

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.

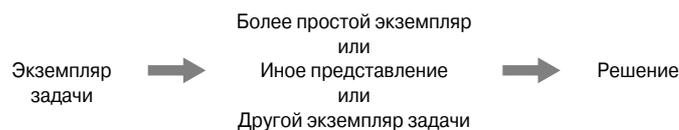


Рис. 6.1. Стратегия “преобразуй и властвуй”

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 практических 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 временная 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  $i \leftarrow 0$  to  $n - 2$  do
    if  $A[i] = A[i + 1]$ 
        return false
return true
```

¹Алгоритмы *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 $\Theta(n \log n)$:

$$T(n) = T_{\text{sort}}(n) + T_{\text{scan}}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n). \quad \blacksquare$$

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

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

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  $i \leq n - 1$  do
     $runlength \leftarrow 1$ ;  $runvalue \leftarrow A[i]$ 
    while  $i + runlength \leq n - 1$  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 $\lceil \log_2 n \rceil + 1$ сравнений в наихудшем случае. Даже при использовании максимально эффективной сортировки класса $n \log n$ общее время работы алгоритма в наихудшем случае составит

$$T(n) = T_{\text{sort}}(n) + T_{\text{search}}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

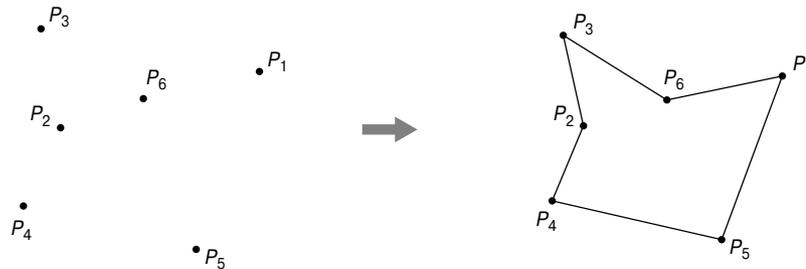
что хуже, чем в случае последовательного поиска. То же самое можно сказать и об эффективности в среднем случае. Конечно, если поиск требуется выполнять неоднократно, то затраты времени на сортировку могут оказаться оправданными (в упражнении 6.1.6 требуется оценить наименьшее количество поисков, при 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 $A = \{a_1, a_2, \dots, a_n\}$ и $B = \{b_1, b_2, \dots, b_m\}$ — 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 ($n \geq m$). 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 $n \geq 3$ 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:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

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:

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 & & a'_{11}x_1 + a'_{12}x_2 + \dots + a'_{1n}x_n = b'_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 & \Rightarrow & a'_{22}x_2 + \dots + a'_{2n}x_n = b'_2 \\ & & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n & & a'_{nn}x_n = b'_n \end{array}$$

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:

²Метод 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} , \dots , 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).

$$2x_1 - x_2 + x_3 = 1$$

$$4x_1 + x_2 - x_3 = 5$$

$$x_1 + x_2 + x_3 = 0$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad \begin{array}{l} \text{Строка 2} - \frac{4}{2} \cdot \text{Строка 1} \\ \text{Строка 3} - \frac{1}{2} \cdot \text{Строка 1} \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \quad \text{Строка 3} - \frac{1}{2} \cdot \text{Строка 2}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

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  $i \leftarrow 1$  to  $n$  do
   $A[i, n + 1] \leftarrow b[i]$  // Расширение матрицы
for  $i \leftarrow 1$  to  $n - 1$  do
  for  $j \leftarrow i + 1$  to  $n$  do
    for  $k \leftarrow i$  to  $n + 1$  do
       $A[j, k] \leftarrow A[j, k] - A[j, i] * A[i, k] / A[i, i]$ 
```

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 поменять 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

³Подробнее 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, называемая **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  $i \leftarrow 1$  to  $n$  do
   $A[i, n + 1] \leftarrow b[i]$  // Расширение матрицы
  for  $i \leftarrow 1$  to  $n - 1$  do
     $pivotrow \leftarrow i$ 
    for  $j \leftarrow i + 1$  to  $n$  do
      if  $|A[j, i]| > |A[pivotrow, i]|$ 
         $pivotrow \leftarrow j$ 
      for  $k \leftarrow i$  to  $n + 1$  do
         $swap(A[i, k], A[pivotrow, k])$ 
      for  $j \leftarrow i + 1$  to  $n$  do
         $temp \leftarrow A[j, i] / A[i, i]$ 
        for  $k \leftarrow i$  to  $n + 1$  do
           $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$ 
```

Давайте определим временную 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-

⁴Как 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:

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (n+2-i) = \\
 &= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i) = \\
 &= (n+1)(n-1) + n(n-2) + \dots + 3 \cdot 1 = \sum_{j=1}^{n-1} (j+2)j = \\
 &= \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2 \frac{(n-1)n}{2} = \\
 &= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3).
 \end{aligned}$$

Поскольку временная эффективность второй стадии (обратной подстановки) алгоритма исключения Гаусса равна $\Theta(n^2)$ (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 —

$$\begin{aligned} x_3 &= (-2)/2 = -1, \\ x_2 &= (3 - (-3)x_3)/3 = 0, \\ x_1 &= (1 - x_3 - (-1)x_2)/2 = 1. \end{aligned}$$

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} , $1 \leq i, j \leq n$ 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 ($1 \leq j \leq n$):

$$Ax^j = e^j.$$

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

$$\begin{aligned} \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} &= a_{11} \det \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12} \det \begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13} \det \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} = \\ &= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{31}a_{22}a_{13} - \\ &\quad - a_{21}a_{12}a_{33} - a_{32}a_{23}a_{11}. \end{aligned}$$

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, используемую 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):

$$x_1 = \frac{\det A_1}{\det A}, \dots, x_j = \frac{\det A_j}{\det A}, \dots, x_n = \frac{\det A_n}{\det A},$$

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:

$$x_1 + x_2 + x_3 = 2$$

$$2x_1 + x_2 + x_3 = 3$$

$$x_1 - x_2 + 3x_3 = 8$$

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?

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} (n+2-i)(n-i) = \\ &= \sum_{i=1}^{n-1} ((n+2)n - i(2n+2) + i^2) = \\ &= \sum_{i=1}^{n-1} (n+2)n - \sum_{i=1}^{n-1} (2n+2)i + \sum_{i=1}^{n-1} i^2. \end{aligned}$$

Поскольку $s_1(n) = \sum_{i=1}^{n-1} (n+2)n \in \Theta(n^3)$, $s_2(n) = \sum_{i=1}^{n-1} (2n+2)i \in \Theta(n^3)$ и $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta(n^3)$, то $s_1(n) - s_2(n) + s_3(n) \in \Theta(n^3)$.

5. Напишите псевдокод стадии обратной подстановки метода исключения Гаусса и покажите, что время его работы равно $\Theta(n^2)$.
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 поддер-

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 вырождаться 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.)
- Вторым подходом 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. 6.2а — AVL-дерево, в то время как бинарное дерево поиска на рис. 6.2б таковым не является.

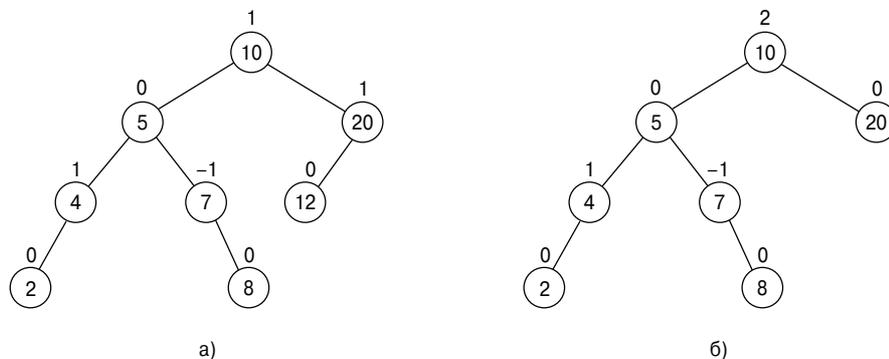


Рис. 6.2. а) AVL-дерево. б) Бинарное дерево поиска, не являющееся AVL-деревом. Показатели сбалансированности показаны над узлами деревьев

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. 6.3а, 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.

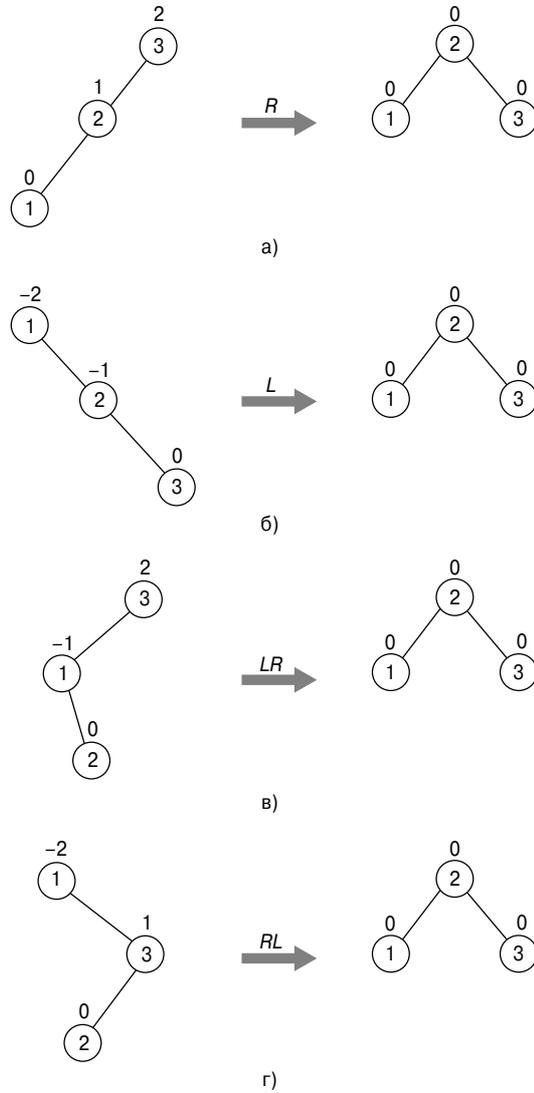


Рис. 6.3. Четыре типа поворотов AVL-деревьев с тремя узлами. а) Единичный R -поворот. б) Единичный L -поворот. в) Двойной LR -поворот. г) Двойной RL -поворот

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 сбалансир-

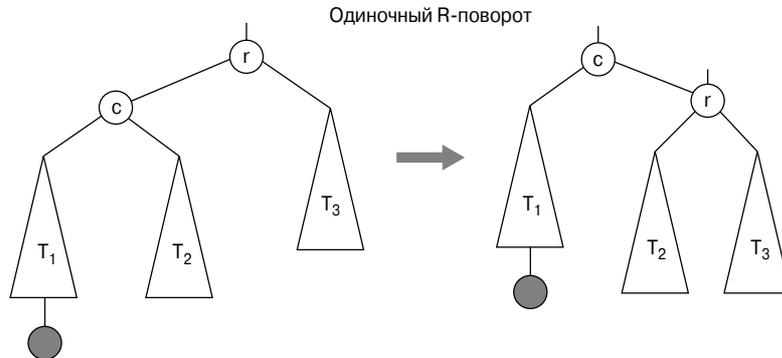


Рис. 6.4. Общий вид R -поворота в AVL-дереве. Последний вставленный узел выделен штриховкой

равности -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 $+1$.

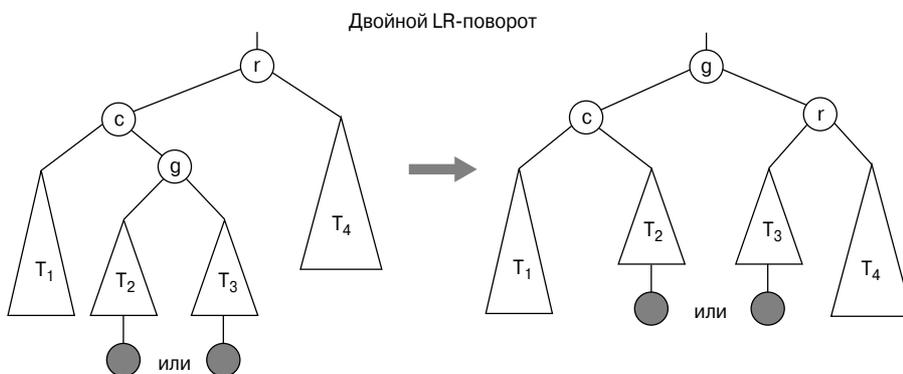


Рис. 6.5. Общий вид двойного LR -поворота в AVL-дереве. Последний вставленный узел выделен штриховкой. Он может быть либо в левом, либо в правом поддерево “внука” корня

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 поддеревы T_1 меньше c , который меньше всех ключей поддеревы 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

$$\lceil \log_2 n \rceil \leq h < 1.4405 \log_2 (n + 2) - 1.3277.$$

(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 даром. 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.

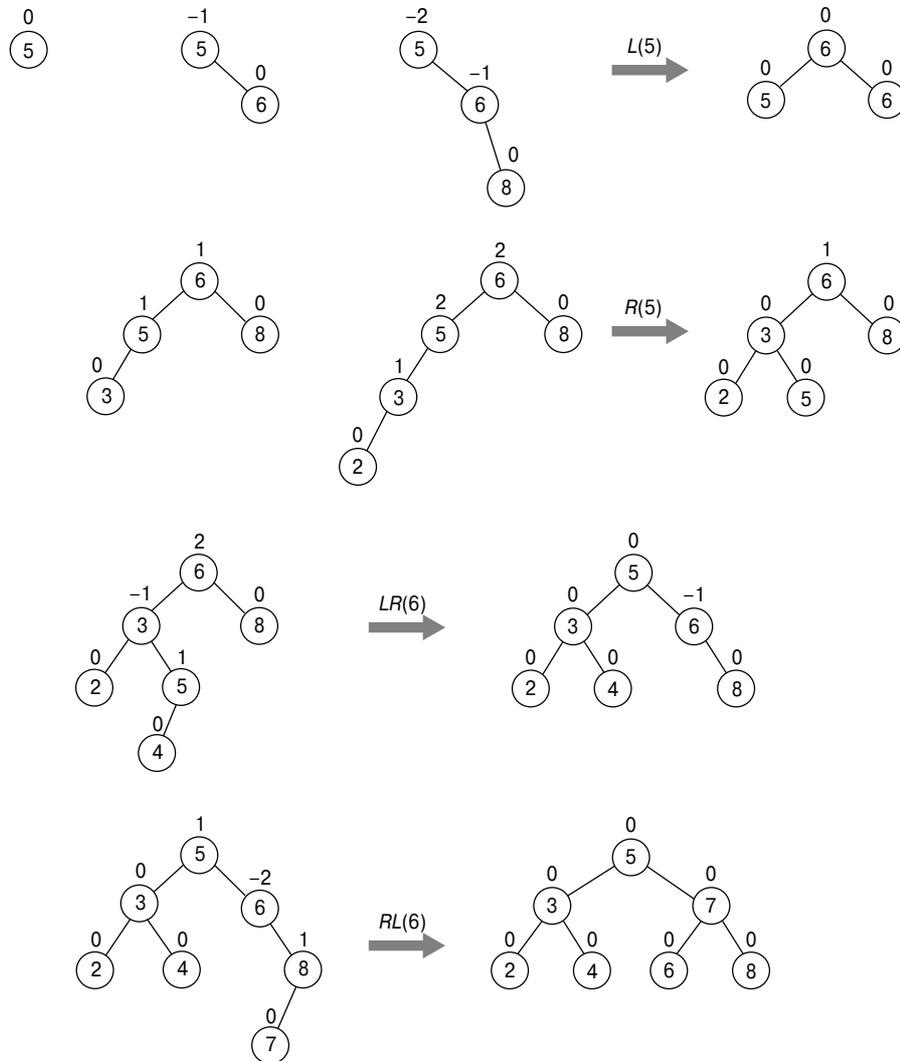


Рис. 6.6. Построение AVL-дерева путем последовательной вставки чисел из списка 5, 6, 8, 3, 2, 4, 7. Число в скобках после указания типа поворота — корень перестраиваемого дерева

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 Нор-

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).

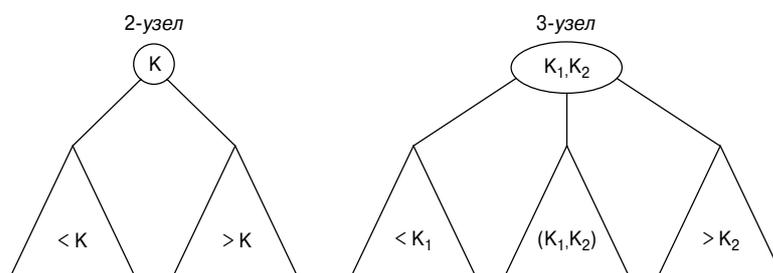


Рис. 6.7. Два вида узлов в 2-3-дереве

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.

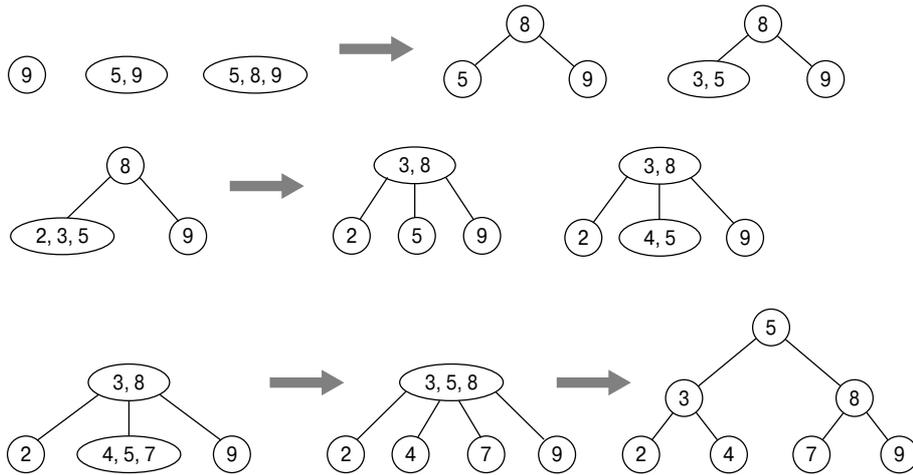


Рис. 6.8. Построение 2-3-дерева путем внесения ключей 9, 5, 8, 3, 2, 4, 7

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

$$n \geq 1 + 2 + \dots + 2^h = 2^{h+1} - 1,$$

следовательно,

$$h \leq \log_2(n + 1) - 1.$$

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

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \dots + 2 \cdot 3^h = 2(1 + 3 + \dots + 3^h) = 3^{h+1} - 1,$$

следовательно,

$$h \geq \log_3(n + 1) - 1.$$

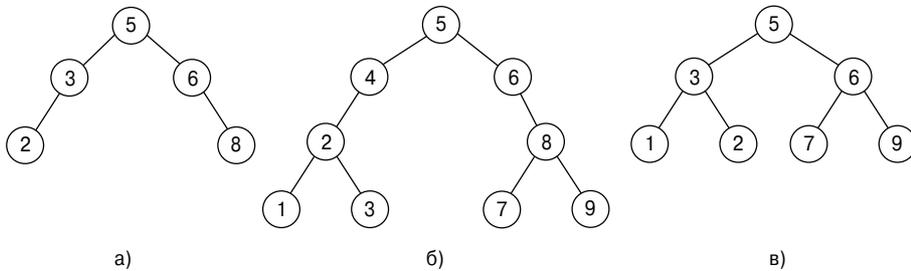
Из полученных таким образом верхней и нижней границ высоты h

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$$

вытекает, что временная 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?



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-деревя для списка из n различных целых чисел.
7. а) Постройте 2-3-дерево для следующего множества символов: С, О, М, Р, U, Т, 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 худшем 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 хитрую 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, называемой *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 алгоритма

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 ■

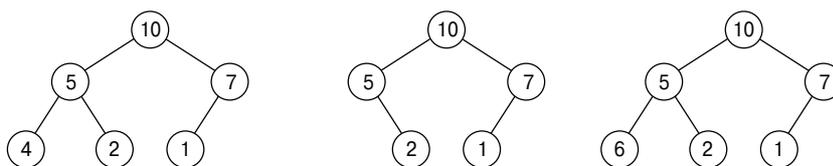


Рис. 6.9. Иллюстрация к определению пирамиды: таковой является только крайнее слева дерево

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.

⁵Некоторые авторы требуют, чтобы ключ в каждом узле *не превышал* ключи в дочерних узлах.

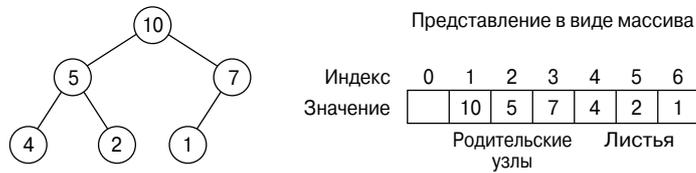


Рис. 6.10. Пирамида и ее представление в виде массива

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 ($1 \leq i \leq \lfloor n/2 \rfloor$) находятся в позициях $2i$ и $2i + 1$, соответственно; родительский ключ для ключа в позиции i ($2 \leq i \leq n$) 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.

$$H[i] \geq \max\{H[2i], H[2i + 1]\} \quad \text{для } i = 1, 2, \dots, \lfloor n/2 \rfloor.$$

(Конечно, если $2i + 1 > n$, то выполняться должно только неравенство $H[i] \geq H[2i]$.) 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

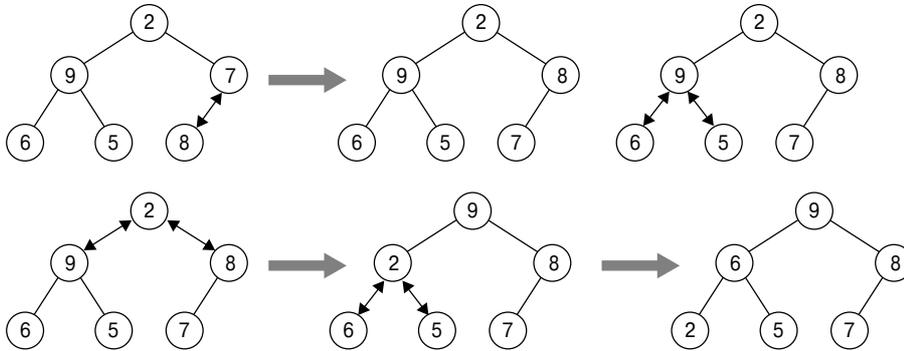


Рис. 6.11. Восходящее построение пирамиды для множества ключей 2, 9, 7, 6, 5, 8

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 пирамиды, 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 большими 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
   $k \leftarrow i; v \leftarrow H[k]$ 
   $heap = \text{false}$ 
  while not  $heap$  and  $2 * k \leq n$  do
     $j \leftarrow 2 * k$ 
    if  $j < n$  // Имеется два дочерних узла
      if  $H[j] < H[j + 1]$ 
```

```

     $j \leftarrow j + 1$ 
    if  $v \geq H[j]$ 
         $heap \leftarrow \mathbf{true}$ 
    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 $\sum_{i=1}^h i 2^i$ (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 .

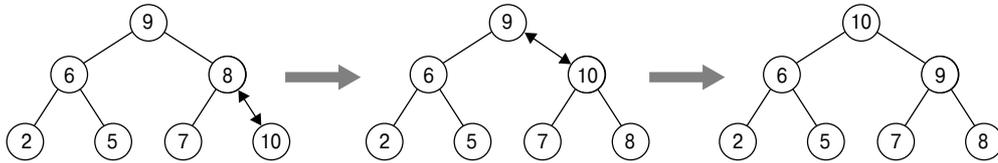


Рис. 6.12. Вставка ключа 10 в пирамиду, построенную на рис. 6.11

Очевидно, что такая вставка не может требовать большего количества сравнений ключей, чем высота пирамиды. Поскольку высота пирамиды с n узлами около $\log_2 n$, временная эффективность вставки составляет $O(\log n)$.

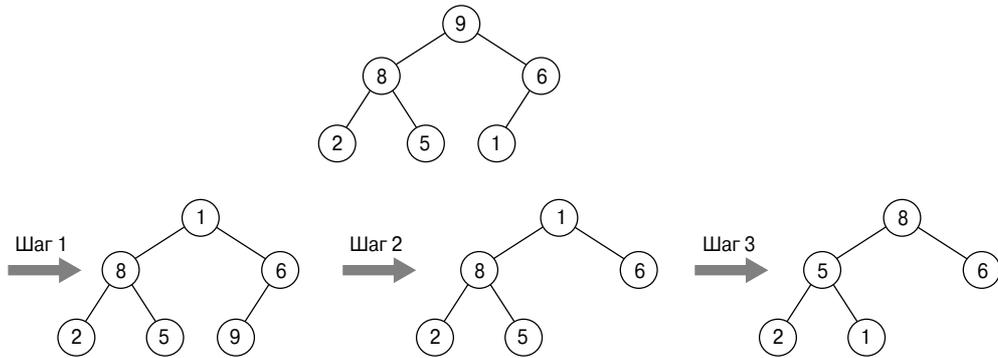


Рис. 6.13. Удаление корневого ключа из пирамиды

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, временная 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 временную 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:

$$\begin{aligned} C(n) &\leq 2 \lfloor \log_2(n-1) \rfloor + 2 \lfloor \log_2(n-2) \rfloor + \dots + 2 \lfloor \log_2 1 \rfloor \leq 2 \sum_{i=1}^{n-1} \log_2 i \leq \\ &\leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \leq 2n \log_2 n. \end{aligned}$$

Это означает, что для второго этапа пирамидальной сортировки $C(n) \in O(n \log n)$. Более подробный анализ показывает, что в действительности временная эффективность пирамидальной сортировки равна $\Theta(n \log n)$ как в среднем, так и в наихудшем случаях. Таким образом, временная эффективность пирамидальной сортировки попадает в тот же класс, что и сортировка слиянием, но,

Этап 1. Построение пирамиды	Этап 2. Удаление наибольших элементов
2 9 7 6 5 8	9 6 8 2 5 7
2 9 8 6 5 7	7 6 8 2 5 9
2 9 8 6 5 7	8 6 7 2 5
9 2 8 6 5 7	5 6 7 2 8
9 6 8 2 5 7	7 6 5 2
	2 6 5 7
	6 2 5
	5 2 6
	5 2
	2 5
	2

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

- Постройте пирамиду для чисел 1, 8, 6, 5, 3, 7, 4 при помощи восходящего алгоритма.
 - Постройте пирамиду для чисел 1, 8, 6, 5, 3, 7, 4 при помощи нисходящего алгоритма.
 - Всегда ли восходящий и нисходящий алгоритмы построения пирамиды дают одну и ту же пирамиду?
- Набросайте схему алгоритма для проверки того, является ли пирамидой массив $H[1..n]$, и определите его временную эффективность.
- Найдите наименьшее и наибольшее количество ключей, которые может содержать пирамида высотой h .
 - 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 временную эффективность.
 б) Разработайте эффективный алгоритм для поиска и удаления из пирамиды элемента с заданным значением v и определите его временную 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 $1, 2, \dots, n$;
 в) последовательность чисел $n, n-1, \dots, 1$.
-  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

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (6.1)$$

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:

$$p(x) = (\dots (a_n x + a_{n-1}) x + \dots) x + a_0. \quad (6.2)$$

Например, для полинома $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ получим:

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 = \\ &= x(2x^3 - x^2 + 3x + 1) - 5 = \\ &= x(x(2x^2 - x + 3) + 1) - 5 = \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned} \quad (6.3)$$

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  $x$ 
 $p \leftarrow P[n]$ 
for  $i \leftarrow n - 1$  downto 0 do
     $p \leftarrow x * p + P[i]$ 
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 $n = b_l \dots b_i \dots b_0$ — битовая строка, представляющая положительное целое n в двоичной системе счисления. Это означает, что значение n может быть вычислено как значение полинома

$$p(x) = b_l x^l + \dots + b_i x^i + \dots + b_0 \quad (6.4)$$

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

$$a^n = a^{p(2)} = a^{b_I x^I + \dots + b_i x^i + \dots + b_0}.$$

Схема Горнера для вычисления бинарного полинома $p(2)$	Следствие для вычисления $a^n = a^{p(2)}$
$p \leftarrow 1$ // Старший разряд всегда 1 при $n \geq 1$	$a^p \leftarrow a^1$
for $i \leftarrow I - 1$ downto 0	for $i \leftarrow I - 1$ downto 0
$p \leftarrow 2p + b_i$	$a^p \leftarrow a^p \leftarrow a^{2p+b_i}$

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  $b_I, \dots, b_0$ 
// в бинарном представлении натурального  $n$ 
// Выходные данные: Значение  $a^n$ 
product  $\leftarrow a$ 
for  $i \leftarrow I - 1$  downto 0 do
  product  $\leftarrow$  product * product
  if  $b_i = 1$ 
    product  $\leftarrow$  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

$$(b - 1) \leq M(n) \leq 2(b - 1),$$

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:

$$a^n = a^{b_I 2^I + \dots + b_i 2^i + \dots + b_0} = a^{b_I 2^I} \cdot \dots \cdot a^{b_i 2^i} \cdot \dots \cdot a^{b_0}.$$

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 $a^{2^i} = (a^{2^{i-1}})^2$. 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  $b_I, \dots, b_0$ 
// в бинарном представлении натурального  $n$ 
// Выходные данные: Значение  $a^n$ 
term  $\leftarrow a$  // Инициализация  $a^{2^i}$ 
if  $b_0 = 1$ 
    product  $\leftarrow a$ 
else
    product  $\leftarrow 1$ 
for  $i \leftarrow 1$  to  $I$  do
    term  $\leftarrow term * term$ 
    if  $b_i = 1$ 
```

```

    product ← 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 $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ 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 $p(x) = x^n + x^{n-1} + \dots + x + 1$?
10. В соответствии со следствием из Основной теоремы алгебры, каждый полином

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

может быть представлен в виде

$$p(x) = a_n (x - x_1)(x - x_2) \dots (x - x_n),$$

где x_1, x_2, \dots, 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).

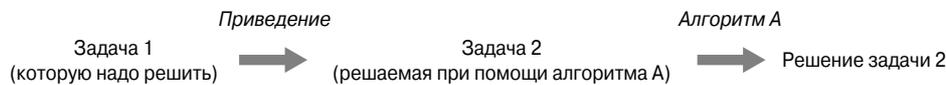


Рис. 6.15. Стратегия приведения задачи

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 $\text{lcm}(m, n)$) определяется как наименьшее натуральное число, делящееся и на m , и на n . Например, $\text{lcm}(24, 60) = 120$ и $\text{lcm}(11, 5) = 55$. 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 $\text{lcm}(m, n)$ 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,

$$\begin{aligned} 24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\ 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ \text{lcm}(24, 60) &= (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120 \end{aligned}$$

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 $\text{lcm}(m, n)$ и $\text{gcd}(m, n)$? Нетрудно увидеть, что произведение $\text{lcm}(m, n)$ и $\text{gcd}(m, n)$ 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

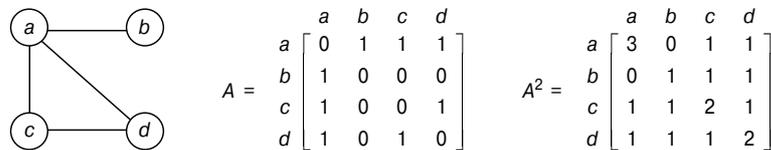
$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)},$$

где $\text{gcd}(m, n)$ 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 : $a - d - c$.



$$A = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix} \quad A^2 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{matrix}$$

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:

$$\min f(x) = -\max[-f(x)].$$

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.

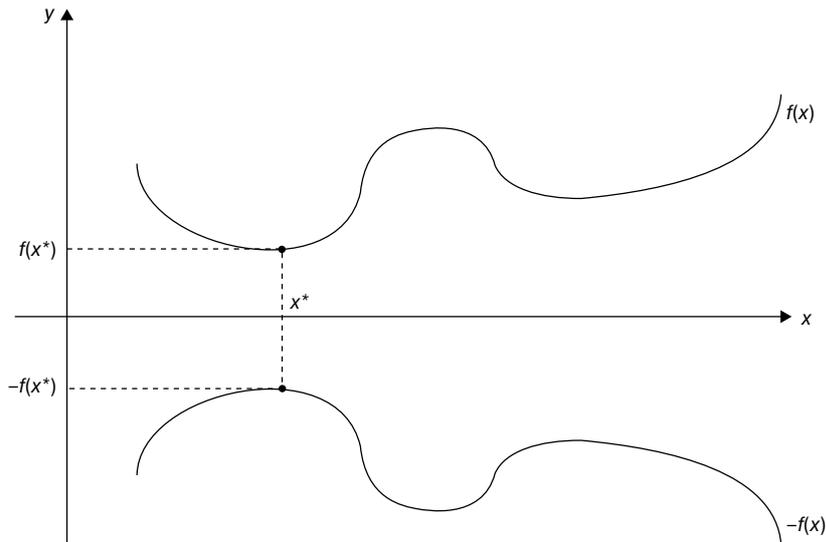


Рис. 6.17. Связь между задачами минимизации и максимизации:
 $\min f(x) = -\max[-f(x)]$

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:

$$\begin{aligned} &\text{Максимизировать} && 0.10x + 0.07y + 0.03z \\ &\text{при условиях} && x + y + z = 100 \\ &&& x \leq \frac{1}{3}y \\ &&& z \geq 0.25(x + y) \\ &&& x \geq 0, y \geq 0, z \geq 0. \end{aligned}$$

■

Хотя эта конкретная задача невелика и проста, она показывает, как задача принятия оптимального решения может быть приведена к экземпляру задачи линейного программирования

$$\begin{aligned} &\text{Максимизировать} && c_1x_1 + \cdots + c_nx_n \\ &(\text{или минимизировать}) && \\ &\text{при условиях} && a_{i1}x_1 + \cdots + a_{in}x_n \leq (\text{или } \geq, \text{ или } =) b_i \\ &&& \text{при } i = 1, \dots, m \\ &&& x_1 \geq 0, \dots, x_n \geq 0. \end{aligned}$$

(Последняя группа ограничений — так называемые ограничения неотрицательности — являются, строго говоря, необязательными, поскольку они представляют

собой частные случаи более общих ограничений $a_{i1}x_1 + \dots + a_{in}x_n \geq b_i$; 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 (Narendra 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, \dots, w_n и ценой v_1, \dots, 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 $x_j, j = 1, \dots, n$ — переменная, представляющая часть предмета j , помещаемую в рюкзак. Очевидно, что x_j должно удовлетворять неравенству $0 \leq x_j \leq 1$. 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:

$$\begin{aligned} & \text{Максимизировать} && \sum_{j=1}^n v_j x_j \\ & \text{при условиях} && \sum_{j=1}^n w_j x_j \leq W \\ & && 0 \leq x_j \leq 1, \quad j = 1, \dots, n. \end{aligned}$$

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:

$$\begin{aligned} & \text{Максимизировать} && \sum_{j=1}^n v_j x_j \\ & \text{при условиях} && \sum_{j=1}^n w_j x_j \leq W \\ & && x_j \in \{0, 1\}, \quad j = 1, \dots, n. \end{aligned}$$

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.

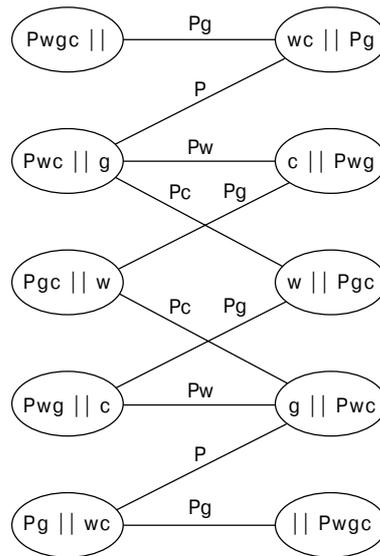


Рис. 6.18. Граф пространства состояний для головоломки о переправе через реку

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 (\parallel) символизируют реку. Для удобства мы также поместили ребра, указывая, кто именно находится в лодке при пересечении реки. В терминах данного графа нас интересует поиск пути из начальной вершины, помеченной P_{wgc} , в конечную — P_{wgc} .

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 $\text{lcm}(m, n)$:

$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)}.$$

- б) Как известно, алгоритм Евклида имеет время работы $O(\log n)$. Если этот алгоритм используется для поиска $\text{gcd}(m, n)$, то какова эффективность алгоритма поиска $\text{lcm}(m, n)$?
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 временной 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 $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$. 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 $i, j = 1, \dots, n$. 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:

$$\begin{array}{ll} \text{Максимизировать} & 0.10x + 0.07y + 0.03z \\ \text{при условиях} & x + y + z = 100 \\ & x \leq \frac{1}{3}y \\ & z \geq 0.25(x + y) \\ & x \geq 0, y \geq 0, z \geq 0. \end{array}$$

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 $n \geq 2$ 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 $n \geq 4$? 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, называемых *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.