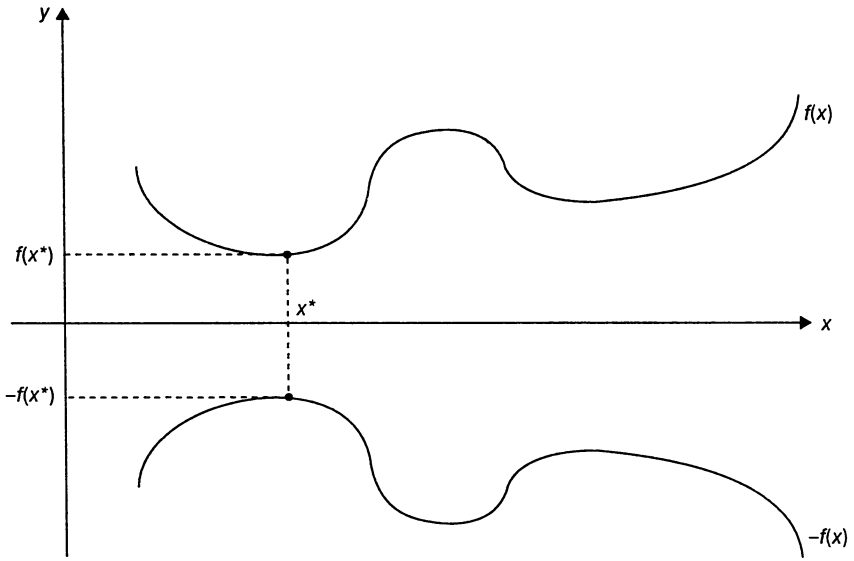


Рис. 6.17. Связь между задачами минимизации и максимизации: $\min f(x) = -\max[-f(x)]$

Само собой, справедлива и формула, позволяющая свести задачу максимизации к задаче минимизации:

$$\max f(x) = -\min[-f(x)].$$

Это соотношение между задачами минимизации и максимизации носит самый общий характер — оно выполняется для функций с любой областью определения D. В частности, его можно применить к функциям нескольких переменных при дополнительных ограничениях. Очень важный класс таких задач будет рассмотрен в следующем подразделе.

Раз уж мы рассматриваем задачи оптимизации функции, стоит отметить, что стандартная процедура поиска точек экстремума функции фактически основана на приведении задачи. Она предполагает поиск производной функции $f'(x)$ и решение уравнения $f'(x) = 0$ для поиска критических точек функции. Другими словами, задача оптимизации приводится к задаче решения уравнения как основной части поиска точек экстремума. Заметим, что мы не называем эту процедуру алгоритмом, поскольку она не является точно определенной. В действительности не существует общих методов решения уравнений. Маленький секрет учебников

заключается в том, что рассматриваемые в них примеры задач тщательно отобраны так, что критические точки всегда можно найти без особых затруднений. Это облегчает жизнь студентам и преподавателям, но может создать у студентов ложное впечатление простоты задач.

Линейное программирование

Многие задачи, в которых требуется принять оптимальное решение, могут быть приведены к экземпляру ***задачи линейного программирования*** (linear programming), которая представляет собой задачу оптимизации линейной функции нескольких переменных при накладываемых ограничениях в виде линейных уравнений и неравенств.

Пример 1. Рассмотрим задачу вклада денег. Имеется сумма в 100 000$, которую можно разместить в акциях, на депозитном счету или на текущем счету. Акции дают годовую прибыль 10%, депозит — 7%, а текущий счет — 3%. Поскольку вкладывание денег в акции — занятие рискованное, в них можно вложить не более трети от суммы, положенной на депозитный счет. Кроме того, не менее 25% от общей суммы, вложенной в акции и на депозит, должно быть положено на текущий счет. Как разместить деньги, чтобы получить максимальную прибыль?

Создадим математическую модель. Пусть x, y и z — суммы в тысячах долларов, вкладываемые, соответственно, в акции, на депозит и на текущий счет. При использовании таких обозначений мы можем сформулировать задачу следующим образом:

$$\begin{array}{ll} \text{Максимизировать} & 0.10x + 0.07y + 0.03z \\ \text{при условиях} & x + y + z = 100 \\ & x \leqslant \frac{1}{3}y \\ & z \geqslant 0.25\,(x + y) \\ & x \geqslant 0, y \geqslant 0, z \geqslant 0. \end{array}$$

■

Хотя эта конкретная задача невелика и проста, она показывает, как задача принятия оптимального решения может быть приведена к экземпляру задачи линейного программирования

$$\begin{array}{ll} \text{Максимизировать} & c_1x_1 + \cdots + c_nx_n \\ \text{(или минимизировать)} & \\ \text{при условиях} & a_{i1}x_1 + \cdots + a_{in}x_n \leqslant \text{ (или } \geqslant \text{, или } =) \; b_i \\ & \qquad \text{при } i = 1, \ldots, m \\ & x_1 \geqslant 0, \ldots, x_n \geqslant 0. \end{array}$$

(Последняя группа ограничений — так называемые ограничения неотрицательности — являются, строго говоря, необязательными, поскольку они представляют

собой частные случаи более общих ограничений $a_{i1}x_1 + \cdots + a_{in}x_n \geqslant b_i$; просто их удобнее рассматривать отдельно.)

Доказано, что линейное программирование достаточно гибко для моделирования широкого диапазона важных приложений, таких как расписание полетов экипажей авиакомпаний, нефтеразведка и нефтедобыча или оптимизация промышленного производства. Многие ученые рассматривают линейное программирование как одно из наиболее важных достижений в истории прикладной математики. Классический алгоритм для решения задачи линейного программирования называется ***симплекс-методом*** (simplex method); он был открыт американским математиком Дж. Данцигом (G. Dantzig) в 1940-х годах [33]. Хотя в наихудшем случае этот алгоритм экспоненциален, для обычных входных данных он работает очень хорошо. Усилия множества кибернетиков за последние 50 лет довели алгоритм и его реализацию до того состояния, когда задача с десятками, если не сотнями тысяч переменных и ограничений решается за вполне разумное время. Кроме того, относительно недавно были открыты и другие алгоритмы для решения задач линейного программирования общего вида; наиболее известен среди них алгоритм Наренды Кармаркара (Narenda Karmarkar) [57]. Теоретическая важность этих новейших алгоритмов заключается в их доказанной полиномиальности в наихудшем случае; алгоритм же Кармаркара, как показали эмпирические тесты, способен, помимо прочего, составить конкуренцию по эффективности симплекс-методу.

Важно отметить, однако, что симплекс-метод и алгоритм Кармаркара в состоянии успешно решать только задачи линейного программирования, в которых отсутствует ограничение, что все переменные могут принимать только целочисленные значения. Если такое ограничение присутствует в задаче, то она называется ***целочисленной задачей линейного программирования*** (integer linear programming). Известно, что целочисленные задачи линейного программирования намного сложнее, и для них неизвестен алгоритм решения с полиномиальным временем работы (как вы узнаете из главы 10, вполне вероятно, что такой алгоритм вообще не существует). Для решения таких задач используются иные подходы, с одним из которых вы познакомитесь в разделе 11.2.

Пример 2. Рассмотрим приведение задачи о рюкзаке к задаче линейного программирования. Вспомним формулировку этой задачи из раздела 3.4: дано n предметов весом $w_1, \ldots, w_n$ и ценой $v_1, \ldots, v_n$, а также рюкзак, выдерживающий вес W. Наша задача — найти подмножество предметов, которые можно поместить в рюкзак и которые имеют при этом максимальную стоимость. Сначала рассмотрим так называемую ***непрерывную*** (continuous), или ***дробную*** (fractional), версию задачи, в которой в рюкзак можно помещать произвольную часть любого предмета. Пусть x_j, $j = 1, \ldots, n$ — переменная, представляющая часть предмета j, помещаемую в рюкзак. Очевидно, что x_j должно удовлетворять неравенству $0 \leqslant x_j \leqslant 1$. Тогда

общий вес выбранных предметов можно выразить как сумму $\sum_{j=1}^{n} w_j x_j$, а общую их стоимость — как $\sum_{j=1}^{n} v_j x_j$. Таким образом, непрерывную версию задачи о рюкзаке можно представить в виде следующей задачи линейного программирования:

$$\begin{aligned} &\text{Максимизировать} && \sum_{j=1}^{n} v_j x_j \\ &\text{при условиях} && \sum_{j=1}^{n} w_j x_j \leqslant W \\ & && 0 \leqslant x_j \leqslant 1, \quad j = 1, \ldots, n. \end{aligned}$$

Не имеет смысла применять универсальные методы решения задач линейного программирования для данной конкретной задачи — она решается гораздо проще при помощи специального алгоритма, с которым мы встретимся в разделе 11.3 (но зачем ждать? Попробуйте разработать его самостоятельно). Приведение задачи о рюкзаке к экземпляру задачи линейного программирования имеет смысл только для доказательства корректности рассматриваемого алгоритма.

В так называемой ***дискретной*** (discrete), или ***0-1***, задаче о рюкзаке предмет можно класть в рюкзак только целиком, или не брать его совсем. Следовательно, эта задача приводится к следующей задаче линейного программирования:

$$\begin{aligned} &\text{Максимизировать} && \sum_{j=1}^{n} v_j x_j \\ &\text{при условиях} && \sum_{j=1}^{n} w_j x_j \leqslant W \\ & && x_j \in \{0, 1\}, \quad j = 1, \ldots, n. \end{aligned}$$

Такое кажущееся совсем незначительным отличие приводит к кардинальному отличию в сложности этой и подобных задач линейного программирования, в которых переменные могут принимать только дискретные значения. Несмотря на то что 0-1 версия задачи выглядит более простой, так как в ней заведомо отбрасываются все подмножества непрерывной версии, в которых используются дробные части предметов, на самом деле такая задача гораздо сложнее своего непрерывного аналога. Читатель, интересующийся алгоритмами решения этой задачи, найдет массу литературы по данному вопросу, включая монографию [77]. ■

Приведение к задачам о графах

Как мы упоминали в разделе 1.3, многие задачи можно решить путем приведения к одной из стандартных задач о графах. Это верно, в частности, для

множества головоломок и игр. В приложениях такого рода вершины графа обычно представляют возможные состояния рассматриваемой задачи, в то время как ребра представляют разрешенные переходы между состояниями. Одна из вершин графа представляет начальное состояние, а некоторая другая — конечное, целевое состояние задачи (таких конечных вершин может быть несколько). Такой граф называется ***графом пространства состояний*** (state-space graph). Таким образом, только что описанное преобразование приводит задачу к вопросу о пути из начальной вершины в конечную.

Пример 3. В качестве примера давайте обратимся к классической головоломке о переправе через реку, с которой мы уже встречались в упражнении 1.2.1: на берегу реки находятся крестьянин, волк, коза и кочан капусты. Крестьянин должен перевезти их на другой берег. Однако в лодке только два места — для крестьянина и еще одного объекта (т.е. либо волка, либо козы, либо капусты). В отсутствие крестьянина волк может съесть козу, а коза — капусту. Помогите крестьянину решить эту задачу или докажите, что она не имеет решения.

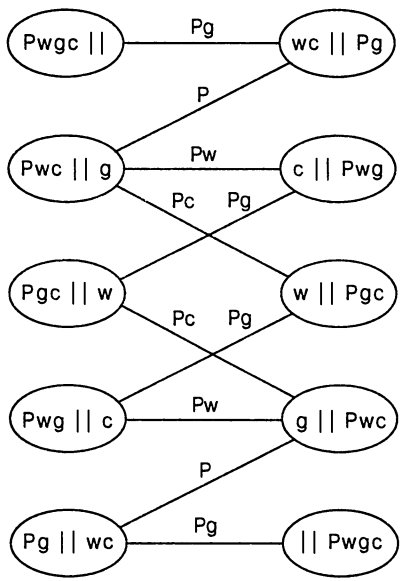


Рис. 6.18. Граф пространства состояний для головоломки о переправе через реку

Граф пространства состояний для головоломки о переправе через реку показан на рис. 6.18. В вершинах имеются метки, показывающие представляемые ими состояния: P, w, g и c означают крестьянина, волка, козу и капусту, соответственно (мы оставили английские обозначения, так как на русском языке три персонажа головоломки начинаются с одной буквы... — *Прим. перев.*); две вертикальные

черты (||) символизируют реку. Для удобства мы также пометили ребра, указывая, кто именно находится в лодке при пересечении реки. В терминах данного графа нас интересует поиск пути из начальной вершины, помеченной Pwgc||, в конечную — ||Pwgc.

Легко видеть, что имеется два различных простых пути из начальной вершины в конечную (какие?). Если мы найдем их при помощи поиска в ширину, это будет служить формальным доказательством того, что данные пути — кратчайшие. Следовательно, головоломка имеет два решения, каждое из которых состоит из семи пересечений реки. ■

Успех в решении этой простой головоломки не должен привести нас к убеждению, что генерация и исследование графа пространства состояний — всегда простая задача. Для лучшего понимания графов пространств состояний обратитесь к книгам по проблеме искусственного интеллекта, области кибернетики, в которой такие задачи являются основным объектом исследования. В книге мы встретимся с важными частными случаями графов пространств состояний в разделах 11.1 и 11.2.

Упражнения 6.6

1. а) Докажите равенство, на котором основан алгоритм поиска $\operatorname{lcm}(m, n)$:

$$\operatorname{lcm}(m, n) = \frac{m \cdot n}{\gcd(m, n)}.$$

б) Как известно, алгоритм Евклида имеет время работы $O(\log n)$. Если этот алгоритм используется для поиска $\gcd(m, n)$, то какова эффективность алгоритма поиска $\operatorname{lcm}(m, n)$?

2. Дано множество чисел, для которого вы должны построить неубывающую пирамиду (в неубывающей пирамиде каждый ключ не превышает дочерних ключей). Как использовать алгоритм построения невозрастающей пирамиды (пирамиды, определенной в разделе 6.4) для построения неубывающей пирамиды?

3. Докажите, что количество различных путей длиной $k > 0$ из i-ой вершины в j-ую вершину графа (неориентированного или ориентированного) равно (i, j)-му элементу A^k, где A — матрица смежности графа.

4. а) Разработайте алгоритм со временно́й эффективностью, лучшей кубической, для проверки того, не содержит ли граф с n вершинами цикла длиной 3 [76].

б) Рассмотрим следующий алгоритм для решения этой задачи. Начиная с произвольной вершины, обходим граф при помощи поиска

в глубину и проверяем, нет ли в лесу поиска в глубину вершины с обратным ребром, ведущим к "деду". Если такая вершина есть, граф содержит треугольник; если нет — в графе нет подграфа в виде треугольника. Корректен ли данный алгоритм?

5. На координатной плоскости дано $n > 3$ точек $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$. Разработайте алгоритм для проверки того факта, что все точки лежат внутри треугольника, вершинами которого являются три точки из данного множества. (Вы можете как разработать собственный алгоритм с нуля, так и привести задачу к другой задаче с известным алгоритмом решения.)

6. Рассмотрим задачу поиска для заданного натурального n пары целых чисел, сумма которых равна n и произведение которых максимально. Разработайте эффективный алгоритм решения этой задачи и определите его класс эффективности.

7. Задача о назначениях, рассматривавшаяся в разделе 3.4, может быть сформулирована следующим образом. Имеется n работников, которые должны выполнить n заданий, по одному заданию каждый (т.е. каждому работнику назначается в точности одно задание, и каждое задание назначается одному человеку). Стоимость выполнения i-ым работником j-го задания известна и равна $C[i, j]$ для всех пар $i, j = 1, \ldots n$. Задача заключается в том, чтобы распределить задания между работниками с наименьшей общей стоимостью. Выразите эту задачу в виде 0-1 задачи линейного программирования.

8. Решите экземпляр задачи линейного программирования, приведенный в разделе 6.6:

$$\begin{aligned} &\text{Максимизировать} && 0.10x + 0.07y + 0.03z \\ &\text{при условиях} && x + y + z = 100 \\ & && x \leqslant \frac{1}{3}y \\ & && z \geqslant 0.25\,(x + y) \\ & && x \geqslant 0, y \geqslant 0, z \geqslant 0. \end{aligned}$$

9. Задача раскрашивания графа обычно формулируется как задача раскраски вершин: распределить минимальное количество цветов по вершинам данного графа так, чтобы никакие две смежные вершины не были окрашены в одинаковый цвет. Рассмотрим задачу ***раскрашивания ребер***: распределить минимальное количество цветов по ребрам данного графа так, чтобы никакие два ребра, имеющие общую точку,

не были окрашены в одинаковый цвет. Поясните, как задача раскрашивания ребер может быть приведена к задаче раскрашивания вершин.

10. *Головоломка о ревнивых мужьях.* Имеется $n \geqslant 2$ семейных пар, которым надо переправиться через реку. У них есть лодка, в которой может поместиться не более двух человек. Все мужья ревнивы, и не хотят, чтобы их жены оставались без них на берегу реки, где есть хоть один мужчина, жена которого находится на другом берегу реки, — даже если при этом присутствуют другие люди. Могут ли они переправиться через реку при таких ограничениях?

а) Решите задачу для $n = 2$.

б) Решите задачу при $n = 3$ (это и есть классическая версия данной головоломки).

в) Имеет ли головоломка решение для всех $n \geqslant 4$? Если да, укажите, сколько пересечений реки потребуется для переправы; если нет, поясните, почему.

Резюме

- *Метод "преобразуй и властвуй"* — четвертая стратегия разработки алгоритмов (и решения задач), рассмотренная в книге. Фактически она представляет собой группу методов, основанных на идее преобразования в более простую для решения задачу.
- Имеется три основных варианта стратегии преобразования: *упрощение экземпляра*, *изменение представления* и *приведение задачи*.
- *Упрощение экземпляра* представляет собой метод преобразования экземпляра задачи в экземпляр той же задачи с некоторыми специальными свойствами, которые делают более простым ее решение. Предварительная сортировка, исключение Гаусса и AVL-деревья являются хорошими примерами применения этого метода.
- *Изменение представления* — это преобразование одного представления экземпляра задачи в другое представление того же экземпляра задачи. Примеры использования этого метода, рассмотренные в данной главе, включают представление множества 2-3-деревом, пирамиды и пирамидальную сортировку, схему Горнера для вычисления полиномов и два алгоритма бинарного возведения в степень.
- *Приведением задачи* называется преобразование данной задачи в другую задачу, которая может быть решена при помощи известного алгоритма. Среди примеров применения этой идеи к решению алгорит-

мических задач особенно важное место занимают приведение к задаче линейного программирования и задаче о графе.

- Некоторые примеры, использованные для иллюстрации метода преобразования, представляют собой очень важные структуры данных и алгоритмы. Это пирамиды и пирамидальная сортировка, AVL-деревья и 2-3-деревья, метод исключения Гаусса и схема Горнера.
- Пирамида представляет собой практически полное бинарное дерево с ключами (по одному на узел), удовлетворяющее требованию доминирования родительских узлов. Будучи определенной как дерево, пирамида обычно реализуется в виде массива. Пирамиды являются наиболее важной структурой данных при реализации очередей с приоритетами, а также для пирамидальной сортировки.
- *Пирамидальная сортировка* представляет собой теоретически важный алгоритм сортировки, основанный на расстановке элементов массива в виде пирамиды с последующим последовательным удалением наибольших элементов пирамид, образующихся в результате удаления. Время работы такого алгоритма равно $\Theta(n \log n)$ как в среднем, так и в наихудшем случае; кроме того, такая сортировка выполняется без привлечения дополнительной памяти.
- *AVL-деревья* представляют собой бинарные деревья поиска, которые всегда сбалансированы в той степени, в которой это возможно для бинарных деревьев. Сбалансированность поддерживается путем четырех видов преобразований, называющихся *поворотами*. Все базовые операции над AVL-деревьями выполняются за время $\Theta(\log n)$; таким образом, в этих деревьях устранена неэффективность классических бинарных деревьев поиска в наихудшем случае.
- *2-3-деревья* достигают идеальной сбалансированности дерева поиска, позволяя узлу содержать до двух упорядоченных ключей и до трех дочерних узлов. Обобщение этой идеи дает очень важные B-деревья, которые рассматриваются позже в этой книге.
- *Метод исключения Гаусса* — алгоритм для решения систем линейных уравнений — представляет собой основной алгоритм линейной алгебры. Он решает систему линейных уравнений путем преобразования ее в эквивалентную систему с верхнетреугольной матрицей коэффициентов, которая легко решается методом обратной подстановки. Метод исключения Гаусса требует около $n^3/3$ умножений.
- *Схема Горнера* является оптимальным алгоритмом вычисления полинома без предварительной обработки коэффициентов и требует только n умножений и n сложений. Кроме того, этот алгоритм дает по-

лезные побочные результаты, как, например, алгоритм синтетического деления.

- Два алгоритма *бинарного возведения в степень* для вычисления a^n рассматриваются в разделе 6.5. В обоих используется бинарное представление показателя степени n, но работают они в разных направлениях: слева направо и справа налево.
- *Линейное программирование* предназначено для оптимизации линейной функции нескольких переменных при ограничениях в виде линейных уравнений и линейных неравенств. Имеются эффективные алгоритмы, способные решать очень большие экземпляры этой задачи со многими тысячами переменных и ограничений, если только не наложено требование целочисленности переменных. В этом случае мы получаем *целочисленную задачу линейного программирования*, относящуюся к классу гораздо более сложных задач.

Глава 7

Пространственно-временной компромисс

> Значащее много никогда не должно находиться во власти значащего мало.
>
> — Иоганн Вольфганг фон Гете (Johann Wolfganh von Goethe) (1749–1832)

Достижение компромисса между пространством и временем, т.е. между используемой памятью и скоростью работы при разработке алгоритмов — вопрос, хорошо известный как теоретикам, так и практикам алгоритмики. Рассмотрим в качестве примера задачу вычисления значений функции во многих точках ее области определения. Если более важным является время работы, значения функции могут быть вычислены заранее и храниться в таблице. Это именно то, что делали люди до эпохи компьютеров для повышения скорости вычислений (возможно, кое-кто из вас еще помнит толстые тома математических таблиц). Поиск в таких таблицах в наше время потерял привлекательность благодаря наличию вычислительных машин, но сама их идея весьма ценна при разработке некоторых важных алгоритмов. Говоря более обобщенно, идея состоит в предварительной полной или частичной обработке входных данных с сохранением полученной дополнительной информации для ускорения позднейшего решения задачи. Мы назовем такой подход ***улучшением входных данных***[1] (input enhancement) и рассмотрим следующие алгоритмы, основанные на этом методе:

- метод сортировки подсчетом (раздел 7.1);
- алгоритм Бойера–Мура для поиска подстрок и его упрощенная версия, предложенная Хорспулом (раздел 7.2).

[1]Стандартный термин, использующийся для обозначения этого метода, — предварительная обработка (preprocessing). К сожалению, этот же термин применим и к методам, которые используют идею предварительной обработки, но не используют дополнительную память (см. главу 6). Поэтому, чтобы избежать неоднозначности, мы будем использовать для рассматриваемых здесь методов повышения производительности за счет использования памяти термин "улучшение входных данных".

Другой метод из этой группы просто использует дополнительную память для того, чтобы обеспечить более быстрый и/или более гибкий доступ к данным. Этот подход мы будем называть ***предварительной структуризацией*** (prestructuring). Это название подчеркивает два аспекта этой версии пространственно-временного компромисса: выполнение некоторой обработки до того, как приступить к собственно решению задачи; однако речь здесь идет не об улучшении входных данных, а о структуризации доступа к ним. Этот подход будет проиллюстрирован следующими алгоритмами:

- хеширование (раздел 7.3);
- индексирование при помощи B-деревьев.

Имеется еще один метод разработки алгоритмов, относящийся к рассматриваемому пространственно-временному компромиссу, — это ***динамическое программирование*** (dynamic programming). Эта стратегия основана на записи решений перекрывающихся подзадач данной задачи в таблице, при помощи которой затем получается решение основной задачи. Мы рассмотрим эту хорошо отработанную методику отдельно, в следующей главе книги.

Следует сделать еще два замечания о взаимодействии между временем и пространством в разработке алгоритмов. Во-первых, эти два ресурса — пространство и время — не обязательно конкурируют друг с другом во всех ситуациях. Могут быть ситуации, когда тщательное проектирование может приводить к алгоритмическому решению, минимизирующему как время работы, так и используемую память. Такая ситуация возникает, в частности, когда алгоритм применяет для представления входных данных эффективную с точки зрения использования памяти структуру данных, что, в свою очередь, приводит к более быстрому алгоритму. Рассмотрим, например, задачу обхода графов. Вспомним, что временна́я эффективность двух основных алгоритмов обхода — поиск в ширину и поиск в глубину — зависит от структуры данных, используемой для представления графов: она равна $\Theta\left(n^2\right)$ для представления с помощью матрицы смежности, и $\Theta\left(n+m\right)$ для представления с применением связанных списков смежности, где n и m — количество вершин и ребер, соответственно. Если входной граф разрежен, т.е. имеет малое количество ребер по отношению к количеству вершин (скажем, $m \in O\left(n\right)$), то представление с применением связанных списков смежности может оказаться более эффективным как с точки зрения используемой памяти, так и с точки зрения времени работы алгоритма. Та же ситуация возникает и при работе с разреженными матрицами или полиномами: если в таких объектах много нулей, то можно сэкономить и память, и время, если игнорировать нули в представлении объектов и при их обработке.

Во-вторых, невозможно обсуждать пространственно-временной компромисс, не упомянув о такой чрезвычайно важной области, как сжатие данных. Однако следует подчеркнуть, что при сжатии данных снижение размера является целью,

а не способом решения другой задачи. В следующей главе мы рассмотрим один алгоритм сжатия данных; читателю, заинтересовавшемуся этой темой, рекомендуем книгу [101], в которой имеется богатый выбор таких алгоритмов.

7.1 Сортировка подсчетом

В качестве первого примера использования метода улучшения входных данных рассмотрим его применение к задаче сортировки. Очевидная идея состоит в том, чтобы подсчитать для каждого элемента сортируемого списка общее количество элементов, меньших данного, и занести полученный результат в таблицу. Полученные числа указывают позиции элементов в отсортированном списке: например, если для некоторого элемента это количество равно 10, то он должен быть одиннадцатым (его индекс будет равен 10, если индексы начинаются с 0) в отсортированном массиве. Таким образом, мы можем отсортировать список, просто копируя его элементы в соответствующие позиции в новом, отсортированном списке. Такой алгоритм называется сортировкой ***подсчетом сравнений*** (comparison counting) (рис. 7.1).

Массив $A[0..5]$		62	31	84	96	19	47
Изначально	*Count*[]	0	0	0	0	0	0
После прохода $i = 0$	*Count*[]	3	0	1	1	0	0
После прохода $i = 1$	*Count*[]		1	2	2	0	1
После прохода $i = 2$	*Count*[]			4	3	0	1
После прохода $i = 3$	*Count*[]				5	0	1
После прохода $i = 4$	*Count*[]					0	2
Конечное состояние	*Count*[]	0	1	4	5	0	2

Массив $S[0..5]$	19	31	47	62	84	96

Рис. 7.1. Пример сортировки подсчетом сравнений

АЛГОРИТМ *ComparisonCountingSort* $(A[0..n-1])$

```
// Сортировка массива подсчетом сравнений
// Входные данные:  Массив A[0..n − 1] упорядочиваемых значений
// Выходные данные: Массив S[0..n − 1] элементов A,
//                  отсортированных в неубывающем порядке
for i ← 0 to n − 1 do
    Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
```

```
        Count[j] ← Count[j] + 1
    else
        Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do
    S[Count[i]] ← A[i]
return S
```

Какова же временна́я эффективность данного алгоритма? Она должна быть квадратична, поскольку алгоритм рассматривает все различные пары n-элементного массива. Более строго, количество выполнений базовой операции, сравнения $A[i] < A[j]$, равно сумме, с которой мы уже не раз встречались:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Поскольку алгоритм выполняет то же количество сравнений ключей, что и алгоритм сортировки выбором, и при этом требуется линейное количество дополнительной памяти, его трудно рекомендовать для практического применения.

Однако идея подсчета хорошо работает в ситуации, когда сортируемые элементы принадлежат небольшому множеству значений. Предположим, необходимо отсортировать список, в котором могут быть только значения 1 и 2. Вместо применения алгоритма сортировки общего назначения мы можем воспользоваться преимуществами, которые дает знание дополнительной информации о сортируемых элементах. Можно просто сканировать список, вычисляя количество единиц и двоек, а затем, на втором проходе, сделать соответствующее количество первых элементов равными 1, а остальные — равными 2. Говоря более обобщенно, если значения элементов представляют собой целые числа в границах от l (нижняя граница) до u (верхняя граница), то мы можем вычислить частоты появления каждого из этих значений и сохранить их в массиве $F[0..u-l]$. Затем первые $F[0]$ позиций в отсортированном списке должны быть заполнены значением l, следующие $F[1]$ позиций — значением $l+1$, и так далее. Все это, конечно, можно делать, только если мы можем перезаписывать данные элементы.

Рассмотрим более реалистическую ситуацию сортировки списка элементов с некоторой сопутствующей информацией, связанной с ключами, так что мы не в состоянии перезаписывать элементы списка. В этом случае мы можем копировать их в новый массив $S[0..n-1]$. Те элементы A, чьи значения ключей равны наименьшему возможному значению l, копируются в первые $F[0]$ элементов S, т.е. в позиции от 0 до $F[0]-1$, элементы со значением ключа $l+1$ — в позиции от $F[0]$ до $(F[0]+F[1])-1$, и т.д. Поскольку такие накапливаемые суммы частот в статистике называются распределением, описанный метод известен как сортировка ***подсчетом распределения*** (distribution counting).

Пример 1. Рассмотрим сортировку массива

13	11	12	13	12	12

значения которого, как нам известно, могут принадлежать множеству $\{11, 12, 13\}$ и не должны перезаписываться в процессе сортировки. Массивы частот и распределений выглядят следующим образом:

Значения массива	11	12	13
Частоты	1	3	2
Значения распределения	1	4	6

Заметим, что значения распределения указывают корректные позиции последних элементов с данным значением в отсортированном массиве (если массив индексируется начиная с 0, то значения распределения для получения корректных позиций должны быть уменьшены на 1).

Более удобно обрабатывать входной массив справа налево. Например, последнее значение равно 12, а поскольку значение распределения для него равно 4, мы помещаем 12 в позицию $4-1=3$ массива S, в котором хранится отсортированный список. Затем мы уменьшаем значение распределения 12 на 1 и переходим к следующему (в направлении справа налево) элементу исходного массива. Полностью рассматриваемый пример показан на рис. 7.2. ■

	$D[0..2]$			$S[0..5]$					
$A[5]=12$	1	**4**	6				12		
$A[4]=12$	1	**3**	6			12			
$A[3]=13$	1	2	**6**						13
$A[2]=12$	1	**2**	5		12				
$A[1]=11$	**1**	1	5	11					
$A[0]=13$	0	1	**5**					13	

Рис. 7.2. Пример сортировки подсчетом распределения. Уменьшаемые значения распределения выделены полужирным шрифтом

Вот псевдокод рассмотренного алгоритма.

АЛГОРИТМ *DistributionCounting*$(A[0..n-1], l, u)$

// Сортировка массива целых чисел из ограниченного диапазона
// при помощи сортировки подсчетом распределения
// **Входные данные:** Массив $A[0..n-1]$ целых чисел
// между l и u $(l \leqslant u)$
// **Выходные данные:** Массив $S[0..n-1]$ элементов A,

```
//                          отсортированных в неубывающем порядке
for j ← 0 to u − l do
    D[j] ← 0                                    // Инициализация частот
for i ← 0 to n − 1 do
    D[A[i] − l] ← D[A[i] − l] + 1               // Вычисление частот
for j ← 1 to u − l do
    D[j] ← D[j − 1] + D[j]                      // Получение распределения
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

В предположении фиксированного диапазона значений очевидно, что этот алгоритм линеен, поскольку выполняет два последовательных прохода по входному массиву A. Это лучший класс эффективности, чем у самых эффективных алгоритмов сортировки сравнением (слиянием, быстрой и пирамидальной). Однако очень важно помнить, что эта эффективность получена благодаря конкретному виду входных данных, для которых работает сортировка подсчетом распределения (помимо использования дополнительной памяти).

Упражнения 7.1

1. Можно ли обменять числовые значения двух переменных без использования дополнительной памяти?
2. Будет ли сортировка подсчетом сравнений корректно работать для массивов с одинаковыми значениями?
3. В предположении, что возможные входные значения ограничены множеством $\{a, b, c, d\}$, отсортируйте в алфавитном порядке при помощи алгоритма сортировки подсчетом распределения следующие входные данные:
$$b, c, d, c, b, a, a, b.$$
4. Является ли алгоритм сортировки подсчетом распределения устойчивым?
5. Разработайте однострочный алгоритм для сортировки массива размером n, содержащего различные целые значения от 1 до n.
6. ***Задача о Российском флаге*** заключается в перестановке элементов массива символов К, С, Б (означающие красный, синий и белый цвета

флага) так, чтобы первыми шли символы Б, затем — С, а последними — К. Разработайте алгоритм, решающий поставленную задачу за линейное время без привлечения дополнительной памяти.

7. ***Задача о родословной*** (ancestry problem) заключается в определении, является ли вершина u предком вершины v в данном бинарном (или, в общем случае, корневом упорядоченном) дереве из n вершин. Разработайте алгоритм с улучшением входных данных и эффективностью $O(n)$, который обеспечивает достаточное количество информации для решения этой задачи для любой пары вершин дерева за постоянное время.

8. Описанный далее метод под названием ***виртуальной инициализации*** (virtual initialization) обеспечивает эффективный способ инициализировать только некоторые из элементов данного массива $A[0..n]$ так, чтобы мы могли для каждого из его элементов за постоянное время сказать, был ли он инициализирован, и если да, то каким значением. Это делается при помощи переменной *counter*, хранящей количество инициализированных элементов A, и двух вспомогательных массивов того же размера, $B[0..n-1]$ и $C[0..n-1]$, определенных следующим образом. Элементы $B[0], \ldots, B[counter-1]$ содержат индексы тех элементов A, которые были инициализированы: $B[0]$ содержит индекс элемента A, инициализированного первым, $B[1]$ — индекс элемента A, инициализированного вторым, и т.д. Кроме того, если $A[i]$ — элемент, инициализированный k-ым $0 \leqslant k \leqslant counter-1$), то $C[i]$ равен k.

   а) Набросайте состояние массивов $A[0..7]$, $B[0..7]$ и $C[0..7]$ после трех присваиваний $A[3] \leftarrow x$; $A[7] \leftarrow z$ и $A[1] \leftarrow y$.

   б) Как при помощи этой схемы в общем случае можно определить, был ли инициализирован элемент $A[i]$, и если да, то каким значением?

9. **а)** Напишите программу для умножения двух разреженных матриц — матрицы A размером $p \times q$, и матрицы B размером $q \times p$.

   б) Напишите программу для умножения двух разреженных полиномов $p(x)$ и $q(x)$ степени m и n, соответственно.

10. Напишите программу, которая играет в крестики-нолики с человеком, путем хранения всех возможных позиций игры на доске 3×3 вместе с наилучшим ходом в данной позиции.

7.2 Улучшение входных данных в поиске подстрок

В этом разделе мы увидим, как метод улучшения входных данных можно применить к задаче поиска подстрок. Вспомним, что задача поиска подстрок состоит в поиске данной строки из m символов (именуемой ***шаблоном***, или ***образцом*** (pattern)) в более длинной строке из n символов (называемой ***текстом*** (text)). Мы рассматривали алгоритм решения данной задачи на основе грубой силы в разделе 3.2: просто проверять равенство в соответствующих парах символов из образца и текста слева направо и при обнаружении различий сдвинуть образец на один символ для выполнения новой проверки. Поскольку наибольшее количество таких проверок — $n - m + 1$ и в наихудшем случае при каждой проверке требуется выполнить m сравнений, общее количество сравнений в наихудшем случае равно $m(n - m + 1)$, так что производительность алгоритма грубой силы в наихудшем случае равна $\Theta(nm)$. В среднем, однако, можно ожидать, что при каждой проверке будет сделано только несколько сравнений; в самом деле, для случайного естественного текста эффективность в среднем случае превращается в $\Theta(n)$.

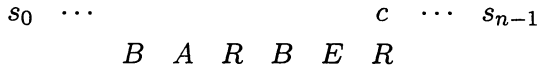
Высокая производительность алгоритма на основе грубой силы в среднем случае — это и хорошо, и плохо одновременно. Это хорошо с практической точки зрения, поскольку делает алгоритм на основе грубой силы вполне достойным кандидатом для практического применения, в особенности при коротких образцах. Но это плохо для теоретика, который хочет найти более быстрый алгоритм. Тем не менее был найден ряд более быстрых алгоритмов, большинство из которых использует идею улучшения входных данных: предварительная обработка образца для получения некоторой информации о нем, сохранение этой информации в таблице, а затем ее использование при реальном поиске этого образца в данном тексте. Эта идея лежит в основе двух наиболее известных алгоритмов этого типа — алгоритме Кнута–Морриса–Пратта [68] и алгоритме Бойера–Мура [21].

Основное различие между этими двумя алгоритмами лежит в способе сравнивания символов образца и текста: алгоритм Кнута–Морриса–Пратта делает это слева направо, а алгоритм Бойера–Мура — справа налево. Поскольку последний способ приводит к более простому алгоритму, мы рассмотрим здесь именно его. (Заметим, что алгоритм Бойера–Мура начинает с выравнивания образца по первому символу текста; если первая проверка неудачна, образец сдвигается вправо. Проверка же выполняется путем сравнения символов образца и текста справа налево, начиная с последнего символа образца.) Несмотря на то что идея, лежащая в основе алгоритма Бойера–Мура, проста, этого не скажешь о его реализации. Поэтому мы начнем рассмотрение с упрощенной версии алгоритма Бойера–Мура, предложенной Р. Хорспулом (R. Horspool) [55]. Несмотря на простоту, алгоритм

Хорспула при работе со случайными строками столь же эффективен, как и алгоритм Бойера–Мура.

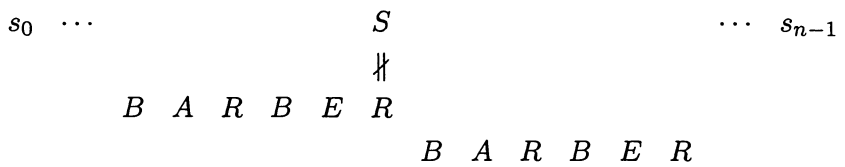
Алгоритм Хорспула

Рассмотрим в качестве примера поиск подстроки *BARBER* в некотором тексте:

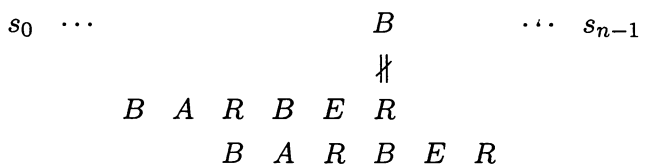


Мы сравниваем соответствующие пары символов из образца и текста, начиная с последнего символа R в образце и перемещаясь справа налево. Если все символы образца соответствуют символам текста, искомая подстрока найдена (после этого поиск может либо завершиться, либо продолжиться, если требуется найти другое вхождение подстроки в текст). Если же мы встретили несоответствие, то должны сдвинуть образец вправо. Понятно, что хотелось бы сдвинуть его как можно сильнее, но без риска пропустить возможное вхождение подстроки в текст. Алгоритм Хорспула определяет величину такого сдвига, рассматривая символ c текста, который при выравнивании находится напротив последнего символа образца. В общем случае могут возникнуть четыре разные ситуации.

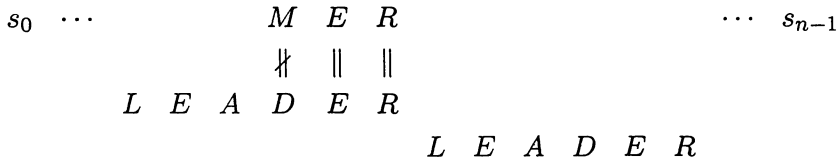
Случай 1. Если символа c в образце нет (например, если c в данном примере представляет собой символ S), то смело можно сдвигать образец на всю его длину (при сдвиге меньшей величины против символа c окажется некоторый символ образца, который заведомо не может быть таким же, как c, которого в образце нет):



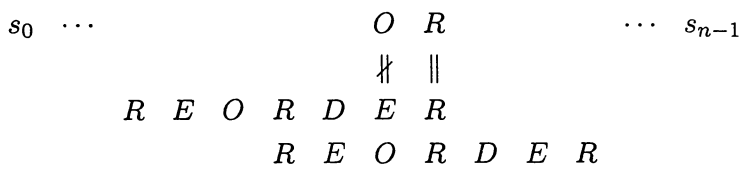
Случай 2. Если символ c в образце есть, но он не последний (например, символ B в нашем примере), то сдвиг должен выровнять образец так, чтобы напротив c в тексте было первое справа вхождение этого символа в образец:



Случай 3. Если c — последний символ образца и среди остальных $m-1$ символов образца такого символа больше нет, то сдвиг должен быть подобен сдвигу в случае 1 — образец следует сдвинуть на всю длину m:



Случай 4. И, наконец, если c — последний символ образца и среди остальных $m-1$ символов образца имеются другие вхождения этого символа, то сдвиг должен быть подобен случаю 2 — крайнее справа вхождение c среди остальных $m-1$ символов образца должно располагаться напротив символа c в тексте:



Эти примеры ясно демонстрируют, что сравнение символов справа налево может привести к бóльшим сдвигам, чем сдвиги на одну позицию в алгоритме на основе грубой силы. Однако если такой алгоритм будет просматривать все символы образца при каждой проверке, то все его преимущество будет потеряно. К счастью, идея улучшения входных данных делает такой просмотр при каждой проверке излишним. Мы можем предварительно вычислить величины сдвигов для каждого возможного символа и хранить их в таблице. Такая таблица индексируется всеми возможными символами, которые могут встретиться в тексте, включая, для естественных языков, пробелы, символы пунктуации и другие специальные символы (заметим, что никакой иной информации о тексте, в котором будет выполняться поиск, не требуется). Элементы таблицы заполняются величинами сдвигов. В частности, для каждого символа c мы можем вычислить величину сдвига по формуле

$$t(c) = \begin{cases} \text{Длина образца } m, \text{ если } c \text{ нет} \\ \text{среди первых } m-1 \text{ символов образца} \\ \\ \text{В противном случае — расстояние от крайнего} \\ \text{справа символа } c \text{ среди первых } m-1 \text{ символов} \\ \text{образца до его последнего символа} \end{cases} \tag{7.1}$$

Например, для образца *BARBER* все элементы таблицы равны 6, за исключением элементов для символов E, B, R и A, для которых они равны 1, 2, 3 и 4, соответственно.

Вот простой алгоритм для вычисления элементов таблицы сдвигов. Все значения в таблице инициализируются длиной образца m, а затем выполняется сканирование образца слева направо с выполнением $m-1$ раз следующих действий: для j-го символа образца $0 \leqslant j \leqslant m-2$) соответствующий ему элемент таблицы перезаписывается значением $m-1-j$, которое представляет собой расстояние от символа до правого конца образца. Заметим, что, поскольку алгоритм сканирует образец слева направо, последняя перезапись выполняется, когда встречается самое правое вхождение символа в образец, т.е. именно так, как требуется.

Алгоритм $ShiftTable(P[0..m-1])$

```
// Заполняет таблицу сдвигов, использующуюся алгоритмами
// Хорспула и Бойера–Мура
// Входные данные:  Образец P[0..m − 1] и алфавит возможных
//                  символов
// Выходные данные: Таблица Table[0..size − 1],
//                  индексированная символами алфавита
//                  и заполненная величинами сдвигов,
//                  вычисленных по формуле (7.1)
Инициализация всех элементов Table значениями m
for j ← 0 to m − 2 do
  Table[P[j]] ← m − 1 − j
return Table
```

А вот как выглядит алгоритм Хорспула целиком.

Алгоритм Хорспула

Шаг 1. Для данного образца длиной m и алфавита, используемого в тексте и образце, описанным выше способом строится таблица сдвигов.

Шаг 2. Выравниваем начало образца с началом текста.

Шаг 3. До тех пор, пока не будет найдена искомая подстрока или пока образец не достигнет последнего символа текста, повторяем следующие действия. Начиная с последнего символа образца, сравниваем соответствующие символы в шаблоне и тексте, пока не будет установлено равенство всех m символов (при этом поиск прекращается) либо пока не будет обнаружена пара разных символов. В последнем случае находим элемент $t(c)$ из таблицы сдвигов, где c — символ текста, находящийся напротив последнего символа образца, и сдвигаем образец вдоль текста на $t(c)$ символов вправо.

Вот как выглядит псевдокод алгоритма Хорспула.

Алгоритм $HorspoolMatching\,(P\,[0..m-1]\,,T\,[0..n-1])$

```
// Реализация алгоритма Хорспула поиска подстрок
// Входные данные:  Образец P[0..m − 1] и текст T[0..n − 1]
// Выходные данные: Индекс левого конца первой найденной
//                  подстроки или −1, если искомой подстроки
//                  в тексте нет
ShiftTable(P[0..m − 1])     // Генерация таблицы сдвигов Table
i ← m − 1                   // Позиция правого конца образца
while i ⩽ n − 1 do
    k ← 0                   // Количество совпадающих символов
    while k ⩽ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else
        i ← i + Table[T[i]]
return −1
```

Пример 1. Рассмотрим в качестве законченного приложения алгоритма Хорспула поиск образца $BARBER$ в тексте, состоящем из английских букв и пробелов (которые указаны символами подчеркивания). Таблица сдвигов для этого случая, как уже говорилось, выглядит следующим образом:

Символ c	A	B	C	D	E	F	...	R	...	Z	_
Сдвиг $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Реальный поиск в конкретном тексте выглядит так:

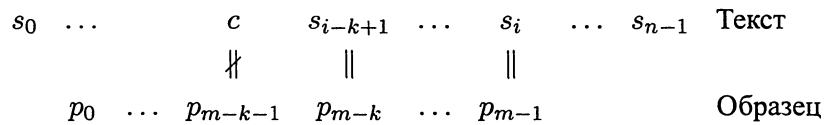
```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R               B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                 B A R B E R
```

■

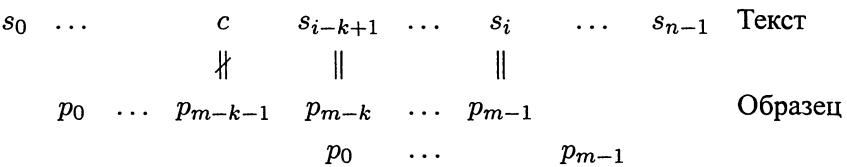
На простом примере можно продемонстрировать, что эффективность алгоритма Хорспула в наихудшем случае составляет $\Theta(nm)$ (см. упражнение 7.2.4). Однако для случайных текстов эффективность алгоритма Хорспула равна $\Theta(n)$. Таким образом, хотя алгоритм Хорспула принадлежит к тому же классу эффективности, что и алгоритм, основанный на грубой силе, очевидно, что в среднем он превосходит последний. В действительности, как уже упоминалось, зачастую он столь же эффективен, как и его более сложный предшественник, открытый Р. Бойером (R. Boyer) и Дж. Муром (J. Moore).

Алгоритм Бойера–Мура

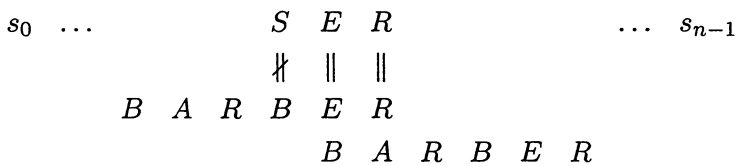
В этом разделе мы рассмотрим работу алгоритма Бойера–Мура. Если первое сравнение крайнего справа символа в шаблоне с соответствующим символом c в тексте показывает, что они различны, алгоритм работает точно так же, как и алгоритм Хорспула, т.е. выполняется сдвиг вправо на количество символов, которое определяется таблицей сдвигов (которая вычисляется в точности так же, как и для алгоритма Хорспула). Однако алгоритмы Бойера–Мура и Хорспула работают по-разному, если некоторое положительное количество k ($0 < k < m$) символов образца совпадает с символами текста, перед тем как встретится первое отличие:



В этой ситуации алгоритм Бойера–Мура определяет величину сдвига, рассматривая две величины. Величина первого сдвига определяется символом текста c, который первым не соответствует символу образца при сравнении справа налево. Назовем его ***сдвигом несовпадающего символа*** (bad-symbol shift). Рассмотрение данного сдвига выполняется так же, как и сдвига в алгоритме Хорспула. Если символ c не входит в образец, мы сдвигаем образец так, чтобы символ c вышел за пределы образца. Значение этого сдвига легко вычислить по формуле $t_1(c) - k$, где $t_1(c)$ — элемент таблицы сдвигов, использующейся в алгоритме Хорспула, а k — количество совпадающих символов:



Например, если мы ищем образец $BARBER$ в некотором тексте и находим совпадение двух последних символов перед обнаружением в тексте символа S, то можем сдвинуть образец на $t_1(S) - 2 = 6 - 2 = 4$ позиции:



Та же формула используется и в случае, когда несовпадающий символ c имеется в образце и при этом $t_1(c) - k > 0$. Например, если мы ищем образец $BARBER$

в некотором тексте и находим совпадение двух последних символов перед обнаружением в тексте символа A, то можем сдвинуть образец на $t_1(A) - 2 = 4 - 2 = 2$ позиции:

$$\begin{array}{cccccccccc} s_0 & \ldots & & & A & E & R & & \ldots & s_{n-1} \\ & & & & \nmid & \| & \| & & & \\ & & B & A & R & B & E & R & & \\ & & & & B & A & R & B & E & R \end{array}$$

Если $t_1(c) - k \leqslant 0$, очевидно, что нельзя сдвигать образец на нулевое или отрицательное количество позиций. Вместо этого мы просто применяем метод грубой силы и сдвигаем образец на одну позицию вправо. Итак, сдвиг несовпадающего символа d_1, вычисляемый в алгоритме Бойера–Мура, равен либо $t_1(c) - k$, если эта величина положительна, либо 1, если она отрицательна или равна нулю. Сказанное можно выразить простой компактной формулой:

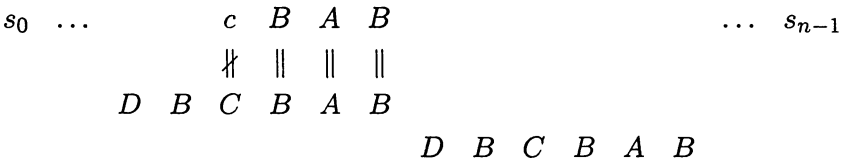
$$d_1 = \max\{t_1(c) - k, 1\}. \tag{7.2}$$

Сдвиг второго типа определяется совпадением последних $k > 0$ символов образца. Мы будем называть конечную часть образца его суффиксом длиной k и обозначать как $suff(k)$. Соответственно, этот тип сдвига мы будем называть ***сдвигом совпадающего суффикса*** (good-suffix shift). Сейчас мы применим рассуждения, которые позволили нам заполнить таблицу сдвигов несовпадающих символов на основании одного символа c, к построению таблицы сдвигов совпадающих суффиксов на основании суффиксов с длинами $1, \ldots, m-1$.

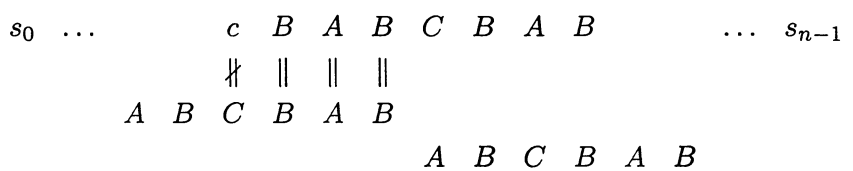
Начнем с рассмотрения случая, когда в образце встречается последовательность символов такая же, как и $suff(k)$, но не являющаяся суффиксом; говоря более строго, имеется другая последовательность $suff(k)$, которую предваряет символ, отличный от символа, предваряющего последнюю такую последовательность символов (было бы бесполезно сдвигать образец так, чтобы на место предыдущей последовательности $suff(k)$ попала новая последовательность с таким же предшествующим символом — в этой ситуации мы бы просто повторили неудачную попытку сравнения). В таком случае мы можем сдвинуть образец на расстояние d_2 между второй справа последовательностью $suff(k)$ (не предваряемой тем же символом, что и последняя) и такой же последовательностью, крайней справа в образце. Например, для образца $ABCBAB$ эти расстояния для $k = 1$ и 2 будут равны, соответственно, 4 и 6:

k	Образец	d_2
1	$ABC\overline{B}A\underline{B}$	2
2	$\overline{AB}CB\underline{AB}$	4

Что следует предпринять, если в образце нет другой последовательности $suff(k)$, которой предшествует символ, отличный от символа, предшествующего суффиксу $suff(k)$? В большинстве случаев мы можем сдвинуть образец на всю его длину m. Например, для образца DBCBAB и $k = 3$ мы можем выполнить сдвиг образца на всю его длину — 6 символов:



К сожалению, такой сдвиг образца на всю его длину m, если в образце нет другой последовательности $suff(k)$, которой предшествует символ, отличный от символа, предшествующего суффиксу $suff(k)$, не всегда корректен. Например, для образца $ABCBAB$ и $k = 3$ сдвиг на 6 может привести к пропуску подстроки, которая начинается с символов AB, расположенных в тексте напротив двух последних символов образца:



Заметим, что сдвиг на 6 символов корректен для образца $DBCBAB$, но не для образца $ABCBAB$, поскольку у него имеется префикс AB, совпадающий с суффиксом. Чтобы избежать такого некорректного сдвига на основании суффикса длиной k, следует найти наибольший префикс длиной $l < k$, совпадающий с суффиксом той же длины l. Если такой префикс имеется, величина сдвига d_2 вычисляется как расстояние между префиксом и суффиксом; в противном случае d_2 устанавливается равным длине образца m. В качестве примера рассмотрим полный список значений d_2 — таблицу сдвигов совпадающих суффиксов алгоритма Бойера–Мура — для образца $ABCBAB$:

k	Образец	d_2
1	$ABC\overline{B}A\underline{B}$	2
2	$\overline{AB}CB\underline{AB}$	4
3	$\overline{AB}C\underline{BAB}$	4
4	$\overline{AB}\underline{CBAB}$	4
5	$\overline{A\underline{B}}\underline{CBAB}$	4

Теперь мы готовы к тому, чтобы полностью описать алгоритм Бойера–Мура.

Алгоритм Бойера–Мура

Шаг 1. Для данного образца и используемого алфавита описанным выше способом строится таблица сдвигов несовпадающих символов.

Шаг 2. Для данного образца описанным выше способом строится таблица сдвигов совпадающих суффиксов.

Шаг 3. Выравниваем начало образца с началом текста.

Шаг 3. До тех пор, пока не будет найдена искомая подстрока или пока образец не достигнет последнего символа текста, повторяем следующие действия. Начиная с последнего символа образца, сравниваем соответствующие символы в шаблоне и тексте, пока не будет установлено равенство всех m символов (при этом поиск прекращается) либо пока не будет обнаружена пара разных символов после $k \geqslant 0$ совпадающих символов. В последнем случае находим элемент $t_1(c)$ из таблицы сдвигов несовпадающих символов, где c — не совпавший символ текста. Если $k > 0$, выбираем, кроме того, соответствующее значение d_2 из таблицы совпавших суффиксов. Сдвигаем образец вправо на количество позиций, которое вычисляется по формуле

$$d = \begin{cases} d_1 & \text{если } k = 0, \\ \max\{d_1, d_2\} & \text{если } k > 0, \end{cases} \tag{7.3}$$

где $d_1 = \max\{t_1(c) - k, 1\}$.

Сдвиг на максимальное из двух допустимых значений логичен. Оба значения получены из наблюдений (одно за не совпавшим символом, второе — за группой совпавших символов), что меньшие значения сдвигов не могут привести к искомой подстроке в тексте. Поскольку нас интересуют как можно бóльшие сдвиги без возможной потери искомой подстроки, выбираем большее из найденных значений.

Пример 2. В качестве завершенного примера давайте рассмотрим поиск подстроки $BAOBAB$ в тексте, состоящем из английских букв и пробелов. Таблица сдвигов несовпадающих символов выглядит следующим образом:

c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

Таблица сдвигов совпадающих суффиксов имеет такой вид:

k	Образец	d_2
1	$BAO\overline{B}A\underline{B}$	2
2	$\overline{B}AOB\underline{AB}$	5
3	$\overline{B}AO\underline{BAB}$	5
4	$\overline{B}A\underline{OBAB}$	5
5	$\overline{B}\underline{AOBAB}$	5

Поиск при помощи алгоритма Бойера–Мура с использованием приведенных таблиц показан на рис. 7.3. После того как последний символ образца B при сравнении не совпадает с символом K в тексте, алгоритм получает значение $t_1(K) = 6$ из таблицы сдвигов несовпадающих символов и перемещает образец на расстояние $d_1 = \max\{t_1(K) - 0, 1\} = 6$ позиций вправо. Новая попытка находит два совпадающих символа, и после неудачного третьего сравнения символов алгоритм получает $t_1(\_) = 6$ из таблицы сдвигов несовпадающих символов и $d_2 = 5$ — из таблицы сдвигов совпадающих суффиксов и сдвигает образец вправо на $\max\{d_1, d_2\} = \max\{6 - 2, 5\} = 5$ символов. Обратите внимание, что на этой итерации правило совпадающих суффиксов дает большее значение сдвига, чем правило последнего несовпадающего символа.

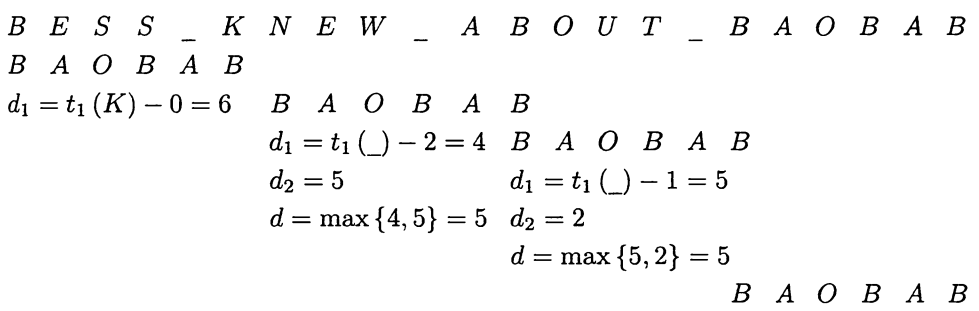


Рис. 7.3. Пример поиска подстроки при помощи алгоритма Бойера–Мура

Очередная проверка успешно находит одну пару совпадающих символов B. После неудачности следующего сравнения с пробелом в тексте алгоритм получает из таблицы сдвигов несовпадающих символов значение $t_1(\_) = 6$, а из таблицы сдвигов совпадающих суффиксов — $d_2 = 2$ и затем сдвигает образец на $\max\{d_1, d_2\} = \max\{6 - 2, 5\} = 5$ символов. На этой итерации, как видите, правило совпадающих суффиксов дает меньшее значение сдвига, чем правило последнего несовпадающего символа. После выполнения сдвига очередная проверка дает совпадение всех шести символов образца с соответствующими символами текста. ■

При поиске первого вхождения образца в текст эффективность алгоритма Бойера–Мура в наихудшем случае оказывается линейной. Несмотря на скорость данного алгоритма, в особенности в случае больших алфавитов (по отношению к длине образца), многие, работая с текстами на естественных языках, предпочитают упрощенные версии данного алгоритма, такие как алгоритм Хорспула.

Упражнения 7.2

1. Примените алгоритм Хорспула к поиску подстроки *BAOBAB* в тексте *BESS_KNEW_ABOUT_BAOBABS*.
2. Рассмотрите задачу поиска генов в последовательности ДНК с использованием алгоритма Хорспула. Последовательность ДНК представляет собой текст на алфавите $\{A, C, G, T\}$, а ген или отрезок гена — образец поиска.
   а) Постройте таблицу сдвигов для отрезка гена *TCCTATTCTT*.
   б) Примените алгоритм Хорспула для поиска этого образца в последовательности ДНК

   TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT.

3. Сколько сравнений символов будет выполнено алгоритмом Хорспула при поиске каждого из следующих образцов в тексте, состоящем из 1000 нулей?
   а) 00001 **б)** 10000 **в)** 01010
4. Для поиска образца длиной m в тексте длиной n ($n \geqslant m$) при помощи алгоритма Хорспула приведите пример
   а) наихудшего случая **б)** наилучшего случая
5. Может ли алгоритм Хорспула выполнить большее количество сравнений символов, чем алгоритм на основе грубой силы, при поиске того же образца в том же тексте?
6. Если алгоритм Хорспула находит искомую подстроку, какой сдвиг он должен сделать для продолжения поиска следующего вхождения подстроки в текст?
7. Сколько сравнений символов будет выполнено алгоритмом Бойера–Мура при поиске каждого из следующих образцов в тексте, состоящем из 1000 нулей?
   а) 00001 **б)** 10000 **в)** 01010

8. **а)** Будет ли корректно работать алгоритм Бойера–Мура при использовании только лишь таблицы сдвигов несовпадающих символов для определения величины очередного сдвига образца?
   б) Будет ли корректно работать алгоритм Бойера–Мура при использовании только лишь таблицы сдвигов совпадающих суффиксов для определения величины очередного сдвига образца?
9. **а)** Если последний символ образца и соответствующий ему символ текста совпадают, должен ли алгоритм Хорспула проверять остальные символы справа налево или он может проверять их слева направо?
   б) Ответьте на тот же вопрос для алгоритма Бойера–Мура.
10. Реализуйте алгоритмы Хорспула, Бойера–Мура и алгоритм на основе грубой силы из раздела 3.2 на языке программирования по вашему выбору и проведите эксперимент по сравнению их эффективности для поиска
    а) случайного бинарного образца в случайном бинарном тексте;
    б) случайный образец на естественном языке в случайном тексте на том же естественном языке.

7.3 Хеширование

В этом разделе мы рассмотрим очень эффективный способ реализации словарей. Вспомним, что словарем называется абстрактный тип данных, представляющий собой множество с операциями поиска, вставки и удаления, определенных над его элементами. Элементы такого множества могут иметь любую природу — быть числами, символами некоторого алфавита, строками и т.д. На практике наиболее важным случаем являются записи (записи с информацией о студентах в институте, о гражданах — в муниципальных органах, о книгах — в библиотеке и т.п.).

Обычно записи включают в себя несколько полей, каждое из которых отвечает за хранение определенного рода информации об объекте, представленном записью. Например, запись о студенте может содержать поля с идентификатором студента, его фамилией, датой рождения, полом, домашним адресом и т.п. Среди полей записи имеется по крайней мере одно, именуемое ***ключом*** (key) и используемое для идентификации объектов, представленных записями (например, идентификатор студента). В данном разделе мы полагаем, что должны разработать словарь из n записей с ключами $K_1, K_2, \ldots, K_n$.

Хеширование (hashing) основано на идее распределения ключей в одномерном массива $H[0..m-1]$, называющемся ***хеш-таблицей*** (hash table). Распределение осуществляется путем вычисления для каждого ключа значения некоторой предопределенной функции h, которая называется ***хеш-функцией*** (hash function).

Эта функция назначает каждому из ключей ***хеш-адрес*** (hash address), который представляет собой целое число от 0 до $m-1$. Например, если ключи представляют собой неотрицательные целые числа, то хеш-функция может иметь вид $h(K) = K \bmod m$ (очевидно, что остаток от деления на m всегда находится в диапазоне от 0 до $m-1$). Если ключи — символы некоторого алфавита, то сначала можно назначить символам их позиции в алфавите (обозначаемые как $ord(K)$), а затем применить к этим значениям ту же функцию, что и для целых чисел. Наконец, если K — символьная строка $c_0c_1 \ldots c_{s-1}$, мы можем использовать (в качестве простейшей и не очень разумной возможности) значение $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m$; существенно лучшим вариантом является вычисление $h(K)$ по формуле[2]

$$h \leftarrow 0;\ \textbf{for}\ i \leftarrow 0\ \textbf{to}\ s-1\ \textbf{do}\ h \leftarrow (h \cdot C + ord(c_i)) \bmod m,$$

где C — константа, бóльшая, чем любое из значений $ord(c_i)$.

В общем случае хеш-функция должна удовлетворять двум несколько противоречивым требованиям:

- хеш-функция должна распределять ключи по ячейкам хеш-таблицы как можно более равномерно. (Исходя из этого требования m обычно выбирается простым. Это же требование делает желательной для большинства приложений зависимость хеш-функции от всех битов ключа, а не только от некоторых из них.)
- Хеш-функция должна легко вычисляться.

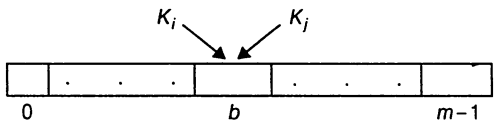


Рис. 7.4. Коллизия двух ключей при хешировании: $h(K_i) = h(K_j)$

Очевидно, что, выбрав размер хеш-таблицы m меньшим, чем количество ключей n, мы обречены на ***коллизии*** (collision) — ситуации, когда два (или более) ключей хешируются в одну и ту же ячейку хеш-таблицы (рис. 7.4). Но коллизий следует ожидать, даже если m значительно больше n (см. упражнение 7.3.5). В действительности в наихудшем случае все ключи могут быть хешированы в одну ячейку хеш-таблицы. К счастью, при соответствующем выборе размера хеш-таблицы и хорошей хеш-функции такая ситуация встречается очень редко. Тем не менее любая схема хеширования должна иметь механизм разрешения коллизий.

[2]Эта формула получена путем рассмотрения $ord(c_i)$ как цифр числа в системе счисления по основанию C, вычисления его десятичного значения по схеме Горнера и поиска остатка от деления получившегося числа на m.

Этот механизм различен в двух основных версиях хеширования — ***открытом хешировании*** (open hashing) (его называют также ***хешированием с раздельными цепочками*** (separate chaining)) и ***закрытом хешировании*** (closed hashing) (называемом также ***хешированием с открытой адресацией*** (open addressing)).

Открытое хеширование (раздельные цепочки)

При открытом хешировании ключи хранятся в связанных списках, присоединенных к ячейкам хеш-таблицы. Каждый список содержит все ключи, хешированные в данную ячейку. Рассмотрим, например, следующий список слов:

$$A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED.$$

В качестве хеш-функции мы будем использовать простую функцию для строк, упоминавшуюся выше, т.е. суммировать позиции в алфавите букв, составляющих слова, и вычислять остаток от деления этой суммы на 13.

Начнем с пустой таблицы. Первый ключ — слово A; его хеш-значение $h(A) = 1 \bmod 13 = 1$. Второй ключ — слово $FOOL$ — попадает в девятую ячейку (поскольку $(6 + 15 + 15 + 12) \bmod 13 = 9$), и т.д. Окончательный результат этого процесса показан на рис. 7.5. Обратите внимание на коллизию ключей ARE и $SOON$ (вызванную тем, что $h(ARE) = (1 + 18 + 5) \bmod 13 = 11$ и $h(SOON) = (19 + 15 + 15 + 14) \bmod 13 = 11$).

Ключи	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
Хеш-адреса	1	9	6	10	7	11	11	12

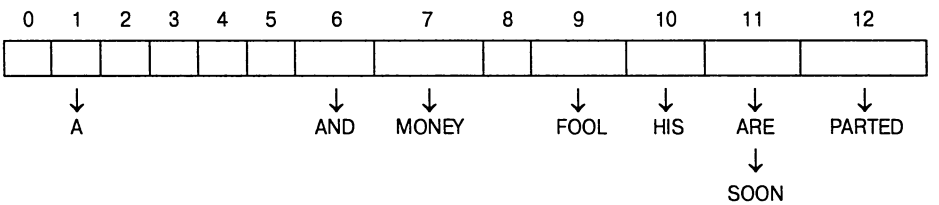


Рис. 7.5. Пример хеш-таблицы с раздельными цепочками

Каким образом осуществляется поиск в такой таблице связанных списков? Для этого к ключу поиска применяется та же процедура, что и при создании таблицы. Для иллюстрации: если мы хотим найти в приведенной на рис. 7.5 хеш-таблице слово *KID*, то должны начать с вычисления для него хеш-функции: $h(KID) = 11$. Поскольку список, присоединенный к ячейке 11, не пуст, он может содержать соответствующий ключ. Однако из-за возможных коллизий мы не можем сказать, так ли это, пока не обойдем весь список. После сравнения слова KID сначала

со словом ARE, а затем со словом $SOON$ мы убеждаемся, что искомого слова в таблице нет.

В общем случае эффективность поиска зависит от длины связанных списков, которая, в свою очередь, зависит от размеров словаря и таблицы и качества хеш-функции. Если хеш-функция распределяет n ключей по m ячейкам равномерно (или практически равномерно), то в каждом списке будет содержаться примерно около n/m ключей. Отношение $\alpha = n/m$ называется ***коэффициентом заполнения*** (load factor) хеш-таблицы и играет ключевую роль в эффективности хеширования. В частности, среднее количество проверяемых узлов цепочек при успешном поиске S и при неудачном U равны, соответственно

$$S \approx 1 + \frac{\alpha}{2} \text{ и } U = \alpha \tag{7.4}$$

(при стандартных предположениях о поиске случайно выбранного элемента и хеш-функции, равномерно распределяющей ключи по ячейкам таблицы). Полученные результаты вполне естественны и, само собой, практически идентичны поиску в связанном списке — все, чего мы достигаем при помощи хеширования, — это снижение среднего размера связанного списка в m раз.

Обычно желательно, чтобы коэффициент заполнения был близок к 1. Слишком малый коэффициент заполнения означает множество пустых списков и неэффективное использование памяти; слишком большой — более длинные списки и более продолжительный поиск. Но если коэффициент заполнения близок к 1, мы получаем наиболее эффективную схему, которая позволяет находить заданный ключ путем одного или двух сравнений в среднем. Конечно, в дополнение к сравнениям мы должны затратить некоторое время на вычисление хеш-функции, но это операция с постоянным временем выполнения, не зависящим от n и m. Заметим, что мы получаем такую замечательную эффективность не только благодаря хитроумности самого метода, но и ценой излишнего потребления памяти.

Две другие словарные операции — вставка и удаление — практически идентичны поиску. Вставка обычно делается в конец списка (однако в упражнении 7.3.6 вы найдете возможные модификации этого правила). Удаление выполняется путем поиска удаляемого ключа с последующим удалением его из связанного списка. Следовательно, эффективность этих операций равна эффективности поиска, и все они в среднем случае имеют эффективность $\Theta(1)$, если количество ключей n примерно равно размеру хеш-таблицы m.

Закрытое хеширование (открытая адресация)

В случае закрытого хеширования все ключи хранятся в хеш-таблице, без использования связанных списков (само собой, это приводит к требованию, чтобы размер хеш-таблицы m был не меньше количества ключей n). Для разрешения коллизий могут применяться разные стратегии. Простейшая из них — ***линейное***

исследование (linear probing), когда в случае коллизии ячейки проверяются одна за другой. Если ячейка пуста, новый ключ вносится в нее; если заполнена — проверяется ячейка, следующая за ней. Если при проверке достигается конец таблицы, поиск переходит к первой ячейке таблицы, которая рассматривается как циклический массив. Этот метод проиллюстрирован на рис. 7.6 с применением тех же слов, которые мы использовали для иллюстрации хеширования с раздельными цепочками (здесь мы используем ту же хеш-функцию, что и ранее).

Ключи	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
Хеш-адреса	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Рис. 7.6. Пример хеш-таблицы, построенной с использованием линейного исследования

Для поиска заданного ключа K мы начинаем с вычисления $h(K)$, где h — хеш-функция, использующаяся при построении таблицы.. Если ячейка $h(K)$ пуста, поиск неудачен. Если ячейка не пуста, мы сравниваем K с ключом, хранящимся в ячейке. Если они равны, то искомый ключ найден; если нет — то мы переходим к следующей ячейке и повторяем описанные действия до тех пор, пока не встретим искомый ключ (успешный поиск) или пустую ячейку (неудачный поиск). Например, если мы ищем слово LIT в таблице на рис. 7.6, то получим $h(LIT) = (12 + 9 + 20) \bmod 13 = 2$ и, поскольку ячейка 2 пуста, тут же прекратим поиск. При поиске слова KID мы находим $h(KID) = (11 + 9 + 4) \bmod 13 = 11$, так что сравниваем KID с ключами ARE, $SOON$, $PARTED$, прежде чем сможем объявить поиск неудачным.

В то время как операции поиска и вставки в такой версии хеширования весьма просты, этого нельзя сказать об удалении. Например, удалив ключ ARE из последнего состояния таблицы на рис. 7.6, мы больше не сможем обнаружить в ней слово $SOON$. Судите сами: $h(SOON) = 11$, и алгоритм поиска, обнаружив пустую ячейку с этим индексом, сообщит об отсутствии такого слова в таблице. Простейшим решением проблемы является "отложенное удаление", т.е. ранее за-

нятая ячейка помечается специальным образом, чтобы можно было отличить ее от ячеек, которые никогда не были заняты.

Математический анализ линейного исследования — существенно более сложная задача, чем анализ хеширования с раздельными цепочками.[3] Упрощенная версия полученных при таком анализе результатов гласит, что среднее количество обращений к хеш-таблице с коэффициентом заполнения α в случае успешного и неудачного поиска равно соответственно

$$S \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \text{ и } U \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) \tag{7.5}$$

(точность этого приближения растет с ростом размера хеш-таблицы). Получающиеся числа на удивление малы даже для плотно заполненных таблиц: так, для достаточно больших значений α получаются следующие величины:

α	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

По мере заполнения хеш-таблицы производительность линейного исследования ухудшается из-за эффекта кластеризации. ***Кластером*** (cluster) в линейном исследовании называется последовательность соседних заполненных ячеек (с возможным переходом из последней ячейки таблицы в первую). Например, в окончательном состоянии таблицы на рис. 7.6 имеются два кластера. В хешировании кластеры представляют собой отрицательное явление, поскольку снижают эффективность словарных операций. Заметим также, что с ростом кластеров растет вероятность того, что новый элемент будет добавлен к кластеру. Кроме того, растет и вероятность слияния кластеров при вставке нового ключа, что еще больше увеличивает кластеризацию.

Для снижения эффекта кластеризации был предложен ряд других стратегий разрешения коллизий. Одна из наиболее важных среди них — ***двойное хеширование*** (double hashing). В этой схеме для определения фиксированного значения шага последовательности исследований при коллизии в ячейке $l = h(K)$ используется другая хеш-функция $s(K)$:

$$(l + s(K)) \bmod m, (l + 2s(K)) \bmod m, \ldots. \tag{7.6}$$

[3]Эта задача была решена в 1962 году молодым аспирантом Дональдом Кнутом (Donald Knuth), который ныне известен как один из ведущих ученых в области кибернетики. Его многотомник *Искусство программирования* [65, 66, 67] остается одной из наиболее полных и важных книг об алгоритмах.

Для каждой ячейки таблицы, исследуемой при помощи такой последовательности (7.6), величина шага $s(K)$ и размер таблицы m должны быть взаимно простыми величинами, т.е. не должны иметь общих делителей, отличных от 1 (это условие выполняется автоматически, если m — простое число (и $s(K) \neq m$)). Вот некоторые функции, рекомендуемые в литературе: $s(k) = m-2-k \bmod (m-2)$, $s(k) = 8 - (k \bmod 8)$ — для небольших таблиц и $s(k) = k \bmod 97 + 1$ — для больших (см. [102, 103]). Математический анализ двойного хеширования очень сложен. Некоторые частичные результаты и значительный практический опыт свидетельствуют о том, что при выборе хороших хеш-функций — как первичной, так и вторичной — двойное хеширование превосходит метод линейных исследований. Однако производительность этого метода также падает по мере приближения таблицы к полностью заполненному состоянию. Единственным решением в такой ситуации является новое хеширование: текущая таблица сканируется, и все ключи из нее хешируются в новую таблицу большего размера.

Со времени открытия хеширования в 1950-х годах исследователями из IBM оно нашло множество важных применений. В частности, хеширование стало стандартным методом для хранения таблиц символов — таблицы символов в компьютерной программе, генерируемой в процессе компиляции последней. С небольшими модификациями метод хеширования, как было доказано, пригоден для хранения очень больших словарей на диске; эта версия хеширования называется ***расширяемым хешированием*** (extensible hashing). Поскольку обращение к диску существенно дороже исследований в оперативной памяти, предпочтительно выполнение существенно большего количества исследований, чем обращений к диску. Соответственно, положения, вычисляемые при помощи хеш-функции в расширяемом хешировании, указывают адрес на диске, где находится ***блок*** (bucket), в котором может храниться до b ключей. После того как определен блок для искомого ключа, все ключи блока считываются в оперативную память и среди них выполняется поиск искомого ключа. В следующем разделе мы рассмотрим B-деревья, главный альтернативный способ хранения больших словарей.

Упражнения 7.3

1. Для входных данных 30, 20, 56, 75, 31, 19 и хеш-функции $h(K) = K \bmod 11$

а) постройте открытую хеш-таблицу;

б) найдите наибольшее количество сравнений ключей при успешном поиске в полученной таблице;

в) найдите среднее количество сравнений ключей при успешном поиске в полученной таблице.

2. Для входных данных 30, 20, 56, 75, 31, 19 и хеш-функции $h(K) = K \bmod 11$
   а) постройте закрытую хеш-таблицу;
   б) найдите наибольшее количество сравнений ключей при успешном поиске в полученной таблице; найдите среднее количество сравнений ключей при успешном поиске в полученной таблице.
3. Чем плоха идея хеш-функции, зависящей только от одной (например, первой) буквы слова естественного языка?
4. Найдите вероятность того, что все n ключей будут хешированы в одну ячейку хеш-таблицы размером m, если хеш-функция распределяет ключи равномерно по всем ячейкам таблицы.

5. ***Парадокс дней рождений***. Парадокс дней рождений заключается в определении количества людей, которых надо собрать в одной комнате, чтобы шансы на то, что среди них найдется хотя бы одна пара родившихся в один день года (имеется в виду день и месяц), превысили шансы того, что все присутствующие родились в разные дни. Найдите (достаточно неожиданный) ответ на этот вопрос. Какое важное следствие из полученного ответа вытекает для хеширования?
6. Ответьте на следующие вопросы для версии хеширования с раздельными цепочками.
   а) Куда вы предпочтете вставлять ключи, если известно, что все ключи в словаре различны? Какие словарные операции могут выиграть от такой модификации?
   б) Мы можем поддерживать ключи в каждом связанном списке в отсортированном порядке. Какие словарные операции могут выиграть от такой модификации? Как можно воспользоваться этим для сортировки всех ключей, хранящихся в таблице?
7. Поясните, как можно применить хеширование для проверки того, что все ключи в списке различны? Какова временна́я эффективность такого применения?
8. Заполните следующую таблицу классами эффективности в среднем случае для пяти реализаций абстрактного типа данных словаря:

	Неупорядоченный массив	Упорядоченный массив	Бинарное дерево поиска	Раздельные цепочки	Линейное исследование
Поиск					
Вставка					
Удаление					

9. Мы рассматривали хеширование в контексте методов, основанных на пространственно-временном компромиссе. Однако оно использует преимущества еще одной общей стратегии. Какой именно?
10. Напишите компьютерную программу, которая использует хеширование для следующей задачи. Имеется текст на естественном языке. Требуется сгенерировать список различных слов в тексте (с количеством вхождений их в этот текст). Внесите в программу необходимые счетчики для сравнения эмпирической эффективности хеширования с соответствующими теоретическими результатами.

7.4 B-деревья

Идея использования дополнительной памяти для ускорения доступа к данным особенно важна, если рассматриваемый набор данных содержит очень большое количество записей, которые требуется хранить на диске. Основной метод организации таких наборов данных — ***индексы***, которые предоставляют определенную информацию о размещении записей с указанными значениями ключей. Для наборов данных, состоящих из структурированных записей (в противоположность таким "неструктурированным" данным, как текст, изображения, звук и видео), наиболее важной организацией индексов являются ***B-деревья*** (B-tree), разработанные Р. Бойером (R. Bayer) и Э. Мак-Грейтом (E. McGreight) [12]. Они расширяют идею 2-3-деревьев (см. раздел 6.3), разрешая иметь в одном узле дерева поиска много ключей.

В B-дереве все записи данных (или ключи) хранятся в листьях, в возрастающем порядке ключей, а родительские узлы используются для индексирования. В частности, каждый родительский узел содержит $n-1$ упорядоченных ключей $K_1 < \cdots < K_{n-1}$ (которые для простоты полагаем различными). Ключи разделены n указателями на дочерние узлы, так что все ключи в поддереве T_0 меньше K_1, все ключи в поддереве T_1 — не меньше K_1и меньше K_2, причем K_1 равен наименьшему ключу в T_1, и так далее до последнего поддерева T_{n-1}, ключи которого не меньше K_{n-1}, причем K_{n-1} равен наименьшему ключу в T_{n-1} (рис. 7.7)[4].

Кроме того, B-дерево порядка $m \geqslant 2$ должно удовлетворять следующим структурным свойствам.

- Корень либо является листом, либо имеет от 2 до m потомков.
- Каждый узел, за исключением корня и листьев, имеет от $\lceil m/2 \rceil$ до m потомков (и, следовательно, от $\lceil m/2 \rceil - 1$ до $m-1$ ключей).

[4]Узел, изображенный на рис. 7.7, называется n-узлом (n-node). Таким образом, все узлы в классическом бинарном дереве поиска являются 2-узлами; 2-3-дерево, рассматривавшееся в разделе 6.3, содержит 2-узлы и 3-узлы.

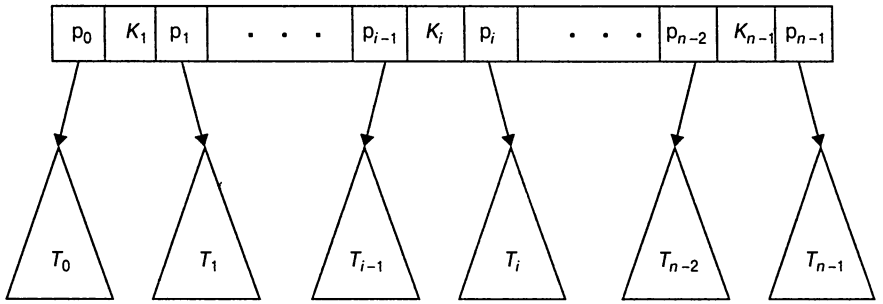


Рис. 7.7. Родительский узел B-дерева

- Дерево (идеально) сбалансировано, т.е. все листья находятся на одном и том же уровне.

Пример B-дерева порядка 4 приведен на рис. 7.8.

Поиск в B-дереве похож на поиск в бинарном дереве поиска, и еще больше — на поиск в 2-3-дереве. Начиная с корня, мы следуем по цепочке указателей к листу, который может содержать искомый ключ. Затем мы ищем ключ среди ключей этого листа. Заметим, что, поскольку ключи хранятся в отсортированном порядке как в родительских узлах, так и в листьях, мы можем воспользоваться бинарным поиском, если количество ключей в узле достаточно велико, чтобы сделать такой поиск оправданным.

Однако при рассмотрении типичного приложения этой структуры данных мы должны в первую очередь рассматривать не количество сравнений ключей. При использовании для хранения больших объемов данных дискового файла узлы B-дерева обычно соответствуют дисковым страницам. Поскольку обычно время, необходимое для обращения к дисковой странице, на несколько порядков превышает время сравнения двух ключей в быстрой основной памяти, основным показателем эффективности для этой и подобных структур данных является количество обращений к диску.

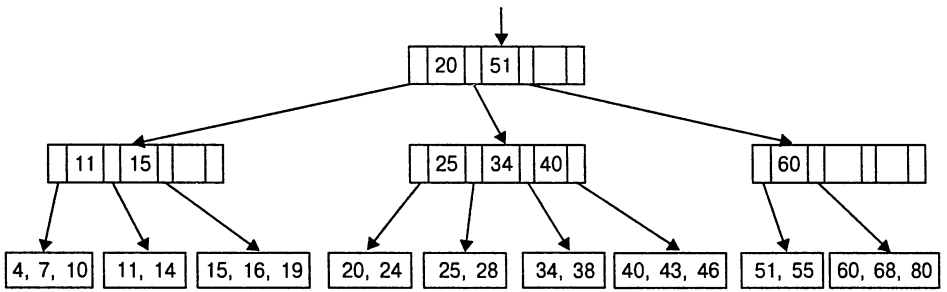


Рис. 7.8. Пример B-дерева порядка 4

К скольким узлам B-дерева требуется доступ при поиске записи с данным значением ключа? Очевидно, что эта величина равна высоте дерева плюс единица. Для оценки высоты найдем, чему равно наименьшее количество ключей в B-дереве порядка m высотой h. Корень дерева содержит как минимум один ключ. На уровне 1 должно быть как минимум два узла с как минимум $\lceil m/2 \rceil - 1$ ключами в каждом, т.е. общее количество ключей на первом уровне не менее $2(\lceil m/2 \rceil - 1)$. На втором уровне имеется не менее $2\lceil m/2 \rceil$ узлов (узлы, дочерние по отношению к узлам первого уровня) с как минимум $\lceil m/2 \rceil - 1$ ключами в каждом, так что общее минимальное количество ключей на втором уровне составляет $2\lceil m/2 \rceil (\lceil m/2 \rceil - 1)$. В общем случае узлы на i-ом уровне ($1 \leqslant i \leqslant h-1$) содержат как минимум $2\lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1)$ ключей. И, наконец, уровень h — уровень листьев — содержит как минимум $2\lceil m/2 \rceil^{h-1}$ узлов с как минимум одним ключом в каждом. Таким образом, для любого B-дерева порядка m с n узлами и высотой $h > 0$ справедливо следующее неравенство:

$$n \geqslant 1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1}.$$

После ряда стандартных упрощений (см. упражнение 7.4.2) это неравенство сводится к

$$n \geqslant 4\lceil m/2 \rceil^{h-1} - 1,$$

которое, в свою очередь, дает следующую верхнюю границу высоты h B-дерева порядка m с n узлами:

$$h \leqslant \left\lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \right\rfloor + 1. \tag{7.7}$$

Из этого неравенства непосредственно следует вывод, что поиск в B-дереве является операцией, принадлежащей классу эффективности $O(\log n)$. Однако нас интересует не только класс эффективности, но и конкретные числа обращений к диску, вычисленные по данной формуле. В приведенной далее таблице содержатся значения правой части формулы для файла со 100 миллионами записей и типичных значений порядка B-дерева m:

Порядок m	50	100	250
Верхняя граница h	6	5	4

Учтите, что в таблице приведены верхние оценки для количества обращений к диску. В реальных приложениях это число редко превосходит 3, к тому же корень B-дерева, а иногда и узлы первого уровня зачастую хранятся в оперативной памяти для минимизации количества обращений к диску.

Операции вставки и удаления оказываются не столь простыми, как операция вставки, но тем не менее они также выполняются за время $O(\log n)$. Здесь мы

набросаем только алгоритм вставки; описание алгоритма удаления можно найти в соответствующей литературе (например, в [5] или [32]).

Наиболее простой алгоритм вставки новой записи в B-дерево очень похож на алгоритм вставки в 2-3-дерево, описанный в разделе 6.3. Сначала мы используем процедуру поиска нового ключа K, для того чтобы найти лист для его вставки. Если в нем есть место для нового ключа, мы вставляем его (в соответствующую позицию, так, чтобы ключи в листе оставались отсортированными), и на этом работа завершается. Если же в листе нет места для новой записи, лист разбивается на два, при этом вторая половина записей переходит в новый узел. После этого наименьший узел K' нового узла и указатель на него вставляются в родительский узел старого листа (непосредственно после ключа и указателя на старый лист). Эта рекурсивная процедура может распространяться вплоть до корня дерева. Если корень также заполнен, старый корень разбивается на две части, и создается новый корень, у которого две части старого корня становятся дочерними узлами. В качестве примера на рис. 7.9 показан результат вставки ключа 65 в B-дерево, изображенное на рис. 7.8 (при ограничении, что листья B-дерева не могут содержать более трех элементов).

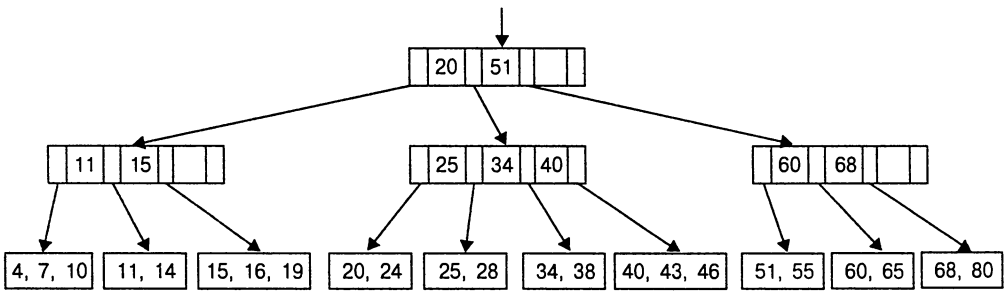


Рис. 7.9. B-дерево, полученное после вставки 65 в B-дерево на рис. 7.8

Вы должны знать, что существуют и другие алгоритмы для реализации вставки в B-дерево. Например, мы можем избежать возможного рекурсивного разделения заполненных узлов, если будем разделять их при поиске листа для новой записи. Еще одна возможность избежать рекурсивного разделения заполненных узлов заключается в перемещении ключа в "братский" узел. Например, вставка 65 в B-дерево на рис. 7.8 может быть выполнена путем перемещения 60, наименьшего ключа в листе, в "братский" лист с 51 и 55, и замещением значения ключа в родительском узле на 65 (новое наименьшее значение ключа у правого потомка). Такое видоизменение позволяет сэкономить память ценой усложнения алгоритма.

B-дерево не обязательно должно быть связано с индексированием большого файла и может рассматриваться как один из вариантов дерева поиска. Как и в случае других типов деревьев поиска — таких как бинарные деревья, AVL-деревья

или 2-3-деревья, — B-деревья могут строиться путем последовательной вставки записей в изначально пустое дерево (пустое дерево также рассматривается как B-дерево). Когда все ключи расположены в листьях, а более высокие уровни организованы как B-дерево, содержащее индекс, вся такая структура обычно называется ***B$^+$-деревом***.

Упражнения 7.4

1. Приведите пример использования индексов в реальной жизни, без компьютеров.
2. **а)** Докажите равенство

$$1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1} = 4\lceil m/2 \rceil^{h-1} - 1,$$

использовавшееся в выводе верхней границы высоты B-дерева (7.7).

б) Завершите вывод неравенства (7.7), дающего верхнюю границу высоты B-дерева.

3. Найдите наименьшее значение порядка B-дерева m, которое гарантирует, что число обращений к диску при поиске в файле со 100 миллионами записей не превысит 3. При этом считаем, что корневая страница хранится в оперативной памяти.
4. Изобразите B-дерево, полученное вставкой 30 и 31 в B-дерево на рис. 7.8, в предположении, что лист не может содержать более трех элементов.
5. Опишите алгоритм для поиска наибольшего ключа B-дерева.
6. **а)** ***Нисходящее 2-3-4-дерево*** (top-down 2-3-4-tree) представляет собой B-дерево порядка 4 со следующей модификацией операции вставки. Когда в процессе поиска листа для нового ключа встречается заполненный узел (т.е. узел с тремя ключами), узел разбивается на два путем передачи среднего ключа в родительский узел (если же полным узлом оказывается корень, то создается новый корень для среднего ключа). Постройте нисходящее 2-3-4-дерево путем вставки в изначально пустое дерево ключей 10, 6, 15, 31, 20, 27, 50, 44, 18.

б) В чем заключается главное преимущество такой процедуры вставки по сравнению со вставкой в 2-3-дерево? В чем ее недостатки?

7. **а)** Напишите программу, реализующую алгоритм вставки ключа в B-дерево.

   б) Напишите программу для визуализации алгоритма вставки ключа в B-дерево.

Резюме

- Пространственно-временной компромисс в проектировании алгоритмов хорошо известен как теоретикам, так и практикам в этой области. В качестве метода проектирования алгоритма использование лишней памяти для экономии времени встречается гораздо чаще, чем лишнее время для экономии памяти.
- *Улучшение входных данных* является одним из двух основных вариантов пространственно-временного компромисса в проектировании алгоритмов. Его идея состоит в предварительной обработке входных данных — целиком или частично — и сохранении полученной дополнительной информации для ускорения решения поставленной задачи. На этом методе основаны, в частности, сортировка подсчетом распределения и несколько важных алгоритмов поиска подстрок.
- *Сортировка подсчетом распределения* представляет собой специальный метод сортировки списков элементов из небольшого множества возможных значений.
- *Алгоритм Хорспула* для поиска подстрок можно рассматривать как упрощенную версию *алгоритма Бойера–Мура*. Оба эти алгоритма основаны на идее улучшения входных данных и сравнении символов образца справа налево. Оба алгоритма используют одну и ту же *таблицу сдвигов несовпадающих символов*, а алгоритм Бойера–Мура использует еще одну таблицу — *таблицу сдвигов совпадающих суффиксов*.
- *Предварительная структуризация* — второй тип методики с применением пространственно-временного компромисса — использует дополнительную память для более быстрого и/или более гибкого доступа к данным. Важными примерами предварительной структуризации являются хеширование и B-деревья.
- *Хеширование* — очень эффективный способ реализации словарей. Он основан на идее отображения ключей в одномерную таблицу. Ограничения, налагаемые на размер такой таблицы, вынуждают применять механизм *разрешения коллизий*. Двумя основными вариантами хеширования являются *открытое хеширование*, или *хеширование с раздельными*

цепочками (когда ключи хранятся в связанных списках вне хеш-таблицы), и *закрытое хеширование*, или *открытая адресация* (когда ключи хранятся внутри хеш-таблицы). Оба варианта в среднем случае обеспечивают эффективность поиска, вставки и удаления, равную $\Theta(1)$.

- *B-дерево* представляет собой сбалансированное дерево поиска, которое обобщает идею 2-3-дерева, разрешая наличие нескольких ключей в одном узле. Основное применение B-деревьев — хранение индексной информации о данных, сохраненных на диске. Путем соответствующего выбора порядка B-дерева можно реализовать операции поиска, вставки и удаления при помощи всего лишь нескольких обращений к диску даже для очень больших файлов.

Глава 8

Динамическое программирование

> Об идее, как и о привидении,... следует немного поговорить, прежде чем она явится.
>
> — Чарльз Диккенс (Charles Dickens) (1812–1870)

Динамическое программирование представляет собой метод проектирования алгоритмов с весьма интересной историей. Оно было открыто известным американским математиком Ричардом Беллманом (Richard Bellman) в 1950-х годах как общий метод оптимизации многостадийных процессов принятия решения. Таким образом, слово "программирование" в названии метода означает "планирование" и не имеет никакого отношения к компьютерному программированию. После того как было доказано, что этот метод представляет собой важный инструмент прикладной математики, динамическое программирование стало рассматриваться, по крайней мере в среде кибернетиков, как метод проектирования алгоритмов, который не ограничивается только узким кругом задач оптимизации. Именно с такой точки зрения мы и будем рассматривать здесь указанный метод.

Динамическое программирование представляет собой метод решения задач с перекрывающимися подзадачами. Обычно такие подзадачи возникают из рекуррентных соотношений, связывающих решение данной задачи с решениями меньших подзадач того же вида. Вместо того чтобы решать перекрывающиеся подзадачи снова и снова, динамическое программирование предлагает решить каждую из меньших подзадач только один раз, записав при этом результат в таблицу, из которой затем можно будет получить решение исходной задачи.

Этот метод можно проиллюстрировать на примере чисел Фибоначчи, рассматривавшихся в разделе 2.5 (даже если вы не читали этот раздел, вы все равно можете следить за ходом рассуждений; однако это интересный вопрос, так что, почувствовав желание прочесть указанный раздел, — не противьтесь ему). Числа Фибоначчи представляют собой элементы последовательности

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

которую можно определить с помощью простого рекуррентного соотношения

$$F(n) = F(n-1) + F(n-2) \quad \text{при } n \geqslant 2 \tag{8.1}$$

и двух начальных условий

$$F(0) = 0, \quad F(1) = 1. \tag{8.2}$$

Если мы попытаемся использовать для вычисления n-го числа Фибоначчи $F(n)$ рекуррентное соотношение (8.1) непосредственно, то придется много раз вычислять одни и те же значения данной функции (см. пример на рис. 2.6). Заметим, что задача вычисления $F(n)$ выражается через перекрывающиеся подзадачи меньшего размера — вычисления $F(n-1)$ и $F(n-2)$. Таким образом, мы можем просто заполнить элементы одномерного массива $n+1$ последовательными значениями $F(n)$, начиная с начальных значений (8.2) и используя рекуррентное соотношение (8.1). Очевидно, что последний элемент данного массива будет содержать значение $F(n)$. Псевдокод такого очень простого алгоритма можно найти в разделе 2.5.

Заметим, что можно обойтись и без использования дополнительного массива, ограничившись запоминанием только двух последних элементов последовательности Фибоначчи (см. упражнение 2.5.6). Такая ситуация не является чем-то необычным, мы встретимся с ней еще в нескольких примерах в данной главе. Хотя непосредственное применение динамического программирования можно рассматривать как частный случай экономии времени за счет памяти, зачастую алгоритмы динамического программирования можно усовершенствовать, избежав использования излишней памяти.

Некоторые алгоритмы вычисляют n-е число Фибоначчи без вычисления всех предшествующих элементов последовательности (см. раздел 2.5). Однако для алгоритмов, основанных на классическом подходе восходящего динамического программирования, типично решение *всех* меньших подзадач данной задачи. Один из вариантов динамического программирования состоит в том, чтобы избежать решения ненужных подзадач. Этот метод, проиллюстрированный в разделе 8.4, использует так называемые функции с запоминанием и может рассматриваться как нисходящая разновидность динамического программирования. Однако используем ли мы классическую восходящую разновидность динамического программирования или нисходящее динамическое программирование с применением функций с запоминанием, основной этап в разработке таких алгоритмов остается один и тот же, а именно: вывод рекурсивного соотношения, связывающего решение экземпляра задачи с решениями меньших (перекрывающихся) подэкземпляров. Непосредственная применимость уравнения (8.1) для вычисления n-го числа Фибоначчи является одним из немногих исключений из этого правила.

В разделах и упражнениях этой главы имеется несколько стандартных примеров алгоритмов динамического программирования (некоторые из них были разработаны до открытия динамического программирования или независимо от него и только позже стали рассматриваться как примеры применения этого метода). Множество других применений динамического программирования лежат в диапазоне от оптимального разбиения текста на строки (например, [8]) до оптимальной триангуляции многоугольника (например, [111]) и решения различных сложных инженерных задач (например, [14, 18]).

8.1 Вычисление биномиальных коэффициентов

Вычисление биномиальных коэффициентов — стандартный пример применения динамического программирования к задаче, не являющейся задачей оптимизации. Вспомним из курса элементарной комбинаторики, что ***биномиальным коэффициентом*** (binomial coefficient), обозначаемым $C(n, k)$, C_n^k или $\binom{n}{k}$, называется количество комбинаций (подмножеств) из k элементов у n-элементного множества $0 \leqslant k \leqslant n$). Название "биномиальные коэффициенты" происходит от участия этих чисел в формуле бинома:

$$(a+b)^n = C_n^0 a^n + \cdots + C_n^k a^{n-k} b^k + \cdots + C_n^n b^n.$$

Из множества свойств биномиальных коэффициентов мы остановимся только на двух:

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k \quad \text{при } n > k > 0 \tag{8.3}$$

и

$$C_n^0 = C_n^n = 1. \tag{8.4}$$

Природа рекуррентного соотношения (8.3), выражающего задачу вычисления C_n^k посредством решения меньших перекрывающихся задач вычисления C_{n-1}^{k-1} и C_{n-1}^k подводит нас к решению задачи с использованием метода динамического программирования. Для этого запишем значения биномиальных коэффициентов в таблицу с $n+1$ строками и $k+1$ столбцами, пронумерованными, соответственно, от 0 до n и от 0 до k (рис. 8.1).

Для вычисления C_n^k мы заполняем таблицу на рис. 8.1 строку за строкой, начиная со строки 0 и заканчивая строкой n. Каждая строка i ($0 \leqslant i \leqslant n$) заполняется слева направо, начиная с 1, так как $C_n^0 = 1$. На главной диагонали таблицы в строках от 0 по k-ую также находятся 1, поскольку $C_i^i = 1$ ($0 \leqslant i \leqslant k$). Остальные элементы таблицы вычисляются по формуле (8.3) путем сложения элементов предыдущей строки — из того же и предшествующего столбца (если в этой таблице вы узнали треугольник Паскаля — очаровательную математическую структуру,

	0	1	2	$\cdots$	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\vdots$						
k	1					1
$\vdots$						
$n-1$	1				C_{n-1}^{k-1}	C_{n-1}^{k}
n	1					C_n^k

Рис. 8.1. Таблица для вычисления биномиальных коэффициентов C_n^k при помощи алгоритма динамического программирования

обычно рассматриваемую при изучении сочетаний, — то вы правы: перед вами действительно треугольник Паскаля). Вот псевдокод, реализующий данный алгоритм.

Алгоритм $Binomial\,(n, k)$

```
// Вычисление C(n, k) при помощи алгоритма динамического
// программирования
// Входные данные:  Пара неотрицательных целых чисел n ⩾ k ⩾ 0
// Выходные данные: Значение C(n, k)
for i ← 0 to n do
  for j ← 0 to min(i, k) do
    if j = 0 or j = i
      C[i, j] ← 1
    else
      C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, k]
```

Какова временна́я эффективность данного алгоритма? Очевидно, что базовой операцией алгоритма является сложение, так что обозначим через $A\,(n, k)$ общее количество сложений, выполняемых при вычислении C_n^k. Заметим, что вычисление каждого элемента по формуле (8.3) требует только одного сложения. Кроме того, поскольку первые $k + 1$ строк таблицы образуют треугольник , а остальные $n - k$ строк — прямоугольник, можно разбить сумму, выражающую значение

$A(n,k)$, на две части:

$$A(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k = \\ = \frac{(k-1)\,k}{2} + k\,(n-k) \in \Theta(nk).$$

В упражнениях от вас потребуется сравнить полученную эффективность с эффективностью некоторых других алгоритмов для решения этой задачи. Еще одно из упражнений состоит в анализе того, нельзя ли сэкономить дополнительную память, используемую данным алгоритмом динамического программирования.

Упражнения 8.1

1. а) Что общего имеет динамическое программирование с методом декомпозиции?

б) В чем главное отличие между этими двумя методами?

2. а) Вычислите C_6^3 при помощи алгоритма динамического программирования.

б) Можно ли вычислить C_n^k, заполняя таблицу из алгоритма динамического программирования по столбцам, а не по строкам?

3. Докажите следующее утверждение, сделанное в тексте раздела при определении временно́й эффективности алгоритма динамического программирования для вычисления C_n^k:

$$\frac{(k-1)\,k}{2} + k\,(n-k) \in \Theta(nk).$$

4. а) Сколько дополнительной памяти требуется алгоритму динамического программирования *Binomial* для вычисления C_n^k?

б) Как можно повысить эффективность использования дополнительной памяти этим алгоритмом (попытайтесь снизить количество дополнительной памяти настолько, насколько сможете).

5. а) Найдите порядок роста следующих функций:

1) C_n^1 **2)** C_n^2 **3)** $C_n^{n/2}$ для четных n

б) Какое основное следствие для вычисления C_n^k вытекает из ответа на часть *а)* данного упражнения?

6. Найдите точное количество сложений, выполняемое следующим рекурсивным алгоритмом, основанным на непосредственном применении формул (8.3) и (8.4):

Алгоритм *BinomCoeff*(n, k)
  if $k = 0$ **or** $k = n$
    return 1
  else
    return *BinomCoeff*$(n-1, k-1)$ + *BinomCoeff*$(n-1, k)$

7. Какой из следующих алгоритмов для вычисления биномиальных коэффициентов наиболее эффективен?

   а) Использование формулы

$$C_n^k = \frac{n!}{k!\,(n-k)!}.$$

   б) Использование формулы

$$C_n^k = \frac{n\,(n-1)\cdots(n-k+1)}{k!}.$$

   в) Рекурсивное применение формулы

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k \quad \text{при } n > k > 0,\ C_n^0 = C_n^n = 1.$$

   г) Применение алгоритма динамического программирования.

8. Докажите, что

$$C_n^k = C_n^{n-k} \quad \text{при } n \geqslant k \geqslant 0,$$

   и поясните, как можно воспользоваться этой формулой при вычислении C_n^k.

9. ***Задача о чемпионате.*** Рассмотрим две команды, A и B, которые проводят серию игр до тех пор, пока одна из них не одержит n побед. Предположим, что вероятность победы команды A одна и та же в каждой игре и равна p, а вероятность проигрыша — $q = 1 - p$ (следовательно, ничьих в игре не бывает). Пусть $P\,(i, j)$ — вероятность победы A в серии игр, если A для победы требуется выиграть еще i игр, а B для победы требуется выиграть еще j игр.

   а) Напишите рекуррентное соотношение для $P\,(i, j)$, которое можно использовать в алгоритме динамического программирования.

   б) Найдите вероятность того, что A победит в серии из 7 игр, если вероятность победы в одной игре равна 0.4.

   в) Напишите псевдокод алгоритма динамического программирования для решения этой задачи и определите его временну́ю и пространственную эффективность.

8.2 Алгоритмы Воршалла и Флойда

В этом разделе мы рассмотрим два хорошо известных алгоритма: алгоритм Воршалла для вычисления транзитивного замыкания ориентированного графа и алгоритм Флойда для решения задачи поиска кратчайших путей между всеми парами вершин. Эти алгоритмы основаны, по сути, на одной и той же идее, которую можно рассматривать как применение метода динамического программирования.

Алгоритм Воршалла

Вспомним, что матрица смежности $A = \{a_{ij}\}$ ориентированного графа представляет собой булеву матрицу, которая содержит на пересечении i-ой строки и j-го столбца 1 тогда и только тогда, когда имеется ориентированное ребро от i-ой вершины к j-ой. Нас может также интересовать матрица, содержащая информацию о существовании ориентированного пути произвольной длины между вершинами графа.

Определение 1. ***Транзитивное замыкание*** (transitive closure) ориентированного графа с n вершинами можно определить как булеву матрицу $T = \{t_{ij}\}$ размером $n \times n$, в которой элемент на пересечении i-ой строки ($1 \leqslant i \leqslant n$) и j-го столбца ($1 \leqslant j \leqslant n$) равен 1, если существует нетривиальный ориентированный путь (т.е. ориентированный путь положительной длины) из вершины i в вершину j; в противном случае значение t_{ij} равно 0. ■

На рис. 8.2 приведен пример ориентированного графа, его матрицы смежности и транзитивного замыкания.

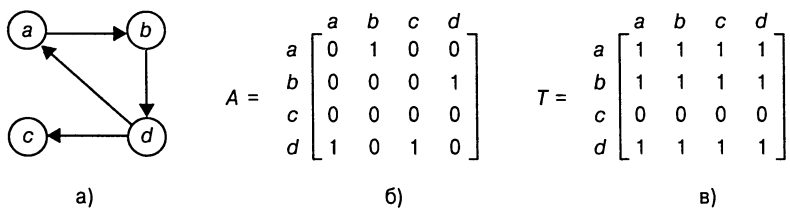


Рис. 8.2. *а*) Ориентированный граф, его *б*) матрица смежности и *в*) транзитивное замыкание

Транзитивное замыкание ориентированного графа можно построить с помощью поиска в глубину или в ширину. Выполнение одного из поисков, начиная от i-ой вершины, дает информацию о вершинах, достижимых из нее, а следовательно, — столбцы, на пересечении которых с i-ой строкой матрицы транзитивного замыкания содержатся единицы. Таким образом, полную матрицу транзитивно-

го замыкания можно получить, выполняя обход графа для каждой его вершины в качестве начальной точки.

Поскольку такой метод неоднократно обходит один и тот же ориентированный граф, можно надеяться на существование более эффективного алгоритма. Такой алгоритм существует и называется ***алгоритмом Воршалла*** (Warshall's algorithm) по имени его автора, С. Воршалла (S. Warshall) [119]. Алгоритм Воршалла строит транзитивное замыкание ориентированного графа с n вершинами как последовательность булевых матриц размером $n \times n$:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}. \tag{8.5}$$

Каждая из этих матриц предоставляет определенную информацию о направленных путях в ориентированном графе. В частности, элемент $r_{ij}^{(k)}$ на пересечении i-ой строки и j-го столбца матрицы $R^{(k)}$ ($k = 0, 1, \ldots, n$) равен 1 тогда и только тогда, когда существует ориентированный путь (положительной длины) из i-ой в j-ую вершину такой, что все промежуточные вершины, если таковые имеются, имеют номера не выше k. Таким образом, пути в $R^{(0)}$ не могут иметь промежуточных вершин, а значит, $R^{(0)}$ представляет собой не что иное, как матрицу смежности ориентированного графа. Матрица $R^{(1)}$ содержит информацию о путях, в которых в качестве промежуточной вершины может выступать вершина с номером 1, а значит, если можно так выразиться, должна содержать больше единиц, чем $R^{(0)}$. В общем случае каждая следующая матрица последовательности (8.5) допускает в качестве промежуточных вершин на одну больше, чем предыдущая, а следовательно, может (но не обязана) содержать большее количество единиц. Последняя матрица последовательности $R^{(n)}$ в качестве промежуточных вершин может использовать все n вершин ориентированного графа, а значит, представляет собой не что иное, как транзитивное замыкание ориентированного графа.

Основная особенность алгоритма заключается в том, что можно вычислить все элементы каждой матрицы $R^{(k)}$ на основании ее непосредственного предшественника $R^{(k-1)}$ в ряду (8.5). Пусть $r_{ij}^{(k)}$, элемент на пересечении i-ой строки и j-го столбца матрицы $R^{(k)}$, равен 1. Это означает, что существует путь из i-ой вершины v_i в j-ую вершину v_j, в котором все промежуточные вершины имеют номера не выше k:

$$v_i \rightsquigarrow \text{Список вершин с номерами, не превышающими } k \rightsquigarrow v_j. \tag{8.6}$$

Что касается такого пути, то возможны две ситуации. Первая: список промежуточных вершин не содержит k-ую. Тогда этот путь из v_i в v_j состоит из вершин, имеющих номера не выше $k-1$, следовательно, элемент $r_{ij}^{(k-1)}$ также равен 1. Вторая: путь (8.6) содержит среди прочих промежуточных вершин k-ую вершину v_k. Без потери общности можно считать, что v_k встречается в списке только один

раз (если это не так, можно создать новый путь от v_i до v_j, в котором будут удалены все вершины между первым и последним вхождениями v_k). С учетом сказанного путь (8.6) можно переписать следующим образом:

$$v_i \rightsquigarrow \text{вершины с номерами } \leqslant k-1, v_k, \text{ вершины с номерами } \leqslant k-1 \rightsquigarrow v_j.$$

Первая часть этого представления означает, что существует путь от v_i до v_k такой, что все промежуточные вершины в нем имеют номера не больше, чем $k-1$ (следовательно, $r_{ik}^{(k-1)} = 1$), а вторая часть означает, что существует путь от v_k до v_j такой, что все промежуточные вершины в нем имеют номера не больше, чем $k-1$ (следовательно, $r_{kj}^{(k-1)} = 1$).

В результате мы только что доказали, что если $r_{ij}^{(k)} = 1$, то либо $r_{ij}^{(k-1)} = 1$, либо и $r_{ik}^{(k-1)} = 1$, и $r_{kj}^{(k-1)} = 1$. Легко увидеть, что истинно утверждение, обратное этому. Таким образом, мы получаем формулу для генерации элементов матрицы $R^{(k)}$ из элементов матрицы $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \textbf{ or } r_{ik}^{(k-1)} \textbf{ and } r_{kj}^{(k-1)}. \tag{8.7}$$

Формула (8.7) представляет собой "сердце" алгоритма Воршалла. Из нее вытекает следующее правило генерации элементов матрицы $R^{(k)}$ из элементов матрицы $R^{(k-1)}$, которое, в частности, удобно при использовании алгоритма Воршалла вручную.

- Если элемент r_{ij} равен 1 в $R^{(k-1)}$, то он остается равен 1 и в $R^{(k)}$.
- Если элемент r_{ij} равен 0 в $R^{(k-1)}$, то он становится равным 1 в $R^{(k)}$ тогда и только тогда, когда в $R^{(k-1)}$ и элемент в i-ой строке и k-ом столбце, и элемент в k-ой строке и j-ом столбце равны 1 (это правило проиллюстрировано на рис. 8.3).

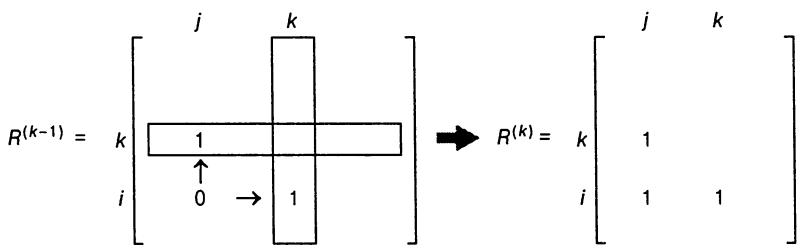
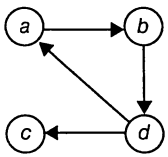


Рис. 8.3. Правило для замены нолей в алгоритме Воршалла

Применение алгоритма Воршалла к ориентированному графу, изображенному на рис. 8.2, показано на рис. 8.4.

Вот как выглядит псевдокод алгоритма Воршалла.



$R^{(0)}=$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Единицы отражают наличие путей без промежуточных вершин ($R^{(0)}$ — матрица смежности); строка и столбец в прямоугольниках используются для вычисления $R^{(1)}$.

$R^{(1)}=$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 1, т.е. только с вершиной *a* (новый путь от *d* к *b*); выделенные прямоугольниками строка и столбец используются для получения $R^{(2)}$.

$R^{(2)}=$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 2, т.е. с вершинами *a* и *b* (два новых пути); выделенные прямоугольниками строка и столбец используются для получения $R^{(3)}$.

$R^{(3)}=$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 3, т.е. с вершинами *a*, *b* и *c* (новых путей нет); выделенные прямоугольниками строка и столбец используются для получения $R^{(4)}$.

$R^{(4)}=$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 4 т.е. с вершинами *a*, *b*, *c* и *d* (пять новых путей).

Рис. 8.4. Применение алгоритма Воршалла к ориентированному графу. Новые единицы в матрицах выделены полужирным шрифтом

Алгоритм *Warshall* $(A[1..n, 1..n])$

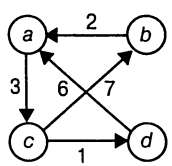
// Реализует алгоритм Воршалла для вычисления транзитивного
// замыкания
// **Входные данные:** Матрица смежности A ориентированного
// графа с n вершинами
// **Выходные данные:** Транзитивное замыкание ориентированного
// графа
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j]$
return $R^{(n)}$

По поводу алгоритма Воршалла требуется сделать несколько замечаний. Во-первых, он очень краток. Во-вторых, его временна́я эффективность составляет $\Theta(n^3)$, так что для разреженных графов, представленных связными списками смежности, алгоритм с использованием обхода графа, упомянутый в начале данного раздела, оказывается асимптотически эффективнее алгоритма Воршалла (почему?). Можно ускорить приведенную выше реализацию алгоритма Воршалла, видоизменив его внутренний цикл (см. упражнение 8.2.4). Еще один путь ускорения алгоритма состоит в рассмотрении строк матрицы как битовых строк и использовании побитовой операции **or**, имеющейся на большинстве современных компьютеров.

Что же касается пространственной эффективности алгоритма Воршалла, то ситуация похожа на два примера, рассматривавшиеся ранее в этой главе: вычисление чисел Фибоначчи и биномиальных коэффициентов. Хотя мы используем отдельные матрицы для промежуточных результатов алгоритма, на самом деле это не является необходимым (в упражнении 8.2.3 спрашивается, как можно избежать такого напрасного расхода памяти). Наконец, позже мы увидим, как идея, лежащая в основе алгоритма Воршалла, может быть применена к более общей задаче поиска длин кратчайших путей во взвешенных графах.

Алгоритм Флойда поиска кратчайших путей между всеми парами вершин

Задача поиска кратчайших путей между всеми парами вершин (all-pairs shortest-paths problem) состоит в поиске для данного взвешенного связного графа (ориентированного или неориентированного) расстояний от каждой вершины до всех других вершин. Для записи длин кратчайших путей удобно воспользоваться матрицей D размером $n \times n$, которая называется ***матрицей расстояний*** (distance matrix): элемент d_{ij} на пересечении i-ой строки и j-го столбца такой матрицы указывает длину кратчайшего пути от i-ой вершины до j-ой вершины ($1 \leqslant i, j \leqslant n$). Пример такой матрицы приведен на рис. 8.5.



а)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

б)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

в)

Рис. 8.5. *а*) Ориентированный граф, *б*) его весовая матрица и *в*) матрица расстояний

Мы можем построить матрицу расстояний при помощи алгоритма, очень похожего на алгоритм Воршалла. Этот алгоритм называется ***алгоритмом Флойда*** (Floyd's algorithm) по имени его изобретателя Р. Флойда (R. Floyd) [36]. Он применим как к ориентированным, так и к неориентированным взвешенным графам, лишь бы они не содержали циклов с отрицательной длиной (естественно, в случае ориентированного графа под путем или циклом мы подразумеваем ориентированный путь или цикл).

Алгоритм Флойда вычисляет матрицу расстояний взвешенного графа с n вершинами посредством построения последовательности

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}. \tag{8.8}$$

Каждая из этих матриц содержит длины кратчайших путей с определенными ограничениями. В частности, элемент $d_{ij}^{(k)}$ на пересечении i-ой строки и j-го столбца матрицы $D^{(k)}$ $k = 0, 1, \ldots, n$) равен длине кратчайшего пути среди всех путей от i-ой вершины к j-ой, в которых промежуточные вершины (если таковые есть) не могут иметь номера, превышающие k. В частности, последовательность начинается с матрицы $D^{(0)}$, в которой в путях не может быть промежуточных вершин (т.е. $D^{(0)}$ представляет собой просто весовую матрицу графа). Последняя матрица последовательности, $D^{(n)}$, содержит длины кратчайших путей среди всех путей, в которых в качестве промежуточных вершин могут быть любые из n вершин графа, так что матрица $D^{(n)}$ и есть искомая матрица расстояний графа.

Как и в алгоритме Воршалла, мы можем вычислить все элементы каждой матрицы $D^{(k)}$ на основании информации об элементах предшествующей ей матрицы $D^{(k-1)}$ в последовательности (8.8). Пусть $d_{ij}^{(k)}$ — элемент на пересечении i-ой строки и j-го столбца матрицы $D^{(k)}$. Это означает, что $d_{ij}^{(k)}$ равен длине кратчайшего пути среди всех путей от i-ой вершины v_i к j-ой вершине v_j, промежуточные вершины которых имеют номера не выше k:

$$v_i \rightsquigarrow \text{Список вершин с номерами, не превышающими } k \rightsquigarrow v_j \tag{8.9}$$

Все такие пути можно разбить на два непересекающихся подмножества: те пути, в которых в качестве промежуточной не участвует k-ая вершина v_k, и те, в которых она является одной из промежуточных. Поскольку пути в первом подмножестве содержат промежуточные вершины с номерами не выше $k-1$, то кратчайший путь между этими вершинами, по определению наших матриц, имеет длину $d_{ij}^{(k-1)}$.

Чему равна длина кратчайшего пути во втором подмножестве? Если граф не содержит циклов отрицательной длины, мы можем ограничить рассмотрение путей во втором подмножестве только теми, в которые вершина v_k входит только один раз (поскольку посещение вершины v_k больше одного раза может только увеличить длину пути). Все такие пути имеют следующий вид:

$$v_i \rightsquigarrow \text{вершины с номерами } \leqslant k-1, v_k, \text{вершины с номерами } \leqslant k-1 \rightsquigarrow v_j.$$

Другими словами, каждый из этих путей состоит из пути от v_i до v_k, причем все промежуточные вершины имеют номера не выше $k-1$, и пути от v_k до v_j, у которого все промежуточные вершины также имеют номера не выше $k-1$. Схематично данная ситуация изображена на рис. 8.6.

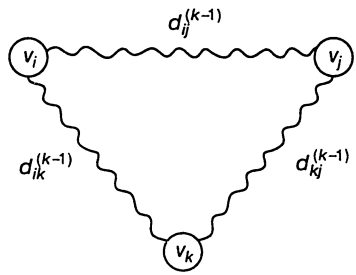


Рис. 8.6. Идея, лежащая в основе алгоритма Флойда

Поскольку длина кратчайшего пути от v_i до v_k среди всех путей, не использующих промежуточных вершин с номерами больше $k-1$, равна $d_{ik}^{(k-1)}$, а длина кратчайшего пути от v_k до v_j среди всех путей, не использующих промежуточных вершин с номерами больше $k-1$, равна $d_{kj}^{(k-1)}$, длина кратчайшего пути от v_i к v_j через вершину v_k равна $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$. С учетом длины кратчайших путей в обоих подмножествах получаем следующее рекуррентное соотношение:

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} \quad \text{для } k \geqslant 1, d_{ij}^{(0)} = w_{ij}. \tag{8.10}$$

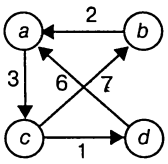
Т.е. элемент на пересечении i-ой строки и j-го столбца текущей матрицы расстояний $D^{(k-1)}$ заменяется суммой элементов на пересечении той же i-ой строки и k-го столбца, и k-ой строки и того же j-го столбца тогда и только тогда, когда эта сумма оказывается меньше текущего значения.

Применение алгоритма Флойда к графу, изображенному на рис. 8.5, показано на рис. 8.7.

Вот как выглядит псевдокод алгоритма Флойда. В нем используется тот факт, что каждая последующая матрица в последовательности (8.8) может быть записана поверх предшественника.

Алгоритм $Floyd(W[1..n, 1..n])$

// Реализует алгоритм Флойда для решения задачи поиска
// кратчайших путей между всеми парами вершин
// **Входные данные:** Весовая матрица W графа
// **Выходные данные:** Матрица длин кратчайших путей
$D \leftarrow W$ // Не требуется, если W может быть перезаписана



$D^{(0)}=$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Длины кратчайших путей без промежуточных вершин ($D^{(0)}$ представляет собой просто весовую матрицу)

$D^{(1)}=$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 1, т.е. только *a* (два новых кратчайших пути от *b* к *c* и от *d* к *c*)

$D^{(2)}=$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 2, т.е. *a* и *b* (новый кратчайший путь от *c* к *a*)

$D^{(3)}=$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 3, т.е. *a*, *b* и *c* (четыре новых кратчайший пути от *a* к *b*, от *a* к *d*, от *b* к *d* и от *d* к *b*)

$D^{(4)}=$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 4, т.е. *a*, *b*, *c* и *d* (новый кратчайший путь от *c* к *a*)

Рис. 8.7. Применение алгоритма Флойда к ориентированному графу. Обновленные элементы выделены полужирным шрифтом

```
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Очевидно, что временна́я эффективность алгоритма Флойда — кубическая, как и временна́я эффективность алгоритма Воршалла. В следующей главе мы рассмотрим еще один метод поиска кратчайших путей — алгоритм Дейкстры.

Мы завершаем этот раздел важным глобальным комментарием, посвященным общему принципу, лежащему в основе применения алгоритмов динамического программирования для оптимизационных задач. Ричард Беллман (Richard Bellman) назвал его ***принципом оптимальности*** (principle of optimality). В несколько измененной по сравнению с оригинальной формулировке он гласит, что оптимальное решение любого экземпляра задачи оптимизации составляется из оптималь-

ных решений его подэкземпляров. Принцип оптимальности выполняется гораздо чаще, чем не выполняется (в качестве редкого примера его невыполнения можно привести задачу поиска длиннейших простых путей). Хотя, конечно же, применимость этого принципа к каждой конкретной задаче должна быть доказана, обычно это не является главной трудностью при разработке алгоритма динамического программирования. Обычно самое сложное — найти, какие наименьшие подэкземпляры должны быть рассмотрены, и вывести уравнение, связывающее решение любого экземпляра с решениями его меньших подэкземпляров. В оставшейся части главы мы рассмотрим еще несколько примеров алгоритмов динамического программирования.

Упражнения 8.2

1. Примените алгоритм Воршалла для поиска транзитивного замыкания ориентированного графа, определенного следующей матрицей смежности:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2. **а)** Докажите, что временна́я эффективность алгоритма Воршалла — кубическая.
   б) Поясните, почему временна́я эффективность алгоритма Воршалла ниже эффективности алгоритма на основе обхода для разреженных графов, представленных при помощи связанных списков смежности.
3. Поясните, как реализовать алгоритм Воршалла без использования дополнительной памяти для хранения элементов промежуточных матриц.
4. Поясните, как реструктуризировать внутренний цикл алгоритма *Warshall* так, чтобы он работал быстрее по крайней мере для некоторых входных данных.
5. Перепишите псевдокод алгоритма Воршалла в предположении, что строки матрицы представлены битовыми строками, к которым может быть применена операция побитового "или" (**or**).
6. **а)** Поясните, как алгоритм Воршалла может использоваться для определения того факта, что данный граф является ориентированным

ациклическим графом. Является ли этот алгоритм подходящим для решения данной задачи?

б) Стоит ли использовать алгоритм Воршалла для поиска транзитивного замыкания неориентированного графа?

7. Решите задачу поиска кратчайших путей между всеми парами вершин в ориентированном графе с весовой матрицей

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

8. Докажите, что очередная матрица в последовательности (8.8) может быть записана поверх предшествующей.

9. Приведите пример ориентированного или неориентированного графа с отрицательными весами, для которого алгоритм Флойда дает неверный результат.

10. Улучшите алгоритм Флойда так, чтобы можно было найти не только длины кратчайших путей, но и сами эти пути.

8.3 Оптимальные бинарные деревья поиска

Бинарное дерево поиска представляет собой одну из наиболее важных структур данных в кибернетике. Одно из главных ее применений — реализация словаря, множества элементов с операциями поиска, вставки и удаления. Если вероятности поиска элементов множества известны (например, из накопленных данных о последних поисках), естественным образом встает вопрос об оптимальном бинарном дереве поиска, для которого среднее количество сравнений является наименьшим возможным (для простоты мы ограничимся минимизацией среднего количества сравнений при успешном поиске. Рассматриваемый метод может быть расширен для включения неудачных поисков).

В качестве примера рассмотрим четыре ключа A, B, C и D, поиск которых осуществляется с вероятностями 0.1, 0.2, 0.4 и 0.3, соответственно. На рис. 8.8 изображены два из четырнадцати возможных бинарных деревьев поиска, содержащих эти ключи. Среднее количество сравнений при успешном поиске в первом из этих деревьев равно $0.1 \cdot 1 + 0.2 \cdot 2 + 0.4 \cdot 3 + 0.3 \cdot 4 = 2.9$, а во втором — $0.1 \cdot 2 + 0.2 \cdot 1 + 0.4 \cdot 2 + 0.3 \cdot 3 = 2.1$. Ни одно из этих деревьев не является оптимальным. (Можете ли вы сказать, какое дерево будет оптимальным?)

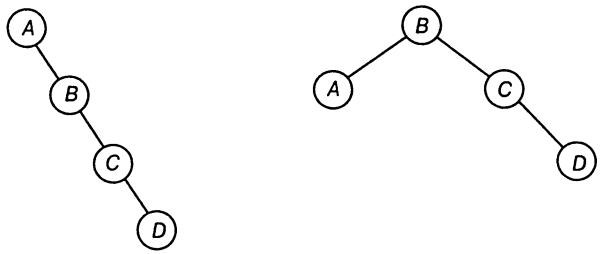


Рис. 8.8. Два из четырнадцати возможных бинарных деревьев поиска с ключами A, B, C и D

В этом небольшом примере мы можем найти оптимальное дерево, построив все 14 бинарных деревьев поиска с данными ключами. Однако в качестве универсального алгоритма метод исчерпывающего перебора нереален: общее количество бинарных деревьев поиска с n ключами равно n-му ***числу Каталана*** (Catalan number):

$$c(n) = \frac{C_{2n}^{n}}{n+1} \quad \text{при } n > 0, c(0) = 1,$$

которое растет с ростом n как $4^n/n^{1.5}$ (см. упражнение 8.3.7).

Итак, пусть $a_1, \ldots, a_n$ — различные ключи, упорядоченные от наименьшего к наибольшему, и пусть $p_1, \ldots p_n$ — вероятности их поиска. Пусть $C[i,j]$ — наименьшее среднее количество сравнений, выполняемых при успешном поиске в бинарном дереве T_i^j, составленном из ключей $a_i, \ldots, a_j$, где i и j — некоторые целые индексы, такие, что $1 \leqslant i \leqslant j \leqslant n$. Таким образом, следуя классическому подходу динамического программирования, мы будем искать значения $C[i,j]$ для всех меньших экземпляров задачи, хотя нас интересует только значение $C[1,n]$. Чтобы вывести рекуррентное соотношение, лежащее в основе алгоритма динамического программирования, надо рассмотреть все возможные способы выбора корня a_k среди узлов $a_i, \ldots, a_j$. Для такого бинарного дерева (рис. 8.9) корень содержит ключ a_k, левое поддерево T_i^{k-1} — оптимально упорядоченные ключи $a_i, \ldots, a_{k-1}$, а правое поддерево T_{k+1}^j — оптимально упорядоченные ключи $a_{k+1}, \ldots, a_j$ (обратите внимание на использование здесь принципа оптимальности).

Если мы будем считать уровни дерева, начиная с 1 (для того, чтобы количество сравнений равнялось уровню ключа), то получим следующее рекуррентное соотношение:

$$C[i,j] = \min_{i \leqslant k \leqslant j} \left\{ p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot \Big(\text{Уровень } a_s \text{ в } T_i^{k-1} + 1\Big) + \right.$$
$$\left. + \sum_{s=k+1}^{j} p_s \cdot \Big(\text{Уровень } a_s \text{ в } T_{k+1}^{j} + 1\Big) \right\} =$$
$$= \min_{i \leqslant k \leqslant j} \left\{ p_k + \sum_{s=i}^{k-1} p_s \cdot \text{Уровень } a_s \text{ в } T_i^{k-1} + \sum_{s=1}^{k-1} p_s + \right.$$
$$\left. + \sum_{s=k+1}^{j} p_s \cdot \text{Уровень } a_s \text{ в } T_{k+1}^{j} + \sum_{s=k+1}^{j} p_s \right\} =$$
$$= \min_{i \leqslant k \leqslant j} \left\{ \sum_{s=i}^{k-1} p_s \cdot \text{Уровень } a_s \text{ в } T_i^{k-1} + \right.$$
$$\left. + \sum_{s=k+1}^{j} p_s \cdot \text{Уровень } a_s \text{ в } T_{k+1}^{j} + \sum_{s=i}^{j} p_s \right\} =$$
$$= \min_{i \leqslant k \leqslant j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s.$$

Таким образом, мы получаем рекуррентное соотношение

$$C[i,j] = \min_{i \leqslant k \leqslant j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{для } 1 \leqslant i \leqslant j \leqslant n. \quad (8.11)$$

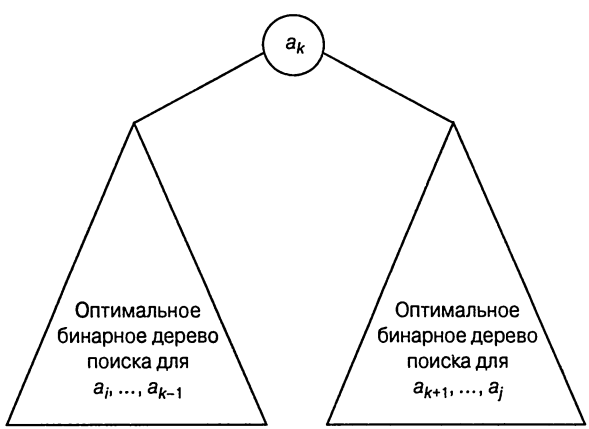


Рис. 8.9. Бинарное дерево поиска с корнем a_k и двумя оптимальными бинарными поддеревьями поиска T_i^{k-1} и T_{k+1}^{j}

В формуле (8.11) мы полагаем, что $C[i, i-1] = 0$ при $1 \leqslant i \leqslant n+1$, что можно рассматривать как количество сравнений в пустом дереве. Заметим: из этой формулы следует, что

$$C[i, i] = p_i \quad \text{при } 1 \leqslant i \leqslant n,$$

как и должно быть для бинарного дерева с одним узлом a_i.

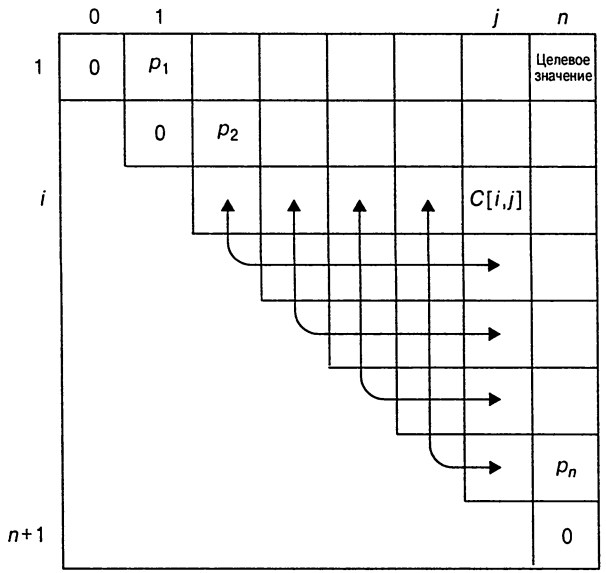


Рис. 8.10. Таблица алгоритма динамического программирования для построения оптимального бинарного дерева поиска

Двумерная таблица на рис. 8.10 показывает значения, необходимые для вычисления $C[i, j]$ по формуле (8.11): они находятся в строке i в столбцах слева от столбца j, и в столбце j ниже строки i. Стрелками показаны пары элементов таблицы, суммы которых вычисляются для поиска наименьшего значения, которое будет записано в качестве значения $C[i, j]$. Сказанное предполагает заполнение таблицы вдоль диагоналей, начиная с нулевых значений на главной диагонали и заданных вероятностей p_i, $1 \leqslant i \leqslant n$, прямо над ней, и перемещаясь к верхнему правому углу.

Описанный алгоритм вычисляет $C[1, n]$ — среднее количество сравнений при успешном поиске в оптимальном бинарном дереве. Если мы, кроме того, хотим получить само оптимальное бинарное дерево поиска, то надо поддерживать еще одну двумерную таблицу для записи значений k, при которых достигается минимум в (8.11). Эта таблица имеет ту же форму, что и таблица на рис. 8.10, и заполняется точно так же, — начиная с элементов $R[i, i] = i$ для $1 \leqslant i \leqslant n$. Когда

таблица оказывается заполненной, ее элементы указывают индексы корней оптимальных поддеревьев, которые позволяют реконструировать оптимальное дерево для всего множества.

Пример 1. Проиллюстрируем данный алгоритм, применяя его ко множеству из четырех ключей, которое было упомянуто в начале раздела:

Ключ	A	B	C	D
Вероятность	0.1	0.2	0.4	0.3

Начальные таблицы выглядят следующим образом:

Главная таблица

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

Таблица корней

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Давайте вычислим $C[1,2]$:

$$C[1,2] = \min \begin{matrix} k=1: \; C[1,0] + C[2,2] + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: \; C[1,1] + C[3,2] + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{matrix} = 0.4.$$

Таким образом, из двух возможных бинарных деревьев, содержащих первые два ключа, A и B, корень оптимального дерева имеет индекс 2 (т.е. содержит B), а среднее количество сравнений при успешном поиске в этом дереве равно 0.4.

Завершите пример самостоятельно. Вы должны получить следующие окончательные таблицы:

Главная таблица

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

Таблица корней

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Таким образом, среднее количество сравнений ключей в оптимальном дереве равно 1.7. Поскольку $R[1,4] = 3$, корень оптимального дерева содержит третий

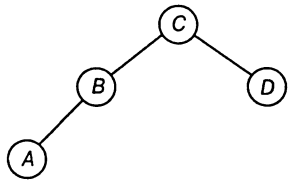


Рис. 8.11. Оптимальное бинарное дерево поиска из примера 1

ключ, т.е. C. Его левое поддерево состоит из ключей A и B, а правое содержит единственный ключ D (почему?). Чтобы определить структуру поддеревьев, мы должны сначала найти их корни с использованием таблицы корней следующим образом. Поскольку $R[1,2] = 2$, корнем оптимального дерева, содержащего A и B, является B, причем A — его левый дочерний узел (и корень дерева с одним узлом: $R[1,1] = 1$). Поскольку $R[4,4] = 4$, корнем этого оптимального дерева с одним узлом является D. Полученное оптимальное бинарное дерево поиска приведено на рис. 8.11. ■

Вот как выглядит псевдокод рассмотренного алгоритма.

АЛГОРИТМ *OptimalBST* $(P[1..n])$
```
// Поиск оптимального бинарного дерева поиска
// с использованием динамического программирования
// Входные данные:  Массив P[1..n] вероятностей поиска для
//                  отсортированного списка из n ключей
// Выходные данные: Среднее количество сравнений при
//                  успешном поиске в оптимальном бинарном
//                  дереве поиска и таблица корней
//                  поддеревьев оптимального бинарного
//                  дерева поиска R
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do        // Счетчик диагоналей
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
```

$minval \leftarrow C[i, k-1] + C[k+1, j]$; $kmin \leftarrow k$
$R[i, j] \leftarrow kmin$
$sum \leftarrow P[i]$;
for $s \leftarrow i+1$ **to** j **do**
$sum \leftarrow sum + P[s]$
$C[i, j] \leftarrow minval + sum$
return $C[1, n], R$

Очевидно, что алгоритму требуется дополнительная память, квадратично зависящая от n; временна́я эффективность алгоритма — кубическая (почему?). Более точный анализ показывает, что элементы таблицы корней всегда находятся в неубывающем порядке как вдоль строк, так и вдоль столбцов. Это ограничивает значения $R[i, j]$ диапазоном $R[i, j-1], \ldots, R[i+1, j]$ и позволяет снизить время работы алгоритма до $\Theta(n^2)$.

Упражнения 8.3

1. Завершите построение оптимального бинарного дерева поиска, начатое в примере в тексте раздела.
2. **а)** Почему временна́я эффективность алгоритма $OptimalBST$ кубическая?
   б) Почему пространственная эффективность алгоритма $OptimalBST$ квадратичная?
3. Напишите псевдокод алгоритма с линейным временем работы, который генерирует оптимальное бинарное дерево поиска по таблице корней.
4. Разработайте способ вычисления за постоянное время сумм $\sum_{s=i}^{j} p_s$, используемых в алгоритме динамического программирования для построения оптимального бинарного дерева поиска.
5. Истинно или ложно следующее утверждение: корень оптимального бинарного дерева поиска всегда содержит ключ с наивысшей вероятностью поиска.
6. Как бы вы построили оптимальное бинарное дерево поиска для множества из n ключей, если вероятности поиска у всех ключей равны? Чему будет равно среднее количество сравнений в дереве, если $n = 2^k$?
7. **а)** Покажите, что количество различных бинарных деревьев поиска $b(n)$, которые могут быть построены для множества из n упорядо-

чиваемых ключей, удовлетворяет рекуррентному соотношению

$$b(n) = \sum_{k=0}^{n-1} b(k)\, b(n-1-k) \quad \text{для } n > 0, b(0) = 1.$$

б) Известно, что решение этого рекуррентного соотношения дает числа Каталана. Убедитесь в этом для $n = 1, 2, \ldots, 5$.

в) Найдите порядок роста $b(n)$. Как ответ на этот вопрос влияет на применение алгоритма исчерпывающего перебора для поиска оптимального бинарного дерева поиска?

8. Разработайте алгоритм поиска оптимальных бинарных деревьев поиска за время $\Theta(n^2)$.

9. Обобщите алгоритм построения оптимальных бинарных деревьев поиска с учетом неудачных поисков.

10. ***Перемножение цепочек матриц***. Рассмотрим задачу минимизации общего количества умножений, выполняемых при вычислении произведения n матриц

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n,$$

размеры которых — $d_0 \times d_1$, $d_1 \times d_2, \ldots, d_{n-1} \times d_n$, соответственно. Считаем, что все промежуточные произведения двух матриц вычисляются при помощи алгоритма грубой силы (основанном на определении умножения матриц).

а) Приведите пример трех матриц, для которых количество умножений при вычислении $(A_1 \cdot A_2) \cdot A_3$ и $A_1 \cdot (A_2 \cdot A_3)$ отличается не меньше чем в 1000 раз.

б) Сколько всего имеется различных способов вычислить произведение n матриц?

в) Разработайте алгоритм динамического программирования для поиска оптимального порядка перемножения n матриц.

8.4 Задача о рюкзаке и функции с запоминанием

Этот раздел мы начнем с разработки алгоритма динамического программирования для решения задачи о рюкзаке: даны n предметов с известными весами $w_1, \ldots, w_n$ и стоимостями $v_1, \ldots, v_n$ и рюкзак вместимостью W. Требуется найти наиболее ценное подмножество предметов, помещающееся в рюкзаке. (Эта задача упоминалась в разделе 3.4, где мы рассматривали ее решение методом исчерпывающего перебора.) Здесь мы считаем, что все веса и емкость рюкзака представляют

собой положительные целые числа; стоимости предметов — не обязательно целые числа.

Для разработки алгоритма динамического программирования мы должны вывести рекуррентное соотношение, которое выражает решение экземпляра задачи о рюкзаке через решения его меньших подэкземпляров. Рассмотрим экземпляр, определяемый первыми i предметами, $1 \leqslant i \leqslant n$, весами $w_1, \ldots, w_i$, стоимостями $v_1, \ldots, v_i$ и емкостью рюкзака $1 \leqslant j \leqslant W$. Пусть $V[i, j]$ — значение оптимального решения этого экземпляра, т.е. стоимость наиболее ценного подмножества из первых i предметов, которое помещается в рюкзак емкостью j. Мы можем разделить все подмножества первых i предметов, которые помещаются в рюкзак емкостью j, на две категории: те, которые не включают i-ый предмет, и те, которые его включают. Заметим следующее.

1. Среди подмножеств, которые не включают i-ый предмет, стоимость оптимального подмножества по определению равна $V[i-1, j]$.
2. Среди подмножеств, которые включают i-ый предмет (следовательно, $j - w_i \geqslant 0$), оптимальное подмножество составляется из этого предмета и оптимального подмножества первых $i-1$ предметов, которое размещается в рюкзаке емкостью $j - w_i$. Стоимость такого оптимального подмножества равна $v_i + V[i-1, j-w_i]$.

Таким образом, стоимость оптимального решения среди всех допустимых подмножеств из первых i предметов представляет собой большее из этих двух значений. Конечно, если i-ый предмет не помещается в рюкзак, стоимость оптимального подмножества, выбранного из первых i предметов, оказывается той же, что и стоимость оптимального подмножества, выбранного из первых $i-1$ предметов. Это наблюдение приводит нас к следующему рекуррентному соотношению:

$$V[i, j] = \begin{cases} \max\{V[i-1, j], v_i + V[i-1, j-w_i]\} & \text{если } j - w_i \geqslant 0, \\ V[i-1, j] & \text{если } j - w_i < 0. \end{cases} \tag{8.12}$$

Начальные условия удобно определить следующим образом:

$$V[0, j] = 0 \text{ при } j \geqslant 0, \text{ и } V[i, 0] = 0 \text{ при} i \geqslant 0. \tag{8.13}$$

Наша цель состоит в том, чтобы найти $V[n, W]$ — максимальную стоимость подмножества из n предметов, которое помещается в рюкзаке емкостью W, и само это подмножество.

На рис. 8.12 показаны значения, входящие в (8.12) и (8.13). При $i, j > 0$ для вычисления элемента таблицы на пересечении i-ой строки и j-го столбца $V[i, j]$ мы берем значение элемента в предыдущей строке и том же столбце и сумму значений v_i и элемента в предыдущей строке и столбце, отстоящем на w_i столбцов слева, и находим максимальное из них. Таким образом мы заполняем таблицу либо строка за строкой, либо столбец за столбцом.

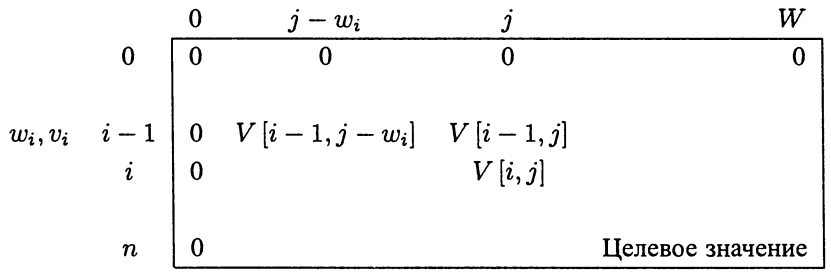


Рис. 8.12. Таблица для решения задачи о рюкзаке методом динамического программирования

Пример 1. Рассмотрим экземпляр задачи, определяемый следующими данными. Емкость рюкзака $W = 5$.

Предмет	Вес	Стоимость
1	2	12
2	1	10
3	3	20
4	2	15

Таблица динамического программирования после заполнения в соответствии с формулами (8.12) и (8.13) показана на рис. 8.13.

		Емкость j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, \quad v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, \quad v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, \quad v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, \quad v_4 = 15$	4	0	10	15	25	30	37

Рис. 8.13. Пример решения экземпляра задачи о рюкзаке при помощи алгоритма динамического программирования

Итак, максимальная стоимость $V[4, 5] = 37$. Мы можем найти состав оптимального подмножества, отслеживая вычисления этого элемента таблицы. Поскольку $V[4, 5] \neq V[3, 5]$, предмет 4 был включен в оптимальное решение вместе с оптимальным подмножеством, заполняющим оставшиеся $5 - 2 = 3$ единицы емкости рюкзака. Последние представлены элементом $V[3, 3]$. Поскольку $V[3, 3] = V[2, 3]$, элемент 3 не является частью оптимального подмножества. Да-

лее, так как $V[2,3] \neq V[1,3]$, предмет 2 также является частью оптимального выбора, после чего элемент $V[1,3-1]$ остается в качестве определения оставшейся части подмножества. Аналогично, так как $V[1,2] \neq V[0,2]$, делаем вывод, что предмет 1 является последней частью оптимального решения, которое представляет собой множество {Предмет 1, Предмет 2, Предмет 4}. ■

Как временна́я, так и пространственная эффективность данного алгоритма равна $\Theta(nW)$. Время, требующееся для поиска состава оптимального подмножества, равно $\Theta(n+W)$. Эти утверждения читателю предлагается доказать самостоятельно в качестве упражнений.

Функции с запоминанием

Как говорилось в начале главы и было показано в последующих разделах, динамическое программирование работает с задачами, решения которых удовлетворяют рекуррентным соотношениям с перекрывающимися подзадачами. Непосредственный нисходящий подход к поиску решения такого рекуррентного соотношения приводит к алгоритму, который решает общие подзадачи несколько раз, а следовательно, крайне неэффективен (обычно такой алгоритм экспоненциален или имеет еще более низкую эффективность). С другой стороны, классический подход динамического программирования работает в восходящем направлении: он заполняет таблицу решениями *всех* подзадач меньшего размера, но зато каждую он решает только один раз. Неприятной стороной такого подхода является то, что решения ряда задач меньшего размера могут быть не нужны для получения решения исходной задачи. Поскольку нисходящий подход лишен этого недостатка, представляется естественным попытаться объединить сильные стороны нисходящего и восходящего подходов. Цель заключается в том, чтобы получить метод, который позволяет решать только необходимые подзадачи и только один раз. Такой метод существует; он основан на ***функциях с запоминанием*** (memory functions) [22].

Этот метод решает поставленную задачу в нисходящем направлении, но, кроме того, поддерживает таблицу такого же вида, как и используемые в восходящих алгоритмах динамического программирования. Изначально все записи таблицы инициализируются специальным значением NULL, которое указывает, что данное значение еще не было вычислено (здесь может пригодиться метод ***виртуальной инициализации*** (virtual initialization), который рассматривался в упражнении 7.1.8). Затем, когда требуется вычислить новое значение, данный метод сначала проверяет, не было ли оно уже вычислено ранее. Если соответствующая запись в таблице не равна NULL, то из таблицы просто извлекается уже вычисленное ранее значение; в противном случае оно вычисляется при помощи рекурсивного вызова, и полученный результат записывается в таблице.

Приведенный далее алгоритм реализует эту идею для задачи о рюкзаке. После инициализации таблицы рекурсивная функция вызывается с $i = n$ (количество предметов) и $j = W$ (емкость рюкзака).

Алгоритм *MFKnapsack*(i, j)

```
// Реализация метода функций с запоминанием для решения
// задачи о рюкзаке
// Входные данные:  Натуральное i, указывающее количество
//                  первых рассматриваемых предметов,
//                  и натуральное j — емкость рюкзака
// Выходные данные: Стоимость оптимального допустимого
//                  подмножества из первых i предметов
// Примечание:      Использует входные массивы
//                  Weights[1..n], Values[1..n] и таблицу
//                  V[0..n, 0..W] (элементы которой
//                  инициализированы значениями −1, за
//                  исключением строки 0 и столбца 0,
//                  элементы которых инициализированы
//                  значениями 0) в качестве глобальных
//                  переменных
if V[i, j] < 0
  if j < Weights[i]
    value ← MFKnapsack(i − 1, j)
  else
    value ← max(MFKnapsack(i − 1, j),
      Values[i] + MFKnapsack(i − 1, j − Weights[i]))
  V[i, j] ← value
return V[i, j]
```

Пример 2. Применим метод функций с запоминанием к экземпляру задачи, рассмотренному в примере 1. На рис. 8.14 показаны результаты вычислений. Как видите, вычислены только 10 из 20 нетривиальных (т.е. не находящихся в нулевой строке или нулевом столбце) значений. Только одно нетривиальное значение, $V[1, 2]$, было повторно выбрано из таблицы вместо вычисления заново. Однако для бóльших экземпляров задачи эта пропорция может оказаться существенно большей. ■

В общем случае мы не можем ожидать повышения скорости работы алгоритма решения задачи о рюкзаке с использованием функций с запоминанием по сравнению с рассмотренным ранее более чем на некоторый постоянный множитель. Это связано с тем, что класс эффективности алгоритма с применением функций с запоминанием тот же, что и у восходящего алгоритма (почему?). Более существенного улучшения можно ожидать для алгоритмов динамического программи-

		Емкость j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,$ $v_1 = 12$	1	0	0	12	–	12	12
$w_2 = 1,$ $v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3,$ $v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2,$ $v_4 = 15$	4	0	–	–	–	–	37

Рис. 8.14. Пример решения экземпляра задачи о рюкзаке при помощи алгоритма с использованием функций с запоминанием

рования, в которых одно вычисление требует больше константного времени. Вы должны также не забывать, что метод функций с запоминанием может оказаться более расточительным в плане использования памяти, чем более эффективный в этом смысле восходящий алгоритм.

Упражнения 8.4

1. а) Воспользуйтесь восходящим алгоритмом динамического программирования для решения следующего экземпляра задачи о рюкзаке. Емкость рюкзака $W = 6$.

Предмет	Вес	Стоимость
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

б) Сколько различных оптимальных подмножеств имеет экземпляр задачи из части *а*)?

в) Как в общем случае мы можем воспользоваться таблицей, сгенерированной алгоритмом динамического программирования, чтобы выяснить, имеется ли у данного экземпляра задачи о рюкзаке более одного оптимального подмножества.

2. а) Напишите псевдокод восходящего алгоритма динамического программирования для задачи о рюкзаке.

б) Напишите псевдокод алгоритма, который находит состав оптимального подмножества по таблице, сгенерированной восходящим алгоритмом динамического программирования для задачи о рюкзаке.

3. Докажите для восходящего алгоритма динамического программирования для задачи о рюкзаке:

а) что его временна́я эффективность равна $\Theta(nW)$,

б) что его пространственная эффективность равна $\Theta(nW)$,

в) что время, требующееся для поиска состава оптимального подмножества по заполненной алгоритмом динамического программирования таблице, равно $\Theta(n+W)$.

4. а) Истинно или ложно следующее утверждение: последовательность стоимостей в строке таблицы динамического программирования для экземпляра задачи о рюкзаке всегда неубывающая.

б) Истинно или ложно следующее утверждение: последовательность стоимостей в столбце таблицы динамического программирования для экземпляра задачи о рюкзаке всегда неубывающая.

5. Примените метод функций с запоминанием для решения экземпляра задачи о рюкзаке из упражнения 1. Укажите элементы таблицы динамического программирования, которые 1) не вычисляются при использовании метода функций с запоминанием; 2) выбираются из таблицы без вычисления.

6. Докажите, что класс эффективности алгоритма с функциями с запоминанием для задачи о рюкзаке тот же, что и у восходящего алгоритма (см. упражнение 3).

7. Напишите псевдокод функции с запоминанием для задачи об оптимальном бинарном дереве поиска (можете ограничиться поиском наименьшего количества сравнений ключей при успешном поиске).

8. Приведите две причины, по которым подход с использованием функций с запоминанием не подходит для задачи вычисления биномиальных коэффициентов.

9. Разработайте алгоритм динамического программирования для ***задачи о размене*** (change-making problem): дана величина n и неограниченное количество монет достоинством $d_1, d_2, \ldots, d_m$. Требуется найти наименьшее количество монет, которые в сумме дают величину n, или указать, что задача не имеет решения.

10. Напишите реферат об одном из хорошо известных применений динамического программирования:

а) Поиск наибольшей общей подпоследовательности двух последовательностей.

б) Оптимальное редактирование строк.

в) Минимальная триангуляция многоугольника.

Резюме

- *Динамическое программирование* представляет собой метод решения задач с перекрывающимися подзадачами. Обычно такие подзадачи возникают из рекуррентных соотношений, связывающих решение поставленной задачи с решениями меньших подзадач того же вида. Динамическое программирование предполагает, что решение каждой из меньших подзадач находится только один раз, и записывает полученные результаты в таблицу, из которой затем получается решение исходной задачи.
- Применимость динамического программирования к задачам оптимизации требует, чтобы задача удовлетворяла *принципу оптимальности*: оптимальное решение любого из ее экземпляров должно слагаться из оптимальных решений ее подэкземпляров.
- Вычисление биномиального коэффициента путем построения треугольника Паскаля может рассматриваться как применение метода динамического программирования к задаче, не являющейся задачей оптимизации.
- *Алгоритм Воршалла* для поиска транзитивного замыкания и *алгоритм Флойда* для решения задачи поиска кратчайших путей между всеми парами вершин основаны на идее, которую можно рассматривать как применение метода динамического программирования.
- Динамическое программирование можно использовать для построения *оптимального бинарного дерева поиска* для данного множества ключей и известных вероятностей их поиска.
- Решение задачи о рюкзаке при помощи алгоритма динамического программирования является примером применения этого метода к сложным задачам комбинаторной оптимизации.
- Метод *функций с запоминанием* призван объединить сильные стороны нисходящего и восходящего подходов к решению задач с перекрывающимися подзадачами. Это достигается решением задачи в нисходящем направлении, с однократным решением только необходимых подзадач исходной задачи и записью полученных решений в таблице.

Глава 9

Жадные методы

Жадность, за отсутствием лучшего слова, — это здорово! Это правильно!

— Майкл Дуглас (Michael Douglas),
американский актер, в роли Гордона Геко (Gordon Gecko) в фильме
"Уолл-стрит" (Wall Street), 1987

Начнем с ***задачи о размене*** (change-making problem), которая постоянно встает перед миллионами кассиров во всем мире: как выплатить сумму n при помощи наименьшего количества монет номиналом $d_1 > d_2 > \cdots > d_m$, использующихся в той или иной стране. Например, в США широко распространены монеты достоинством $d_1 = 25$ центов (quarter), $d_2 = 10$ центов (dime (гривенник)), $d_3 = 5$ центов (nickel (пятак)) и $d_4 = 1$ цент (penny). Как дать такими монетами сдачу, например, в 48 центов? Если вы ответите, что это одна монета в 25 центов, две — по 10 центов и три — по 1 центу, то, сознательно или нет, вы следуете стратегии, которая позволяет сделать оптимальный выбор среди возможных альтернатив. На первом шаге вы можете выбрать монету любого номинала из четырех приведенных. "Жадное" мышление приводит вас к мысли, что это должна быть монета в 25 центов, что максимально снизит остающуюся к выдаче сумму — а именно до 23 центов. На втором шаге вы уже не можете выбрать монету в 25 центов, поскольку это нарушило бы ограничения, поставленные в условии задачи. Поэтому наилучшим выбором в данной ситуации оказывается выбор монеты в 10 центов, что снижает сумму до 13 центов. Еще один выбор монеты в 10 центов приводит нас к остатку в 3 цента, которые можно дать только тремя монетами по 1 центу.

Является ли это решение экземпляра задачи о размене оптимальным? Да. Можно доказать, что жадный алгоритм для данных номиналов монет дает оптимальное решение для любой суммы, выражающейся натуральным числом. В то же время легко привести пример необычных номиналов монет (например, $d_1 = 7$, $d_2 = 5$, $d_3 = 1$), которые для определенных сумм могут дать некорректные решения (именно по этой причине при разработке алгоритма динамического программирования для данной задачи в упражнении 8.4.9 оговаривалось, что алго-

ритм либо должен вернуть оптимальное решение, либо сообщить об отсутствии решения).

Такой подход к решению задачи о размене называется ***жадным*** (greedy). Кибернетики рассматривают его как общий метод проектирования алгоритмов, несмотря на то, что он применим только к задачам оптимизации. Жадный подход предполагает построение решения посредством выбора последовательности шагов, на каждом из которых получается частичное решение поставленной задачи, пока не будет получено полное решение. При этом на каждом шаге — и это является главным в рассматриваемом методе — выбор должен быть

- *допустимым*, т.е. удовлетворять ограничениям задачи;
- *локально оптимальным*, т.е. наилучшим локальным выбором среди всех допустимых вариантов, доступных на каждом шаге;
- *окончательным*, т.е. будучи сделан, он не может быть изменен последующими шагами алгоритма.

Эти требования поясняют название метода: на каждом шаге он предполагает "жадный" выбор наилучшей доступной альтернативы в надежде, что последовательность локально оптимальных выборов приведет к (глобально) оптимальному решению всей задачи. Мы не будем вдаваться в философский спор о том, хорошо ли быть жадным (если вы не смотрели фильм, упомянутый в эпиграфе, то для справки: его герой плохо кончил). С точки зрения алгоритмов нас интересует лишь то, работает метод или нет. Как мы увидим, существуют задачи, для которых последовательность локально оптимальных выборов приводит к оптимальному решению для любого экземпляра рассматриваемой задачи. Однако есть и другие задачи, для которых это не так; для задач такого рода жадный алгоритм может представлять интерес только в том случае, если нас устраивает приближенное решение.

В двух первых разделах главы мы рассмотрим два классических алгоритма для решения задачи о минимальном остовном дереве: алгоритмы Прима и Крускала. В этих алгоритмах примечателен тот факт, что они решают одну и ту же задачу с применением жадного подхода различными способами, и оба дают оптимальное решение. В разделе 9.3 мы рассмотрим еще один классический алгоритм — алгоритм Дейкстры для поиска кратчайшего пути во взвешенном графе. Раздел 9.4 посвящен деревьям Хаффмана и их основному приложению — кодам Хаффмана — важному методу сжатия данных, который может рассматриваться как применение жадной технологии. Наконец, в разделе 11.3 вы встретитесь с несколькими примерами приближенных алгоритмов, основанных на жадном методе.

Как правило, жадные алгоритмы интуитивно привлекательны и просты. Несмотря на несомненную простоту, за ними стоит сложная теория, основанная на абстрактной комбинаторной структуре, которая называется "матроид" (matroid).

Мы не будем рассматривать матроиды; заинтересованным читателям рекомендуем обратиться к другим книгам, например к [32].

9.1 Алгоритм Прима

Во многих практических ситуациях естественным образом возникает следующая задача: требуется соединить n данных точек таким образом, чтобы из каждой точки существовал путь в любую другую, причем суммарная стоимость соединений должна быть минимальна. Точки можно представить вершинами графа, а стоимости соединений — весами ребер. В такой модели задача превращается в задачу о минимальном остовном дереве, формально определяемом следующим образом.

Определение 1. ***Остовное дерево*** (spanning tree) связного графа представляет собой связный ациклический подграф (т.е. дерево), которое содержит все вершины графа. ***Минимальное остовное дерево*** (minimum spanning tree) взвешенного связного графа представляет собой остовное дерево с наименьшим весом, где ***вес*** дерева определяется как сумма весов всех его ребер. Задача о минимальном остовном дереве представляет собой задачу поиска минимального остовного дерева для данного взвешенного связного графа. ■

На рис. 9.1 показан простой пример, иллюстрирующий приведенные определения.

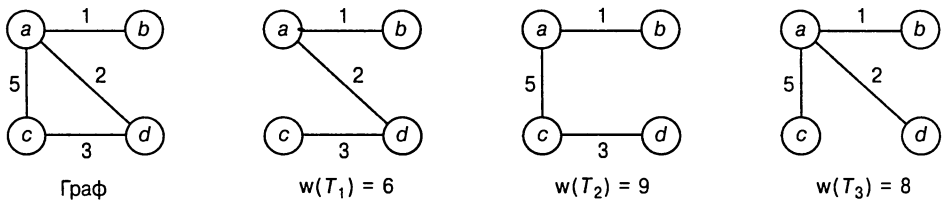


Рис. 9.1. Граф и его остовные деревья. T_1 — минимальное остовное дерево

Если мы попытаемся применить к построению минимального остовного дерева метод исчерпывающего перебора, то встретим два серьезных препятствия. Во-первых, количество остовных деревьев экспоненциально растет с ростом размера графа (как минимум для плотных графов). Во-вторых, генерация всех остовных деревьев для графа — задача непростая; в действительности она гораздо сложнее поиска *минимального* остовного дерева взвешенного графа одним из эффективных алгоритмов для решения этой задачи. В этом разделе мы рассмотрим ***алгоритм Прима*** (Prim's algorithm), который был разработан еще в 1957 году [92].

Алгоритм Прима строит минимальное остовное дерево как последовательность расширяющихся поддеревьев. Начальное поддерево такой последовательности состоит из единственной вершины, произвольно выбранной из множества вершин графа V. На каждой итерации мы расширяем текущее дерево жадным образом, добавляя к нему ближайшую вершину, не входящую в дерево (под ближайшей вершиной подразумевается вершина, не входящая в дерево и соединенная с вершиной дерева ребром с минимальным весом. Неоднозначности разрешаются произвольным образом). Алгоритм завершает работу после того, как все вершины оказываются включены в строящееся дерево. Поскольку алгоритм расширяет дерево по одной вершине за итерацию, общее количество таких итераций равно $n - 1$, где n — количество вершин графа. Дерево, сгенерированное этим алгоритмом, представляет собой множество ребер, использованных при расширении дерева.

Вот как выглядит псевдокод данного алгоритма.

Алгоритм $Prim\,(G)$

// Алгоритм Прима построения минимального остовного дерева
// **Входные данные:** Взвешенный связный граф $G = \langle V, E \rangle$
// **Выходные данные:** E_T, множество ребер, составляющих
// минимальное остовное дерево G
$V_T \leftarrow \{v_0\}$ // Множество вершин дерева инициализируется
// произвольной вершиной
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
Поиск ребра с минимальным весом $e^* = (v^*, u^*)$ среди всех
ребер (v, u) таких, что $v \in V_T$ и $u \in V - V_T$
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Природа алгоритма Прима заставляет снабдить каждую вершину, не входящую в текущее дерево, информацией о кратчайшем ребре, соединяющем ее с вершиной дерева. Мы можем сделать это, присоединяя к вершинам по две метки: имя ближайшей вершины дерева и длину (вес) соответствующего ребра. Вершины, не являющиеся смежными ни для одной из вершин дерева, могут быть помечены меткой ∞, указывающей "бесконечное" расстояние до вершин дерева и нулевым значением в поле ближайшей вершины дерева. (Альтернативный вариант состоит в разделении всех вершин, не входящих в дерево, на два множества — "приграничных" и "незамеченных". Приграничное множество содержит только те вершины, которые, не принадлежа дереву, смежны по крайней мере с одной из его вершин. Незамеченные вершины — все остальные вершины графа, которые называются так потому, что еще не исследованы алгоритмом.) При использовании таких ме-

ток поиск следующей добавляемой в текущее дерево $T = \langle V_T, E_T \rangle$ вершины становится простой задачей поиска вершины с наименьшей меткой расстояния среди вершин множества $V - V_T$. Разрешение неоднозначностей выполняется произвольным образом.

После того как мы найдем вершину u^*, которая должна быть добавлена в дерево, надо выполнить следующие две операции.

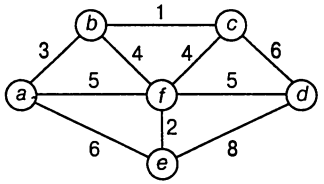
- Переместить вершину u^* из множества $V - V_T$ во множество вершин дерева V_T.
- Для каждой вершины из остающихся во множестве $V - V_T$ и связанных с u^* более коротким ребром, чем ее текущая метка расстояния, следует обновить ее метку имени смежной вершины именем u^*, а метку расстояния — расстоянием до вершины u^*.[1]

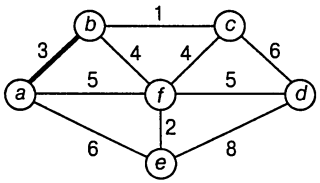
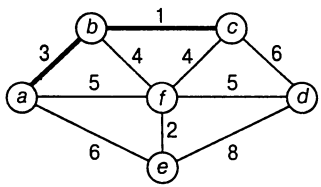
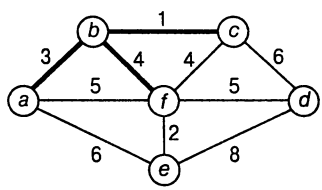
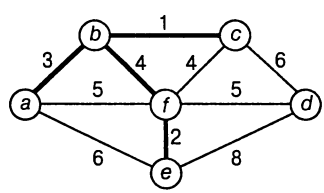
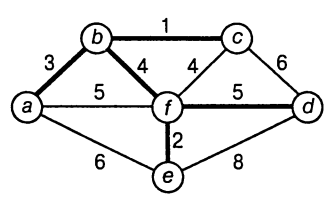
На рис. 9.2 продемонстрировано применение алгоритма Прима к показанному на рисунке графу.

Всегда ли алгоритм Прима дает минимальное остовное дерево? Ответ на этот вопрос — да. Докажем по индукции, что каждое поддерево T_i, $i = 0, \ldots, n-1$, генерируемое алгоритмом Прима, является частью (т.е. подграфом) некоторого минимального остовного дерева. (Отсюда, очевидно, непосредственно следует, что последнее дерево последовательности, T_{n-1}, представляет собой само минимальное остовное дерево, поскольку содержит все n вершин графа.) Базис индукции тривиален, поскольку дерево T_0 состоит из одной вершины, а значит, должно быть частью любого минимального остовного дерева. Для выполнения шага индукции предположим, что T_{i-1} является частью некоторого минимального остовного дерева T. Надо доказать, что дерево T_i, сгенерированное алгоритмом Прима из дерева T_{i-1}, также является частью минимального остовного дерева. Мы докажем это от противного, предположив, что не существует минимального остовного дерева графа, содержащего T_i. Пусть $e_i = (v, u)$ — ребро минимального веса от вершины в дереве T_{i-1} к вершине, которая не входит в T_{i-1} и используется алгоритмом Прима для расширения дерева T_{i-1} до T_i. В соответствии с нашими предположениями e_i не может принадлежать ни одному из минимальных остовных деревьев, включая T. Таким образом, если мы добавим e_i к дереву T, должен образоваться цикл (рис. 9.3).

Кроме ребра $e_i = (v, u)$ цикл должен содержать другое ребро (v', u'), которое соединяет вершину $v' \in T_{i-1}$ с вершиной $u' \notin T_{i-1}$ (v' может совпадать с v или u' может совпадать с u, но не оба одновременно). Если теперь мы удалим ребро (v', u') из цикла, то получим другое остовное дерево всего графа, вес которого не превышает веса дерева T, поскольку вес e_i не превышает веса (v', u').

[1] При использовании реализации с разделением на приграничные и незамеченные вершины все незамеченные вершины, смежные с u^*, должны быть перенесены во множество приграничных вершин.



Вершины дерева	Остальные вершины*	Рисунок
a(–,–)	**b(a,3)** c(–,∞) d(–,∞) e(a,6) f(a,5)	
b(a,3)	**c(b,1)** d(–,∞) e(a,6) f(b,4)	
c(b,1)	d(c,6) e(a,6) **f(b,4)**	
f(b,4)	d(f,5) **e(f,2)**	
e(f,2)	**d(f,5)**	
d(f,5)		

* В скобках показаны метки, указывающие ближайшую вершину дерева и вес ребра; выбираемые для добавления в дерево вершины показаны полужирным шрифтом

Рис. 9.2. Применение алгоритма Прима

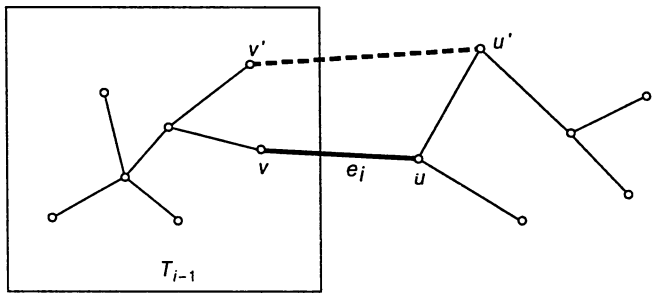


Рис. 9.3. Доказательство корректности алгоритма Прима

Следовательно, это остовное дерево является минимальным остовным деревом, что входит в противоречие с предположением, что не существует минимальное остовное дерево, содержащее T_i. Тем самым доказывается корректность алгоритма Прима.

Какова эффективность алгоритма Прима? Ответ зависит от того, какие структуры данных выбраны для представления графа и для очереди с приоритетами для вершин из множества $V - V_T$ (приоритетами вершин являются расстояния от них до ближайших вершин в текущем дереве; взгляните еще раз на рис. 9.2 и внимательное рассмотрите, как работает очередь с приоритетами в алгоритме Прима). Например, если граф представлен матрицей весов, а очередь с приоритетами реализована при помощи неупорядоченного массива, время работы алгоритма будет равно $\Theta(|V|^2)$. В самом деле, на каждой из $|V|-1$ итераций требуется полный обход массива для поиска и удаления элемента с минимальным расстоянием (и, при необходимости, для последующего обновления приоритетов оставшихся вершин).

Очередь с приоритетами можно реализовать и как ***неубывающую пирамиду***, которая представляет собой зеркальное отражение пирамиды, рассматривавшейся в разделе 6.4 (на самом деле ее можно реализовать просто путем изменения знака всех значений ключей). Неубывающая пирамида является полным бинарным деревом, в котором каждый элемент не превышает дочерние элементы. Все основные свойства пирамид выполняются и для неубывающих пирамид, с некоторыми очевидными модификациями. Например, корень неубывающей пирамиды содержит наименьший, а не наибольший элемент. Удаление наименьшего элемента и вставка нового элемента в неубывающую пирамиду размером n представляют собой операции, принадлежащие классу эффективности $O(\log n)$, как и операция изменения приоритета элемента (см. упражнение 9.1.10).

Если граф представлен с помощью связанного списка смежности, а очередь с приоритетами реализована с использованием неубывающей пирамиды, время работы алгоритма Прима равно $O(|E| \log |V|)$, поскольку алгоритм выполняет $|V| - 1$ удалений наименьшего элемента и делает $|E|$ проверок (и, возможно,

изменений приоритета) элементов в неубывающей пирамиде размером не более $|V|$. Каждая из этих операций, как упоминалось ранее, принадлежит классу эффективности $O(\log|V|)$. Следовательно, общее время работы алгоритма Прима равно

$$(|V|-1+|E|)\,O(\log|V|)=O(|E|\log|V|),$$

поскольку в связном графе $|V|-1\leqslant|E|$.

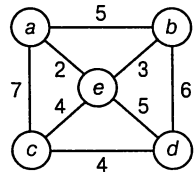
В следующем разделе мы познакомимся с еще одним жадным алгоритмом для решения задачи о минимальном остовном дереве, который работает "жадным" способом, отличным от способа алгоритма Прима.

Упражнения 9.1

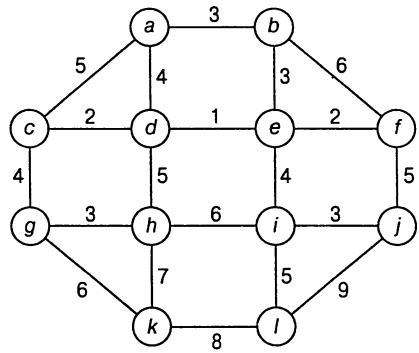
1. Приведите пример экземпляра задачи о размене, для которой жадный алгоритм не дает оптимального решения.
2. Напишите псевдокод жадного алгоритма для задачи о размене для суммы n и монет с номиналами $d_1 > d_2 > \cdots > d_m$ в качестве входных данных. Чему равна временна́я эффективность вашего алгоритма как функция от n?
3. Рассмотрим задачу составления расписания выполнения n заданий с известными продолжительностями $t_1,\ldots,t_n$ на одном процессоре. Задания могут выполняться по одному в любом порядке. Необходимо найти расписание, которое минимизирует время, затраченное на все задания в системе (время, затрачиваемое на задание, представляет собой сумму времени, затраченного на выполнение задания, и времени ожидания выполнения задания).
   а) Разработайте жадный алгоритм для решения этой задачи.
   б) Всегда ли жадный алгоритм приводит к оптимальному решению?
4. **а)** Разработайте жадный алгоритм для задачи о назначениях (см. раздел 3.4).
   б) Всегда ли жадный алгоритм приводит к оптимальному решению?

5. ***Задача о гирях***. Найдите оптимальное множество n весов гирь $\{w_1, w_2, \ldots, w_n\}$, позволяющее взвесить на рычажных весах любой целочисленный вес от 1 до наибольшего возможного значения W, в случае, если
   а) гири могут размещаться только на одной чашке весов;
   б) гири могут размещаться на обеих чашках весов.

6. а) Примените алгоритм Прима к приведенному графу. Включите в очередь с приоритетами все вершины, которые не входят в дерево.



б) Примените алгоритм Прима к приведенному графу. Включите в очередь с приоритетами только пограничные вершины (которые не входят в текущее дерево, но каждая из которых смежна по крайней мере одной из вершин дерева).



7. Понятие минимального остовного дерева применимо к связному взвешенному графу. Требуется ли проверять связность графа перед применением к нему алгоритма Прима, или он способен выяснить это самостоятельно?

8. а) Как можно воспользоваться алгоритмом Прима для поиска остовного дерева связного графа без весов его ребер?

б) Насколько этот алгоритм подходит для решения поставленной задачи?

9. Докажите, что любой взвешенный связный граф, у которого веса всех ребер различны, имеет только одно минимальное остовное дерево.

10. Набросайте эффективный алгоритм для изменения значений элементов в неубывающей пирамиде. Какова временна́я эффективность вашего алгоритма?

9.2 Алгоритм Крускала

В предыдущем разделе мы рассмотрели жадный алгоритм, который "выращивает" минимальное остовное дерево посредством жадного включения в него вершины, ближайшей к вершинам дерева. Интересно, что существует и другой жадный алгоритм, который решает задачу построения минимального остовного дерева и также дает оптимальное решение. Это — ***алгоритм Крускала*** (Kruskal's algorithm) [71], названный так по имени Джозефа Крускала (Joseph Kruskal), который открыл его, будучи студентом-второкурсником. Алгоритм Крускала ищет минимальное остовное дерево взвешенного связного графа $G = \langle V, E \rangle$ как ациклический подграф с $|V| - 1$ ребрами, сумма весов которых минимальна (нетрудно доказать, что такой подграф должен быть деревом). Следовательно, алгоритм строит минимальное остовное дерево как расширяемую последовательность подграфов, которые всегда ациклични, но на промежуточных стадиях не всегда связны.

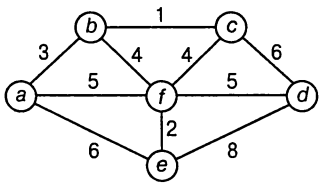
Алгоритм начинает с сортировки ребер графа в неубывающем порядке их весов. Затем, начиная с пустого подграфа, просматривает отсортированный список и добавляет очередное ребро в список текущего подграфа, если при этом не создается цикл; в противном случае ребро просто пропускается.

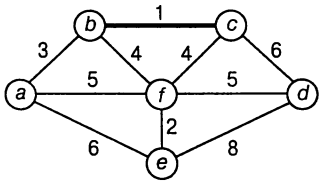
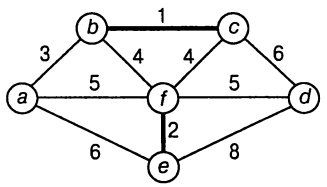
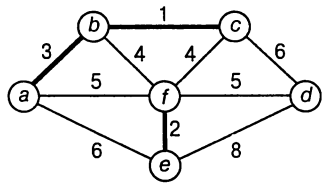
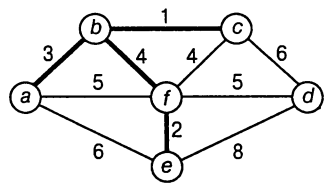
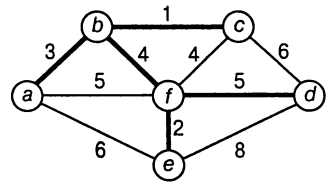
Алгоритм $Kruskal(G)$

```
// Алгоритм Крускала построения минимального остовного
// дерева
// Входные данные:  Взвешенный связный граф G = ⟨V, E⟩
// Выходные данные: Множество ребер E_T, составляющее
//                  минимальное остовное дерево графа G
Сортировка множества E в неубывающем порядке весов
ребер w(e_{i_1}) ⩽ ··· ⩽ w(e_{i_|E|})
E_T ← ∅                  // Инициализация множества ребер дерева
ecounter ← 0             // и его размера
k ← 0                    // Инициализация количества обработанных ребер
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} — ациклический граф
        E_T ← E_T ∪ {e_{i_k}}; ecounter ← ecounter + 1
return E_T
```

Корректность алгоритма Крускала можно доказать, повторяя основные шаги доказательства алгоритма Прима, приведенного в предыдущем разделе. Тот факт, что E_T в алгоритме Прима является деревом, а в алгоритме Крускала в общем случае — ациклическим подграфом, — трудность, которую несложно преодолеть.

На рис. 9.4 продемонстрировано применение алгоритма Крускала к рассматривавшемуся в разделе 9.1 графу, где к нему был применен алгоритм Прима. При



Вершины дерева	Отсортированный список ребер*	Рисунок
	bc ef ad bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ad bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ad** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ad **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ad bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

* Выбранные ребра показаны полужирным шрифтом

Рис. 9.4. Применение алгоритма Крускала

рассмотрении выполняемых алгоритмом операций обратите внимание на несвязность некоторых промежуточных графов.

Применение алгоритмов Прима и Крускала к одному и тому же небольшому графу вручную может создать ложное ощущение большей простоты алгоритма Крускала. Беда в том, что на каждой итерации алгоритм Крускала должен проверить, не приведет ли добавление очередного ребра к появлению цикла. Нетрудно увидеть, что новый цикл образуется тогда и только тогда, когда новое ребро соединяет две вершины, уже соединенные некоторым путем, т.е. тогда и только тогда, когда эти две вершины принадлежат одному и тому же связному компоненту (рис. 9.5). Заметим также, что каждый связный компонент подграфа, генерируемого алгоритмом Крускала, является деревом, поскольку не содержит циклов.

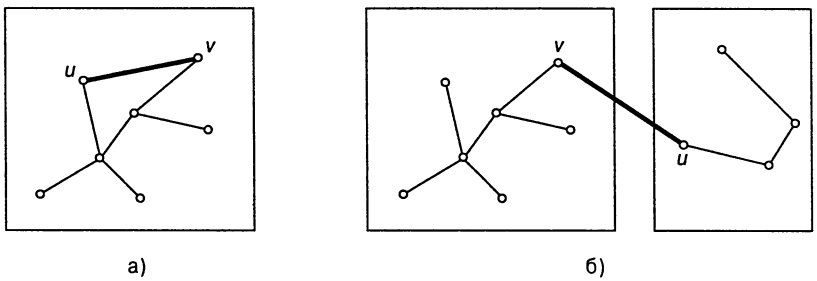


Рис. 9.5. Новое ребро, соединяющее две вершины, может как *а*) образовать цикл, так и *б*) не образовывать его

С учетом этого удобнее использовать немного отличную интерпретацию алгоритма Крускала. Мы можем рассматривать операции алгоритма как продвижение по последовательности лесов, содержащих *все* вершины исходного графа и только *некоторые* из его ребер. Изначально лес состоит из $|V|$ тривиальных деревьев, каждое из которых представляет собой отдельную вершину графа. Конечный лес представляет собой единое дерево, являющееся минимальным остовным деревом графа. На каждой итерации алгоритм выбирает очередное ребро (u, v) из отсортированного списка ребер графа, находит деревья, содержащие вершины u и v, и, если это различные деревья, объединяет их в одно большее дерево путем добавления ребра (u, v).

К счастью, имеется эффективный алгоритм для выполнения этих действий, включая самое важное — выяснение того, принадлежат ли две вершины одному дереву. Он называется ***алгоритмом поиска объединений*** (union-find algorithm) и будет рассмотрен в следующем подразделе. При использовании эффективного алгоритма поиска объединений время работы алгоритма Крускала определяется временем, необходимым для сортировки ребер исходного графа. Следовательно, применяя эффективный алгоритм сортировки, мы найдем, что время работы алгоритма Крускала равно $\Theta(|E| \log |E|)$.

Непересекающиеся подмножества и алгоритмы поиска объединений

Алгоритм Крускала — один из множества алгоритмов, которым требуется динамическое разделение некоторого n-элементного множества S на непересекающиеся подмножества $S_1, S_2, \ldots, S_k$. После инициализации набора из n одноэлементных подмножеств, каждое из которых содержит по одному элементу множества S, этот набор подвергается последовательности операций объединений и поисков (заметим, что количество операций объединения в любой такой последовательности операций не может превышать $n-1$, так как каждое объединение уменьшает количество подмножеств как минимум на 1, а во всем множестве S всего содержится n элементов). Таким образом, мы имеем дело с абстрактным типом данных, который представляет набор непересекающихся подмножеств некоторого конечного множества и поддерживает следующие операции:

$makeset\,(x)$ создает одноэлементное множество $\{x\}$. Предполагается, что эта операция может быть применена к каждому элементу множества S только один раз;

$find\,(x)$ находит подмножество, содержащее x;

$union\,(x, y)$ строит объединение непересекающихся подмножеств S_x и S_y, содержащих, соответственно, x и y, и добавляет его в набор вместо S_x и S_y, которые из набора удаляются.

Пусть, например, $S = \{1, 2, 3, 4, 5, 6\}$. Тогда $make\,(i)$ создает множество $\{i\}$, и применение этой операции шесть раз инициализирует набор шестью одноэлементными множествами

$$\{1\}, \quad \{2\}, \quad \{3\}, \quad \{4\}, \quad \{5\}, \quad \{6\}.$$

Выполнение операций $union\,(1, 4)$ и $union\,(5, 2)$ дает набор

$$\{1, 4\}, \quad \{5, 2\}, \quad \{3\}, \quad \{6\}.$$

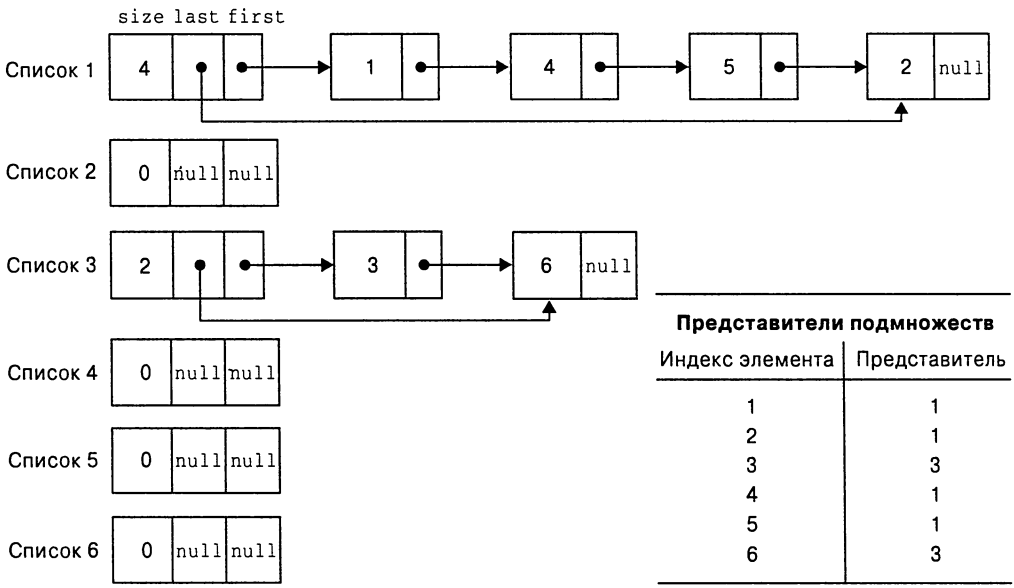
Дальнейшее выполнение $union\,(4, 5)$ и $union\,(3, 6)$ дает непересекающиеся подмножества

$$\{1, 4, 5, 2\} \text{ и } \{3, 6\}.$$

Большинство реализаций этого абстрактного типа данных используют по одному элементу из каждого из непересекающихся подмножеств набора в качестве ***представителя*** (representative). Некоторые реализации не накладывают на представителей каких-либо особых ограничений; другие требуют, например, чтобы представитель был наименьшим элементом подмножества. Кроме того, обычно считается, что элементы множества представляют собой (или могут быть отображены на) целые числа. Имеется две основные альтернативные реализации этой

структуры данных. Первая, называющаяся ***быстрым поиском*** (quick find), оптимизирует временну́ю эффективность операции поиска; вторая — ***быстрое объединение*** (quick union), как следует из названия, оптимизирует временну́ю эффективность операции объединения.

Быстрый поиск использует массив, индексированный элементами множества S; значения массива указывают представителей подмассивов, содержащих соответствующие элементы. Каждое подмножество реализовано как связный список, заголовок которого содержит указатели на первый (`first`) и последний (`last`) элементы списка наряду с количеством элементов списка (`size`) (пример такой реализации показан на рис. 9.6).



Представители подмножеств

Индекс элемента	Представитель
1	1
2	1
3	3
4	1
5	1
6	3

Рис. 9.6. Представление подмножеств $\{1,4,5,2\}$ и $\{3,6\}$ с использованием связанных списков по методу быстрого поиска. Подмножества получены после выполнения операций $union\,(1,4)$, $union\,(5,2)$, $union\,(4,5)$ и $union\,(3,6)$. Списки нулевого размера рассматриваются как удаленные из набора

При использовании такой схемы реализация $makeset\,(x)$ требует присваивания соответствующего элемента в массиве представителей элементу, индексированному значением x, и инициализации соответствующего связанного списка единственным узлом со значением x. Временна́я эффективность такой операции, очевидно, равна $\Theta\,(1)$, следовательно, инициализация n одноэлементных подмножеств требует $\Theta\,(n)$ времени. Эффективность $find\,(x)$ также равна $\Theta\,(1)$: все, что нам надо, — это получить представителя x из массива представителей. Выполнение $union\,(x,y)$ более продолжительное. Простейшее решение состоит в присоединении связанного списка y в конец связанного списка x, обновлении

информации о представителе для всех элементов списка y и удалении списка y из набора. Однако, как легко убедиться, при использовании описанного алгоритма последовательность операций объединения

$$union(2,1), union(3,2), \ldots, union(i+1,i), \ldots, union(n,n-1)$$

выполняется за время $\Theta(n^2)$, что слишком медленно по сравнению с некоторыми известными альтернативами.

Простой путь повышения общей эффективности последовательности операций *union* состоит в том, чтобы всегда добавлять более короткий из двух списков к более длинному, разрешая неоднозначности произвольным образом. Конечно, предполагается, что размер списка доступен (например, при помощи хранения количества элементов в заголовке списка). Такая модификация называется ***объединением по размеру*** (union by size). Хотя она и не улучшает эффективность единичного применения операции *union* в наихудшем случае (она остается равной $\Theta(n)$), время работы в наихудшем случае произвольной корректной последовательности операций *union-by-size* равно $O(n \log n)$.[2]

Вот доказательство этого утверждения. Пусть a_i — элемент множества S, с подмножествами которого мы работаем, и пусть A_i — количество обновлений представителя a_i при выполнении последовательности операций *union-by-size*. Насколько большим может быть значение A_i, если множество S состоит из n элементов? Каждый раз при обновлении представителя a_i элемент a_i должен находиться в меньшем подмножестве, участвующем в объединении, так что получающееся в результате объединение должно содержать как минимум вдвое больше элементов, чем в подмножестве, содержавшем a_i. Следовательно, при первом обновлении представителя a_i получающееся объединение должно состоять как минимум из двух элементов; после второго — как минимум из четырех. В общем случае при обновлении A_i раз получающееся в результате множество должно содержать как минимум 2^{A_i} элементов. Поскольку все множество S содержит n элементов, $2^{A_i} \leqslant n$ и, следовательно, $A_i \leqslant \log_2 n$. Таким образом, общее количество возможных обновлений для всех n элементов S не может превышать $n \log_2 n$.

Итак, при использовании операции *union-by-size* временная эффективность последовательности не более $n-1$ объединений и m поисков равна $O(n \log n + m)$.

Альтернативный метод ***быстрого объединения*** представляет каждое подмножество в виде корневого дерева. Узлы дерева содержат элементы подмножества, по одному в узле; корневой узел рассматривается как представитель подмножества. Ребра дерева направлены от дочерних узлов к родительским (рис. 9.7). Кроме

[2] Это — пример полезности *амортизированной эффективности*, о которой упоминалось в главе 2. Временна́я эффективность любой последовательности из n операций *union-by-size* выше эффективности в наихудшем случае отдельной операции, выполненной n раз.

того, поддерживается не показанное на рисунке из соображений простоты отображение множества элементов на соответствующие им узлы дерева, реализованное, например, при помощи массива указателей.

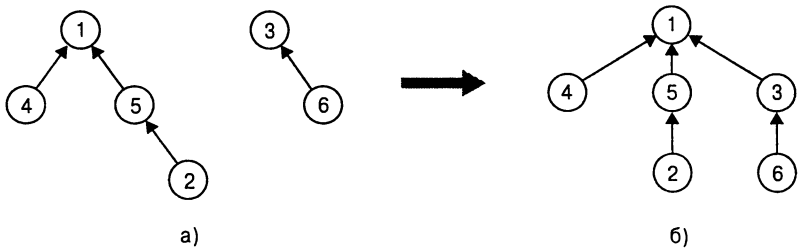


Рис. 9.7. а) Представление методом быстрого объединения подмножеств $\{1,4,5,2\}$ и $\{3,6\}$ в виде леса.$\{1,4,5,2\}$ $\{3,6\}$ б) Результат выполнения операции $union\,(5,6)$

При такой реализации операция $makeset\,(x)$ создает одноузловое дерево за время $\Theta\,(1)$; следовательно, инициализация n одноэлементных подмножеств требует $\Theta\,(n)$ времени. Операция $union\,(x,y)$ реализуется присоединением корня дерева y к корню дерева x (и удалением дерева y из набора, делая указатель на его корень нулевым). Очевидно, что временна́я эффективность данной операции равна $\Theta\,(1)$. Операция $find\,(x)$ выполняется путем следования по указателям на родительский узел до корня дерева (элемент которого возвращается в качестве представителя подмножества). Соответственно, временна́я эффективность одной операции $find$ равна $O\,(n)$, поскольку дерево, представляющее подмножество, может выродиться в связный список с n узлами.

Эту временну́ю границу можно улучшить. Простейший путь для достижения этого заключается в том, чтобы всегда при выполнении операции $union$ присоединять меньшее дерево к корню большего дерева, с произвольным разрешением неоднозначностей. Размер дерева можно измерять либо количеством входящих в него узлов (эта версия называется ***объединением по размеру*** (union by size)), либо его высотой (эта версия называется ***объединением по рангу*** (union by rank)). Конечно, эти варианты требуют хранения для каждого узла дерева либо количества его потомков, либо высоты поддерева, для которого данный узел является корнем. Можно легко доказать, что в любом случае высота дерева будет логарифмической, что делает возможным выполнение каждого поиска за время $O\,(\log n)$. Таким образом, при *быстром объединении* временна́я эффективность последовательности не более $n-1$ объединений и m поисков равна $O\,(n+m\log n)$.

В действительности можно получить еще более высокую эффективность, если скомбинировать любой из описанных методов быстрого объединения со ***сжатием пути*** (path compression). Эта модификация заставляет каждый узел, встреченный в процессе выполнения операции поиска, указывать на корень дерева (рис. 9.8).

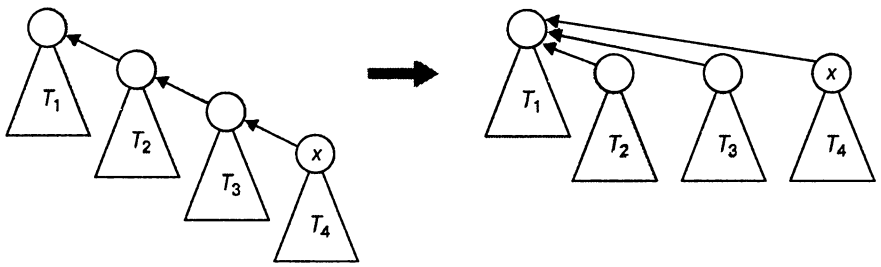


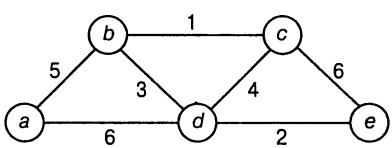
Рис. 9.8. Сжатие пути

Согласно достаточно сложному анализу, который превосходит уровень данной книги (см. [115]), этот и подобные методы повышают эффективность последовательности из не более $n - 1$ объединения и m поисков до лишь слегка худшей, чем линейная.

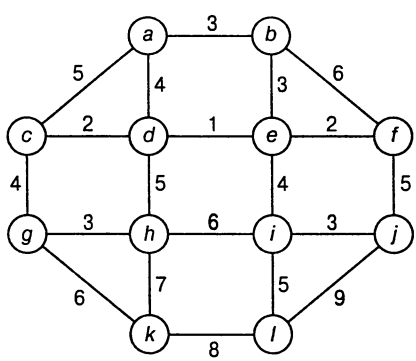
Упражнения 9.2

1. Примените алгоритм Крускала для поиска минимальных остовных деревьев следующих графов.

а)



б)



2. Истинно или ложно каждое из следующих утверждений?
   а) Если e — ребро с минимальным весом в связном взвешенном графе, оно должно быть среди ребер как минимум одного минимального остовного дерева графа.

б) Если e — ребро с минимальным весом в связном взвешенном графе, оно должно быть среди ребер каждого минимального остовного дерева графа.

в) Если веса всех ребер связного взвешенного графа различны, он должен иметь только одно минимальное остовное дерево.

г) Если не все веса ребер связного взвешенного графа различны, он должен иметь больше одного минимального остовного дерева.

3. Какие изменения следует внести в алгоритм $Kruskal$ (если таковые требуются), чтобы он мог находить ***минимальный остовный лес*** произвольного графа? (Минимальный остовный лес — это такой лес, деревья которого представляют собой минимальные остовные деревья связных компонентов графа.)
4. Будут ли корректно работать алгоритмы Крускала и Прима с графом, у которого имеются ребра с отрицательными весами?
5. Разработайте алгоритм для поиска ***максимального остовного дерева*** — т.е. остовного дерева с наибольшим возможным весом ребер — связного взвешенного графа.
6. Перепишите алгоритм Крускала с применением операций над абстрактным типом данных непересекающихся подмножеств.
7. Докажите корректность алгоритма Крускала.
8. Докажите, что временна́я эффективность операции $find(x)$ в версии $union\text{-}by\text{-}size$ быстрого объединения равна $O(\log n)$.
9. Найдите как минимум два Web-узла с анимацией алгоритмов Крускала и Прима. Обсудите достоинства и недостатки этих анимаций.
10. Разработайте и выполните эксперимент по эмпирическому сравнению эффективностей алгоритмов Прима и Крускала на случайных графах с различными размерами и плотностями.

9.3 Алгоритм Дейкстры

В этом разделе мы рассмотрим ***задачу поиска кратчайших путей из одной вершины*** (single-source shortest-paths problem): для данной вершины взвешенного связного графа, называющейся ***исходной*** (source), надо найти кратчайшие пути ко всем остальным вершинам. Важно подчеркнуть, что здесь нас не интересует единый кратчайший путь, начинающийся в исходной вершине и проходящий через все остальные. Это существенно более сложная задача (представляющая собой версию задачи коммивояжера, которая упоминалась в разделе 3.4 и будет рассмотрена позже в этой книге). Задача поиска кратчайших путей из одной

вершины требует найти семейство путей, каждый из которых ведет от исходной вершины к другой вершине графа, хотя некоторые пути, конечно же, могут иметь общие ребра.

Множество практических применений задачи поиска кратчайших путей из одной вершины делает ее очень популярным объектом изучения. Имеется ряд широко известных алгоритмов для ее решения, включая алгоритм Флойда для решения более общей задачи поиска кратчайших путей между всеми вершинами, рассматривавшийся в главе 8. Здесь мы рассмотрим наиболее известный алгоритм решения задачи поиска кратчайших путей из одной вершины, который носит имя ***алгоритм Дейкстры*** (Dijkstra algorithm).[3] Этот алгоритм применим только к графам с неотрицательными весами. Поскольку в большинстве приложений это условие выполняется, такое ограничение не снижает популярность алгоритма Дейкстры.

Алгоритм Дейкстры находит кратчайшие пути к вершинам графа в порядке их удаления от данной исходной вершины. Сначала он находит кратчайший путь от исходной вершины до ближайшей, затем — до второй ближайшей и т.д. В общем случае перед началом i-ой итерации алгоритм определяет кратчайшие пути к $i-1$ другим вершинам, ближайшим к исходной. Эти вершины, исходная вершина и ребра кратчайших путей, ведущих к ним из исходной вершины образуют поддерево T_i данного графа (рис. 9.9). Поскольку веса всех ребер неотрицательны, очередная ближайшая к исходной вершина может быть найдена среди вершин, смежных с T_i. Множество вершин, смежных с вершинами в T_i, можно назвать "пограничными"; именно из них алгоритм Дейкстры выбирает очередную вершину, ближайшую к исходной. (В действительности все прочие вершины можно рассматривать как пограничные, соединенные с вершинами дерева ребрами с бесконечными весами.) Для определения i-ой ближайшей вершины алгоритм вычисляет для каждой пограничной вершины u сумму расстояния до ближайшей вершины дерева v (определяемого весом ребра (u, v)) и длины d_v кратчайшего пути из исходной вершины в вершину v (ранее определенную алгоритмом), а затем выбирает вершину с наименьшей суммой. Главным в алгоритме Дейкстры является то, что достаточно сравнить длины таких специальных путей.

Для упрощения работы алгоритма помечаем каждую вершину двумя метками. Числовая метка d указывает длину кратчайшего пути к данной вершине от исходной, определенную алгоритмом. Вторая метка указывает имя предыдущей вершины на таком пути, т.е. родительский узел строящегося дерева (эта метка остается пустой у исходной вершины и вершин, которые не являются смежными ни с одной

[3] Эдсгер В. Дейкстра (Edsger W. Dijkstra) (1930–2002), знаменитый голландский ученый в области кибернетики, открыл этот алгоритм в средине 1950-х годов. Сам Дейкстра говорил об этом алгоритме: "Это была первая задача, посвященная графам, которую я самостоятельно поставил и решил. Как это ни удивительно, но я не опубликовал ее. В те времена алгоритмы рассматривались сугубо как научные работы."

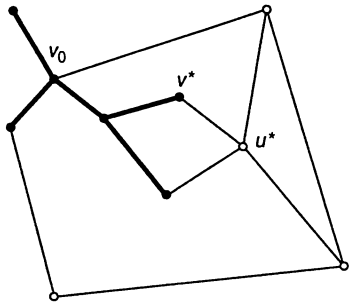


Рис. 9.9. Идея алгоритма Дейкстры. Выделено поддерево найденных кратчайших путей. Очередная вершина, ближайшая к исходной v_0, — вершина u^* выбрана путем сравнения длин путей поддерева, увеличенных на расстояния до вершин, смежных с вершинами поддерева

вершиной текущего дерева). При использовании таких меток поиск следующей ближайшей вершины u^* становится простой задачей поиска пограничной вершины с наименьшим значением d. Неоднозначности разрешаются произвольным образом.

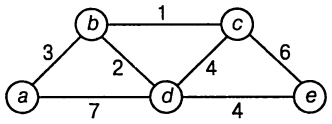
Определив добавляемую в дерево вершину u^*, мы должны выполнить две операции.

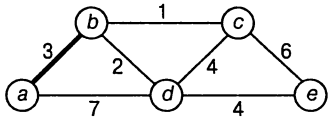
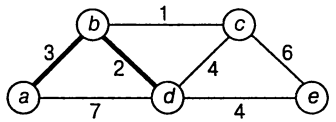
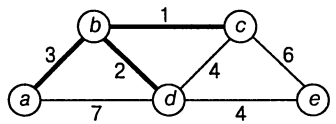
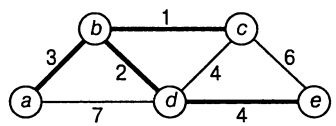
- Переместить вершину u^* из множества пограничных во множество вершин дерева.
- Для каждой остающейся вершины u, связанной с вершиной u^* ребром с весом $w(u^*, u)$ такой, что $d_{u^*} + w(u^*, u) < d_u$, обновить метки u, заменив их на u^* и $d_{u^*} + w(u^*, u)$, соответственно.

На рис. 9.10 продемонстрировано применение алгоритма Дейкстры к конкретному графу.

Использование меток и механика алгоритма Дейкстры весьма схожи с используемыми алгоритмом Прима (см. раздел 9.1). Оба они строят и расширяют поддерево вершин путем выбора очередной вершины из очереди с приоритетами, содержащей остающиеся вне дерева вершины. Однако очень важно не спутать их. Они решают разные задачи и, следовательно, работают с приоритетами, вычисляемыми различными способами: алгоритм Дейкстры сравнивает длины путей и должен суммировать веса ребер, в то время как алгоритм Прима сравнивает веса ребер как они есть.

Теперь мы можем привести псевдокод алгоритма Дейкстры. Он описывает работу алгоритма с использованием явного указания операций над двумя множествами вершин с метками: множества V_T вершин, для которых уже найдены



Вершины дерева	Остальные вершины	Рисунок
a(–,0)	**b(a,3)** c(–,∞) d(a,7) e(–,∞)	
b(a,3)	c(b,3+4) **d(b,3+2)** e(–,∞)	
d(b,5)	**c(b,7)** e(d,5+4)	
c(b,7)	**e(d,9)**	
e(d,9)		

Кратчайшие пути (определяемые буквенными метками от вершины назначения к исходной в левом столбце) и их длины (числовые метки у вершин дерева):

От *a* к *b*:	*a* – *b*	длиной 3
От *a* к *d*:	*a* – *b* – *d*	длиной 5
От *a* к *c*:	*a* – *b* – *c*	длиной 7
От *a* к *e*:	*a* – *b* – *d* – *e*	длиной 9

Рис. 9.10. Применение алгоритма Дейкстры. Очередные ближайшие вершины выделены полужирным шрифтом

кратчайшие пути, и очереди с приоритетами Q, содержащей пограничные вершины. (Заметим, что в приведенном псевдокоде V_T содержит заданную исходную вершину, а очередь — смежные с ней вершины *после* выполнения итерации 0.)

Алгоритм $Dijkstra\,(G, s)$

// Алгоритм Дейкстры для поиска кратчайших путей из одной
// вершины
// **Входные данные:** Взвешенный связный граф $G = \langle V, E \rangle$ и его
// вершина s
// **Выходные данные:** Длина d_v кратчайшего пути от s к v

```
//                          и предпоследняя вершина p_v на этом пути
//                          для всех вершин v ∈ V
Initialize(Q)               // Инициализация пустой очереди
for (Для) каждой вершины v ∈ V do
  d_v ← ∞; p_v ← NULL
  Insert(Q, v, d_v)         // Инициализация приоритета вершины
                            // в очереди с приоритетами
d_s ← 0;
Decrease(Q, s, d_s)         // Обновление приоритета s значением d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
  u* ← DeleteMin(Q)         // Удаление элемента с минимальным
                            // приоритетом
  V_T ← V_T ∪ {u*}
  for (Для) каждой вершины u из V − V_T, смежной с u* do
    if d_u* + w(u*, u) < d_u
      d_u ← d_u* + w(u*, u); p_u ← u*
      Decrease(Q, u, d_u)
```

Временна́я эффективность алгоритма Дейкстры зависит от структур данных, используемых для реализации очереди с приоритетами и представления входного графа. По причинам, пояснявшимся при анализе алгоритма Прима в разделе 9.1, она равна $\Theta(|V|^2)$ для графов, представленных их весовыми матрицами и очередью с приоритетами, реализованной в виде неупорядоченного массива. Для графов, представленных связанными списками смежности и очередью с приоритетами, реализованной как неубывающая пирамида, эффективность равна $O(|E| \log |V|)$. Еще лучшую верхнюю границу можно получить как для алгоритма Прима, так и для алгоритма Дейкстры, если реализовать очередь с приоритетами с использованием такой сложной структуры данных, как ***пирамида Фибоначчи*** (Fibonacci heap) (см., например, [121]). Однако ее сложность и значительные накладные расходы делают такое усовершенствование представляющим в первую очередь теоретический интерес.

Упражнения 9.3

1. Поясните, какие изменения (если таковые нужны) требуется внести в алгоритм Дейкстры и/или в граф для решения следующих задач.
   а) Решение задачи поиска кратчайших путей из одной вершины для ориентированных взвешенных графов.
   б) Поиск кратчайшего пути между двумя заданными вершинами взвешенного ориентированного или неориентированного графа (***задача***

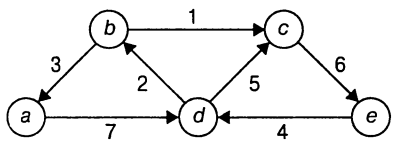
поиска кратчайшего пути между парой вершин (single-pair shortest-path problem)).

в) Поиск кратчайшего пути к данной вершине из всех прочих вершин взвешенного ориентированного или неориентированного графа (***задача поиска кратчайших путей в одну вершину*** (single-destination shortest-paths problem)).

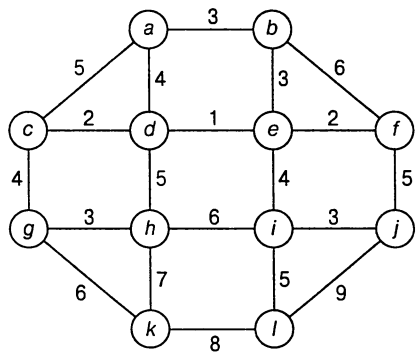
г) Решение задачи поиска кратчайших путей из одной вершины для графа с неотрицательными числами, присвоенными его вершинам (длина пути определяется как сумма чисел в вершинах, составляющих путь).

2. Решите следующие экземпляры задачи поиска кратчайших путей из одной вершины для вершины a в качестве исходной.

а)



б)



3. Приведите контрпример, который показывает, что алгоритм Дейкстры не может корректно работать со взвешенными связными графами с отрицательными весами.

4. Пусть T — дерево, построенное алгоритмом Дейкстры в процессе решения задачи поиска кратчайших путей из одной вершины для взвешенного связного графа G. Истинно или ложно каждое из следующих утверждений?

а) T — остовное дерево G.

б) T — минимальное остовное дерево G.

5. Напишите псевдокод более простой версии алгоритма Дейкстры, которая находит только расстояния (т.е. длины кратчайших путей, но не

сами эти пути) от данной вершины ко всем остальным вершинам графа, представленного весовой матрицей.

6. Докажите корректность алгоритма Дейкстры для графов с положительными весами.
7. Разработайте алгоритм с линейным временем работы для решения задачи поиска кратчайших путей из одной вершины для ориентированных ациклических графов, представленных связанными списками смежности.
8. Разработайте эффективный алгоритм для поиска длины самого длинного пути в ориентированном ациклическом графе. (Эта задача важна в связи с тем, что она определяет нижнюю границу общего времени, необходимого для завершения проекта, состоящего из заданий с ограничениями по предшествованию.)

9. Предположим, что у нас есть модель взвешенного связного графа, сделанная из шаров (представляющих вершины), соединенных веревками соответствующих длин (представляющих ребра).
   а) Опишите, как можно решить задачу поиска кратчайших путей между парой вершин при помощи такой модели.
   б) Опишите, как можно решить задачу поиска кратчайших путей из одной вершины при помощи такой модели.
10. Вернемся к упражнению 1.3.6 об определении наилучшего маршрута для пассажира метрополитена, который бы позволял ему перемещаться от одной станции до другой по развитой системе подземных коммуникаций, наподобие вашингтонской или лондонской. Напишите программу для этой задачи.

9.4 Деревья Хаффмана

Предположим, требуется закодировать текст, состоящий из символов некоторого n-символьного алфавита, назначая каждому из символов текста некоторую последовательность битов, именуемую ***кодом*** (codeword). Например, мы можем использовать ***кодирование фиксированной длины*** (fixed-length encoding), когда каждому символу назначается битовая строка одной и той же длины m ($m \geqslant \log_2 n$). Это именно тот способ, который используется в стандартном семибитовом кодировании ASCII. Один из способов получить схему кодирования, которая в среднем дает более короткую битовую строку, основан на старой идее назначать более короткие коды чаще встречающимся символам, а более длинные — тем, что встречаются реже. (Эта идея, в частности, использована в телеграфных кодах, разработанных в средине XIX века Сэмюэлем Морзе (Samuel Morse). В этом

коде часто встречающимся буквам, таким как e ($\cdot$) или a ($\cdot$ –), назначены короткие последовательности точек и тире, в то время как редкие буквы, такие как q (– – $\cdot$ –) и z (– – $\cdot$ $\cdot$), имеют длинные последовательности.)

Использование ***кодирования переменной длины*** (variable-length encoding), при котором различным символам назначаются коды разной длины, создает проблемы, которых нет при кодировании постоянной длины, а именно: как определить, сколько битов кодированного текста представляют первый (или, в общем случае, i-ый) символ? Для того чтобы избежать этой сложности, мы можем ограничиться ***префиксными кодами*** (prefix code). При использовании префиксного кода ни один код не является префиксом другого кода. Следовательно, при таком кодировании мы просто сканируем битовую строку, пока не получим первую группу битов, являющуюся кодом некоторого символа, заменяем эти биты соответствующим символом и повторяем операцию, пока не будет достигнут конец битовой строки.

Если мы хотим создать бинарный префиксный код для некоторого алфавита, естественным представляется связать его символы с листьями бинарного дерева, в котором все левые ребра помечены 0, а правые — 1 (или наоборот). Код символа можно получить путем записи меток ребер на простом пути от корня дерева к листу символа. Поскольку простого пути к листу, который бы продолжался к другому листу, не существует, нет и кода, который бы был префиксом другого кода; таким образом, все такие деревья приводят к префиксному кодированию.

Как среди множества деревьев, которые могут быть построены таким образом для алфавита с известной частотой употребления символов, найти дерево, которое назначает короткие строки часто встречающимся символам и длинные — редко встречающимся? Это можно сделать при помощи следующего жадного алгоритма, открытого Дэвидом Хаффманом (David Huffman) во время его учебы в Массачуссетском технологическом институте [56].

Алгоритм Хаффмана

Шаг 1. Инициализируем n одноузловых деревьев и помечаем их символами алфавита. Записываем частоту каждого символа в корне его дерева в качестве ***веса*** дерева. (В общем случае вес дерева равен сумме частот, указанных в листьях дерева.)

Шаг 2. Повторяем следующую операцию до тех пор, пока не получим единое дерево. Находим два дерева с наименьшими весами (неоднозначности могут быть разрешены произвольным образом, однако см. упражнение 9.4.2) и делаем их левым и правым поддеревьями нового дерева, в корне которого записываем сумму их весов в качестве веса образованного дерева.

Дерево, построенное по такому алгоритму, называется ***деревом Хаффмана*** (Huffman tree), а код, который оно определяет, — ***кодом Хаффмана*** (Huffman code).

Пример 1. Рассмотрим пятисимвольный алфавит $\{A, B, C, D, \_\}$ со следующими вероятностями символов:

Символ	A	B	C	D	_
Вероятность	0.35	0.1	0.2	0.2	0.15

Построение дерева Хаффмана для приведенных входных данных показано на рис. 9.11. В результате мы получаем следующие коды символов:

Символ	A	B	C	D	_
Вероятность	0.35	0.1	0.2	0.2	0.15
Код	11	100	00	01	101

Следовательно, DAD кодируется как 011101, а 10011011011101 декодируется как BAD_AD.

Для данных вероятностей символов и полученных кодов математическое ожидание количества битов для кодирования одного символа составляет

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

Если бы мы применили код фиксированной длины для того же алфавита, нам бы пришлось использовать как минимум три бита для каждого символа. Таким образом, в этом примере применение кода Хаффмана позволило достичь ***степени сжатия*** (compression ratio), стандартной меры эффективности алгоритма сжатия информации, равной $(3 - 2.25)/3 \cdot 100\% = 25\%$. Другими словами, мы должны ожидать, что кодирование Хаффмана, примененное к некоторому тексту на основе данного алфавита, использует на 25% меньше памяти, чем кодирование фиксированной длины (большое количество экспериментов, выполненных с кодами Хаффмана, показывает, что степень сжатия в данной схеме обычно оказывается в диапазоне от 20% до 80%, в зависимости от характеристик сжимаемого файла). ■

Кодирование Хаффмана — один из наиболее важных методов сжатия файлов. Кроме простоты и универсальности он обладает тем достоинством, что приводит к оптимальному (т.е. минимальной длины) кодированию (при условии, что вероятности появления символов независимы и известны заранее). Простейшая версия сжатия Хаффмана в действительности использует предварительное сканирование текста для подсчета частот появления в нем различных символов. Затем на основании этих частот строится дерево Хаффмана, и с его помощью выполняется кодирование текста. Однако такая схема делает необходимым включение в закодированный текст информации о дереве кодирования для того, чтобы текст было можно декодировать. Преодолеть этот недостаток можно с помощью так называемого ***динамического кодирования Хаффмана*** (dynamic Huffman encoding), при

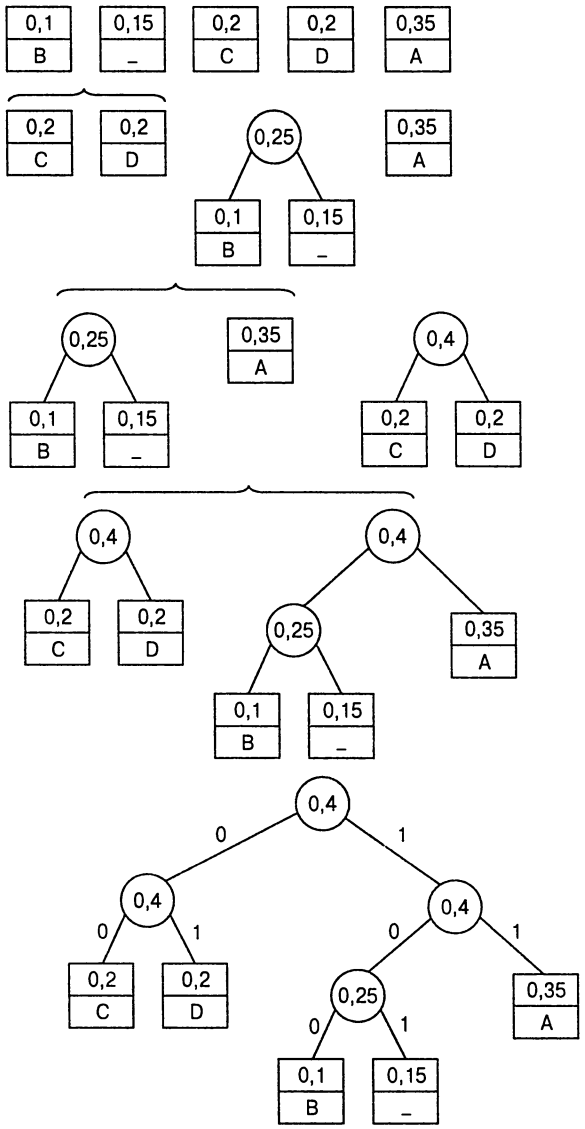


Рис. 9.11. Пример построения дерева кодирования Хаффмана

котором дерево кодирования обновляется всякий раз, когда из исходного текста считывается очередной символ (см., например, [101]).

Важно отметить, что алгоритм Хаффмана не ограничивается сжатием данных. Предположим, у нас есть n положительных чисел $w_1, w_2, \ldots, w_n$, которые назначены n листьям бинарного дерева, по одному на узел. Если мы определим ***взвешенную длину пути*** как сумму $\sum_{i=1}^{n} l_i w_i$, где l_i — длина простого пути от

корня к i-му листу, то как можно построить бинарное дерево с минимальной взвешенной длиной пути? Это — более общая задача, решаемая алгоритмом Хаффмана (в рассматриваемом случае кодирования l_i и w_i представляют собой, соответственно, длину кода и частоту появления i-го символа). Эта задача возникает во многих ситуациях, включающих принятие решения. Рассмотрим, например, игру с угадыванием выбранного предмета из n возможных (например, целого числа от 1 до n) с помощью вопросов, ответы на которые должны быть "да" или "нет". Различные стратегии, применяемые во время игры, могут быть смоделированы при помощи ***деревьев принятия решений*** (decision trees)[4]. Пример такого дерева для $n = 4$ приведен на рис. 9.12.

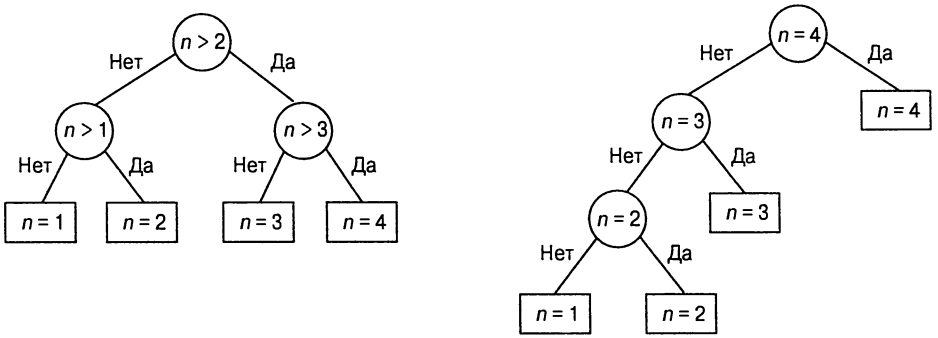


Рис. 9.12. Два дерева принятия решения для угадывания целого числа от 1 до 4

Длина простого пути от корня к листу в таком дереве равна количеству вопросов, требующихся для выбора числа, представленного в этом листе. Если число i выбирается с вероятностью p_i, то сумма $\sum_{i=1}^{n} l_i p_i$ (где l_i — длина пути от корня до i-го листа) определяет среднее количество вопросов, необходимых для "угадывания" выбранного числа при использовании стратегии игры, представленной этим деревом принятия решения. Если каждое из чисел выбирается с одной и той же вероятностью, равной $1/n$, то наилучшая стратегия состоит в последовательном исключении половины (или почти половины) кандидатов, как это делается при бинарном поиске. Однако в случае произвольных p_i это может быть неподходящей стратегией (например, при $n = 4$, $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$ и $p_4 = 0.4$ дерево с минимальной взвешенной длиной пути выглядит так, как показано на рис. 9.12 справа). Таким образом, для решения этой задачи в общем виде мы должны воспользоваться алгоритмом Хаффмана.

В заключение стоит вспомнить, что это второй случай, когда мы сталкиваемся с задачей построения оптимального бинарного дерева. В разделе 8.3 мы рассматривали задачу построения оптимального бинарного дерева поиска с положительными числами (вероятностями поиска), назначенными каждому узлу де-

[4]Более подробно деревья принятия решений рассматриваются в разделе 10.2.

рева. В этом разделе заданные числа назначаются только листьям. Такая задача оказывается более легкой: ее можно решить при помощи жадного алгоритма, в то время как первая задача решается более сложным алгоритмом динамического программирования.

Упражнения 9.4

1. **а)** Постройте код Хаффмана для следующих данных:

Символ	A	B	C	D	_
Вероятность	0.4	0.1	0.2	0.15	0.15

   б) Закодируйте при помощи полученного кода текст $ABACABAD$.

   в) Декодируйте текст, кодирование которого с применением кода из части *а)* упражнения дает 100010111001010.

2. При передаче данных зачастую желательно использовать кодирование с минимальной дисперсией длин кодов (среди кодов с одинаковой средней длиной кода). Вычислите среднее значение длины кода и ее дисперсию для двух разных кодов Хаффмана, получающихся при различном разрешении неоднозначностей в процессе построения дерева Хаффмана для следующих данных:

Символ	A	B	C	D	E
Вероятность	0.1	0.1	0.2	0.2	0.4

3. Скажите, справедливы ли следующие утверждения для любого кода Хаффмана.

   а) Коды двух наименее часто встречающихся символов имеют одинаковую длину.

   б) Длина кода более часто встречающегося символа всегда меньше или равна длине кода менее часто встречающегося символа.

4. Какова максимально возможная длина кода при кодировании Хаффмана алфавита из n символов?

5. **а)** Напишите псевдокод алгоритма построения дерева Хаффмана.

   б) Какому классу эффективности как функции размера алфавита n принадлежит алгоритм построения дерева Хаффмана?

6. Покажите, что дерево Хаффмана можно построить за линейное время, если символы алфавита даны в отсортированном порядке их частот.

7. Какой алгоритм вы используете для получения кодов всех символов для заданного дерева Хаффмана? Какому классу эффективности как функции размера алфавита n принадлежит этот алгоритм?
8. Поясните, как можно генерировать код Хаффмана без явной генерации дерева кодирования Хаффмана.
9. **а)** Напишите программу, которая строит код Хаффмана для заданного английского (русского, украинского или иного) текста и кодирует его.
   б) Напишите программу для декодирования текста, закодированного с использованием кодов Хаффмана.
   в) Поэкспериментируйте с вашей программой и определите типичный диапазон степени сжатия при помощи кодирования Хаффмана обычного текста из примерно 1000 слов.
   г) Поэкспериментируйте с вашей программой и определите, насколько изменяется степень сжатия при переходе от использования действительных частот символов в тексте к их стандартным оценкам.

10. Разработайте стратегию для минимизации математического ожидания количества вопросов в следующей игре, взятой в [39]. У вас имеется колода карт, состоящая из одной единицы, двух двоек, трех троек и т.д. до девяти девяток, всего 45 карт. Некто выбирает карту из перетасованной колоды. Вы должны определить ее, задавая вопросы, на которые можно отвечать только "да" или "нет".

Резюме

- *Жадный метод* состоит в построении решения задачи оптимизации путем последовательности шагов, каждый из которых расширяет частично построенное решение до тех пор, пока не будет достигнуто полное решение исходной задачи. На каждом шаге должен делаться выбор, являющийся *допустимым*, *локально оптимальным* и *окончательным*.
- *Алгоритм Прима* представляет собой жадный алгоритм для построения минимального остовного дерева взвешенного связного графа. Он работает путем присоединения к ранее построенному поддереву вершины, ближайшей к вершинам, уже находящимся в дереве.
- *Алгоритм Крускала* представляет собой еще один жадный алгоритм для построения минимального остовного дерева взвешенного связного графа. Он строит минимальное остовное дерево путем выбора ребер в возрастающем порядке их весов так, чтобы при этом не образовыва-

лись циклы. Эффективная проверка этого условия требует применения одного из так называемых *алгоритмов объединения и поиска*.

- *Алгоритм Дейкстры* решает задачу поиска кратчайших путей из одной (исходной) вершины ко всем другим вершинам взвешенного ориентированного или неориентированного графа. Он работает подобно алгоритму Прима, но сравнивает не длины ребер, а длины путей. Алгоритм Дейкстры всегда дает корректное решение для графа с неотрицательными весами.
- *Дерево Хаффмана* представляет собой бинарное дерево, которое минимизирует взвешенную длину пути от корня к листьям, содержащим множество предопределенных весов. Наиболее важным приложением деревьев Хаффмана являются коды Хаффмана.
- *Код Хаффмана* представляет собой оптимальную префиксную схему кодирования переменной длины, которая назначает символам битовые строки на основе частот появления этих символов в данном тексте. Это достигается путем жадного построения бинарного дерева, листья которого представляют символы алфавита, а ребра помечены нулями и единицами.

Глава 10

Ограничения мощи алгоритмов

Разум различает возможное и невозможное; рассудок различает осмысленное и бессмысленное. Однако возможное может оказаться бессмысленным.

— Макс Борн (Max Born) (1882–1970)
"Моя жизнь и взгляды" (My Life and My Views), 1968

В предыдущих главах книги мы встретились с десятками алгоритмов для решения множества задач. Избежать определения алгоритмов как инструментов для решения задач невозможно: это очень мощные инструменты, в особенности при работе на современных компьютерах. Но мощь алгоритмов не безгранична, и именно эти границы являются предметом рассмотрения в данной главе. Как мы увидим, некоторые задачи не могут быть решены ни одним алгоритмом. Некоторые могут быть решены, но не за полиномиальное время работы. Но даже если задача может быть решена некоторыми алгоритмами за полиномиальное время, все равно обычно имеются нижние границы эффективности алгоритмов.

В разделе 10.1 мы рассмотрим методы получения нижних границ, т.е. оценки минимально необходимого количества работы для решения задачи. В общем случае получить нетривиальную нижнюю границу, даже для задачи с простой формулировкой, обычно очень непросто. В отличие от определения эффективности конкретного алгоритма, в этом случае нас интересует граница эффективности *любого* алгоритма — известного или неизвестного. Это также требует очень точного и аккуратного описания операций, которые может выполнять такой алгоритм. Если мы ошибемся в точном описании "правил игры", то место нашим выводам может оказаться на свалке, как это случилось, например, с известным британским физиком лордом Кельвином, который в 1895 году заявил, что летающие машины тяжелее воздуха невозможны...

В разделе 10.2 мы познакомимся с деревьями принятия решения. Этот метод позволяет кроме прочего, установить нижнюю границу эффективности алгоритмов для сортировки и поиска в отсортированных массивах, основанных на операции сравнения. В результате мы узнаем, можно ли разработать более быстрый алгоритм сортировки, чем сортировка слиянием, и является ли бинарный поиск

самым быстрым алгоритмом поиска в отсортированном массиве (что вам подсказывает ваша интуиция по поводу этих вопросов?). Кстати, деревья принятия решения оказываются отличными помощниками при решении некоторых головоломок, например задачи о поиске фальшивой монеты (см. раздел 5.5).

В разделе 10.3 рассматривается вопрос сложности: какие задачи можно, а какие нельзя решить за полиномиальное время. Эта область теоретической информатики называется "теорией вычислительной сложности". Здесь мы познакомимся только с основными элементами этой теории и неформально рассмотрим такие фундаментальные понятия, как P, NP и NP-полнота, включая самую важную нерешенную задачу теоретической информатики о соотношении задач классов P и NP.

Последний раздел этой главы посвящен численному анализу. Эта область информатики посвящена алгоритмам для решения задач "непрерывной" математики — решения уравнений и систем уравнений, вычисление таких функций, как $\sin x$ и $\ln x$, вычисление интегралов и т.д. Природа этих задач накладывает два типа ограничений. Во-первых, большинство из них не может быть решено точно. Во-вторых, даже приближенное их решение приводит к работе с числами, которые могут быть представлены в цифровых компьютерах только с ограниченным уровнем точности. Работа с приближенными числами без должной аккуратности может привести к очень неточным результатам. Мы увидим, что даже решение обычного квадратного уравнения на компьютере может сопровождаться значительными трудностями и потребовать модификации канонической формулы для корней квадратного уравнения.

10.1 Доказательства нижних границ

Рассматривать эффективность алгоритмов можно двумя путями. Можно установить класс асимптотической эффективности (скажем, для наихудшего случая) и посмотреть, где он находится в иерархии классов эффективности (см. раздел 2.2). Например, сортировка выбором, эффективность которой квадратична, является достаточно быстрым алгоритмом, в то время как алгоритм решения задачи о ханойских башнях работает очень медленно в силу своей экспоненциальности. Однако можно возразить, что такое сравнение алгоритмов сродни сравнению яблок с вишнями, поскольку эти алгоритмы предназначены для решения совершенно разных задач. Альтернативный, более "честный" подход состоит в ответе на вопрос о том, как соотносится эффективность конкретного алгоритма с другими алгоритмами для решения той же задачи. В этом случае сортировка выбором оказывается медленным алгоритмом, поскольку имеются алгоритмы сортировки со временем работы $O(n \log n)$; алгоритм решения задачи о ханойской башне,

с другой стороны, оказывается наиболее быстрым из возможных алгоритмов для решения поставленной задачи.

Когда мы хотим выяснить, как соотносится эффективность данного алгоритма с эффективностями других алгоритмов для решения той же задачи, желательно знать, какую наивысшую эффективность может иметь *любой* алгоритм, решающий рассматриваемую задачу. Знание такой ***нижней границы*** (lower bound) может подсказать, на что мы можем надеяться при попытках получить более эффективный алгоритм для решения нашей задачи. Если такая граница ***плотная*** (tight), т.е. уже имеется алгоритм, принадлежащий тому же классу эффективности, что и нижняя граница, мы можем в лучшем случае получить только улучшение эффективности на постоянный множитель. Если же между эффективностью наиболее быстрого алгоритма и наилучшей нижней границей имеется "зазор", то дверь для дальнейшего возможного усовершенствования алгоритма остается не запертой: либо может существовать более быстрый алгоритм, соответствующий нижней границе, либо можно доказать существование лучшей нижней границы.

В этом разделе представлено несколько методов для определения нижних границ и проиллюстрировано их применение для конкретных примеров. Как и в случае анализа эффективности конкретных алгоритмов в предыдущих разделах, следует различать класс нижней границы и минимально необходимое количество определенных операций. Как правило, определение второй величины — задача существенно более сложная, чем первая. Например, мы можем сделать немедленный вывод о том, что любой алгоритм для поиска медианы n чисел должен иметь эффективность $\Omega(n)$ (почему?), но не так просто доказать, что любой алгоритм для решения этой задачи, основанный на сравнениях, должен выполнить как минимум $3(n-1)/2$ сравнений в наихудшем случае (при нечетном n).

Тривиальные нижние границы

Простейший метод получения класса нижней границы основан на подсчете количества элементов входных данных, которые следует обработать. Поскольку любой алгоритм должен как минимум "прочесть" все данные, с которыми он будет работать, и "записать" все выходные данные, такой подсчет приводит к ***тривиальной нижней границе*** (trivial lower bound). Например, любой алгоритм для генерации всех перестановок n различных элементов должен принадлежать $\Omega(n!)$, поскольку размер выходных данных равен $n!$. Эта граница — плотная, так как хорошие алгоритмы для генерации перестановок затрачивают постоянное время для поиска каждой из них, за исключением первой (см. раздел 5.4).

В качестве другого примера рассмотрим вычисление полинома степени n

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

в конкретной точке x для заданных коэффициентов $a_n, a_{n-1}, \ldots, a_0$. Легко видеть, что все коэффициенты должны быть обработаны любым алгоритмом вычисления полинома. Если бы это было не так, то мы бы могли изменить этот необрабатываемый коэффициент, что должно привести к изменению значения полинома в ненулевой точке x. Таким образом, любой алгоритм вычисления полинома должен иметь эффективность $\Omega(n)$. Эта нижняя граница — плотная, поскольку имеется алгоритм вычисления справа налево (упражнение 6.5.2) и схема Горнера (раздел 6.5) — оба с линейным временем работы.

Аналогичным способом тривиальная граница для вычисления произведения двух матриц размером $n \times n$ оказывается равной $\Omega(n^2)$, так как любой такой алгоритм должен обработать $2n^2$ элементов входных матриц и сгенерировать n^2 элементов произведения. Однако неизвестно, является ли эта граница плотной.

Тривиальные нижние границы зачастую слишком малы, чтобы быть полезными. Например, тривиальная граница для задачи коммивояжера равна $\Omega(n^2)$. поскольку на вход подается $n(n-1)/2$ расстояний между городами, а на выходе получается список из $n+1$ городов, образующих оптимальный маршрут. Однако эта граница бесполезна, поскольку неизвестен алгоритм решения данной задачи, у которого время работы выражалось бы полиномиальной функцией любой степени.

Имеется еще одно препятствие для вывода значимой нижней границы этим методом. Оно связано с определением того, какая часть входных данных должна быть обработана любым алгоритмом решения рассматриваемой задачи. Например, поиск элемента с данным значением в отсортированном массиве не требует обработки всех его элементов (почему?). В качестве еще одного примера рассмотрим задачу определения связности неориентированного графа, заданного его матрицей смежности. Естественно ожидать, что любой такой алгоритм должен проверить существование каждого из $n(n-1)/2$ потенциальных ребер, но доказательство этого факта нетривиально.

Информационно-теоретические доказательства

В то время как описанный выше подход принимает во внимание размер выхода задачи, информационно-теоретический подход пытается установить нижнюю границу алгоритма на основе количества информации, которую он производит. Рассмотрим, например, хорошо известную игру с отгадыванием натурального числа от 1 до n при помощи вопросов, на которые можно давать ответ "да" или "нет". Количество неопределенностей, которые должен разрешить любой алгоритм для решения этой задачи, можно оценить как $\lceil \log_2 n \rceil$ — количество битов, необходимых для указания конкретного числа из n возможных. Каждый вопрос (вернее, ответы на них) можно рассматривать как получение не более одного бита информации о выходе алгоритма (загаданном числе). Следовательно, любой такой

алгоритм должен выполнить как минимум $\lceil \log_2 n \rceil$ шагов, перед тем как сможет определить загаданное число в наихудшем случае.

Подход, которым мы только что воспользовались, называется ***информационно-теоретическим доказательством*** (information-theoretic argument) из-за его связи с теорией информации. Как доказано, он очень полезен для поиска так называемых ***информационно-теоретических нижних границ*** (information-theoretic lower bounds) для многих задач с использованием сравнений, включая сортировку и поиск. Идея, лежащая в основе этого подхода, может быть гораздо более точно осуществлена посредством механизма ***деревьев принятия решения*** (decision trees). Из-за важности этого метода он будет рассмотрен отдельно, в разделе 10.2.

Доказательство "от противника"

Вернемся еще раз к игре с угадыванием числа, на примере которой мы разбирали информационно-теоретический метод. Можно доказать, что любой алгоритм, который решает эту задачу, должен в наихудшем случае задать как минимум $\lceil \log_2 n \rceil$ вопросов, если мы сыграем роль противника, который хочет, чтобы алгоритм задал максимально возможное число вопросов. Противник начинает с рассмотрения каждого из потенциально выбираемых чисел от 1 до n. (Конечно, это жульничество, если рассматривать игру, а не доказательство нашего утверждения.) После каждого вопроса противник дает ответ, который оставляет для него наибольшее множество чисел, соответствующих ответам на все ранее заданные вопросы. (Такая стратегия оставляет противнику как минимум половину чисел, которые были у него перед последним вопросом.) Если остановить выполнение алгоритма до того, как размер множества окажется равным единице, противник сможет продемонстрировать число, которое является корректным входным данным и которое алгоритм не смог определить. После этого показать, что требуется $\lceil \log_2 n \rceil$ итераций для приведения n-элементного множества к одному элементу путем снижения его размера вдвое и округления в большую сторону — дело техники. Следовательно, в наихудшем случае алгоритму требуется задать как минимум $\lceil \log_2 n \rceil$ вопросов.

Этот пример иллюстрирует ***метод противника*** (adversary method) для выяснения нижних границ. Он основан на логике злого, но честного противника, который делает все, чтобы алгоритм выполнил как можно больше действий, но честность заставляет его согласовывать свои действия с уже сделанным выбором. В таком случае нижняя граница получается путем измерения количества работы, которая требуется для снижения размера потенциального входного множества до одного элемента по самому длинному пути.

В качестве еще одного примера рассмотрим задачу слияния двух отсортированных списков размером n элементов:

$$a_1 < a_2 < \cdots < a_n \text{ и } b_1 < b_2 < \cdots < b_n$$

в один отсортированный список размером $2n$. Для простоты полагаем, что все a и b различны, так что задача имеет единственное решение. Мы уже встречались с этой задачей при рассмотрении сортировки слиянием в разделе 4.1. Вспомним, что мы выполняли слияние путем многократного сравнения первых элементов в остающихся списках и вывода в результирующий список меньшего из них. Количество сравнений, выполняемых этим алгоритмом в наихудшем случае, равно $2n - 1$.

Существует ли алгоритм, который мог бы выполнить слияние быстрее? Ответ на этот вопрос отрицательный. Кнут [67] использовал метод противника для доказательства того, что $2n - 1$ — нижняя граница количества сравнений ключей, которые должен сделать любой алгоритм, основанный на сравнениях, для решения поставленной задачи. Противник пользуется следующим правилом: отвечает "да" на вопрос $a_i < b_j$ тогда и только тогда, когда $i < j$. Это заставляет любой корректный алгоритм слияния выдать единственно возможный список, согласующийся с этим правилом:

$$b_1 < a_1 < b_2 < a_2 < \cdots < b_n < a_n.$$

Чтобы получить такой список, любой корректный алгоритм должен явным образом сравнить $2n - 1$ соседних пар его элементов, т.е. b_1 — с a_1, a_1 — с b_2 и т.д. Если одно из этих сравнений не будет сделано, например, a_1 не будет сравнено с b_2, то мы можем поменять эти ключи местами и получить последовательность

$$b_1 < b_2 < a_1 < a_2 < \cdots < b_n < a_n,$$

согласованную со всеми сделанными сравнениями, но неотличимую от корректной последовательности, приведенной выше. Следовательно, $2n - 1$ в действительности представляет собой нижнюю границу количества сравнений ключей, необходимых для алгоритма слияния.

Приведение задачи

Мы уже встречались с приведением задачи в разделе 6.6. Там мы рассмотрели метод получения алгоритма для решения задачи P путем приведения ее к другой задаче Q, разрешимой при помощи известного алгоритма. Подобная идея приведения может использоваться и для поиска нижней границы. Чтобы показать, что задача P как минимум столь же сложна, как и задача Q с известной нижней границей, мы должны привести Q к P (не P к Q). Другими словами, мы должны показать, что произвольный экземпляр задачи Q может быть преобразован (достаточно эффективным способом) к экземпляру задачи P, так что любой алгоритм, который решает задачу P, будет также решать и задачу Q. Тогда нижняя граница для задачи Q будет нижней границей и для задачи P. В табл. 10.1 перечислены несколько важных задач, которые часто используются для этой цели.

Таблица 10.1

Задача	Нижняя граница	Плотность
Сортировка	$\Omega(n \log n)$	Да
Поиск в отсортированном массиве	$\Omega(\log n)$	Да
Задача единственности элемента	$\Omega(n \log n)$	Да
Умножение n-значных целых чисел	$\Omega(n)$	Неизвестно
Умножение квадратных матриц	$\Omega(n^2)$	Неизвестно

Мы определим нижние границы для сортировки и поиска в следующем разделе. Задача единственности элемента состоит в выяснении того, есть ли среди n заданных чисел дубли (мы уже сталкивались с этой задачей в разделах 2.3 и 6.1). Доказательство нижней границы для этой на вид простой задачи основано на очень сложном математическом анализе, который выходит за рамки нашей книги (см., например, [90]). Что касается двух последних алгебраических задач, то их нижние границы, приведенные в табл. 10.1, тривиальны, но пока неизвестно, могут они быть улучшены или нет.

В качестве примера определения нижней границы методом приведения рассмотрим задачу поиска евклидова минимального остовного дерева: требуется построить дерево минимальной длины, узлами которого являются данные n точек на декартовой плоскости. В качестве задачи с известной нижней границей воспользуемся задачей единственности элементов. Мы можем преобразовать любое множество $x_1, x_2, \ldots, x_n$ из n действительных чисел во множество n точек на декартовой плоскости, просто добавляя к ним 0 в качестве координаты y: $(x_1, 0)$, $(x_2, 0), \ldots, (x_n, 0)$. Пусть T — минимальное остовное дерево, найденное для этого множества точек. Поскольку T должно содержать кратчайшее ребро, проверка, содержит ли T ребро нулевой длины, отвечает на вопрос об единственности данных чисел. Из этого приведения следует, что нижней границей задачи поиска евклидова минимального остовного дерева также является $\Omega(n \log n)$.

Поскольку сложность многих задач неизвестна, метод приведения часто используется для сравнения относительной сложности задач. Например, формулы

$$x \cdot y = \frac{(x+y)^2 - (x-y)^2}{4} \text{ и } x^2 = x \cdot x$$

показывают, что задачи вычисления произведения двух n-разрядных чисел и возведение n-разрядного числа в квадрат принадлежат одному и тому же классу эффективности, несмотря на кажущуюся большую простоту второй задачи по сравнению с первой.

Для операций с матрицами имеется ряд аналогичных результатов. Например, умножение двух симметричных матриц принадлежит тому же классу сложности, что и произведение двух произвольных квадратных матриц. Этот результат осно-

ван на том наблюдении, что не только первая задача является частным случаем второй, но и задача перемножения двух произвольных матриц порядка n, скажем, A и B, может быть приведена к задаче перемножения двух симметричных матриц

$$X = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \text{ и } Y = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix},$$

где A^T и B^T являются транспонированными матрицами A и B (т.е. $A^T_{ij} = A_{ji}$ и $B^T_{ij} = B_{ji}$, а 0 означает нулевую матрицу размером $n \times n$ (все элементы которой равны 0). В самом деле,

$$XY = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix},$$

откуда легко получить интересующее нас произведение матриц AB (да, мы дважды перемножаем матрицы исходного размера, но это всего лишь небольшое техническое усложнение, не влияющее на класс сложности).

Хотя такие результаты и интересны, в разделе 10.3 мы покажем более важное применение метода приведения для сравнения сложности задач.

Упражнения 10.1

1. Докажите, что классический рекурсивный алгоритм решения задачи о ханойской башне (раздел 2.4) делает минимальное количество перемещений дисков, необходимых для решения задачи.
2. Найдите тривиальные классы нижних границ для следующих задач и по возможности укажите, плотная ли эта граница.
   а) Поиск наибольшего элемента массива.
   б) Проверка полноты графа, представленного матрицей смежности.
   в) Генерация всех подмножеств n-элементного множества.
   г) Определение того, все ли n действительных чисел различны.
3. Рассмотрим задачу определения более легкой фальшивой монеты среди n одинаковых по виду монет при помощи рычажных весов. Можем ли мы использовать то же информационно-теоретическое доказательство, что и приведенное в тексте для количества вопросов в игре на отгадывание, чтобы сделать вывод о том, что для определения фальшивой монеты требуется как минимум $\lceil \log_2 n \rceil$ взвешиваний в наихудшем случае?

4. Докажите, что любой алгоритм поиска наибольшего среди n заданных чисел в худшем случае должен выполнить $n-1$ сравнений.

5. Приведите доказательство методом противника того факта, что временна́я эффективность любого алгоритма, проверяющего связность графа с n вершинами, равна $\Omega\left(n^2\right)$, при наличии единственной операции, состоящей в выяснении наличия ребра между двумя вершинами графа. Является ли эта нижняя граница плотной?

6. Чему равно наименьшее число сравнений, необходимых алгоритму сортировки, основанному на операции сравнения, чтобы объединить два отсортированных списка размером n и $n+1$ элементов соответственно? Докажите корректность вашего ответа.

7. Найдите произведение матриц A и B путем преобразования к произведению двух симметричных матриц, если

$$A = \begin{bmatrix} 1 & -1 \\ 2 & 3 \end{bmatrix} \text{ и } B = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix}.$$

8. **а)** Можно ли использовать формулы этого раздела, которые указывают эквивалентность сложности задач умножения и возведения в квадрат двух целых чисел для того, чтобы показать эквивалентность задач умножения и возведения в квадрат двух матриц?

   б) Покажите, что умножение двух матриц порядка n может быть приведено к возведению в квадрат матрицы размером $2n$.

9. Найдите класс плотной нижней границы задачи поиска двух ближайших чисел среди n действительных чисел $x_1, x_2, \ldots, x_n$.

10. Дана плитка шоколада, состоящая из $n \times m$ ячеек. Ее надо разломить на nm кусочков. Вы можете разламывать плитку только по прямым линиям, и только одну плитку за раз. Разработайте алгоритм для решения поставленной задачи при помощи минимального количества разломов.

10.2 Деревья принятия решения

Многие важные алгоритмы, в особенности алгоритмы сортировки и поиска, работают посредством сравнения элементов входных данных. Мы можем изучать производительность таких алгоритмов при помощи устройства под названием ***дерево принятия решения*** (decision tree). В качестве примера на рис. 10.1 представлено дерево принятия решения для поиска минимума из трех чисел. Каждый внутренний узел бинарного дерева принятия решения представляет показанное в узле сравнение ключа, например $k < k'$. Левое поддерево узла содержит информацию о последующих сравнениях, выполняемых, если $k < k'$; правое поддерево

содержит аналогичную информацию для случая $k > k'$ (для простоты в этом разделе мы считаем, что все входные элементы различны). Каждый лист представляет возможный выход алгоритма при обработке некоторых входных данных размера n. Заметим, что количество листьев может оказаться большим, чем количество вариантов выходных данных, поскольку в некоторых алгоритмах один и тот же выход может быть достигнут разными цепочками сравнений (именно такой случай изображен на рис. 10.1). Однако важно отметить, что количество листьев должно быть не меньше возможных выходных данных. Работу алгоритма с конкретными входными данными размером n можно представить как проход по пути по дереву принятия решения от корня до листа; количество сравнений при выполнении алгоритма равно количеству ребер на указанном пути. Следовательно, количество сравнений в наихудшем случае равно высоте дерева принятия решения алгоритма.

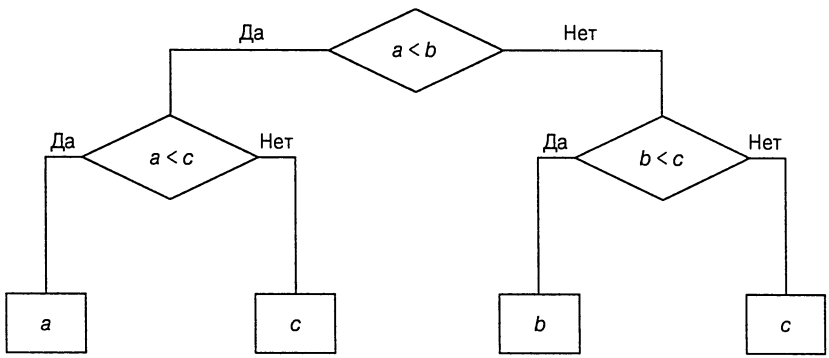


Рис. 10.1. Дерево принятия решения для поиска минимального из трех чисел

Главная идея этого метода заключается в наблюдении, что дерево с данным количеством листьев, которое определяется количеством вариантов выходных данных, должно быть достаточно высоким, чтобы содержать листья в требуемом количестве. В частности, нетрудно доказать, что для любого бинарного дерева с l листьями и высотой h

$$h \geqslant \lceil \log_2 l \rceil . \tag{10.1}$$

В самом деле, бинарное дерево высотой h с наибольшим количеством листьев содержит все листья на последнем уровне (почему?). Следовательно, наибольшее количество листьев в таком дереве равно 2^h. Другими словами, $2^h \geqslant l$, откуда непосредственно следует (10.1).

Неравенство (10.1) определяет нижнюю границу высоты бинарных деревьев принятия решения, а следовательно, количество сравнений, выполняемых в наихудшем случае любым алгоритмом для решения рассматриваемой задачи, основанным на сравнениях. Такая граница называется ***информационно-теоретиче-***

ской нижней границей (см. раздел 10.1). Мы проиллюстрируем этот метод ниже на примере двух важных задач: сортировки и поиска в отсортированном массиве.

Деревья принятия решения для алгоритмов сортировки

Большинство алгоритмов сортировки основано на использовании сравнений, т.е. они работают путем сравнения элементов сортируемого списка. Кроме того, за исключением бинарной сортировки вставкой (см. упражнение 5.1.9), для таких алгоритмов базовой операцией является сравнение двух элементов. Следовательно, изучая свойства деревьев принятия решения для алгоритмов сортировки, основанных на сравнении, мы можем получить важные нижние границы временно́й эффективности таких алгоритмов.

Выход алгоритма сортировки можно рассматривать как поиск перестановки индексов элементов входного списка, которая располагает элементы в возрастающем порядке. Например, для выхода $a < c < b$, полученного при сортировке списка a, b, c (см. рис. 10.2), интересующая нас перестановка — $1, 3, 2$. Следовательно, количество возможных выходов при сортировке произвольного n-элементного списка равно $n!$.

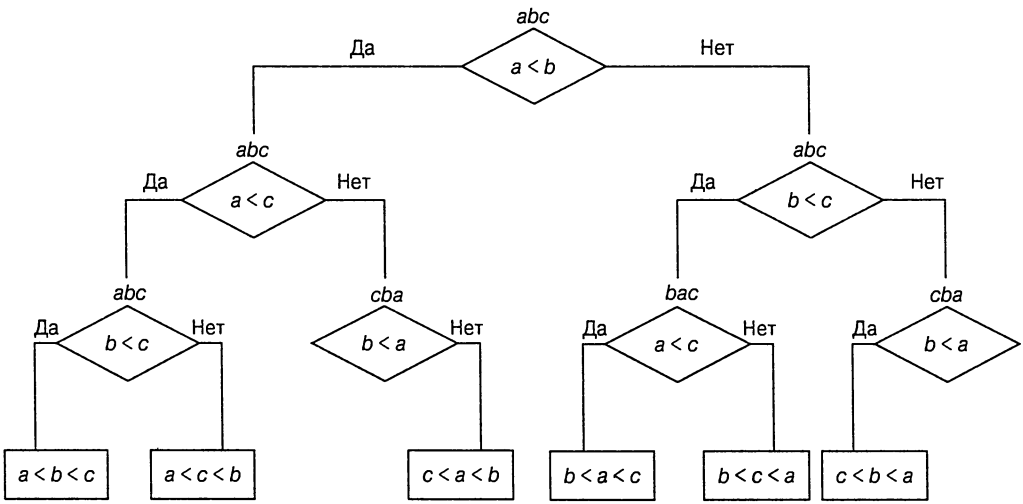


Рис. 10.2. Дерево принятия решения для сортировки выбором трехэлементного списка. Тройка над каждым узлом дерева указывает состояние сортируемого массива. Обратите внимание на два избыточных сравнения $b < a$ с единственным возможным выходом, что связано с выполненными ранее сравнениями

Из неравенства (10.1) следует, что высота бинарного дерева принятия решения для произвольного алгоритма сортировки на основе сравнений (а следовательно, и количество сравнений в наихудшем случае, выполняемое таким алгоритмом) не

может быть меньше $\lceil \log_2 n! \rceil$:

$$C_{worst}(n) \geqslant \lceil \log_2 n! \rceil . \tag{10.2}$$

Используя формулу Стирлинга для $n!$, получим

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}\,(n/e)^n = n\log_2 n - n\log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n\log_2 n.$$

Другими словами, необходимо примерно $n\log_2 n$ сравнений для сортировки произвольного n-элементного списка любым алгоритмом сортировки, который основан на использовании сравнений. Заметим, что сортировка слиянием выполняет примерно такое количество сравнений в худшем случае, а следовательно, является асимптотически оптимальной. Отсюда также следует плотность асимптотической нижней границы $n\log_2 n$, а значит, она не может быть существенно улучшена. Мы должны отметить, однако, что нижняя граница $\lceil \log_2 n! \rceil$ может быть улучшена для некоторых значений n. Например, $\lceil \log_2 12! \rceil = 29$, но доказано, что для сортировки массива из 12 элементов в худшем случае необходимо (и достаточно) 30 сравнений.

Мы можем также использовать деревья принятия решения и для анализа поведения алгоритма сортировки, использующего сравнения, в среднем случае. Можно вычислить среднее количество сравнений для конкретного алгоритма как среднюю глубину листьев его дерева принятия решения, т.е. среднюю длину пути от корня до листьев. Например, для сортировки трех элементов вставкой, дерево принятия решения которой показано на рис. 10.3, это количество равно $(2+3+3+2+3+3)/6 = 2\frac{2}{3}$.

Стандартное предположение, что все $n!$ выходов алгоритма сортировки равновероятны, приводит нас к следующей нижней границе среднего количества сравнений, выполняемых любым алгоритмом сортировки с использованием сравнений n-элементного списка:

$$C_{avg}(n) \geqslant \log_2 n!. \tag{10.3}$$

Как мы видели ранее, эта нижняя граница равна примерно $n\log_2 n$. Вы можете удивиться, что нижние границы в наихудшем и среднем случае практически идентичны. Вспомним, однако, что эти границы получены путем максимизации количества сравнений, выполняемых соответственно в наихудшем и в среднем случаях. Для отдельных алгоритмов сортировки эффективность в среднем случае, конечно, может значительно превышать эффективность в наихудшем случае.

Деревья принятия решения для поиска в отсортированном массиве

В этом разделе мы покажем, как деревья принятия решения могут использоваться для определения нижних границ количества сравнений ключей при поиске

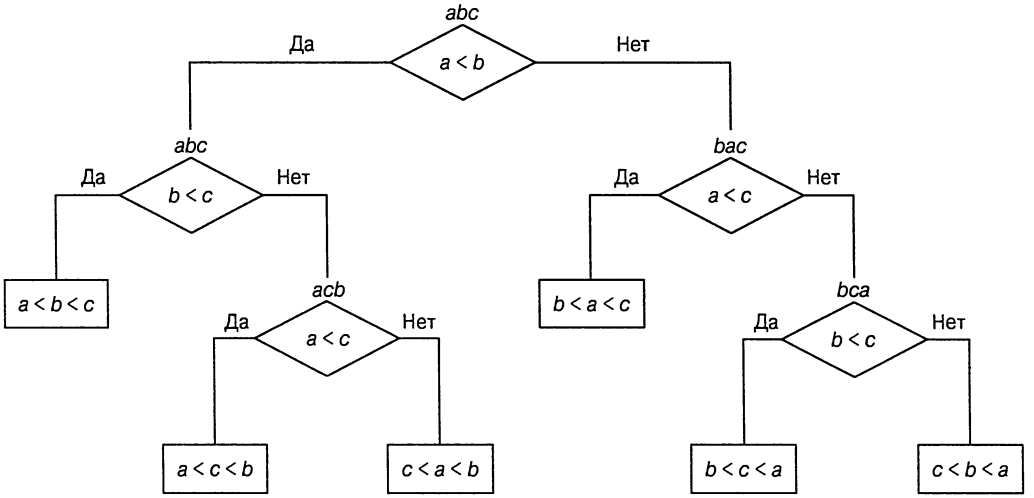


Рис. 10.3. Дерево принятия решения для сортировки трех элементов вставкой

в отсортированном массиве из n ключей $A[0] < A[1] < \cdots < A[n-1]$. Основным алгоритмом для решения этой задачи является бинарный поиск. Как мы видели в разделе 4.3, количество сравнений, выполняемых бинарным поиском в наихудшем случае, $C^{bc}_{worst}(n)$, определяется формулой

$$C^{bc}_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil . \qquad (10.4)$$

Воспользуемся деревьями принятия решения для того, чтобы выяснить, является ли это количество сравнений наименьшим возможным.

Поскольку здесь мы имеем дело с тернарным сравнением, при котором сравнение искомого ключа K с некоторым элементом $A[i]$ дает один из ответов — $K < A[i]$, $K = A[i]$ или $K > A[i]$, естественным представляется применение тернарных деревьев принятия решения. На рис. 10.4 показано такое дерево для случая $n = 4$. Внутренние узлы этого дерева указывают элементы массива, сравниваемые с искомым ключом. Листья указывают либо найденный элемент при успешном поиске, либо найденный интервал, которому принадлежит искомый ключ, при неудачном поиске.

Любой алгоритм для поиска в отсортированном массиве с использованием тернарного сравнения можно представить при помощи тернарного дерева принятия решения, подобного приведенному на рис. 10.4. Для массива из n элементов все такие деревья будут иметь $2n + 1$ листьев (n для успешных поисков и $n + 1$ для неудачных). Поскольку минимальная высота h тернарного дерева с l листьями равна $\lceil \log_3 l \rceil$, получаем следующую нижнюю границу количества сравнений в наихудшем случае:

$$C_{worst}(n) \geqslant \lceil \log_3 (2n+1) \rceil .$$

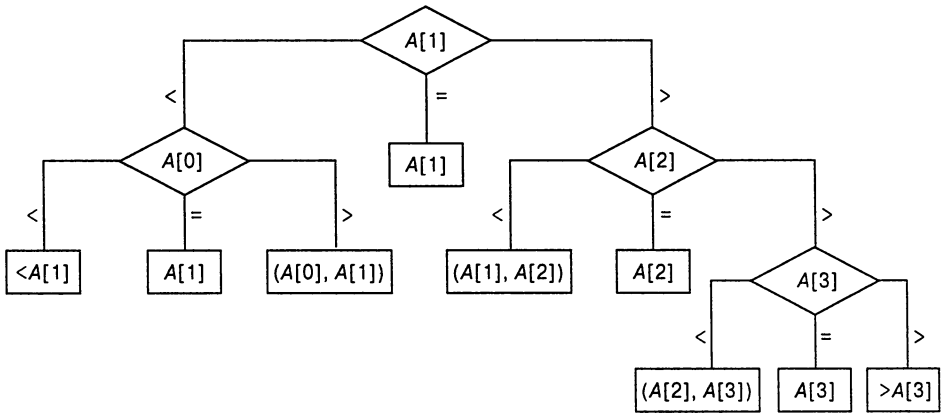


Рис. 10.4. Тернарное дерево принятия решения для бинарного поиска в четырехэлементном массиве

Эта нижняя граница меньше, чем количество сравнений при бинарном поиске в наихудшем случае, равное $\lceil \log_2 (n+1) \rceil$ для больших значений n (и не превышает $\lceil \log_2 (n+1) \rceil$ для всех натуральных n — см. упражнение 10.2.7). Можем ли мы доказать лучшую нижнюю границу или бинарный поиск оптимален? Для получения лучшей нижней границы мы должны рассмотреть бинарное, а не тернарное дерево принятия решения, такое, как показано на рис. 10.5. Внутренние узлы такого дерева соответствуют тем же тернарным сравнениям, что и ранее, но теперь они служат завершающими узлами при успешном поиске. Листья, таким образом, представляют только неудачные поиски, и всего их $n+1$ при поиске в n-элементном массиве.

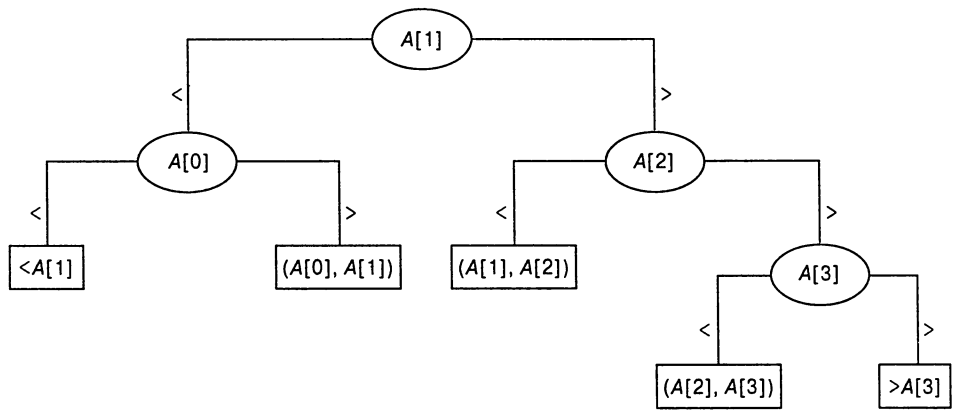


Рис. 10.5. Бинарное дерево принятия решения для бинарного поиска в четырехэлементном массиве

Сравнивая деревья принятия решения на рис. 10.4 и 10.5, мы видим, что бинарное дерево принятия решения — это просто тернарное дерево принятия решения с удаленными средними поддеревьями. Применение неравенства (10.1) к такому бинарному дереву принятия решения дает

$$C_{worst}(n) \geqslant \lceil \log_2 (n+1) \rceil . \tag{10.5}$$

Это неравенство заполняет промежуток между нижней границей и количеством сравнений, выполняемых бинарным поиском в наихудшем случае, которое также равно $\lceil \log_2 (n+1) \rceil$. Существенно более сложный анализ (см., например, [67], раздел 6.2.1) показывает, что при стандартных предположениях о параметрах поиска бинарный поиск выполняет в среднем наименьшее количество сравнений. Среднее количество сравнений у этого алгоритма равно примерно $\log_2 n - 1$ для успешного и $\log_2 (n+1)$ для неудачного поисков.

Упражнения 10.2

1. Докажите методом математической индукции, что
   а) $h \geqslant \lceil \log_2 l \rceil$ для любого бинарного дерева с высотой h и количеством листьев l.
   б) $h \geqslant \lceil \log_3 l \rceil$ для любого тернарного дерева с высотой h и количеством листьев l.
2. Рассмотрим задачу поиска медианы трехэлементного множества $\{a, b, c\}$.
   а) Чему равна информационно-теоретическая нижняя граница алгоритма для решения этой задачи, основанного на сравнении?
   б) Изобразите дерево принятия решения для алгоритма, решающего указанную задачу.
   в) Если количество сравнений в наихудшем случае у вашего алгоритма больше информационно-теоретической нижней границы, как вы думаете, существует ли алгоритм, соответствующий этой нижней границе или нет? (Либо найдите такой алгоритм, либо докажите невозможность его существования.)
3. Изобразите дерево принятия решения и найдите количество сравнений ключей в наихудшем и среднем случае для
   а) базовой трехэлементной пузырьковой сортировки;
   б) улучшенной трехэлементной пузырьковой сортировки (которая прекращает работу, если при последнем проходе не было сделано ни одного обмена).

4. Разработайте алгоритм на основе сравнений для сортировки четырехэлементного массива с наименьшим возможным количеством сравнений элементов.

5. Разработайте алгоритм на основе сравнений для сортировки пятиэлементного массива с наименьшим возможным количеством сравнений элементов.

6. Изобразите бинарное дерево принятия решения для последовательного поиска в четырехэлементном упорядоченном списке.

7. Сравните две нижние границы поиска в отсортированном массиве — $\lceil \log_3 (2n+1) \rceil$ и $\lceil \log_2 (n+1) \rceil$ — и покажите, что
   а) $\lceil \log_3 (2n+1) \rceil \leqslant \lceil \log_2 (n+1) \rceil$ для всех натуральных n;
   б) $\lceil \log_3 (2n+1) \rceil < \lceil \log_2 (n+1) \rceil$ для всех натуральных $n \geqslant n_0$.

8. *Усложненная задача поиска фальшивой монеты.* Имеется $n \geqslant 3$ идентично выглядящих монет; либо все они подлинные, либо одна из них фальшивая. Кроме того, неизвестно, легче фальшивая монета подлинной или тяжелее. У вас есть рычажные весы, при помощи которых вы можете сравнить вес любых двух множеств монет. Т.е. вы можете определить, весит ли некоторое множество монет столько же, больше или меньше другого множества, но не можете сказать, насколько. Задача состоит в том, чтобы определить, все ли монеты подлинные, и если не все, то найти фальшивую и выяснить, легче она подлинной или тяжелее.
   а) Докажите, что любой алгоритм для решения этой задачи должен выполнить как минимум $\lceil \log_3 (2n+1) \rceil$ взвешиваний в наихудшем случае.
   б) Изобразите дерево принятия решения для алгоритма, который решает эту задачу для $n = 3$ монет за два взвешивания.

9. **а)** Докажите, что не существует алгоритма, который решает задачу из упражнения 8 для $n = 4$ за два взвешивания.
   б) Изобразите дерево принятия решения для алгоритма, который решает задачу из упражнения 8 для $n = 4$ за два взвешивания при помощи дополнительной монеты, о которой достоверно известно, что она подлинная.
   в) Изобразите дерево принятия решения для алгоритма, который решает задачу из упражнения 8 для $n = 12$ за три взвешивания без использования дополнительных монет.

10. ***Турнирное дерево*** представляет собой полное бинарное дерево, отражающее результаты турнира с выбыванием: его листья представляют n игроков, участвующих в турнире, а каждый внутренний узел — побе-

дителя в игре между игроками, представленными дочерними узлами. Следовательно, победитель турнира представлен корнем дерева.

а) Чему равно общее количество игр, сыгранных в таком турнире?

б) Сколько всего раундов в таком турнире?

в) Разработайте эффективный алгоритм для определения игрока, оказывающегося на втором месте, на основании информации, полученной в ходе проведения турнира. Сколько дополнительных игр требует провести ваш алгоритм?

10.3 P, NP и NP-полные задачи

При изучении вычислительной сложности задач первое, на что смотрят и ученые, и практики в области информатики, — может ли данная задача быть разрешена при помощи некоторого алгоритма за полиномиальное время.

Определение 1. Мы говорим, что алгоритм решает задачу за полиномиальное время, если его временна́я эффективность в наихудшем случае принадлежит классу $O(p(n))$, где $p(n)$ — полином от размера входных данных n. (Обратите внимание, что, поскольку мы используем запись с большим "O", задачи, решаемые, скажем, за логарифмическое время, решаются также и за полиномиальное время.) Задачу, которая может быть решена за полиномиальное время, будем называть ***легкой*** (tractable), а задачу, которая не может быть решена за полиномиальное время, — ***трудной*** (intractable). ■

Имеется ряд причин для определения трудности задач таким образом. Во-первых, записи в табл. 2.1 и их обсуждение в разделе 2.1 говорят о том, что мы не можем решить произвольные экземпляры трудных задач за реальное время, за исключением только очень малых экземпляров. Во-вторых, хотя время работы может очень сильно отличаться для разных степеней в формуле $O(P(n))$, имеется очень мало практичных полиномиальных алгоритмов со степенью полинома больше трех. Кроме того, полиномы, ограничивающие время работы алгоритма, обычно не содержат очень больших коэффициентов. В-третьих, полиномиальные функции обладают многими удобными свойствами; в частности, как сумма, так и композиция двух полиномов всегда остаются полиномами. В-четвертых, выбор этого класса привел к разработке теории ***вычислительной сложности*** (computational complexity), которая классифицирует задачи в соответствии со сложностью их решения. Согласно этой теории, трудность остается одной и той же для всех основных моделей вычислений и всех разумных схем кодирования входной информации рассматриваемой задачи. В этом разделе мы только слегка затронем основные понятия и идеи теории сложности. Если вас интересует более полное

формальное изложение этой теории, вы без труда найдете массу литературы, посвященной этой теме (например, [110, 86]).

P и NP-задачи

Большинство задач, рассматривавшихся в этой книге, могут быть решены некоторым алгоритмом за полиномиальное время. Сюда входит вычисление произведения и наибольшего общего делителя двух целых чисел, сортировка, поиск (ключа в списке или образца в тексте), проверка связности и ацикличности графа, поиск минимального остовного дерева и кратчайших путей во взвешенном графе. (Вы без труда добавите свои примеры в этот список.) Говоря нестрого, задачи, решаемые за полиномиальное время, можно рассматривать как множество, которое ученые-теоретики в области информатики называют P. Более формальное определение включает в P только задачи ***принятия решения*** (decision problems), представляющие собой задачи, ответ на которые — "да" либо "нет".

Определение 2. Класс P представляет собой класс задач принятия решения, которые могут быть решены (детерминистическим) алгоритмом за полиномиальное время. Этот класс задач называется ***полиномиальным***. ■

Ограничение класса P только задачами принятия решения можно объяснить следующими причинами. Во-первых, разумно исключить задачи, не разрешимые за полиномиальное время из-за экспоненциально большого размера выходных данных. Такие задачи возникают естественным путем — например, генерация всех подмножеств данного множества или всех перестановок n различных элементов, — но очевидно, что по этой причине они просто не могут быть решены за полиномиальное время. Во-вторых, многие важные задачи, не являющиеся задачами принятия решения в своей естественной формулировке, могут быть приведены к ряду проблем принятия решения, которые проще изучить. Например, вместо выяснения, какое наименьшее количество цветов надо для раскраски вершин графа, так, чтобы никакие две смежные вершины не были одного цвета, мы можем выяснять, существует ли такая раскраска вершин графа не более чем m цветами при $m = 1, 2, \ldots$. (Задача раскраски вершин m цветами называется ***задачей m-раскраски*** (m-coloring problem).) Первое значение m в этом ряду, для которого решение задачи m-раскраски имеет решение, дает ответ на оптимизационную задачу раскраски графа.

Естественно задаться вопросом: не может ли *каждая* задача принятия решения быть решена за полиномиальное время? Оказывается, нет. В действительности некоторые задачи принятия решения не могут быть решены вообще, никаким алгоритмом. Такие задачи называются ***неразрешимыми*** (undecidable). Знаменитый

пример такой задачи был приведен Тьюрингом (Turing) в 1936 году.[1] Это знаменитая ***задача останова*** (halting problem): для данной компьютерной программы и входных данных определить, завершится ли выполнение программы или она будет выполняться бесконечно.

Вот на удивление короткое доказательство этого замечательного факта. От противного предположим, что A — алгоритм, который решает задачу останова, т.е. для любой программы P и входных данных I

$$A(P,I) = \begin{cases} 1, & \text{если программа } P \text{ завершается при входных данных } I; \\ 0, & \text{если программа } P \text{ не завершается при входных данных } I. \end{cases}$$

Мы можем рассматривать программу P как входные данные для нее самой и использовать выход алгоритма A для пары (P,P) для построения программы Q следующим образом:

$$Q(P) = \begin{cases} \text{завершается, если } A(P,P) = 0, \text{ т.е. если программа } P \text{ не} \\ \text{завершается при входных данных } P; \\ \text{не завершается, если } A(P,P) = 1, \text{ т.е. если программа } P \\ \text{завершается при входных данных } P. \end{cases}$$

Тогда, подставляя вместо P программу Q, получим

$$Q(Q) = \begin{cases} \text{завершается, если } A(Q,Q) = 0, \text{ т.е. если программа } Q \text{ не} \\ \text{завершается при входных данных } Q; \\ \text{не завершается, если } A(Q,Q) = 1, \text{ т.е. если программа } Q \\ \text{завершается при входных данных } Q. \end{cases}$$

Мы пришли к противоречию, поскольку ни один из двух выходов программы Q невозможен, — что и завершает доказательство.

Существуют ли задачи разрешимые, но трудные? Да, но количество известных примеров невелико, в особенности тех, которые возникают естественным путем, а не построены в качестве теоретического доказательства.

Однако имеется большое количество важных задач, для которых не найден алгоритм с полиномиальным временем работы, но и не доказана невозможность его существования. Классическая монография [40] содержит список из нескольких

[1] Это только один из множества вкладов в теоретическую кибернетику, сделанный английским ученым Аланом Тьюрингом (Alan Turing) (1912–1954). В ознаменование его заслуг ACM (Association for Computing Machinery — Ассоциация по вычислительной технике), общество, объединяющее профессионалов и исследователей в области вычислительной техники, учредило премию его имени, присуждаемую за выдающийся вклад в теоретическую кибернетику. В своем выступлении при вручении ему премии Ричард Карп (Richard Karp) [59] привел множество интересных исторических фактов о развитии теории сложности.

сотен таких задач из различных областей информатики, математики и исследования операций. Вот только несколько из наиболее известных задач, попадающих в эту категорию.

Гамильтонов цикл. Определить, имеется ли в данном графе гамильтонов цикл (путь, который начинается и заканчивается в одной и той же вершине и проходит по все остальным вершинам ровно по одному разу).

Задача коммивояжера. Найти кратчайший маршрут по n городам с известными положительными целыми расстояниями между ними (найти кратчайший гамильтонов цикл в полном графе с положительными целыми весами).

Задача о рюкзаке. Найти подмножество с наибольшей стоимостью из n предметов с заданными положительными целыми весами и стоимостями, которое может быть помещено в рюкзак с заданной положительной целой емкостью.

Задача о разделении. Даны n положительных целых чисел; требуется определить, можно ли разделить их на два непересекающихся подмножества с одинаковыми суммами.

Упаковка корзин. Даны n предметов, размеры которых представляют собой положительные рациональные числа, не превышающие 1. Их надо разместить в наименьшее количество корзин размером 1.

Раскраска графа. Для данного графа найти его хроматическое число (наименьшее количество цветов, которыми можно раскрасить вершины графа так, чтобы никакие две смежные вершины не были окрашены в один и тот же цвет).

Целочисленное линейное программирование. Найти максимальное (или минимальное) значение линейной функции нескольких целочисленных переменных при условии выполнения конечного множества ограничений в виде линейных равенств и/или неравенств.

Некоторые из этих задач являются задачами принятия решения и, само собой, являясь таковой, не имеют двойственной задачи принятия решения (как, например, задача m-раскраски для задачи раскраски графа). Общее у всех этих задач то, что они обладают экспоненциальным (или еще более быстрым) ростом количества вариантов выбора решения с ростом размера задачи n. Заметим, однако, что имеется ряд задач с тем же свойством, но решаемых за полиномиальное время. Например, задача об эйлеровом цикле, т.е. о существовании цикла, который проходит по всем ребрам данного графа ровно по одному разу, может быть решена за время $O\left(n^2\right)$ путем проверки в дополнение к связности графа, все ли его

вершины имеют четную степень. Этот пример особенно поразителен: интуитивно совершенно не ожидаешь того, что задача обхода всех ребер по одному разу (эйлеров цикл) настолько проще кажущейся похожей задачи о цикле, обходящем по одному разу все вершины (гамильтонов цикл).

Еще одно общее свойство огромного большинства задач принятия решения заключается в том, что при том, что решение задач может быть вычислительно сложным, проверка предложенного решения обычно достаточно проста и может быть выполнена за полиномиальное время (такие решения можно представить как случайным образом генерируемые кем-то и предлагаемые нам для проверки их корректности). Например, легко проверить, является ли предложенный список вершин гамильтоновым циклом для данного графа с n вершинами. Все, что для этого надо, — убедиться, что список содержит $n+1$ вершин графа, что первые n вершин в списке различны, а последняя совпадает с первой, и что каждая пара соседних вершин в списке соединяется ребром. Это наблюдение привело ученых к понятию недетерминистического алгоритма.

Определение 3. ***Недетерминистическим алгоритмом*** (nondeterministic algorithm) называется двухэтапная процедура, которая получает в качестве входа экземпляр I задачи принятия решения и делает следующее.

Недетерминистический этап ("угадывания"): генерируется произвольная строка S, которая может рассматриваться как кандидат в решение данной задачи I (но может оказаться и полной ерундой).

Детерминистический этап ("проверки"): детерминистический алгоритм получает I и S в качестве входных данных и выдает "да", если S представляет собой решение экземпляра I. (Если S не является решением I, алгоритм либо возвращает "нет", либо может вообще не завершить работу.) ■

Мы говорим, что недетерминистический алгоритм решает задачу принятия решения тогда и только тогда, когда для каждого экземпляра задачи с ответом "да" он возвращает при некотором выполнении "да" (другими словами, требуется, чтобы недетерминистический алгоритм был способен как минимум один раз "угадать" решение, и был способен проверить его корректность. И, конечно, мы не хотим, чтобы он мог ответить "да" для экземпляра, ответ для которого должен быть "нет"). Наконец, недетерминистический алгоритм является ***недетерминистическим полиномиальным*** (nondeterministic polynomial), если временна́я эффективность этапа проверки полиномиальная.

Теперь определим класс NP-задач.

Определение 4. Класс NP — это класс задач принятия решения, которые могут быть решены недетерминистическим полиномиальным алгоритмом. Этот класс задач называется ***недетерминистическим полиномиальным*** (nondeterministic polynomial). ■

Большинство задач принятия решения принадлежат классу NP. Прежде всего этот класс включает все задачи класса P: $P \subseteq NP$. Это соотношение истинно, так как если задача принадлежит классу P, мы можем использовать детерминистический полиномиальный алгоритм, который решает ее, на этапе проверки недетерминистического алгоритма, просто проигнорировав строку S, генерируемую на этапе угадывания. Но, кроме того, класс NP включает также задачу поиска гамильтонова цикла, задачу о разделении, задачу о рюкзаке, задачу коммивояжера и сотни других сложных комбинаторных задач оптимизации, перечисленных в [40]. С другой стороны, задача останова находится среди тех редких задач принятия решения, о которых известно, что они не входят в класс NP.

Это приводит к наиболее важному открытому вопросу теоретической информатики: является ли класс P истинным подмножеством NP или на самом деле оба эти класса совпадают? Мы можем записать этот вопрос как

$$P \stackrel{?}{=} NP.$$

Давайте обсудим эти два варианта. $P = NP$ должно приводить к тому, что каждая из многих сотен сложных комбинаторных задач принятия решения может быть решена алгоритмом с полиномиальным временем работы, хотя ученые не в состоянии найти такие алгоритмы несмотря на многолетние усилия. Кроме того, многие хорошо известные задачи принятия решения являются ***NP-полными*** (NP-complete). Говоря нестрого, NP-полная задача — это задача класса NP, которая так же сложна, как и любая другая задача этого класса, поскольку по определению любая другая задача класса NP может быть приведена к ней за полиномиальное время (что схематически показано на рис. 10.6).

Вот более строгое определение этой концепции.

Определение 5. Задача принятия решения D_1 называется ***полиномиально приводимой*** (polynomially reducible) к задаче принятия решения D_2, если имеется функция t, которая преобразует экземпляры D_1 в экземпляры D_2 так, что

1. t отображает все экземпляры D_1 с положительным ответом на экземпляры D_2 с положительным ответом, и все экземпляры D_1 с отрицательным ответом на экземпляры D_2 с отрицательным ответом;
2. t вычислима при помощи алгоритма с полиномиальными временем работы. ■

Из этого определения непосредственно следует, что если задача D_1 полиномиально приводима к некоторой задаче D_2, которая может быть решена за полиномиальное время, то задача D_1 также может быть решена за полиномиальное время (почему?).

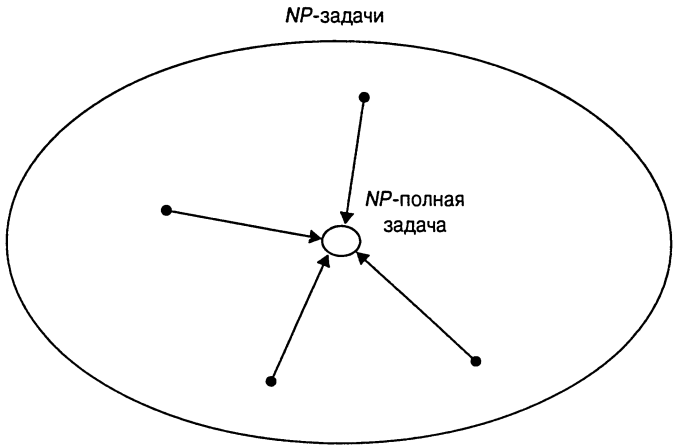


Рис. 10.6. Понятие NP-сложной задачи. Стрелками показано приведение NP-задач к NP-полной задаче за полиномиальное время

NP-полные задачи

Введем еще одно важное понятие теории вычислительной сложности — NP-полноты.

Определение 6. Задача принятия решения D является ***NP-полной*** (NP-complete), если

1. она принадлежит классу NP;
2. любая задача в NP полиномиально приводима к D. ■

Тот факт, что тесно связанные задачи принятия решения полиномиально приводимы одна к другой, не слишком удивителен. Например, докажем, что задача поиска гамильтонова цикла полиномиально приводима к версии принятия решения задачи коммивояжера. Последнюю можно сформулировать как задачу определения существования гамильтонова цикла в полном графе с положительными целыми весами, полная длина которых не превышает заданного положительного целого числа m. Мы можем отобразить граф G данного экземпляра задачи о гамильтоновом цикле на полный взвешенный граф G', представляющий экземпляр задачи коммивояжера, назначая вес, равный 1, каждому ребру из G, и добавляя ребра весом 2 между всеми парами вершин, несмежных в G. В качестве верхней границы m длины гамильтонова цикла возьмем $n = m$, где n — количество вершин в G (и G'). Очевидно, что такое преобразование может быть выполнено за полиномиальное время.

Пусть G — экземпляр задачи о гамильтоновом цикле с положительным ответом. Тогда G имеет гамильтонов цикл, и его образ в G' имеет длину n, давая

образ положительного экземпляра версии принятия решения задачи коммивояжера. И обратно: если в G' у нас есть гамильтонов цикл длиной не более n, то его длина должна быть в точности n (почему?), а следовательно, цикл должен состоять из ребер, имеющихся в G, что дает обратное отображение экземпляра версии принятия решения задачи коммивояжера с положительным ответом на экземпляр задачи о гамильтоновом цикле с положительным ответом. Полнота доказана.

Понятие NP-полноты требует, однако, полиномиальной приводимости *всех* задач в NP, как известных, так и неизвестных, к рассматриваемой задаче. При огромном количестве задач принятия решения вызывает изумление тот факт, что были найдены конкретные примеры NP-полных задач. Тем не менее этот математический подвиг был независимо совершен Стефеном Куком (Stephen Cook) из США и Леонидом Левиным из СССР.[2] В своей статье [30], датированной 1971 годом, Кук показал, что так называемая ***задача CNF-выполнимости*** является NP-полной. Эта задача работает с булевыми выражениями. Каждое булево выражение может быть представлено в нормальной конъюнктивной форме, такой как следующее выражение, в котором участвуют три булевы переменные x_1, x_2 и x_3 и их отрицания, обозначаемые как $\bar{x}_1$, $\bar{x}_2$ и $\bar{x}_3$ соответственно:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_2) \ \& \ (\bar{x}_1 \vee x_2) \ \& \ (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3).$$

В задаче CNF-выполнимости спрашивается, можно ли назначить переменным в булевом выражении значения $true$ и $false$ так, чтобы результат вычисления всего выражения был равен $true$. (Легко видеть, что для приведенной формулы это возможно: если $x_1 = true$, $x_2 = true$ и $x_3 = false$, все выражение принимает значение $true$.)

С того времени как Кук и Левин нашли первую NP-полную задачу, найдены сотни, если не тысячи других примеров NP-полных задач. В частности, широко известные задачи (или их версии принятия решения), упомянутые выше, — задача о гамильтоновом цикле, задача коммивояжера, разделение множества, упаковка корзин и раскраска графа — все они являются NP-полными задачами. Известно, однако, что если $P \neq NP$, то должны существовать NP-задачи, которые не являются ни P-задачами, ни NP-полными задачами.

Много лет ведущим кандидатом в примеры такой задачи была задача определения того, является ли заданное число простым или составным. В 2002 году профессор Маниндра Агравал (Manindra Agrawal) и его студенты Нирай Каял (Neeraj Kayal) и Нитин Саксена (Nitin Saxena) из Индийского института технологий в Капуре объявили об открытии детерминистического полиномиального

[2] Как часто случается в истории науки, переломные открытия совершаются независимо и практически одновременно несколькими учеными. На самом деле Левин рассматривал более общее понятие, чем NP-полнота, которое не ограничивалось задачами принятия решения, но его статья [73] была опубликована на два года позже работы Кука.

алгоритма для проверки простоты [3]. Их алгоритм, однако, не решает задачу разложения больших составных чисел на множители, которая представляет собой центральную часть широко используемого метода шифрования, называющегося ***алгоритмом RSA*** [96].

Показать, что задача принятия решения является NP-полной, можно в два этапа. Сначала надо показать, что рассматриваемая задача является NP-задачей, т.е. что случайным образом сгенерированная строка может быть проверена на пригодность в качестве решения задачи за полиномиальное время. Обычно этот этап весьма прост. Второй этап заключается в том, чтобы показать, что любая задача из множества NP приводима к рассматриваемой задаче за полиномиальное время. Благодаря транзитивности полиномиального приведения для выполнения этого этапа достаточно показать, что известная NP-полная задача может быть приведена к данной за полиномиальное время (см. рис. 10.7). Хотя такое преобразование может потребовать немалой изобретательности, это несравнимо проще, чем доказывать существование преобразования для каждой задачи из NP. Например, если мы уже знаем, что задача о гамильтоновом цикле является NP-полной, из ее полиномиальной приводимости к версии принятия решения задачи коммивояжера доказывает, что последняя также является NP-полной задачей (после простой проверки того, что версия принятия решения задачи коммивояжера принадлежит классу NP).

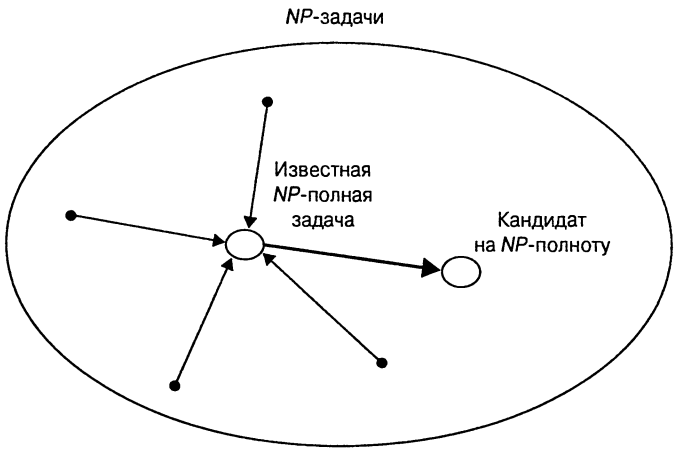


Рис. 10.7. Доказательство NP-полноты приведением

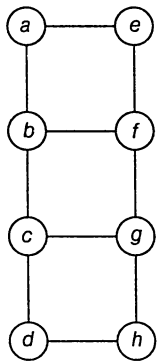
Непосредственно из определения NP-полноты следует, что если будет найден детерминистический алгоритм решения одной из NP-полных задач, то все задачи в NP могут быть решены за полиномиальное время при помощи детерминистического алгоритма, следовательно, $P = NP$. Другими словами, нахождение алгоритма с полиномиальным временем работы для одной NP-полной задачи будет

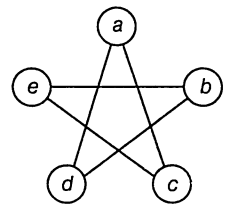
означать, что не существует качественного различия между сложностью проверки предлагаемого решения и поиска его за полиномиальное время для подавляющего большинства задач принятия решения всех видов. Такое следствие заставляет большинство ученых верить, что $P \neq NP$, хотя никто не нашел математического доказательства этой интригующей гипотезы. Интересно, что в интервью с авторами книги о жизни и открытиях 15 выдающихся ученых-кибернетиков [106] Кук не смог сказать ничего определенного о решении этой дилеммы, в то время как Левин утверждал, что следует ожидать, что $P = NP$.

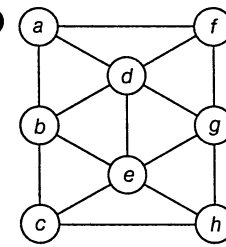
Каким бы ни был окончательный ответ на вопрос $P \stackrel{?}{=} NP$, на сегодняшний день знание того, что задача является NP-полной, имеет важные практические следствия. Это означает, что, столкнувшись с NP-полной задачей, не стоит устраивать революцию в информатике, разрабатывая полиномиальный алгоритм для всех ее экземпляров. Вместо этого следует сконцентрироваться на нескольких подходах поиска уменьшения сложности стоящей перед вами задачи. Эти подходы будут вкратце рассмотрены в следующей главе.

Упражнения 10.3

1. Игра в шахматы может рассматриваться как следующая задача принятия решения: зная, чей ход, определить для данной корректной позиции шахматных фигур, какая сторона может выиграть. Является ли эта задача принятия решения разрешимой?
2. Задача может быть решена при помощи алгоритма со временем работы $O\left(n^{\log_2 n}\right)$. Какие из следующих утверждений истинны?
   а) Задача легкая.
   б) Задача трудная.
   в) Ни одно из приведенных утверждений.
3. Приведите примеры следующих графов или поясните, почему такие примеры не могут существовать:
   а) графа с гамильтоновым циклом, но без эйлерова цикла;
   б) графа с эйлеровым циклом, но без гамильтонова цикла;
   в) графа как с гамильтоновым, так и с эйлеровым циклом;
   г) графа с циклом, включающим все вершины, но не являющимся ни гамильтоновым, ни эйлеровым.
4. Найдите хроматическое число каждого из показанных графов.

а) 

б) 

в) 

5. Разработайте алгоритм с полиномиальным временем работы для задачи 2-раскраски графа: определить, могут ли вершины данного графа быть раскрашены не более чем двумя цветами так, чтобы никакие две смежные вершины не были окрашены в один и тот же цвет.

6. Рассмотрим следующий алгоритм грубой силы для решения задачи о составных числах: последовательно проверяем все целые числа от 2 до $\lfloor n/2 \rfloor$ в качестве возможных делителей n. Если на одно из них n делится нацело, возвращаем ответ "да" (т.е. число составное), если не делится ни на одно — возвращаем ответ "нет". Почему этот алгоритм не делает задачу принадлежащей классу P?

7. Сформулируйте версию принятия решения для каждой из следующих задач и набросайте полиномиальный алгоритм для проверки, является ли предложенное решение корректным решением задачи или нет. (Считаем, что предлагаемое решение представляет корректные входные данные для вашего алгоритма верификации.)

   а) Задача о рюкзаке

   б) Задача об упаковке корзин

8. Покажите, что задача о разделении полиномиально приводима к версии принятия решения задачи о рюкзаке.

9. Покажите, что следующие три задачи полиномиально приводимы одна к другой.

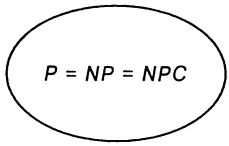
   Определить для графа $G = (V, E)$ и положительного целого $m \leqslant |V|$, содержит ли граф G ***клику*** (clique) размером m и более. (Кликой размером k графа является его полный подграф из k вершин.)

   Определить для графа $G = (V, E)$ и положительного целого $m \leqslant |V|$, имеется ли ***вершинное покрытие*** (vertex cover) графа G размером не более m. (Вершинным покрытием размером k графа $G = (V, E)$ является подмножество $V' \subseteq V$ такое, что $|V'| = k$ и для каждого ребра $(u, v) \in E$ как минимум одна из вершин u или v принадлежит V'.)

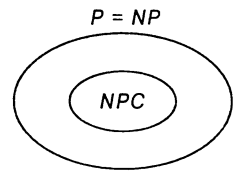
Определить для графа $G = (V, E)$ и положительного целого $m \leqslant |V|$, содержит ли граф G ***независимое множество*** (independent set) размером не менее m. (Независимым множеством размером k графа $G = (V, E)$ является подмножество $V' \subseteq V$ такое, что $|V'| = k$ и для всех $u, v \in V'$ вершины u и v *не* являются смежными в G.)

10. Какие из приведенных диаграмм не противоречат текущим знаниям о классах сложности P, NP и NPC (NP-полных задач)?

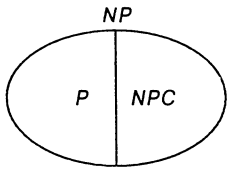
а)



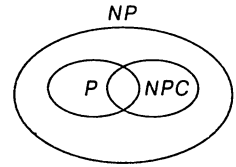
б)



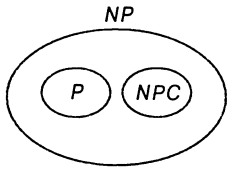
в)



г)



д)



11. Король Артур ожидает на ежегодный обед в Камелоте 150 рыцарей. К сожалению, некоторые из рыцарей в ссоре друг с другом (и король Артур знает, кто с кем). Король хочет рассадить гостей за круглым столом так, чтобы никакие два рыцаря, находящиеся в ссоре друг с другом, не оказались рядом.

а) Какая стандартная задача может использоваться для моделирования задачи короля Артура?

б) В качестве исследовательского проекта найдите доказательство того, что задача короля Артура имеет решение, если каждый рыцарь не находится в ссоре как минимум с 75 другими рыцарями.

10.4 Численные алгоритмы

Численный анализ обычно описывается как раздел информатики, посвященный алгоритмам для решения математических задач. Такое определение требует важного пояснения: рассматриваемые задачи являются задачами "непрерывной" математики — решение уравнений и систем уравнений, вычисление таких функ-

ций, как $\sin x$ и $\ln x$, вычисление интегралов и т.д. — в противоположность задачам дискретной математики, работающим с такими структурами, как графы, деревья, перестановки и сочетания. Наш интерес к эффективным алгоритмам для решения математических задач обусловлен тем, что эти задачи возникают в качестве моделей многих явлений реальной жизни как в природе, так и в обществе. Численный анализ является основной областью исследований, изучения и применения информатики. С проникновением компьютеров в бизнес и повседневную жизнь, где приходится иметь дело в основном с хранением и получением информации, *относительная* важность численного анализа сокращалась последние 30 лет. Однако его приложения, усиленные мощью современных компьютеров, продолжают распространяться во всех областях фундаментальных исследований и технологий. Итак, в какой бы области огромного мира современной информатики не находились ваши интересы, очень важно понимать специфику задач непрерывной математики.

Здесь мы не будем останавливаться на различных трудностях моделирования — задачи описания явлений реального мира математическими формулами. Считая, что это уже сделано, с какими основными препятствиями при решении математических задач нам придется столкнуться? Главное препятствие заключается в том, что большинство задач численного анализа не может быть решено точно.[3] Они должны решаться приближенно, и это обычно делается путем замены бесконечного объекта конечным приближением. Например, значение e^x в данной точке x может быть вычислено путем аппроксимации бесконечного ряда Тейлора вблизи $x = 0$ конечной суммой его первых элементов, называющейся ***полиномом Тейлора n-ой степени***:

$$e^x \approx 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}. \tag{10.6}$$

В качестве другого примера определенный интеграл функции может быть аппроксимирован конечной взвешенной суммой его значений в соответствии с ***формулой трапеций***, которую вы можете помнить с лекций по вычислительной математике:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right], \tag{10.7}$$

где $h = (b-a)/n$, $x_i = a + ih$ при $i = 0, 1, \ldots, n$ (рис. 10.8).

Ошибки таких приближений называются ***ошибками усечения***[4] (truncating error). Одной из важных задач в численном анализе является оценка величины

[3] Решение систем линейных уравнений и вычисление полиномов, рассматривавшихся в разделах 6.2 и 6.5 соответственно, являются редкими исключениями из этого правила.

[4] Используются также термины *"ошибка метода"*, *"ошибка обрыва"*. — *Прим. перев.*

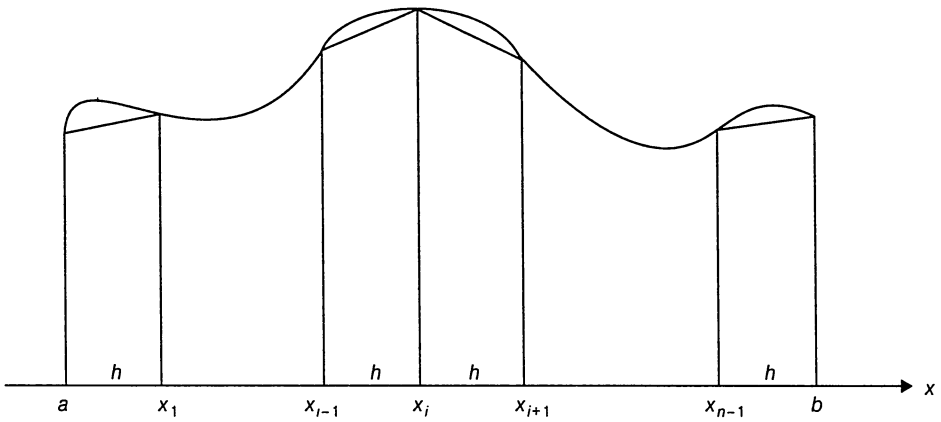


Рис. 10.8. Формула трапеций для вычисления определенного интеграла

ошибок усечения. Обычно это делается с помощью соответствующего математического инструментария. Например, для приближения (10.6) имеем:

$$\left| e^x - \left[1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} \right] \right| \leqslant \frac{M}{(n+1)!} |x|^{n+1}, \tag{10.8}$$

где $M = \max e^{\xi}$ на отрезке с конечными точками 0 и x. Эта формула позволяет определить степень полинома Тейлора, необходимую для гарантирования заданной степени точности приближения (10.6).

Например, если мы хотим вычислить $e^{0.5}$ по формуле (10.6) и гарантировать ошибку усечения не выше 10^{-4}, то можем поступить следующим образом. Во-первых, оценим значение M из формулы (10.8):

$$M = \max_{0 \leqslant \xi \leqslant 0.5} e^{\xi} \leqslant e^{0.5} < 2.$$

Используя эту границу и требуемую точность 10^{-4}, из (10.8) получим

$$\frac{M}{(n+1)!} |0.5|^{n+1} < \frac{2}{(n+1)!} 0.5^{n+1} < 10^{-4}.$$

Для решения последнего неравенства можно вычислить несколько первых значений

$$\frac{2}{(n+1)!} 0.5^{n+1} = \frac{2^{-n}}{(n+1)!},$$

что позволит найти наименьшее значение n, для которого выполняется это неравенство: оно равно 5.

Аналогично, для приближения (10.7) стандартная граница ошибки усечения дается неравенством

$$\left| \int_a^b f(x)\,dx - \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b) \right] \right| \leqslant \frac{(b-a)\,h^2}{12} M_2, \tag{10.9}$$

где $M_2 = \max |f''(x)|$ на интервале $a \leqslant x \leqslant b$. Вам придется использовать это неравенство в упражнениях к данному разделу.

Второй тип ошибки, именуемой ***ошибкой округления*** (round-off error), вызван ограниченной точностью, с которой действительные числа могут быть представлены в цифровом компьютере. Эти ошибки возникают не только для всех иррациональных чисел (которые по определению требуют бесконечного количества цифр для точного представления), но и для многих рациональных чисел. В абсолютном большинстве случаев действительные числа представляются как числа с плавающей точкой

$$\pm\, d_1 d_2 \ldots d_p \cdot B^E, \tag{10.10}$$

где B — основание системы счисления, обычно 2 или 16 (или, для глупых калькуляторов — 10), $d_1, d_2, \ldots, d_p$ — цифры ($0 \leqslant d_i < B$ для $i = 1, 2, \ldots, p$ и $d_1 > 0$, если только само число не равно 0), представляющие дробную часть числа и именуемые ***мантиссой*** (mantissa); E — целый ***показатель степени***, или ***экспонента*** (exponent) с диапазоном значений, приблизительно симметричным относительно 0.

Точность представления числа с плавающей точкой зависит от количества ***значащих цифр*** (significant digits) p в представлении (10.10). Большинство компьютеров позволяют использовать два или даже три уровня точности: ***однократную точность*** (single precision) (обычно от 6 до 7 значащих десятичных цифр), ***двойную точность*** (double precision) (от 13 до 14 значащих десятичных цифр) и ***расширенную точность*** (extended precision) (от 19 до 20 значащих десятичных цифр). Использование арифметики с большей степенью точности замедляет вычисления, но позволяет преодолеть некоторые проблемы, связанные с ошибками округлений. Может потребоваться использовать более высокую точность только на некоторых этапах работы алгоритма.

Как и в случае любого типа приближений, важно различать ***абсолютную ошибку*** (absolute error) и ***относительную ошибку*** (relative error) представления числа α^* его приближением α:

$$\text{Абсолютная ошибка} = |\alpha - \alpha^*|, \tag{10.11}$$

$$\text{Относительная ошибка} = \frac{|\alpha - \alpha^*|}{|\alpha^*|}. \tag{10.12}$$

(Относительная ошибка не определена при $\alpha^* = 0$.)

Очень большие и очень маленькие числа не могут быть представлены в арифметике с плавающей точкой из-за явлений, называющихся соответственно ***переполнением*** (overflow) и ***опустошением***, или ***антипереполнением*** (underflow). Типичные примеры переполнения возникают при перемножении больших чисел или делении на очень маленькие числа. Иногда таких проблем можно избежать, просто внеся небольшие изменения в порядок вычисления выражения (например, $(10^{29} \cdot 11^{30})/12^{30} = 10^{29} \cdot (11/12)^{30}$), заменяя выражение эквивалентным (например, вычисление C_{100}^2 не как $100!/(2!\,(100-2)!)$, а как $(100 \cdot 99)/2$) или вычисляя логарифм выражения вместо его значения.

Опустошение происходит, когда результат операции представляет собой ненулевую дробь со столь малым значением, что она не может быть представлена ненулевым числом с плавающей точкой. Обычно такие числа заменяются нулем, но при этом аппаратным обеспечением генерируется специальный сигнал, указывающий, что произошло опустошение.

Важно помнить, что кроме неточностей представления чисел арифметические операции, выполняемые компьютером, также не всегда точны. В частности, вычитание двух близких чисел с плавающей точкой может вызвать резкий рост относительной ошибки. Это явление носит название ***потери значащих разрядов*** (subtractive cancellation).

Пример 1. Рассмотрим два иррациональных числа

$$\alpha^* = \pi = 3.14159265\ldots \text{ и } \beta^* = \pi - 6 \cdot 10^{-7} = 3.14159205\ldots,$$

представленных числами с плавающей точкой $\alpha = 0.3141593 \cdot 10^1$ и $\beta = 0.3141592 \cdot 10^1$ соответственно. Относительные ошибки этих приближений малы:

$$\frac{|\alpha - \alpha^*|}{\alpha^*} = \frac{0.0000003\ldots}{\pi} < \frac{4}{3} \cdot 10^{-7}$$

и

$$\frac{|\beta - \beta^*|}{\beta^*} = \frac{0.00000005\ldots}{\pi - 6 \cdot 10^{-7}} < \frac{1}{3} \cdot 10^{-7}$$

соответственно. Относительная ошибка представления разности $\gamma^* = \alpha^* - \beta^*$ разностью представлений числами с плавающей точкой $\gamma = \alpha - \beta$ равна

$$\frac{|\gamma - \gamma^*|}{\gamma^*} = \frac{10^{-6} - 6 \cdot 10^{-7}}{6 \cdot 10^{-7}} = \frac{2}{3},$$

что является очень большим значением для относительной ошибки, несмотря на достаточно точные приближения α и β. ■

Заметим, что можно столкнуться со значительным увеличением ошибки округления, если разность с пониженной точностью будет использована в качестве делителя (мы уже сталкивались с этой проблемой при рассмотрении метода исключения Гаусса в разделе 6.2 и воспользовались для ее решения выбором ведущего элемента). Многие численные алгоритмы выполняют для типичных входных данных тысячи или даже миллионы арифметических операций, и распространение ошибок округления в них становится основным вопросом как с теоретической, так и с практической точек зрения. Для некоторых алгоритмов распространение ошибок округления идет с нарастанием. Это крайне нежелательное свойство численного алгоритма называется ***нестабильностью*** (instability). Некоторые задачи демонстрируют настолько высокую степень чувствительности к изменениям входных данных, что невозможно разработать стабильный алгоритм для их решения. Такие задачи называются ***плохо обусловленными*** (ill-conditioned).

Пример 2. Рассмотрим следующую систему двух линейных уравнений с двумя неизвестными:

$$1.001x + 0.999y = 2,$$
$$0.999x + 1.001y = 2.$$

Ее единственным решением является $x = 1$, $y = 1$. Чтобы увидеть, насколько эта система чувствительна к малым изменениям в правой части, рассмотрим систему линейных уравнений с той же матрицей коэффициентов и слегка измененной правой частью:

$$1.001x + 0.999y = 2.002,$$
$$0.999x + 1.001y = 1.998.$$

Единственным решением данной системы является $x = 2$ и $y = 0$, что очень далеко от решения предыдущей системы линейных уравнений. Обратите внимание, что матрица коэффициентов близка к сингулярной (почему?). Следовательно, малые изменения коэффициентов этой системы линейных уравнений могут привести к отсутствию решения или наличию бесконечного количества решений, в зависимости от коэффициентов в правой части. Более строгое и подробное рассмотрение оценки степени обусловленности матрицы коэффициентов можно найти в соответствующих учебниках (например, [41]). ■

Мы завершим раздел рассмотрением хорошо известной задачи поиска действительных корней квадратного уравнения

$$ax^2 + bx + c = 0 \tag{10.13}$$

для произвольных действительных коэффициентов a, b и c ($a \neq 0$). В соответствии со школьным учебником по алгебре, уравнение (10.13) имеет действительные корни тогда и только тогда, когда его дискриминант $D = b^2 - 4ac$ неотрицателен, и эти корни находятся по формуле

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{10.14}$$

Хотя формула (10.14) дает полное решение поставленной задачи с точки зрения математика, это еще не полное решение с точки зрения разработчика алгоритма. Первое серьезное препятствие заключается в вычислении квадратного корня. Даже для большинства целых D $\sqrt{D}$ является иррациональным числом, которое может быть вычислено только приближенно. Имеется метод вычисления квадратных корней, существенно лучший, чем тот, о котором обычно рассказывают в школе (он следует из ***метода Ньютона***, очень важного алгоритма для решения уравнений, который мы рассмотрим в разделе 11.4). Этот метод генерирует последовательность $\{x_n\}$ приближений $\sqrt{D}$, где D — неотрицательное число, в соответствии с формулой

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{D}{x_n}\right) \quad \text{при } n = 0, 1, \ldots, \tag{10.15}$$

где начальное приближение x_0 может быть выбрано (среди прочих возможных значений) как $x_0 = (1 + D)/2$. Нетрудно доказать, что последовательность (10.15) убывающая (если $D \neq 1$) и всегда сходится в $\sqrt{D}$. Мы можем прекратить генерацию элементов либо когда разность между двумя последовательными элементами станет меньше предопределенной допустимой ошибки $\varepsilon > 0$

$$x_n - x_{n+1} < \varepsilon,$$

либо когда x_{n+1}^2 станет достаточно близким к D. Последовательность приближений (10.15) очень быстро сходится к $\sqrt{D}$ для большинства значений D. В частности, можно доказать, что если $0.25 \leqslant D < 1$, то требуется не более четырех итераций, чтобы гарантировать, что

$$\left|x_n - \sqrt{D}\right| < 4 \cdot 10^{-15},$$

а мы всегда можем масштабировать данное значение d в интервал $[0.25, 1)$ по формуле $d = D2^p$, где p — четное целое число.

Пример 3. Давайте применим метод Ньютона для вычисления $\sqrt{2}$ (для простоты игнорируем масштабирование). Мы округляем числа до шестого знака после

запятой и используем стандартный символ $\doteq$ для указания округления.

$$x_0 = \frac{1}{2}(1+2) = 1.500000,$$
$$x_1 = \frac{1}{2}\left(x_0 + \frac{2}{x_0}\right) \doteq 1.416667,$$
$$x_2 = \frac{1}{2}\left(x_1 + \frac{2}{x_1}\right) \doteq 1.414216,$$
$$x_3 = \frac{1}{2}\left(x_2 + \frac{2}{x_2}\right) \doteq 1.414214,$$
$$x_4 = \frac{1}{2}\left(x_3 + \frac{2}{x_3}\right) \doteq 1.414214.$$

На этом мы остановимся, так как $x_4 = x_3 \doteq 1.414214$, следовательно, все остальные приближения будут такими же. Точное значение $\sqrt{2}$ равно $1.41421356\ldots$. ■

Есть ли другие препятствия, кроме вычисления квадратного корня, при использовании формулы (10.14)? Оказывается, да. Среди прочего мы сталкиваемся с потерей значащих разрядов. Если b^2 гораздо больше $4ac$, то значение $\sqrt{b^2 - 4ac}$ оказывается очень близким к $|b|$, и корень, вычисляемый по формуле (10.14), может иметь большую относительную ошибку.

Пример 4. Будем следовать статье Джорджа Форсайта (George Forsythe)[5] [38] и рассмотрим уравнение

$$x^2 - 10^5 x + 1 = 0.$$

Его истинными корнями до одиннадцатой значащей цифры являются

$$x_1^* \doteq 99999.999990 \text{ и } x_2^* \doteq 0.000010000000001.$$

Если мы воспользуемся формулой (10.14) и выполним все вычисления с использованием десятичной арифметики с плавающей точкой со, скажем, семью значащими цифрами, то получим

$$(-b)^2 \doteq 0.1000000 \cdot 10^{11},$$
$$4ac \doteq 0.4000000 \cdot 10^{1},$$
$$D \doteq 0.1000000 \cdot 10^{11},$$
$$\sqrt{D} \doteq 0.1000000 \cdot 10^{6},$$

[5] Джордж Форсайт (1917–1972), знаменитый ученый, игравший ведущую роль в становлении информатики как отдельной академической дисциплины в США. Его слова использованы в качестве эпиграфа в предисловии к данной книге.

$$x_1 = \frac{-b + \sqrt{D}}{2a} \doteq 0.1000000 \cdot 10^6,$$
$$x_2 = \frac{-b - \sqrt{D}}{2a} \doteq 0.$$

В то время как относительная ошибка приближения x_1^* к x_1 очень мала, для второго корня она оказывается очень большой:

$$\frac{|x_2 - x_2^*|}{x_2} = 1,$$

т.е. составляет 100%. ■

Чтобы избежать возникновения большой относительной ошибки, можно воспользоваться другой формулой, полученной следующим образом:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} =$$
$$= \frac{\left(-b + \sqrt{b^2 - 4ac}\right)\left(-b - \sqrt{b^2 - 4ac}\right)}{2a\left(-b - \sqrt{b^2 - 4ac}\right)} =$$
$$= \frac{2c}{-b - \sqrt{b^2 - 4ac}},$$

лишенной опасности потери значащих разрядов в знаменателе, если $b > 0$. Что касается x_2, то его можно вычислить по стандартной формуле

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

без опасности потери значащих разрядов при положительных значениях b.

Случай $b < 0$ симметричен: мы можем воспользоваться формулами

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

и

$$x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}.$$

(Случай $b = 0$ можно отнести к любому из приведенных.)

Имеется ряд иных препятствий при использовании формулы (10.14), которые связаны с ограничениями арифметики с плавающей точкой: если a слишком мало́, деление на него может вызвать переполнение; похоже, нет никакого способа избежать потери значащих разрядов при вычислении $b^2 - 4ac$, кроме использования вычислений с двойной точностью, и т.д. Все эти проблемы были преодолены

Вильямом Каганом (William Kahan) из университета в Торонто, и его алгоритм рассматривается как значительное достижение в истории численного анализа.

Будем надеяться, что этот краткий обзор пробудил в вас достаточный интерес для поиска дополнительной информации в книгах, посвященных численным алгоритмам. В следующей главе мы рассмотрим еще одну тему — три классических метода решения уравнений с одним неизвестным.

Упражнения 10.4

1. В некоторых учебниках число значащих цифр в приближении числа α^* числом α определяется как наибольшее неотрицательное целое k, для которого
$$\frac{|\alpha - \alpha^*|}{|\alpha^*|} < 5 \cdot 10^{-k}.$$
Сколько, согласно этому определению, значащих цифр в приближении числа π значением
а) 3.1415? **б)** 3.1417?
2. Известно, что $\alpha = 1.5$ является приближением некоторого числа α^* с абсолютной ошибкой, не превышающей 10^{-2}. Найдите
а) диапазон возможных значений α^*;
б) диапазон относительных ошибок такого приближения.
3. Найдите приближенное значение $\sqrt{e} = 1.648721\ldots$, полученное при помощи полинома Тейлора пятой степени около 0, и вычислите ошибку усечения такой аппроксимации. Согласуется ли полученный результат с теоретическим предсказанием, сделанным в тексте раздела?
4. Выведите формулу трапеций (10.7).
5. Воспользуйтесь формулой трапеций при $n = 4$ для приближенного вычисления указанных ниже определенных интегралов. Найдите ошибку усечения для каждого приближения и сравните ее с вычисляемой по формуле (10.9).
а) $\int_0^1 x^2 dx$ **б)** $\int_1^3 x^{-1} dx$
6. На сколько отрезков следует разделить диапазон интегрирования при вычислении $\int_0^1 e^{\sin x} dx$ по формуле трапеций, чтобы ошибка усечения была гарантированно меньше 10^{-4}? меньше 10^{-6}?
7. Решите две системы линейных уравнений и укажите, являются ли они хорошо или плохо обусловленными.

а) $\begin{aligned} 2x + 5y &= 7 \\ 2x + 5.000001y &= 7.000001 \end{aligned}$ **б)** $\begin{aligned} 2x + 5y &= 7 \\ 2x + 4.999999y &= 7.000002 \end{aligned}$

8. Напишите программу для решения квадратного уравнения $ax^2 + bx + c = 0$.

9. а) Докажите, что для любого неотрицательного числа D последовательность, полученная методом Ньютона для вычисления $\sqrt{D}$, является строго убывающей и сходится к $\sqrt{D}$ для любого значения начального приближения $x_0 > \sqrt{D}$.

б) Докажите, что если $0.25 \leqslant D < 1$ и $x_0 = (1 + D)/2$, то потребуется не более четырех итераций метода Ньютона, чтобы гарантировать, что

$$\left| x_n - \sqrt{D} \right| < 4 \cdot 10^{-15}.$$

10. Примените четыре итерации метода Ньютона для вычисления $\sqrt{3}$ и оцените абсолютную и относительную погрешность полученного приближения.

Резюме

- Для данного класса алгоритмов для решения определенной задачи *нижняя граница* указывает наилучшую возможную эффективность, которую может иметь *любой* алгоритм этого класса.
- *Тривиальная нижняя граница* основана на подсчете количества элементов входных данных задачи, которые должны быть обработаны, и количестве выходных элементов, которые должны быть произведены.
- *Информационно-теоретическая нижняя граница* обычно получается при помощи механизма *деревьев принятия решения*. Этот метод особенно полезен для алгоритмов сортировки и поиска, основанных на сравнениях. В частности, любой алгоритм сортировки на основе сравнений должен выполнить как минимум $\lceil \log_2 n! \rceil \approx n \log_2 n$ сравнений ключей в наихудшем случае, а любой алгоритм поиска в отсортированном массиве на основе сравнений в наихудшем случае должен выполнить как минимум $\lceil \log_2 (n + 1) \rceil$ сравнений ключей.
- *Метод противника* для установления нижней границы основан на следовании логике злонамеренного противника, который направляет алгоритм по наиболее длинному с точки зрения времени работы пути.
- Нижняя граница может быть установлена при помощи *приведения*, т.е. путем приведения задачи с известной нижней границей к рассматриваемой задаче.

- Теория вычислительной сложности классифицирует задачи в соответствии с их вычислительной сложностью. Принципиальное разделение задач — на *легкие* и *сложные*, которые могут и не могут быть решены за полиномиальное время. По техническим причинам теория сложности работает с *задачами принятия решения*, т.е. задачами, на которые нужно дать ответ да/нет.
- *Задача останова* представляет собой пример *неразрешимой* задачи принятия решения, т.е. она не может быть решена никаким алгоритмом.
- P представляет собой класс всех задач принятия решения, которые могут быть решены за полиномиальное время. NP — это класс задач, включающий все задачи, для которых за полиномиальное время можно проверить корректность предложенного случайным образом решения.
- Многие важные задачи из NP (такие как задача о гамильтоновом цикле) являются NP-полными: все другие задачи из множества NP приводимы к такой задаче за полиномиальное время. Первое доказательство NP-полноты было опубликовано С. Куком для *задачи о CNF-выполнимости*.
- Неизвестно, является ли P истинным подмножеством NP или $P = NP$. Это наиболее важный нерешенный вопрос теоретической информатики. Открытие алгоритма с полиномиальным временем работы для любой из тысяч известных NP-полных задач докажет, что $P = NP$.
- *Численный анализ* является разделом информатики, посвященным решению задач "непрерывной" математики. При решении таких задач возникают два основных типа ошибок — ошибки усечения (или метода), и ошибки округления. *Ошибки усечения* связаны с заменой бесконечных объектов их конечными приближениями, а *ошибки округления* — с неточностями представления чисел в цифровых компьютерах.
- *Потеря значащих разрядов* происходит вследствие вычитания двух близких чисел с плавающей точкой и может привести к резкому увеличению относительной ошибки округления, поэтому таких ситуаций следует избегать (либо изменяя вид выражения, либо используя более высокую точность при вычислении такой разности).
- Написание универсальной программы для решения квадратного уравнения $ax^2 + bx + c = 0$ — сложная задача. Вычисление квадратного корня осуществляется при помощи *метода Ньютона*; избежать потери значащих разрядов можно, используя различные формулы для корней уравнения (в зависимости от того, положительно или отрицательно значение b) и вычисляя дискриминант $b^2 - 4ac$ с удвоенной точностью.

Глава 11

Преодоление ограничений

> Продолжайте искать новые идеи даже среди успешно использующихся другими. Идея будет оригинальна применением к задаче, над которой вы работаете.
>
> — Томас Эдисон (Thomas Edison) (1847–1931)

Как вы узнали из предыдущей главы, многие задачи трудно решить алгоритмически. В то же время многие из них настолько важны, что мы не можем просто оставить все как есть и ничего не предпринимать. В этой главе будут рассмотрены несколько способов работы с такими сложными задачами.

В разделах 11.1 и 11.2 рассматриваются два метода разработки алгоритмов — ***поиск с возвратом*** (backtracking) и ***метод ветвей и границ*** (branch-and-bound) — которые зачастую делают возможным решение как минимум некоторых больших экземпляров сложных комбинаторных задач. Обе стратегии можно рассматривать как усовершенствование исчерпывающего перебора, рассмотренного в разделе 3.4. В отличие от исчерпывающего перебора указанные методы строят кандидатов в решения по одному компоненту и оценивают частично построенные решения: если потенциальных значений оставшихся компонентов, которые могли бы привести к корректному решению, не имеется, то оставшиеся компоненты решения не генерируются. Такой подход делает возможным решение некоторых больших экземпляров сложных комбинаторных задач, хотя в наихудшем случае мы все равно сталкиваемся с экспоненциальным ростом, присущим исчерпывающему перебору.

Оба метода основаны на построении ***дерева пространства состояний*** (state-space-tree), узлы которого отражают конкретные выборы, сделанные для компонентов решения. Оба метода останавливаются в узле, если можно гарантировать, что невозможно получить решение, рассматривая решения, соответствующие потомкам данного узла. Между собой методы отличаются природой задач, к которым они могут быть применены. Метод ветвей и границ применим только к оптимизационным задачам, поскольку основан на вычислении границ возможных значений целевой функции. Метод поиска с возвратом чаще применим к задачам, не являющимся задачами оптимизации, но не ограничивается ими. Другое отличие

между этими методами заключается в порядке генерации узлов дерева пространства состояний. При поиске с возвратом это дерево обычно строится в глубину (аналогично поиску в глубину). Метод ветвей и границ может генерировать узлы согласно различным правилам; наиболее естественным является так называемое правило выбора наилучшего варианта, о котором рассказывается в разделе 11.2.

В разделе 11.3 мы отходим от идеи точного решения задачи. Представленные здесь алгоритмы решают задачи приближенно, но быстро. В частности, мы рассмотрим некоторые приближенные алгоритмы для задачи коммивояжера и задачи о рюкзаке. При рассмотрении задачи коммивояжера мы познакомимся с простым жадным алгоритмом и алгоритмом, основанным на тесной связи между минимальным остовным деревом и кратчайшим гамильтоновым циклом. При рассмотрении задачи о рюкзаке мы сначала познакомимся с жадным алгоритмом, а затем – с параметрическим семейством полиномиальных алгоритмов, которые позволяют получить сколь угодно хорошие приближения.

Раздел 11.4 посвящен алгоритмам решения нелинейных уравнений. После краткого обсуждения этой очень важной задачи мы рассмотрим три классических метода для поиска приближенных значений корней: метод деления пополам, метод ложной позиции и метод Ньютона.

11.1 Поиск с возвратом

В книге (в частности, в разделах 3.4 и 10.3) мы уже сталкивались с задачами, в которых требовалось найти элемент с некоторыми свойствами из области, растущей экспоненциально (или даже быстрее) с ростом размера входных данных: гамильтонов цикл среди всех перестановок вершин графа, наиболее ценное подмножество предметов в экземпляре задачи о рюкзаке и т.п. Из раздела 10.3 вы узнали, почему многие такие задачи вряд ли могут когда-либо быть решены за полиномиальное время. Вспомним также, что в разделе 3.4 мы выяснили, что такие задачи могут быть решены (по крайней мере в принципе) методом исчерпывающего перебора. Метод исчерпывающего перебора предполагает генерацию всех решений-кандидатов и выбор из них корректного решения (или решений) с требуемыми свойствами.

Поиск с возвратом представляет собой более интеллектуальный вариант этого подхода. Основная идея состоит в построении решения по одному компоненту и выяснении, может ли дальнейшее построение привести к корректному решению. Если продолжить построение можно без нарушения ограничений задачи, оно продолжается путем выбора первого допустимого варианта для следующего компонента. Если таких вариантов нет, то никакие варианты для *всех* оставшихся компонентов не рассматриваются. Алгоритм в этой ситуации возвращается

к последнему построенному компоненту и заменяет его следующим допустимым компонентом этого уровня.

Удобно реализовать работу такого алгоритма при помощи построения дерева рассмотренных вариантов выбора, которое называется ***деревом пространства состояний*** (state-space tree). Его корень представляет начальное состояние перед началом поиска. Узлы на первом уровне дерева представляют варианты выбора, сделанные для первого компонента решения, узлы на втором уровне — варианты выбора второго компонента и т.д. Узел дерева пространства состояний называется ***обещающим*** (promising), если он соответствует частично построенному решению, которое может привести к полному решению; в противном случае он называется ***бесперспективным*** (unpromising). Листья представляют собой либо бесперспективные тупики, либо полные решения, найденные алгоритмом. В большинстве случаев дерево пространства состояний для алгоритма поиска с возвратом строится способом поиска в глубину. Если текущий узел является обещающим, его дочерний узел генерируется путем добавления первого из остающихся корректных вариантов для следующего компонента решения, и алгоритм переходит к работе с этим дочерним узлом. Если текущий узел бесперспективен, алгоритм возвращается к родительскому узлу и рассматривает очередной возможный вариант для его последнего компонента; если такого варианта нет, алгоритм поднимается вверх по дереву на один уровень и продолжает работу. Наконец, по достижении полного решения задачи, алгоритм либо завершает работу (если требуется только одно решение), либо продолжает поиск прочих возможных решений.

Задача о n ферзях

В качестве первого примера рассмотрим постоянную любимицу авторов учебников — ***задачу о n ферзях***. Задача заключается в размещении n ферзей на шахматной доске размером $n \times n$ так, чтобы никакие два ферзя не угрожали друг другу, находясь на одной горизонтали, вертикали или диагонали. Для $n = 1$ задача имеет тривиальное решение; легко убедиться, что для $n = 2$ и $n = 3$ решений не существует. Поэтому начнем с задачи с четырьмя ферзями и решим ее с помощью поиска с возвратом. Поскольку каждый ферзь должен находиться на собственной горизонтали, все, что надо, — это указать вертикаль для каждого ферзя на доске, показанной на рис. 11.1.

Начинаем с пустой доски и помещаем ферзя 1 в первую возможную позицию на его горизонтали; это позиция на вертикали 1. Затем мы размещаем ферзя 2 в первую допустимую позицию (после неудачных попыток размещения на вертикалях 1 и 2) на вертикали 3, в квадрате $(2, 3)$. Это тупиковая позиция, поскольку при ней оказывается невозможно разместить третьего ферзя так, чтобы он не был под боем. Соответственно, алгоритм осуществляет возврат к предыдущему состоянию и помещает ферзя в позицию $(2, 4)$. После этого третий ферзь может

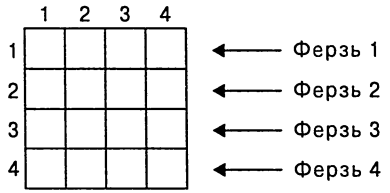


Рис. 11.1. Доска для задачи о четырех ферзях

быть помещен только в позицию $(3, 2)$, что приводит к очередному тупику. После этого алгоритм осуществляет возврат к ферзю 1 и помещает его в очередной допустимой позиции $(1, 2)$. Ферзь 2 может быть размещен только в позиции $(2, 4)$, ферзь 3 — только в позиции $(3, 1)$, а ферзь 4 — в позиции $(4, 3)$, что дает решение поставленной задачи. Соответствующее описанному поиску решения дерево пространства состояний показано на рис. 11.2.

Если требуется найти другие решения поставленной задачи (сколько всего их у задачи о четырех ферзях?), алгоритм может просто продолжить операции с листа, в котором оказалось найдено решение. В качестве альтернативы можно использовать симметрию шахматной доски.

Задача о гамильтоновом цикле

В качестве следующего примера рассмотрим задачу поиска гамильтонова цикла графа, показанного на рис. 11.3*а*.

Без потери общности можно считать, что если гамильтонов цикл существует, то он начинается в вершине a. Соответственно, мы делаем вершину a корнем дерева пространства состояний (рис. 11.3*б*). Первый компонент будущего решения, если таковое существует, является первой промежуточной вершиной строящегося гамильтонова цикла. Используя алфавитный порядок для разрешения неоднозначности при выборе вершин, смежных с вершиной a, мы выбираем вершину b. После вершины b алгоритм обращается к вершинам c, d, e и, наконец, к f, которая является тупиком. Соответственно алгоритм возвращается к вершине c — первой вершине, которая позволяет получить отличное от рассмотренного решение. Переход от c к e не приводит к решению, и алгоритм должен вернуться к вершине b. Из вершины b мы попадаем в вершину f, а за ней — в вершины e, c и d, из которой можно вполне законным путем вернуться в a, что дает гамильтонов цикл a, b, f, e, c, d, a. Если мы хотим найти еще какой-то гамильтонов цикл, то можем продолжить процесс возврата из листа, соответствующего найденному решению.

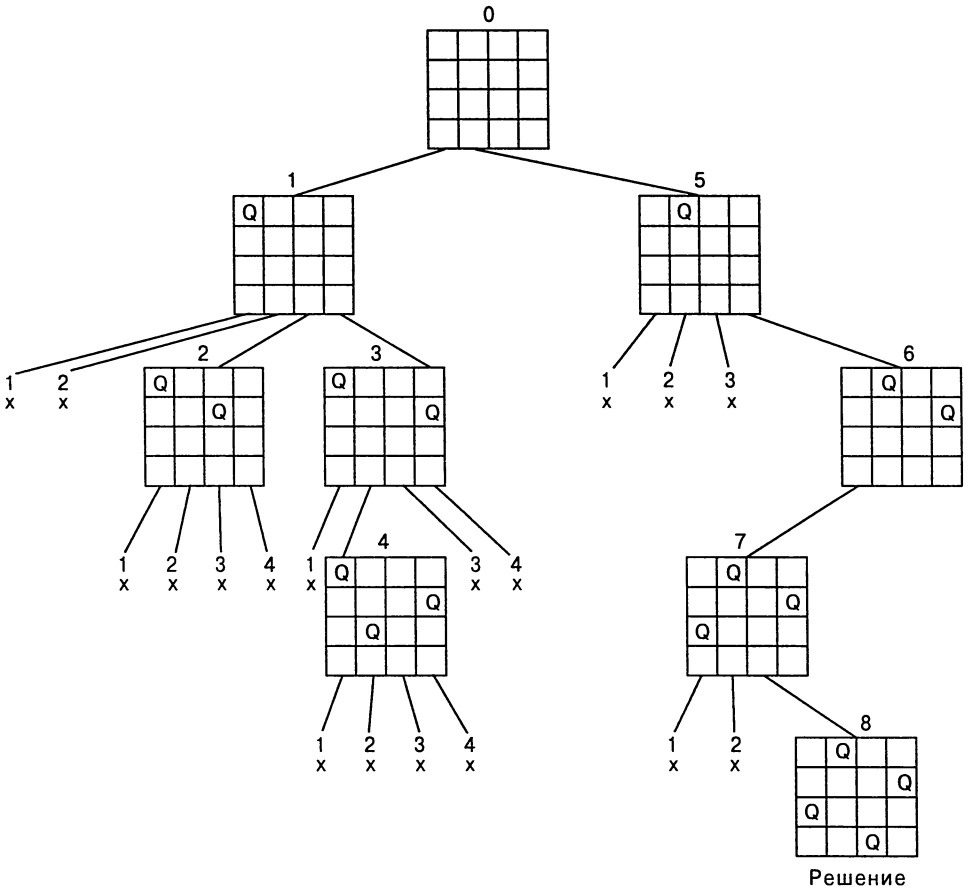


Рис. 11.2. Дерево пространства состояний задачи о четырех ферзях. Крестиками помечены неудачные попытки разместить ферзей на указанных вертикалях. Числа вверху указывают порядок генерации узлов

Задача о сумме подмножества

Последним примером применения метода поиска с возвратом будет ***задача о сумме подмножества*** (subset-sum problem): требуется найти подмножество данного множества $S = \{s_1, \ldots, s_n\}$, состоящего из n натуральных чисел, такое, что сумма его элементов равна заданному натуральному числу d. Например, для $S = \{1, 2, 5, 6, 8\}$ и $d = 9$ имеется два решения — $\{1, 2, 6\}$ и $\{1, 8\}$. Само собой, некоторые экземпляры данной задачи могут не иметь решений.

Удобно отсортировать элементы множества в возрастающем порядке. Будем считать, что

$$s_1 \leqslant s_2 \leqslant \cdots \leqslant s_n.$$

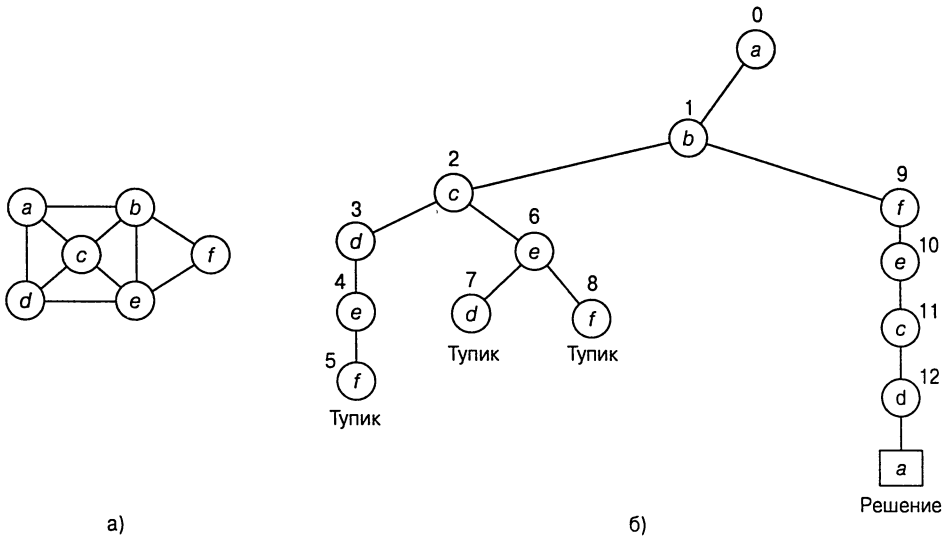


Рис. 11.3. *а*) Граф. *б*) Дерево пространства состояний для поиска гамильтонова цикла. Числа над узлами указывают порядок генерации узлов

Дерево пространства состояний может быть построено как бинарное дерево, как показано на рис. 11.4 для экземпляра $S = \{3, 5, 6, 7\}$ и $d = 15$. Корень дерева представляет начальную точку, в которой не принято решение ни по одному из элементов множества. Его левый и правый дочерние узлы представляют, соответственно, включение s_1 в искомое подмножество и отсутствие в нем этого элемента. Аналогично, переход влево от узла на первом уровне означает включение s_2 в искомое подмножество, а переход вправо — отсутствие этого элемента в искомом множестве. Таким образом, путь от корня к узлу на i-ом уровне дерева указывает, какие из первых i чисел будут включены в подмножество, представленное этим узлом.

Мы записываем значение s', суммы этих чисел, в узле. Если s' равно d, мы получаем решение поставленной задачи. Мы можем либо сообщить о нем и прекратить работу алгоритма, либо, если надо найти все решения, выполнить очередной возврат к родительскому узлу и продолжить работу. Если s' не равно d, мы можем завершить работу с узлом как с бесперспективным при выполнении любого из двух условий:

$$s' + s_{i+1} > d \text{ (сумма } s' \text{ слишком велика)},$$

$$s' + \sum_{j=i+1}^{n} s_j < d \text{ (сумма } s' \text{ слишком мала)}.$$

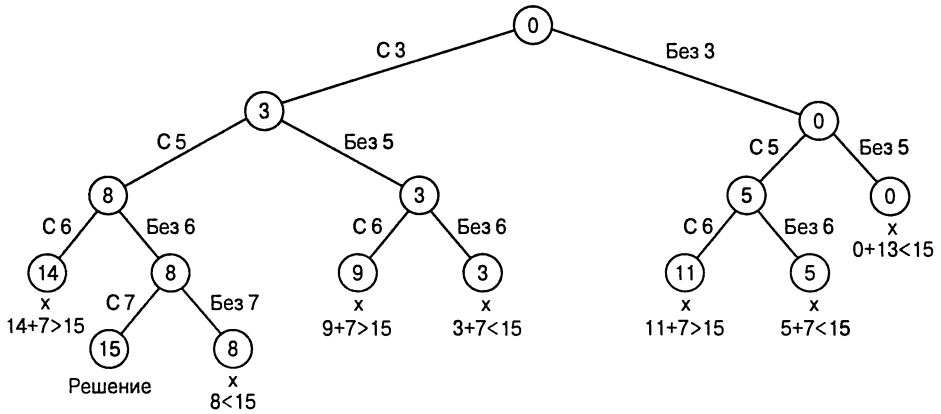


Рис. 11.4. Полное дерево пространства состояний алгоритма поиска с возвратом, примененного к экземпляру задачи о сумме подмножества $S = \{3, 5, 6, 7\}$ и $d = 15$. Число внутри узла представляет собой сумму элементов, включенных в подмножество, представленное этим узлом. Неравенства под листьями указывают причину прекращения их обработки

Общие замечания

Большинство алгоритмов поиска с возвратом удовлетворяют следующему описанию. Выход алгоритма поиска с возвратом можно рассматривать как кортеж из n элементов $(x_1, x_2, \ldots, x_n)$, где каждый элемент x_i является элементом некоторого конечного линейного упорядоченного множества S_i. Например, для задачи о n ферзях каждое S_i представляет собой множество целых чисел (номер вертикали) от 1 до n. Может потребоваться, чтобы кортеж удовлетворял некоторым дополнительным ограничениям (например, чтобы ни один ферзь не угрожал другому в задаче о n ферзях). В зависимости от задачи все кортежи-решения могут быть одной длины (как, например, в задачах о n ферзях или о гамильтоновом цикле) или иметь разные размеры (как в задаче о сумме подмножества). Алгоритм поиска с возвратом генерирует, явно или неявно, дерево пространства состояний; его узлы представляют частично построенные кортежи с первыми i элементами, определенными предыдущими действиями алгоритма. Если такой кортеж $(x_1, x_2, \ldots, x_i)$ не является решением, алгоритм ищет следующий элемент в S_{i+1}, который согласуется со значениями из $(x_1, x_2, \ldots, x_i)$ и ограничениями задачи, и добавляет его в кортеж в качестве $(i + 1)$-го элемента. Если такой элемент не существует, алгоритм возвращается к рассмотрению следующего значения x_i, и т.д.

Для запуска алгоритма поиска с возвратом приведенный псевдокод должен быть вызван для $i = 0$; $X\,[1..0]$ представляет пустой кортеж.

Алгоритм $Backtrack(X[1..i])$
// Шаблон обобщенного алгоритма поиска с возвратом
// **Входные данные:** Массив $X[1..i]$, определяющий первые i
// обещающих компонентов решения
// **Выходные данные:** Все кортежи, представляющие решения
// задачи
if $X[1..i]$ является решением
write $X[1..i]$
else // См. упражнение 8
for (для) каждого элемента $x \in S_{i+1}$, согласующегося
с $X[1..i]$ и удовлетворяющего ограничениям **do**
$X[i+1] \leftarrow x$
$Backtrack(X[1..i+1])$

Успешность решения небольших экземпляров трех различных задач ранее в этом разделе не должна ввести вас в заблуждение и заставить сделать неверный вывод о том, что поиск с возвратом — очень эффективный метод. В наихудшем случае ему может потребоваться сгенерировать всех возможных кандидатов из экспоненциально (или еще быстрее) растущего пространства состояний решаемой задачи. Надежда остается на то, что алгоритм поиска с возвратом будет способен обрéзать достаточное количество ветвей дерева пространства решений до того, как исчерпает отведенное ему время или память (или и то, и другое). Успешность этой стратегии колеблется в очень широких пределах, и не только от задачи к задаче, но и между экземплярами одной и той же задачи.

Имеется несколько хитростей, которые помогают снизить размер дерева пространства состояний. Одна из них — воспользоваться часто присущей комбинаторным задачам симметрией. Например, доска из задачи о n ферзях обладает несколькими симметриями, так что некоторые решения могут быть получены из других с помощью отражений или поворотов. Отсюда, в частности, следует, что нам не надо рассматривать размещение первого ферзя на последних $\lfloor n/2 \rfloor$ вертикалях, поскольку любое решение с первым ферзем в клетке $(1, i)$, $\lceil n/2 \rceil \leqslant i \leqslant n$, может быть получено путем отражения (какого?) из решения, в котором первый ферзь находится в клетке $(1, n-i+1)$. Это наблюдение снижает размер дерева примерно наполовину. Второй трюк заключается в предварительном назначении значений одному или нескольким компонентам решения, как было сделано в примере с гамильтоновым циклом. Предварительная сортировка в примере задачи о сумме подмножества продемонстрировала потенциальные достоинства еще одной возможности — перегруппировки данных экземпляра задачи.

Было бы крайне желательно оценить размер дерева пространства состояний алгоритма поиска с возвратом. Однако, как правило, это слишком сложная для аналитического решения задача. Кнут [62] предложил генерировать случайный путь

от корня к листу и использовать информацию о количестве выборов, доступных в процессе генерации пути, для оценки размера дерева. В частности, пусть c_1 — количество значений первого компонента x_1, которые согласуются с ограничениями задачи. Мы случайным образом выбираем одно из этих значений (с равной вероятностью $1/c_1$) для перехода к одному из c_1 дочерних по отношению к корню узлов. Повторяя эту операцию для c_2 возможных значений x_2, которые согласуются с x_1 и другими ограничениями, мы переходим в один из c_2 возможных дочерних по отношению к рассматриваемому узлов. Этот процесс продолжается до тех пор, пока после случайного выбора значений для $x_1, x_2, \ldots, x_n$ не будет достигнут лист. Считая, что узлы на уровне i имеют в среднем по c_i дочерних узлов, общее количество узлов в дереве можно оценить как

$$1 + c_1 + c_1c_2 + \cdots + c_1c_2 \ldots c_n.$$

Генерируя несколько таких оценок и вычисляя их среднее, можно получить полезную информацию о реальном размере дерева, хотя стандартное отклонение этой случайной величины может быть достаточно большим.

В заключение следует сделать три замечания по поводу поиска с возвратом. Во-первых, этот метод обычно он применяется для сложных комбинаторных задач, для точного решения которых, возможно, не существует эффективных алгоритмов. Во-вторых, в отличие от исчерпывающего поиска, который чрезвычайно медленно работает для всех экземпляров задачи, поиск с возвратом как минимум оставляет надежду на то, что решение некоторых экземпляров нетривиальных размеров будет найдено за приемлемое время. Это в особенности справедливо для оптимизационных задач, где идея поиска с возвратом может получить дальнейшее развитие при помощи вычисления качества частично построенных решений. Как именно это делается, рассказано в следующем разделе. И, в-третьих, даже если поиск с возвратом не удаляет ни одного элемента из пространства состояний задачи и в результате генерирует все его элементы, то он все равно обеспечивает особый метод генерации, который может быть ценен сам по себе.

Упражнения 11.1

1. а) Продолжите поиск с возвратом в задаче о четырех ферзях, начатый в тексте раздела, и найдите второе решение этой задачи.

б) Поясните, как можно использовать симметрию доски для поиска второго решения задачи о четырех ферзях.

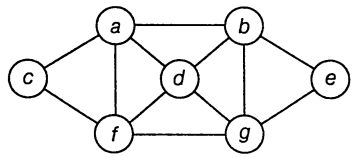
2. а) Каким будет *последнее* решение задачи о пяти ферзях, найденное алгоритмом поиска с возвратом?

б) Воспользуйтесь симметрией доски для поиска как минимум четырех других решений данной задачи.

3. **а)** Реализуйте алгоритм поиска с возвратом для задачи о n ферзях на языке программирования по вашему выбору. Выполните вашу программу для различных значений n и выясните количество узлов в соответствующих деревьях пространств состояний. Сравните полученные значения с количеством решений-кандидатов, генерируемых алгоритмом исчерпывающего перебора.

   б) Для каждого значения n, для которого выполняется программа из части a данного упражнения, оцените размер дерева пространства состояний методом, описанным в тексте раздела 11.1, и сравните полученную оценку с реальным количеством узлов дерева.

4. Примените поиск с возвратом к задаче о гамильтоновом цикле в следующем графе:



5. Примените поиск с возвратом для решения задачи о 3-раскраске графа, показанного на рис. 11.3*а*.

6. Сгенерируйте все перестановки множества $\{1, 2, 3, 4\}$ методом поиска с возвратом.

7. **а)** Примените поиск с возвратом для решения следующего экземпляра задачи о сумме подмножества: $S = \{1, 3, 4, 5\}$ и $d = 11$.

   б) Будет ли алгоритм поиска с возвратом корректно работать при использовании только одного из двух неравенств для завершения обработки узла как бесперспективного?

8. Общий шаблон алгоритма поиска с возвратом, приведенный в тексте раздела 11.1, работает корректно только в случае, если не существует решения, представляющего собой префикс другого решения задачи. Измените псевдокод таким образом, чтобы он корректно работал и для задач с решениями такого рода.

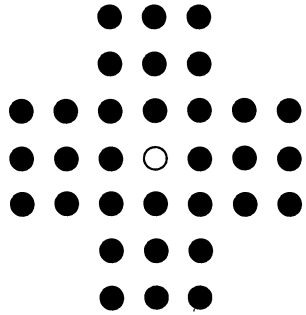
9. Напишите программу, реализующую алгоритм поиска с возвратом

   а) для задачи о гамильтоновом цикле;

   б) для задачи m-раскраски графа.

10. *Игра Hi-Q.* В этой игре-головоломке участвуют 32 шашки, изначально размещенные на доске так, как показано ниже. Каждая шашка может перемещаться, перепрыгивая через непосредственного соседа по вертикали или горизонтали на пустое место; шашка, через которую был

осуществлен прыжок, удаляется с доски. Цель игры — удалить с доски 31 шашку так, чтобы последняя шашка оказалась в центре доски.



Разработайте и реализуйте алгоритм поиска с возвратом для решения этой головоломки.

11.2 Метод ветвей и границ

Вспомним основную идею метода поиска с возвратом, рассматривавшегося в предыдущем разделе, — обрезка ветви дерева пространства состояний задачи, как только можно сделать вывод, что она не ведет к решению. Эта идея может быть усилена при работе с оптимизационными задачами, которые должны минимизировать или максимизировать целевую функцию, обычно при наличии некоторых ограничений (длина маршрута, стоимость выбранных предметов, стоимость назначений и т.п.). Заметим, что в стандартной терминологии задач оптимизации ***допустимое решение*** (feasible solution) представляет собой точку в пространстве состояний задачи, которая удовлетворяет всем ее ограничениям (например, гамильтонов цикл в задаче коммивояжера, подмножество предметов с общим весом, не превышающим емкость рюкзака), в то время как ***оптимальное решение*** (optimal solution) является допустимым решением с наилучшим значением целевой функции (например, кратчайший гамильтонов цикл, наиболее ценное подмножество предметов, помещающихся в рюкзак).

По сравнению с методом поиска с возвратом метод ветвей и границ требует:

- способа получить для каждого узла дерева пространства состояний границу наилучшего значения целевой функции[1] для всех решений, которые могут быть получены путем дальнейшего добавления компонентов к частичному решению, представленному узлом;
- значение наилучшего решения, полученного к этому моменту.

[1]Эта граница должна быть нижней границей для задачи минимизации и верхней — для задачи максимизации.

Если такая информация доступна, мы можем сравнивать значение границы узла со значением наилучшего решения, полученного к этому моменту: если значение границы не лучше значения уже имеющегося наилучшего решения — т.е. не меньше в случае задачи минимизации или не больше в случае задачи максимизации, — то такой узел является бесперспективным и его обработка может быть завершена (иногда говорят, что эта ветвь обрезается), поскольку ни одно получаемое из него решение не может оказаться лучше того, что уже имеется. В этом заключается основная идея метода ветвей и границ.

В общем случае мы завершаем путь поиска в текущем узле дерева пространства состояний алгоритма ветвей и границ по одной из трех следующих причин.

- Значение границы узла не лучше значения наилучшего решения, найденного к этому моменту.
- Узел не представляет допустимых решений из-за нарушения ограничений, налагаемых задачей.
- Подмножество допустимых решений, представляемое узлом, состоит из одного элемента (следовательно, дальнейший выбор невозможен) — в этом случае мы сравниваем значение целевой функции для этого допустимого решения со значением целевой функции наилучшего полученного к настоящему моменту решения и обновляем последнее текущим, если новое решение оказывается лучше.

Задача о назначениях

Проиллюстрируем метод ветвей и границ, применяя его к задаче о назначениях, когда имеется n работников, которым надо выполнить n заданий, и надо найти распределение заданий по работникам с наименьшей общей стоимостью. С этой задачей мы познакомились в разделе 3.4, где она решалась методом исчерпывающего перебора. Вспомним, что экземпляр задачи о назначениях определяется матрицей C размером $n \times n$, так что мы можем сформулировать задачу следующим образом: выбрать по одному элементу в каждой строке матрицы так, чтобы никакие два выбранных элемента не располагались в одном столбце, а общая сумма выбранных элементов была минимальной. Мы продемонстрируем решение этой задачи при помощи метода ветвей и границ на том же небольшом экземпляре задачи, который рассматривался в разделе 3.4:

$$C = \begin{array}{|cccc|l} \multicolumn{4}{c}{\text{Задание 1}\quad \text{Задание 2}\quad \text{Задание 3}\quad \text{Задание 4}} & \\ \hline 9 & 2 & 7 & 8 & \text{Работник } a \\ 6 & 4 & 3 & 7 & \text{Работник } b \\ 5 & 8 & 1 & 8 & \text{Работник } c \\ 7 & 6 & 9 & 4 & \text{Работник } d \\ \hline \end{array}$$

Как найти нижнюю границу стоимости оптимального решения без реального решения задачи? Можно сделать это разными путями. Например, ясно, что стоимость любого решения, включая оптимальное, не может быть меньше суммы наименьших элементов в каждой из строк матрицы. Для приведенного примера эта сумма равна $2 + 3 + 1 + 4 = 10$. Важно отметить, что это значение не является суммой ни одного из допустимых выборов (3 и 1 находятся в одном столбце матрицы), это просто нижняя граница стоимости любого выбора, соответствующего ограничениям задачи. Мы можем (и будем) применять такие же рассуждения и к частично построенным решениям. Например, для любого допустимого решения, которое выбирает из первой строки 9, нижняя граница равна $9 + 3 + 1 + 4 = 17$.

Перед тем как приступить к построению дерева пространства состояний, надо сделать еще один комментарий. Он касается порядка генерации узлов дерева. Вместо генерации одного дочернего узла по отношению к последнему обещающему, как это делалось при поиске с возвратом, мы будем генерировать все дочерние узлы для наиболее обещающего среди незавершенных листьев текущего дерева (незавершенные, т.е. все еще обещающие листья иногда называют ***живыми*** (live)). Как можно определить, какие из узлов являются наиболее обещающими? Это можно сделать, сравнивая нижние границы живых листьев. Разумно рассматривать как наиболее обещающий узел с наилучшей границей, хотя это, конечно, не препятствует тому, что оптимальное решение будет в конечном итоге принадлежать иной ветви дерева пространства состояний. Такая стратегия называется ***методом ветвей и границ с выбором наилучшего варианта*** (best-first branch-and-bound).

Вернемся к экземпляру задачи о назначениях, с которой мы сталкивались ранее, и начнем с корня, который соответствует отсутствию какого-либо выбора элементов матрицы стоимости. Как уже говорилось, значение нижней границы (которое мы обозначим как lb) для корня равно 10. Узлы на первом уровне дерева соответствуют четырем элементам (заданиям) в первой строке матрицы, поскольку все они могут быть потенциальными первыми компонентами решения (рис. 11.5).

Итак, мы имеем четыре живых листа (узлы от 1 до 4), которые могут содержать оптимальное решение. Наиболее обещающим среди них является узел 2, поскольку он имеет наименьшее значение нижней границы. Следуя стратегии выбора наилучшего варианта, исследуем сперва эту ветвь, перед тем как перейти к остальным. Из рассматриваемого листа выходят три ветви, соответствующие выбору элемента из второй строки, не находящегося во втором столбце, и представляющие три различные задания, назначенные работнику b (рис. 11.6).

Из шести листьев (узлы 1, 3, 4, 5, 6 и 7), которые могут содержать оптимальное решение, мы вновь выбираем лист с наименьшим значением нижней границы, а именно — узел 5. Сначала рассмотрим выбор третьего элемента из строки c (т.е. назначение задания 3 работнику c) — при этом у нас не остается иного выбора, кроме назначения задания 4 работнику d, что дает нам лист 8 (рис. 11.7),

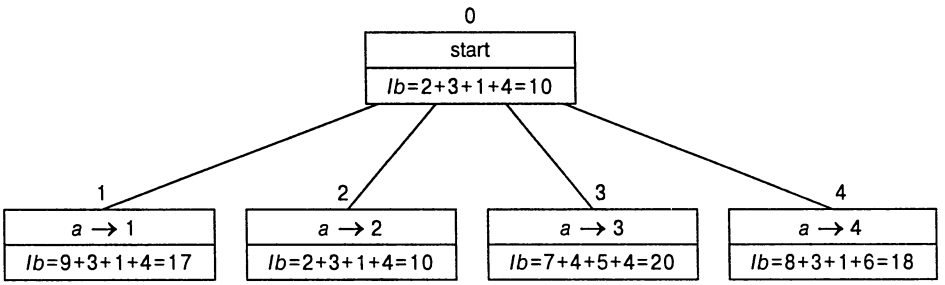


Рис. 11.5. Уровни 0 и 1 дерева пространства состояний для экземпляра задачи о назначениях, решаемой методом ветвей и границ с выбором наилучшего варианта. Числа над узлами указывают порядок генерации последних. Поля узла указывают номер назначенного задания работнику a и значение нижней границы lb для данного узла

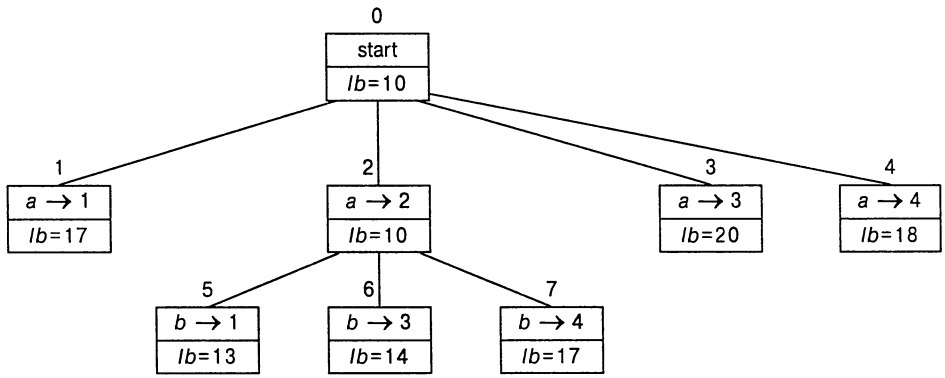


Рис. 11.6. Уровни 0, 1 и 2 дерева пространства состояний для экземпляра задачи о назначениях, решаемой методом ветвей и границ с выбором наилучшего варианта

который соответствует допустимому решению $\{a \to 2, b \to 1, c \to 3, d \to 4\}$ с общей стоимостью 13. "Сестринский" узел 9 соответствует допустимому решению $\{a \to 2, b \to 1, c \to 4, d \to 3\}$ с общей стоимостью 25. Поскольку эта стоимость больше стоимости решения, соответствующего узлу 8, работа с узлом 9 просто прекращается (если бы его стоимость была меньше 13, пришлось бы заменить информацию о найденном наилучшем решении данными, соответствующими этому листу).

Теперь, если мы обратимся к каждому из пяти листьев последнего дерева пространства состояний (узлы 1, 3, 4, 6 и 7 на рис. 11.7), то обнаружим, что их значения нижних границ не меньше значения целевой функции наилучшего решения, обнаруженного к настоящему времени (равного 13, в листе 8). Следовательно, работа с ними прекращается, и решение, представленное листом 8, является оптимальным решением поставленной задачи.

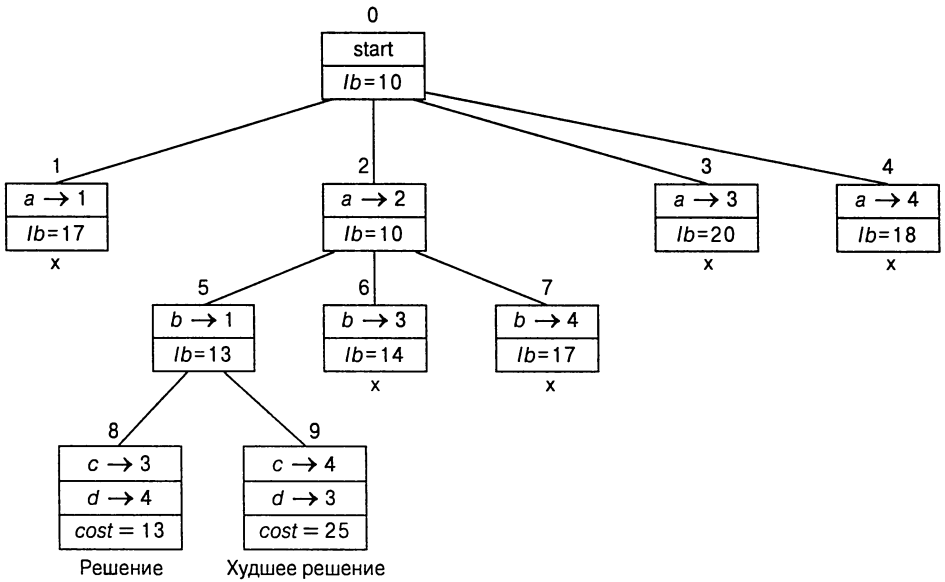


Рис. 11.7. Полное дерево пространства состояний для экземпляра задачи о назначениях, решаемой методом ветвей и границ с выбором наилучшего варианта

Перед тем как распрощаться с задачей о назначениях, напомним самим себе еще раз, что, в отличие от следующего примера, для данной задачи имеется полиномиальный алгоритм решения под названием Венгерский метод (см., например, [87]). В свете наличия такого эффективного алгоритма решение проблемы о назначениях методом ветвей и границ следует рассматривать как учебный пример, а не как практическую рекомендацию.

Задача о рюкзаке

Давайте применим метод ветвей и границ к решению задачи о рюкзаке. С этой задачей мы также познакомились в разделе 3.4: дано n предметов с весами $w_1, \ldots, w_n$ и ценами $v_1, \ldots, v_n$, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке и которое имеет при этом максимальную цену. Оказывается удобным упорядочить предметы в убывающем порядке по их удельной цене (отношению цены к весу), с разрешением неоднозначностей произвольным образом:

$$v_1/w_1 \geqslant v_2/w_2 \geqslant \cdots \geqslant v_n/w_n.$$

Естественной структурой дерева пространства состояний для данной задачи является бинарное дерево, построенное следующим образом (рис. 11.8). Каждый узел на уровне $0 \leqslant i \leqslant n$ представляет все подмножества из n элементов, которые включают определенный выбор из первых i упорядоченных элементов. Этот

частичный выбор однозначно определяется путем от корня к узлу: ветвь, идущая влево, указывает на включение очередного элемента в подмножество, в то время как правая ветвь указывает на отсутствие элемента в подмножестве. Мы записываем общий вес w и общую стоимость v выбора, соответствующего узлу, вместе с верхней границей ub значения для любого подмножества, которое может быть получено путем добавления некоторых элементов (возможно, никаких) к этому выбору.

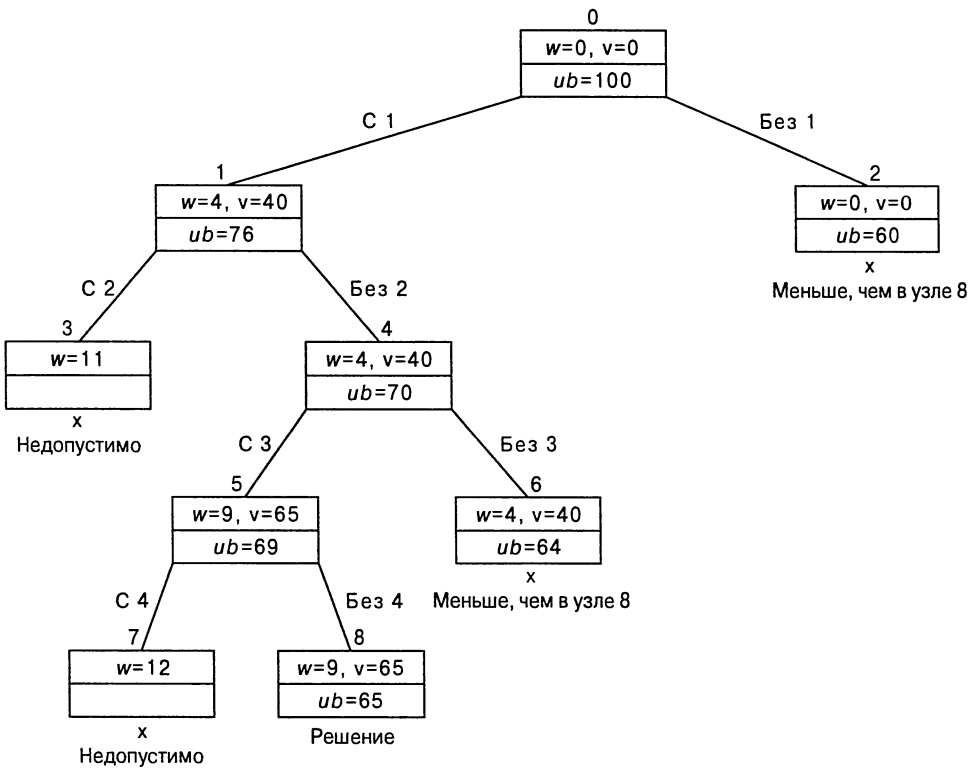


Рис. 11.8. Дерево пространства состояний алгоритма ветвей и границ для экземпляра задачи о рюкзаке

Простым способом вычисления верхней границы ub является добавление к общей стоимости уже выбранных элементов v произведения оставшейся емкости рюкзака $W - w$ и наибольшего значения удельной стоимости среди оставшихся элементов, которое равно v_{i+1}/w_{i+1}:

$$ub = v + (W - w)\,(v_{i+1}/w_{i+1})\,. \tag{11.1}$$

В качестве конкретного примера применим метод ветвей и границ к тому же экземпляру задачи о рюкзаке, который мы решали в разделе 3.4 методом исчер-

пывающего перебора (здесь мы переупорядочили элементы в порядке убывания их удельных стоимостей). Емкость рюкзака $W = 10$.

Предмет	Вес	Стоимость	Удельная стоимость
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

В корне дерева пространства состояний не выбран ни один элемент. Следовательно, как общий вес, так и общая стоимость выбранных элементов равны 0. Значение верхней границы, вычисленное по формуле (11.1), равно 100. Узел 1, левый дочерний узел корня, представляет подмножество, состоящее из одного предмета, 1; общий вес и стоимость в этом узле равны, соответственно, 4 и 40, а значение верхней границы — $40 + (10 - 4) \cdot 6 = 76$. Узел 2 представляет подмножество, которое не включает предмет 1, так что в этом узле $w = 0$, $v = 0$ и $ub = 0 + (10 - 0) \cdot 6 = 60$. Поскольку узел 1 имеет бóльшую верхнюю границу, чем узел 2, он является более обещающим для данной задачи максимизации, и мы начинаем ветвление с узла 1. Его дочерние узлы — 3 и 4 — представляют подмножества с элементом 1 и с и без элемента 2, соответственно. Поскольку общий вес любого подмножества, представленного узлом 3, превосходит емкость рюкзака, работа с этим узлом завершается. У узла 4 те же значения общего веса и общей стоимости, что и у родительского, так что значение верхней границы у этого узла $ub = 40 + (10 - 4) \cdot 5 = 70$. Из узлов 2 и 4 для дальнейшего ветвления мы выбираем узел 4 (почему?) и получаем узлы 5 и 6, включающий и не включающий, соответственно, предмет 3. Общие вес и стоимость и значение верхней границы для этих узлов вычисляются точно так же, как и ранее. Ветвление из узла 5 дает узел 7, который дает недопустимое решение, и узел 8, представляющий подмножество $\{1, 3\}$. (Поскольку никаких дополнительных предметов нет, верхняя граница для узла 8 просто равна сумме стоимостей указанных предметов.) Оставшиеся живые узлы 2 и 6 имеют меньшие значения верхней границы, чем решение, представленное узлом 8. Следовательно, работа с этими узлами завершается, и множество $\{1, 3\}$ из узла 8 является оптимальным решением задачи.

Решение задачи о рюкзаке методом ветвей и границ имеет весьма необычные характеристики. Обычно внутренние узлы дерева пространства состояний не определяют точку пространства поиска задачи, поскольку некоторые из компонентов решения остаются неопределенными (см., например, дерево для задачи о назначениях, рассматривавшейся в предыдущем подразделе). В задаче же о рюкзаке каждый узел дерева представляет подмножество данных предметов. Этот факт можно использовать для обновления информации о наилучшем подмножестве после генерации каждого нового узла дерева. Для рассмотренного экземпляра это означает, что мы могли бы прекратить работу с узлами 2 и 6 еще до генерации уз-

ла 8, так как значения верхних границ рассматриваемых узлов меньше стоимости подмножества в узле 5, равной 65.

Задача коммивояжера

Мы сможем применить метод ветвей и границ к экземпляру задачи коммивояжера, если найдем подходящий метод оценки нижней границы длины маршрута. Одна очень простая нижняя граница может быть получена путем поиска наименьшего элемента в матрице расстояний между городами и умножении его на количество городов n. Однако имеется менее очевидная и более информативная нижняя граница, не требующая большого количества вычислений. Нетрудно показать (упражнение 11.2.8), что можно вычислить нижнюю границу длины l любого маршрута следующим образом. Для каждого i-го города ($1 \leqslant i \leqslant n$) находим сумму s_i расстояний от города i до двух ближайших городов, после чего вычисляем сумму n этих чисел и делим результат на 2. Если все расстояния — целые числа, округляем результат до ближайшего целого:

$$lb = \lceil s/2 \rceil . \tag{11.2}$$

Например, для экземпляра задачи, показанного на рис. 11.9*а*, формула (11.2) дает

$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$

Кроме того, для любого подмножества маршрутов, которое должно включать некоторые ребра данного графа, мы можем соответственно изменить нижнюю границу (11.2). Например, для всех гамильтоновых циклов графа на рис. 11.9*а*, которые должны включать ребро (a, d), мы получим следующую нижнюю границу путем суммирования длин двух кратчайших ребер, инцидентных каждой из вершин (с необходимым включением ребер (a, d) и (d, a)):

$$\lceil [(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2 \rceil = 16.$$

Теперь применим алгоритм ветвей и границ с ограничивающей функцией (11.2) для поиска кратчайшего гамильтонова цикла графа, показанного на рис. 11.9*а*. Чтобы уменьшить количество потенциальной работы, воспользуемся двумя наблюдениями, сделанными в разделе 3.4. Во-первых, без потери общности мы можем рассматривать только маршруты, начинающиеся в a. Во-вторых, поскольку граф неориентированный, мы можем генерировать только маршруты, в которых b посещается перед c. И в дополнение: после посещения $n - 1 = 4$ городов нет выбора, кроме как посетить оставшийся город и вернуться в начало маршрута. Дерево пространства состояний для данного алгоритма, примененного к графу на рис. 11.9*а*, показано на рис. 11.9*б*.

Комментарии, которые были сделаны в конце предыдущего раздела о сильных и слабых сторонах поиска с возвратом, в той же мере применимы и к методу

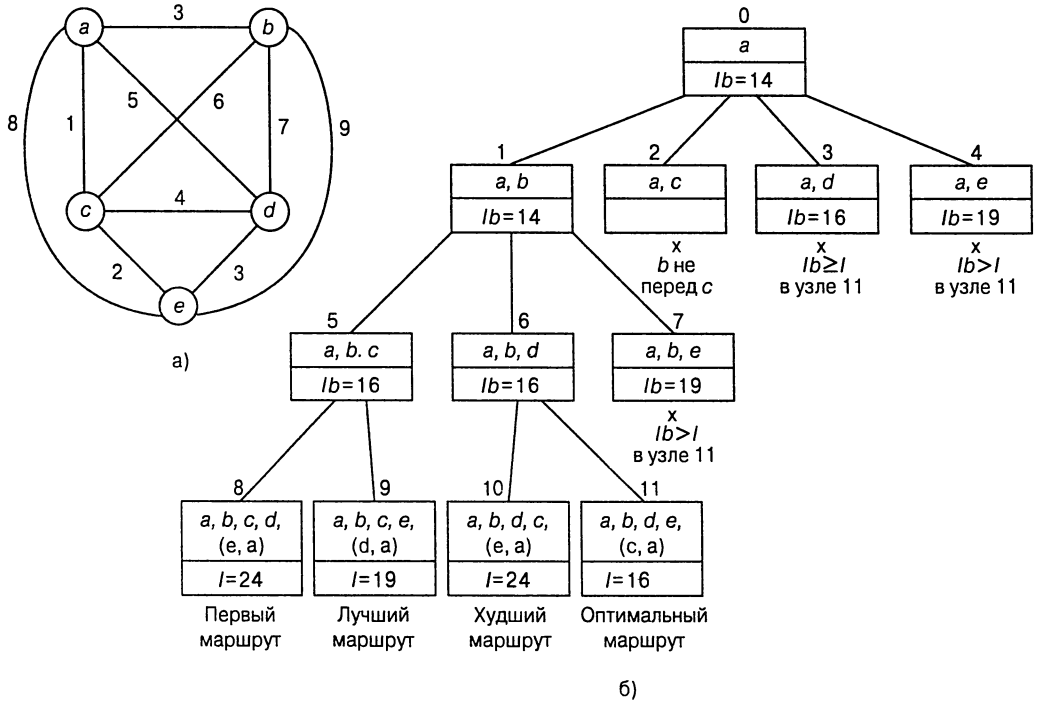


Рис. 11.9. *а*) Взвешенный граф. *б*) Дерево пространства состояний алгоритма ветвей и границ для данного графа. Список вершин в узле указывает начальную часть гамильтонова цикла, представленного узлом

ветвей и границ. Самое главное, — что оба эти метода позволяют решать многие большие экземпляры сложных комбинаторных задач. Однако, как правило, почти невозможно предсказать, какие именно экземпляры окажутся решаемы за реальное время, а какие — нет.

Использование дополнительной информации, такой как симметрия доски в игре, могут расширить диапазон решаемых экземпляров задач. Алгоритм ветвей и границ зачастую может быть ускорен, если знать значение целевой функции для некоторого нетривиального допустимого решения. Такую информацию может оказаться возможным получить до начала построения дерева пространства состояний, скажем, воспользовавшись конкретными данными, а для некоторых задач даже сгенерировав допустимое решение случайным образом. Такое решение тут же может использоваться в качестве наилучшего, полученного к данному моменту, что позволяет не ожидать, пока ветвление приведет к какому-либо допустимому решению.

В противоположность поиску с возвратом решение задачи методом ветвей и границ включает как необходимость выбора порядка генерации узлов, так и поиска хорошей функции для вычисления границ. Хотя правило выбора наилучшего

варианта, использовавшееся в этом разделе, представляет собой достаточно разумный подход, оно может и не приводить к решению быстрее, чем другие стратегии. (Кстати, область кибернетики, посвященная искусственному интеллекту, среди прочего интересуется стратегиями разработки деревьев пространств состояний.)

Поиск хорошей функции для вычисления границ — задача обычно непростая. С одной стороны, требуется, чтобы эта функция была легко вычислимой. С другой, она не может быть слишком упрощенной — в противном случае она не сможет выполнять свою основную задачу отсечения как можно большего количества ветвей дерева пространства состояний. Поиск компромисса между этими конкурирующими требованиями может потребовать интенсивных экспериментов с широким диапазоном экземпляров рассматриваемой задачи.

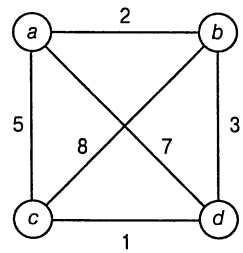
Упражнения 11.2

1. Какую структуру данных вы бы предложили использовать для отслеживания живых узлов в алгоритме ветвей и границ с выбором наилучшего варианта?
2. Решите тот же экземпляр задачи о назначениях, который решен в тексте раздела, при помощи алгоритма ветвей и границ с выбором наилучшего варианта, но с функцией для вычисления границ, использующей не строки, а столбцы матрицы.
3. **а)** Приведите пример входных данных для алгоритма ветвей и границ для задачи о назначениях, соответствующий наилучшему случаю.
   б) Сколько узлов окажется в дереве пространства состояний в наилучшем случае алгоритма ветвей и границ, примененного к задаче о назначениях?
4. Напишите программу для решения задачи о назначениях методом ветвей и границ. Поэкспериментируйте с программой, чтобы определить средний размер матрицы стоимости, для которого задача решается на вашем компьютере за одну минуту.
5. Решите следующую задачу о рюкзаке методом ветвей и границ:

Предмет	Вес	Стоимость
1	10	100
2	7	63
3	8	56
4	4	12

Емкость рюкзака $W = 16$.

6. **а)** Предложите более сложную функцию для вычисления границ в задаче о рюкзаке, чем использованная в тексте раздела.
   б) Воспользуйтесь вашей функцией в алгоритме ветвей и границ, примененном к экземпляру задачи из упражнения 5.
7. Напишите программу для решения задачи о рюкзаке методом ветвей и границ.
8. **а)** Докажите корректность нижней границы, вычисляемой по формуле (11.2) для экземпляров задачи коммивояжера с целой симметричной матрицей расстояний между городами.
   б) Как нужно изменить нижнюю границу (11.2) для несимметричной матрицы расстояний?
9. Примените алгоритм ветвей и границ к решению задачи коммивояжера для приведенного графа (мы решали эту задачу в разделе 3.4 методом исчерпывающего перебора).



10. В качестве исследовательского проекта напишите реферат об использовании деревьев пространств состояний для программирования таких игр, как шахматы, шашки и крестики-нолики. При работе над рефератом вы должны ознакомиться с двумя основными алгоритмами — минимаксным и альфа-бета отсечения.

11.3 Приближенные алгоритмы для NP-сложных задач

В этом разделе мы рассмотрим, как работать со сложными задачами комбинаторной оптимизации, такими как задача коммивояжера или задача о рюкзаке. Как указывалось в разделе 10.3, версии принятия решения этих задач являются NP-полными. Оптимизационные версии таких сложных задач попадают в класс ***NP-сложных*** (NP-hard) — задач, которые имеют сложность как минимум ту же,

что и NP-полные задачи.[2] Следовательно, нет известных алгоритмов с полиномиальным временем работы, которые бы решали указанные задачи, и имеются серьезнейшие теоретические причины, по которым следует ожидать, что таких алгоритмов не существует в принципе. Как же тогда подходить к этим задачам, многие из которых весьма важны с практической точки зрения?

Если экземпляр задачи очень мал, мы можем решить его при помощи исчерпывающего перебора (см. раздел 3.4). Некоторые из таких задач можно решить при помощи динамического программирования, как было показано в разделе 8.4. Но даже когда эти подходы применимы в принципе, на практике они ограничены только относительно малыми экземплярами. Открытие метода ветвей и границ оказалось переломным моментом, поскольку этот метод сделал возможным получить решение для многих больших экземпляров сложных задач комбинаторной оптимизации за приемлемое время. Однако обычно гарантировать такую хорошую производительность невозможно.

Имеется радикально иной подход к сложным оптимизационным задачам — решать их приближенно при помощи быстрого алгоритма. Этот подход в особенности подходит для приложений, где достаточно хорошего, но не обязательно оптимального решения. Кроме того, в реальных приложениях мы зачастую работаем с неточными данными. В этих условиях может оказаться вполне разумным получение приближенного решения.

Хотя диапазон сложности приближенных алгоритмов весьма широк, многие из них представляют собой жадные алгоритмы, основанные на некоторой специфичной для данной предметной области эвристике. ***Эвристика*** (heuristic) — это основанные на здравом смысле правила, вытекающие из опыта, а не из строгих математически доказанных положений. Иллюстрацией этого понятия может служить, например, выбор ближайшего непосещенного города в задаче коммивояжера. Мы рассмотрим алгоритм, основанный на этой эвристике, позже в данном разделе.

Конечно, если мы используем алгоритм, выходом которого является приближение настоящего оптимального решения, надо знать, насколько точным является это приближение. Мы можем измерять точность приближенного решения s_a задачи минимизации некоторой функции f величиной относительной ошибки этого приближения

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)},$$

где s^* — точное решение задачи. В качестве альтернативы, поскольку $re(s_a) = f(s_a)/f(s^*) - 1$, мы можем в качестве меры точности s_a использовать ***отно-***

[2]Понятие NP-сложной задачи может быть более строго определено путем расширения понятия полиномиальной приводимости к задачам, которые не обязательно входят в класс NP, включая задачи оптимизации, рассматривающиеся в этом разделе (см. главу 5 в [40]).

шение точности (accuracy ratio)

$$r(s_a) = \frac{f(s_a)}{f(s^*)}.$$

Заметим, что для единообразия отношение точности приближенного решения задачи максимизации зачастую вычисляется как

$$r(s_a) = \frac{f(s^*)}{f(s_a)},$$

чтобы это отношение, как и для задачи минимизации, было не меньше 1.

Очевидно, что чем ближе значение $r(s_a)$ к 1, тем лучшим является приближение решения. Наилучшая (т.е. наинизшая) верхняя граница возможных значений $r(s_a)$, взятая по всем экземплярам задачи, называется ***коэффициентом производительности*** (performance ratio) алгоритма и обозначается R_A. Коэффициент производительности служит в качестве основной меры, указывающей качество приближенного алгоритма. Желательно, конечно, иметь алгоритм со значением R_A по возможности близким к 1, но, к сожалению, как мы увидим, некоторые простые алгоритмы имеют неограниченно большие коэффициенты производительности $R_A = \infty$). Это не означает, что такие алгоритмы нельзя использовать; надо просто быть осторожными с результатами их работы.

Мы говорим, что приближенный алгоритм с полиномиальным временем работы является ***c-приближенным алгоритмом*** (c-approximation algorithm), если его коэффициент производительности не превышает c, т.е. для любого экземпляра рассматриваемой задачи

$$f(s_a) \leqslant cf(s^*).$$

Еще одним важным фактом, о котором надо помнить при решении сложных задач комбинаторной оптимизации, заключается в следующем. Хотя уровень сложности большинства таких задач тот же, что и у полиномиально приводимых к ним задач, эта эквивалентность не распространяется на приближенные алгоритмы. Как мы увидим, поиск приближенного решения с разумной степенью точности гораздо проще для одних из таких приводимых задач, чем для других.

Приближенный алгоритм для решения задачи коммивояжера

В разделе 3.4 задачу коммивояжера мы решали путем исчерпывающего перебора. В разделе 10.3 мы выяснили, что это одна из NP-полных задач, а в разделе 11.2 увидели, как ее экземпляры могут быть решены при помощи метода ветвей и границ. Здесь мы рассмотрим два простых приближенных алгоритма для решения этой задачи.

Алгоритм ближайшего соседа

Этот простой жадный алгоритм основан на эвристике ***ближайшего соседа***: всегда идти в ближайший непосещенный город.

Шаг 1. Выбрать произвольный город в качестве начального.

Шаг 2. Повторять следующую операцию до тех пор, пока не будут посещены все города: идти в непосещенный город, ближайший к последнему посещенному (неоднозначности разрешаются произвольным образом).

Шаг 3. Вернуться в начальный город.

Пример 1. Для экземпляра задачи, представленного на рис. 11.10, с a в качестве начальной вершины, алгоритм ближайшего соседа приводит к следующему маршруту (гамильтонову циклу): $s_a : a - b - c - d - a$ длиной 10.

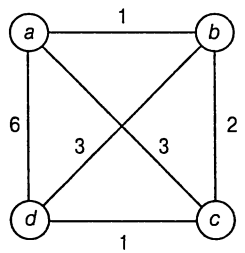


Рис. 11.10. Экземпляр задачи коммивояжера для иллюстрации алгоритма ближайшего соседа

Оптимальным решением, как легко проверить исчерпывающим перебором, является маршрут $s^* : a - b - d - c - a$ длиной 8. Таким образом, отношение точности этого приближения равно

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(т.е. маршрут s_a на 25% длиннее оптимального маршрута s^*). ■

К сожалению, кроме простоты, алгоритм ближайшего соседа ничем хорошим не отличается. В частности, в общем случае нельзя ничего сказать о точности получаемых им решений, так как он может заставить нас идти по очень длинному ребру на последнем этапе пути. В самом деле, если в примере 1 мы изменим вес ребра (a, d) с 6 на произвольное сколь угодно большое число $w \geqslant 6$, то алгоритм все равно будет давать в качестве решения маршрут $a - b - c - d - a$, длина которого равна $4 + w$, в то время как оптимальный маршрут $a - b - d - c - a$ будет иметь длину 8. Следовательно,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8},$$

и мы можем сделать это отношение произвольно большим, выбирая подходящее значение w. Следовательно, для этого алгоритма $R_A = \infty$.

Это, конечно, неприятное известие. Как мы увидим в конце этого подраздела, это связано не с простотой алгоритма ближайшего соседа, а со сложностью приближенного решения задачи коммивояжера. Однако имеется важное подмножество экземпляров этой задачи, называющихся ***евклидовыми***, для которых мы можем сделать нетривиальное заключение о точности алгоритма ближайшего соседа. Это экземпляры, в которых расстояния между городами удовлетворяют следующим естественным условиям:

- *неравенство треугольника*

$$d[i,j] \leqslant d[i,k] + d[k,j] \quad \text{для любой тройки городов } i,\ j \text{ и } k$$

(непосредственное расстояние между городами i и j не может превосходить длину пути из города i в город j через некоторый промежуточный город k);

- *симметрия*

$$d[i,j] = d[j,i] \quad \text{для любой пары городов } i \text{ и } j$$

(расстояние от i до j равно расстоянию от j до i).

Можно доказать (см., например, [95]), что хотя коэффициент производительности алгоритма ближайшего соседа остается неограниченным для евклидовых экземпляров, отношение точности для любого такого экземпляра с $n \geqslant 2$ городами удовлетворяет неравенству

$$\frac{f(s_a)}{f(s^*)} \leqslant \frac{1}{2}\left(\lceil \log_2 n \rceil + 1\right),$$

где $f(s_a)$ и $f(s^*)$ — соответственно, длина маршрута, полученного при помощи алгоритма ближайшего соседа, и длина кратчайшего маршрута.

Алгоритм двойного обхода дерева

Теперь давайте рассмотрим простой приближенный алгоритм с конечным отношением производительности для евклидовых экземпляров задачи коммивояжера. Этот алгоритм использует связь между гамильтоновым циклом и остовными деревьями одного и того же графа.

Шаг 1. Построить минимальное остовное дерево графа, соответствующего данному экземпляру задачи коммивояжера.

Шаг 2. Начиная с произвольной вершины, обойти минимальное остовное дерево, записывая пройденные вершины.

Шаг 3. Просканировать список, полученный на шаге 2, и убрать из него все повторяющиеся вершины, за исключением вершины в конце списка. Вершины, остающиеся в списке, образуют гамильтонов цикл, который и является выходом данного алгоритма.

Пример 2. Применим этот алгоритм к графу, показанному на рис. 11.11*а*. Минимальное остовное дерево этого графа состоит из ребер (a, b), (b, c), (b, d) и (d, e) (см. рис. 11.11*б*). Двойной обход минимального остовного дерева , начатый в вершине f, дает список вершин a, b, c, b, d, e, d, b, a.

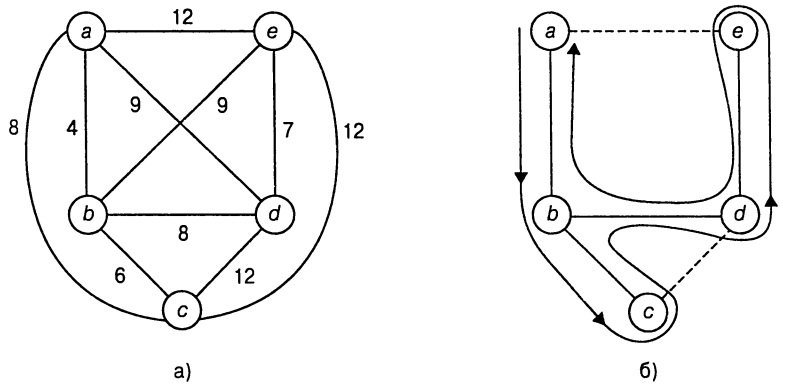


Рис. 11.11. Иллюстрация работы алгоритма двойного обхода дерева. *а*) Граф. *б*) Обход минимального остовного дерева с удалением повторяющихся вершин

Удаляя из списка вторую вершину b, вторую вершину d и третью вершину b, мы получим гамильтонов цикл a, b, c, d, e, a длиной 41. ■

Маршрут, полученный в примере 2, не оптимальный. Хотя рассмотренный экземпляр достаточно мал, чтобы можно было найти решение исчерпывающим перебором или методом ветвей и границ, мы воздержимся от этого для большей обобщенности рассмотрения. Как правило, мы не знаем, какова на самом деле длина оптимального маршрута, а потому не можем вычислить и отношение точности $f(s_a)/f(s^*)$. Однако в случае алгоритма двойного обхода дерева мы можем как минимум оценить его сверху, если рассматриваемый граф — евклидов.

Теорема 1. Алгоритм двойного обхода дерева является 2-приближенным алгоритмом решения задачи коммивояжера с евклидовыми расстояниями.

Доказательство. Очевидно, что алгоритм двойного обхода дерева полиномиален, если воспользоваться на первом шаге эффективным алгоритмом поиска минимального остовного дерева, таким как алгоритм Прима или Крускала. Нам

надо показать, что для любого евклидового экземпляра задачи коммивояжера длина маршрута s_a, полученного при помощи алгоритма двойного обхода дерева, не более чем в два раза превосходит длину оптимального маршрута s^*, т.е.

$$f(s_a) \leqslant 2f(s^*).$$

Поскольку удаление любого ребра из s^* дает остовное дерево T с весом $w(T)$, который должен быть не меньше веса минимального остовного дерева графа $w(T^*)$, получаем неравенство

$$f(s^*) > w(T) \geqslant w(T^*).$$

Из этого неравенства следует, что

$$2f(s^*) > 2w(T^*) = \text{длина пути, полученного на шаге 2 алгоритма.}$$

Все возможные сокращения пути на шаге 3 алгоритма, приводящие к получению маршрута s_a, не могут увеличить общую длину обхода евклидова графа, т.е.

$$\text{Длина пути, полученного на шаге 2} \geqslant \text{Длина маршрута } s_a.$$

Объединяя два последние неравенства, получим неравенство

$$2f(s^*) > f(s_a),$$

которое на самом деле несколько более строгое чем то, которое нам требовалось доказать. ■

Для евклидовой задачи коммивояжера имеются приближенные алгоритмы с лучшим коэффициентом производительности. Например, ***алгоритм Кристофидеса*** (Christofides' algorithm), который также основан на применении минимальных остовных деревьев, но более интеллектуальным и сложным способом, чем алгоритм двойного обхода дерева, имеет коэффициент производительности 1.5 [40].

Можно ли надеяться найти полиномиальный приближенный алгоритм для задачи коммивояжера с конечным коэффициентом производительности для *всех* экземпляров задачи? Как показывает следующая теорема [100], ответ отрицателен, если только не выполняется соотношение $P = NP$.

Теорема 2. Если $P \neq NP$, то не существует c-приближенного алгоритма для задачи коммивояжера, т.е. не существует приближенного алгоритма с полиномиальным временем работы для решения этой задачи такого, что для всех экземпляров

$$f(s_a) \leqslant cf(s^*)$$

для некоторой константы c.

Доказательство. Будем вести доказательство от противного. Предположим, что такой приближенный алгоритм A и константа c существуют (без потери общности можно считать, что c — натуральное число). Покажем, что такой алгоритм можно использовать для решения задачи о гамильтоновом цикле за полиномиальное время. Мы воспользуемся вариацией преобразования, использованного в разделе 10.3 для приведения задачи о гамильтоновом цикле к задаче коммивояжера. Пусть G — произвольный граф с n вершинами. Отобразим граф G на полный взвешенный граф G', назначая вес 1 для каждого из его ребер и добавляя ребра весом $cn + 1$ между каждой парой вершин, не смежных в G. Если G содержит гамильтонов цикл, то его длина в G' равна n; следовательно, этот цикл является точным решением s^* задачи коммивояжера для графа G'. Заметим, что если s_a — приближенное решение, полученное для графа G' алгоритмом A, то $f(s_a) \leqslant cn$ по предположению. Если G не имеет гамильтонова цикла, то кратчайший маршрут в G' содержит как минимум одно ребро весом $cn + 1$, а следовательно, $f(s_a) \geqslant f(s^*) > cn$. С учетом двух полученных неравенств мы можем решить задачу о гамильтоновом цикле для графа G за полиномиальное время, отображая граф G на G', применяя алгоритм A для получения кратчайшего маршрута в G' и сравнивая его длину с cn. Поскольку задача о гамильтоновом цикле NP-полная, мы получаем противоречие, если только не выполняется соотношение $P = NP$. ■

Приближенные алгоритмы для задачи о рюкзаке

Еще одна широко известная NP-сложная задача — задача о рюкзаке, с которой мы познакомились в разделе 3.4. Дано n предметов с весами $w_1, \ldots, w_n$ и ценами $v_1, \ldots, v_n$, а также рюкзак, выдерживающий вес W. Наша задача — найти подмножество предметов, которое можно разместить в рюкзаке и которое имеет при этом максимальную цену. Мы видели, как можно решить эту задачу методом исчерпывающего перебора (раздел 3.4), динамического программирования (раздел 8.4) и ветвей и границ (раздел 11.2). Теперь мы будем решать эту задачу при помощи приближенных алгоритмов.

Жадные алгоритмы для задачи о рюкзаке

Можно рассмотреть несколько жадных подходов к данной задаче. Один из них состоит в выборе предметов в убывающем порядке по их весам; беда в том, что более тяжелые предметы могут не быть наиболее ценными в множестве. Другой вариант состоит в выборе предметов в порядке уменьшения их стоимости, однако он не гарантирует эффективное использование емкости рюкзака. Можно ли найти жадную стратегию, которая бы принимала во внимание как вес, так и стоимость предметов? Да, можно: вычисляя удельную стоимость предметов v_i/w_i, $i = 1, 2, \ldots, n$ и выбирая предметы в порядке уменьшения удельной стоимости (в действительности мы уже использовали этот подход при разработке алгоритма

ветвей и границ для задачи о рюкзаке в разделе 11.2). Вот как выглядит алгоритм, основанный на этой жадной эвристике.

Шаг 1. Вычислим удельные стоимости всех предметов множества $r_i = v_i/w_i$, $i = 1, 2, \ldots, n$.

Шаг 2. Отсортируем предметы в невозрастающем порядке по их удельным стоимостям, вычисленным на шаге 1 (неоднозначности разрешаются произвольным образом).

Шаг 3. До тех пор пока в отсортированном списке не останется ни одного предмета, повторяем следующие действия: если текущий предмет помещается в рюкзак, мы помещаем его туда; в противном случае переходим к следующему предмету.

Пример 3. Давайте рассмотрим экземпляр задачи о рюкзаке емкостью 10 и следующей информацией о предметах:

Предмет	Вес	Стоимость
1	7	42
2	3	12
3	4	40
4	5	25

Вычислим удельные стоимости и отсортируем предметы в невозрастающем порядке их удельных стоимостей, что даст нам следующую таблицу:

Предмет	Вес	Стоимость	Стоимость / Вес
3	4	40	10
1	7	42	6
4	5	25	5
2	3	12	4

Жадный алгоритм выбирает предмет 3 с весом 4, пропускает предмет 1 с весом 7, затем выбирает предмет 4 с весом 5 и пропускает предмет 2 с весом 3. Полученное решение оказывается оптимальным для данного экземпляра задачи (см. раздел 11.2, где этот же экземпляр задачи решен методом ветвей и границ). ■

Может быть, жадный алгоритм всегда приводит к оптимальному решению? Конечно же, нет — если бы это было так, мы бы имели полиномиальный алгоритм для NP-сложной задачи. Приведенный далее пример показывает, что при помощи этого алгоритма нельзя получить гарантированную конечную верхнюю границу точности.

Пример 4. Емкость рюкзака $W > 2$.

Предмет	Вес	Стоимость	Стоимость / Вес
1	1	2	2
2	W	W	1

Поскольку предметы уже расположены в требуемом порядке, алгоритм выбирает первый из них и пропускает второй; общая стоимость подмножества равна при этом 2. Оптимальным же является выбор второго предмета стоимостью W. Следовательно, отношение точности $r(s_a)$ этого приближенного решения равно $W/2$ — величине, не ограниченной сверху. ■

Этот алгоритм очень легко модифицировать, получив приближенный алгоритм с конечным коэффициентом производительности. Все, что для этого надо, — выбирать лучшее из двух решений: одно из них получается при помощи жадного алгоритма, а второе — из одного предмета наибольшей стоимости, который может поместиться в рюкзаке (заметим, что второй вариант в последнем примере оказывается лучше первого). Нетрудно доказать, что коэффициент производительности такого ***усовершенствованного жадного алгоритма*** равен 2. Таким образом, стоимость оптимального подмножества s^* не более чем в два раза больше стоимости подмножества s_a, полученного при помощи усовершенствованного жадного алгоритма, причем 2 — наименьший множитель, для которого можно сформулировать такое утверждение.

Поучительно также рассмотреть непрерывную версию задачи о рюкзаке. В этой версии мы можем класть в рюкзак произвольные части предметов. В таком случае представляется естественной следующая модификация жадного алгоритма.

Шаг 1. Вычислим удельные стоимости всех предметов множества предметов $r_i = v_i/w_i$, $i = 1, 2, \ldots, n$.

Шаг 2. Отсортируем предметы в невозрастающем порядке по их удельным стоимостям, вычисленным на шаге 1 (неоднозначности разрешаются произвольным образом).

Шаг 3. До тех пор пока рюкзак не будет полностью заполнен или в отсортированном списке не останется ни одного предмета, повторяем следующие действия: если текущий предмет полностью помещается в рюкзак, мы берем его и переходим к следующему предмету; в противном случае мы берем только ту часть, которая может поместиться в рюкзаке (до конца заполнив его), и на этом завершаем работу.

Например, для использованного в примере 3 экземпляра задачи с четырьмя предметами данный алгоритм выбирает предмет 3 с весом 4 и $6/7$ предмета 1 весом 7, после чего рюкзак оказывается заполненным.

Вас не должно удивлять, что этот алгоритм всегда приводит к оптимальному решению непрерывной версии задачи о рюкзаке. В самом деле, все предметы отсортированы по их эффективности использования емкости рюкзака. Если первый предмет в отсортированном списке имеет вес w_1 и стоимость v_1, то ни одно решение не может использовать w_1 единиц емкости рюкзака с большей стоимостью, чем v_1. Если нельзя заполнить рюкзак первым предметом или его частью полностью, мы должны добавить максимально возможное количество второго предмета, и т.д. По сути, это набросок строгого доказательства корректности описанного алгоритма, которое остается читателю в качестве упражнения.

Заметим также, что стоимость оптимального решения непрерывной версии задачи о рюкзаке может служить верхней границей для дискретной версии того же экземпляра задачи. Это наблюдение обеспечивает лучший способ вычисления верхней границы при решении дискретной задачи о рюкзаке методом ветвей и границ, чем тот, который использовался в разделе 11.2.

Схемы приближений

Вернемся к дискретной задаче о рюкзаке. Для этой задачи, в отличие от задачи коммивояжера, существуют полиномиальные ***схемы приближений*** (approximation schemes), которые представляют собой параметрические семейства алгоритмов, позволяющие получить приближения $s_a^{(k)}$ с предопределенным уровнем точности:

$$\frac{f\left(s_a^{(k)}\right)}{f\left(s^*\right)} \leqslant 1 + \frac{1}{k} \quad \text{для любого экземпляра размера } n,$$

где k — целый параметр из диапазона $0 \leqslant k < n$. Первая схема приближений была предложена С. Сахни (S. Sahni) в 1975 году [99]. Этот алгоритм генерирует все подмножества из k или меньшего количества предметов и для каждого из подмножеств, которое помещается в рюкзак, добавляет оставшиеся предметы так же, как это делает описанный ранее жадный алгоритм (т.е. в невозрастающем порядке их удельных стоимостей). Подмножество с наибольшей суммарной стоимостью, полученное таким методом, и является выходом данного алгоритма.

Пример 5. Небольшой пример схемы приближения с $k = 2$ показан на рис. 11.12. В качестве оптимального решения данного экземпляра задачи алгоритм возвращает подмножество $\{1, 3, 4\}$. ■

Вряд ли на вас произвел большое впечатление приведенный пример, и это понятно. Важность этой схемы носит в основном теоретический, а не практический характер. Она заключается в том факте, что в дополнение к приближению оптимального решения с заданным уровнем точности временна́я эффективность данного алгоритма полиномиально зависит от n. Действительно, общее количество подмножеств, генерируемых перед добавлением дополнительных элементов,

Предмет	Вес	Стоимость	Стоимость / Вес
1	4	40	10
2	7	42	6
3	5	25	5
4	1	4	4

Емкость $W = 10$

а)

Подмножество	Добавляемые предметы	Стоимость
$\varnothing$	1,3,4	69
$\{1\}$	3,4	69
$\{2\}$	4	46
$\{3\}$	1,4	69
$\{4\}$	1,3	69
$\{1,2\}$	Недопустимо	
$\{1,3\}$	4	69
$\{1,4\}$	3	69
$\{2,3\}$	Недопустимо	
$\{2,4\}$		46
$\{3,4\}$	1	69

б)

Рис. 11.12. Пример применения схемы приближения Сахни при $k = 2$. *а*) Экземпляр задачи. *б*) Подмножества, генерируемые алгоритмом

равно

$$\sum_{j=0}^{k} C_n^j = \sum_{j=0}^{k} \frac{n\,(n-1)\cdots(n-j+1)}{j!} \leqslant \sum_{j=0}^{k} n^j \leqslant \sum_{j=0}^{k} n^k = (k+1)\,n^k.$$

Для каждого из этих подмножеств требуется время $O(n)$, чтобы найти его возможное расширение. Таким образом, эффективность данного алгоритма — $O\left(kn^{k+1}\right)$. Обратите внимание: в то время как эффективность схемы Сахни полиномиально зависит от n, от k она зависит экспоненциально. Более сложные схемы приближений, называющиеся ***полностью полиномиальными схемами*** (fully polynomial schemes), лишены этого недостатка. Среди ряда книг, в которых рассматриваются эти алгоритмы, хочется выделить монографию С. Мартелло (S. Martello)

и П. Тота (P. Toth) [77], которая содержит богатый материал, посвященный задаче о рюкзаке.

Упражнения 11.3

1. **а)** Примените алгоритм ближайшего соседа к экземпляру задачи, определенному приведенной ниже матрицей расстояний. Начните маршрут с города 1, считая, что города пронумерованы от 1 до 5.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & \infty \\ 14 & 0 & 5 & 8 & 7 \\ 4 & 5 & 0 & 9 & 16 \\ 10 & 8 & 9 & 0 & 32 \\ \infty & 7 & 16 & 32 & 0 \end{bmatrix}$$

   б) Вычислите отношение точности этого приближенного решения.

2. **а)** Напишите псевдокод алгоритма ближайшего соседа. Считайте, что входными данными алгоритма является матрица расстояний размером $n \times n$.

   б) Какова временна́я эффективность алгоритма ближайшего соседа?

3. Примените алгоритм двойного обхода дерева к графу на рис. 11.11*а* с обходом минимального остовного дерева, который начинается в вершине a, но отличается от обхода, показанного на рис. 11.11*б*. Равна ли длина полученного маршрута длине маршрута, показанного на рис. 11.11*б*?

4. Докажите, что удаление повторяющихся вершин, используемое алгоритмом двойного обхода дерева, не может привести к увеличению длины маршрута в евклидовом графе.

5. К какому классу временно́й эффективности принадлежит жадный алгоритм для решения задачи о рюкзаке?

6. Докажите, что коэффициент производительности R_A усовершенствованного жадного алгоритма для задачи о рюкзаке равен 2.

7. Рассмотрим жадный алгоритм для задачи об упаковке корзин, который называется ***алгоритмом первого подходящего*** (first-fit algorithm): помещаем каждый предмет в порядке поступления в первую корзину,

в которую он может поместиться; если таких корзин нет, — помещаем его в новую корзину и добавляем ее в список корзин.

а) Примените этот алгоритм к экземпляру $s_1 = 0.4$, $s_2 = 0.7$, $s_3 = 0.2$, $s_4 = 0.1$, $s_5 = 0.5$ и определите, является ли полученное решение оптимальным.

б) Определите эффективность этого алгоритма в наихудшем случае.

в) Докажите, что описанный алгоритм является 2-приближенным алгоритмом.

8. Приближенный ***алгоритм первого подходящего с убыванием*** (first-fit decreasing algorithm) для задачи об упаковке корзин начинается с сортировки предметов в невозрастающем порядке по их размерам, после чего работает так же, как и алгоритм первого подходящего.

а) Примените этот алгоритм к экземпляру $s_1 = 0.4$, $s_2 = 0.7$, $s_3 = 0.2$, $s_4 = 0.1$, $s_5 = 0.5$ и определите, является ли полученное решение оптимальным.

б) Всегда ли описанный алгоритм дает оптимальное решение? Обоснуйте ваш ответ.

в) Докажите, что описанный алгоритм является 1.5-приближенным алгоритмом.

г) Проведите эксперимент по выяснению того, какой из алгоритмов — первого подходящего или первого подходящего с убыванием — дает более точное приближение для случайно выбранного экземпляра задачи.

9. а) Разработайте простой 2-приближенный алгоритм для поиска ***минимального вершинного покрытия*** (minimal vertex cover) — вершинного покрытия с минимальным количеством вершин — для данного графа.

б) Рассмотрим следующий приближенный алгоритм для поиска ***максимального независимого множества*** (maximal independent set) — независимого множества с наибольшим количеством вершин — для данного графа: применим 2-приближенный алгоритм из части *а* упражнения и выведем все вершины, не входящие в полученное вершинное покрытие. Можно ли утверждать, что этот алгоритм также является 2-приближенным алгоритмом?

10. а) Разработайте жадный алгоритм с полиномиальным временем работы для задачи о раскраске графа.

б) Покажите, что коэффициент производительности вашего приближенного алгоритма неограниченно велик.

11.4 Алгоритмы для решения нелинейных уравнений

В этом разделе мы рассмотрим несколько алгоритмов для решения нелинейных уравнений с одним неизвестным

$$f(x) = 0. \tag{11.3}$$

Имеется несколько причин такого выбора среди прочих областей численного анализа. Во-первых, это очень важная задача как с практической, так и с теоретической точки зрения. Она возникает в качестве математической модели множества явлений в науке и технике, как непосредственно, так и опосредованно (вспомним, например, что стандартный метод поиска точек экстремума функции $f(x)$ основан на поиске критических точек, являющихся корнями уравнения $f'(x) = 0$). Во-вторых, она представляет наиболее доступный раздел численного анализа и в то же время демонстрирует его типичные инструменты и понятия. В-третьих, некоторые методы решения уравнений близки алгоритмам поиска в массиве, а следовательно, являются примером применения общих методов проектирования алгоритмов к задачам непрерывной математики.

Начнем с часто встречающегося заблуждения, имеющего корни в курсе математики средней школы и заключающегося в уверенности, что любое уравнение можно решить "разложением" или по готовой формуле. Но дело в том, что все встречающиеся в школе уравнения решаются таким образом только потому, что они представляют собой результат тщательного отбора среди всех уравнений таких, которые можно решить указанными способами. В общем случае мы не в состоянии точно решить уравнение, и требуется алгоритм для его приближенного решения.

Это справедливо даже для решения квадратного уравнения

$$ax^2 + bx + c = 0,$$

поскольку стандартная формула для его корней

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

требует вычисления квадратного корня, что для большинства положительных чисел может быть сделано только приближенно. Кроме того, как говорилось в разделе 10.4, эта каноническая формула должна быть модифицирована для того, чтобы избежать возможной потери точности.

А что можно сказать о формулах для корней полиномов более высоких степеней? Формулы для корней полиномов третьей и четвертой степени существуют,

но они слишком громоздки, чтобы иметь практическое значение. Для полиномов более высоких степеней не может существовать формул, которые бы включали только коэффициенты полиномов, арифметические операции и радикалы. Этот замечательный результат был впервые опубликован в 1799 году итальянским математиком и физиком Паоло Руффини (Paolo Ruffini) (1765–1822) и вновь открыт четверть века спустя норвежским математиком Нильсом Абелем (Niels Abel) (1802–1829) и получил дальнейшее развитие в работах французского математика Эвариста Галуа (Evariste Galois) (1811–1832)[3].

Невозможность такой формулы вряд ли можно рассматривать как большую неприятность. Как отметил в своей диссертации в 1801 году великий немецкий математик Карл Фридрих Гаусс (Carl Friedrich Gauss) (1777–1855), алгебраическое решение уравнения ничуть не лучше разработки символа для корня такого уравнения и утверждения, что уравнение имеет корень, равный этому символу [84].

Мы можем интерпретировать решение уравнения (11.3) как точку, в которой график функции $f(x)$ пересекается с осью абсцисс x. В этом разделе мы рассмотрим три алгоритма, использующие данную интерпретацию. Конечно, график $f(x)$ может пересекать ось абсцисс в одной точке (например, $x^3 = 0$), многих или даже в бесконечном количестве точек (например, $\sin x = 0$) или не пересекать ее вообще ($e^x + 1 = 0$). Соответственно, уравнение (11.3) будет иметь один корень, несколько корней или не иметь корней вовсе. Неплохо перед тем как начинать поиск корней функции, набросать ее график. Это может помочь определить количество корней и их приближенное местоположение. В общем случае неплохо выделить корни, т.е. определить интервалы, содержащие по одному корню рассматриваемого уравнения.

Метод деления пополам

Этот алгоритм основан на том наблюдении, что график непрерывной функции должен пересечь ось абсцисс между двумя точками a и b как минимум один раз, если значения функции имеют в этих точках разные знаки (рис. 11.13).

Корректность этого наблюдения доказывается в качестве теоремы в курсе вычислительной математики, так что здесь мы просто воспользуемся этим результатом. На нем основан следующий алгоритм решения уравнения (11.3), называющийся ***методом деления пополам*** (bisection method). Начиная с отрезка $[a, b]$, на концах которого $f(x)$ имеет разные знаки, алгоритм вычисляет среднюю точку $x_{mid} = (a + b)/2$. Если $f(x_{mid}) = 0$, корень найден, и алгоритм завершает работу. В противном случае он продолжает поиск корня либо на отрезке $[a, x_{mid}]$, либо

[3]Открытие Руффини было полностью проигнорировано всеми ведущими математиками того времени. Абель умер молодым, прожив трудную жизнь в нищете. Галуа был убит на дуэли, когда ему был всего лишь 21 год. Их результаты по решению уравнений высших степеней сейчас рассматриваются как одни из высочайших достижений математики.

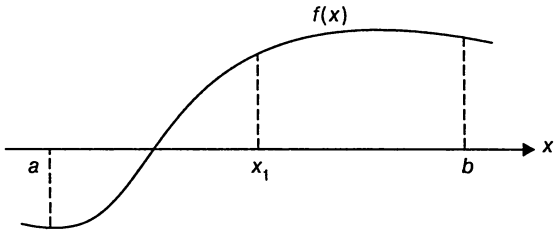


Рис. 11.13. Первая итерация метода деления пополам: x_1 — средняя точка отрезка $[a, b]$

на отрезке $[x_{mid}, b]$ — в зависимости от того, на какой из двух половин отрезка функция $f(x)$ имеет разные знаки на концах нового отрезка.

Поскольку мы не можем ожидать, что алгоритм "наткнется" на точное решение уравнения и завершит работу, нужен иной критерий для его завершения. Можно остановиться, когда отрезок $[a_n, b_n]$, содержащий некоторый корень x^*, станет столь мал, что можно гарантировать, что абсолютная ошибка приближения x^* величиной x_n (средней точкой отрезка) меньше некоторого заранее выбранного значения $\varepsilon > 0$. Поскольку x_n — средняя точка отрезка $[a_n, b_n]$ и x^* лежит внутри этого отрезка, мы имеем

$$|x_n - x^*| \leqslant \frac{b_n - a_n}{2}. \tag{11.4}$$

Следовательно, мы можем завершать работу алгоритма, как только $(b_n - a_n)/2 < \varepsilon$, или, что то же самое,

$$x_n - a_n < \varepsilon. \tag{11.5}$$

Нетрудно доказать, что

$$|x_n - x^*| \leqslant \frac{b_1 - a_1}{2^n} \text{ для } n = 1, 2, \ldots. \tag{11.6}$$

Из этого неравенства следует, что последовательность приближений $\{x_n\}$ может быть сделана сколь угодно близкой к x^* выбором достаточно большого значения n. Другими словами, мы можем сказать, что $\{x_n\}$ ***сходится*** к корню x^*. Заметим, однако, что, поскольку все цифровые компьютеры представляют малые значения нулем (см. раздел 10.4), утверждение о сходимости верно в теории, но не всегда верно на практике. В действительности, если мы выберем значение ε меньшим, чем определенное пороговое для данной машины значение, алгоритм никогда не завершится! Еще одним источником потенциальных сложностей являются ошибки округления при вычислении значений рассматриваемой функции. Таким образом, желательно включать в программу, реализующую метод деления пополам, ограничение на количество итераций, которое может выполнить алгоритм.

Вот псевдокод метода деления пополам.

Алгоритм $Bisection\,(f\,(x)\,, a, b, eps, N)$

```
// Реализует метод деления пополам для поиска корня
// уравнения f(x) = 0
// Входные данные:  Два действительных числа a и b, a < b,
//                  непрерывная на [a, b] функция f(x),
//                  f(a) * f(b) < 0,
//                  верхняя граница абсолютной ошибки eps > 0,
//                  верхняя граница количества итераций N
// Выходные данные: Приближенное (или точное) значение x
//                  корня уравнения на отрезке (a, b) или
//                  отрезок, содержащий корень (если
//                  достигнут предел количества итераций)
n ← 1  // Количество итераций
while n ⩽ N do
  x ← (a + b)/2
  if x − a < eps
    return x
  fval ← f(x)
  if fval = 0
    return x
  if fval * f(a) < 0
    b ← x
  else
    a ← x
  n ← n + 1
return "Предел количества итераций", a, b
```

С помощью неравенства (11.6) можно заранее определить количество итераций, достаточное (по крайней мере теоретически) для достижения требуемой степени точности. Выбрав n достаточно большим, чтобы выполнялось неравенство $(b_1 - a_1)/2^n < \varepsilon$, т.е.

$$n > \log_2 \frac{b_1 - a_1}{\varepsilon}, \tag{11.7}$$

мы получим достаточное для обеспечения заданной точности количество итераций.

Пример 1. Рассмотрим уравнение

$$x^3 - x - 1 = 0. \tag{11.8}$$

Оно имеет один действительный корень (см. график функции $f\,(x) = x^3 - x - 1$ на рис. 11.14). Поскольку $f\,(0) < 0$ и $f\,(2) > 0$, корень должен находиться

в интервале $(0, 2)$. Выбрав допустимую ошибку $\varepsilon = 10^{-2}$, из неравенства (11.7) мы получим, что потребуется $n > \log_2 (2/10^{-2})$, или $n \geqslant 8$ итераций.

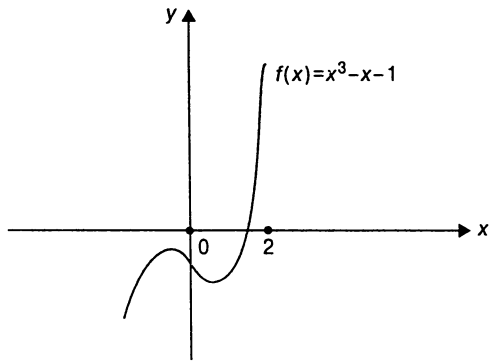


Рис. 11.14. График функции $f(x) = x^3 - x - 1$

В таблице на рис. 11.15 показаны результаты пошагового выполнения алгоритма деления пополам для решения уравнения (11.8). Так мы получаем $x_8 = 1.3203125$ в качестве приближенного значения корня x^* уравнения (11.8), гарантируя при этом, что

$$|1.3203125 - x^*| < 10^{-2}.$$

Кроме того, приняв во внимание знаки левой части уравнения (11.8) в точках a_8, b_8 и x_8, мы можем утверждать, что корень лежит между 1.3203125 и 1.328125. ■

n	a_n	b_n	x_n	$f(x_n)$
1	0.0−	2.0+	1.0	−1.0
2	1.0−	2.0+	1.5	0.875
3	1.0−	1.5+	1.25	−0.296875
4	1.25−	1.5+	1.375	0.224609
5	1.25−	1.375+	1.3125	−0.051514
6	1.3125−	1.375+	1.34375	0.082611
7	1.3125−	1.34375+	1.328125	0.014576
8	1.3125−	1.328125+	1.3203125	−0.018711

Рис. 11.15. Пошаговое выполнение алгоритма деления пополам для решения уравнения (11.8). Знаки после чисел во втором и третьем столбцах указывают знак $f(x) = x^3 - x - 1$ в соответствующих конечных точках отрезка

Основной недостаток метода деления пополам в качестве алгоритма общего назначения для решения уравнений заключается в его низкой скорости сходимости по сравнению с другими известными методами. Именно по этой причине описанный метод редко используется на практике. Кроме того, его нельзя распространить на уравнения более общего вида и на системы уравнений. Однако у этого метода имеются и сильные стороны. Он всегда сходится к корню, каким бы ни был начальный отрезок (свойства которого легко проверяются). Этот метод не использует производную функции $f(x)$, как это делают некоторые более быстрые методы.

Какой важный алгоритм напоминает вам метод деления пополам? Если вы сочтете, что он очень похож на бинарный поиск, то окажетесь правы. Оба эти алгоритма решают варианты задачи поиска, и оба являются алгоритмами уменьшения размера задачи в 2 раза. Основное отличие между ними — область применения: дискретная у алгоритма бинарного поиска и непрерывная у алгоритма деления пополам. Заметим также, что в то время как алгоритм бинарного поиска требует, чтобы массив был отсортирован, алгоритм деления пополам не накладывает на функцию условия невозрастания или неубывания. Наконец, в то время как бинарный поиск очень быстр, метод деления пополам достаточно медленный.

Метод секущих

Метод секущих[4] (method of false position), известный также по его названию на латыни — ***regula falsi***, соотносится с интерполяционным поиском так же, как метод половинного деления — с бинарным поиском. Как и алгоритм деления пополам, на каждой итерации он получает некоторый отрезок $[a_n, b_n]$, содержащий корень непрерывной функции $f(x)$, которая имеет значения противоположных знаков в точках a_n и b_n. В отличие от метода деления пополам очередное приближение вычисляется не как средина отрезка $[a_n, b_n]$, а как точка пересечения оси абсцисс с прямой линией, проведенной через точки $(a_n, f(a_n))$ и $(b_n, f(b_n))$ (рис. 11.16).

Убедитесь самостоятельно, что формула для точки пересечения выглядит следующим образом:

$$x_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}. \tag{11.9}$$

Пример 2. Таблица на рис. 11.17 содержит результаты первых восьми итераций этого метода при решении уравнения (11.8).

Хотя для данного примера метод секущих не работает так же хорошо, как метод деления пополам, для многих экземпляров он дает более быстро сходящуюся последовательность. ■

[4] В отечественной литературе встречаются и другие названия метода, такие как ***метод ложного положения***, ***метод хорд***. — *Прим. ред.*

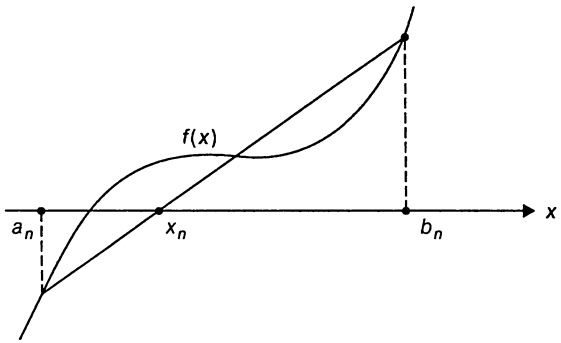


Рис. 11.16. Итерация метода секущих

n	a_n	b_n	x_n	$f(x_n)$
1	0.0−	2.0+	0.333333	−1.296296
2	0.333333−	2.0+	0.676471	−1.366909
3	0.676471−	2.0+	0.960619	−1.074171
4	0.960619−	2.0+	1.144425	−0.645561
5	1.144425−	2.0+	1.242259	−0.325196
6	1.242259−	2.0+	1.288532	−0.149163
7	1.288532−	2.0+	1.309142	−0.065464
8	1.309142−	2.0+	1.318071	−0.028173

Рис. 11.17. Пошаговое выполнение метода секущих для решения уравнения (11.8). Знаки после чисел во втором и третьем столбцах указывают знак $f(x) = x^3 - x - 1$ в соответствующих конечных точках отрезка

Метод Ньютона

Метод Ньютона (Newton's method), именуемый также методом Ньютона–Рафсона (Newton–Raphson), представляет собой один из наиболее важных алгоритмов общего назначения для решения уравнений. Его применение к решению уравнения (11.3) с одним неизвестным проиллюстрировано на рис. 11.18: очередной элемент x_{n+1} последовательности приближений получается как пересечение касательной к графику функции $f(x)$ в точке x_n с осью абсцисс.

Аналитическая формула для элементов последовательности приближений метода Ньютона выглядит следующим образом:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \text{ для } n = 0, 1, \ldots. \tag{11.10}$$

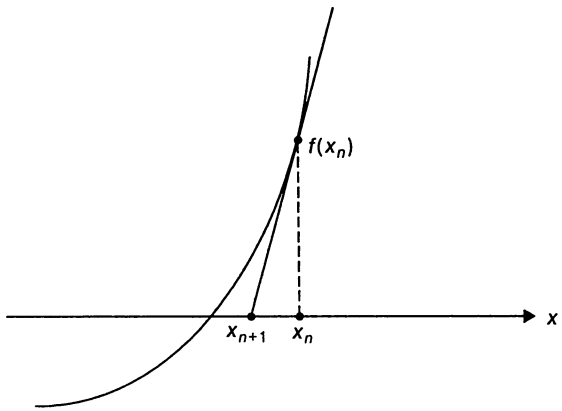


Рис. 11.18. Итерация метода Ньютона

В большинстве случаев метод Ньютона гарантирует сходимость последовательности (11.10), если начальное приближение x_0 выбрано "достаточно близко" к корню (точное определение требований к выбору x_0 можно найти в учебниках по численным методам). Последовательность может сходиться и при далеком от корня начальном приближении, но это выполняется не всегда.

Пример 3. Вычисление $\sqrt{a}$ можно выполнить путем поиска неотрицательного корня уравнения $x^2 - a = 0$. Если мы воспользуемся формулой (11.10) для $f(x) = x^2 - a$ и $f'(x) = 2x$, то получим формулу

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x_n^2 + a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right),$$

совпадающую с формулой приближенного вычисления квадратного корня из раздела 10.4. ■

Пример 4. Давайте применим метод Ньютона к уравнению (11.8), которое мы уже решали методом деления пополам и методом секущих. Формула (11.10) в этом случае превращается в

$$x_{n+1} = x_n - \frac{x_n^3 - x_n - 1}{3x_n^2 - 1}.$$

В качестве начального приближения выберем, скажем, точку $x_0 = 2$. На рис. 11.19 показаны результаты первых пяти итераций метода Ньютона. ■

Обратите внимание, насколько последовательность приближений, полученная методом Ньютона, быстрее сходится к точному решению по сравнению с методом половинного деления и методом секущих. Такая очень быстрая сходимость типична для метода Ньютона, если начальное приближение оказывается близким

n	x_n	x_{n+1}	$f(x_{n+1})$
0	2.0	1.545455	1.145755
1	1.545455	1.359615	0.153705
2	1.359615	1.325801	0.004625
3	1.325801	1.324719	$4.7 \cdot 10^{-6}$
4	1.324719	1.324718	$5.0 \cdot 10^{-12}$

Рис. 11.19. Итерации метода Ньютона при решении уравнения (11.8)

к корню уравнения. Заметим, однако, что на каждой итерации этого метода мы должны вычислять новое значение функции и ее производной, в то время как предыдущие методы требовали вычисления только самóй функции. Кроме того, метод Ньютона не указывает точно границы, в которых находится корень уравнения, как это делают рассмотренные ранее методы. Фактически при произвольно выбранных функции и начальном приближении последовательность приближений может оказаться расходящейся. И, поскольку в формуле (11.10) в знаменателе находится производная функции, метод может оказаться неработоспособным, если она окажется равной нулю. Метод Ньютона наиболее эффективен, когда $f'(x)$ ограничена ненулевым значением вблизи корня x^*. В частности, если

$$|f'(x)| \geqslant m_1 > 0$$

на отрезке между x_n и x^*, мы можем оценить расстояние от x_n до x^* при помощи теоремы о среднем значении следующим образом:

$$f(x_n) - f(x^*) = f'(c)(x_n - x^*),$$

где c — некоторая точка между x_n и x^*. Поскольку $f(x^*) = 0$, а $|f'(c)| \geqslant m_1$, мы получим

$$|x_n - x^*| \leqslant \frac{|f(x_n)|}{m_1}. \tag{11.11}$$

Формулу (11.11) можно использовать в качестве критерия для завершения работы метода Ньютона, когда ее правая часть станет меньше заранее выбранного уровня точности ε. Другими возможными критериями могут быть

$$|x_n - x^*| < \varepsilon$$

и

$$|f(x_n) < \varepsilon|,$$

где ε — малое положительное число. Поскольку два последних критерия не обязательно означают x_n к корню x^*, они должны рассматриваться как худшие по сравнению с (11.11).

Недостатки метода Ньютона не должны скрывать его главного преимущества: быстрая сходимость для произвольного начального приближения и применимость для большого количества типов уравнений и систем уравнений.

Упражнения 11.4

1. а) Найдите в Internet или в вашей библиотеке процедуру для поиска действительного корня кубического уравнения общего вида $ax^3 + bx^2 + cx + d = 0$ с действительными коэффициентами.

б) На каком методе проектирования алгоритмов основана эта процедура?

2. Сколько корней имеют следующие уравнения:

а) $xe^x - 1 = 0$ **б)** $x - \ln x = 0$ **в)** $x \sin x - 1 = 0$

3. а) Докажите, что если $p(x)$ — полином нечетной степени, то он должен иметь как минимум один действительный корень.

б) Докажите, что если x_0 — корень полинома $p(x)$ n-ой степени, то этот полином может быть разложен на множители следующим образом:

$$p(x) = (x - x_0)\, q(x),$$

где $q(x)$ — полином степени $n-1$. Поясните важность этой теоремы для поиска корней полиномов.

в) Докажите, что если x_0 — корень полинома $p(x)$ n-ой степени, то

$$p'(x_0) = q(x_0),$$

где $q(x)$ — частное от деления $p(x)$ на $x - x_0$.

4. Докажите неравенство (11.6).

5. Примените метод деления пополам для поиска корня уравнения

$$x^3 + x - 1 = 0$$

с абсолютной ошибкой меньшей 10^{-2}.

6. Выведите формулу (11.9), лежащую в основе метода секущих.

7. Примените метод секущих для поиска корня уравнения

$$x^3 + x - 1 = 0$$

с абсолютной ошибкой меньшей 10^{-2}.

8. Выведите формулу (11.10), лежащую в основе метода Ньютона.

9. Примените метод Ньютона для поиска корня уравнения

$$x^3 + x - 1 = 0$$

с абсолютной ошибкой меньшей 10^{-2}.

10. Приведите пример, показывающий, что последовательность приближений метода Ньютона может быть расходящейся.

Резюме

- *Поиск с возвратом и метод ветвей и границ* представляют собой два метода разработки алгоритмов для решения задач, в которых количество вариантов выбора растет как минимум экспоненциально с ростом размера экземпляра задачи. В обоих методах решение строится по одному компоненту, с попытками завершить построение как только становится понятным, что невозможно получить решение с уже выбранными компонентами. Такой подход делает возможным решение многих больших экземпляров NP-сложных задач за приемлемое время.
- Как поиск с возвратом, так и метод ветвей и границ используют в качестве основного механизма *дерево пространства состояний* — корневое дерево, узлы которого представляют частично сконструированные решения рассматриваемой задачи. Оба метода завершают работу с узлом, как только становится возможным гарантировать, что решение невозможно получить при рассмотрении решений, являющихся потомками данного узла.
- *Поиск с возвратом* строит дерево пространства состояний в глубину в большинстве случаев применения данного алгоритма. Если последовательность выбора, представленная текущим узлом дерева пространства состояний, может продолжаться без нарушения ограничений задачи, рассмотрение продолжается с первым разрешенным вариантом очередного компонента. В противном случае метод возвращается к последнему компоненту частично построенного решения и заменяет его очередным вариантом.
- *Метод ветвей и границ* представляет собой метод проектирования алгоритмов, который усовершенствует генерацию дерева пространства состояний при помощи оценки наилучшего значения, которое можно получить из данного узла дерева. Если такая оценка не превосходит наилучшее решение, найденное к данному моменту, узел удаляется из дальнейшего рассмотрения.

- Зачастую для поиска приближенных решений задач комбинаторной оптимизации используются приближенные алгоритмы. *Коэффициент производительности* представляет собой основную меру точности таких приближенных алгоритмов.
- Алгоритм *ближайшего соседа* является простым жадным алгоритмом для решения задачи коммивояжера. Коэффициент производительности этого алгоритма не ограничен сверху, даже для важного подмножества *евклидовых графов*.
- Алгоритм *двойного обхода дерева* является приближенным алгоритмом для решения задачи коммивояжера с коэффициентом производительности, равным 2 для евклидовых графов. Алгоритм основан на внесении изменений в путь обхода минимального остовного дерева.
- Практичный жадный алгоритм для решения задачи о рюкзаке основан на работе с предметами в порядке убывания их удельных стоимостей. Для непрерывной версии задачи алгоритм дает точное оптимальное решение.
- *Полиномиальные схемы приближений* для задачи о рюкзаке представляют собой параметрические алгоритмы с полиномиальным временем работы и произвольным предопределенным уровнем точности.
- Решение нелинейных уравнений является одной из важных областей численного анализа. При отсутствии формул для корней нелинейных уравнений они, за небольшим исключением, могут быть решены приближенно рядом алгоритмов.
- *Метод деления пополам* и *метод секущих* представляют собой непрерывный аналог бинарного и интерполяционного поисков, соответственно. Их главное преимущество в том, что на каждой итерации алгоритма они указывают отрезок, в котором находится корень уравнения.
- *Метод Ньютона* генерирует последовательность приближений корня, представляющих собой точки пересечения касательных к графику функции с осью абсцисс. При хорошем начальном приближении этот метод обычно требует только нескольких итераций для получения приближенного значения корня с высокой точностью.

Эпилог

Наука — не что иное, как обученный и организованный здравый смысл.

— Томас Хаксли (Thomas H. Huxley) (1825–1895),
английский биолог и педагог

Итак, мы добрались до конца. Это была долгая дорога. Конечно, не такая долгая, как дорога человечества от алгоритма Евклида до последних разработок в области алгоритмов, но все равно достаточно долгая. Давайте оглянемся и бросим взгляд на все, что вы узнали во время нашего путешествия.

Мы начали с того, что понятие алгоритма является краеугольным камнем информатики, а поскольку компьютерные программы всего лишь реализуют те или иные алгоритмы, последние являются также и основой практического программирования.

Как и любая другая наука, информатика классифицирует объекты своего изучения. Хотя алгоритмы можно классифицировать множеством разных способов, особенно важны два из них. Можно классифицировать алгоритмы по лежащему в их основе методу проектирования и по их эффективности.

В этой книге мы рассмотрели девять основных методов проектирования:

грубая сила
декомпозиция
уменьшение размера задачи
преобразование
пространственно-временной компромисс
жадные методы
динамическое программирование
поиск с возвратом
метод ветвей и границ

Мы показали, как эти методы применяются к различным задачам, таким как сортировка, поиск, работа со строками, графами, некоторые геометрические и численные задачи. Хотя все эти методы неприменимы к каждой из задач, взятые вместе, они образуют отличный инструментарий для разработки новых алгоритмов и классификации старых. Кроме того, эти методы могут рассматриваться и как

обобщенные способы решения задач, не ограниченных только компьютерной тематикой. Головоломки, приведенные в книге, ясно доказывают это.

Анализ времени работы алгоритмов классифицирует их по порядку роста времени работы как функции от размера входных данных. Это делается путем выяснения количества выполнений базовой операции алгоритма. Основным инструментарием здесь являются формулы суммирования и рекуррентные соотношения, соответственно, для нерекурсивных и рекурсивных алгоритмов. Мы видели, что удивительно большое количество алгоритмов попадает в один из нескольких классов из следующего списка:

Класс	Обозначение	Важные примеры
Постоянное время	$\Theta(1)$	Хеширование (в среднем)
Логарифмическое	$\Theta(\log n)$	Бинарный поиск (в среднем и наихудшем случаях)
Линейное	$\Theta(n)$	Последовательный поиск (в среднем и наихудшем случаях)
$n \log n$	$\Theta(n \log n)$	Эффективные алгоритмы сортировки
Квадратичное	$\Theta(n^2)$	Элементарные алгоритмы сортировки
Кубическое	$\Theta(n^3)$	Метод исключения Гаусса
Экспоненциальное	$\Omega(2^n)$	Комбинаторные задачи

Для некоторых алгоритмов мы должны различать эффективности в наихудшем, наилучшем и среднем случае. Средний случай наиболее сложен для анализа, так что мы рассматривали только, как это можно сделать эмпирически.

Мы коснулись и ограничений алгоритмов. Мы видели, что имеется две основных причины для ограничений: внутренняя сложность задачи и необходимость работы с округленными числами в большинстве численных задач. Мы также рассмотрели способы преодоления этих ограничений.

Естественно, вас не должно удивлять наличие областей алгоритмики, никак не затронутых в нашей книге. Наиболее важные из них — рандомизированные и параллельные алгоритмы. Рандомизированный алгоритм — это алгоритм, который в процессе работы делает случайные выборы. Например, мы можем случайным образом выбирать опорный элемент в алгоритме быстрой сортировки. В отличие от детерминистических алгоритмов рандомизированные алгоритмы ведут себя поразному при разных запусках даже для одних и тех же входных данных и даже могут приводить к разным результатам. Для многих приложений рандомизированные алгоритмы оказываются быстрее и/или проще, чем их детерминистические двойники.

Вероятно, наиболее впечатляющими рандомизированными алгоритмами, открытыми до сегодняшнего дня, являются рандомизированные алгоритмы проверки простоты чисел, такие как алгоритм Миллера–Рабина (см., например, [54]).

Этот рандомизированный алгоритм за приемлемое время решает задачу проверки простоты тысячезначных чисел с вероятностью ошибки, не превышающей вероятности ошибки аппаратного обеспечения. В настоящее время эффективные детерминистические алгоритмы для решения этой задачи неизвестны, что является ключевым моментом современной криптологии. Если вы хотите побольше узнать о рандомизированных алгоритмах, можете обратиться к монографии Р. Мотвани (R. Motwani) и П. Рагавана (P. Raghavan) [81] и превосходному обзору Р. Карпа (R. M. Karp) [60].

Подавляющее большинство современных компьютеров очень схожи с машиной, описанной более полувека назад фон Нейманом (John von Neumann). Основным положением в этой архитектуре является последовательное выполнение команд. Соответственно, алгоритмы, разработанные для выполнения на таких машинах, называются *последовательными*. Именно эти алгоритмы и рассматриваются в данной книге. Однако это центральное положение фон Неймана не выполняется для тех современных компьютеров, которые могут выполнять команды одновременно, т.е. параллельно. Алгоритмы, использующие такие возможности компьютеров, называются *параллельными*.

Рассмотрим, например, задачу вычисления суммы n чисел, хранящихся, скажем, в массиве $A[0..n-1]$. Можно доказать, что любой последовательный алгоритм, который использует только умножение, сложение и вычитание, требует как минимум $n-1$ шагов для решения этой задачи. Однако если разбить числа попарно и найти суммы элементов $A[0]$ и $A[1]$, $A[2]$ и $A[3]$ и т.д. параллельно, размер задачи уменьшится вдвое. Повторяя эту операцию до тех пор, пока не будет вычислена вся сумма, мы получим алгоритм, требующий только $\lceil \log_2 n \rceil$ шагов.

Имеется богатый выбор книг, посвященных параллельным алгоритмам. Многие учебники по алгоритмам включают отдельные главы, посвященные параллельным алгоритмам (например, [54]) или рассматривают их вместе с последовательными алгоритмами (например, [17, 78]).

Безудержное развитие технологий дает многообещающие результаты, такие, например, как квантовые вычислители, которые могут привести к значительным изменениям вычислительных возможностей и алгоритмов в будущем. В квантовых компьютерах (см. [83]) планируется использовать возможность пребывания атома в двух разных квантовых состояниях, т.е. теоретически система из n таких атомов (называемая кюбитом (qubit)) может хранить 2^n битов информации. В 1994 году Питер Шор (Peter Shor) из AT&T Research Labs представил алгоритм разложения целых чисел, который использует эту теоретическую возможность [109]. Более того, исследователи из IBM смогли построить семикюбитовый компьютер, реализующий алгоритм Шора и успешно разложивший число 15 на множители 3 и 5. Хотя технологические проблемы масштабирования этой и подобных задач и перехода к задачам реальных размеров могут оказаться непреодолимыми,

квантовые компьютеры потенциально в состоянии изменить наши представления о трудных задачах.

Если квантовые компьютеры пытаются воспользоваться для решения сложных вычислительных задач мощью квантовой физики, то ДНК-компьютеры пытаются сделать то же с использованием механизма генного отбора. Наиболее известный пример такого подхода относится к тому же 1994 году, когда кибернетик из США Лен Адлеман (Len Adleman) [2, 8], известный участием в разработке алгоритма шифрования RSA, показал, каким образом задача поиска гамильтонова цикла в ориентированном графе может быть принципиально решена путем генерации цепочек ДНК, представляющих пути в графе, и отбрасыванием тех из них, которые не удовлетворяют определению такого пути. Задача о существовании гамильтонова пути — NP-полная, и подход Адлемана подобен исчерпывающему перебору. Однако одновременное выполнение большого количества параллельных биохимических процессов оставляет надежду получить решение за приемлемое время. Адлеман решил задачу о гамильтоновом пути для небольшого графа из семи вершин, хотя был вынужден повторять части процедуры по несколько раз. Масштабирование подхода Адлемана для больших графов требует экспоненциально растущего количества нуклеотидов для выполнения данной процедуры. Таким образом, потенциал ДНК-вычислений остается пока что неясным, хотя нельзя отрицать его интеллектуальную привлекательность.

Итак, какое бы направление вы не выбрали для будущих путешествий по стране алгоритмов в учебе и профессиональной деятельности, предстоящая дорога столь же волнующа и привлекательна, как и уже пройденная. Вы только в начале большого пути!

Приложение A

Формулы, использующиеся при анализе алгоритмов

Это приложение содержит список полезных формул и правил, которые могут пригодиться при математическом анализе алгоритмов. Более полный материал можно найти в [45, 46, 93] и [105].

Свойства логарифмов

В приведенных ниже формулах основание алгоритмов считается величиной, большей 1; его конкретная величина значения не имеет. Запись $\lg x$ в данной книге означает логарифм по основанию 2, $\ln x$ — логарифм по основанию $e = 2.718281828\ldots$; x и y — произвольные положительные числа.

1. $\log x^y = y \log x$
2. $\log xy = \log x + \log y$
3. $\log \dfrac{x}{y} = \log x - \log y$
4. $\log_a x = \log_a b \ \log_b x$
5. $a^{\log_b x} = x^{\log_b a}$

Комбинаторика

1. Количество перестановок n-элементного множества: $P(n) = n!$
2. Количество k-комбинаций n-элементного множества (сочетаний из по k элементов из n): $C_n^k = \dfrac{n!}{k!\,(n-k)!}$
3. Количество подмножеств n-элементного множества: 2^n

Важные формулы суммирования

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ раз}} = u-l+1$ (l, u — целые пределы суммирования; $l \leqslant u$); $\sum_{i=1}^{n} 1 = n$
2. $\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
3. $\sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$
4. $\sum_{i=1}^{n} i^3 = 1^3+2^3+\cdots+n^3 = \frac{n^2(n+1)^2}{4} \approx \frac{1}{4}n^4$
5. $\sum_{i=1}^{n} i^k = 1^k+2^k+\cdots+n^k \approx \frac{1}{k+1}n^{k+1}$
6. $\sum_{i=0}^{n} a^i = 1+a+\cdots+a^n = \frac{a^{n+1}-1}{a-1} (a \neq 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$
7. $\sum_{i=1}^{n} i2^i = 1\cdot 2+2\cdot 2^2+\cdots+n\cdot 2^n = (n-1)2^{n+1}+2$
8. $\sum_{i=1}^{n} \frac{1}{i} = 1+\frac{1}{2}+\cdots+\frac{1}{n} \approx \ln n+\gamma$, где $\gamma \approx 0.577216\ldots$ (эта сумма называется n-ым гармоническим числом и обозначается H_n; константа γ называется константой Эйлера)
9. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Правила работы с суммами

1. $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\sum_{i=l}^{u} (ca_i + b_i) = c\sum_{i=l}^{u} a_i + \sum_{i=l}^{u} b_i$

4. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, где $l \leqslant m < u$
5. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Приближение суммы определенным интегралом

1. $\int_{l-1}^{u} f(x)\,dx \leqslant \sum_{i=l}^{u} f(i) \leqslant \int_{l}^{u+1} f(x)\,dx$ для неубывающей функции $f(x)$
2. $\int_{l}^{u+1} f(x)\,dx \leqslant \sum_{i=l}^{u} f(i) \leqslant \int_{l-1}^{u} f(x)\,dx$ для невозрастающей функции $f(x)$

Формулы для округлений снизу и сверху

Округление действительного числа x снизу (функция "пол"), обозначаемое как $\lfloor x \rfloor$, определяется как наибольшее целое, не превосходящее x (например, $\lfloor 3.8 \rfloor = 3$, $\lfloor -3.8 \rfloor = -4$, $\lfloor 3 \rfloor = 3$). Округление действительного числа x сверху (функция "потолок"), обозначаемое как $\lceil x \rceil$, определяется как наименьшее целое, не меньшее x (например, $\lceil 3.8 \rceil = 4$, $\lceil -3.8 \rceil = -3$, $\lceil 3 \rceil = 3$).

1. $x - 1 \leqslant \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1$
2. $\lfloor x + n \rfloor = \lfloor x \rfloor + n$ и $\lceil x + n \rceil = \lceil x \rceil + n$ для действительного x и целого n
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$
4. $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$

Разное

1. $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n}$, где $n \geqslant 1$ и $\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$, т.е. $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ при $n \to \infty$ (формула Стирлинга)
2. Арифметика по модулю (n, m — целые, p — положительное целое)

$$(n + m) \bmod p = (n \bmod p + m \bmod p) \bmod p$$
$$(n \cdot m) \bmod p = ((n \bmod p) \cdot (m \bmod p)) \bmod p$$

Приложение Б

Краткое руководство по рекуррентным соотношениям

Последовательности и рекуррентные соотношения

Определение 1. (Числовая) ***последовательность*** (sequence) представляет собой упорядоченный список чисел. ■

Примеры:	2, 4, 6, 8, 10, 12, ...	(положительные четные числа)
	0, 1, 1, 2, 3, 5, 8, ...	(числа Фибоначчи)
	0, 1, 3, 6, 10, 15, ...	(количество сравнений ключей при сортировке выбором)

Последовательность обычно обозначается буквой (например, x или a) с подстрочным индексом (к примеру, n или i), записываемой в фигурных скобках, например $\{x_n\}$. Мы воспользуемся альтернативным обозначением $x(n)$, которое подчеркивает тот факт, что последовательность является функцией: ее аргумент n указывает позицию числа в списке, а значение функции $x(n)$ — само число. $x(n)$ называется ***обобщенным членом*** (generic term) последовательности.

Имеется два основных способа определения последовательности:

- явной формулой, выражающей обобщенный член как функцию от n, например $x(n) = 2n$ для $n \geqslant 0$;
- уравнением, связывающим обобщенный член с одним или несколькими другими членами последовательности, в комбинации с одним или несколькими явными значениями первых членов, например

$$x(n) = x(n-1) + n \text{ для } n > 0 \qquad (Б.1)$$

$$x(0) = 0 \qquad (Б.2)$$

Второй метод особенно важен для анализа рекурсивных алгоритмов (см. раздел 2.4).

Уравнение наподобие (Б.1) называется ***рекуррентным уравнением*** (recurrence equation) или ***рекуррентным соотношением*** (recurrence relation) (или просто ***рекуррентностью*** (recurrence)), а (Б.2) — ***начальным условием*** (initial condition). Начальное значение может быть указано для значения n, отличного от 0, например, для $n = 1$, а для некоторых рекуррентных соотношений (например, для рекуррентного соотношения $F(n) = F(n-1) + F(n-2)$, определяющего числа Фибоначчи — см. раздел 2.5) начальные условия должны определять несколько значений.

Решить рекуррентное соотношение при заданных начальных условиях означает найти явную формулу обобщенного члена последовательности, которая удовлетворяет как рекуррентному соотношению, так и начальным условиям, или доказать, что такая последовательность не существует. Например, решение рекуррентного соотношения (Б.1) при начальных условиях (Б.2) равно

$$x(n) = \frac{n(n+1)}{2} \text{ для } n \geqslant 0. \qquad (Б.3)$$

Непосредственной подстановкой в формулу (Б.1) можно убедиться, что уравнение выполняется при всех $n > 0$, т.е. что

$$\frac{n(n+1)}{2} = \frac{(n-1)(n-1+1)}{2} + n,$$

а подстановкой в (Б.2) — в выполнении начального условия $x(0) = 0$, т.е. что

$$\frac{0(0+1)}{2} = 0.$$

Иногда удобно различать общее и частное решения рекуррентного соотношения. Обычно рекуррентное уравнение имеет бесконечное количество последовательностей, удовлетворяющих ему. ***Общее решение*** (general solution) рекуррентного уравнения представляет собой формулу, определяющую все такие последовательности. Обычно общее решение включает одну или несколько произвольных констант. Например, общее решение рекуррентного уравнения (Б.1) дается формулой

$$x(n) = c + \frac{n(n+1)}{2}, \qquad (Б.4)$$

где c — такая произвольная константа. Присваивая c различные значения, мы можем получить все решения (Б.1), и только их.

Частным решением (particular solution) рекуррентного уравнения является конкретная последовательность, удовлетворяющая данному рекуррентному уравнению. Обычно нас интересует частное решение, удовлетворяющее данному начальному условию. Например, последовательность (Б.3) представляет собой частное решение рекуррентного уравнения (Б.1) при начальном условии (Б.2).

Методы решения рекуррентных соотношений

Не существует универсального метода решения, который позволил бы решать любое рекуррентное соотношение (это не удивительно, поскольку у нас нет такого метода даже для решения существенно более простого уравнения с одним неизвестным $f(x) = 0$ для произвольной функции $f(x)$). Однако имеется ряд методов, которые позволяют решать ряд рекуррентных соотношений.

Метод прямой подстановки

Начиная с начального члена (или членов) последовательности для данных начальных условий, мы можем использовать рекуррентное соотношение для генерации нескольких первых членов его решения в надежде понять, как именно может выглядеть конечная формула. Если такая формула найдена, ее корректность должна быть проверена непосредственной подстановкой в рекуррентное уравнение и в начальные условия (как мы поступали для (Б.1) и (Б.2)) или доказана методом математической индукции.

Например, рассмотрим рекуррентное соотношение

$$x(n) = 2x(n-1) + 1 \text{ для } n > 1 \quad \text{(Б.5)}$$

$$x(1) = 1 \quad \text{(Б.6)}$$

Первые члены мы получим следующим образом:

$$x(1) = 1,$$
$$x(2) = 2x(1) + 1 = 2 \cdot 1 + 1 = 3,$$
$$x(3) = 2x(2) + 1 = 2 \cdot 3 + 1 = 7,$$
$$x(4) = 2x(3) + 1 = 2 \cdot 7 + 1 = 15.$$

Нетрудно заметить, что полученные числа на 1 меньше последовательных степеней 2:

$$x(n) = 2^n - 1 \text{ для } n = 1 \ldots 4.$$

Можно доказать, что эта гипотеза дает обобщенный член решения (Б.5)–(Б.6) либо непосредственной подстановкой формулы в (Б.5) и (Б.6), либо при помощи математической индукции.

С практической точки зрения этот метод работает только для очень ограниченного количества случаев, поскольку обычно очень сложно распознать вид формулы по нескольким первым членам последовательности.

Метод обратной подстановки

Этот метод решения рекуррентных соотношений работает так, как указано в его названии: используя рассматриваемое рекуррентное соотношение, мы выражаем $x(n-1)$ как функцию от $x(n-2)$ и подставляем результат в исходное

уравнение, чтобы получить $x(n)$ как функцию от $x(n-2)$. Повторение этого шага для $x(n-2)$ дает выражение $x(n)$ как функции от $x(n-3)$. Для многих рекуррентных соотношений после этого мы оказываемся в состоянии выявить зависимость и выразить $x(n)$ как функцию от $x(n-i)$ для произвольного $i = 1, 2, \ldots$. Выбирая i так, чтобы $n - i$ достигало начального условия и используя одну их формул суммирования, зачастую удается получить явную формулу для решения рекуррентного соотношения.

В качестве примера применим этот метод к рекуррентному соотношению (Б.1)–(Б.2). Итак, у нас имеется рекуррентное соотношение

$$x(n) = x(n-1) + n.$$

Заменяя в уравнении n на $n-1$, мы получим $x(n-1) = x(n-2) + n - 1$. После подстановки этого выражения для $x(n-1)$ в исходное уравнение мы получим

$$x(n) = (x(n-2) + n - 1) + n = x(n-2) + (n-1) + n.$$

Замена n на $n-2$ дает $x(n-2) = x(n-3) + n - 2$; после подстановки этого выражения для $x(n-2)$ мы получим

$$x(n) = (x(n-3) + n - 2) + (n-1) + n = x(n-3) + (n-2) + (n-1) + n.$$

Сравнивая три формулы для $x(n)$, мы видим, что общий вид формулы после i подстановок имеет вид[1]

$$x(n) = x(n-i) + (n-i+1) + (n-i+2) + \cdots + n.$$

Поскольку начальное условие (Б.2) указано для $n = 0$, для его достижения требуется, чтобы выполнялось соотношение $n - i = 0$, т.е. $i = n$:

$$x(n) = x(0) + 1 + 2 + \cdots + n = 0 + 1 + 2 + \cdots + n = n(n+1)/2.$$

Метод обратной подстановки на удивление хорошо работает для большого количества простых рекуррентных уравнений. Вы можете найти немало примеров его успешного применения в данной книге (см., в частности, раздел 2.4 и упражнения к нему).

Линейные рекуррентные соотношения второго порядка с постоянными коэффициентами

Важный класс рекуррентных соотношений, который не может быть решен ни прямой, ни обратной подстановкой, — это рекуррентные соотношения вида

$$ax(n) + bx(n-1) + cx(n-2) = f(n), \qquad \text{(Б.7)}$$

[1] Строго говоря, корректность общей формулы необходимо доказать методом математической индукции по i. Однако зачастую проще сначала получить решение, а потом проверить его (например, как мы делали это ранее для $x(n) = n(n+1)/2$).

где a, b и c — действительные числа, $a \neq 0$. Такое рекуррентное соотношение называется ***линейным рекуррентным соотношением второго порядка с постоянными коэффициентами*** (second-order linear recurrence with constant coefficients). "Второго порядка", поскольку в неизвестной последовательности элементы $x(n)$ и $x(n-2)$ отстоят друг от друга на расстоянии двух позиций. Оно линейно, поскольку левая часть представляет собой линейную комбинацию неизвестных членов последовательности. Это уравнение с постоянными коэффициентами, так как a, b и c представляют собой фиксированные числа, не зависящие от n. Если $f(n) \equiv 0$ для всех n, рекуррентное соотношение называется ***однородным*** (homogeneous); в противном случае оно неоднородно (inhomogeneous).

Начнем с рассмотрения однородного случая:

$$ax(n) + bx(n-1) + cx(n-2) = 0. \qquad \text{(Б.8)}$$

Исключая вырожденный случай $b = c = 0$, уравнение (Б.8) имеет бесконечно много решений. Все эти решения, в совокупности образующие общее решение уравнения (Б.8), могут быть получены по одной из трех приведенных ниже формул. Какая именно формула применима в каждом конкретном случае, зависит от корней квадратного уравнения с теми же коэффициентами, что и у рекуррентного уравнения (Б.8):

$$ar^2 + br + c = 0. \qquad \text{(Б.9)}$$

Квадратное уравнение (Б.9) называется ***характеристическим уравнением*** (characteristic equation) рекуррентного соотношения (Б.8).

Теорема 1. Пусть r_1 и r_2 — два корня характеристического уравнения (Б.9) для рекуррентного соотношения (Б.8).

Случай 1. Если r_1 и r_2 — действительны и различны, то общее решение рекуррентного соотношения (Б.8) можно получить по формуле

$$x(n) = \alpha r_1^n + \beta r_2^n,$$

где α и β — две произвольные действительные константы.

Случай 2. Если r_1 и r_2 равны, то общее решение рекуррентного соотношения (Б.8) можно получить по формуле

$$x(n) = \alpha r^n + \beta n r^n,$$

где $r = r_1 = r_2$, и α и β — две произвольные действительные константы.

Случай 3. Если $r_{1,2} = u \pm iv$ — два различных комплексных корня, то общее решение рекуррентного соотношения (Б.8) можно получить по формуле

$$x(n) = \gamma^n (\alpha \cos n\theta + \beta \sin n\theta),$$

где $\gamma = \sqrt{u^2 + v^2}$, $\theta = arctg\,(v/u)$, и α и β — две произвольные действительные константы. ■

Первый случай данной теоремы реализуется, в частности, при выводе явной формулы для n-го числа Фибоначчи (раздел 2.5). В качестве другого примера решим рекуррентное соотношение

$$x\,(n) - 6x\,(n-1) + 9x\,(n-2) = 0.$$

Его характеристическое уравнение

$$r^2 - 6r + 9 = 0$$

имеет два одинаковых корня $r_1 = r_2 = 3$. Следовательно, согласно случаю 2 теоремы 1, его общее решение дает формула

$$x\,(n) = \alpha 3^n + \beta n 3^n.$$

Если вы хотите найти частное решение, для которого, скажем, $x\,(0) = 0$ и $x\,(1) = 3$, подставьте $n = 0$ и $n = 1$ в последнее уравнение, чтобы получить систему двух линейных уравнений с двумя неизвестными. Для указанных начальных условий $\alpha = 0$ и $\beta = 1$, следовательно, искомое частное решение —

$$x\,(n) = n3^n.$$

Давайте теперь вернемся к случаю неоднородного линейного рекуррентного соотношения второго порядка с постоянными коэффициентами.

Теорема 2. Общее решение неоднородного уравнения (Б.7) может быть получено как сумма общего решения соответствующего однородного уравнения (Б.8) и частного решения неоднородного уравнения (Б.7). ■

Поскольку теорема 1 дает возможность получить общее решение однородного линейного рекуррентного уравнения второго порядка с постоянными коэффициентами, теорема 2 сводит задачу поиска всех решений уравнения (Б.7) к поиску одного частного решения. Для произвольной функции $f\,(n)$ в правой части уравнения (Б.7) задача остается весьма сложной, не имеющей общих способов решения. Однако для некоторых классов функций может быть найдено частное решение. В частности, если $f\,(n)$ — ненулевая константа, мы можем искать частное решение, также представляющее собой константу.

В качестве примера найдем общее решение неоднородного рекуррентного уравнения

$$x\,(n) - 6x\,(n-1) + 9x\,(n-2) = 4.$$

Если $x(n) = c$ — частное решение этого рекуррентного уравнения, то величина c должна удовлетворять уравнению

$$c - 6c + 9c = 4,$$

откуда $c = 1$. Поскольку мы уже нашли общее решение соответствующего однородного рекуррентного уравнения

$$x(n) - 6x(n-1) + 9x(n-2) = 0,$$

общее решение рекуррентного соотношения $x(n) - 6x(n-1) + 9x(n-2) = 4$ можно записать в виде

$$x(n) = \alpha 3^n + \beta n 3^n + 1.$$

Перед тем как завершить рассмотрение этой темы, заметим, что результаты, аналогичные теоремам 1 и 2, справедливы и для ***обобщенного линейного рекуррентного уравнения n-ой степени с постоянными коэффициентами*** (general linear kth degree recurrence with constant coefficients)

$$a_k x(n) + a_{k-1} x(n-1) + \cdots + a_0 x(n-k) = f(n). \qquad (Б.10)$$

Однако практичность этого обобщения весьма ограничена, поскольку при этом требуется искать корни полинома k-ой степени:

$$a_k r^k + a_{k-1} r^{k-1} + \cdots + a_0 = 0. \qquad (Б.11)$$

Уравнение (Б.11) является характеристическим уравнением для рекуррентного соотношения (Б.10).

Имеется также несколько других, более сложных методов решения рекуррентных соотношений. Особенно богатым материалом на эту тему, рассмотренным с точки зрения анализа алгоритмов, отличается книга Пурдома (Purdom) и Брауна (Brown) [93].

Распространенные типы рекуррентных соотношений в анализе алгоритмов

Имеется несколько видов рекуррентных соотношений, которые с завидной регулярностью появляются при анализе алгоритмов. Это связано с тем, что они отражают один из фундаментальных методов проектирования алгоритмов.

Уменьшение на единицу

Алгоритм уменьшения на единицу решает поставленную задачу, используя соотношение между данным экземпляром размером n и меньшим экземпляром размером $n-1$. К конкретным примерам относится рекурсивное вычисление $n!$ (раздел 2.4) и сортировка вставкой (раздел 5.1). Рекуррентное соотношение, возникающее при изучении временно́й эффективности таких алгоритмов, обычно имеет вид

$$T(n) = T(n-1) + f(n), \qquad (Б.12)$$

где функция $f(n)$ учитывает время, необходимое для приведения экземпляра задачи к меньшему и возврата от решения меньшего экземпляра к решению большего экземпляра. Применение обратной подстановки к (Б.12) дает

$$\begin{aligned} T(n) &= T(n-1) + f(n) = \\ &= T(n-2) + f(n-1) + f(n) = \\ &= \cdots = \\ &= T(0) + \sum_{j=1}^{n} f(j). \end{aligned}$$

Для конкретной функции $f(n)$ сумма $\sum_{j=1}^{n} f(j)$ обычно либо вычисляется точно, либо выясняется ее порядок роста. Например, если $f(n) = 1$, то $\sum_{j=1}^{n} f(j) = n$; если $f(n) = \log n$, то $\sum_{j=1}^{n} f(j) \in \Theta(n \log n)$; если $f(n) = n^k$, то $\sum_{j=1}^{n} f(j) \in \in \Theta\left(n^{k+1}\right)$. Сумма $\sum_{j=1}^{n} f(j)$ может быть также приближенно вычислена при помощи формул, включающих интегралы (см., в частности, соответствующие формулы в приложении А).

Уменьшение на постоянный множитель

Алгоритм уменьшения на постоянный множитель решает задачу путем приведения ее экземпляра размером n к экземпляру размером n/b (для большинства, но не для всех алгоритмов $b = 2$). Алгоритм рекурсивно решает меньшие экземпляры; затем при необходимости он распространяет решение меньшего экземпляра на решение исходного экземпляра задачи. Наиболее важным примером такого алгоритма является бинарный поиск; другие примеры включают возведение в степень посредством вычисления квадратов (см. введение к главе 5), умножение по-русски и задачу поиска фальшивой монеты (раздел 5.5). Рекуррентное соотношение, возникающее при изучении временно́й эффективности таких алгоритмов, обычно имеет вид

$$T(n) = T(n/b) + f(n), \qquad (Б.13)$$

где $b > 1$ и функция $f(n)$ учитывает время, необходимое для приведения экземпляра задачи к меньшему и возврата от решения меньшего экземпляра к решению большего экземпляра. Строго говоря, уравнение (Б.13) корректно только для

$n = b^k$, $k = 0, 1, \ldots$. Для значений n, не являющихся степенями b, обычно применяется округление при помощи функций для округления сверху и снизу ("пол" и "потолок"). Стандартный подход к таким уравнениям заключается в первоначальном решении для $n = b^k$. После этого решение либо немного корректируется, чтобы быть верным для всех n (см., например, раздел 4.3 и упражнение 4.3.3), либо на основании ***правила гладкости*** (smoothness rule), сформулированного в виде теоремы 4 в данном приложении, определяется порядок роста решения для всех n.

Рассматривая $n = b^k$, $k = 0, 1, \ldots$ и применяя обратную подстановку к (Б.13), получаем:

$$\begin{aligned} T\left(b^k\right) &= T\left(b^{k-1}\right) + f\left(b^k\right) = \\ &= T\left(b^{k-2}\right) + f\left(b^{k-1}\right) + f\left(b^k\right) = \\ &= \cdots = \\ &= T(1) + \sum_{j=1}^{k} f\left(b^j\right). \end{aligned}$$

Для конкретной функции $f(n)$ сумма $\sum_{j-1}^{k} f\left(b^j\right)$ обычно либо может быть вычислена точно, либо получен ее порядок роста. Например, если $f(n) = 1$,

$$\sum_{j=1}^{k} f\left(b^j\right) = k = \log_b n.$$

Если $f(n) = n$, мы получаем другой пример точного вычисления суммы:

$$\sum_{j=1}^{k} f\left(b^j\right) = \sum_{j=1}^{k} b^j = b\frac{b^k - 1}{b - 1} = b\frac{n - 1}{b - 1}.$$

Кроме того, рекуррентное соотношение (Б.13) является частным случаем рекуррентного соотношения (Б.14), для которого выполняется ***Основная теорема*** (Master theorem) (теорема 5 из данного приложения). Согласно этой теореме, в частности, если $f(n) \in \Omega\left(n^d\right)$, где $d > 0$, то и $T(n) \in \Omega\left(n^d\right)$.

Декомпозиция

Алгоритм декомпозиции решает задачу путем разделения данного экземпляра на несколько меньших, рекурсивного решения каждого из них и при необходимости комбинации решений меньших экземпляров в решение данного экземпляра. В предположении, что все меньшие экземпляры имеют один и тот же размер n/b и реально решаются a из них, мы получим следующее рекуррентное соотношение, справедливое для $n = b^k$, $k = 1, 2, \ldots$:

$$T(n) = aT(n/b) + f(n), \qquad \text{(Б.14)}$$

где $a \geqslant 1$, $b \geqslant 2$, а $f(n)$ — функция, учитывающая время, затрачиваемое на разделение задачи на меньшие и комбинацию их решений. Рекуррентное соотношение (Б.14) называется ***общим рекуррентным соотношением декомпозиции*** (general divide-and-conquer recurrence)[2].

Применение обратной подстановки к (Б.14) приводит к следующему результату:

$$\begin{aligned}
T\left(b^k\right) &= aT\left(b^{k-1}\right) + f\left(b^k\right) = \\
&= a\left(T\left(b^{k-2}\right) + f\left(b^{k-1}\right)\right) + f\left(b^k\right) = \\
&= a^2T\left(b^{k-2}\right) + af\left(b^{k-1}\right) + f\left(b^k\right) = \\
&= a^2\left(aT\left(b^{k-3}\right) + f\left(b^{k-2}\right)\right) + af\left(b^{k-1}\right) + f\left(b^k\right) = \\
&= a^3T\left(b^{k-3}\right) + a^2f\left(b^{k-2}\right) + af\left(b^{k-1}\right) + f\left(b^k\right) = \\
&= \cdots = \\
&= a^kT(1) + a^{k-1}f\left(b^1\right) + a^{k-2}f\left(b^2\right) + \cdots + a^0f\left(b^k\right) = \\
&= a^k\left(T(1) + \sum_{j=1}^{k} f\left(b^j\right)/a^j\right).
\end{aligned}$$

Поскольку $a^k = a^{\log_b n} = n^{\log_b a}$, получаем следующую формулу для решения рекуррентного соотношения (Б.14) при $n = b^k$:

$$T(n) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n} f\left(b^j\right)/a^j\right). \qquad \text{(Б.15)}$$

Очевидно, что порядок роста решения $T(n)$ зависит от значений констант a и b и порядка роста функции $f(n)$. При определенных предположениях о свойствах функции $f(n)$, рассматриваемых в этом разделе, упростив формулу (Б.15), можно получить точный результат для порядка роста $T(n)$.

Правило гладкости и Основная теорема

Ранее мы упоминали, что временна́я эффективность алгоритмов уменьшения на постоянный множитель и декомпозиции обычно сначала рассматривается для значений n, являющихся степенями b (чаще всего $b = 2$, как в случае бинарного поиска или сортировки слиянием; иногда $b = 3$, как в случае лучшего алгоритма решения головоломки о взвешивании монет из раздела 5.5, но вообще говоря

[2] В используемой нами терминологии при $a = 1$ соотношение описывает алгоритмы уменьшения на постоянный множитель, а не декомпозиции.

b может принимать любое значение, не меньшее 2). Вопрос, который мы рассмотрим, заключается в распространении результата поиска порядка роста для n, являющихся степенями b, на все значения n.

Определение 1. Пусть $f(n)$ — неотрицательная функция, определенная на множестве натуральных чисел. $f(n)$ называется ***в конечном счете неубывающей*** (eventually nondecreasing), если существует некоторое неотрицательное целое число n_0 такое, что $f(n)$ является неубывающей функцией на интервале $[n_0, \infty)$, т.е. $f(n_1) \leqslant f(n_2)$ для любых $n_2 > n_1 \geqslant n_0$. ■

Например, функция $(n-100)^2$ является в конечном счете неубывающей, хотя она и убывает на отрезке $[1, 100]$; функция же $\sin^2 \pi n/2$ представляет собой функцию, не являющуюся в конечном счете неубывающей. Подавляющее большинство функций, с которыми мы сталкивались при анализе алгоритмов, *являются* в конечном счете неубывающими (большинство из них на самом деле являются неубывающими во всей их области определения).

Определение 2. Пусть $f(n)$ — неотрицательная функция, определенная на множестве натуральных чисел. $f(n)$ называется ***гладкой*** (smooth), если она в конечном счете неубывающая и $f(2n) \in \Theta(f(n))$. ■

Легко убедиться, что функции, растущие не слишком быстро, включая $\log n$, n, $n \log n$ и n^α, где $\alpha \geqslant 0$, являются гладкими. Например, $f(n) = n \log n$ — гладкая, поскольку

$$f(2n) = 2n \log 2n = 2n(\log 2 + \log n) = (2 \log 2)\, n + 2n \log n \in \Theta(n \log n).$$

Быстро растущие функции, такие как a^n, где $a > 1$ и $n!$, не являются гладкими. Например, функция $f(n) = 2^n$ не является гладкой, так как

$$f(2n) = 2^{2n} = 4^n \notin \Theta(2^n).$$

Теорема 3. Пусть $f(n)$ — гладкая функция. Тогда для любого целого $b \geqslant 2$

$$f(bn) \in \Theta(f(n)),$$

т.е. существуют положительные константы c_b и d_b и неотрицательное целое число n_0 такие, что

$$d_b f(n) \leqslant f(bn) \leqslant c_b f(n) \text{ при } n \geqslant n_0.$$

(То же утверждение, с очевидными изменениями, справедливо и для O и Ω обозначений.)

Доказательство. Докажем теорему только для O-обозначения; доказательство для Ω аналогичное. По индукции легко убедиться, что если $f(2n) \leqslant c_2 f(n)$ для $n \geqslant n_0$, то

$$f\left(2^k n\right) \leqslant c_2^k f(n) \text{ при } k = 1, 2, \ldots \text{ и } n \geqslant n_0.$$

Гипотеза индукции при $k = 1$ проверяется тривиально. В общем случае, полагая, что $f\left(2^{k-1} n\right) \leqslant c_2^{k-1} f(n)$ при $n \geqslant n_0$, получим

$$f\left(2^k n\right) = f\left(2 \cdot 2^{k-1} n\right) \leqslant c_2 f\left(2^{k-1} n\right) \leqslant c_2 c_2^{k-1} f(n) = c_2^k f(n).$$

(Тем самым теорема доказывается для $b = 2^k$.) Рассмотрим теперь произвольное целое $b \geqslant 2$. Пусть k — положительное целое такое, что $2^{k-1} \leqslant b < 2^k$. Мы можем оценить $f(bn)$ сверху, без потери общности, считая, что $f(n)$ — неубывающая функция при $n \geqslant n_0$:

$$f(bn) \leqslant f\left(2^k n\right) \leqslant c_2^k f(n).$$

Следовательно, мы можем использовать c_2^k в качестве константы для данного значения b, что и завершает доказательство. ■

Важность введенных понятий объясняется следующей теоремой.

Теорема 4 (Правило гладкости). Пусть $T(n)$ — в конечном счете неубывающая функция, а $f(n)$ — гладкая функция. Если

$$T(n) \in \Theta(f(n)) \text{ для значений } n, \text{ равных степени } b,$$

где $b \geqslant 2$, то

$$T(n) \in \Theta(f(n)).$$

(Аналогичные результаты выполняются для O и Ω обозначений.)

Доказательство. Мы докажем только O-часть; доказательство для Ω выполняется аналогично. В соответствии с формулировкой теоремы, имеется положительная константа c и положительное целое $n_0 = b^{k_0}$ такие, что

$$T\left(b^k\right) \leqslant c f\left(b^k\right) \text{ для } b^k \geqslant n_0,$$

$T(n)$ — невозрастающая функция при $n \geqslant n_0$, а $f(bn) \leqslant c_b f(n)$ при $n \geqslant n_0$ в соответствии с теоремой 3. Рассмотрим произвольное значение $n \geqslant n_0$. Оно ограничено двумя последовательными степенями b: $n_0 \leqslant b^k \leqslant n < b^{k+1}$. Таким образом,

$$T(n) \leqslant T\left(b^{k+1}\right) \leqslant c f\left(b^{k+1}\right) = c f\left(b b^k\right) \leqslant c c_b f\left(b^k\right) \leqslant c c_b f(n).$$

Следовательно, мы можем использовать произведение cc_b в качестве константы, требующейся в определении $O(f(n))$, что и завершает доказательство теоремы. ■

Теорема 4 позволяет распространить информацию о порядке роста, найденную для $T(n)$ для удобного подмножества значений (степеней b), на всю область определения данной функции.

Теорема 5 (Основная теорема). Пусть $T(n)$ — в конечном счете неубывающая функция, удовлетворяющая рекуррентному соотношению

$$T(n) = aT(n/b) + f(n) \text{ при } n = b^k, k = 1, 2, \ldots; T(1) = c,$$

где $a \geqslant 1$, $b \geqslant 2$, $c > 0$. Если $f(n) \in \Theta\left(n^d\right)$, где $d \geqslant 0$, то

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{если } a < b^d, \\ \Theta\left(n^d \log n\right) & \text{если } a = b^d, \\ \Theta\left(n^{\log_b a}\right) & \text{если } a > b^d. \end{cases}$$

(Аналогичные результаты справедливы для O и Ω обозначений.)

Доказательство. Мы докажем теорему для важного частного случая $f(n) = n^d$ (доказательство для общего случая представляет собой небольшое техническое изменение приведенного — см., например, [32]). Если $f(n) = n^d$, уравнение (Б.15) дает для $n = b^k$, $k = 0, 1, \ldots$

$$T(n) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n} b^{jd}/a^j\right) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n}\left(b^d/a\right)^j\right).$$

Сумма в этой формуле представляет собой геометрическую прогрессию, так что

$$\sum_{j=1}^{\log_b n}\left(b^d/a\right)^j = \left(b^d/a\right)\frac{\left(b^d/a\right)^{\log_b n} - 1}{\left(b^d/a\right) - 1}, \text{ если } b^d \neq a,$$

и

$$\sum_{j=1}^{\log_b n}\left(b^d/a\right)^j = \log_b n, \text{ если } b^d = a.$$

Если $a < b^d$, то $b^d/a > 1$ и, таким образом,

$$\sum_{j=1}^{\log_b n}\left(b^d/a\right)^j = \left(b^d/a\right)\frac{\left(b^d/a\right)^{\log_b n} - 1}{\left(b^d/a\right) - 1} \in \Theta\left(\left(b^d/a\right)^{\log_b n}\right).$$

Следовательно,

$$T(n) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n}\left(b^d/a\right)^j\right) \in n^{\log_b a}\Theta\left(\left(b^d/a\right)^{\log_b n}\right) =$$
$$= \Theta\left(n^{\log_b a}\left(b^d/a\right)^{\log_b n}\right) = \Theta\left(a^{\log_b n}\left(b^d/a\right)^{\log_b n}\right) =$$
$$= \Theta\left(b^{d\log_b n}\right) = \Theta\left(b^{\log_b n^d}\right) = \Theta\left(n^d\right).$$

Если $a > b^d$, то $b^d/a < 1$ и, таким образом,

$$\sum_{j=1}^{\log_b n}\left(b^d/a\right)^j = \left(b^d/a\right)\frac{\left(b^d/a\right)^{\log_b n} - 1}{\left(b^d/a\right) - 1} \in \Theta(1).$$

Следовательно,

$$T(n) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n}\left(b^d/a\right)^j\right) \in \Theta\left(n^{\log_b a}\right).$$

Если $a = b^d$, то $b^d/a = 1$ и, следовательно,

$$T(n) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n}\left(b^d/a\right)^j\right) =$$
$$= n^{\log_b a}\left(T(1) + \log_b n\right) \in \Theta\left(n^{\log_b a}\log_b n\right) =$$
$$= \Theta\left(n^{\log_b b^d}\log_b n\right) = \Theta\left(n^d \log_b n\right).$$

Поскольку $f(n) = n^d$ — гладкая функция при любом d, обращение к теореме 4 завершает доказательство. ■

Теорема 5 предоставляет очень удобный инструмент для быстрого анализа эффективности алгоритмов декомпозиции и уменьшения размера на постоянный множитель. В книге имеется много примеров его применения.

Список литературы

[1] Adelson–Velsky, G.M. and Landis, E.M. An algorithm for organization of information. *Soviet Mathematics Doklady*, vol. 3, 1962, 1259–1263.

[2] Adleman, L.M. Molecular computation of solutions to combinatorial problems. *Science*, vol. 266, 1994, 1021–1024.

[3] Agrawal, M., Kayal, N., and Saxena, N. *PRIMES Is in P.* Preprint, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur-208016, India, August 6, 2002.

[4] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, Reading, MA, 1974.

[5] Aho, A.V, Hopcroft, J.E., and Ullman, J.D. *Data Structures and Algorithms.* Addison–Wesley, Reading, MA, 1983. (Альфред Ахо, Джон Хопкрофт, Джеффри Ульман, *Структуры данных и алгоритмы*. М., ИД "Вильямс", 2000.)

[6] Albert, R., Hawoong Jeong, and Barbбsi, A.-L. Diameter of the World-Wide Web. *Nature*, vol. 401, 1999, 130–131.

[7] Atallah, M.J., editor. *Algorithms and Theory of Computation Handbook.* CRC Press, Boca Raton, FL, 1998.

[8] Baase, S., and Van Gelder, A. *Computer Algorithms: Introduction to Design and Analysis*, 3rd ed. Addison–Wesley, Reading, MA, 2000.

[9] Baecker, R. (with assistance of D. Sherman). *Sorting Out Sorting.* 30-minute color sound film. Dynamic Graphics Project, University of Toronto, 1981. (Distributed by Morgan Kaufmann, Publishers.)

[10] Baecker, R. Sorting out sorting: a case study of software visualization for teaching computer science. In *Software Visualization: Programming as a Multimedia Experience*, edited by J. Stasko, J. Domingue, M.C. Brown, and B.A. Price. MIT Press, Cambridge, MA, 1998, 369–381.

[11] Baeza–Yates, R.A. Teaching Algorithms. *ACM SIGACT News*, vol. 26, no. 4, Dec. 1995, 51–59.

[12] Bayer, R., and McGreight, E.M. Organization and maintenance of large ordered indices. *Acta Informatica*, vol. 1, no. 3, 1972, 173–189.

[13] Bellman, R.E. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[14] Bellman, R.E., and Dreyfus, S.E. *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, 1962.

[15] Bentley, J. *Programming Pearls*, 2nd ed. Addison–Wesley, Reading, MA, 2000 (Бентли Дж. Жемчужины программирования. 2-е изд. — СПб. Питер — 2002.).

[16] Berlinski, D. *The Advent of the Algorithm.* Harcourt, Inc., New York, 2000.

[17] Berman, K.A., and Paul, J.L. *Fundamentals of Sequential and Parallel Algorithms.* PWS Publishing, Boston, 1996.

[18] Berstekas, D.P. *Dynamic Programming and Optimal Control*, 2nd Edition (2 vols.), Athena Scientific, 2001.

[19] Bloom, M., Floyd, R.W., Pratt, V, Rivest, R.L., and Tarjan, R.E. Time bounds for selection. *Journal of Computer and System Sciences*, vol. 7, no. 4, 1973, 448–461.

[20] Bogomolny, A. *Interactive Mathematics Miscellany and Puzzles.* `http://www.cut-the-knot.org`, 1996.

[21] Boyer, R.S., and Moore, J.S. A fast string searching algorithm. *Communications of the ACM*, vol. 21, no. 10,1977, 762–772.

[22] Brassard, G., and Bratley, P. *Fundamentals of Algorithmics.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[23] Brown, M.R. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991, 4–9.

[24] Brown, M., and Sedgewick, R. A system for algorithm animation. In *Proceedings of ACM SIGGRAPH '84*, Minneapolis, July 1984, 177–186.

[25] Brualdi, R. *Introductory Combinatorics*, 3rd ed. Prentice Hall, Upper Saddle River, NJ, 1999.

[26] Chabert, Jean–Luc, editor. *A History of Algorithms: From the Pebble to the Microchip.* Translated by Chris Weeks. Springer–Verlag, Berlin, 1998.

[27] Chandler, J.P. Patent protection of computer programs. *Minnesota Intellectual Property Review*, vol. 1, no. 1, 2000, 33–46.

[28] Comer, D. The ubiquitous B-tree. *ACM Computing Surveys*, vol. 11, no. 2, 1979, 121–137.

[29] *Computing Curricula 2001.* The Joint Task Force on Computing Curricula. IEEE Computer Society, Association for Computing Machinery, 2001.

[30] Cook, S.A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, 1971, 151–158.

[31] Coopersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proceedings of Nineteenth Annual ACM Symposium on the Theory of Computing*, 1987, 1–6.

[32] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and C. Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA, 2001. (Кормен Томас, Лейзерсон Чарльз, Ривест Рональд, Стайн Клиффорд, *Введение в алгоритмы. Второе издание*. М., ИД "Вильямс", 2005.)

[33] Dantzig, G.B. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[34] Dewdney, A.K. *The (New) Turing Omnibus.* Computer Science Press, New York, 1993.

[35] Dijkstra, E.W. A note on two problems in connection with graphs. *Numerische Mathematik*, vol. 1, 1959, 269–271.

[36] Floyd, R.W. Algorithm 97: shortest path. *Communications of the ACM*, vol. 5, no. 6, 1962, 345.

[37] Forsythe, G.E. What to do till the computer scientist comes. *American Mathematical Monthly*, vol. 75, 1968, 454–462.

[38] Forsythe, G.E. Solving a quadratic equation on a computer. In *The Mathematical Sciences*, edited by COSRIMS and George Boehm, MIT Press, Cambridge, MA, 1969, 138–152.

[39] Gardner, M. *My Best Mathematical and Logic Puzzles.* Dover Publications, New York, 1994.

[40] Garey, M.R., and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[41] Gerald, C.F., and Wheatley, P.O. *Applied Numerical Analysis*, 6th ed. Addison–Wesley, Reading, MA, 1999.

[42] Gloor, P.A. User interface issues for algorithm animation. In *Software Visualization: Programming as a Multimedia Experience*, edited by John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. MIT Press, Cambridge, MA, 1998, 145–152.

[43] Gonnet, G.H., and Baeza–Yates, R. *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd ed. Addison–Wesley, Reading, MA, 1991.

[44] Goodrich, M.T., and Tamassia, R. *Algorithm Design: Foundations, Analysis, and Internet Examples.* John Wiley & Sons, New York, 2002.

[45] Graham, R.L., Knuth, D.E., and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison–Wesley, Reading, MA, 1994. (Р. Грэхем, Д. Кнут, О. Паташник. *Конкретная математика. Основание информатики.* М., "Мир", 1998.)

[46] Green, D.H., and Knuth, D.E. *Mathematics for Analysis of Algorithms*, 2nd ed. Birkhduser, Boston, 1982.

[47] Gries, D. *The Science of Programming.* Springer–Verlag, New York, 1981.

[48] Harel, D. *Algorithmics: The Spirit of Computing*, 2nd ed. Addison–Wesley, Reading, MA, 1992.

[49] Hartmanis, J., and Steams, R.E. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, vol. 117, 1965, 285–306.

[50] Heap, B.R. Permutations by interchanges. *Computer Journal*, vol. 6, 1963, 293–294.

[51] Hoare, C.A.R. Quicksort. In *Great Papers in Computer Science*, Phillip Laplante, ed. West Publishing Company, Minneapolis/St. Paul, 1996, pp. 31–39.

[52] Hochbaum, D.S., editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, Boston, 1997.

[53] Hopcroft, J.E. Computer science: The emergence of a discipline. *Communications of the ACM*, vol. 30, no. 3, 1987, pp. 198–202.

[54] Horowitz, E., Sahni, S., and Rajasekaran, S. *Computer Algorithms.* Computer Science Press, New York, 1998.

[55] Horspool, R.N. Practical fast searching in strings. *Software—Practice and Experience*, vol. 10, 1980, 501–506.

[56] Huffman, D.A. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, vol. 40, 1952, 1098–1101.

[57] Karmarkar, N. A new polynomial-time algorithm for linear programming. *Combinatorica*, vol. 4, 1984, 373–395.

[58] Karp, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Communications*, edited by R.E. Miller and J.W. Thatcher. Plenum Press, New York, 1972, 85–103.

[59] Karp, R.M. Combinatorics, complexity, and randomness. *Communications of the ACM*, vol. 29, no. 2, 1986, 89–109.

[60] Karp, R.M. An introduction to randomized algorithms. *Discrete Applied Mathematics*, vol. 34, 1991, 165–201.

[61] Kernighan, B.W, and Pike. R. *The Practice of Programming.* Addison–Wesley, Reading, MA, 1999. (Керниган Б.В., Пайк Р. *Практика программирования.* С.-Пб., "Невский Диалект", 2001.)

[62] Knuth, D.E. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, vol. 29, 1975, 121–136.

[63] Knuth, D.E. Big omicron and big omega and big theta. *ACM SIGACT News*, vol. 8, no. 2, 1976, 18–23.

[64] Knuth, D.E. *Selected Papers on Computer Science*. CSLI Publications and Cambridge University Press, New York, 1996.

[65] Knuth, D.E. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, 3rd ed. Addison–Wesley, Reading, MA, 1997. (Дональд Э. Кнут. *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* М., ИД "Вильямс", 2000.)

[66] Knuth, D.E. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, 3rd ed. Addison–Wesley, Reading, MA, 1998. (Дональд Э. Кнут. *Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.* М., ИД "Вильямс", 2000.)

[67] Knuth, D.E. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, 2nd ed. Addison–Wesley, Reading, MA, 1998. (Дональд Э. Кнут. *Искусство программирования, том 3. Сортировка и поиск, 2-е изд.* М., ИД "Вильямс", 2000.)

[68] Knuth, D.E., Morris, Jr., J.H., and Pratt, V.R. Fast pattern matching in strings. *SIAM Journal on Computing*, vol. 5, no. 2, 1977, 323–350.

[69] Kolman, B., and Beck, R.E. *Elementary Linear Programming with Applications*, 2nd ed. Academic Press, San Diego, 1995.

[70] Kordemsky, B.A. *The Moscow Puzzles*. Charles Scribners's Sons, New York, 1972.

[71] Kruskal, J.B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, vol. 7, 1956, 48–50.

[72] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., editors. *The Traveling Salesman Problem*. John Wiley & Sons, New York, 1985.

[73] Levin, L.A. Universal sorting problems. *Problemy Peredachi Informatsii*, vol. 9, no. 3, 1973, 115–116 (in Russian). English translation in *Problems of Information Transmission*, vol. 9, 265–266. (Левин Л.А., Универсальные проблемы сортировки. *Проблемы передачи информации*, том 9, [1] 3, 1973, с. 115–116)

[74] Levitin, A. Do we teach the right algorithm design techniques? In *Proceedings SIGCSE '99*, New Orleans, 1999, 179–183.

[75] Levitin, A., and Papalaskari, M-A. Using puzzles in teaching algorithms. *SIGCSE '02*, Cincinnati, 2002.

[76] Manber, U. *Introduction to Algorithms: A Creative Approach.* Addison–Wesley, Reading, MA, 1989.

[77] Martello, S., and Toth, P. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, Chichester, England, 1990.

[78] Miller, R., and Boxer, L. *A Unified Approach to Sequential and Parallel Algorithms.* Prentice Hall, Upper Saddle River, NJ, 2000.

[79] Moret, B.M.E., and Shapiro, H.D. *Algorithms from P to NP*, Volume I: *Design and Efficiency.* Benjamin Cummings, Redwood City, CA, 1991.

[80] Moscovich, I. *1000 Play Thinks: Puzzles, Paradoxes, Illusions, and Games.* Workman, New York, 2001.

[81] Motwani, R., and Raghavan, P. *Randomized Algorithms.* Cambridge University Press, New York, 1995.

[82] Neapolitan, R., and Naimipour, K. *Foundations of Algorithms: With C++ Pseudocode*, 2nd ed. Jones and Bartlett, Sudbury, MA, 1998.

[83] Nielsen, M.A., and Chuang, I.L. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, UK, 2000.

[84] O'Connor, J.J., and Robertson, E.F. *The MacTutor History of Mathematics Archive*, June 1998, `http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Abel.html`.

[85] Pan, V.Y. Strassen's algorithm is not optimal. In *Proceedings of Ninteenth Annual IEEE Symposium on the Foundations of Computer Science.* 1978, 166–176.

[86] Papadimitriou, C.H. *Computational Complexity.* Addison–Wesley, Reading, MA, 1994.

[87] Papadimitriou, C.H., and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity.* Prentice–Hall, Englewood Cliffs, NJ, 1982.

[88] Parberry, I. *Problems on Algorithms.* Prentice–Hall, Englewood Cliffs, NJ, 1995.

[89] Polya, G. *How to Solve It.* Doubleday, Garden City, NY, 1957.

[90] Preparata, EP, and Shamos, M.I. *Computational Geometry: An Introduction.* Springer–Verlag, New York, 1985.

[91] Price B., Baecker, R., and Small, I. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, vol. 4, no. 3, 1993, 211–266.

[92] Prim, R.C. Shortest connection networks and some generalizations. *Bell System Technical Journal*, vol. 36, no. 1, 1957, 1389–1401.

[93] Purdom, P.W., Jr., and Brown, C. *The Analysis of Algorithms.* Holt, Rinehart and Winston, New York, 1985.

[94] Rawlins, G.J.E. *Compared to What? An Introduction to the Analysis of Algorithms.* Computer Science Press, New York, 1991.

[95] Reingold, E.M., Nievergelt, J., and Deo, N. *Combinatorial Algorithms: Theory and Practice.* Prentice–Hall, Englewood Cliffs, NJ, 1977.

[96] Rivest, R.L., Shamir, A., and Adleman, L.M. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, vol. 21, no. 2, 1978, 120–126.

[97] Rosen, K.H. *Discrete Mathematics and Its Applications*, 4th ed. McGraw–Hill, New York, 1999.

[98] Rosenkrantz, D.J., Steams, R.E., and Lewis, P.M. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal of Computing*, vol. 6, 1977, 563–581.

[99] Sahni, S. Approximation algorithms for the 0/1 knapsack problem. *Journal of the ACM*, vol. 22, 1975, 115–124.

[100] Sahni, S., and Gonzalez, T. *P*-complete approximation problems. *Journal of the ACM*, vol. 23, 1976, 555–565.

[101] Sayood, K. *Introduction to Data Compression*, 2nd ed. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[102] Sedgewick, R. *Algorithms*, 2nd ed. Addison–Wesley, Reading, MA, 1988.

[103] Sedgewick, R. *Algorithms in C*, 3rd ed. Parts 1–4: Fundamentals, Data Structures, Sorting, Searching. Addison–Wesley, Reading, MA, 1998.

[104] Sedgewick, R. *Algorithms in C*, 3rd ed. Part 5: Graph Algorithms, Addison–Wesley, Boston, 2002.

[105] Sedgewick, R., and Flajolet, P. *An Introduction to the Analysis of Algorithms.* Addison–Wesley, Reading, MA, 1996.

[106] Shasha, D., and Lazere, C. *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists.* Copernicus, New York, 1998.

[107] Shell, D.L. A high-speed sorting procedure. *Communications of the ACM*, vol. 2, no. 7, July 1959, 30–32.

[108] Shen, A. Algorithms and Programming: Problems and Solutions. Birkhдuser, Boston, 1997.

[109] Shor, P.W. Algorithms for quantum computation: Discrete algorithms and factoring. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (Shafi Goldwasser, editor). IEEE Computer Society Press, 1994, 124–134.

[110] Sipser, M. *Introduction to the Theory of Computation.* PSW Publishing, Boston, 1997.

[111] Skiena, S.S. *Algorithm Design Manual.* Springer–Verlag, New York, 1998.

[112] Stasko, J. TANGO: A framework and system for algorithm animation. *Computer*, vol. 23, no. 9, 1990, 27–39.

[113] Strassen, V. Gaussian elimination is not optimal. *Numerische Mathematik*, vol. 13, 354–356.

[114] Tarjan, R.E. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[115] Tarjan, R.E., and van Leeuwen, J. Worst-case analysis of set union algorithms. *Journal of the ACM*, vol. 31, no. 2, 1984, 245–281.

[116] Tarjan, R.E. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, no. 2, 1985, 306-318.

[117] Tarjan, R.E. Algorithm design. *Communications of the ACM*, vol. 30, no. 3, 1987, 204–212.

[118] Tucker, A. *Applied Combinatorics.* John Wiley & Sons, New York, 1980.

[119] Warshall, S. A theorem on boolean matrices. *Journal of the ACM*, vol. 9, no. 1, 1962, 11–12.

[120] Weide, B. A survey of analysis techniques for discrete algorithms. *Computing Surveys*, vol. 9, no. 4, 1977, 291–313.

[121] Weiss, M.A. Data Structures and Algorithm Analysis, Benjamin/Cummings, 1991.

[122] Williams, J.W.J. Algorithm 232 (heapsort). *Communications of the ACM*, vol. 7, no. 6, 1964, 347–348.

[123] Wirth, N. *Algorithms + Data Structures = Programs.* Prentice–Hall, Englewood Cliffs, NJ, 1976. (Вирт Н. *Алгоритмы + структуры данных = программы.* — М.: Мир, 1985.)

[124] Yao, R Speed-up in dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, vol. 3, no. 4, 1982, 532–540.

Указания к упражнениям

Глава 1

Упражнения 1.1

1. Вероятно, быстрее всего выполнить поиск в Web, но может помочь и ваша библиотека.
2. Легко найти аргументы как "за", так и "против". Есть один основополагающий принцип — научные факты или математические выражения не могут быть запатентованы (как вы думаете, применим ли он в данном случае?). Но должен ли этот принцип препятствовать патентованию всех алгоритмов?
3. Вы можете считать, что составляете алгоритм для человека, а не для машины. Тем не менее убедитесь, что описание не содержит явных неоднозначностей. У Кнута в ([65], с. 6) имеется интересное сравнение алгоритмов с кулинарными рецептами.
4. **а)** Просто следуйте описанному в тексте алгоритму Евклида.
   б) Сравните количество делений, выполненных обоими алгоритмами.
5. Докажите, что если d является делителем как m, так и n (т.е. $m = sd$ и $n = td$ для некоторых натуральных s и t), то оно является и делителем как n, так и $r = m \bmod n$, и наоборот. Воспользуйтесь формулой $m = qn + r$ $(0 \leqslant r < n)$ и тем фактом, что если d является делителем u и v, то оно является делителем $u + v$ и $u - v$ (почему?).
6. Выполните одну итерацию алгоритма для двух произвольно выбранных чисел $m < n$.
7. Ответ на часть а) можно получить непосредственно; ответ на часть б) можно получить, исследуя производительность алгоритма для всех пар $1 < m < n \leqslant 10$.
8. **а)** Воспользуйтесь тождеством $\gcd(m, n) = \gcd(m - n, n)$ для $m \geqslant n > 0$.
   б) Вопрос заключается в том, чтобы вычислить количество различных чисел, которые могут быть написаны на доске, начиная с пары $m > n \geqslant 1$. Вы

должны воспользоваться связью этого вопроса с вопросом из части *а*). Рассмотрите несколько небольших примеров, в частности, для $n = 1$ и $n = 2$ — это должно вам помочь.

9. Имеется достаточно простой алгоритм, основанный на определении $\lfloor \sqrt{n} \rfloor$.

Упражнения 1.2

1. Крестьянину придется совершить несколько поездок через реку, начиная с единственно возможной.
2. В отличие от древней народной головоломки, первый ход в данном случае неочевиден.
3. Главным вопросом в данном случае является возможная неоднозначность.
4. Ваш алгоритм должен корректно работать для всех возможных значений коэффициентов, включая ноли.
5. Вы уже должны были встречаться с этим алгоритмом в одном из учебных курсов по программированию. Если вам удалось не встретиться с этим алгоритмом ранее, попробуйте либо самостоятельно разработать его, либо найти его описание в литературе.
6. Возможно, вам стоит прогуляться к ближайшему банкомату, чтобы освежить свою память.
7. Вопрос а) достаточно сложен, хотя ответ на него, найденный в 1760-х годах немецким математиком Иоганном Ламбертом (Johann Lambert), хорошо известен. Вопрос б) существенно проще.
8. Думаю, вам известно не менее двух алгоритмов сортировки массива чисел.
9. Можно уменьшить количество итераций внутреннего цикла, ускорить работу цикла (как минимум для некоторых входных данных) или, что существенно важнее, разработать более быстрый алгоритм с нуля.

Упражнения 1.3

1. Отследите работу алгоритма для предоставленных входных данных. Воспользуйтесь при ответе на вопросы определениями устойчивости и обменного алгоритма, приведенными в тексте раздела.
2. Если вы не можете вспомнить ни одного алгоритма поиска, вам придется самостоятельно разработать его. (Не поддавайтесь искушению заглянуть в конец книги.)
3. Такой алгоритм будет приведен далее в книге, но его разработка не должна вызвать у вас проблем.

4. Если вы не сталкивались с этой задачей ранее, можете поискать ответ в Web или в книге по дискретным структурам. Ответ удивительно прост.
5. Не существует эффективного алгоритма решения подобной задачи на произвольном графе. Однако найти цепь Гамильтона для данного конкретного графа несложно (вам надо найти только одну из множества возможных цепей).
6. **а)** Представьте себя на месте пассажира и спросите сами себя, что для вас означает "наилучший маршрут". Затем подумайте о других пассажирах, с иными требованиями к маршруту.

   б) Представить задачу в виде графа очень просто. Подумать надо только над станциями пересадок.
7. **а)** Что собой представляет маршрут коммивояжера с точки зрения комбинаторики?

   б) Представляется вполне естественным рассматривать вершины одного цвета как элементы одного и того же подмножества.
8. Создайте граф, вершины которого соответствуют областям карты. Чему соответствуют ребра графа, — решите сами.
9. Предположите, что данная окружность существует, и начните с поиска ее центра. Не забудьте о частном случае $n \leqslant 2$.
10. Будьте внимательны и не забудьте о каком-нибудь из частных случаев данной задачи.

Упражнения 1.4

1. **а)** Воспользуйтесь тем, что массив не отсортирован.

   б) Мы использовали этот трюк при реализации одного из алгоритмов в разделе 1.1.
2. **а)** В случае отсортированного массива имеется алгоритм, о котором вы, без сомнения, уже слышали.

   б) Неуспешный поиск может быть сделан более быстрым.
3. **а)** $push(x)$ помещает x на вершину стека, а pop снимает элемент с вершины стека.

   б) $enqueue(x)$ добавляет x в хвост очереди, а $dequeue$ удаляет элемент из ее головы.
4. Воспользуйтесь определениями соответствующих свойств графа и используемой структуры данных.
5. Есть два хорошо известных алгоритма, способных решить поставленную задачу. Первый использует стек, а второй — очередь. Хотя эти алгоритмы

рассматриваются в книге немного позже, не упустите шанс разработать их самостоятельно!

6. Неравенство $h \leqslant n - 1$ следует непосредственно из определения высоты дерева. Нижняя граница неравенства следует из неравенства $2^{h+1} - 1 \geqslant n$, которое можно доказать, рассматривая наибольшее количество вершин, которое может иметь бинарное дерево высотой h.
7. Вы должны указать, как реализовать каждую из трех операций очереди с приоритетами.
8. Из-за вставок и удалений использование (отсортированного или несортированного) массива элементов словаря — не лучшее из возможных решений.
9. Для ответа на один из поставленных вопросов вам требуется знать о постфиксной записи (если вы с ней незнакомы — про*гугля*йтесь в Internet).
10. Имеется несколько алгоритмов, решающих поставленную задачу. При решении не забывайте о том, что в слове может содержаться несколько одинаковых букв.

Глава 2

Упражнения 2.1

1. Вопросы и в самом деле такие простые, какими кажутся, хотя некоторые из них могут иметь различные ответы. Кроме того, не забывайте о том, как измеряется размер целых чисел.
2. **а)** Сумма двух матриц определяется как матрица, элементы которой представляют собой суммы соответствующих элементов в матрицах-слагаемых.

   б) Умножение матриц требует применения двух операций — умножения и сложения. Какая из них должна рассматриваться как основная и почему?
3. Будет ли изменяться эффективность алгоритма для разных входных данных одного и того же размера?
4. **а)** Перчатки — не носки: они могут быть левыми и правыми.

   б) У вас есть только две качественно разные возможности. Подсчитайте количество способов получить каждую из них.
5. **а)** Сначала докажите, что если положительное десятичное целое число n имеет b цифр в двоичном представлении, то

$$2^{b-1} \leqslant n < 2^b.$$

Затем возьмите логарифм по основанию 2 от всех частей данного неравенства.

б) Формула будет точно такой же, с небольшим уточнением для учета другого основания системы счисления.

в) Каким образом перейти от логарифма по одному основанию к логарифму по другому?

6. Добавьте проверку, не отсортирован ли уже исходный массив.
7. Аналогичный вопрос был разобран в тексте раздела.
8. Воспользуйтесь разностью либо отношением $f(4n)$ и $f(n)$, что более подходит для получения компактного ответа. (Если возможно, попытайтесь получить ответ, не зависящий от n.)
9. При необходимости упростите функцию, сведя ее к одному члену, определяющему порядок роста с точностью до постоянного множителя. (Формальные методы ответа на такие вопросы мы рассмотрим в следующем разделе; однако на данный вопрос можно ответить и без знания таких методов.)
10. Воспользуйтесь формулой $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

Упражнения 2.2

1. Воспользуйтесь соответствующими количествами основных операций (см. раздел 2.1) и определением обозначений O, Θ и Ω.
2. Сначала определите порядок роста $n(n+1)/2$, а затем воспользуйтесь нестрогим определением обозначений O, Θ и Ω (подобный пример можно найти в тексте раздела).
3. Упростите данные функции до одного члена, определяющего порядок роста.
4. **а)** Внимательно просмотрите соответствующие определения.

   б) Вычислите пределы для каждой пары соседних функций в списке.
5. Сначала упростите некоторые из заданных функций. Затем воспользуйтесь списком функций в табл. 2.2 в качестве "привязки" для каждой из функций. Докажите корректность размещения функций, вычислив соответствующие пределы.
6. **а)** Вы можете доказать данное утверждение, либо вычисляя соответствующий предел, либо применяя математическую индукцию.

   б) Вычислите $\lim_{n\to\infty} a_1^n/a_2^n$.
7. Докажите корректность а), б) и в) при помощи соответствующих определений. Для г) приведите контрпример (например, путем построения двух функций, которые по-разному ведут себя для четных и нечетных значений аргументов.

8. Доказательство а) аналогично приведенному в разделе. Само собой, для ограничения функции снизу должны использоваться другие неравенства.
9. Следуйте плану анализа, использовавшемуся в тексте раздела, когда был впервые упомянут данный алгоритм.
10. Вы должны поочередно отходить от исходного положения то влево, то вправо, пока не обнаружите дверь.

Упражнения 2.3

1. Воспользуйтесь формулами и правилами, приведенными в приложении А. Возможно, вам придется выполнить простые алгебраические операции перед их применением.
2. Найдите среди сумм в приложении А суммы, выглядящие так же, как и суммы из этого упражнения, и попытайтесь привести суммы из упражнения к виду сумм из приложения А. Заметим, что для выяснения порядка роста суммы не требуется вычислять ее точное значение.
3. Просто следуйте приведенным в задании формулам.
4. **а)** Отследите выполнение алгоритма для нескольких небольших значений n (например, $n = 1, 2, 3$) — это должно помочь справиться с задачей.
   б) С этим вопросом вы уже встречались в примерах в тексте раздела.
   в) Следуйте плану, приведенному в тексте раздела.
   г) Ответ должен немедленно следовать из ответа на пункт в). Кроме того, вы можете захотеть дать ответ как функцию числа битов в двоичном представлении числа n (почему?).
   д) Вам не приходилось встречаться с этой суммой где-нибудь еще, кроме этой книги?
5. **а)** Отследите выполнение алгоритма для нескольких небольших значений n (например, $n = 1, 2, 3$) — это должно помочь справиться с задачей.
   б) С этим вопросом вы уже встречались в примерах в тексте раздела.
   в) Вы можете следовать плану, приведенному в тексте раздела, либо ответить на вопрос сразу (попробуйте оба метода).
   г) Ответ должен немедленно следовать из ответа на пункт в).
   д) Должен ли алгоритм всегда делать два сравнения на каждой итерации? Эта мысль может получить дальнейшее развитие и привести к значительному усовершенствованию алгоритма. Примените ее к двухэлементному массиву, а затем обобщите. Но можем ли мы надеяться получить алгоритм с эффективностью, превышающей линейную?

6. **а)** Элементы $A[i, j]$ и $A[j, i]$ симметричны по отношению к главной диагонали матрицы.
   б) Имеется только один кандидат.
   в) Вы можете исследовать только наихудший случай.
   г) Ответ должен немедленно следовать из ответа на пункт в).
   д) Сравните решаемую алгоритмом задачу с тем, каким образом он это делает.
7. Вычисление суммы n чисел может быть выполнено при помощи $n-1$ сложений. Сколько сложений выполняет алгоритм при вычислении каждого элемента матрицы-произведения?
8. Воспользуйтесь формулой, которая связывает n и количество битов в его двоичном представлении.
9. При доказательстве по индукции воспользуйтесь формулой

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n.$$

   Юный Гаусс заметил, что сумму $1+2+\cdots+99+100$ можно вычислить как сумму 50 пар, имеющих одинаковое значение суммы.
10. **а)** Запись сумм не должна вызывать сложностей. Однако использование стандартных формул и правил суммирования потребует бóльших усилий, чем в предыдущих примерах.
    б) Оптимизируйте внутренний цикл алгоритма.

Упражнения 2.4

1. Каждое из приведенных уравнений можно решить с помощью метода обратной подстановки.
2. Интересующее нас рекуррентное уравнение практически идентично рекуррентному соотношению для количества умножений, которое было решено в данном разделе.
3. **а)** Поставленный вопрос идентичен вопросу об эффективности рекурсивного алгоритма вычисления факториала.
   б) Напишите псевдокод нерекурсивного алгоритма и определите его эффективность.
4. **а)** Обратите внимание, что в упражнении спрашивается о рекуррентном уравнении для значений вычисляемой функции, а не для количества выполняемых операций. Для его составления следуйте представленному

в условии псевдокоду. Проще всего найти решение полученного уравнения при помощи метода обратных подстановок (см. приложение Б).

б) Этот вопрос очень похож на уже рассмотренный нами.

в) Вы можете захотеть включить операции вычитания, необходимые для уменьшения значения n.

5. а) Воспользуйтесь формулой для общего количества перемещений дисков, которая была приведена в тексте раздела.

б) Решите задачу для трех дисков, чтобы выяснить, сколько перемещений совершает каждый диск. Затем обобщите свои наблюдения и докажите их корректность для общего случая n дисков.

в) Если вы не справитесь с этим заданием, не расстраивайтесь: хотя алгоритм достаточно прост, его не так-то легко разработать. В качестве утешения — поищите решение в Web.

6. а) Рассмотрите отдельно случаи четных и нечетных значений n и покажите, что для обоих случаев $\lfloor \log_2 n \rfloor$ удовлетворяет рекуррентному уравнению и начальному условию.

б) Просто следуйте псевдокоду алгоритма.

7. а) Воспользуйтесь формулой $2^n = 2^{n-1} + 2^{n-1}$, не упрощая ее. Не забудьте об условии прекращения рекурсивных вызовов.

б) Подобный алгоритм рассматривался в разделе 2.4.

в) Подобный вопрос рассматривался в разделе 2.4.

г) Плохой класс эффективности алгоритма еще не означает, что плох сам алгоритм. Например, классический алгоритм для задачи о Ханойских башнях оптимален, несмотря на его экспоненциальную эффективность. Таким образом, утверждение о том, что конкретный алгоритм неэффективен, должно содержать указание на более эффективный алгоритм.

8. а) Вам должно помочь отслеживание работы алгоритма при $n = 1$ и $n = 2$.

б) Очень похожий вопрос обсуждался в одном из примеров в данном разделе.

9. Вам должно помочь отслеживание работы алгоритма при $n = 1$ и $n = 4$.

10. а) Воспользуйтесь формулой из определения детерминанта, чтобы получить рекуррентное уравнение для количества умножений, выполняемых алгоритмом.

б) Исследуйте правую часть рекуррентного соотношения. Вычисление нескольких первых значений $M(n)$ также может помочь вам в решении задачи.

Упражнения 2.5

2. Будет проще подставить в рекуррентное соотношение ϕ^n и $\widehat{\phi}^n$ по отдельности. Почему этого достаточно?
3. Воспользуйтесь приближенной формулой для $F(n)$, чтобы найти минимальные значения n, при которых $F(n)$ превышает заданные числа.
4. Имеется несколько способов решения этой задачи. Наиболее элегантный из них позволяет поместить эту задачу в данный раздел.
5. Постройте рекуррентные соотношения для $C(n)$ и $Z(n)$, само собой, с соответствующими начальными условиями.
6. Вся информация, необходимая для очередной итерации алгоритма, — это значения двух последних последовательных чисел Фибоначчи. Измените алгоритм так, чтобы воспользоваться данным фактом.
7. При доказательстве воспользуйтесь методом математической индукции.
8. Рассмотрите небольшой пример; скажем, вычисление $\gcd(13, 8)$.
9. **а)** Проведите доказательство методом математической индукции.
   б) Знаменитые авторы *Конкретной математики* — Р. Грэхем, Д. Кнут и О. Паташник — отмечают в своей широко известной книге по дискретной математике [45], что это была одна из любимых головоломок Льюиса Кэролла (Lewis Carroll). К сожалению, они также пишут: "*Этот парадокс объясняется тем, что... — впрочем, волшебство объяснять не принято*". Я не могу идти против таких выдающихся ученых, и должен воздержаться от подсказки.
10. Последние k цифр числа N можно получить путем вычисления $N \bmod 10^k$. Выполнение всех операций вашего алгоритма по модулю 10^k (см. приложение А) позволит справиться с экспоненциальным ростом чисел Фибоначчи. Попутно заметим, что раздел 2.6 посвящен эмпирическому анализу алгоритмов.

Упражнения 2.6

1. Будет ли получено корректное количество сравнений для всех массивов размером 2?
2. Начните с отладки счетчика сравнений и генератора массива входных данных.
3. В случае сравнительно быстрой машины вы можете получить нулевое время работы, по крайней мере для малых размеров входных данных. В разделе 2.6 рассказано, как справиться с этой неприятностью.

4. Проверьте, насколько быстро растет количество операций при удвоении размера данных.
5. Подобный вопрос рассматривался в тексте раздела.
6. Сравните значения функций $\lg \lg n$ и $\lg n$ при $n = 2^k$.
7. Вставьте счетчик числа делений в программу, реализующую алгоритм, и выполните программу для входных пар чисел из указанного диапазона.
8. Получите эмпирические данные для случайных значений n в диапазоне, скажем, от 10^2 до 10^4 или 10^5 и постройте график по полученным данным (возможно, вы захотите использовать различные масштабы по разным осям системы координат).

Глава 3

Упражнения 3.1

1. **а)** Подумайте об алгоритмах, которые поразили вас своей эффективностью и/или интеллектуальностью. Ни одна из этих характеристик не является признаком алгоритма, основанного на методе грубой силы.
   б) Интересно, что на этот вопрос ответить не так-то легко. Источником требующихся примеров задач могут послужить математические задачи, с которыми вы сталкивались на лекциях по высшей математике.
2. **а)** Ответ на первый вопрос, по сути, был дан в тексте раздела. Ответ легко выразить как функцию от числа битов, если воспользоваться формулой, связывающей две эти метрики.
   б) Как можно вычислить $(ab) \bmod m$?
3. Для начала надо выполнить указанные задания.
4. **а)** Наиболее простой алгоритм, основанный на непосредственной подстановке значения x_0 в формулу для $p(x)$, имеет квадратичное время работы.
   б) Анализ излишних вычислений в квадратичном алгоритме должен привести вас к разработке линейного алгоритма вычисления полинома.
   в) Сколько коэффициентов имеет полином степени n? Можно ли вычислить значение полинома в произвольной точке, не обращаясь ко всем им?
5. Просто пошагово пройдите алгоритм для приведенных входных данных (в начале раздела это было сделано для других входных данных).
6. Хотя большинство элементарных алгоритмов сортировки устойчивы, не спешите с ответом.

7. Вообще говоря, реализация алгоритма для связанных списков усложняется, если алгоритм требует обращения к элементам списка не в последовательном порядке.
8. Просто пошагово пройдите алгоритм для приведенных входных данных (в начале раздела это было сделано для других входных данных).
9. **а)** Список является отсортированным тогда и только тогда, когда все его соседние элементы находятся в надлежащем порядке. Почему?
   б) Добавьте логический флаг для регистрации наличия или отсутствия обменов.
   в) Начните с выяснения, какие входные данные приводят к наихудшему случаю.
10. Может ли пузырьковая сортировка изменить порядок двух одинаковых элементов во входных данных?

Упражнения 3.2

1. Модифицируйте анализ версии алгоритма в разделе 2.1.
2. Какой вид имеет функция $C_{avg}(p)$?
3. См. дельные советы в разделе 2.6.
4. Цитата Махатмы Ганди стоит того, чтобы подумать не только над упражнением, но и над ней.
5. Для каждого из входных данных одна итерация алгоритма дает всю необходимую для ответа информацию.
6. Будет достаточно, если вы ограничитесь бинарными текстами и шаблонами.
7. Измените псевдокод алгоритма поиска подстрок, приведенный в данном разделе. На сколько символов следует выполнить сдвиг после того, как искомая подстрока найдена?
8. Для нас представляется естественным выполнять проверку символов слева направо только потому, что это способ чтения и записи, принятый в европейских и ряде других языков.
9. При работе методом грубой силы вы можете блеснуть тем, что не будете проверять слова в тех направлениях, в которых букв недостаточно для размещения искомого слова.
10. Алгоритм с использованием (очень) грубой силы может просто обстреливать все клетки подряд, начав, например, с одного из углов поля. Не могли бы вы предложить лучшую стратегию? (Вы бы могли исследовать относительные эффективности разных стратегий, создав две программы, реализующие

их, и заставив их играть друг с другом.) Оказалась ли предложенная вами стратегия лучше стратегии случайного обстрела клеток поля противника?

Упражнения 3.3

1. Сортировка n действительных чисел требует $O(n \log n)$ времени.

2. **а)** Убедитесь в выполнении требований 2) и 3) при помощи основных свойств абсолютных значений.

б) При использовании манхэттенского расстояния интересующие нас точки задаются уравнением $|x-0|+|y-0|=1$. Вы можете изобразить соответствующие точки в первом квадранте, в котором все точки имеют положительные координаты ($x, y \geqslant 0$), а затем воспользоваться симметрией.

в) Предположение ложно. Возьмите две точки, $P_1(0,0)$ и $P_2(1,0)$, и найдите точку P_3, которая завершает контрпример.

3. Ваш ответ должен быть функцией двух параметров: n и k. Частный случай $k=2$ был рассмотрен в тексте раздела.

4. Обратитесь к приведенным в разделе примерам.

5. Может ли количество угловых точек выпуклой оболочки множества из n различных точек быть равным n? А нулю?

6. Найти некоторые из угловых точек выпуклой оболочки проще, чем другие.

7. Если на прямой линии, проведенной между точками P_i и P_j, располагаются и другие точки, то какие из них должны быть сохранены для последующей обработки?

8. Ваша программа должна работать с произвольным множеством из n различных точек, включая множества, в которых на одной прямой может находиться более двух точек.

9. **а)** Множество точек, удовлетворяющих неравенству $ax+by \leqslant c$, представляет собой полуплоскость, располагающуюся с одной стороны от прямой линии $ax+by=c$, включая точки на самой линии. Изобразите такие полуплоскости для каждого их неравенств и найдите их пересечение.

б) Угловые точки представляют собой вершины многоугольника, полученные в части *а)* данного упражнения.

в) Вычислите и сравните значения целевой функции в угловых точках.

Упражнения 3.4

1. **а)** Определите основную операцию алгоритма и подсчитайте количество ее выполнений.
   б) Для каждого из приведенных значений времени определите наибольшее n, для которого этот предел не будет перейден.
2. Чем задача коммивояжера отличается от задачи поиска гамильтонова цикла?
3. Ваш алгоритм должен проверять хорошо известное условие, необходимое и достаточное для существования эйлерова цикла в связанном графе.
4. Сгенерируйте оставшиеся $4! - 6 = 18$ возможных назначений, вычислите их стоимости и найдите среди полученных значений минимальное.
5. Приведите контрпример как можно меньшего размера.
6. Венгерский метод обычно рассматривается в книгах, посвященных исследованию операций. Узнайте у преподавателя, следует ли вам реализовать данный алгоритм в виде компьютерной программы как часть вашего реферата.
7. Измените условие задачи так, чтобы надо было искать сумму элементов только в одном подмножестве, а не в двух.
8. Следуйте определению клики и алгоритма на основании исчерпывающего перебора.
9. Рассмотрите все возможные перестановки данных элементов.
10. **а)** Просуммируйте все элементы магического квадрата двумя различными способами.
    б) Какие комбинаторные элементы должны быть сгенерированы при ответе на этот вопрос?

Глава 4

Упражнения 4.1

1. В нескольких аспектах этот вопрос сходен с вопросом о вычислении суммы методом декомпозиции. Подобный алгоритм рассматривался и в упражнениях 2.4.
2. В отличие от задачи 1 декомпозиционный алгоритм может быть более эффективным, чем алгоритм на основе грубой силы, за счет меньшего постоянного множителя.
3. Каким образом вы вычислите a^8 путем решения задачи о возведении в степень размера 4? А a^9?

4. Взгляните на обозначения, используемые в формулировке теоремы.
5. Примените Основную теорему.
6. Пошагово выполните алгоритм, как это было сделано для другого примера в тексте раздела.
7. Как сортировка слиянием может изменить относительный порядок двух элементов?
8. **а)** Как обычно, воспользуйтесь обратной подстановкой.
   б) Какие входные данные минимизируют количество сравнений ключей, выполняемых сортировкой слиянием? Сколько сравнений выполнит сортировка слиянием для этих данных на этапе слияния?
   в) Не забудьте включить перемещения ключей как перед разбиением, так и в процессе слияния.
10. Метод декомпозиции требует, чтобы исходный экземпляр задачи был сведен к нескольким меньшим экземплярам *той же* задачи.

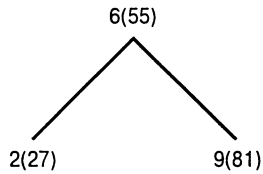
Упражнения 4.2

1. Мы уже отслеживали в этом разделе работу алгоритма для других входных данных.
2. Еще раз обратитесь к описанию процесса разбиения.
   а) Воспользуйтесь условиями прекращения сканирования.
   б) Воспользуйтесь условиями прекращения сканирования.
   в) Рассмотрите массив, состоящий из одинаковых элементов.
3. Определение *устойчивости* алгоритма сортировки было приведено в разделе 1.3. В общем случае алгоритмы, которые используют обмен далеко отстоящих друг от друга элементов, устойчивыми не являются.
4. Пошагово проследите за работой алгоритма, чтобы увидеть, для каких входных данных индекс i выходит за пределы границы.
5. Рассмотрите, как именно ведет себя быстрая сортировка с указанными массивами. Естественно, ваш ответ должен основываться на количестве выполняемых сравнений.
6. В каком месте будет выполняться разделение для описанных в упражнении входных данных?
7. Для решения этого рекуррентного соотношения следует воспользоваться стандартными приемами решения рекуррентных уравнений. Это решение можно найти в большинстве книг, посвященных разработке и анализу алгоритмов.
8. Воспользуйтесь идеей, лежащей в основе разбиения.

9. Просто выполните указанное в упражнении задание, и не забудьте, что проверку следует выполнять на входных данных в широком диапазоне их размеров.
10. Воспользуйтесь идеей, лежащей в основе разбиения.

Упражнения 4.3

1. **а)** Воспользуйтесь формулой, непосредственно дающей требуемый ответ.
   б–г) Воспользуйтесь бинарным деревом, отображающим операции алгоритма при поиске произвольного ключа. Первые три узла данного дерева для приведенного в условии экземпляра задачи будут иметь следующий вид:



   (Первое число в узле показывает индекс m элемента массива, с которым выполняется сравнение, а число в скобках указывает значение сравниваемого элемента, т.е. $A[m]$).
2. Если вам требуется освежить память, — обратитесь к разделу 2.4, где уже решались подобные уравнения, а также к приложению Б.
3. **а)** Воспользуйтесь тем фактом, что n ограничено снизу и сверху степенями двойки, т.е. $2^k \leqslant n < 2^{k+1}$.
   б) Случай четного n ($n = 2i$) рассмотрен в тексте раздела. Для случая нечетного n ($n = 2i + 1$) выполните подстановку в обе части уравнения и покажите их тождественность. При решении вам может пригодиться формула из части *а*). Не забудьте также проверить выполнение начальных условий.
4. Оцените отношение среднего количества сравнения ключей при успешном последовательном поиске и при успешном бинарном поиске.
5. Как достичь среднего элемента связанного списка?
6. Найдите отдельно элементы, не меньшие L, и элементы, не превышающие U. Не забывайте, что ни L, ни U не обязаны быть элементами массива.
7. Вам может помочь диаграмма бинарного поиска, приведенная в тексте раздела.
8. Воспользуйтесь сравнением $K \leqslant A[m]$, где $m \leftarrow \lfloor (l + r)/2 \rfloor$, пока l не равно r. После этого проверьте, найден ли искомый элемент.

9. Анализ практически идентичен анализу представленного в тексте раздела алгоритма бинарного поиска.
10. Перенумеруйте рисунки и используйте их номера в ваших вопросах.[3]

Упражнения 4.4

1. Эта задача практически идентична рассмотренной в тексте раздела.
2. Это стандартное упражнение, которое вы, должно быть, не раз выполняли в процессе изучения структур данных. Определения вариантов обхода, приведенные в конце раздела, должны помочь вам пошагово выполнить соответствующие алгоритмы, даже если вы никогда не делали этого ранее.
3. Псевдокод просто отражает описание процесса обхода.
4. Сколько раз узел вносится в стек обхода? Могут ли все узлы одновременно оказаться в стеке?
5. Сравните алгоритм и определения обходов.
6. Если вы не знаете ответа на этот важный вопрос, можете просто посмотреть на результаты различных обходов небольшого бинарного дерева поиска. Для доказательства ответьте на вопрос: что можно сказать о двух узлах с ключами k_1 и k_2, если $k_1 < k_2$?
7. Модифицируйте алгоритм вычисления высоты бинарного дерева.
8. Выполните алгоритм пошагово для небольшого входного дерева.
9. Воспользуйтесь математической индукцией (как это было сделано при доказательстве равенства $x = n + 1$ в тексте раздела).

Упражнения 4.5

1. Попробуйте сначала ответить на вопрос при $n = 2$, а затем обобщить свой ответ для произвольного n.
2. Пошагово пройдите алгоритм для приведенных в условии входных данных. Конечно, в процессе вычислений вам придется применить этот же алгоритм для вычисления произведения двузначных чисел.
3. **а)** Прологарифмируйте обе части равенства.
   б) Для чего используется точное решение рекуррентного уравнения?
4. **а)** Как выполняется умножение на степень 10?
   б) Попробуйте повторить вывод для, например, произведения $98 \cdot 76$.

[3] Можно перенумеровать столбцы и строки — это даст ту же степень эффективности решения поставленной задачи. — *Прим. ред.*

5. Подсчитайте количество сложений при умножении в столбик, например, двух четырехзначных чисел — это должно вам помочь.
6. Для проверки достаточно выполнить ряд простых алгебраических действий.
7. Пошагово пройдите алгоритм Штрассена для приведенных в условии входных данных (количество работы могло бы оказаться существенно большим, если бы рекурсию требовалось остановить не при $n = 2$, а при $n = 1$). По завершении работы было бы неплохо сравнить результат с полученным при помощи алгоритма грубой силы.
8. Примените метод обратной подстановки для решения рекуррентного соотношения из текста раздела.
9. Рекуррентное уравнение для количества умножений в алгоритме Пана подобно рекуррентному уравнению для алгоритма Штрассена. Для получения порядка роста его решения воспользуйтесь Основной теоремой.

Упражнения 4.6

1. **а)** Сколько точек требуется рассмотреть на стадии комбинации решения?
   б) Разработайте более простой алгоритм, принадлежащий тому же классу эффективности.
2. Вспомните (см. раздел 4.1), что количество сравнений, выполняемых сортировкой слиянием в наихудшем случае равно $C_{worst}(n) = n \log_2 n - n + 1$ при $n = 2^k$. В требуемом от вас рекуррентном соотношении вы можете использовать только один член наивысшего порядка из приведенной формулы.
5. Ответ на вопрос *а* следует непосредственно из учебника по геометрии.
6. Воспользуйтесь формулой, связывающей определитель с площадью треугольника.
7. Вернитесь к определению множества S_1 в описании алгоритма.
8. Естественно, она должна быть $\Omega(n)$. (Почему?)
9. Создайте последовательность n точек, для которых алгоритм при каждом рекурсивном вызове уменьшает размер задачи на единицу.

Глава 5

Упражнения 5.1

1. Решите задачу для $n = 1$.

2. Как найти наименьший элемент в массиве из n чисел, если вам известен наименьший среди его $n-1$ первых элементов? Кроме того, заметим, что почти та же задача решалась одним из алгоритмов в упражнениях к разделу 2.4.

3. Воспользуйтесь тем фактом, что все подмножества n-элементного множества $S = \{a_1, \ldots, a_n\}$ могут быть разделены на две группы — те, которые содержат элемент a_n, и те, которые его не содержат.

4. Пройдите алгоритм пошагово, как это было сделано в тексте для других входных данных (см. рис. 5.4).

5. а) Ограничитель должен предотвратить перемещение наименьшего элемента за пределы массива.

б) Примените анализ эффективности алгоритма, проведенный в тексте раздела, к версии с ограничителем.

6. Вспомните, что к элементам однократно связанного списка возможен только последовательный доступ.

7. Поскольку единственное отличие между представленными версиями алгоритмов заключается в операциях во внутреннем цикле, вы должны оценить отличие времени выполнения одной итерации внутреннего цикла обоих алгоритмов.

8. а) Ответ на вопрос для массива из трех элементов должен помочь ответить на вопрос в общем случае.

б) Положим для простоты, что все элементы различны и что вставка $A[i]$ в каждую из $i+1$ возможных позиций среди предшествующих элементов равновероятна. Вначале проанализируйте версию алгоритма с ограничителем.

9. Воспользуйтесь формулой для количества сравнений ключей алгоритмом бинарного поиска в худшем случае (раздел 4.3) и соответствующей формулой суммирования из приложения А для определения порядка роста. Для рассматриваемого алгоритма класс эффективности определяет не данная операция — а какая?

10. а) Заметим, что более удобно сортировать подфайлы параллельно, т.е. сравнивать $A[0]$ с $A[h_i]$, затем $A[1]$ с $A[1+h_i]$ и т.д.

б) Вспомним, что, вообще говоря, алгоритмы сортировки, в которых выполняется обмен далеких элементов, устойчивыми не являются.

Упражнения 5.2

1. **а)** Воспользуйтесь определениями матрицы смежности и связанных списков смежности, приведенными в разделе 1.4.
   б) Выполните поиск в глубину точно так же, как это было сделано в тексте раздела (см. рис. 5.5).
2. Сравните классы эффективности двух версий поиска в глубину для разреженного графа.
3. **а)** Чему равно количество деревьев в лесу?
   б) Сначала ответьте на этот вопрос для связного графа.
4. Выполните поиск в ширину так же, как это было сделано в тексте раздела (см. рис. 5.6).
5. Вы можете воспользоваться тем фактом, что уровень вершины в дереве поиска в ширину указывает количество ребер в кратчайшем пути от корня до этой вершины.
6. **а)** Какое свойство леса поиска в ширину указывает наличие циклов? (Ответ аналогичен ответу на этот же вопрос для поиска в глубину.)
   б) Ответ отрицательный. Найдите два примера, подтверждающие это.
7. Исходя из того факта, что при любом из поисков новая вершина может быть достигнута тогда и только тогда, когда она смежна одной из ранее посещенных вершин, какие вершины будут посещены к моменту прекращения обхода (т.е. когда стек или очередь обхода оказываются пустыми)?
8. Воспользуйтесь для решения частей *а* и *б* упражнения лесом поиска, соответственно, в глубину и в ширину.
9. Используйте в программе поиск в глубину или в ширину.
10. **а)** Строго следуйте инструкциям из условия упражнения.
    б) Использование обоих способов должно быстро привести вас к выходу из лабиринта.

Упражнения 5.3

1. Пошагово выполните алгоритм так, как это было сделано в тексте раздела для другого ориентированного графа (см. рис. 5.10).
2. **а)** Вам надо доказать два утверждения: 1) если ориентированный граф имеет ориентированный цикл, то задача топологической сортировки такого ориентированного графа не имеет решения; и 2) если ориентированный граф не содержит ориентированных циклов, то задача топологической сортировки такого ориентированного графа разрешима.
   б) Рассмотрите вырожденный случай ориентированного графа.

3. **а)** Как временна́я эффективность алгоритма топологической сортировки на основе поиска в глубину связана с временно́й эффективностью поиска в глубину?

   б) Знаете ли вы длину списка, который будет сгенерирован алгоритмом? Где вы должны разместить, например, первую снимаемую со стека обхода в глубину вершину, чтобы она находилась в своей окончательной позиции?[4]

4. Попробуйте сделать это для одного-двух небольших примеров ориентированных графов.

5. Пошагово выполните алгоритм, так, как это было сделано в тексте раздела для другого ориентированного графа (см. рис. 5.11).

6. **а)** Воспользуйтесь доказательством от противного.

   б) Если у вас возникли трудности при ответе на этот вопрос, рассмотрите пример ориентированного графа с вершиной, в которую не входит ни одно ребро, и запишите его матрицу смежности.

   в) Ответ следует из определений источника и связанных списков смежности.

7. Для каждой вершины храните количество ребер, входящих в эту вершину в остающемся после удаления источника подграфе. Используйте очередь вершин-источников.

9. **а)** Пошагово выполните алгоритм для приведенных в условии входных данных, строго следуя описанным шагам алгоритма.

   б) Определите эффективность каждого из трех основных шагов алгоритма, а затем определите его общую временну́ю эффективность. Разумеется, ответ зависит от способа представления графа — при помощи матрицы смежности или с использованием связанных списков смежности.

10. Можно воспользоваться известным алгоритмом обхода, хотя это и не самый эффективный способ решения задачи.

Упражнения 5.4

1. Воспользуйтесь стандартными формулами для количества упомянутых комбинаторных объектов. Для простоты можно считать, что генерация комбинаторного объекта занимает то же время, что и, например, присваивание.

2. Примеры работы этих алгоритмов (для множества меньшего размера) рассматривались в тексте раздела.

3. Набросок данного алгоритма приведен в тексте раздела.

[4] Подумайте также о возможном использовании двух стеков. — *Прим. ред.*

4. **а)** Пошагово пройдите алгоритм при $n = 2$; воспользуйтесь полученными результатами при проходе при $n = 3$, а затем — полученными при последнем проходе результатами при $n = 4$.

   б) Покажите, что алгоритм генерирует $n!$ перестановок и что все они различны. Воспользуйтесь методом математической индукции.

   в) Запишите рекуррентное соотношение для количества обменов, выполняемых алгоритмом. Найдите его решение и порядок роста этого решения. Вам может потребоваться формула $e \approx \sum_{i=0}^{n} (1/i!)$.

5. Примеры работы этих алгоритмов (для задач меньшего размера) рассматривались в тексте раздела.

6. После пояснений трюки становятся неинтересны...

7. Это несложное упражнение, поскольку способ получения битовых строк длиной n из битовых строк длиной $n - 1$ очевиден.

8. Вы можете имитировать бинарное сложение без явного его использования.

9. Код Грея для $n = 3$ приведен в конце раздела. Нетрудно увидеть, каким образом можно воспользоваться имеющимся кодом для генерации кода Грея при $n = 4$. Коды Грея имеют полезную геометрическую интерпретацию, основанную на отображении их битовых строк на вершины n-мерного куба. Найдите такое отображение для $n = 1$, 2 и 3. Эта геометрическая интерпретация может помочь вам при разработке алгоритма для генерации кода Грея произвольного порядка n.

10. Для решения поставленной задачи имеется несколько алгоритмов, использующих метод уменьшения размера задачи. Они несколько тоньше, чем можно было бы ожидать. Генерация комбинаций в предопределенном порядке (возрастающем, убывающем, лексикографическом) может помочь как при разработке алгоритма, так и при доказательстве корректности. Очень полезно также следующее простое свойство. Предположим (без потери общности), что исходное множество — $\{1, 2, \ldots, n\}$. Тогда имеется C_{n-i}^{k-1} k-подмножеств, наименьший элемент которых — i ($i = 1, \ldots, n - k + 1$).

Упражнения 5.5

1. Если экземпляр задачи размером n состоит в вычислении $\lfloor \log_2 n \rfloor$, то что из себя представляет экземпляр размером $n/2$? Каким соотношением связаны решения этих двух экземпляров?

2. Алгоритм, конечно, очень похож на бинарный поиск. Сколько сравнений ключей выполняется в одной итерации в наихудшем случае и какая часть массива остается для дальнейшего поиска?

3. Совершенно очевидно, как следует поступать при $n \bmod 3 = 0$ или $n \bmod 3 = 1$; но что делать при $n \bmod 3 = 2$?
4. Пошагово выполните алгоритм так же, как это было сделано в тексте раздела для других входных данных (см. рис. 5.13*б*).
5. Сколько итераций выполняет данный алгоритм?
6. Вы можете реализовать алгоритм как рекурсивно, так и нерекурсивно.
7. Скорейший путь получения решения для $n = 40$ — воспользоваться формулой, использующей бинарное представление числа n (см. конец раздела 5.5).
8. Используйте бинарное представление n.
9. **а)** Воспользуйтесь прямой подстановкой (см. приложение Б) в рекуррентное соотношение, приведенное в тексте раздела.
   б) Выявив закономерность среди 15 значений, полученных в части *а*) упражнения, запишите ее аналитически и затем докажите по методу математической индукции.
   в) Начните с бинарного представления n, и преобразуйте в бинарное представление формулу для $J(n)$, полученную в части *б*) упражнения.

Упражнения 5.6

1. **а)** Ответ следует непосредственно из формулы, лежащей в основе алгоритма Евклида.
   б) Пусть $r = m \bmod n$. Исследуйте два возможных случая величины r по отношению к величине n.
2. У вас не должно возникнуть никаких трудностей при решении этого упражнения.
3. Поскольку указанный алгоритм основан на той же идее разбиения, что и быстрый поиск, естественно ожидать, что эффективность в наихудшем случае у обоих алгоритмов окажется одинаковой.
4. Запишите уравнение прямой, проходящей через точки $(l, A[l])$ и $(r, A[r])$, и найдите координату x точки на прямой, координата y которой равна v.
5. Постройте массив, для которого интерполяция уменьшает остающийся массив на один элемент при каждой итерации.
6. **а)** Решите неравенство $\log_2 \log_2 n + 1 > 6$.
   б) Вычислите $\lim\limits_{n \to \infty} \frac{\log \log n}{\log n}$. Обратите внимание, что с точностью до постоянного множителя можно считать логарифм натуральным.
7. **а)** Определение бинарного дерева поиска непосредственно приводит к искомому алгоритму.

б) Каким будет наихудший случай у вашего алгоритма? Сколько сравнений ключей придется сделать для входных данных такого вида?

8. а) Рассмотрите по отдельности три случая: 1) узел представляет собой лист; 2) узел имеет один дочерний узел; 3) узел имеет два дочерних узла.

б) Считайте, что вам известно расположение удаляемого ключа.

9. Поиск выигрышной стратегии (если таковая существует) для $n = 1, 2, \ldots,$ 10 — один из путей получения общего решения данной задачи.

10. Не ждите подсказки — придумайте алгоритм сами. Он не обязательно должен быть оптимальным, но все же должен быть достаточно эффективным.

Глава 6

Упражнения 6.1

1. Алгоритм уже описан в условии упражнения. Его анализ аналогичен анализу алгоритмов в тексте раздела.

2. Задача подобна одной из рассматривавшихся в тексте раздела.

3. а) Сравните каждый элемент одного множества со всеми элементами второго.

б) В действительности предварительную сортировку можно использовать тремя разными путями: отсортировать элементы только одного массива, элементы обоих массивов в отдельности и элементы двух массивов вместе.

4. а) Как найти наименьшее и наибольшее значения в отсортированном списке?

б) И алгоритм грубой силы, и алгоритм декомпозиции линейны.

5. Воспользуйтесь уже известными результатами для эффективности указанных в условии алгоритмов в среднем случае.

6. Считайте, что сортировка требует около $n \log_2 n$ сравнений. Воспользуйтесь известными результатами о количестве сравнений при успешном поиске в среднем случае для алгоритма бинарного поиска и для последовательного поиска.

7. а) Поставленная задача подобна задаче из одного из упражнений к данному разделу.

б) Как бы вы решали эту задачу, если бы информация о студентах была записана на бумажных карточках? Еще лучше: представьте, что получить нужные сведения поручили человеку со здравым смыслом, но никогда не изучавшему алгоритмы.

8. **а)** Многие задачи такого вида обладают исключением для одной конфигурации точек. Что же касается единственности решения, то попробуйте рассмотреть несколько "случайных" экземпляров задачи.

   б) Постройте многоугольники для нескольких небольших "случайных" экземпляров задачи, а затем попытайтесь вывести общее правило построения вами таких многоугольников.

9. Возможно, вам поможет представление чисел в виде упорядоченных точек на числовой оси. Еще один вариант, который может оказаться полезным: рассмотрите случай $s = 0$, причем элементы массива могут быть как положительными, так и отрицательными.

10. Дважды воспользуйтесь идеей предварительной сортировки.

Упражнения 6.2

1. Пошагово выполните алгоритм так, как это было сделано в тексте раздела для другой системы линейных уравнений.

2. **а)** Воспользуйтесь результатами метода исключения Гаусса так, как пояснялось в тексте раздела.

   б) Это одна из разновидностей метода "преобразуй и властвуй". Какая именно?

3. Для поиска обратной матрицы вы можете либо решить систему линейных уравнений одновременно с тремя столбцами свободных членов, представляющих столбцы единичной матрицы размера 3×3, либо воспользоваться LU-разложением матрицы коэффициентов исходной системы линейных уравнений, найденным при решении упражнения 2.

4. Хотя конечный ответ корректен, вывод содержит ошибку, которую вам и надо найти.

5. Псевдокод этого алгоритма очень прост. Если у вас все же возникли трудности, вернитесь к пошаговому выполнению этого алгоритма в тексте раздела. Класс эффективности алгоритма можно оценить, следуя стандартному плану анализа нерекурсивных алгоритмов.

6. Оцените отношение времени работы алгоритмов с использованием приближенных формул для количества делений и умножений в обоих алгоритмах.

7. **а)** Это "нормальный" случай — одно из двух уравнений не пропорционально второму.

   б) Коэффициенты в одном уравнении должны быть такими же или пропорциональными коэффициентам второго уравнения, но для свободных членов это не должно выполняться.

в) Два уравнения должны быть одинаковы или пропорциональны друг другу (включая свободные члены).

8. а) Работайте со строками матрицы над опорной строкой точно так же, как и со строками под ней.

б) Основаны ли методы Гаусса–Джордана и Гаусса на разных способах разработки алгоритмов или на одном и том же?

в) Выведите формулу для количества умножений в методе Гаусса–Джордана так же, как это было сделано для метода исключения Гаусса в тексте раздела.

9. Сколько времени потребуется для вычисления определителя по сравнению со временем, необходимым для решения системы линейных уравнений методом исключения Гаусса?

10. а) Просто примените правило Крамера к данной системе линейных уравнений.

б) Сколько различных определителей в формулах правила Крамера?

Упражнения 6.3

1. Воспользуйтесь определением AVL-деревьев. Не забывайте, что AVL-деревья — частный случай бинарных деревьев поиска.

2. Будет проще строить деревья в порядке возрастания n.

3. Одиночный L-поворот и двойной RL-поворот представляют собой зеркальное отражение одиночного R-поворота и двойного LR-поворота, диаграммы которых имеются в тексте раздела.

4. Вставьте ключи в указанном порядке, выполняя необходимые повороты так, как это делалось в примере, приведенном в тексте раздела.

5. а) Эффективный алгоритм вытекает непосредственно из определения бинарного дерева поиска, частным случаем которого является AVL-дерево.

б) Правильный ответ противоположен тому, который первым приходит в голову.

7. а) Пошагово выполните алгоритм для указанных входных данных (см. пример на рис. 6.8).

б) Не забывайте, что количество сравнений ключей, выполняемое при поиске в 2-3-дереве, зависит не только от глубины узла, но и от того, является ли ключ первым или вторым в узле.

8. Ложно; найдите простой контрпример.

9. Где располагаются наименьший и наибольший ключи?

Упражнения 6.4

1. **а)** Пошагово выполните описанный в тексте раздела алгоритм для приведенных входных данных.
   б) Пошагово выполните описанный в тексте раздела алгоритм для приведенных входных данных.
   в) Математический факт не может быть установлен путем проверки корректности единственного примера.
2. Для пирамиды, представленной в виде массива, требуется только проверка доминирования родительских узлов.
3. **а)** Какую структуру имеет заполненное дерево высотой h с максимальным количеством узлов? А с минимальным количеством узлов?
   б) Воспользуйтесь результатами выполнения части *а*) данного упражнения.
4. Сначала выразите правую часть формулы как функцию от h. Затем докажите полученное равенство либо с использованием формулы для суммы $\sum i2^i$ из приложения А, либо с использованием математической индукции по h.
5. **а)** Где следует искать наименьший элемент пирамиды?
   б) Удалить произвольный элемент пирамиды можно при помощи обобщения алгоритма для удаления корня.
6. Пошагово выполните алгоритм для приведенных входных данных (см. пример на рис. 6.14).
7. Как правило, алгоритм сортировки, который может обменивать далеко отстоящие друг от друга элементы, устойчивым не является.
8. Ответы могут оказаться различными для разных представлений пирамиды.
10. Соберите спагетти в пучок и поставьте их на столе вертикально.

Упражнения 6.5

1. Запишите соответствующую сумму и упростите ее с использованием стандартных формул и правил для работы с суммами. Не забудьте включить в нее умножения вне внутреннего цикла.
2. Воспользуйтесь тем фактом, что значение x^i легко вычислить на основе ранее вычисленного значения x^{i-1}.
3. **а)** Используйте формулы для количества умножений (и сложений) для обоих алгоритмов.
   б) Требуется ли дополнительная память для работы схемы Горнера?
4. Примените схему Горнера для данного экземпляра так же, как это было сделано для другого экземпляра в тексте раздела.

5. Если вы эффективно реализуете алгоритм для длинного деления на $x - c$, ответ может вас удивить.
6. **а)** Пошагово выполните алгоритм бинарного возведения в степень слева направо для данного экземпляра задачи так же, как это было сделано для другого экземпляра в тексте раздела.
   б) Ответ — да. Алгоритм может быть расширен для работы с нулевой степенью. Как?
7. Пошагово выполните алгоритм бинарного возведения в степень справа налево для данного экземпляра задачи так же, как это было сделано для другого экземпляра в тексте раздела.
8. Вычисляйте и используйте биты n "на лету".
9. Используйте формулу для суммы членов этого полинома специального вида.
10. Сравните количество операций, необходимое для реализации задач, перечисленных в условии.

Упражнения 6.6

1. **а)** Воспользуйтесь правилом для вычисления $\operatorname{lcm}(m, n)$ и $\gcd(m, n)$ из разложений на простые множители m и n.
   б) Ответ следует непосредственно из формулы для вычисления $\operatorname{lcm}(m, n)$.
2. Воспользуйтесь соотношением между задачами минимизации и максимизации.
3. Докажите утверждение путем индукции по k.
4. **а)** Постройте ваш алгоритм на следующем наблюдении: граф содержит цикл длиной 3 тогда и только тогда, когда он имеет две соседние вершины i и j, соединенные также путем длиной 2.
   б) Не торопитесь с выводом, отвечая на данный вопрос.
5. Простейшее решение — привести задачу к другой, для которой известен алгоритм решения. Поскольку мы рассматривали не так уж много геометрических алгоритмов, выбрать подходящий должно быть несложно.
6. Выразите данную задачу как задачу максимизации функции одной переменной.
7. Введите двухиндексную переменную x_{ij}, указывающую назначение j-го задания i-му работнику.
8. Воспользуйтесь особенностью данной задачи, чтобы уменьшить количество неизвестных на одно.
9. Создайте новый граф.

10. **а, б)** Постройте граф пространства состояний для данной задачи, как это было сделано для задачи о переправе волка, козы и капусты.

    в) Обратите внимание на состояние после шести пересечений реки в решении части *б*).

Глава 7

Упражнения 7.1

1. Да, это возможно. Как?
2. Обратитесь к псевдокоду алгоритма и посмотрите, что он делает в случае одинаковых значений.
3. Пошагово пройдите алгоритм для данных значений (см. рис. 7.2 в качестве примера).
4. Проверьте, может ли алгоритм обратить относительный порядок равных элементов.
5. Где именно в отсортированном массиве находится элемент $A[i]$?
6. Попробуйте для начала решить задачу об украинском флаге, т.е. эффективно преобразовать массив символов Ж и С (с подобной задачей мы уже встречались в упражнении 4.2.8).
7. Воспользуйтесь стандартными обходами таких деревьев.
8. **а)** Следуйте определению массивов B и C в описании метода.

   б) Найдите, скажем, $B[C[3]]$ для примера из части *а*).
9. **а)** Воспользуйтесь для хранения ненулевых элементов матрицы связанными списками.

   б) Представьте каждый из данных полиномов связанным списком с узлами, содержащими степень i и коэффициент a_i для каждого ненулевого члена $a_i x^i$.
10. Вы можете воспользоваться симметрией доски для уменьшения позиций, которые следует хранить.

Упражнения 7.2

1. Пошагово выполните алгоритм так же, как это было сделано в тексте раздела для другого экземпляра задачи поиска подстроки.
2. Несмотря на специальный алфавит, это приложение алгоритма ничем не отличается от приложения к строкам на естественном языке.

3. Для каждого образца заполните таблицу сдвигов и определите количество сравнений символов (как удачных, так и неуспешных) при каждой попытке, а также количество выполненных попыток.
4. Найдите пример бинарной строки длиной m и бинарной строки длиной n ($n \geqslant m$) таких, что алгоритм Хорспула выполняет
   а) наибольшее возможное количество сравнений символов перед выполнением минимального возможного сдвига;
   б) наименьшее возможное количество сравнений символов.
5. Логично рассмотреть наихудший случай для алгоритма Хорспула.
6. Может ли алгоритм сдвинуть образец более чем на одну позицию без возможного пропуска другого вхождения искомой подстроки?
7. Для каждого образца заполните две таблицы сдвигов и определите количество сравнений символов (как удачных, так и неуспешных) при каждой попытке, а также количество выполненных попыток.
8. Обратитесь к описанию алгоритма Бойера–Мура.
9. Обратитесь к описаниям алгоритмов.

Упражнения 7.3

1. Примените схему открытого хеширования (с раздельными цепочками) для указанных входных данных, как это было сделано в главе для других входных данных (см. рис. 7.5). Затем вычислите наибольшее и среднее количества сравнений для успешного поиска в построенной таблице.
2. Примените схему закрытого хеширования (с открытой адресацией) для указанных входных данных, как это было сделано в главе для других входных данных (см. рис. 7.6). Затем вычислите наибольшее и среднее количества сравнений для успешного поиска в построенной таблице.
3. Сколько различных адресов может выдать такая хеш-функция? Будет ли распределение ключей равномерным?
4. Вопрос подобен вопросу о вычислении вероятности получения одного и того же результата при n бросках кости в честной игре.
5. Найдите вероятность того, что n человек имеют различные дни рождения. Что касается хеширования, то какое явление в нем связано с совпадениями?
6. а) Нет необходимости вставлять новый ключ в конец связанного списка, в который он хеширован.
   б) Какая операция выполняется быстрее с отсортированным связанным списком и почему? В случае сортировки должны ли мы копировать все элементы из непустых списков в массив и затем применять один из уни-

версальных алгоритмов сортировки, или имеется способ воспользоваться отсортированностью ключей в непустых связанных списках?

7. После того как вы ответите на поставленные в этом упражнении вопросы, сравните эффективность вашего алгоритма с алгоритмом на основе грубой силы (раздел 2.3) и алгоритма с использованием предварительной сортировки (раздел 6.1).
8. Рассматривайте этот вопрос как небольшое повторение пройденного материала: ответы для последних двух столбцов содержатся в разделе 7.3, для остальных столбцов вы найдете ответы в соответствующих разделах книги (само собой, для ответов вы должны использовать наилучшие из имеющихся алгоритмов).
9. Если вам надо обновить память, просто загляните в оглавление книги.

Упражнения 7.4

1. Размышления о поиске информации должны натолкнуть вас на множество примеров.
2. **а)** Воспользуйтесь стандартными правилами работы с суммами, в частности формулой суммы геометрической прогрессии.
   б) Вам надо применить логарифмирование по основанию $\lceil m/2 \rceil$.
3. Найдите искомое значение из неравенства для верхней границы высоты B-дерева, приведенного в тексте раздела.
4. Следуйте алгоритму вставки, описанному в разделе.
5. Алгоритм следует из определения B-дерева.
6. **а)** Просто следуйте описанию алгоритма, данному в условии задачи. Обратите внимание, что ключ всегда вставляется в лист и что полные узлы всегда разбиваются по пути вниз, несмотря на то, что лист для нового ключа может иметь достаточно места для его размещения.
   б) Может ли разделение полного узла вызвать каскадное разделение по цепочке предков? Можем ли мы получить более высокое дерево, чем это необходимо?

Глава 8

Упражнения 8.1

1. Сравните определения двух методов.

2. а) Пошагово выполните алгоритм *Binomial* для $n = 6$ и $k = 3$ и заполните таблицу, аналогичную таблице на рис. 8.1.

б) Обратитесь к формуле алгоритма еще раз и посмотрите, какие значения требуются для вычисления C_n^k.

3. Покажите, что существуют положительные константы c_1 и c_2 и натуральное число n_0 такие, что для всех пар целых чисел n и k ($n \geqslant n_0$ и $0 \leqslant k \leqslant n$)

$$c_2 nk \leqslant \frac{(k-1)\,k}{2} + k\,(n-k) \leqslant c_1 nk.$$

4. а) Вычислить пространственную эффективность алгоритма можно тем же способом, как и временну́ю эффективность в разделе 8.1.

б) С определенными предосторожностями новую строку таблицы можно записать поверх предыдущей.

5. Воспользуйтесь явной формулой для C_n^k и при необходимости формулой Стирлинга.

6. Запишите рекуррентное соотношение для количества сложений $A(n, k)$ и решите его подстановкой $A(n, k) = B(n, k) - 1$.

7. Найдите и сравните классы эффективности указанных алгоритмов.

8. Формулу можно доказать либо с использованием явной формулы для C_n^k, либо с применением комбинаторной интерпретации этой величины.

9. а) В ситуации, когда командам A и B требуется для победы в серии i и j игр, соответственно, рассмотрите результат победы в игре команды A и результат ее поражения.

б) Заполните таблицу из 5 строк ($0 \leqslant i \leqslant 4$) и 5 столбцов ($0 \leqslant j \leqslant 4$) с использованием рекуррентного соотношения из части *а)* упражнения.

в) Псевдокод должен руководствоваться рекуррентным соотношением из части *а)* упражнения. Ответ об эффективности должен следовать непосредственно из размера таблицы и времени, затраченного на вычисление каждого из ее элементов.

Упражнения 8.2

1. Примените алгоритм к матрице смежности так, как это делалось в тексте раздела для другой матрицы.

2. а) Ответ может быть получен либо путем рассмотрения количества вычисляемых алгоритмом значений, либо при помощи стандартного плана для анализа нерекурсивных алгоритмов (т.е. путем записи суммы для количества выполнений базовых операций).

б) К какому классу эффективности принадлежит алгоритм на основе обхода для разреженных графов, представленных при помощи связанных списков смежности?

3. Покажите, что можно просто перезаписывать элементы $R^{(k-1)}$ элементами $R^{(k)}$ без каких-либо изменений в алгоритме.

4. Что будет, если $R^{(k-1)}[i,k] = 0$?

5. Сначала покажите, что формула (8.7) (из которой можно удалить верхние индексы в соответствии с упражнением 3)

$$r_{ij} = r_{ij} \textbf{ or } r_{ik} \textbf{ and } r_{kj}$$

эквивалентна формуле

$$\textbf{if } r_{ik} \quad r_{ij} \leftarrow r_{ij} \textbf{ or } r_{kj}.$$

6. а) Какое свойство транзитивного замыкания указывает наличие ориентированного цикла? Имеется ли лучший алгоритм для проверки этого факта?

б) Какие элементы транзитивного замыкания неориентированного графа равны 1? Можно ли найти такие элементы при помощи более быстрого алгоритма?

7. См. пример применения алгоритма к другому экземпляру задачи в тексте раздела.

8. От каких элементов матрицы $D^{(k-1)}$ зависит элемент $d_{ij}^{(k)}$ на пересечении i-ой строки и j-го столбца матрицы $D^{(k)}$? Могут ли эти значения быть изменены при перезаписывании?

9. Ваш контрпример должен содержать цикл отрицательной длины.

10. Достаточно хранить в отдельной матрице P индексы промежуточных вершин, использованных при обновлении матрицы расстояний. Такую матрицу можно инициализировать, скажем, значениями -1.

Упражнения 8.3

1. Продолжите применение формулы (8.11), как предписывает алгоритм.

2. а) Временну́ю эффективность алгоритма можно найти в соответствии со стандартным планом анализа временно́й эффективности нерекурсивного алгоритма.

б) Сколько памяти требуется для двух таблиц, генерируемых алгоритмом?

3. $k = R[1, n]$ указывает, что корень оптимального дерева представляет собой k-ый ключ в списке упорядоченных ключей $a_1, a_2, \ldots, a_n$. Корни его левого и правого поддеревьев определяются как $R[1, k-1]$ и $R[k+1, n]$, соответственно.

4. Воспользуйтесь методом пространственно-временного компромисса.

5. Если предложенное утверждение истинно, то разве это не означает наличия более простого алгоритма построения оптимального бинарного дерева поиска?

6. Структура дерева должна просто минимизировать среднюю глубину его узлов. Не забудьте указать способ распределения ключей по узлам дерева.

7. **а)** Поскольку имеется однозначное соответствие между бинарными деревьями поиска для данного множества из n упорядочиваемых ключей и бинарными деревьями с n узлами (почему?), вы можете пересчитать последние. Рассмотрите все возможные разделения узлов между левым и правым поддеревьями.

б) Вычислите интересующие вас значения с использованием двух формул.

в) Воспользуйтесь формулой для n-го числа Каталана и формулой Стирлинга для $n!$.

8. Измените границы внутреннего цикла из алгоритма $OptimalBST$, воспользовавшись монотонностью таблицы корней, о которой упоминается в конце раздела 8.3.

9. Предположим, что $a_1, \ldots, a_n$ — различные ключи, упорядоченные от наименьшего к наибольшему; $p_1, \ldots, p_n$ — вероятности их поиска, а $q_0, q_1, \ldots, q_n$ — вероятности неудачных поисков ключей в интервалах $(-\infty, a_1)$, (a_1, a_2), $\ldots$, (a_n, ∞), соответственно; $(p_1 + \cdots + p_n) + (q_0 + \cdots + q_n) = 1$. Запишите рекуррентное соотношение, аналогичное соотношению (8.11), для ожидаемого количества сравнений ключей с учетом как успешных, так и неудачных поисков.

10. **а)** Проще всего найти общую формулу для количества умножений, необходимых для вычислений $(A_1 \cdot A_2) \cdot A_3$ и $A_1 \cdot (A_2 \cdot A_3)$ для матриц A_1 размером $d_0 \times d_1$, A_2 размером $d_1 \times d_2$ и A_3 размером $d_2 \times d_3$, а затем выбрать конкретные размеры так, чтобы одно из значений как минимум в 1000 раз превышало другое.

б) Ответ можно получить, следуя подходу, использовавшемуся для подсчета количества бинарных деревьев.

в) Рекуррентное отношение для оптимального количества умножений при вычислении $A_i \cdot \ldots \cdot A_j$ очень похоже на рекуррентное соотношение для оптимального количества сравнений в бинарном дереве поиска, составленном из ключей $a_i, \ldots, a_j$.

Упражнения 8.4

1. **а)** Воспользуйтесь формулами (8.12) и (8.13) для заполнения соответствующей таблицы, как это было сделано для другого экземпляра задачи в тексте раздела.
   б) Что означает равенство членов в формуле
   $$\max\{V[i-1,j], v_i + V[i-1, j-w_i]\}?$$
2. **а)** Напишите псевдокод для заполнения таблицы на рис. 8.12 (скажем, строка за строкой) с использованием формул (8.12) и (8.13).
   б) Алгоритм для определения состава оптимального подмножества описан в разделе на конкретном примере.
3. Сколько значений должен вычислить алгоритм? Сколько времени занимает вычисление одного значения? Сколько ячеек таблицы следует обойти для того, чтобы определить состав оптимального подмножества?
4. Воспользуйтесь определением $V[i,j]$ для проверки истинности
   а) $V[i,j-1] \leqslant V[i,j]$ для $1 \leqslant j \leqslant W$;
   б) $V[i-1,j] \leqslant V[i,j]$ для $1 \leqslant i \leqslant n$.
5. Пройдите пошагово вызов функции *MFKnapsack* (i,j) для указанного экземпляра задачи (применение этой функции к другому экземпляру задачи можно найти в тексте раздела).
6. Алгоритм использует формулу (8.12) для заполнения *некоторых* ячеек таблицы. Почему мы можем утверждать, что его эффективность остается принадлежащей классу $\Theta(nW)$?
7. Модифицируйте восходящий алгоритм из раздела 8.3 так же, как был модифицирован восходящий алгоритм в разделе 8.4.
8. Одна из причин связана со временно́й эффективностью, а вторая — с эффективностью пространственной.
9. Запишите рекуррентное соотношение для $C[i,j]$, наименьшего количества монет в экземпляре задачи при использовании монет первых i достоинств ($1 \leqslant i \leqslant m$) и сумме j ($0 \leqslant j \leqslant n$). Для экземпляра задачи без решения используйте в качестве значения $C[i,j]$ величину $+\infty$.

Глава 9

Упражнения 9.1

1. В качестве контрпримера можете использовать набор номиналов, упомянутый в тексте: $d_1 = 7$, $d_2 = 5$, $d_3 = 1$.

2. В своем алгоритме вы можете использовать деление нацело.
3. Вам может помочь рассмотрение случая двух заданий. Конечно, сформулировав гипотезу, вы должны либо доказать оптимальность выхода для произвольных входных данных, либо найти контрпример, показывающий, что это не так.
4. Вы можете применить жадный подход либо ко всей матрице стоимости, либо к каждой из ее строк (или столбцов).
5. В обоих версиях задачи несложно выдвинуть гипотезу о виде решения после рассмотрения случаев $n = 1, 2$ и 3. Главным в задаче является доказательство корректности ее решения.
6. **а)** Пошагово выполните алгоритм для данного в условии графа. Пример выполнения можно найти в тексте раздела.
   б) После того как очередная пограничная вершина добавляется в дерево, в очередь с приоритетами следует добавить все незамеченные вершины, смежные с данной.
7. Применение алгоритма Прима ко взвешенному несвязному графу должно помочь вам ответить на поставленный вопрос.
8. **а)** Поскольку алгоритм Прима требует наличия весов у ребер графа, им следует назначить некоторые веса.
   б) Знаете ли вы другой алгоритм, способный справиться с этой задачей?
9. Строго говоря, формулировка задачи требует доказать две вещи: что существует по крайней мере одно минимальное остовное дерево для любого взвешенного связного графа и что минимальное остовное дерево единственно, если все веса различны. Доказательство первого утверждения вытекает из очевидного наблюдения о конечности количества остовных деревьев у взвешенного связанного графа. Доказательство второго утверждения может быть получено путем повторения доказательства корректности алгоритма Прима с небольшим изменением в конце.
10. Рассмотрите два случая: уменьшение значения ключа (этот случай реализуется в алгоритме Прима) и увеличение значения ключа.

Упражнения 9.2

1. Пошагово выполните алгоритм для данных графов так же, как это было сделано для другого графа в тексте раздела.
2. Два из четырех утверждений истинны, два — ложны.
3. Примените алгоритм Крускала к несвязному графу; это должно помочь вам ответить на поставленный вопрос.

4. Ответ один и тот же для обоих алгоритмов. Если вы считаете, что алгоритмы будут корректно работать для графов с отрицательными весами, докажите это; если нет, приведите контрпримеры для каждого алгоритма.
5. Применим ли здесь метод преобразования минимизации в задачу максимизации из раздела 6.6?
6. Подставьте три операции над абстрактным типом данных непересекающихся подмножеств — $makeset(x)$, $find(x)$ и $union(x, y)$ — в соответствующие места псевдокода, приведенного в тексте раздела.
7. Следуйте плану, использованному в разделе 9.1 для доказательства корректности алгоритма Прима.
8. Доказательство очень похоже на доказательство, выполненное в тексте для версии *union-by-size* быстрого поиска.
9. Вы можете воспользоваться списком желательных характеристик визуализаций алгоритмов, приведенном в разделе 2.7.

Упражнения 9.3

1. Один из вопросов не требует внесения изменений в алгоритм или граф; остальные требуют простых модификаций.
2. Пошагово выполните алгоритм для данных графов так же, как это было сделано для другого графа в тексте раздела.
3. В таком графе ближайшая вершина не обязательно должна быть смежной с исходной.
4. Верно только одно из утверждений. Найдите небольшой контрпример для второго.
5. Упростите приведенный в тексте раздела псевдокод, реализуя очередь с приоритетами в виде неупорядоченного массива и игнорируя метки вершин, указывающие родительские узлы.
6. Докажите корректность по индукции по количеству вершин, включенных в дерево, строящееся алгоритмом.
7. Сначала топологически отсортируйте вершины ориентированного ациклического графа.
8. Сначала топологически отсортируйте вершины ориентированного ациклического графа.
9. Воспользуйтесь преимуществами геометрического и физического мышления.
10. Перед тем как приступать к реализации алгоритма поиска кратчайших путей, решите, какой критерий определяет "наилучший маршрут". Желательно,

чтобы ваша программа спрашивала пользователя, какой из нескольких критериев оптимальности следует применить.

Упражнения 9.4

1. См. пример, приведенный в тексте раздела.
2. После объединения двух узлов с наименьшими вероятностями разрешить возникающую неоднозначность можно двумя путями. Для каждого из двух полученных кодов Хаффмана вычислите среднее и дисперсию длин кодов.
3. Вы можете основываться при ответе на методе работы алгоритма Хаффмана или на том факте, что коды Хаффмана являются оптимальными префиксными кодами.
4. Максимальная длина кода очевидным образом связана с высотой дерева Хаффмана. Попытайтесь найти множество n конкретных частот для алфавита из n символов, для которых дерево имеет вид, приводящий к максимально возможной длине кода.
5. **а)** Какая структура данных наиболее подходит для реализации алгоритма, основной операцией которого является поиск двух минимальных элементов данного множества и их замена их суммой?

   б) Определите базовую операцию алгоритма, количество ее выполнений и эффективность для выбранной структуры данных.
6. Поддерживайте две очереди: одну для заданных частот, вторую — для весов новых деревьев.
7. Естественно было бы использовать один из алгоритмов обхода дерева.
8. Генерируйте коды справа налево.
10. Аналогичный пример рассматривался в конце раздела 9.4. Постройте дерево Хаффмана, а затем подумайте над вопросами, которые должно давать такое дерево (учтите, что можно, например, спросить: "Это единица или семерка, или восьмерка?").

Глава 10

Упражнения 10.1

1. Поскольку вы знаете, что количество перемещений дисков, выполняемых классическим алгоритмом, равно $2^n - 1$, то можете доказать (например, при помощи математической индукции), что для любого алгоритма, решающего эту задачу, количество перемещений дисков $M(n)$, выполняемое этим алгоритмом, не менее $2^n - 1$. В качестве альтернативного метода можете показать,

что если $M^*(n)$ — минимальное количество перемещений дисков, то $M^*(n)$ удовлетворяет рекуррентному соотношению

$$M^*(n) = 2M^*(n) + 1 \text{ при } n > 1 \text{ и } M^*(1) = 1,$$

решение которого — $2^n - 1$.

2. Все эти вопросы имеют достаточно простые ответы. Если тривиальная нижняя граница плотная, не забудьте указать конкретный алгоритм, который доказывает ее плотность.
3. Вернитесь к разделу 5.5, где вы впервые встретились с задачей о фальшивой монете; это должно помочь в решении поставленной задачи.
4. Подумайте о числах, проигрывающих при сравнении.
5. Разделите множество вершин графа на два непересекающихся подмножества U и W, имеющих по $\lfloor n/2 \rfloor$ и $\lceil n/2 \rceil$ вершин, соответственно, и покажите, что любой алгоритм будет проверять наличие ребер между всеми парами вершин (u, w), где $u \in U$ и $w \in W$ перед тем как установит связность графа.
6. Вопрос и ответ очень похожи на случай двух n-элементных отсортированных списков, рассмотренный в тексте раздела. Как и доказательство нижней границы.
7. Просто следуйте формулам преобразования, предложенным в тексте раздела.
8. **а)** Проверьте, выполняются ли формулы для двух произвольных квадратных матриц.
   б) Воспользуйтесь формулой, аналогичной формуле в тексте раздела, и покажите, что умножение произвольных квадратных матриц может быть приведено к возведению в квадрат большей матрицы.
9. Какая задача с известной нижней границей наиболее подобна поставленной в упражнении? После того, как вы найдете подходящее приведение, не забудьте указать алгоритм, который делает нижнюю границу плотной.
10. Вам может помочь рассмотрение частного случая $n = 1$.

Упражнения 10.2

1. **а)** Сначала докажите, что $2^h \geqslant l$ при помощи индукции по h.
   б) Сначала докажите, что $3^h \geqslant l$ при помощи индукции по h.
2. **а)** Сколько всего выходов имеет данная задача?
   б) Естественно, имеется много способов решения этой простой задачи.
   в) Вам должно помочь представление a, b и c как точек на числовой оси.

3. Это очень простой вопрос. Вы можете считать, что все три сортируемых элемента различны. (Если вам нужна помощь, обратитесь к деревьям принятия решения для трехэлементной сортировки выбором и вставкой в тексте раздела.)
4. Вычислите нетривиальную нижнюю границу для сортировки четырехэлементного массива, а затем определите, у какого алгоритма количество сравнений в наихудшем случае соответствует этой нижней границе.
5. Это не такая уж простая задача. Ни один из стандартных алгоритмов сортировки не способен на это. Попытайтесь разработать специальный алгоритм, который получает из каждого сравнения как можно больше информации.
6. Это очень простое упражнение. Воспользуйтесь тем очевидным фактом, что последовательный поиск в отсортированном массиве может прекращаться по достижении элемента, большего, чем искомый ключ.
7. **а)** Начните с приведения логарифмов к одному основанию.
   б) Простейший способ состоит в доказательстве того, что

$$\lim_{n\to\infty} \frac{\lceil \log_2 (n+1) \rceil}{\lceil \log_3 (2n+1) \rceil} > 1.$$

   Чтобы избавиться от функции "потолок", можно воспользоваться тем, что

$$\frac{f(n)-1}{g(n)+1} < \frac{\lceil f(n) \rceil}{\lceil g(n) \rceil} < \frac{f(n)+1}{g(n)-1},$$

   где $f(n) = \log_2 (n+1)$ и $g(n) = \log_3 (2n+1)$, и показать, что

$$\lim_{n\to\infty} \frac{f(n)-1}{g(n)+1} = \lim_{n\to\infty} \frac{f(n)+1}{g(n)-1} > 1.$$

8. **а)** Сколько выходов имеет данная задача?
   б) Изобразите тернарное дерево принятия решения для данной задачи.
9. **а)** Покажите, что каждый из двух случаев — взвешивания двух монет (по одной на каждой чашке) или четырех монет (по две на каждой чашке) — приводит как минимум к одной ситуации с более чем тремя возможными выходами. Получающаяся ситуация не может быть однозначно разрешена при помощи единственного взвешивания.[5]
   б) Сначала решите, следует ли начинать со взвешивания двух монет. Не забудьте, что у вас есть подлинная дополнительная монета.
   в) Это знаменитая головоломка. Главное для ее решения — разобраться с частью *б)* упражнения.

[5] Такой информационно-теоретический подход был предложен Брассардом (Brassard) и Брейтли (Bratley) [22].

10. а) Подумайте о проигравших.

б) Рассмотрите высоту турнирного дерева (или количество шагов, необходимых для приведения n-элементного множества к одноэлементному путем деления пополам).

в) После того как определен победитель, какой игрок оказывается на втором месте?

Упражнения 10.3

1. Обратитесь к определению разрешимой задачи принятия решения.

2. Сначала выясните, является ли $n^{\log_2 n}$ полиномиальной функцией. Затем внимательно прочтите определения легких и трудных задач.

3. Все четыре комбинации возможны, и для каждой имеются примеры небольшого размера.

4. Просто воспользуйтесь определением хроматического числа. Решение упражнения 5 может помочь вам (хотя и не обязательно).

5. Эта задача должна быть вам уже знакома.

6. Что именно является мерой размера входных данных для рассматриваемой задачи?

7. Обратитесь к формулировке версии принятия решения задачи о раскраске графа и алгоритму верификации для задачи о гамильтоновом цикле.

8. Можно начать с выражения задачи о разделении в виде линейного уравнения с переменными x_i, $i = 1, \ldots, n$, которые могут принимать только значения 0 и 1.

9. Если вы не знакомы с понятиями клики, вершинного покрытия и независимого множества, неплохо начать с поиска клики максимального размера, вершинного покрытия минимального размера и независимого множества максимального размера для некоторых простых графов, наподобие рассматриваемых в упражнении 4. Затем попытайтесь найти связь между этими тремя понятиями. Вам может помочь рассмотрение *дополнения* вашего графа, которое представляет собой граф с теми же вершинами и ребрами между теми вершинами, которые не являются смежными в исходном графе.

10. Современным представлениям о классах сложности не противоречат две из приведенных диаграмм.

11. Необходимая вам задача явно упоминается в тексте раздела 10.3.

Упражнения 10.4

1. Вычислите относительные ошибки приближений и воспользуйтесь определением числа значащих цифр в приближении. Один из ответов не согласуется с нашим интуитивным представлением о данном понятии.
2. Воспользуйтесь определением абсолютной и относительной ошибок и свойствами модуля.
3. Вычислите значение $\sum_{i=0}^{5} 0.5^i/i!$ и величину разности между полученным значением и $\sqrt{e} = 1.648721\ldots$.
4. Примените формулу площади трапеции к каждой из трапеций и вычислите сумму полученных площадей.
5. Примените формулы (10.7) и (10.9) к данным интегралам.
6. Найдите верхнюю границу второй производной $e^{\sin x}$ и воспользуйтесь формулой (10.9) для поиска значения n, гарантирующего ошибку усечения меньше указанного предела.
7. Аналогичная задача была разобрана в тексте раздела.
8. Рассмотрите все возможные значения параметров a, b и c. Не забывайте о том, что решение уравнения состоит в поиске его корней или доказательстве их отсутствия.
9. **а)** Докажите, что каждый элемент x_n последовательности 1) положителен, 2) больше $\sqrt{D}$ (вычисляя $x_{n+1} - \sqrt{D}$) и 3) меньше предыдущего (вычисляя $x_{n+1} - x_n$). Затем найдите предел каждой из частей равенства (10.15) при n, стремящемся к бесконечности.

   б) Воспользуйтесь уравнением $x_{n+1} - \sqrt{D} = \frac{(x_n - \sqrt{D})^2}{2x_n}$.
10. Это было сделано в тексте раздела для $\sqrt{2}$.

Глава 11

Упражнения 11.1

1. **а)** Продолжите алгоритм путем возврата из листа, соответствующего первому решению.

   б) Как можно получить второе решение из первого, используя симметрию доски?
2. Подумайте о применении поиска с возвратом в обратном направлении.
3. **а)** Воспользуйтесь общим шаблоном алгоритма поиска с возвратом. Вы должны найти способ определить, имеются ли при данной расстановке два ферзя, угрожающие друг другу.

Чтобы упростить сравнение с алгоритмом исчерпывающего поиска, вы можете рассмотреть версию, которая находит все решения задачи, не пользуясь симметрией доски. Заметьте также, что алгоритм исчерпывающего перебора может либо рассматривать все возможные размещения n ферзей в n различных клетках на доске размером $n \times n$, либо размещение ферзей только на различных горизонталях (или вертикалях), или только на различных вертикалях и различных горизонталях.

б) Алгоритм для оценки размера дерева можно получить из алгоритма для решения задачи методом поиска с возвратом путем внесения небольших изменений. Хотя более поучительно посмотреть, насколько точна такая оценка для одного случайного пути, вы можете захотеть вычислить среднее значение для нескольких путей, чтобы получить более точную оценку размера дерева.

4. В тексте раздела был решен другой экземпляр этой задачи.

5. Заметим, что без потери общности можно считать, что вершина a окрашена цветом 1, и связать эту информацию с корнем дерева пространства состояний.

6. Такое применение поиска с возвратом достаточно тривиально.

7. а) В тексте раздела был решен другой экземпляр этой задачи.

б) Некоторые из узлов будут рассматриваться как обещающие, в то время как они не являются таковыми.

8. Для выполнения задания требуется внести в шаблон минимальное изменение.

10. Не забудьте о симметрии доски при решении этой задачи.

Упражнения 11.2

1. Какие операции алгоритм ветвей и границ с выбором наилучшего варианта выполняет над живыми узлами в дереве пространства состояний?

2. Воспользуйтесь для вычисления нижних границ наименьшими числами, выбранными из столбцов матрицы стоимости. При применении такой функции для вычисления границ логичнее рассматривать четыре варианта назначения задания 1 в качестве узлов первого уровня дерева пространства состояний.

3. а) Ваш ответ должен быть матрицей размером $n \times n$ с простой структурой, которая приводит к максимально быстрой работе алгоритма.

б) Набросайте структуру дерева пространства состояний для вашего ответа к части *а)* упражнения.

5. Аналогичная задача была решена в тексте раздела.

6. Примите во внимание более одного предмета из тех, которые не включены в рассматриваемое подмножество.
8. Гамильтонов цикл для каждой вершины графа должен иметь ровно два инцидентных ребра.
9. Аналогичная задача была решена в тексте раздела.

Упражнения 11.3

1. **а)** Начните с маркировки первого столбца матрицы и поиска наименьшего элемента в первой строке и немаркированном столбце.
   б) Вы должны найти оптимальное решение путем исчерпывающего перебора или при помощи алгоритма ветвей и границ (или каким-то иным методом).
2. **а)** Простейший подход заключается в маркировке столбцов, соответствующих посещенным городам. В качестве альтернативы можно использовать связанный список непосещенных городов.
   б) Следование стандартному плану анализа эффективности алгоритма не должно вызвать сложностей (и привести к одному и тому же результату для обоих вариантов, упомянутых в указании к части *а* упражнения).
3. Двигайтесь по часовой стрелке.
4. Распространите неравенство треугольника на случай $k \geqslant 1$ промежуточных вершин и докажите его корректность методом математической индукции.
5. Сначала определите временну́ю эффективность каждого из трех шагов алгоритма.
6. Вы должны доказать два факта.
   1) $f(s^*) \leqslant 2f(s_a)$ для любого экземпляра задачи о рюкзаке, где $f(s_a)$ — значение приближенного решения, полученное усовершенствованным жадным алгоритмом, а $f(s^*)$ — оптимальное значение точного решения того же экземпляра задачи.
   2) 2 — наименьшая константа, для которой справедливо это утверждение.
   Для доказательства 1) воспользуйтесь значением оптимального решения непрерывной версии задачи и ее связью со значением приближенного решения. Для доказательства 2) найдите семейство трехпредметных экземпляров, которое доказывает данное утверждение (два предмета могут иметь вес $W/2$, а третий может иметь вес немного больший, чем $W/2$).

7. а) Пошагово пройдите алгоритм для данного экземпляра, а затем ответьте на вопрос, можно ли разместить те же предметы в меньшем количестве корзин.

б) Какова базовая операция данного алгоритма? Какие входные данные заставляют алгоритм дольше работать?

в) Сначала докажите неравенство

$$B_{FF} < 2\sum_{i=1}^{n} s_i \text{ для любого экземпляра с } B_{FF} > 1,$$

где B_{FF} — количество корзин, полученных при применении алгоритма первого подходящего к экземпляру с размерами $s_1, s_2, \ldots, s_n$. Чтобы доказать это, воспользуйтесь тем фактом, что может быть не более одной корзины, заполненной наполовину или менее.

8. а) Пошагово пройдите алгоритм для данного экземпляра, а затем ответьте на вопрос, можно ли разместить те же предметы в меньшем количестве корзин.

б) На вопрос можно ответить либо при помощи теоретического доказательства, либо при помощи контрпримера.

в) Воспользуйтесь двумя следующими свойствами.

1) Все предметы, размещаемые алгоритмом первого подходящего с убыванием в дополнительные корзины, т.е. корзины после первых B^*, имеют размер не более $1/3$.

2) Общее количество предметов, размещенных в дополнительных корзинах, не превышает $B^* - 1$ (B^* — оптимальное количество корзин).

г) Эта задача имеет две версии с существенно разными уровнями сложности. Какие это версии?

9. а) Один такой алгоритм основан на идее, подобной использованной в алгоритме поиска транзитивного замыкания, с тем отличием, что он начинает работу с произвольного ребра графа.

б) Вспомните предупреждение о том, что эквивалентность сложности полиномиально приводимых NP-сложных задач не распространяется на приближенные алгоритмы.

10. а) Раскрашивайте вершины, не используя новые цвета без крайней необходимости.

б) Найдите последовательность графов G_n, для которых отношение

$$\frac{\kappa_a(G_n)}{\kappa^*(G_n)}$$

(где $\kappa_a(G_n)$ и $\kappa^*(G_n)$ — соответственно, количество цветов, полученное жадным алгоритмом, и оптимальное количество цветов) может быть сделано произвольно большим.

Упражнения 11.4

1. В ваших поисках может помочь информация о том, что это решение было впервые опубликовано итальянским математиком эпохи Возрождения Джироламо Кардано (Girolamo Cardano).
2. На эти вопросы можно ответить без применения дифференциального исчисления или сложных вычислений, просто представив уравнения в виде $f_1(x) = f_2(x)$ и нарисовав графики $f_1(x)$ и $f_2(x)$.
3. **а)** Воспользуйтесь свойством, лежащим в основе метода деления пополам.
   б) Воспользуйтесь определением деления полинома $p(x)$ на $x - x_0$, т.е. уравнением $p(x) = q(x)(x - x_0) + r$, где x_0 — корень полинома $p(x)$, а $q(x)$ и r — соответственно, частное и остаток.
   в) Продифференцируйте обе стороны уравнения из части *б* и подставьте x_0 в полученный результат.
4. Используйте тот факт, что $|x_n - x^*|$ — расстояние между x_n, срединой отрезка $[a_n, b_n]$, и корнем x^*.
5. Набросайте график для определения общего положения корня и выберите подходящий отрезок, в котором находится корень. Воспользуйтесь неравенством из раздела 11.4 для определения наименьшего количества необходимых итераций. Выполните итерации алгоритма, как это было сделано для примера в тексте раздела.
6. Напишите уравнение прямой, проходящей через точки $(a_n, f(a_n))$ и $(b_n, f(b_n))$, и найдите точку ее пересечения с осью абсцисс.
7. Обратитесь к примеру в тексте раздела. В качестве критерия останова можно использовать длину отрезка (a_n, b_n) либо неравенство (11.11).
8. Напишите уравнение касательной к графику функции в точке $(x_n, f(x_n))$ и найдите точку ее пересечения с осью абсцисс.
9. Обратитесь к примеру в тексте раздела. Конечно, вы можете выбрать другое начальное значение x_0.
10. Рассмотрите, например, $f(x) = \sqrt[3]{x}$.

Предметный указатель

Научно-популярное издание

Ананий В. Левитин

Алгоритмы: введение в разработку и анализ

Литературный редактор *Ж.Е. Прусакова*
Верстка *А.Н. Полинчик*
Художественный редактор *В.Г. Павлютин*
Корректоры *Л.А. Гордиенко*, *Ж.Е. Прусакова*

Издательский дом "Вильямс".
127055, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 10.03.2006. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 46,4. Уч.-изд. л. 30,1.
Тираж 3000 экз. Заказ № 2966.

Отпечатано с диапозитивов
в ОАО "Печатный двор" им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.