ЛЕКЦИИ ШКОЛЫ
АНАЛИЗА ДАННЫХ ЯНДЕКСА

М. А. Бабенко, М. В. Левин

Введение в теорию алгоритмов и структур данных

Яндекс

ИЗДАНИЕ ОСУЩЕСТВЛЯЕТСЯ
ПРИ ПОДДЕРЖКЕ КОМПАНИИ «ЯНДЕКС»

М. А. Бабенко, М. В. Левин

Введение в теорию алгоритмов и структур данных

Электронное издание

Москва
МЦНМО
2016

УДК 519.72
ББК 32.81
Б12

Бабенко М. А., Левин М. В.
Введение в теорию алгоритмов и структур данных
Электронное издание
М.: МЦНМО, 2016
144 с.
ISBN 978-5-4439-2396-3

В курсе дается краткое изложение классических способов построения и анализа алгоритмов. Первая часть курса, представленная в данном пособии, в большей степени сконцентрирована на базовых структурах данных, а также задачах сортировки и поиска. Теоретический материал дополняется рядом задач.

Несмотря на «олимпиадный» вид, многие из них имеют под собой вполне практическую основу и представляют собой модельные варианты тех проблем, с которыми приходится сталкиваться на практике.

Знания, которые даются в этой книге, представляют собой необходимую (хотя и недостаточную) базу для работы с произвольными данными большого объема, дают понимание о возможности или невозможности точного решения конкретных задач за приемлемое на практике время.

Подготовлено на основе книги:
Бабенко М. А., Левин М. В. Введение в теорию алгоритмов и структур данных. — М.: ФМОП, МЦНМО, 2012. — 144 с. — ISBN 978-5-94057-957-1

Издательство Московского центра
непрерывного математического образования
119002, Москва, Большой Власьевский пер., 11,
тел. (499)241–08–04.
`http://www.mccme.ru`

ISBN 978-5-4439-2396-3

Оглавление

Предисловие

Книга М. А. Бабенко и М. В. Левина является учебным пособием к курсу «Алгоритмы и структуры данных поиска», который авторы читают в течение последних 5 лет в Школе анализа данных Яндекса.

Это пособие примечательно тем, что помимо теории, содержащейся обычно в книгах и заметках лекций по Computer Science, к каждой главе приведено несколько примеров практических задач с решениями и техническими комментариями по реализации.

Пособием можно пользоваться и как самостоятельным введением в теорию алгоритмов, однако авторы убеждены, что решение практических задач, реализация их на каком-нибудь языке программирования и прохождение Code Review являются важнейшей и неотъемлемой частью курса, которую, к сожалению, невозможно включить в книгу.

Данная книга покрывает только первую часть курса алгоритмов, вторая часть находится в разработке.

Елена Бунина,
Директор отделения Computer Science,
профессор кафедры высшей алгебры
механико-математического факультета
МГУ им. М. В. Ломоносова,
доктор физико-математических наук.

Глава 1. Введение

Представленная вашему вниманию работа составлена по мотивам лекций и семинаров курса «Алгоритмы и структуры данных для поиска», читаемого в Школе анализа данных Яндекса. Полный курс состоит из двух семестров, мы же пока публикуем материалы первого из них с надеждой, что в скором времени мы найдем в себе силы подготовить также и вторую часть.

Как уже было отмечено выше, занятия состоят из лекций и семинаров. На лекциях обсуждаются разнообразные теоретические вопросы, а также изучаются приемы реализации стандартных алгоритмов. Семинары в большей степени ориентированы на решение задач.

Новые комплекты практических заданий периодически выдаются на семинарах, и слушатели должны в течение установленного срока прислать свое решение. Решение состоит из двух частей: теоретическое обоснование выбранного метода (включая оценки сложности) и практическая реализация. В качестве последней принимается код на языке C++.

Реализации, присылаемые студентами Школы, проверяются в автоматическом режиме на наборе заранее подготовленных тестов. Для прохождения теста необходимо выдать правильный ответ, не превышая при этом установленного предела по ресурсам (времени и памяти). Решение засчитывается в случае, если оно успешно проходит все тесты.

С учетом этого процесса и был написан данный конспект. Структурно он состоит из набора *глав*, каждая из которых фокусируется на определенном разделе теории. В конце каждой главы вы найдете примеры *задач*. Для многих задач мы приводим формат ввода-вывода, который ожидается от программ, а также несколько примеров входных и выходных данных. Часть из этих задач, впрочем, более теоретическая, поэтому для них подобных форматов не приводится.

Для всех задач вы найдете описание теоретической части решения в форме, близкой к той, что ожидается от слушателей. Некоторые из

этих описаний не вполне формальны, но надеемся, что придирчивый читатель простит нам подобные вольности.

Несмотря на то что данный курс сам по себе является вводным, мы предполагаем, что читатель имеет хотя бы минимальное представление о том, что из себя представляет программирование. В частности, мы предполагаем известными основные понятия какого-либо типичного императивного языка программирования.

Большинство примеров в данной книге будут проиллюстрированы на языке C++. Причиной этого является его широкая распространенность в качестве практического средства, позволяющего писать эффективный и переносимый код.

Помимо знания C++ как такового, мы считаем, что читатель знаком со стандартной библиотекой языка. Это, конечно, не предполагает автоматического понимания устройства структур данных и алгоритмов, присутствующих там. Мы считаем известным разделение на *интерфейс* и *реализацию*. В случае со стандартной библиотекой речь идет о понимании ее интерфейсов, а также их неотъемлемой части — гарантии *сложности* операций. Такие гарантии определяют поведение библиотечного кода в худшем случае.

И все же наша цель — изучение базовых эффективных алгоритмов. Такая цель не предполагает привязку к какой-либо платформе и библиотеке. В частности, желательно разобраться с тем, как именно реализованы библиотечные средства.

1.1. Массивы переменного размера

Читатель, без сомнения, знаком с понятием *массива*. Массив — это средство языка C++, а не библиотеки. Библиотека же предлагает более удобный аналог массива, а именно *вектор* (класс `std::vector`). С точки зрения интерфейса ключевое различие между вектором и массивом состоит в том, что массив имеет фиксированный, выбираемый в момент его создания размер, а вектор — напротив, может изменять количество лежащих в нем элементов динамически во время работы. Как же реализуется такая возможность на практике? Вопрос этот не вполне тривиальный, но достаточно простой, чтобы начать изложение материалов курса с него.

Будем рассматривать простейший случай: нам необходимо предложить структуру данных, хранящую последовательность элементов переменной длины и поддерживающую операцию INSERT добавления нового элемента в конец последовательности. Конечно, операция INSERT должна быть по возможности быстрой. Кроме того, должна быть возможность обратиться к любому элементу по его индексу за время $O(1)$.

Последнее требование означает, что наиболее близкая к требуемой структура — это обычный массив фиксированного размера, не меньшего текущей длины последовательности. Такое решение также упрощает (по крайней мере в языке C++) взаимодействие с кодом, который ожидает получить на вход обычный массив. Этот массив обычно задается указателем на свой начальный (нулевой) элемент, и в случае вектора такой указатель всегда легко получить. Иными словами, проектируемый нами вектор гарантирует непрерывное расположение своих элементов в памяти.

Поскольку длина массива в общем случае больше текущей длины последовательности, последнюю нужно явно хранить. Возникает естественный вопрос: что же делать в случае, когда происходит вызов INSERT, а длина последовательности уже совпадает с длиной массива, так что свободных элементов в конце его нет?

В этом случае придется выполнить операцию *перевыделения* (*reallocation*): создать новый массив большего размера и скопировать в него информацию из старого массива. Эта операция, конечно, не бесплатная: ее временна́я сложность пропорциональна количеству элементов, подлежащих копированию.

Но как выбрать этот новый размер массива? К примеру, можно было бы каждый раз увеличивать размер массива при перевыделении на какое-либо постоянное количество элементов d. Такой метод называется *аддитивным*. Несложно убедиться, что общее время, которое будет затрачено на выполнение последовательности из n операций INSERT, составляет[1] $\Omega(n^2)$.

[1] Мы используем ряд стандартных обозначений асимптотического анализа. Для положительных функций f и g записи $f = O(g)$, $f = \Omega(g)$ обозначают соответственно, что найдется такая константа $C > 0$, что неравенства $f \leqslant C \cdot g$ и $f \geqslant C \cdot g$ выполнены для всех достаточно больших значений аргументов. Если $f = O(g)$ и $g = O(f)$, то пишем $f = \Theta(g)$.

Упражнение 1.1.1. Скрытая константа в этой оценке зависит от d. Как именно?

Квадратичную общую сложность последовательности из n вставок, оказывается, возможно уменьшить до линейной. Для этого нужно использовать *мультипликативный* метод: получать новую длину умножением старой на некоторую константу $\alpha > 1$. Для простоты положим $\alpha = 2$, так что размер массива каждый раз при необходимости будет удваиваться (при этом, конечно, начинать нужно с длины 1, а не 0).

Доказать, что последовательность из n операций INSERT будет выполняться за время $O(n)$, несложно и непосредственно. Сейчас, однако, будет описан некоторый общий метод получения подобных оценок, который впоследствии нам не раз пригодится.

1.2. Анализ учетных стоимостей

Будем изучать общую ситуацию, при которой имеется некоторая структура данных S. Предположим, что над данной структурой данных выполняется последовательность операций $a_1, \ldots, a_n$. Каждая операция a_i требует определенных временны́х затрат, которые мы обозначим через $c(a_i)$. Числа $c(a_i)$ будем называть *фактическими* стоимостями операций.

Обозначим через s_i состояние структуры S после выполнения i операций $(0 \leqslant i \leqslant n)$. В частности, s_0 — это начальное состояние до выполнения операций, а s_n — заключительное состояние, в которое структура переходит после выполнения всех операций из списка.

Наша задача состоит в том, чтобы оценить сумму времен, которые затрачиваются на выполнение всех n операций, т. е.

$$\sum_{i=1}^{n} c(a_i). \tag{1.1}$$

Трудность получения оценки данной суммы связана с тем, что фактические стоимости зачастую слишком сильно различаются. Поэтому оценка вида Cn, где C означает максимально возможную фактическую стоимость, может оказаться слишком грубой.

Предположим, что с каждым из состояний s_i связано некоторое вещественное значение φ_i $(0 \leqslant i \leqslant n)$, называемое *потенциалом*. Тогда

можно определить значения

$$c'(a_i) := c(a_i) + \varphi_i - \varphi_{i-1} \quad \text{для всех } i = 1, \ldots, n,$$

которые мы будем называть *приведенными* (или *учетными*) стоимостями. Идея состоит в том, что подходящим выбором потенциалов иногда удается добиться того, что максимальное возможное значение приведенной стоимости операции (обозначим его C') оказывается намного меньше максимально возможного фактического значения. Это позволяет оценить сумму (1.1) так:

$$\sum_{i=1}^{n} c(a_i) = \sum_{i=1}^{n} c'(a_i) + \varphi_0 - \varphi_n \leqslant C'n + (\varphi_0 - \varphi_n). \tag{1.2}$$

Если дополнительно известно, что $\varphi_n \geqslant \varphi_0$, то оценка упрощается до $C'n$.

Применим введенную технику для анализа суммарной сложности последовательности из n вставок в вектор. Фактическая стоимость одной операции INSERT зависит от того, происходит ли на данном шаге перевыделение. Если нет, то время работы можно принять за одну условную единицу. Если же перевыделение необходимо, то время работы составляет $m+1$ условных единиц, где m — текущий размер массива (m единиц уходят на копирование существующих значений, а еще одна уходит на сохранение нового значения в перевыделенном участке памяти).

Как видно, фактические времена выполнения операций сильно разнятся. Чтобы выровнять их, введем потенциал следующим образом. Заметим, что если размер массива на каждом перевыделении удваивается, то в любой момент времени, кроме начального (когда вектор пуст), количество занятых элементов s в массиве не меньше половины его длины l. Потенциал φ состояния структуры данных примем равным $2s - l$.

В случае добавления без перевыделения потенциал меняется на $O(1)$, значит, учетная стоимость операции также составляет $O(1)$.

Более интересен случай добавления с перевыделением. Тогда $l = s$, и операция INSERT стоит $s + 1$ условных единиц. После перевыделения потенциал равен 2, а значит, увеличение потенциала равно

$2 - (2s - s) = 2 - s$. Учетная стоимость операции INSERT в итоге составляет $(s+1) + (2-s) = 3$ единицы.

В обоих случаях получаем оценку $O(1)$ на учетную стоимость операции INSERT. Кроме того, конечный потенциал не меньше начального. Следовательно, по формуле (1.2) общая сложность последовательности операций составляет $O(n)$.

Упражнение 1.2.1. Покажите, что оценка $O(n)$ на общее время выполнения n операций INSERT остается справедливой и для любого другого коэффициента перевыделения $\alpha > 1$. Как зависит скрытая в оценке $O(n)$ константа от α?

1.3. Задачи

Задача 1. *Правильной скобочной последовательностью* называется строка, состоящая только из скобок, в которой все скобки можно разбить на пары таким образом, что

- в каждой паре есть левая и правая скобка, причем левая скобка расположена левее правой;
- для любых двух пар скобок либо одна из них полностью находится внутри другой пары, либо промежутки между скобками в парах не пересекаются;
- в паре с круглой скобкой может быть только круглая скобка, с квадратной — квадратная, с фигурной — фигурная.

Примеры:

- если разрешены только круглые скобки:
  - правильные последовательности: `()`, `(())`, `()()`, `()()()`, `(())()`, `((()))`;
  - неправильные последовательности: `)(`, `))`, `((`, `())()(`, `()),` `))((`;
- если разрешены круглые и квадратные скобки:
  - правильные последовательности: `[]`,`()`, `[()]`, `[[([])]()]`;
  - неправильные последовательности: `[)`, `([)]`, `(())()[]][`;
- если разрешены еще и фигурные скобки:
  - правильные последовательности: `[{(())}({})]`, `[]{}()`, `{}`, `()`, `[]`;
  - неправильные последовательности: `[{()}]`, `[()())]{}`.

Во входном файле задана строка α, состоящая только из скобок (круглых, квадратных и фигурных). Требуется определить, является ли она правильной скобочной последовательностью. Если да, выведите в выходной файл слово `CORRECT`. Если нет, выведите длину максимального префикса α, который либо сам является правильной скобочной последовательностью, либо может быть продолжен до таковой.

Например, для строки `(()))` ответом будет 4, так как строка `(())` является правильной скобочной последовательностью, а строку `(()))` уже нельзя никаким образом продолжить вправо, чтобы получить правильную скобочную последовательность. Для строки `])()(` ответ 0, поскольку строку `]` нельзя продолжить вправо, чтобы получить правильную скобочную последовательность. Для строки `[(()){()()[[]]}]` ответ `CORRECT`.

Указание. Понадобится стек. У класса `std::vector` есть методы `push_back` и `pop_back`, с помощью которых его можно легко имитировать.

Пример входа	Пример выхода
`(())`	`CORRECT`
`([)]`	`2`
`(([{`	`4`

Решение. Рассмотрим первую правую скобку, встречающуюся в строке. Если она находится на первой же позиции, то строка уже некорректна и нельзя ничего приписать справа, чтобы она стала корректной (не найдется пары к самой первой правой скобке).

Если же данная скобка встречается на позиции $r > 0$, то скобка на позиции $r - 1$ является левой, и она является парной к данной правой скобке. Действительно, если бы какая-то левая скобка на позиции $l < r - 1$ была парой к правой скобке на позиции r, то пара к левой скобке на позиции $r - 1$ находилась бы правее r, и тогда эти две пары пересекались бы, что противоречит определению. С другой стороны, если самая первая правая скобка стоит на позиции $r > 0$, то строку еще можно продолжить так, чтобы она стала корректной: нужно дописать столько правых скобок, чтобы закрыть все левые скобки, и остановиться. Поэтому самая первая правая скобка и левая скобка сразу перед ней соответствуют друг другу. Теперь, если выкинуть эту пару

скобок из строки, корректность оставшейся строки определяется тем же самым способом, так как пара скобок, стоящая непосредственно рядом, без промежуточных скобок, не мешает выполнению ни одного из наложенных условий. Из этих соображений вытекает следующий алгоритм решения задачи.

Из определения корректной скобочной последовательности следует, что при чтении такой последовательности слева направо всегда есть набор открывающих скобок, ожидающих себе парных закрывающих скобок (по первому свойству). При этом если встречается очередная закрывающая скобка, то она должна быть парной именно к последней открывающей скобке, иначе нарушилось бы второе свойство. В тот момент, когда для последней открывающей скобки в наборе находится парная закрывающая, эта открывающая скобка из набора удаляется, ожидающих открывающих скобок становится на одну меньше, и последней становится та, что была предпоследней. В конце, когда мы прочитали все символы строки, стек должен оказаться пустым: к каждой открывающей скобке нашлась парная. Кроме того, в процессе обработки никогда не должно получиться, что мы читаем закрывающую скобку, а стек либо пуст, либо на его вершине находится неправильная открывающая скобка. В этом случае последовательность сразу же получается некорректной, и к ней уже справа ничего нельзя дописать, чтобы последовательность стала корректной.

Такое поведение ожидающих открывающих скобок позволяет хранить их в стеке, используя следующий алгоритм. Изначально стек пустой. Идем по строке слева направо, считывая по одному символу. В зависимости от очередного символа, выполняем действия.

- Если очередной символ — открывающая скобка, то кладем его на стеки, двигаемся дальше.
- Если очередной символ — закрывающая скобка, то посмотрим на состояние стека. Если на стеке нет ни одного элемента или открывающая скобка на вершине стека не соответствует текущей закрывающей скобке, то последовательность неправильная. При этом она могла быть дополнена до правильной последовательности до текущего момента (обоснуйте), а после того, как найдена неправильная закрывающая скобка, она уже не может быть дополнена до правильной. Соответственно, нужно вывести дли-

ну прочитанного префикса, не включая последнюю прочитанную скобку, и выйти из программы. Если стек не пуст и в вершине стека лежит соответствующая открывающая скобка, удаляем из стека эту скобку и двигаемся дальше.

В конце, если пройдена вся строка до конца, нужно еще не забыть определить, является ли прочитанная строка правильной скобочной последовательностью или же только началом правильной скобочной последовательности. Если стек в конце просмотра строки пуст, то прочитанная строка является правильной скобочной последовательностью и нужно выдать `CORRECT`. Если же стек не пуст, то строку можно дополнить до правильной скобочной последовательности, но она еще такой не является, поэтому нужно выдать длину всей строки.

Для анализа каждого очередного символа строки требуется константное время, поэтому общая сложность алгоритма составляет $O(n)$. Памяти требуется $O(n)$ на хранение самой строки и $O(n)$ на стек.

Задача 2. Дана последовательность $a_1, a_2, \ldots, a_n$ из n различных целых чисел. Для каждого числа a_i найдите наименьшее такое j, что $j > i$ и $a_j > a_i$.

В первой строке входного файла записано число n. Во второй строке — числа $a_1, a_2, \ldots, a_n$; $1 \leqslant n \leqslant 10\,000\,000$, $-2^{31} \leqslant a_i \leqslant 2^{31} - 1$.

В выходной файл для каждого i, $1 \leqslant i \leqslant n$, выведите соответствующее j. Если правее a_i нет чисел, больших его, в качестве j выведите -1.

Пример входа	Пример выхода
5 1 3 2 4 5	2 4 4 5 -1
1 100	-1

Решение. Будем просматривать последовательность слева направо. Заведем стек, в котором будем хранить наибольшее из уже просмотренных чисел, затем следующее по величине из чисел, стоящих правее него, затем следующее по величине из чисел, стоящих правее, и т. д. Пусть просмотрены все числа $a_1, a_2, \ldots, a_i$. Тогда в вершине стека находится число $a_{k_1} = \max\limits_{1 \leqslant j \leqslant i} a_j$, следующее число в стеке

$a_{k_2} = \max\limits_{k_1<j\leqslant i} a_j$ либо следующего числа нет, если $k_1 = i$. Третье число в стеке, если есть второе и $k_2 < i$, равно $a_{k_3} = \max\limits_{k_2<j\leqslant i} a_j$, а если $k_2 = i$, то в стеке ровно два элемента, и т. д. Число в вершине стека будет минимальным, и далее числа возрастают от головы к хвосту.

Изначально стек пуст. Когда мы рассматриваем очередной элемент последовательности, то сравниваем его с вершиной стека (если стек непуст). До тех пор пока стек непуст и значение в вершине стека меньше очередного элемента, удаляем вершину из стека, при этом удаляя элемент a_i из стека. Если текущий рассматриваемый элемент a_j, то ответ для i будет равен j — записываем этот результат. Действительно, до тех пор пока мы не рассмотрели j-й элемент последовательности, из определения множества элементов, хранимых в стеке, следует, что для каждого из них еще не нашлось ни одного элемента последовательности правее его и больше. Соответственно, для всех элементов стека, которые меньше очередного, ответ определяется в этот момент. Когда из стека удалены все элементы, меньшие нового, добавляем новый элемент в стек, так как он является максимальным среди всех элементов правее вершины стека. Таким образом, в стеке всегда будет правильное множество элементов, кроме того, в нем всегда будет содержаться самый правый рассмотренный элемент. Алгоритм даст нам правильный ответ, а для чисел, которые останутся в стеке к моменту, когда уже просмотрена вся последовательность, ответ равен -1.

Сложность каждого отдельного шага не оценивается как $O(1)$, так как в течение одного шага может быть удалено вплоть до $n-1$ элементов из стека, если последовательность была $n-1, n-2, n-3, \ldots, 1, n$ и рассматривается n-й элемент, т. е. в худшем случае сложность каждого шага $O(n)$. Однако если оценить сложность всего алгоритма с помощью амортизационного анализа, то получается, что сложность линейна по n. Действительно, каждый элемент добавляем в стек и удаляем из стека не более одного раза. Значит, в сумме сделаем $O(n)$ операций со стеком, а каждая операция со стеком работает за время $O(1)$. Памяти необходимо $O(n)$.

Упражнение 1.3.1. Каково формальное определение потенциала, использованного в вышеуказанном анализе?

Глава 2. Сортировка

2.1. Введение

Одна из простейших, но одновременно фундаментальных алгоритмических задач — это *сортировка* (или *упорядочение*) набора данных. В общей форме ее можно описать следующим образом: на вход алгоритма поступает последовательность *ключей* $A = (k_1, \ldots, k_n)$. Ключи представляют собой объекты, выбираемые из некоторого линейно упорядоченного универсума.

Напомним, что множество E называется *линейно упорядоченным* относительно бинарного отношения $<$ на множестве E, если выполнены следующие аксиомы:

1) если $x < y$ и $y < z$, то $x < z$;
2) если $x \neq y$, то либо $x < y$, либо $y < x$;
3) неверно, что $x < x$.

Порядок, установленный на множестве ключей, сообщается алгоритму с помощью внешней процедуры, или, как говорят, *оракула сравнения* (*comparision oracle*). Данная процедура позволяет по переданной ей паре элементов (x, y) выяснить, верно ли, что $x < y$. Как обычно, мы пишем $x \leqslant y$ для обозначения того факта, что $x < y$ или $x = y$.

Целью сортировки является построение такой перестановки

$$\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\},$$

что в последовательности $A' = (k'_1, \ldots, k'_n)$, где $k'_{\pi(i)} = k_i$, ключи идут в неубывающем порядке:

$$k'_1 \leqslant k'_2 \leqslant \ldots \leqslant k'_n.$$

Иными словами, перестановка π для каждого индекса i в исходной последовательности показывает, на какую позицию следует поставить элемент k_i, с тем чтобы получилась неубывающая последовательность.

Отметим, что во входной последовательности могут встречаться равные ключи. Сортировка называется *устойчивой*, если из того, что $k_i = k_j$ при $i \leqslant j$, следует, что $\pi(i) \leqslant \pi(j)$. Иными словами, относительный порядок пар с равными ключами не меняется.

Напомним, что всю информацию о ключах наш алгоритм получает лишь с помощью обращения к оракулу сравнения. Поэтому такой подход называют *сортировкой, основанной на сравнениях* (*comparision sort*). Сложность алгоритма будет измеряться количеством запросов на сравнение, им осуществляемых.

При практической реализации алгоритм либо производит явное копирование и перестановку ключей k_i, либо поддерживает перестановку π, задающую перемещения ключей в неявной форме. В первом случае, вообще говоря, в результате образуется лишь последовательность k'_i, но не перестановка π. Второй метод предпочтителен, если размеры ключей велики.

Кроме того, ключи могут быть также снабжены дополнительными *данными* произвольной природы, так что вместо отдельных ключей k_i алгоритму приходится работать с парами (k_i, d_i).

2.2. Квадратичная сортировка

Начнем изучение с простейших алгоритмов сортировки, например с *сортировки выбором* (*selection sort*). Пусть задана последовательность ключей $A = (k_1, \ldots, k_n)$. План чрезвычайно прост: найдем в данной последовательности минимальный ключ (в случае, если таких ключей несколько, то любой из таковых). Для этого просканируем A, поддерживая текущий «рекорд» (т. е. ключ, представляющий собой минимум по уже просмотренной части последовательности A) и обновляя его очевидным образом.

Как только минимальный ключ будет обнаружен, выпишем его в качестве первого элемента результирующей последовательности A', вычеркнем из A и повторим процесс поиска минимума. После того как будет проделано n таких итераций, последовательность A опустеет, все ее элементы окажутся выписанными в порядке неубывания в виде последовательности A'.

Практическая реализация такого метода даже проще, чем его словесное описание. Дополнительных действий, связанных с удале-

нием выбранного элемента A, можно легко избежать применением *сортировки на месте* (*in-place sort*). Для этого нужно использовать один и тот же массив a для хранения последовательностей A и A'. А именно, последовательность A' будет занимать k начальных элементов в a, а A — последние $n-k$. Вначале $k := 0$. На очередном шаге производится поиск минимума среди последних $n-k$ элементов a, после чего найденный элемент переставляется с $a[k+1]$, значение k увеличивается на единицу, и при $k < n$ алгоритм переходит к новой итерации.

Упражнение 2.2.1. Реализуйте алгоритм сортировки выбором.

Очевидно, сложность такого алгоритма составляет $\Theta(n^2)$. Его основное преимущество — крайняя простота реализации и малая константа в квадратичной зависимости.

Изложенный «квадратичный» метод сортировки — далеко не единственный возможный. Подобных методов известно порядка десятка, упомянем только некоторые из них: *сортировка пузырьком* (*bubble sort*), *сортировка вставками* (*insertion sort*), *шейкер-сортировка* (*shaker sort*). Их описание можно найти, например, в книгах [1, 3].

2.3. Оптимальная сортировка, основанная на сравнениях

Конечно, при больших значениях n квадратичные методы сортировки оказываются неприменимыми. Основная цель данной главы — изучить алгоритмы, достигающие (в худшем или среднем случае) оценки сложности $O(n \log n)$. Мы рассмотрим два таких алгоритма: в разделе 2.4 речь пойдет о сортировке слиянием, а в разделе 2.5 — о быстрой сортировке. Еще один алгоритм (*сортировка кучей*) будет изложен в разделе 4.3 при изучении очередей с приоритетом.

Насколько оптимальны эти алгоритмы? Оказывается, для алгоритмов, основанных на сравнениях ключей, данная оценка асимптотически неулучшаема.

Теорема 2.3.1. *Всякий корректный алгоритм сортировки, основанный на сравнениях, должен совершить* $\Omega(n \log n)$ *сравнений в худшем случае.*

Доказательство. Воспользуемся так называемой *моделью разрешающих деревьев*. А именно, фиксируем длину n последовательности и какой-либо алгоритм сортировки $\mathcal{A}$. Мысленно запустим данный алгоритм на всех возможных последовательностях и понаблюдаем за ходом его выполнения и тем, какие он осуществляет сравнения. Результаты этих наблюдений представим в виде бинарного дерева T_n, устроенного следующим образом. Всякая внутренняя (не являющаяся листом) вершина помечена парой вида «$i\,?\,j$», что означает сравнение i-го элемента последовательности с j-м. Вершина имеет два сына: левый отвечает ситуации $a_i < a_j$, правый — $a_i \geqslant a_j$. Каждый лист дерева помечен некоторой перестановкой π.

Процесс работы алгоритма соответствует движению по данному дереву от корня к листу. Во всякой внутренней вершине алгоритм задает соответствующий вопрос оракулу сравнения и в зависимости от полученного ответа переходит либо в правое поддерево, либо в левое. Когда в процессе движения достигается лист, то сортировка выполнена, результатом ее работы считается записанная в листе перестановка.

Отметим, что каждый алгоритм $\mathcal{A}$ порождает набор $\{T_n\}$ таких деревьев, по одному дереву для каждого значения n. С другой стороны, обратного соответствия, сопоставляющего набору деревьев алгоритм, может не быть.

В дереве T_n не менее $n!$ листьев, поскольку ответом может служить любая перестановка чисел $\{1, \ldots, n\}$. Обозначим через h_n высоту дерева T_n, т. е. максимальную длину пути вниз от корня до листа, выраженную в количестве ребер. Это значение имеет простой смысл в терминах исходного алгоритма: h_n равно максимальному количеству сравнений, которое может выполнить алгоритм на входной последовательности длины n.

Наша задача — получить нижнюю оценку для h_n. Этого легко добиться, если заметить, что число листьев в дереве высоты h не превосходит 2^h. Следовательно, $n! \leqslant 2^{h_n}$, а значит, $h_n \geqslant \log_2 n!$. По формуле Стирлинга [2] получаем $h_n = \Omega(n \log n)$, что и требовалось доказать. □

Итак, любой алгоритм сортировки, основанный на сравнениях, совершает не менее $\Omega(n \log n)$ сравнений в худшем случае. Следовательно, и время его работы оценивается как $\Omega(n \log n)$. Еще раз

подчеркнем, что доказанная оценка распространяется лишь на алгоритмы, которые могут получать информацию об относительном порядке элементов исключительно путем попарных сравнений. В указанном смысле элементы ведут себя как «черные ящики», структура которых непрозначна для алгоритма сортировки. Если ослабить эти предположения, то более быстрая сортировка станет возможной.

2.4. Сортировка слиянием

Сортировка слиянием (*merge-sort*) основана на идее *разделяй и властвуй* (*divide and conquer*): исходная последовательность A длины n разделяется на две меньшие части A_1 и A_2, которые рекурсивно сортируются. Размеры частей выбираются максимально близкими, например, можно отнести произвольные $\lfloor n/2 \rfloor$ элементов к последовательности A_1, а оставшиеся $\lceil n/2 \rceil$ элементов — к последовательности A_2.

Пусть B_1 и B_2 обозначают упорядоченные последовательности, получаемые в результате сортировки последовательностей A_1 и A_2 соответственно. Тогда искомая последовательность B (результат сортировки A) получается путем *слияния* (*merge*) B_1 и B_2. Это слияние, как мы далее увидим, осуществимо за $O(n)$ сравнений.

Конечно, рекурсивное деление в данном алгоритме должно прерываться, как только в последовательности остается единственный элемент. (На практике его можно прекратить и раньше, как только число сортируемых ключей окажется меньше некоторого порога, начиная с которого эффективной оказывается «квадратичная» сортировка.)

Итак, осталось научиться сливать уже упорядоченные последовательности B_1 и B_2 в одну общую упорядоченную последовательность B. Указанные последовательности мы будем считать представленными одноименными массивами длины n_1, n_2 и $n = n_1 + n_2$ соответственно.

Будем поддерживать пару индексов i $(1 \leqslant i \leqslant n_1)$, j $(1 \leqslant j \leqslant n_2)$, отмечающих позиции, из которых будут выбираться очередные элементы сливаемых последовательностей, а также индекс k $(1 \leqslant k \leqslant n)$, хранящий текущую позицию в заполняемом массиве B. Исходно $i := 1$, $j := 1$, $k := 1$. Дальнейшие действия должны производиться

до тех пор, пока обе последовательности B_1 и B_2 не будут полностью просмотрены, т. е. до тех пор, пока не окажется, что $i > n_1$ и $j > n_2$. Инвариантом наших итераций будут следующие условия: $B[k-1] \leqslant B_1[i]$ и $B[k-1] \leqslant B_2[j]$. При этом условно считаем, что $B[0] = -\infty$, $B_1[n_1+1] = B_2[n_2+1] = +\infty$. Здесь $+\infty$ (соответственно $-\infty$) обозначает условный ключ, который больше (соответственно меньше) любого «обычного» ключа. Иными словами, последний выписанный элемент не превосходит очередного элемента как в последовательности B_1, так и в B_2.

Итерация алгоритма протекает следующим образом: в качестве очередного элемента $B[k]$ возьмем наименьший из $B_1[i]$ и $B_2[j]$ и продвинем соответствующие указатели. Формально говоря, если $B_1[i] < B_2[j]$, то положим $B[k] := B_1[i]$ и увеличим на единицу значения i и k. Если же $B_1[i] \geqslant B_2[j]$, то положим $B[k] := B_2[j]$, а затем увеличим на единицу j и k.

Введенный выше инвариант, во-первых, очевидно, сохраняется на протяжении итераций, а во-вторых, показывает, что построенная последовательность B оказывается упорядоченной по неубыванию. Число сравнений, осуществляемых алгоритмом, не превосходит числа итераций, т. е. $n_1 + n_2 = n$.

Если обозначить количество выполняемых им сравнений для n-элементного массива за $T(n)$, то оказывается справедливым рекуррентное соотношение

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n).$$

Отсюда следует, что $T(n) = O(n \log n)$ (см., например, [1]).

Существует также наглядный геометрический способ доказательства оценки $O(n \log n)$ на время работы алгоритма Merge-Sort. Для этого изобразим дерево рекурсии и заметим, что в пределах каждого уровня работа происходит с набором отрезков суммарной длины $O(n)$. Учитывая, что дерево имеет высоту $O(\log n)$, получаем, что общее время составляет $O(n \log n)$ (см. рис. 2.1[1]).

Сортировка слиянием (в том виде, как она была только что изложена) невозможна без использования дополнительной памяти. Нам

[1] Автор данного рисунка — Мануэл Кирш.

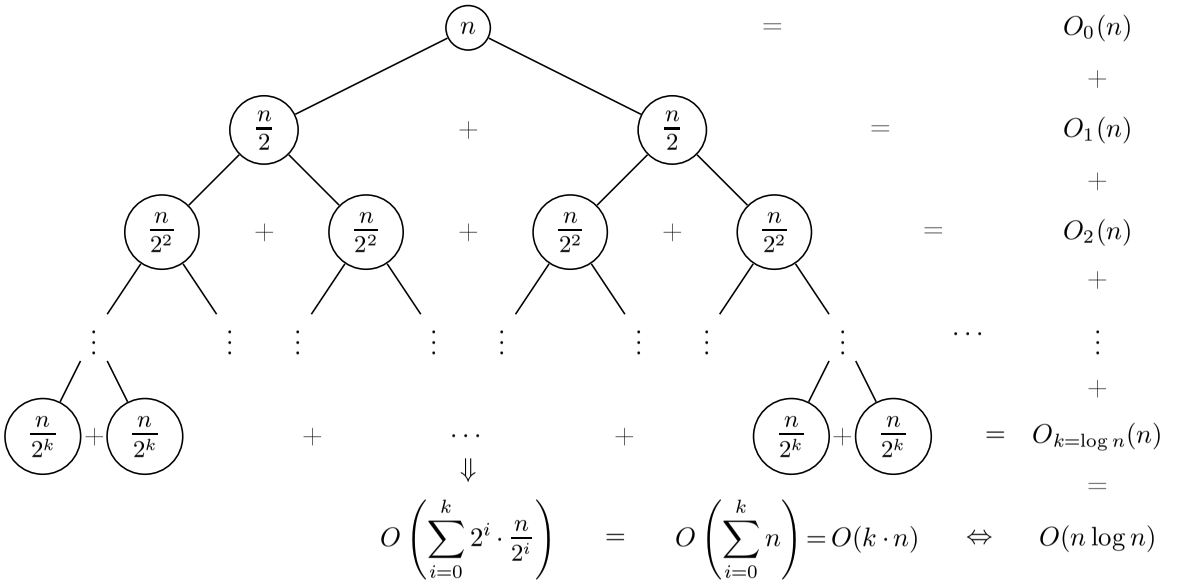

Рис. 2.1. Дерево рекурсии в алгоритме Merge-Sort

потребуется $O(n)$ ячеек для хранения результата операции слияния упорядоченных массивов. Существует, однако, вариант сортировки слиянием, способный переупорядочивать элементы на месте.

Упражнение 2.4.1. Реализуйте алгоритм сортировки слиянием с использованием рекурсии.

Упражнение 2.4.2. Разработайте и реализуйте нерекурсивный вариант алгоритма сортировки слиянием.

2.5. Быстрая сортировка

Алгоритм так называемой *быстрой сортировки* (QUICK-SORT), изобретенный Хоаром, несомненно, представляет собой один из наиболее популярных практических методов упорядочения массивов. В том или ином виде он реализован почти в любой библиотеке стандартных алгоритмов.

В основе метода быстрой сортировки лежит тот же принцип «разделяй и властвуй», что и в алгоритме сортировки слиянием. Разница, в первую очередь, заключается в том, что если MERGE-SORT разделяет ключи, подлежащие сортировке, на две части фактически произвольным образом, то QUICK-SORT отбирает ключи в две группы по особому принципу.

Итак, предположим, что нам потребовалось отсортировать массив A. Выберем один из ключей A в качестве *разделителя* (по-английски *pivot*) и обозначим его λ. Удалим данный ключ из A, а затем разделим оставшиеся ключи на две части A_1 и A_2 так, что в A_1 попадают ключи, не большие λ, а в A_2 — ключи, большие λ.

После того как разбиение A на части A_1 и A_2 построено, достаточно отсортировать A_1 и A_2, построив упорядоченные последовательности B_1 и B_2. Теперь результатом сортировки A будет последовательность $B_1 \circ (\lambda) \circ B_2$ (получающаяся выписыванием B_1, затем λ и наконец B_2).

Интересной особенностью, обуславливающей преимущество алгоритма QUICK-SORT перед MERGE-SORT, является тот факт, что все действия осуществляются без использования дополнительной памяти. Каждый рекурсивный вызов QUICK-SORT получает на вход пару

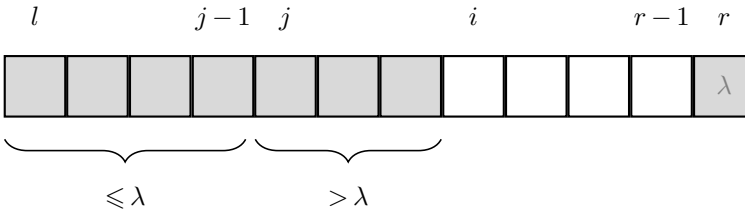

Рис. 2.2. Работа процедуры Partition

индексов l и r, задающую отрезок $[l, r]$ в массиве A, подлежащий сортировке. Если $l \geqslant r$, то никакие действия не требуются.

В противном случае некоторым способом (который будет объяснен далее) выбирается индекс p $(l \leqslant p \leqslant r)$, и тем самым назначается *разделитель* $\lambda = A[p]$. Мы хотим переставить элементы на отрезке $A[l : r]$ так, чтобы сначала шла часть A_1, затем λ и наконец A_2.

Вначале произведем обмен $A[p] \leftrightarrow A[r]$, поместив разделитель в конец отрезка, и постараемся разделить оставшиеся элементы $A[l : r-1]$ на две части. Будем просматривать отрезок $A[l : r-1]$ последовательно слева направо, обозначая через i текущий индекс. Будем также хранить индекс j начала области, содержащий ключи из группы A_2. На любой итерации поддерживаются следующие свойства (см. рис. 2.2):

$$\begin{aligned} &1)\ \text{если } l \leqslant k < j, \text{ то } A[k] \leqslant \lambda; \\ &2)\ \text{если } j \leqslant k < i, \text{ то } A[k] > \lambda. \end{aligned} \tag{2.1}$$

В начале первой итерации $i = j = l$, A_1 и A_2 пусты, и условия (2.1) тривиально выполнены. Сравним $A[i]$ с λ и решим, в какую часть мы помещаем этот ключ. Если $A[i] \leqslant \lambda$, то необходимо добавить $A[i]$ в группу A_1. Для этого обменяем $A[j]$ с $A[i]$ и увеличим i и j на единицу. Таким образом, $A[i]$ окажется в множестве A_1, а перемещенный элемент $A[j]$ по-прежнему остается в A_2. Если же $A[i] > \lambda$, то просто увеличим i, тогда $A[i]$ попадет в A_2.

По окончании итераций все элементы $A[l : r-1]$ будут просмотрены, и i окажется равным r. Требуемое перераспределение ключей почти выполнено. Осталось только поместить разделитель λ между A_1 и A_2. Для этого выполним обмен $A[j] \leftrightarrow A[r]$. Полученный отрезок

$A[l:r]$ обладает следующими свойствами:
1) если $l \leqslant k < j$, то $A[k] \leqslant \lambda$;
2) если $k = j$, то $A[k] = \lambda$;
3) если $j < k \leqslant r$, то $A[k] > \lambda$.

Теперь достаточно выполнить рекурсивные вызовы Quick-Sort для подотрезков $A[l:j-1]$ и $A[j+1:r]$, результаты их работы будут «склеены» автоматически. Отметим, что разделитель (который оказался в j-й позиции массива) не участвует в дальнейшей сортировке.

Описание алгоритма Quick-Sort завершено, но нужно также объяснить, каким образом выбирается разделитель λ. От того, насколько «качественно» данный выбор производится, существенно зависит сложность. Желательно, чтобы при каждом разделении отрезка $A[l:r]$ на $A[l:j-1]$ и $A[j+1:r]$ части были близки по размеру. В противном случае время работы может стать квадратичным. Такой эффект будет наблюдаться, например, если на каждом шаге разделителем будет оказываться минимальный или максимальный элемент отрезка $A[l:r]$. Тогда одна из частей разбиения будет содержать все элементы отрезка, кроме одного, а другая будет пустой.

В качестве одного из вариантов можно брать разделитель из середины отрезка, т. е. $\lambda := A[p]$ для $p = \lfloor (l+r)/2 \rfloor$. Такой выбор неплох, но существует совсем простой пример входного массива, на котором время работы квадратично.

Упражнение 2.5.1. Докажите, что на массиве вида

$$(1, 2, \ldots, n-1, n, n-1, n-2, \ldots, 2, 1)$$

изложенный способ выбора разделителя дает квадратичную сложность алгоритма.

Можно воспользоваться *рандомизацией*: брать в качестве p случайное число от l до r (с равномерным распределением на множестве вариантов). Оказывается, математическое ожидание числа сравнений при такой стратегии растет как $O(n \log n)$.

Теорема 2.5.1. *Пусть на вход алгоритма* Quick-Sort *поступает n различных ключей. Тогда матожидание времени его работы при случайном равномерном и независимом выборе разделителя составляет $O(n \log n)$. Математическое ожидание глубины рекурсии при этом оценивается как $O(\log n)$.*

Доказательство. Обозначим через $T(n)$ математическое ожидание времени, которое требуется алгоритму для сортировки массива размера n. Процедура Partition занимает время $O(n)$ и разделяет массив на две части с размерами i и $n-i-1$. Учитывая, что разделитель выбирается равномерно, заключаем, что каждое из значений i от 1 до $n-1$ имеет одинаковые шансы. Тем самым получаем соотношение

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

(Слагаемые, отвечающие $i=0$ и $i=1$, были опущены, можно считать, что их включает член $O(n)$.)

Обозначим через $\alpha > 0$ скрытую константу в слагаемом $O(n)$. Докажем (используя индукцию по n), что найдется константа $\beta > 0$, для которой справедливо неравенство $T(n) \leqslant \beta n \log n$ (для всех $n \geqslant 2$).

База индукции тривиальна, рассмотрим шаг. Используя индуктивное предположение и введенное обозначение α, перепишем рекуррентное соотношение следующим образом:

$$T(n) \leqslant \alpha n + \frac{2}{n-1}\sum_{i=2}^{n-1} \beta i \log i.$$

Положим $n' := \lfloor n/2 \rfloor$ и оценим сумму отдельно при $i \leqslant n'$ и при $i > n'$:

$$\begin{aligned}
T(n) &\leqslant \alpha n + \frac{2\beta}{n-1}\left(\sum_{i=2}^{n'} i\log i + \sum_{i=n'+1}^{n-1} i \log i\right) \leqslant \\
&\leqslant \alpha n + \frac{2\beta}{n-1}\left(\log\frac{n}{2}\cdot\sum_{i=2}^{n'} i + \log n \cdot \sum_{i=n'+1}^{n-1} i\right) \leqslant \\
&\leqslant \alpha n + \frac{2\beta}{n-1}\left(\log n \cdot \sum_{i=2}^{n-1} i - \sum_{i=2}^{n'} i\right) = \\
&= \alpha n + \frac{2\beta}{n-1}\left(\log n \cdot \frac{(n+1)(n-2)}{2} - \frac{(n'+2)(n'-1)}{2}\right) \leqslant \\
&\leqslant \alpha n + \beta(n+1)\log n - \beta\frac{n-4}{4} = \\
&= \beta n \log n + \left(\alpha n + \beta \log n - \beta\frac{n-4}{4}\right).
\end{aligned}$$

Из последней оценки видно, что, выбирая β достаточно большим, можно добиться выполнения неравенства $T(n) \leqslant \beta n \log n$, что и требовалось.

Переходим теперь к оценке средней глубины рекурсии. Обозначая через $D(n)$ математическое ожидание этой глубины, получаем соотношение

$$D(n) = 1 + \frac{1}{n-1} \sum_{i=0}^{n-1} \max(D(i), D(n-i-1)).$$

Аналогичные индуктивные рассуждения, которые мы оставляем читателю, показывают, что справедлива оценка $D(n) = O(\log n)$. □

Важно подчеркнуть, что здесь идет речь о математическом ожидании по датчику случайных чисел, используемому алгоритмом, а не по какому-либо «распределению на входе». Можно, впрочем, сформулировать и аналогичный результат для случайных ключей на входе.

Теорема 2.5.2. *Рассмотрим вариацию алгоритма* Quick-Sort*, детерминированно выбирающего в качестве разделителя первый элемент текущего отрезка. Пусть на вход алгоритму поступает случайная последовательность, в которой все ключи различны, а все их перестановки равновероятны. Тогда математическое ожидание времени работы алгоритма будет равно $O(n \log n)$, а для математического ожидания глубины рекурсии справедлива оценка $O(\log n)$.*

Доказательство аналогично теореме 2.5.1.

Практическая разница между двумя доказанными теоремами фундаментальна: относительно внутреннего датчика случайности не бывает «плохих» входных данных. Поэтому даже если рандомизированный алгоритм работает медленно при данном конкретном наборе значений, полученных от датчика, при повторном запуске ситуация может исправиться. В противоположность этому, детерминированный алгоритм, получивший на вход «неудачные» данные, обречен на медленную работу и при повторных запусках.

Помимо затрат по времени следует также учитывать и затраты по памяти. Каковы они в случае алгоритма Quick-Sort? Поскольку в процессе работы он не выделяет дополнительной памяти, а переставляет элементы на месте, эти затраты связаны исключительно

с поддержанием стека рекурсии. Теоремы 2.5.1 и 2.5.2 утверждают, что математическое ожидание глубины стека логарифмическое. Следовательно, в среднем дополнительные расходы памяти невелики.

Существует также простой способ добиться того, чтобы логарифмическая оценка была справедлива не в среднем, а в худшем случае. Рассмотрим псевдокод алгоритма, представленный на рис. 2.3 (а). Процедура Quick-Sort заканчивается рекурсивным вызовом самой себя. Подобную ситуацию называют *хвостовой рекурсией* (*tail recursion*). Этот «хвостовой» вызов можно удалить, или, как говорят, *элиминировать* (*eliminate*) следующим образом: выполним присваивание $l := j + 1$ и передадим управление на начало процедуры. (При подобном объяснении может показаться, что мы предлагаем читателю воспользоваться конструкцией `goto`. Можно, конечно, поступить и так, но лучше обернуть тело процедуры в цикл.)

Итак, последний рекурсивный вызов может быть устранен. Это, к сожалению, не дает никаких особых гарантий на глубину рекурсии в худшем случае. Действительно, несложно представить ситуацию, когда все разбиения, которые строит алгоритм, оказываются сильно несбалансированными, так что первый (левый) подотрезок намного больше правого. В этом случае глубина рекурсии будет линейной.

Заметим теперь, что мы сами вправе выбирать, какой из рекурсивных вызовов делать первым, а какой — вторым. Переставим эти вызовы так, чтобы больший подотрезок обрабатывался последним (см. рис. 2.3 (б)). Произведем в представленном алгоритме элиминацию хвостовой рекурсии. Глубина стека вызовов даже в худшем случае будет составлять $O(\log n)$, поскольку при каждом рекурсивном вызове длина текущего отрезка сокращается как минимум в два раза.

Вышеописанные модификации позволяют сделать алгоритм Quick-Sort действительно быстрым и оправдывающим свое название. Однако всегда остается шанс, что неудачный подбор входных данных (или — в случае внутренней рандомизации — датчика случайности) приведет к тому, что число операций, совершаемых алгоритмом, сильно превысит оценку математического ожидания $O(n \log n)$. Для приложений, критичных к быстродействию, такую опасность нельзя игнорировать. Поэтому большинство качественных библиотечных реализаций алгоритма использует его вариацию, изобретенную Мас-

```
void QuickSort(A, l, r) {
   ...
   QuickSort(A, l, j-1);
   QuickSort(A, j+1, r);
}
```

(а)

```
void QuickSort(A, l, r) {
   ...
   if (j >=2 * (l+r))
 {
   QuickSort(A, l, j-1);
   QuickSort(A, j+1, r);
 }
 else
 {
   QuickSort(A, j+1, r);
   QuickSort(A, l, j-1);
 }
}
```

(б)

Рис. 2.3. Подготовка к элиминации хвостовой рекурсии в алгоритме Quick-Sort

сером и называемую Intro-Sort. Суть ее весьма проста — запустим обычный алгоритм Quick-Sort, выбирая в качестве разделителя на каждом шаге медиану из трех элементов текущего отрезка массива: левой границы, правой границы и середины. Кроме того, будем в процессе рекурсии подсчитывать ее глубину, и если она превысит значение $c \cdot \log_2 n$ для подходящим образом выбранной константы c, то объявим попытку отсортировать массив по методу быстрой сортировки неудачной. Конечно, мы не собираемся при этом сдаваться и сообщать пользователю о постигшей неудаче. Вместо этого запустим другой алгоритм, для которого $O(n \log n)$ является оценкой худшего случая. В качестве такого «запасного» алгоритма обычно используют сортировку кучей, см. раздел 4.3.

В большинстве случаев алгоритм Intro-Sort сводится к алгоритму Quick-Sort, преимущество которого состоит в весьма небольшой скрытой константе асимптотики. Если Quick-Sort прерывается из-за превышения глубины рекурсии, то, как несложно понять, затраченное время к тому моменту составляет $O(n \log n)$. Поэтому запуск еще одного алгоритма с оценкой сложности $O(n \log n)$ не повлияет на асимптотику. В итоге сложность в худшем случае составляет $O(n \log n)$.

2.6. Порядковые статистики

Еще одна задача, тесно связанная с сортировкой, — это поиск так называемых *порядковых статистик*. А именно, предположим, что известна последовательность из n ключей, представленная массивом A. Ключи, как и ранее, предполагаются взятыми из некоторого линейно упорядоченного универсума. По определению *k-й порядковой статистикой* (при $1 \leqslant k \leqslant n$) называют k-й в порядке возрастания ключ. Иными словами, если массив A отсортировать, то k-я порядковая статистика окажется равной $A[k]$. Задача состоит в том, чтобы по заданным A и k найти k-ю порядковую статистику.

В общем случае, конечно, можно просто упорядочить весь массив A за время $O(n \log n)$ — и задача решена. Но при $k = 1$ получаем задачу поиска минимума, а при $k = n$ — задачу поиска максимума. Обе они могут быть решены за линейное время. Возникает естественный вопрос, нельзя ли добиться линейной сложности в общем случае.

Ответ оказывается положительным. Вначале опишем рандомизированный алгоритм, который находит k-ю порядковую статистику за линейное в среднем время. Этот метод (назовем его RANDOM-SELECT) похож на алгоритм QUICK-SORT. На вход ему поступают массив A, пара индексов l и r, отмечающих отрезок в A, на котором предлагается искать статистику, а также число k — номер искомой статистики ($1 \leqslant k \leqslant r - l + 1$).

Если $l = r$, то с необходимостью $k = 1$, и ответ готов. Иначе, как и в случае с QUICK-SORT, вначале мы выбираем разделитель λ. Поскольку уже было объявлено, что наш алгоритм рандомизированный, просто положим $\lambda := A[p]$, где p — случайное равномерно распределенное на отрезке $[l, r]$ значение. Затем производится вызов процедуры PARTITION, в результате чего отрезок $A[l : r]$ переупорядочивается и разбивается на две части; обозначим их через $A[l : m]$ и $A[m + 1 : r]$. Процедура PARTITION гарантирует, что элементы в первой части не превосходят λ, в то время как элементы во второй части не меньше λ. (Разделитель λ был произвольным образом отнесен к одной из этих двух половин.)

Алгоритм QUICK-SORT в данной ситуации закончил бы работу парой рекурсивных вызовов. Но мы поступим иначе. Нас интересует

не вся упорядоченная последовательность, а лишь значение, которое находится на k-й позиции. Поэтому посмотрим, в какой из двух подотрезков эта позиция попадает. В случае $k \leqslant m - l + 1$ искомый ответ находится в левом подотрезке, а иначе в правом. Поэтому вместо пары рекурсивных вызовов (совершаемых алгоритмом QUICK-SORT) достаточно выполнить лишь один, относящийся к нужному подотрезку. Можно также вспомнить про прием элиминации хвостовой рекурсии и переписать алгоритм в нерекурсивной форме.

Упражнение 2.6.1. Реализуйте рекурсивную и нерекурсивную версии алгоритма RANDOM-SELECT.

Расходы дополнительной памяти в описанном методе равны $O(1)$, а временну́ю сложность оценим следующим образом.

Теорема 2.6.1. *Математическое ожидание времени работы алгоритма* RANDOM-SELECT *составляет* $O(n)$.

Доказательство. Обозначим через $T(n)$ математическое ожидание времени работы на массиве длины n. Тогда по аналогии с анализом алгоритма QUICK-SORT получаем

$$T(1) = O(1),$$

$$T(n) = O(n) + \frac{1}{n-1}\sum_{i=1}^{n-1} T(i) \quad \text{при } n \geqslant 2.$$

Более точно, для некоторой константы $\alpha > 0$ при $n \geqslant 2$ справедливо неравенство

$$T(n) \leqslant \alpha n + \frac{1}{n-1}\sum_{i=1}^{n-1} T(i).$$

Будем доказывать неравенство $T(n) \leqslant \beta n$ (для подходящей константы $\beta > 0$) индукцией по n. Базу индукции можно получить, выбрав β достаточно большим, чтобы учесть время $T(1)$. Шаг индукции таков:

$$T(n) \leqslant \alpha n + \frac{1}{n-1}\sum_{i=1}^{n-1} \beta i = \Big(\alpha + \frac{\beta}{2}\Big)n.$$

Необходимо, чтобы выполнялось неравенство $\alpha + \dfrac{\beta}{2} \leqslant \beta$, а этого легко достичь, взяв β достаточно большим. $\square$

В заключение покажем, как находить k-ю порядковую статистику за линейное *в худшем случае* время. Новый алгоритм (назовем его Linear-Select) строится по аналогии с Random-Select, но не использует рандомизированный выбор разделителя. Вместо этого нам бы хотелось использовать в качестве разделителя медиану значений текущего отрезка массива. Здесь, конечно, немедленно возникает трудность — как найти эту медиану за линейное время? Эта подзадача вновь представляет собой вопрос о порядковой статистике. И хотя на этот раз значение k весьма специфическое (оно равно половине количества элементов), не видно причины, по которой это свойство могло бы оказаться полезным.

Для решения возникшего затруднения внесем в наш первоначальный план два уточнения:

1) будем искать не точную, а *приближенную* медиану;
2) поиск разделителя будем производить с помощью рекурсивного вызова Linear-Select.

Опишем детали более подробно. Пусть B обозначает текущий отрезок исходного массива. Положим $n := |B|$ и разобьем значения B произвольным образом на $n/5$ пятерок b_{ij}, $i = 1, \ldots, n/5$, $j = 1, \ldots, 5$. В случае, если значение n не делится на 5, добавим к B нужное число фиктивных элементов (бóльших максимального имеющегося). Для простоты анализа также будем предполагать, что все числа в массиве B различны.

Каждую пятерку $b_{i1}, \ldots, b_{i5}$ упорядочим простыми вставками за время $O(n)$ и будем далее считать, что

$$b_{i1} < b_{i2} < b_{i3} < b_{i4} < b_{i5} \quad \text{для всех } i = 1, \ldots, n/5.$$

Рассмотрим элементы b_{i3}, $i = 1, \ldots, n/5$, — они представляют собой медианы пятерок. Рекурсивно запустим для этих элементов алгоритм Linear-Select, попросив его найти медиану этих медиан. Обозначим полученное значение через λ.

Произведем вызов процедуры Partition для отрезка B относительно λ, переставив в нем элементы и разбив его на два: B_1 и B_2. После этого задача поиска порядковой статистики сведется к меньшей по размеру, вопрос лишь в том, насколько меньшей.

Лемма 2.6.2. *На отрезке B как минимум $3n/10 + O(1)$ элементов строго меньше λ и аналогично как минимум $3n/10 + O(1)$ элементов строго больше λ.*

Доказательство. Всего медиан пятерок $n/5 + O(1)$, среди них $n/10 + O(1)$ строго меньше λ. В каждой пятерке $b_{i1}, \ldots, b_{i5}$, медиана которой b_{i3} меньше λ, есть как минимум три элемента, меньших λ, — это b_{i1}, b_{i2}, b_{i3}. Всего, тем самым, нашлось $3n/10 + O(1)$ элементов отрезка B, которые меньше λ. Для элементов, бóльших λ, рассуждения аналогично. □

Итак, процедура PARTITION не обязана делить отрезок на два равных по длине, но сильного дисбаланса возникнуть не может:

$$\max(|B_1|, |B_2|) \leqslant n - 3n/10 + O(1) = 7n/10 + O(1).$$

Пренебрегая слагаемым $O(1)$, можно сказать, что за одну итерацию мы уменьшили размер текущего множества кандидатов не менее чем в $10/7$ раз. Это наблюдение, впрочем, не влечет за собой немедленных гарантий, что время работы линейно. Дело в том, что выбор медианы пятерок требует рекурсивного вызова алгоритма LINEAR-SELECT, так что итоговая оценка получается рекуррентной.

А именно, если через $T(n)$ обозначить время работы алгоритма LINEAR-SELECT на массиве размера n и пренебречь слагаемыми вида $O(1)$ и той опасностью, что n не разделится нацело на 10, то возникает соотношение

$$T(n) = T(n/5) + T(7n/10) + O(n).$$

Поскольку $1/5 + 7/10 = 9/10$, получаем, что $T(n) = O(n)$, как это следует из общей теории (см. [1]).

Упражнение 2.6.2. Реализуйте алгоритм LINEAR-SELECT.

2.7. Задачи

Задача 1. В очереди к врачу сидят n пациентов. Про каждого из них заранее известно, какое время уйдет на визит к врачу. Для i-го пациента это время равно t_i. Врач может принимать не более одного пациента одновременно. Необходимо принять всех пациентов таким образом, чтобы суммарное время, которое ожидал приема каждый из пациентов, было минимальным.

В первой строке входа находится целое число $n, 1 \leqslant n \leqslant 100\,000$, во второй строке — n целых чисел $t_i, 1 \leqslant t_i \leqslant 100\,000$.

Выведите единственное число — минимальное суммарное время ожидания всех пациентов.

Пример входа	Пример выхода
5 3 2 5 4 1	20
1 100000	0

Решение. Ясно, что пациентов необходимо принимать подряд, без перерывов, так как из-за перерывов суммарное время ожидания может только увеличиваться. Осталось определить, в каком порядке врачу нужно принимать пациентов.

Предположим, что есть два таких пациента, i-й и j-й, что $t_i > t_j$, при этом врач примет в какой-то момент i-го пациента, а сразу после него — j-го. Посмотрим, как изменится суммарное время ожидания пациентов, если поменять i-го пациента с j-м местами в порядке приема. Время ожидания для всех пациентов до i-го никак не изменится. Время ожидания для всех пациентов после j-го тоже никак не изменится, так как в сумме i-го с j-м будут принимать столько же времени, сколько и раньше. Время ожидания i-го пациента увеличится на t_j, а время ожидания j-го уменьшится на t_i. Итого, сумма уменьшится на $t_i - t_j$. Таким образом, невыгодно обслуживать более «медленного» пациента перед более «быстрым».

Значит, необходимо упорядочить всех пациентов по возрастанию времени приема и принимать в таком порядке. Для этого отсортируем все числа t_i по возрастанию — получим набор $t'_1, t'_2, t'_3, \ldots, t'_n$. Теперь осталось только посчитать ответ. Мы знаем, что 1-й пациент никого не ждет, 2-й ждет 1-го t'_1 времени, 3-й ждет первых двоих $t'_1 + t'_2$ времени и т. д., n-й ждет $t'_1 + t'_2 + \ldots + t'_{n-1}$ времени. Суммарное время ожидания тогда равно $(n-1)t'_1 + (n-2)t'_2 + \ldots + t'_{n-1}$.

Сложность такого решения равна $O(n \log n)$, так как столько времени уходит на сортировку массива, а ответ потом считается в одном

цикле за время $O(n)$. Памяти требуется $O(n)$ на хранение массива и константа дополнительной памяти.

Деталь реализации: ответ в задаче может превзойти ограничения 32-битного типа, так как при $n = 100\,000$ и $t_1 = t_2 = \ldots = t_n = 100\,000$ он равен

$$\frac{n(n-1)}{2} t_1 = \frac{100000 \cdot 99999 \cdot 100000}{2} > 2^{31} - 1.$$

Поэтому для подсчета ответа необходимо использовать 64-битный целый тип. Более того, даже выражение $(n-1)t_1'$ может быть больше $2^{31} - 1$, поэтому перед умножением необходимо привести один из аргументов к 64-битному типу.

Задача 2. Даны последовательность $A_1, \ldots, A_n$, состоящая из n различных целых чисел, и число S. Найдите все такие пары индексов (i, j), что $1 \leqslant i < j \leqslant n$ и $A_i + A_j = S$.

В первой строке входа находятся два целых числа n и S; $1 \leqslant n \leqslant 1\,000\,000$, $-10^9 \leqslant S \leqslant 10^9$, во второй строке — n целых чисел A_i, $-10^9 \leqslant A_i \leqslant 10^9$.

Выведите количество пар (i, j) указанного вида.

Пример входа	Пример выхода
5 6 1 2 3 4 5	2
5 2 1 1 1 1 1	10
6 3 1 2 1 2 1 2	9

Решение. Простейший способ решить задачу таков: переберем все такие пары (i, j), что $1 \leqslant i < j \leqslant n$, и для каждой из них проверим, верно ли, что $A_i + A_j = S$. Количество подходящих пар и есть ответ. Однако это решение работает за время $O(n^2)$, что неприемлемо при ограничениях $1 \leqslant n \leqslant 1\,000\,000$.

Чтобы ускорить алгоритм, сначала отсортируем все числа в массиве A. Ясно, что ответ от этого никак не изменится. Теперь установим

два указателя l и r: l изначально будет указывать на первый элемент в A, а r — на последний.

Будем поддерживать следующий инвариант:

$$\begin{aligned}&1)\ r > l,\\ &2)\ \text{если } A_l + A_n < S, \text{ то } r = n;\\ &3)\ \text{иначе } r - \text{минимальный такой индекс, что } r > l \text{ и}\\ &\quad A_l + A_r \geqslant S.\end{aligned} \tag{2.2}$$

Отметим, что если $A_1 + A_n \geqslant S$, то нам придется уменьшить начальное значение r, чтобы выполнить условия (2.2).

Далее, при фиксированном l, если инвариант (2.2) выполнен, а $A_l + A_r \neq S$, то не существует пары индексов (l, m), содержащей l в качестве первого элемента и удовлетворяющей условию задачи. Если же $A_l + A_r = S$, то пара (l, r) удовлетворяет условию, а никакая другая пара вида (l, m) не удовлетворяет, так как все числа A_i различны.

Будем двигать l вправо начиная с самого первого элемента. На каждом шаге сначала определим, подходит ли текущее r в пару к l, и если подходит, то добавим единицу к ответу. Далее увеличим l, а затем, чтобы поддержать инвариант, нам, возможно, потребуется несколько раз сдвинуть r влево, так как A_l увеличилось, значит, и $A_l + A_r$ увеличилось, и $A_l + A_n$ увеличилось. Однажды нам может понадобиться сдвинуть r вправо, чтобы выполнялось условие $r > l$, но в этот момент можно завершить выполнение алгоритма, так как для всех l начиная с этого уже не найдется такого значения $r > l$, что $A_l + A_r = S$.

Будем сдвигать r влево, до тех пор пока инвариант (2.2) не выполнится. Затем опять проверим, подходит ли r в пару к l и т. д. Таким образом, мы для каждого l определим, есть ли для него (единственно возможное) r в пару или нет.

Сложность этого решения составляет $O(n \log n)$, так как столько времени уходит на сортировку, а далее требуется только $O(n)$ действий для получения результата. Действительно, в нашем алгоритме l двигается только вправо, а r только влево, и при этом всегда $r > l$, значит, они оба сдвинутся в сумме не более n раз, а на каждый сдвиг и проверку уходит константное количество действий. Памя-

ти требуется $O(n)$ на массив и константный объем дополнительной памяти.

Заметим, что ответ к задаче помещается в ограничения 32-битного целого типа, так как для каждого l есть не более одного r в пару, а значит, ответ не превосходит n. Если бы в массиве могли быть одинаковые числа, то, во-первых, ответ в худшем случае мог быть C_n^2, а во-вторых, в таком случае уже нельзя было бы всегда передвигать r только влево. Первая проблема решается использованием 64-битного типа. А для решения второй проблемы пришлось бы после сортировки одинаковые числа, идущие подряд, пересчитать и записать в параллельный массив, после чего считать, что числа в массиве уже обязательно одинаковые, но у них есть кратности, и при добавлении к ответу учитывать кратность обоих чисел в паре.

Упражнение 2.7.1. Сформулируйте и решите аналогичную задачу, в которой числа A_i не обязаны быть различными.

Задача 3. Дано множество отрезков с целыми концами на прямой. Найдите наибольшее количество отрезков, имеющих общую точку.

В первой строке входа находится целое число n, $1 \leqslant n \leqslant 1\,000\,000$, а в каждой из следующих n строк — по два целых числа a и b, $-10^9 \leqslant a < b \leqslant 10^9$.

Выведите наибольшее количество отрезков, которые имеют общую точку.

Пример входа	Пример выхода
3 0 1 0 2 1 2	3
3 0 1 2 3 4 5	1
3 0 1 1 3 -100 -99	2

Решение. Преобразуем каждый из $2n$ концов отрезков в пару вида (x, b), где x — координата конца отрезка, $b = 0$ для левого конца и $b = 1$ для правого.

После этого отсортируем все пары в лексикографическом порядке (сначала по x, потом по b). Теперь все концы отрезков будут упорядочены по координате, а среди концов с равными координатами сначала будут идти левые концы отрезков, а потом правые.

Установим счетчик количества отрезков, покрывающих данную точку, в нуль, и будем перемещать воображаемую точку слева направо из $-\infty$ в $+\infty$. Нас будут интересовать не все положения точки, а только те моменты, когда она совпадает с одним из концов отрезков. Когда точка поравняется с левым концом некоторого отрезка, это означает, что соответствующий отрезок покрывает эту точку. Если она поравняется с правым концом, то соответствующий отрезок больше эту точку уже не накрывает. Здесь важно, что после сортировки среди концов с одинаковой координатой сначала идут левые концы, поэтому, просмотрев все левые концы с координатой x, можно дальше считать, что правые концы с этой координатой уже не принадлежат отрезкам. Таким образом, текущая точка оказывается уже не ровно на позиции x, а на позиции $x + \epsilon$, которую покрывают все отрезки, кроме тех, у которых правый конец имеет координату не более x.

Итак, устанавливаем счетчик в 0, затем проходим по массиву с концами отрезков и при нахождении левого конца увеличиваем счетчик на единицу, при нахождении правого — уменьшаем на единицу. В процессе этого находим максимальное значение счетчика за все время — это и будет ответ к задаче.

Сложность алгоритма составляет $O(n \log n)$, именно столько времени уйдет на сортировку. Дальнейшие действия занимают линейное время. Дополнительной памяти требуется $O(n)$ для хранения массива пар (x, b).

Глава 3. Поиск

3.1. Введение

В простейшем случае задачу точного поиска можно описать следующим образом: пусть задано множество A, состоящее из *ключей*. Их природа в данный момент не важна для нас, достаточно лишь того, что произвольную пару ключей можно проверить на равенство. Требуется по заданному ключу a выяснить, лежит ли a в A.

При оценке сложности будем предполагать, что каждая проверка ключей на равенство осуществима за время $O(1)$. Это предположение верно, скажем, если ключи представляют собой целые числа (размер которых не превосходит машинного слова), но, очевидно, нарушается в более сложных случаях. Скажем, для строк проверка выполняется за время, пропорциональное длине. В этом случае оценки сложности алгоритмов необходимо соответствующим образом скорректировать, и мы оставляем это читателю в качестве упражнения.

Конечно, многое зависит от того, как именно представлено множество A, может ли оно изменяться в процессе работы, есть ли время на предобработку и т. д. В случае, когда помимо вышеуказанной проверки на принадлежность можно добавлять и удалять элементы из A, говорят, что соответствующая структура данных реализует интерфейс *множества* (*set*).

Более общей является ситуация, когда каждому ключу из множества A сопоставлены *данные* (конкретная природа которых не важна). Тем самым A можно рассматривать как множество пар вида (key, $data$) (где key — ключ, а $data$ — отвечающие ему данные; при этом в A не может присутствовать двух пар с равными ключами) или же как частичное отображение из множества ключей в множество данных. В таком случае запрос на поиск состоит в вычислении образа заданного ключа при указанном отображении. Конечно же, при добавлении в A теперь нужно указывать не только ключ, но

еще и отвечающие ему данные. При удалении достаточно указывать лишь ключ. Структура данных, способная выполнять эти операции, реализует интерфейс *словаря* (*dictionary*, *map*), или, как еще говорят, *ассоциативной таблицы* (*associative table*).

В данной главе мы рассмотрим различные эффективные способы реализации таких структур. Здесь и далее через n будет обозначаться количество элементов (отдельных ключей или пар, состоящих из ключей и данных) в множестве A. Для простоты изложения будем считать, что мы реализуем интерфейс множества; переход к интерфейсу словаря делается тривиальным образом.

3.2. Линейный поиск

Выше было указано, что от ключей требуется лишь возможность проверять их на равенство. Конечно, при столь слабых предположениях невозможно получить сколь-либо нетривиальные результаты. А именно, для поиска в множестве A придется перебрать все его элементы. Итак, получаем оценку $O(n)$, которая, очевидно, достигается, если искомого ключа в A нет. Такой метод называют *линейным поиском*.

Как именно следует хранить элементы множества A? Если множество статично, то подойдет обычный массив фиксированного размера. Если элементы лишь добавляются в A, то можно использовать массив переменного размера, записывая новые значения в конец. Наконец, если структура должна поддерживать как добавление, так и удаление, то подойдет двусвязный список. В каждом из этих случаев добавление и удаление будет отнимать $O(1)$ единиц времени (для массива переменного размера эта оценка будет учетной).

3.3. Бинарный поиск

Нетривиальные результаты можно получить, если ввести дополнительные предположения о том, что собой представляют ключи. Пусть все возможные ключи образуют линейно упорядоченное множество (см. гл. 2). Рассмотрим сначала простейший случай, когда множество A известно заранее и не меняется. Тогда с помощью изученных ранее алгоритмов сортировки его элементы можно упорядочить. Та-

ким образом, $A = \{a_1, \ldots, a_n\}$, где $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. Задача поиска элемента в упорядоченной последовательности эффективно решается с помощью широкоизвестного алгоритма *бинарного поиска* (*binary search*).

А именно, пусть необходимо выяснить, присутствует ли в множестве A ключ k. Будем поддерживать пару индексов i, j, обладающих следующими свойствами:

$$\begin{aligned} &1)\ 1 \leqslant i < j \leqslant n+1; \\ &2)\ \text{если } k \in A, \text{ то } k = a_p \text{ для некоторого } p,\ i \leqslant p < j. \end{aligned} \tag{3.1}$$

Изначально положим $i := 1$, $j := n+1$. Свойство (3.1) тривиально выполнено. Далее произведем последовательность итераций следующим образом. Если $i = j - 1$, то из свойства (3.1) следует, что либо $k = a_i$, либо $k \notin A$. Сравнение k с a_i завершает поиск.

Иначе пусть $i < j - 1$. Положим $m := \left\lfloor \frac{i+j}{2} \right\rfloor$, тогда, очевидно, $i < m < j$. Сравним a_m с k. При этом возможны два случая:

1) если $a_m \leqslant k$, то положим $i := m$;
2) если $a_m > k$, то положим $j := m$.

Докажем, что инвариант (3.1) сохраняется. В первом случае при $a_m = k$ ключ k встречается на отрезке $[m, j-1]$, так что заключение импликации, а вместе с ним — и вся импликация истинны. Если $a_m < k$, то в силу упорядоченности последовательности ключ k не может встречаться на отрезке $[i, m-1]$, следовательно, присваивание $i := m$ допустимо.

Во втором случае снова воспользуемся монотонностью последовательности и заметим, что на отрезке $[m, j-1]$ ключа k нет, так что присваивание $j := m$ не нарушает инварианта.

На каждом шаге длина текущего отрезка $[i, j-1]$ сокращается в два раза (с возможным округлением в большую сторону в случае нечетности). Следовательно, максимальное количество итераций оценивается как $O(\log n)$.

Упражнение 3.3.1. Реализуйте алгоритм бинарного поиска.

Упражнение 3.3.2. Пусть задана последовательность $a_1, \ldots, a_n$, состоящая из нулей и единиц. Известно, что $a_1 \neq a_n$. Предложите алгоритм, который за время $O(\log n)$ находит такой индекс i, что $a_i \neq a_{i+1}$.

Упражнение 3.3.3. Выпуклый n-угольник P задан координатами своих вершин в порядке обхода против часовой стрелки. Разработайте алгоритм, который по заданной точке Q за время $O(\log n)$ выясняет, содержится Q внутри P или нет.

3.4. Деревья поиска

Упорядоченный массив фиксированного размера в сочетании с бинарным поиском представляет собой неплохой вариант для реализации типа «множество», но лишь при условии статичности набора ключей. Если же разрешить вставки и удаления, то ситуация осложнится. Проблему со вставками можно решить, если применить массив переменного (растущего) размера, однако при этом теряется свойство упорядоченности, так что потребуются дополнительные расходы времени на его восстановление. Удаление элементов создает пропуски в массиве, которые можно либо игнорировать (в таком случае страдает время поиска), либо пытаться ликвидировать (что приводит к необходимости перемещать ключи).

Ни один из этих способов не может дать гарантированной оценки $O(\log n)$ на время работы операций вставки, удаления и поиска. К счастью, существуют методы, которые способы такие гарантии дать. Наиболее популярные из них основаны на понятии *дерева поиска*, к изучению которого мы и переходим.

Формальное определение таково: рассмотрим дерево T с выделенным корнем. Каждая вершина v такого дерева может иметь не более двух сыновей, обозначаемых $\mathit{left}(v)$ и $\mathit{right}(v)$. В случае, если сын отсутствует, мы используем фиктивное значение NULL. Кроме того, для многих алгоритмов важно уметь по вершине v получать ссылку на ее родителя. Для этого в вершине хранится указатель $\mathit{parent}(v)$ (последний равен NULL в корне).

Пусть теперь каждая вершина дерева T снабжена ключом $\mathit{key}(v)$. Скажем, что T — *дерево поиска*, если для любой вершины v в дереве выполнены следующие свойства:

1) для любой вершины x в левом поддереве v справедливо неравенство $\mathit{key}(x) \leqslant \mathit{key}(v)$,

2) для любой вершины y в правом поддереве v справедливо неравенство $key(v) \leqslant key(y)$.

Данное определение можно переформулировать в терминах *обхода* дерева. В общем случае алгоритм обхода представляет собой процедуру, которая перечисляет все вершины дерева в определенном порядке (каждую вершину по одному разу). Среди всего разнообразия способов нас сейчас интересует ровно один — так называемый *inorder*-обход. Этот обход выражается рекурсивной процедурой INORDER-TRAVERSE, которая получает на вход вершину v, а затем выполняет следующие действия. В случае, если у v есть левый сын, для него происходит рекурсивный вызов процедуры INORDER-TRAVERSE. Затем выписывается вершина v. Наконец, симметрично, если у v присутствует правый сын, то производится рекурсивный вызов для этого правого сына.

Упражнение 3.4.1. Реализуйте вышеописанный алгоритм INORDER-TRAVERSE.

Доказательство следующего утверждения оставляется читателю в качестве упражнения.

Лемма 3.4.1. *Дерево T является деревом поиска, если и только если процедура inorder-обхода, примененная к данному дереву, выписывает ключи в неубывающем порядке.*

Итак, мы собираемся хранить ключи, составляющие текущее множество, в виде бинарного дерева поиска. Поскольку во множестве кратность вхождения любого элемента не превосходит единицы, то все ключи в дереве будут различными. В частности, неравенства в общем определении дерева поиска можно считать строгими. Покажем, как реализовать операции поиска, вставки и удаления, а также оценим время их работы.

Процедура поиска SEARCH проста. Предположим, что требуется найти в дереве ключ x либо убедиться в его отсутствии. Если x совпадает с ключом $key(r)$ корня r дерева, то поиск завершен успешно. Иначе, если $key(r) > x$, поскольку по свойству дерева поиска в правом поддереве r ключи строго больше x, их рассматривать нет смысла. Следовательно, поиск нужно продолжить в левом поддереве r. Если же $key(r) < x$, то, симметрично, нужно перейти в правое поддере-

во. Каждый такой переход сводит задачу к новой, при которой область поиска ограничена поддеревом в заданной вершине. Тем самым несложно написать процедуру (не обязательно рекурсивную), которая осуществляет описанный спуск вниз по дереву. Остановка происходит либо в тот момент, когда мы находим ключ x, либо когда оказывается, что нужно перейти в несуществующее NULL-поддерево. Данный метод полностью аналогичен бинарному поиску. Его сложность, однако, зависит от глубины дерева. Если последнюю обозначить через h, то время поиска будет составлять $O(h)$.

Опишем теперь операцию INSERT вставки нового ключа x. Вначале попробуем найти ключ x в дереве T, для чего будем производить спуск по дереву (аналогично процедуре поиска SEARCH). Если ключ найден, то вставка не требуется. Иначе поиск завершился неудачей; это означает, что на последнем шаге мы пытались пройти по NULL-ссылке из текущей вершины v. Создадим новую вершину w, куда поместим ключ x. Вершину w объявим сыном вершины v (левым или правым, в зависимости от того, какой ключ больше, $key(v)$ или $key(w)$). Проверка того факта, что новое дерево по-прежнему образует дерево поиска, оставляется читателю.

Наконец, займемся удалением REMOVE. Перед тем как удалять вершину v с ключом x, необходимо обнаружить эту вершину в дереве. Для этого используем процедуру SEARCH. (Впрочем, если удаляемая вершина задается не ключом, а ссылкой, то данный шаг избыточен.)

Если v — лист, то его удаление тривиально. Но с произвольной вершиной v возникают сложности, поскольку поддеревья, растущие из сыновей вершины v, необходимо сохранить. Рассмотрим два случая. Если у v есть ровно один сын w, то все просто: удалим v, а вершину w расположим на месте, которое занимала вершина v до удаления.

Осталось рассмотреть случай, когда у удаляемой вершины есть и левый, и правый сын. Рассмотрим вершину v', которая непосредственно предшествует v в inorder-обходе. Эта вершина является последней в inorder-обходе дерева, подвешенного в левом сыне вершины v. Для того чтобы найти v', необходимо пойти из v налево, а затем двигаться направо по ссылкам, пока это возможно. Отметим, что у v' уже нет правого сына. Скопируем ключ из v' в v. Теперь v' можно удалять, а это делать мы уже умеем.

Отметим, что описанный прием с копированием ключа имеет неприятную особенность: он разрушает (*инвалидирует*) ссылки на вершину v'. С точки зрения внешнего пользователя ситуация весьма странная: удалению должна быть подвергнута вершина v, а вместо этого была удалена некоторая другая вершина v'. Такое поведение часто нежелательно, поскольку пользователь вправе ожидать, что ссылки на все вершины дерева (кроме удаляемой вершины v) останутся правильными. Чтобы избежать этого, мы можем обменять в дереве вершины v и v' (перенаправив ссылки на детей и родителей естественным образом). После этого удалять нужно будет v, как и ожидалось.

При использовании описанных выше операций вставки и удаления нет гарантии, что дерево поиска будет иметь глубину, сильно меньшую n. Поэтому о том, чтобы поддерживать глубину h на уровне $O(\log n)$, придется заботиться дополнительно. Большое значение h возможно, если дерево «разбалансировано», т. е. размеры левых и правых поддеревьев в нем сильно отличаются. Для того чтобы избежать этой ситуации, деревья поиска подвергают операциям *балансировки*.

Основной тип балансировки, используемый для деревьев поиска, — это *вращения*. Вращение представляет собой локальное преобразование, применяемое к паре вершин, одна из которых является родителем другой. Например, пусть x обозначает родителя вершины y, причем y является левым сыном. Тогда на ребре (x, y) возможно *правое вращение* (см. стрелку слева направо на рис. 3.1). Вершина x становится левым сыном y, а сама вершина y занимает место x в дереве. Кроме того, правое поддерево с корнем в вершине y (обозначим его β) становится левым поддеревом вершины x. Другие поддеревья, а именно левое поддерево вершины y (обозначим его α) и правое поддерево вершины x (обозначение: γ), не меняют своих связей в дереве.

Правое вращение представляет собой обратимую операцию: от конфигурации, изображенной в правой части рис. 3.1, можно вернуться назад к конфигурации, изображенной слева. Это обратное преобразование называется *левым* вращением (относительно ребра (y, x)). Запомнить названия (*левое* или *правое*) весьма несложно, достаточно мысленно представить, в какую сторону поворачивается ребро дерева, относительного которого вращение производится.

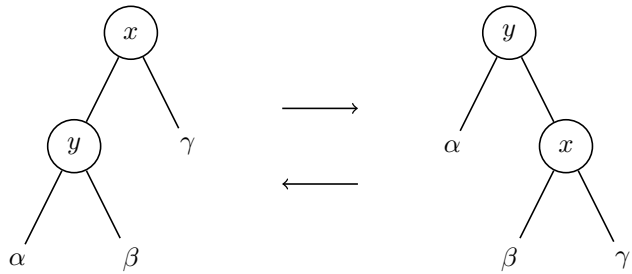

Рис. 3.1. Вращение деревьев поиска

Ключевое свойство вращений таково.

Лемма 3.4.2. *Вращения сохраняют inorder-порядок обхода дерева.*

Доказательство. Достаточно доказать, что сохраняется inorder-порядок в поддереве, к которому вращение применяется. Это действительно так. Преобразование, изображенное на рис. 3.1, отвечает следующему изменению в порядке обхода вершин:

$$(\alpha\ y\ \beta)\ x\ \gamma \leftrightarrows \alpha\ y\ (\beta\ x\ \gamma).$$

Здесь скобки использованы для обозначения порядка рекурсивных вызовов при обходе. Символы x, y, z обозначают сами вершины, а α, β, γ — последовательности, порождаемые inorder-обходом соответствующих поддеревьев. Видно, что если скобки опустить, то последовательности окажутся одинаковыми, что и требовалось доказать. □

Учитывая лемму 3.4.1, получаем, что вращения не выводят нас за класс деревьев поиска. Доказательство также показывает, что возможность применять операции вращения к деревьям поиска выражает, в некотором смысле, свойство «ассоциативности».

Обычно эти операции производятся после вставок и удалений, занимают время $O(\log n)$ и поддерживают глубину дерева также на уровне $O(\log n)$. Среди наиболее популярных техник балансировки упомянем *красно-черные* и *AVL-деревья*. К сожалению, их описание весьма громоздко и требует рассмотрения немалого количества частных случаев, поэтому мы не приводим его здесь, а просто отсылаем читателя к книге [1].

3.5. Сплей-деревья

Как уже упоминалось ранее, реализация сбалансированных деревьев поиска требует, помимо прочего, весьма кропотливого разбора возникающих при операциях вставки и (особенно!) удаления случаев нарушения баланса. Существует, впрочем, одна удивительная структура данных, балансировка которой сводится к разбору всего трех случаев (не считая симметричных). По сравнению с AVL- или красно-черными деревьями это совсем немного.

Структура, о которой пойдет речь, носит название *сплей-дерева* (*splay tree*) и была придумана Слитором и Таржаном (см., например, [4]). В отличие от «обычных» сбалансированных деревьев, сплей-деревья не поддерживают баланс постоянно. Вместо этого они остаются сбалансированными «в среднем». Точнее говоря, они сбалансированы настолько, что учетное время операций поиска, вставки и удаления составляет $O(\log n)$.

Краеугольным камнем, лежащим в основе конструкции сплей-деревьев, является операция, называемая SPLAY. Она получает на вход вершину x и поднимается вверх по дереву до корня, совершая при этом вращения. В конце ее работы x оказывается в корне дерева.

Лемма 3.5.1. *Истинная стоимость операции* SPLAY(x) *пропорциональна глубине вершины* x.

Приведенное выше утверждение не кажется особенно примечательным. Любое отдельное вращение требует константного времени, и если при этом еще и выбирать подходящий тип вращения за константное время в процессе подъема к корню, то лемма 3.5.1 будет справедлива.

Операция SPLAY обладает еще одним дополнительным свойством (напомним, что n обозначает число вершин в дереве).

Лемма 3.5.2. *Существует такая целочисленная функция потенциала, определенная на деревьях и принимающая неотрицательные значения, не большие* $O(n \log n)$, *что относительно нее любая операция* SPLAY *имеет учетную сложность* $O(\log n)$.

Примененные вместе, леммы 3.5.1 и 3.5.2 имеют нетривиальные следствия. Действительно, предположим, что мы начинаем работать

с некоторым деревом, выбираем в нем самую глубокую вершину и вызываем для нее операцию Splay. Затем опять выбираем самую глубокую вершину и снова вызываем Splay. Повторим эти действия большое количество раз (скажем, m при $m > n$). С одной стороны, каждый вызов требует времени, пропорционального глубине дерева. С другой стороны, все m вызовов совместно потребуют $O(m \log n + n \log n) = O(m \log n)$ единиц времени (первое слагаемое в сумме отвечает учетным стоимостям операций, а второе — изменению потенциала). Тем самым в среднем высота дерева равна $O(\log n)$. Дерево самобалансируется за счет вызовов операции Splay.

Описание способа работы операции Splay, а также доказательство леммы 3.5.2 будет дано ниже. Пока же будем пользоваться этой операцией как черным ящиком. Тогда не составляет труда понять, что можно реализовать операцию поиска ключа Search(k) за учетное время $O(\log n)$. Действительно, будем спускаться по дереву от корня, как в обычной реализации алгоритма Search. В итоге будет просмотрен некоторый путь от корня вниз по дереву и будет потрачено время, пропорциональное его длине. Пусть x обозначает последнюю вершину на данном пути. (Если поиск был успешным, то ключ в вершине x равен k. В противном случае при поиске мы оказались в вершине x, а затем пошли из нее по нулевой ссылке.) Вызовем в конце поиска Splay(x).

С одной стороны, добавление вызова Splay изменяет время поиска не более чем в константу раз (согласно лемме 3.5.1). С другой стороны, одни только операции Splay работают за $O(\log n)$ учетного времени. Тем самым поиск ключей теперь требует $O(\log n)$ учетного времени. Говоря неформально, выполнение в конце поиска операции Splay выравнивает дерево, поднимая часто используемые вершины ближе к корню.

Можно ли использовать такой прием для ускорения операции Insert? Кажется, что да: обе операции просматривают некоторый путь в дереве, а потому тратят время, пропорциональное его длине. Кроме того, вызывая в конце Splay, мы «заставляем» данные операции работать за $O(\log n)$ учетного времени.

В этом рассуждении, однако, есть один пробел. Согласно лемме 3.5.2 операция Splay действительно работает быстро, но только лишь в предположении, что дерево изменяется лишь ей самой. В отли-

чие от SEARCH, операция INSERT преобразует дерево, а значит, вообще говоря, изменяет его потенциал. Но к счастью, как мы далее увидим, изменение потенциала при добавлении новой вершины составляет $O(\log n)$, так что для INSERT вместе со SPLAY справедлива оценка $O(\log n)$ на учетное время работы.

Существует также способ скомбинировать REMOVE и SPLAY, но его обсуждение мы отложим до конца раздела. Сейчас же настало время изучить во всех подробностях, как работает процедура SPLAY, и доказать лемму 3.5.2.

Рассмотрим вызов процедуры SPLAY(x). Напомним, что побочным результатом его работы должен являться подъем вершины x в корень. В частности, если x — уже корень дерева, то никаких действий не требуется.

Пусть теперь вершина x не является корнем, но ее отец (обозначим его y) будет корнем. Без ограничения общности считаем, что x — левый сын вершины y, другой случай аналогичен. Процедура выполняет *zig-шаг*: производит правое вращение вокруг ребра (y, x), см. рис. 3.2.

Далее, пусть y не является корнем. Поднимемся еще на один уровень и обозначим отца вершины y через z. Возможны два подслучая. Либо, спускаясь от z до x, мы оба раза идем в одну сторону, либо в разные. Пусть x — левый сын вершины y, а y — левый сын вершины z. Тогда происходит *zigzig-шаг*, при котором производятся два правых вращения: вокруг ребра (z, y), а затем вокруг (y, x), см. рис. 3.3. Аналогично (с точностью до симметрии) поступаем, когда x — правый сын вершины y, а y — правый сын вершины z.

Наконец, предположим, что, двигаясь от z к x, мы производим спуски в разных направлениях. Скажем, x — левый сын вершины y, а y — правый сын вершины z. Здесь происходит *zigzag-шаг*, который также состоит из пары вращений, см. рис. 3.4. Конечно, как и ранее, для zigzag-шага есть симметричный вариант, однако он полностью аналогичен рассматриваемому.

На этом описание процедуры SPLAY завершается. Лемма 3.5.1 очевидно следует из построения. Докажем теперь лемму 3.5.2, для чего покажем, как выбрать функцию потенциала. Для каждой вершины x положим ее *вес* $w(x)$ равным количеству вершин в поддереве с корнем

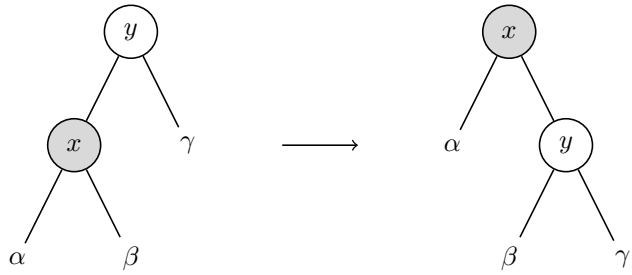

Рис. 3.2. Шаг типа zig.

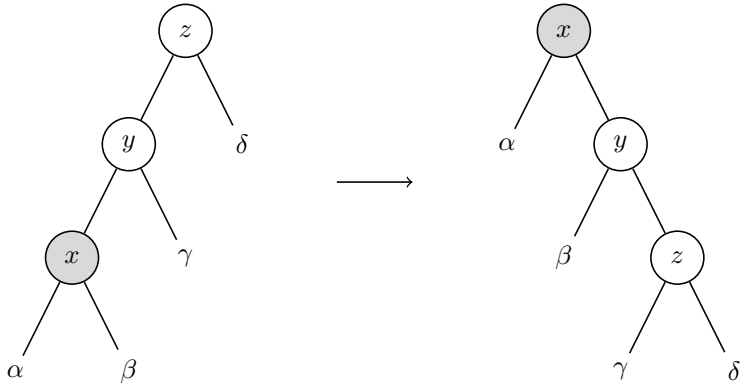

Рис. 3.3. Шаг типа zigzig

в x. Далее объявим потенциал $\Phi(x)$ вершины x равным $\lfloor \log_2 w(x) \rfloor$. Наконец, положим полный потенциал $\Phi(T)$ дерева T равным сумме потенциалов его вершин.

Неформально говоря, в каждой вершине x дерева будет лежать $\Phi(x)$ монеток, которыми мы будем расплачиваться за выполнение операций Splay. Очевидно, что потенциал любой вершины не превосходит $O(\log n)$, а значит, общий потенциал дерева может быть оценен как $O(n \log n)$.

Рассмотрим вызов процедуры Splay(x). Будем обозначать через Φ_0 потенциалы вершин, какими они были в момент начала вы-

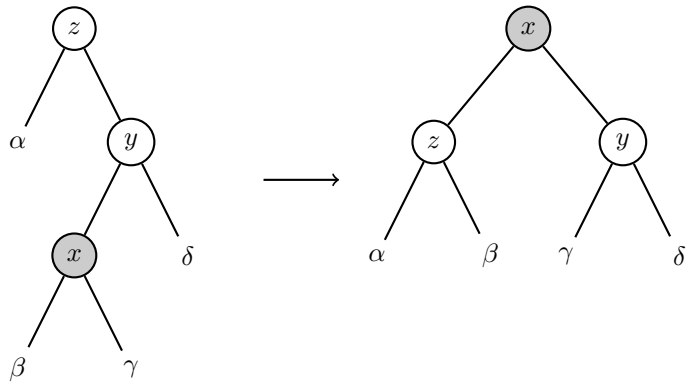

Рис. 3.4. Шаг типа zigzag

полнения данной операции. Как уже было отмечено, когда Splay завершит работу, x попадает в корень. Отметим также, что потенциал корня дерева не меняется в процессе работы процедуры Splay (поскольку зависит лишь от количества вершин в дереве). Тем самым вызов процедуры Splay(x) увеличивает потенциал вершины x на $\Phi_0(r) - \Phi_0(x)$, где r обозначает корень дерева в начальный момент.

Покажем, что учетная стоимость операции Splay(x) не превосходит $3(\Phi_0(r) - \Phi_0(x)) + 1$. Поскольку для потенциалов вершин справедлива логарифмическая оценка, учетная стоимость операции будет составлять $O(\log n)$.

Для доказательства заявленного факта разобьем работу Splay на шаги, которые, как уже отмечено выше, могут быть одного из трех типов: zig, zigzig и zigzag. Рассмотрим отдельный шаг, и пусть Φ обозначает значения потенциалов вершин до его начала, а Φ' — после окончания. Легко заметить, что потенциал текущей вершины x не уменьшается ($\Phi'(x) \geqslant \Phi(x)$), поскольку она продвигается по направлению к корню, и размер ее поддерева только увеличивается. Докажем следующее:

1) учетная стоимость zig-шага не превосходит $3(\Phi'(x) - \Phi(x)) + 1$;
2) учетная стоимость zigzig- или zigzag-шага не превосходит $3(\Phi'(x) - \Phi(x))$.

Суммируя эти оценки по всем шагам, получим требуемое выражение $3(\Phi_0(r) - \Phi_0(x)) + 1$. Начнем с анализа zig-шага. В оценке учетной стоимости необходимо учесть два фактора: время, затраченное на выполнение шага, и изменение потенциала дерева. Выбирая подходящим образом единицу измерения, можно считать, что шаг имеет истинную стоимость 1. Вычислим теперь изменение потенциала дерева. Несложно видеть, что потенциалы могут измениться лишь в вершинах x и y (см. рис. 3.2). Увеличение потенциала равно $\Delta\Phi := \Phi'(x) + \Phi'(y) - \Phi(x) - \Phi(y)$. Заметим, что $\Phi(y) = \Phi'(x)$, поскольку общее количество вершин в поддереве с корнем y не изменилось. Получаем $\Delta\Phi = \Phi'(y) - \Phi(x) \leqslant \Phi'(x) - \Phi(x) \leqslant 3(\Phi'(x) - \Phi(x))$ (в процессе преобразований мы воспользовались тем фактом, что потенциал монотонно не убывает при движении вверх по дереву). Итак, складывая истинную стоимость и изменение потенциала, получаем оценку $3(\Phi'(x) - \Phi(x)) + 1$.

Наличие множителя 3 в полученном выражении, конечно, вызывает некоторое удивление. Однако далее будет показано, что в случае zigzig- и zigzag-шагов он возникает уже по существу, так что в случае zig-шага он включен для единообразия.

Итак, рассмотрим теперь zigzig-шаг, см. рис. 3.3. Начнем с вычисления изменения потенциала дерева. В нашем случае оно равно

$$\Delta\Phi := \Phi'(x) + \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) - \Phi(z) = \\ = \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) \leqslant 2(\Phi'(x) - \Phi(x))$$

(здесь мы вновь воспользовались сохранением потенциала корня поддерева и его монотонностью). На этот раз в оценке появился множитель 2, но где же обещанная тройка? Для ответа на этот вопрос отметим, что нам также потребуется оплатить истинную стоимость шага, которая равна 1. В случае $\Phi'(x) > \Phi(x)$ из целочисленности потенциала следует, что $2(\Phi'(x) - \Phi(x)) + 1 \leqslant 3(\Phi'(x) - \Phi(x))$, так что зазор между 2 и 3 позволяет произвести оплату шага.

Что же делать, если $\Phi'(x) = \Phi(x)$ и требуемого зазора нет? Теперь покажем, что $\Delta\Phi < 0 = 2(\Phi'(x) - \Phi(x)) = 3(\Phi'(x) - \Phi(x))$, так что $\Delta\Phi + 1 \leqslant 3(\Phi'(x) - \Phi(x))$. Иными словами, при оценке $\Delta\Phi$ сверху по соображениям монотонности было использовано хотя бы одно строгое неравенство.

Предположим противное, т. е. допустим, что $\Phi'(y) = \Phi'(z) = \Phi'(x)$ и $\Phi(y) = \Phi(x)$. Учитывая, что $\Phi'(x) = \Phi(z)$ и $\Phi'(x) = \Phi(x)$, получаем, что все шесть потенциалов $\Phi(x)$, $\Phi(y)$, $\Phi(z)$, $\Phi'(x)$, $\Phi'(y)$, $\Phi'(z)$ равны между собой. Обозначим это общее значение через k. Пусть w и w', как обычно, обозначают веса вершин до и после шага. Будем использовать символы w и w' для обозначения весов не только вершин x, y, z, но и поддеревьев α, β, γ, δ. Конечно, для указанных поддеревьев w- и w'-веса совпадают. Тогда

$$\begin{aligned} w'(x) &= w'(\alpha) + w'(\beta) + w'(\gamma) + w'(\delta) + 3 = \\ &= (1 + w(\alpha) + w(\beta)) + (1 + w'(\gamma) + w'(\delta)) + 1 = \\ &= w(x) + w'(z') + 1 \geqslant 2^k + 2^k + 1 > 2^{k+1}, \end{aligned}$$

а значит, $\Phi'(x) \geqslant k + 1$ — противоречие.

Итак, между $3(\Phi'(x) - \Phi(x))$ и $\Delta\Phi$ всегда есть положительный зазор, которым мы и воспользуемся для оплаты истинной стоимости zigzig-шага. Анализ zigzag-шага проводится аналогично, и мы оставляем его читателю в качестве упражнения.

Упражнение 3.5.1. Проведите вышеуказанный анализ.

Лемма 3.5.2 доказана. Попробуем подвести итоги и выяснить, удалось ли нам получить «самобалансирующиеся» деревья поиска. Как уже было рассказано ранее, вызовы Splay в конце операции Search позволяют выполнять поиск за логарифмическое учетное время. Однако операции Insert и Remove пока рассмотрены не были.

Insert действует аналогично Search и просматривает некоторый путь в дереве для нахождения точки вставки. Вызовем Splay для конца x этого пути. Учитывая лемму 3.5.2, получим логарифмическую сложность, однако нужно еще учесть увеличение потенциала дерева за счет добавления нового листа. Анализ этого изменения несложен: при добавлении нового листа y к вершине x веса всех вершин на пути от корня до x увеличиваются на единицу. Соответствующие потенциалы также могут возрасти на единицу, но лишь в случае, если они становятся равны очередной степени двойки. Последовательность весов вершин на пути от корня до x строго убывающая, поэтому при увеличении всех ее элементов на единицу лишь $O(\log n)$ из них могут «переступить через границу» и вызывать увеличение потенциала. Таким образом,

к учетной сложности Insert необходимо дополнительно добавить $O(\log n)$, что не меняет асимптотики $O(\log n)$, установленной ранее.

Наконец, рассмотрим операцию Remove(x) для случая сплей-деревьев. Здесь удобно поступать не совсем так, как мы делали ранее для обычных деревьев поиска. Вначале выполним Splay(x), сделав x корнем. На это потребуется учетное время $O(\log n)$. Теперь удалить x несложно, однако вместо исходного дерева возникнет два поддерева: α — левый сын корня и β — правый сын. Разделение дерева на три части (левое поддерево, корень, правое поддерево) лишь уменьшает потенциалы вершин, поэтому его можно выполнить «бесплатно».

Заметим, что всякий ключ из α не превосходит всякого ключа из β. Осталось объединить α и β. Именно эту задачу и решает процедура Merge(α, β), которая работает следующим образом. Если α пусто, то ответом будет β. Иначе пройдемся по правому пути от корня в дереве α, найдя там максимальный элемент z. Выполним Splay(z), для того чтобы оплатить просмотр пути. Теперь элемент z стал корнем дерева α, причем правого сына у него нет (в силу максимальности). Подключим β в качестве правого сына вершины z. При этом вес вершины z возрастет на размер дерева β, а значит, также возможен и рост потенциала $\Phi(z)$. Однако этот рост не превышает $O(\log n)$, поэтому учетная сложность процедуры Merge (а вместе с ней и Remove) остается логарифмической.

Отметим, что в процессе реализации процедуры Remove мы заодно научились по заданной вершине x разделять дерево на две части, содержащие соответственно вершины, предшествующие x в inorder-обходе, и вершины, следующие за x. Такая операция обычно обозначается Split. Прием реализации Remove через последовательность Split и Merge является чрезвычайно полезным, еще одно его приложение будет изложено в гл. 4.

Упражнение 3.5.2. Реализуйте сплей-деревья с операциями Search, Insert и Remove.

3.6. Задачи

Задача 1. Дано дерево, в листах которого записаны целые числа. Всего в дереве n вершин, $1 \leqslant n \leqslant 10\,000$. Для каждой внутренней

вершины дерева найдите минимальную по модулю из всех попарных разностей между числами, записанными в листах этого поддерева.

В первой строке входа записано число n. Корень дерева — вершина номер 1. Во второй строке входа описаны дети вершины 1: сначала количество детей, затем список вершин — номеров детей. В третьей строке аналогичным образом описаны дети второй вершины, затем третьей и т. д. В последней строке находятся числа, записанные в листах дерева, в порядке номеров листов. Все номера вершин — числа от 1 до n. Числа, записанные в листах, — от $-1\,000\,000\,000$ до $1\,000\,000\,000$.

Для каждой вершины выведите искомую разность в порядке возрастания номеров вершин. Для листов выводите 0.

Пример входа	Пример выхода
3 2 2 3 0 0 3 5	2 0 0
7 2 2 3 2 4 5 2 6 7 0 0 0 0 1 5 6 8	1 4 2 0 0 0 0

Решение. Для каждой вершины дерева начиная с листов и вверх по дереву найдем множество всех чисел, записанных в листах этого поддерева, и будем хранить все эти числа в сбалансированном бинарном дереве. Кроме того, найдем одновременно с этим ответ к задаче для каждой внутренней вершины. Для листов это тривиально: в каждом таком множестве только один элемент, а никаких попарных разностей в поддереве листа нет, так что можно считать, что минимум попарных разностей — плюс бесконечность. При выводе

эту фиктивную плюс бесконечность нужно будет заменить на нуль по условию задачи.

На очередном шаге рассматривается вершина p с детьми $c_1, c_2, \ldots, c_k$, для каждого из которых уже посчитан ответ к задаче, а также множество A_i, в котором хранятся все числа из поддерева c_i. Найдем минимум из ответов для c_i и временно присвоим величине k_p — ответу для p — это значение. Будем строить множество $A(p)$, начиная с A_1 и постепенно добавляя в него элементы из подмножеств $A_2, A_3, \ldots, A_k$. Добавляя элемент в множество, сначала найдем двух его соседей перед добавлением. При этом сравним модули разностей между этим элементом и соседями с текущим минимумом k_p и при необходимости обновим k_p. В итоге получим правильное значение k_p и множество $A(p)$, в котором будут содержаться все числа из поддерева p.

Это решение пока что работает долго, за время $O(n^2 \log n)$. Действительно, в худшем случае мы будем сначала добавлять множество из одного элемента к множеству из одного элемента, потом множество из 2 элементов к множеству из одного элемента, потом множество из 3 элементов к множеству из одного элемента и т. д., и в конце множество из $n-2$ элементов к множеству из n элементов. Это произойдет, если дерево бинарное, у корня два сына, у его левого сына нет сыновей, а у правого — опять два сына, у левого из них нет сыновей, у правого — два сына и т. д. В таком случае всего произойдет $1+2+\ldots+n=O(n^2)$ перекладываний из одного множества в другое, а на каждое перекладывание уйдет $O(\log n)$ действий.

Однако можно это решение легко улучшить так, чтобы оно работало уже за время $O(n \log^2 n)$. Действительно, каждый раз, когда нужно добавить множество A_k к множеству A_1, посмотрим на их размеры. Если размер множества A_k меньше размера множества A_1, то добавляем A_k к A_1 по одному элементу, как и собирались. Если же A_k больше A_1, то, наоборот, добавляем A_1 к A_k по одному элементу, т. е. всегда добавляем меньшее множество к большему. В этом случае каждый элемент будет за все время переложен не более чем $O(\log n)$ раз: действительно, изначально он находится в множестве из одного элемента (в листе), затем, если его перекладывают в первый раз, он оказывается в множестве из по крайней мере двух элементов,

после следующего перекладывания — в множестве по крайней мере из четырех элементов и т. д., а после k перекладываний — в множестве по крайней мере из 2^k элементов. При этом ни в каком множестве не может быть более n элементов, значит, $k = O(\log n)$. Таким образом, получаем сложность алгоритма $O(n \log^2 n)$, так как каждый элемент перекладывается не более $\log n$ раз, всего есть n элементов и на одно перекладывание уходит $O(\log n)$ действий. Памяти уходит $O(n)$, так как на каждом шаге память требуется только для объединения всех текущих множеств, а в них в сумме не более n элементов.

Задача 2. На плоскости дано n отрезков. Определите, есть ли среди них два пересекающихся.

В первой строке входа записано число n, а в каждой из следующих n строк — по четыре целых числа — координаты концов отрезка x_1, y_1, x_2, y_2,

$$1 \leqslant n \leqslant 1\,000\,000, \quad -1\,000\,000\,000 \leqslant x_1, y_1, x_2, y_2 \leqslant 1\,000\,000\,000.$$

Выведите `NO`, если никакие два отрезка не пересекаются. Иначе в первой строке выведите `YES`, а во второй — номера любых двух пересекающихся отрезков. Отрезки нумеруются с единицы.

Пример входа	Пример выхода
3 0 0 2 2 0 1 0 2 0 2 2 0	YES 1 3
3 0 0 2 2 0 1 0 2 1 0 2 0	NO

Решение. Тривиальное решение, перебирающее все пары отрезков и проверяющее их на пересечение, работает за время $O(n^2)$. Мы улучшим его до $O(n \log n)$ с помощью метода сканирующей прямой.

Изначально примем следующие ограничения. Во-первых, пусть среди данных отрезков нет вертикальных. Во-вторых, пусть никакие

три отрезка не пересекаются в одной точке. Далее отдельно покажем, что эти ограничения можно снять.

Будем двигать вертикальную сканирующую прямую слева направо из минус бесконечности в плюс бесконечность. В процессе движения иногда будут происходить «события» двух видов:

1) начался отрезок — встретился левый конец некоторого отрезка;
2) закончился отрезок — встретился правый конец некоторого отрезка.

Так как вертикальных отрезков нет, левый и правый концы определены однозначно у всех отрезков.

Кроме того, в каждый момент времени будем поддерживать множество всех отрезков, которые сейчас пересекают сканирующую прямую. Это множество полностью упорядочено снизу вверх по ординате точки пересечения отрезка с прямой. Так как вертикальных отрезков нет, каждый отрезок пересекает прямую ровно в одной точке.

При этом легко видеть, что при движении прямой слева направо порядок всех отрезков не меняется, до тех пор пока какие-то два из них не пересекутся и прямая не пройдет через эту точку пересечения. Этот факт мы и будем использовать в решении задачи.

Итак, опишем процедуру определения того, есть ли пересекающиеся отрезки, и нахождения какой-либо пары пересекающихся отрезков в случае, когда она есть.

Изначально множество отрезков S, пересекающих прямую l, пусто. Все события упорядочим по возрастанию абсциссы, а при равенстве абсцисс сначала обрабатываем левые концы отрезков, потом правые. Если равны абсциссы нескольких левых или нескольких правых концов отрезков, упорядочим их снизу вверх по ординате.

Когда происходит событие «начало отрезка s», нужно добавить отрезок в множество S, затем найти там его соседей сверху и снизу a и b и попробовать пересечь s с ними (если какого-либо из соседей нет, то его не рассматриваем). Если с одним из них s пересекается, то найдены пара пересекающихся отрезков и их точка пересечения и алгоритм завершает работу.

Когда происходит событие «конец отрезка s», нужно найти его соседей сверху и снизу a и b в текущем множестве (если какого-либо из соседей нет, не рассматриваем его), а затем попробовать пересечь a

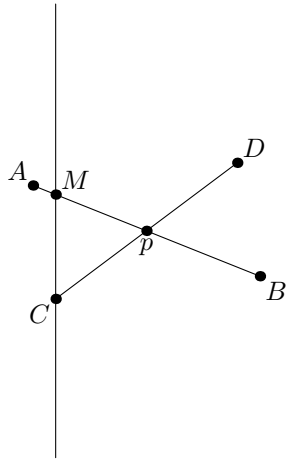

Рис. 3.5. Пересечение двух отрезков

с b, a с s, b с s. Если среди них есть пара пересекающихся, то найдены пара пересекающихся отрезков и их точка пересечения и алгоритм завершается. Если пересекающихся отрезков нет, то удаляем s из множества.

Очевидно, что если никакие два отрезка не пересекаются, то указанный алгоритм не найдет пересечения и остановится. Осталось доказать, что если есть пара пересекающихся отрезков, то алгоритм найдет пересечение. Пусть p — самая левая из всех точек пересечения отрезков. Утверждается, что, во-первых, алгоритм остановится, прежде чем сканирующая прямая пройдет за точку p, а во-вторых, он при этом найдет некоторую точку пересечения. В частности, это означает, что порядок отрезков в множестве S ни разу не поменяется за время выполнения алгоритма.

Действительно, пусть в точке p пересекаются два отрезка AB и CD (мы предположили, что в каждой точке пересекаются не более чем два отрезка), A и C — их левые концы (см. рис. 3.5). Пусть отрезок CD был добавлен в множество S позже (т. е., в частности, абсцисса точки C не меньше абсциссы точки A). Если в этот момент соседом отрезка CD был отрезок AB, то алгоритм нашел пересечение,

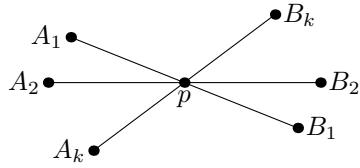

Рис. 3.6. Пересечение k отрезков

как и должно было быть. Если же нет, то обозначим через M точку пересечения вертикальной прямой, проходящей через точку C, с отрезком AB — тогда существует отрезок, пересекающий треугольник CMp. Рассмотрим множество Z всех таких отрезков. Ни один из них не может пересекать ни сторону Cp, ни сторону Mp треугольника CMp, так как тогда точка пересечения будет либо находиться строго левее p, либо совпадать с p, но этого также не может быть, так как в точке p уже и так пересекаются отрезки AB и CD. Значит, ни один из этих отрезков не пересекает ломаную CpM. А это означает, что если обозначить через x самый правый из всех концов отрезков в Z, то x находится левее p. Рассмотрим момент, когда происходит событие x. В этот момент последний отрезок, разделяющий AB и CD в множестве S, удаляется из множества S, и мы проверяем AB и CD на пересечение и находим его, что и требовалось.

Что касается вертикальных отрезков, то с ними можно проводить те же самые рассуждения, если левым концом вертикального отрезка считать его нижний конец, правым — верхний, а точкой пересечения со сканирующей прямой — нижний конец. Упорядочение отрезков остается корректным, за исключением случая, когда некоторый отрезок пересекает вертикальный отрезок, но это пересечение будет определено еще до добавления вертикального отрезка в множество S.

Приведенное выше рассуждение можно видоизменить для случая многих отрезков, пересекающихся в одной точке: вместо пары отрезков AB и CD рассмотрим сразу множество всех отрезков $A_1B_1, A_2B_2, \ldots, A_kB_k$, пересекающихся в точке p (см. рис. 3.6). Пусть $A_1, A_2, \ldots, A_k$ к тому же упорядочены по углу относительно p. Ни один из остальных отрезков не может пересекать ни один из отрезков $A_1p, A_2p, \ldots, A_kp$. Поэтому либо в момент добавления одного

из отрезков A_iB_i будет найдено его пересечение с соседом A_jB_j, либо в момент удаления самого правого конца среди концов всех отрезков, пересекающихся с треугольником A_1pA_k, какие-то два из отрезков A_iB_i станут соседями и будет обнаружено их пересечение. Если разрешить вертикальные отрезки, то один из отрезков A_iB_i может быть вертикальным, но это ничего не меняет. А вертикальные отрезки, расположенные левее p, не играют роли.

Таким образом, алгоритм работает правильно. Чтобы оценить его сложность, необходимо сначала определиться со структурой данных для множества S. Воспользуемся сбалансированным бинарным деревом поиска. Свойство неизменности порядка сегментов в S при перемещении сканирующей прямой в ходе алгоритма гарантирует, что дерево всегда будет оставаться корректным. А функция сравнения при этом может даже формально зависеть от времени, реально вычисляя ординаты точек пересечения отрезков с *текущей* сканирующей прямой и сравнивая их. Можно поступить и по-другому: для непересекающихся отрезков можно определить, какой из них выше, используя только векторные произведения (и, таким образом, не выходя за пределы целых чисел при вычислениях), и факт пересечения отрезков можно также определить с помощью операции векторного произведения, не выходя за пределы целых чисел. В случае сравнения двух пересекающихся отрезков в процессе вставки в дерево можно сразу же завершать работу алгоритма и выводить точку пересечения. Чтобы ее найти, уже придется проводить вычисления в вещественных числах.

Хранить нам нужно только сами отрезки, отсортированные по абсциссе концы этих отрезков и множество S. Итого, алгоритм работает за время $O(n \log n)$ и требует $O(n)$ памяти.

Глава 4. Кучи

4.1. Приоритетные очереди

В данной главе мы изучим ряд структур данных, реализующих интерфейс *приоритетные очереди*. Использование термина «очередь» здесь является не вполне удачным, но оно исторически сложилось. Очередь с приоритетом хранит набор из n элементов, снабженных числовыми *приоритетами*.

Набор операций, поддерживаемый приоритетной очередью, зависит от ее конкретной модификации. Рассмотрим простейший вариант:

- CREATE — создать пустую очередь;
- INSERT(p) — добавить в очередь новый элемент с приоритетом p и вернуть итератор на него;
- REMOVE(it) — удалить из очереди элемент, на который указывает итератор it;
- GET-MIN — получить элемент с минимальным значением приоритета среди содержащихся в очереди;
- CHANGE-PRIORITY(it, p) — заменить на p приоритет элемента, на который указывает итератор it;
- EXTRACT-MIN — удалить из очереди элемент с минимальным приоритетом.

Несложно заметить, что все эти операции могут быть реализованы с помощью сбалансированных деревьев поиска (например, красно-черных деревьев) за время $O(\log n)$. Такой подход, однако, неэффективен с точки зрения затрат памяти и времени. Кроме того, чуть позднее мы познакомимся со «сливаемыми очередями с приоритетами», реализовать которые с помощью деревьев поиска не представляется возможным. Очередь с приоритетом может быть реализована с помощью связного списка, хранящего элементы вместе с соответствующими им приоритетами. При этом время вставки и удаления

заданного элемента составит $O(1)$, но на поиск минимума (и его удаление) придется потратить время $O(n)$.

Подавляющее большинство эффективных реализаций очередей с приоритетом основаны на применении деревьев, в вершинах которых хранятся элементы очереди. Эти деревья (а иногда даже наборы из таких деревьев) и называют *кучами*. Аналогично деревьям поиска, кучи накладывают специальные условия на размещение приоритетов. Часто термины *очередь с приоритетом* и *куча* употребляются как синонимы, но, строго говоря, первый обозначает контракт, а второй — его реализацию.

4.2. Бинарные кучи

Итак, рассмотрим дерево T, в котором вершинам приписаны элементы с приоритетами. Будем обозначать через $pri(v)$ приоритет, записанный в вершине v, а через $parent(v)$ — родителя вершины v. Скажем, что дерево T удовлетворяет *свойству кучи*, если выполнено следующее условие:

$$pri(parent(v)) \leqslant pri(v) \text{ для всех вершин } v, \text{ кроме корня.} \qquad (4.1)$$

Будем также говорить, что написанное выше неравенство задает *свойство кучи* на ребре, соединяющем v с родителем. Эквивалентно, дерево со свойством кучи определяется следующим условием:

$$pri(u) \leqslant pri(v) \text{ для любых вершины } u \text{ и ее сына } v. \qquad (4.2)$$

В куче ключи не возрастают при движении от вершины вверх к корню. Следовательно, справедлив такой результат.

Утверждение 4.2.1. *Минимум приоритетов элементов, лежащих в куче, достигается в ее корне.*

Это свойство объясняет, отчего кучи полезны для реализации очередей с приоритетами: минимальный приоритет находится в фиксированном положении, так что его не нужно специально искать. Напомним, что именно нахождение минимального элемента было неэффективной операцией для реализации, использующей связные списки.

Итак, в куче операция Get-Min выполнима за время $O(1)$.

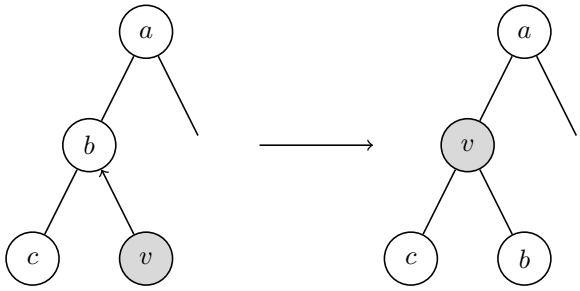

Рис. 4.1. Операция Sift-Up

Конечно, за полученное преимущество придется платить: понадобится так организовать вставки и удаления, чтобы они не только работали быстро, но также сохраняли бы свойство кучи.

Для начала ограничимся специальным видом деревьев. А именно, будем считать, что дерево T бинарное, т. е. у любой вершины есть не более двух сыновей (называемых *левым* и *правым*). Для вершины v будем обозначать через $left(v)$ (соответственно $right(v)$) ее левого (соответственно правого) сына. В случае, если соответствующий сын отсутствует, воспользуемся специальным значением NULL.

Операция Create создает пустое дерево и потому тривиальна.

Опишем, как можно реализовать операцию Insert для такого дерева. Получив на входе новый элемент, создадим в дереве новый лист v и положим этот элемент в v. Конечно, эта операция может нарушить свойство кучи. Впрочем, единственное нарушение может состоять в том, что ключ в родителе вершины v больше, чем ключ в вершине v. Чтобы восстановить свойство кучи, применим операцию, называемую *просеиванием вверх* и обозначаемую Sift-Up.

Данная процедура получает на вход вершину v. При этом гарантируется, что неравенство $pri(parent(x)) \leqslant pri(x)$ выполнено для всех вершин x, кроме, возможно, v. После завершения работы данной процедуры свойство кучи будет выполнено для всего дерева. Ее работа организована следующим образом. Если $pri(parent(v)) \leqslant pri(v)$, то никакая коррекция не требуется, Sift-Up завершает работу. Иначе пусть b обозначает родителя вершины v, a — родителя вершины b, c — второго сына вершины b, не равного v (вершины a и c могут

отсутствовать). Произведем обмен вершин b и v в дереве, тем самым «просеивая» вершину v вверх. Докажем, что после этого свойство кучи снова выполнено везде, кроме, возможно, ребра (a, v). Для ребра (v, b) оно справедливо, поскольку выполнен обмен. Проверим данное свойство для (v, c): $pri(c) \geqslant pri(v)$ поскольку $pri(b) \leqslant pri(c)$ (по условию) и $pri(b) > pri(v)$. Итак, текущая вершина v, в которой возможно нарушение свойства кучи, переместилась на уровень выше. Время работы Sift-Up, очевидно, оценивается как $O(h)$, где h — высота кучи.

Далее опишем, как работает процедура Extract-Min. Как уже было отмечено, минимальный приоритет среди элементов кучи находится в ее корне r. Удаление корня будем выполнять в два приема. Во-первых, найдем какой-нибудь лист v в куче (если его нет, то куча состоит только из корня, так что после удаления она становится пустой). Далее обменяем местами вершины v и r. Теперь корнем кучи служит v, а r становится листом. Удалим r из дерева. Осталось разобраться с сохранением свойства кучи. Легко видеть, что данное свойство могло быть нарушено для ребер, соединяющих v с сыновьями. Для того чтобы исправить ситуацию, вызовем процедуру Sift-Down, которая опустит вершину v вглубь дерева, тем самым избавившись от нарушений.

Более точно, данная процедура получает на вход такую вершину v, что свойство кучи выполнено всюду, кроме ребер, идущих из этой вершины в ее сыновей. Кроме того, если у v есть родитель a, а также сын x, то должно выполняться неравенство $pri(a) \leqslant pri(x)$. (Заметим, что для вызова процедуры Sift-Down, о котором идет речь в процедуре Extract-Min, v является корнем, так что вершины a нет. Однако, во-первых, мы будем далее использовать процедуру Sift-Down и в более общей ситуации, а во-вторых, это дополнительное условие составляет инвариант процедуры Sift-Down и необходимо для доказательства корректности ее работы.)

Обозначим сыновей вершины v через b и c (одна из этих вершин может отсутствовать). Не ограничивая общности, будем считать, что $pri(b) \leqslant pri(c)$. Поскольку условие кучи нарушается на каком-то ребре, идущем из вершины v в ее сына, оно заведомо нарушается на ребре

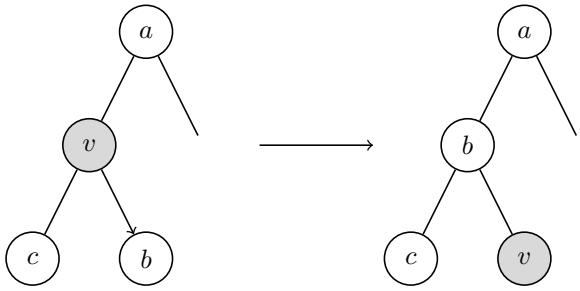

Рис. 4.2. Операция Sift-Down

(v, b). Обменяем вершины b и v в дереве. Проверим, что свойство кучи выполнено на ребрах (a, b), (b, v) и (b, c). Для ребра (a, b) используем дополнительное предположение о связи приоритета родителя вершины v с приоритетом сыновей. Для ребра (b, v) условие выполнено, поскольку был произведен обмен. Наконец, для ребра (b, c) условие выполнено, поскольку в качестве b был выбран сын вершины v с минимальным приоритетом. Осталось показать, что теперь приоритет родителя v, т. е. b, не больше приоритета любого сына вершины v. Это, однако, следует из того, что до обмена свойство кучи было выполнено на всех ребрах, идущих из b в ее сыновей.

Итак, нам снова удалось переместить «проблемную» вершину в дереве, но на этот раз вниз. Итерируя этот процесс, можно восстановить свойство кучи во всем дереве. Сложность процедуры составляет $O(h)$, где h — высота поддерева с корнем в вершине v, для которой был изначально произведен запуск.

После этих подготовительных действий процедура Change-Priority совершенно тривиальна. Вначале обновляем приоритет вершины v. Затем если указанный приоритет увеличился, то производим для v операцию Sift-Down, а если уменьшился, то Sift-Up.

Наконец покажем, как удалить произвольную вершину v из кучи, т. е. выполнить операцию Remove. Для этого воспользуемся простым, но полезным приемом: изменим (с помощью Change-Priority) приоритет вершины v на $-\infty$, после чего извлечем v, вызвав процедуру Extract-Min.

В представленном выше описании остался небольшой пробел: не указано, как именно при выполнении операции Insert выбирается положение добавляемого листа, а также какой из листьев используется при обмене с удаляемой вершиной в процедуре Extract-Min. Учитывая, что время работы операций зависит от высоты дерева, резонно поддерживать его сбалансированным. При этом ни Sift-Up, ни Sift-Down не меняют структуры дерева, так что у нас нет необходимости в балансировке, которая возникала в деревьях поиска. Более того, существует простой способ реализации бинарной кучи, при котором вершины дерева вовсе не хранят никаких указателей.

А именно, кучу из n вершин будем представлять массивом pri длины n, элементы которого нумеруются от 1 до n. Элемент 1 будет хранить корень дерева. Для элемента i его сыновьями будут вершины $2i$ и $2i+1$ (в случае, если эти номера попадают в отрезок $[1, n]$, иначе соответствующие сыновья у i отсутствуют). Для вершины $i > 1$ ее родитель имеет номер $\lfloor i/2 \rfloor$. Если обозначить через $p[i]$ приоритет вершины, хранящейся в i-м элементе массива, то условие (4.2) может быть переписано в следующем виде:

$$pri[i] \leqslant pri[2i] \quad \text{и} \quad pri[i] \leqslant pri[2i+1] \quad \text{для всех } 1 \leqslant i \leqslant n. \qquad (4.3)$$

В случае, если при вычислении правой части неравенства в формуле (4.3) происходит выход индекса массива за границы $[1, n]$, данное неравенство считается выполненным. Получающееся при таком построении дерево будем называть *почти полным бинарным*, см. рис. 4.3. Легко видеть, что его высота составляет $O(\log n)$.

Поскольку значение n может изменяться в процессе работы, массив придется сделать переменного размера, для чего подходит метод, изложенный в разделе 1.1. При вставке нового элемента в кучу добавляем его в конец массива, увеличивая n. При этом существующая структура дерева сохраняется, но у него появляется новый лист. При удалении минимума обмениваем первый элемент массива (корень) с последним элементом (листом), после чего уменьшаем длину на 1.

Итак, бинарная куча позволяет реализовать интерфейс очереди с приоритетами и достигает сложности $O(\log n)$ для каждой из операций. Более того, операция Get-Min оказывается выполнимой за время $O(1)$.

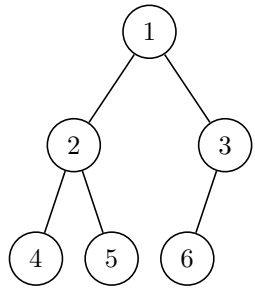

Рис. 4.3. Почти полное бинарное дерево с 6 вершинами

Упражнение 4.2.1. Реализуйте процедуры INSERT, REMOVE, GET-MIN, EXTRACT-MIN и CHANGE-PRIORITY с помощью бинарной кучи.

4.3. Сортировка кучей

Очередь с приоритетом позволяет дать еще одно решение классической задачи о сортировке, которая изучалась в гл. 2. А именно, рассмотрим числовую последовательность $A = (a_1, \ldots, a_n)$. Создадим очередь с приоритетом Q. Изначально она пуста. Затем добавим последовательно все элементы последовательности A в Q. Наконец произведем n операций извлечения минимума из Q. Получим элементы последовательности A в порядке неубывания. Если вспомнить, что бинарная куча позволяет реализовать вставку и извлечение минимума за время $O(\log n)$, то получится алгоритм со сложностью $O(n \log n)$. Его называют *алгоритмом сортировки кучей*, и мы будем обозначать его HEAP-SORT.

В отличие от стандартного метода MERGE-SORT, сортировка кучей использует лишь $O(1)$ дополнительной памяти, так что массив будет отсортирован *на месте* (*in-place*). HEAP-SORT не является устойчивым методом, т. е. может переставлять элементы с равными ключами.

При практической реализации метод HEAP-SORT обычно уступает методу QUICK-SORT, но зато имеет гарантированное время работы $O(n \log n)$.

Как видно из представленного выше описания, HEAP-SORT состоит из двух фаз: фазы построения кучи из набора A и фазы извлечения. Обе фазы отнимают $O(n \log n)$ единиц времени. Полезно, однако, знать, что первая фаза может быть реализована за линейное время.

А именно, опишем процедуру HEAPIFY, которая получает на вход последовательность A и переставляет в ней элементы так, что выполняется свойство (4.3). Работа алгоритма состоит из итераций. При этом поддерживается такое значение параметра t, что неравенства (4.3) выполнены для всех $i = t, \ldots, n$. Изначально $t := \lfloor n/2 \rfloor + 1$, так что требуемые условия выполнены тривиально. На очередной итерации значение t уменьшается на единицу, после чего последовательность A перестраивается с помощью обменов. Когда параметр t оказывается равным единице, работа алгоритма завершается.

Чтобы объяснить, как алгоритм восстанавливает инвариант, заметим, что отрезок $[t, n]$ выделяет в почти полном бинарном дереве некоторый набор непересекающихся поддеревьев. Обозначим соответствующий лес через F. После уменьшения t условия (4.3) выполнены для всех $i = t+1, \ldots, n$. Вершина t является корнем некоторого дерева T в лесе F. Тем самым нам лишь необходимо восстановить свойство кучи в T, причем известно, что нарушаться данное свойство может лишь на ребрах, идущих из корня к сыновьям. Для этого поступаем так же, как и ранее в процедуре EXTRACT-MIN: вызываем операцию SIFT-DOWN для корня t.

На этом описание алгоритма HEAPIFY завершено. Сразу видно, что время его работы можно оценить как $O(n \log n)$. Однако эту оценку можно уточнить. Напомним, что время работы алгоритма SIFT-DOWN оценивается как $O(h)$, где h — высота поддерева, для которого данная процедура запущена. Будем, не ограничивая общности, считать, что $n = 2^k - 1$, так что вся куча представляет собой полное бинарное дерево высоты k (где высота выражается количеством вершин на длиннейшем пути от корня до листа). Тогда всего в куче есть 2^{k-j} поддеревьев высоты j. Суммируя по всем поддеревьям получаем:

$$\sum_{j=1}^{k} 2^{k-j} j = 2^k \sum_{j=1}^{k} \frac{j}{2^j} = O(2^k) = O(n). \tag{4.4}$$

Итак, время работы HEAPIFY линейно.

Упражнение 4.3.1. Докажите первое из равенств в строке (4.4).

Упражнение 4.3.2. Реализуйте процедуру HEAPIFY и алгоритм HEAP-SORT на его основе.

Алгоритм HEAP-SORT эффективен также в ситуации, когда требуется *частичная сортировка* (*partial sort*) массива. А именно, по заданному k ($1 \leqslant k \leqslant n$) и массиву A длины n нужно отобрать среди ключей k минимальных и их упорядочить. В этом случае возможно за линейное время преобразовать A в двоичную кучу, а затем выполнить k операций EXTRACT-MIN. Общая сложность алгоритма оказывается равной $O(n + k \log n)$, так что он может быть полезен в случае, когда $k \ll n$.

4.4. *k*-ичные кучи

Метод из предыдущего раздела может быть обобщен следующим образом: вместо того чтобы рассматривать почти полные двоичные (бинарные) деревья, можно рассматривать почти полные k-ичные деревья (где $k \geqslant 2$). Вершины такого дерева нумеруются числами от 1 до n, вершина 1 служит корнем. Сыновьями вершины i будут вершины с номерами $ki, ki + 1, \ldots, ki + k - 1$ (как и ранее, если индекс не попадает в отрезок $[1, n]$, то соответствующий сын отсутствует). Родителем вершины i становится вершина $\lfloor i/k \rfloor$. Процедура SIFT-UP дословно переносится на случай k-ичной кучи. В процедуре SIFT-DOWN при просеивании вершины v вниз нужно искать минимум приоритетов среди всех сыновей вершины v (а их может быть не более k).

Посмотрим, как отражается изменение параметра k на оценке сложности алгоритмов. Увеличение степени ветвления приводит к тому, что высота кучи уменьшается. Оценка сложности процедуры SIFT-UP теперь составляет $O(\log_k n)$, т. е. время работы, безусловно, понижается с увеличением k. В процедуре SIFT-DOWN, однако, нам предстоит на каждом шаге выбирать минимум среди приоритетов сыновей, поэтому оценка сложности составляет $O(k \log_k n)$. Здесь с ростом k время работы, наоборот, увеличивается.

Положительный эффект от применения k-ичных куч может быть достигнут, если известны оценки на количество операций, осуществляемых с кучей. Приведем типичный пример: известный в алгоритмической теории графов алгоритм Дейкстры (см. [1]) ищет кратчайшие

пути в направленном графе от выделенной вершины до остальных при условии, что длины дуг неотрицательны. Данный алгоритм в процессе своей работы использует очередь с приоритетом и выполняет $O(n)$ операций Extract-Min и $O(m)$ операций Change-Priority, где n — число вершин, а m — число дуг в графе. При этом каждый вызов Change-Priority может лишь уменьшить существующий приоритет. Таким образом, выполняется $O(n)$ операций Sift-Down и $O(m)$ операций Sift-Up.

Если использовать бинарные кучи в алгоритме Дейкстры, то оценка его сложности составит $O(m \log_2 n)$. Попробуем ее улучшить. Для этого фиксируем значение k и применим k-ичные кучи. Тогда оценка сложности заменится на $O(nk \log_k n + m \log_k n)$. Выбирая $k := \lfloor m/n \rfloor$ (считаем, что $m \geqslant 2n$), получаем сложность $O(m \log_{\frac{m}{n}} n)$, что лучше, чем у бинарных куч.

4.5. Сливаемые приоритетные очереди

Для некоторых приложений список операций, приведенный в разделе 4.1, оказывается недостаточным. А именно, требуется еще одна операция *слияния*:

- Meld — объединить множество элементов в двух очередях, организовав из них новую очередь.

При этом очереди, переданные в процедуру Meld в качестве аргументов, считаются разрушенными после ее вызова.

Сливаемые очереди с приоритетом применяются, например, при реализации алгоритмов нахождения кратчайшего дерева путей или построения минимальных остовных деревьев [1, 4]. Отметим, что реализовать этот интерфейс с помощью сбалансированных деревьев поиска уже не получится. Некоторые деревья поиска поддерживают операцию объединения Merge (таковы, например, *дучи*, описание которых последует в разделе 4.9). Но эта операция требует, чтобы подаваемые ей на вход деревья обладали дополнительным свойством: всякий ключ первого дерева не должен превосходить всякого ключа во втором дереве. Операция слияния, в отличие от операции объединения, таких требований не накладывает.

Одна из возможных реализаций сливаемых очередей с приоритетом — это так называемые *биномиальные очереди* [1]. Мы, однако, не

будем их изучать в рамках данного курса, а вместо этого рассмотрим другие способы — *левацкие* и *косые* кучи. Они хотя и не поддерживают все операции интерфейса очереди с приоритетом (к примеру, в них нельзя изменять значения приоритетов после добавления, а также нельзя удалить произвольный элемент), но обладают рядом дополнительных интересных качеств.

4.6. Левацкие кучи

Будем снова рассматривать бинарные деревья со свойством кучи, но, в отличие от реализации на основе массива, у нас не будет столь жестких ограничений на структуру дерева.

Пусть v обозначает вершину кучи или NULL. Определим *ранг* $r(v)$ следующим образом: если $v = \text{NULL}$, то $r(v) = 0$; иначе $r(v) = 1 + \\ + \min(\mathit{left}(v), \mathit{right}(v))$. Очевидно, ранг вершины v равен длине (выраженной числом вершин) кратчайшего пути вниз по дереву от v до вершины, у которой отсутствует хотя бы один сын.

Ранги вершин любого бинарного дерева несложно вычислить за линейное время с помощью рекурсивной процедуры обхода.

Лемма 4.6.1. *В поддереве вершины v содержится не менее $2^{r(v)} - 1$ вершин.*

Доказательство. Любой путь вниз по дереву из вершины v длины не больше $r(v)$ не выходит за пределы дерева. Разные пути заканчиваются в разных вершинах, а всего таких путей $2^{r(v)} - 1$, откуда следует заявленная оценка. □

Следствие 1. *Ранг любой вершины кучи с n элементами равен $O(\log n)$.*

Куча называется *левацкой* (*leftist*), если справедливо следующее свойство:

$$r(\mathit{left}(v)) \geqslant r(\mathit{right}(v)) \quad \text{для любой вершины } v. \tag{4.5}$$

Любую кучу можно превратить в левацкую, если обменять левого и правого сына в тех вершинах v, для которых указанное неравенство нарушается.

Будем говорить, что последовательность вершин вида $v, right(v), right(right(v)), \ldots$ образует *правый путь* из вершины v. Из определения вытекает следующее утверждение.

Лемма 4.6.2. *При движении по правому пути левацкой кучи ранги вершин уменьшаются ровно на* 1 *при каждом шаге.*

Левацкая куча хранит в каждой вершине помимо ее приоритета также ранг. Покажем, как с помощью левацких куч реализовать операции Create, Get-Min, Insert, Extract-Min и Meld.

Как обычно, операции Create и Get-Min тривиальны.

Операция Insert сводится к созданию кучи из одного элемента, а затем к слиянию. Операция Extract-Min удаляет корень из кучи, а затем сливает левое и правое поддеревья.

Итак, осталась самая трудная часть — операция Meld. Будем считать, что кучи задаются указателями на свои корни u и v. При этом данным указателям разрешается быть равными null. По соглашению это означает, что соответствующая куча пуста.

Таким образом, если $u = \text{null}$, то в качестве результата можно вернуть v. Аналогично если $v = \text{null}$, то вернем u.

Теперь можно считать, что $u \neq \text{null}$ и $v \neq \text{null}$. Также без ограничения общности предполагаем, что $pri(u) \leqslant pri(v)$ (если это не так, то u и v можно обменять местами).

Операция слияния будет рекурсивной. Первым делом объявим вершину u корнем кучи. (Действительно, именно в ней содержится минимальный приоритет среди всех вершин сливаемых куч.) Левого сына вершины u оставим без изменений, а правым назначим результат слияния $right(u)$ с v (именно здесь и происходит рекурсивный вызов).

Предположим, что мы бы закончили на этом работу процедуры Meld и вернули бы вершину u в качестве корня построенного дерева. Был бы такой метод корректным? Не совсем. При слиянии двух левацких куч получится дерево со свойством кучи, но оно не было бы в общем случае левацким. А именно, в некоторых вершинах на правом пути из корня (в результирующем дереве) условие (4.5) могло бы оказаться нарушено. Кроме того, слияние приводит к тому, что теряется информация о рангах указанных вершин. Эти две трудности можно решить одновременно. Для этого, перед тем как производить

возврат из рекурсии, мы, во-первых, заново вычисляем ранг вершины u (пользуясь определением ранга), а затем проверяем для u свойство (4.5), и если оно нарушается, то обмениваем местами левое и правое поддеревья вершины u.

Корректность изложенного метода очевидна, поэтому осталось оценить его сложность. Легко видеть, что время работы процедуры MELD, запущенной для пары вершин u и v, равно (с учетом рекурсивных вызовов) $O(r(u) + r(v))$. Действительно, в процессе слияния процедура на каждом уровне рекурсии переходит к правому сыну одной из вершин u или v, отчего сумма рангов $r(u) + r(v)$ убывает ровно на 1. Поскольку ранги растут логарифмически с ростом числа вершин, для времени выполнения MELD получается оценка $O(\log n)$.

4.7. Косые кучи

По сравнению с обычной бинарной кучей, реализованной на основе массива, у левацкой кучи есть два недостатка в плане ее хранения в памяти: во-первых, каждая вершина должна помнить указатели на левого и правого сына, во-вторых, каждая вершина должна помнить собственный ранг. (Отметим, что указатель на родителя вершине не нужен, поскольку изложенные алгоритмы всегда просматривают кучу сверху вниз, от корня к листьям.)

Но если первый недостаток, по-видимому, неустраним, то от необходимости помнить ранги в вершинах можно отказаться с помощью элегантного приема с амортизацией сложности. Получающая структура данных называется *косой кучей* (*skew heap*).

А именно, откажемся от поддержания свойства (4.5) во всех вершинах кучи. Пусть выполняется операция MELD для непустых поддеревьев u и v, причем, как и ранее, $pri(u) \leqslant pri(v)$. В случае левацкой кучи после рекурсивного вызова MELD для $right(u)$ и v алгоритм вынужден пересчитать ранг вершины u, а затем при необходимости переставить местами ее сыновей. В случае косой кучи пересчет рангов не выполняется, а обмен сыновей производится всегда.

На первый взгляд такое решение кажется странным, однако сейчас будет доказано, что в косой куче учетная сложность выполнения

операции MELD (а значит, и остальных поддерживаемых операций из интерфейса сливаемой очереди с приоритетом) составляет $O(\log n)$.

Для этого введем несколько вспомогательных определений. Назовем *весом* $w(v)$ вершины v количество вершин в поддереве с корнем v (считая саму вершину v). По соглашению считаем $w(\text{NULL}) = 0$. Вершину v (отличную от корня) назовем *тяжелой*, если $w(v) > \frac{1}{2}w(parent(v))$. Вершины, не являющиеся тяжелыми, назовем *легкими*. Вершину будем называть *плохой*, если у нее тяжелый правый сын. *Потенциалом* $\Phi(H)$ кучи H будем считать число плохих вершин в ней. Следующие свойства очевидны.

Лемма 4.7.1. *В дереве из n вершин на любом пути, идущем вниз, содержится не более* $\log n$ *легких вершин.*

Лемма 4.7.2. *У вершины не может быть два тяжелых сына.*

Докажем, что относительно так выбранной функции потенциала достигается учетная оценка $O(\log n)$ для операции MELD. Действительно, пусть происходит слияние куч H_1 и H_2 и в результате образуется куча H из n вершин. Обозначим через k_1 и k_2 количество плохих вершин на правом пути из корня в H_1 и H_2 соответственно. Время работы операции MELD пропорционально сумме длин правых путей в H_1 и H_2. Вышеуказанные пути состоят из легких и тяжелых вершин. Легких вершин всего не более $2\log n$, а тяжелых $-\ k_1 + k_2$. Итак, истинная стоимость операции MELD оценивается как $k_1 + k_2 + 2\log n$.

Представим процесс слияния состоящим из двух фаз. На первой фазе происходит сканирование правых путей от корня в H_1 и H_2 и образуется дерево H_0. Во время второй фазы происходит обмен сыновей на правом пути от корня в H_0, в результате чего формируется результирующее дерево H. Пусть k обозначает количество плохих вершин на пути от корня в H. Тогда $\Phi(H_0) = \Phi(H_1) + \Phi(H_2) - k_1 - k_2 + k$. Для подсчета потенциала при переходе от H_0 к H достаточно рассмотреть вершины на правом пути от корня. Если вершина была плохой до обмена ее сыновей, то она перестает быть таковой. С другой стороны, если вершина не была плохой, но стала таковой, то, значит, до обмена у нее был легкий правый сын. Все такие сыновья лежат на правом пути от корня, и их общее количество не превышает $\log n$. Итак, $\Phi(H) \leqslant \Phi(H_0) - k + \log n$, а значит,

$\Phi(H_1) + \Phi(H_2) - \Phi(H_0) \leqslant \log n - (k_1 + k_2)$. Таким образом, учетная стоимость операции MELD не превосходит $3 \log n$.

4.8. Структуры данных с хранением истории

Для операции MELD в интерфейсе сливаемой очереди с приоритетом было сказано, что она разрушает кучи, поступившие ей на вход в качестве аргументов. При реализации с помощью косых куч этой неприятной особенности можно избежать, так что новая куча H, представляющая собой результат объединения H_1 и H_2, будет существовать наравне с исходными кучами H_1 и H_2. Так получается структура данных, которая помимо своего текущего состояния также помнит и все ему предшествующие. По-английски такие структуры данных называются *persistent*.

Этого можно достичь, если договориться, что содержимое вершин кучи никогда не меняется после создания (*immutablility*). Особое внимание к неизменяемым структурам в последнее время обусловлено тем, что для таких структур нет необходимости в синхронизации, когда данные разделяются одновременно между несколькими потоками исполнения. Неизменяемые структуры естественным образом возникают в функциональных языках программирования.

Неизменяемая версия косой кучи устроена следующим образом. При слиянии поддеревьев с корнями u и v происходит выбор той из них, у которой приоритет минимален. Обозначим ее через w. Алгоритм, описанный в предыдущем разделе, обновляет ссылки на детей у вершины w. Неизменяемая реализация поступает иначе. На каждом шаге слияния создается новая вершина w', в которую копируется информация из w. В качестве результата возвращается ссылка на w' (а не w, как у обычного алгоритма).

Отметим два обстоятельства. Во-первых, столь простая неизменяемая реализация стала возможной потому, что вершины в куче не хранят ни рангов, ни указателей на родителей. В противном случае не получилось бы эффективно разделять одни и те же вершины между разными поддеревьями.

Во-вторых, неизменяемые реализации требуют более аккуратного управления памятью. Действительно, каждая операция слияния

выделяет все новые и новые участки памяти для хранения копий вершин. Но когда производить очистку? Для этого придется использовать *сборщик мусора* (*garbage collector*). К сожалению, его описание далеко выходит за рамки настоящего конспекта.

4.9. Декартовы деревья и дучи

В гл. 3 было изучено понятие дерева поиска. Сейчас мы опишем интересную структуру данных, представляющую собой гибрид дерева поиска и дерева со свойством кучи. Такое дерево T будет бинарным, а в вершине его будет храниться пара значений: *ключ* $key(v)$ и *приоритет* $pri(v)$. При этом относительно ключей T будет деревом поиска, а относительно приоритетов будет выполняться условие из определения дерева со свойством кучи. Такие деревья будем называть *декартовыми* (*cartesian*).

Для всякого ли набора пар приоритетов и ключей $\{(p_i, k_i)\}$ можно построить декартово дерево? Да, для любого. Чтобы доказать это, отсортируем пары в порядке неубывания ключей:

$$(p_1, k_1), (p_2, k_2), \ldots, (p_n, k_n), \quad k_1 \leqslant k_2 \leqslant \ldots \leqslant k_n. \tag{4.6}$$

Будем строить искомое декартово дерево с корня. По свойству дерева со свойством кучи в корне должна находиться пара с минимальным значением приоритета. Пусть (p_m, k_m) — такая пара. Отметим, что, поскольку приоритеты не обязаны быть различными, выбор m может быть неоднозначным. Поместим данную пару в корень, после чего рассмотрим части, на которые пара (p_m, k_m) разбивает последовательность (4.6):

$$(p_1, k_1), \ldots, (p_{m-1}, k_{m-1}) \quad \text{и} \quad (p_{m+1}, k_{m+1}), \ldots, (p_n, k_n).$$

По свойству ключей в дереве поиска первая часть должна быть целиком представлена в левом поддереве корня, а вторая — в правом. Тем самым для построения декартова дерева можно применить рекурсию. Параметрами рекурсивной процедуры Construct будут индексы l и r, определяющие левый и правый концы сегмента в формуле (4.6), для которого выполняется построение.

Какова сложность предложенного способа построения T? Предварительная сортировка требует $O(n \log n)$ времени. Однако в процессе

алгоритма каждый такой вызов просматривает выданный ему отрезок в поисках пары с минимальным значением приоритета. Тем самым сложность может оказаться в худшем случае квадратичной.

Получить более эффективный алгоритм построения декартова дерева можно, но для этого придется отказаться от вышеизложенной рекурсивной процедуры. Вместо этого предлагается просматривать последовательность (4.6) в порядке возрастания ключей и на очередном шаге i добавлять к уже построенному декартову дереву T_{i-1} пару (p_i, k_i), получая новое декартово дерево T_i.

Для того чтобы понять, как соотносятся деревья T_{i-1} и T_i, заметим, что новый ключ k_i не меньше всех уже добавленных ключей $k_1, \ldots, k_{i-1}$. Поэтому новую вершину следует разместить на конце правого пути, ведущего из корня дерева T_{i-1}. Если при этом удастся сохранить inorder-порядок всех уже существующих вершин, то требование к T_i быть деревом поиска относительно ключей окажется выполненным.

Перейдем теперь к условиям на приоритеты. Пусть новый приоритет p_i не больше всех уже рассмотренных приоритетов $p_1, \ldots p_{i-1}$. Это условие можно проверить совсем просто: необходимо и достаточно, чтобы он был не больше приоритета в корне T_{i-1}. В данной ситуации пару (p_i, k_i) можно объявить корнем дерева T_i, а все дерево T_{i-1} целиком объявить левым сыном пары (p_i, k_i). При таком перестроении оба условия на T_i окажутся выполненными.

Что же делать, если приоритет p_i не наименьший? Пусть $v_1, \ldots, v_m$ обозначает последовательность вершин на правом пути из корня дерева T_{i-1}. В частности, v_1 — это корень дерева T_{i-1}, а значит, вершина с минимальным приоритетом, а v_m — это последняя добавленная вершина. Поскольку T_{i-1} обладает свойством кучи,

$$pri(v_1) \leqslant pri(v_2) \leqslant \ldots \leqslant pri(v_m). \tag{4.7}$$

Мы уже знаем, что $p_i > pri(v_1)$. Найдем такой индекс j, что $pri(v_j) \leqslant \leqslant p_i \leqslant pri(v_{j+1})$ (возможно, $j = m$). Относительно приоритетов место для новой пары (p_i, k_i) — между вершинами v_j и v_{j+1}. Конечно, нельзя просто поместить пару (p_i, k_i) туда, поскольку тогда она не будет концом правого пути от корня, а потому в общем случае не будут выполнены неравенства на ключи. Так что придется действовать

более сложным образом: пара (p_i, k_i) объявляется правым сыном вершины v_j, а все поддерево с корнем v_{j+1} (если оно вообще существует) оказывается левым сыном для (p_i, k_i). Условия на приоритеты в новом дереве T_i следуют из таковых для T_i и неравенства $pri(v_j) \leqslant p_i \leqslant pri(v_{j+1})$. Условия на ключи также проверяются тривиально: inorder-обход для T_i получается из inorder-обхода для T_{i-1} дописыванием в конец пары (p_i, k_i), как и требовалось.

Очевидно, что, коль скоро вершина v_j найдена, необходимые перестроения дерева можно выполнить за время $O(1)$. Остается выяснить, как эффективно искать v_j. Возможное решение стоит в том, чтобы спускаться по правому пути от корня дерева T_i, пока значение приоритета $pri(v_j)$ в текущей просматриваемой вершине v_j меньше p_i. Но этот способ вновь дает квадратичную сложность построения.

Упражнение 4.9.1. Приведите пример последовательности пар ключей и приоритетов длины n, на котором достигается квадратичное время работы вышеописанной процедуры построения декартова дерева.

Изменение, позволяющее достичь линейного времени построения (в предположении, что ключи уже заранее упорядочены), совсем незначительное. А именно, нужно просматривать последовательность (4.7) не от корня вниз, а наоборот, от конца правого пути вверх до корня. Для этого достаточно в алгоритме помнить указатель на последнюю добавленную в дерево вершину (ей будет v_m), а также указатели на родителей.

Объясним, почему при таком методе время работы линейно. Для этого удобно использовать учетные стоимости операций (см. раздел 1.2). *Потенциалом* текущего декартова дерева объявим длину правого пути от корня в нем, т. е. значение m. Легко видеть, что если для нахождения подходящего индекса j алгоритму пришлось просмотреть t вершин (начиная с v_m и поднимаясь выше по дереву), то на этом шаге уменьшение потенциала составит $t-1$ (минус единица возникает в оценке из-за того, что пара (p_i, k_i) становится новым концом правого пути, так что помимо «отрезания» этого пути сразу после v_j к нему добавляется еще одна вершина). На вычисление j уходит t единиц времени, а значит, $t+1$ единиц достаточно для всей процедуры добавления. Но следовательно, учетная стоимость одного

добавления равна $(t+1)-(t-1)=O(1)$. В конечный момент потенциал не меньше, чем в начальный, значит, общее время построения декартова дерева линейно.

Упражнение 4.9.2. Реализуйте алгоритм построения декартова дерева за линейное время.

Разобравшись с алгоритмами построения декартовых деревьев, выясним теперь, для чего эти деревья могут оказаться полезными. Об одном из важных применений таких деревьев будет рассказано в главе 7, в которой речь пойдет о задачах RMQ и LCA. Сейчас же декартовы деревья интересуют нас лишь как средство реализации дерева поиска с малой (логарифмической) глубиной. На первый взгляд, они плохо подходят для этой цели: не составляет трудности привести пример такого набора ключей и приоритетов, что соответствующее ему бинарное дерево единственно и имеет линейную глубину. Скажем, годится последовательность $(1,1),(2,2),\ldots,(n,n)$.

Кроме того, при реализации структуры типа *множество* нам выдаются лишь ключи, поэтому неясно, что может служить приоритетами. Но именно свободой в выборе приоритетов мы и воспользуемся: будем при поступлении на вход для вставки очередного ключа k назначать ему случайный приоритет p. Эти приоритеты следует выбирать из достаточно обширного диапазона, чтобы минимизировать вероятность их совпадения. Далее в анализе вовсе будем предполагать, что приоритеты — это независимые случайные вещественные числа, так что событие их совпадения является невозможным.

Декартовы деревья, в которых приоритеты выбираются независимо и случайно, будем называть *дучами* по аналогии с английским словом *treap*, представляющим собой производную от слов *tree* (дерево) и *heap* (куча)[1].

Оценим математическое ожидание глубины дучи. Провести анализ нам поможет, как это ни странно звучит, алгоритм Quick-Sort, описанный в разделе 2.5. Существует прямое соответствие между процессом сортировки набора ключей, при котором разделитель выбирается случайно, равномерно и независимо, и описанной в самом

[1] Еще один вариант названия можно получить, взяв слова в другом порядке: *куча+дерево=курево*.

начале раздела рекурсивной процедурой построения декартова дерева. А именно, рассмотрим последовательность (4.6). Минимальный среди приоритетов равновероятно находится на одной из n позиций. Производимые в обоих алгоритмах рекурсивные вызовы также полностью аналогичны. Кроме того, свойство случайности, равномерности и независимости выбора не теряется при движении вниз по дереву рекурсии, так что ко всем происходящим далее вызовам применимы аналогичные рассуждения. Глубина образующегося декартова дерева совпадает с глубиной дерева рекурсии, а последняя по теореме 2.5.1 имеет оценку математического ожидания $O(\log n)$.

Тем самым математическое ожидание высоты дучи логарифмическое. Отметим, что здесь не было сделано никаких предположений о значениях ключей. Иными словами, вся происходящая рандомизация *внутренняя*, и для дучи не бывает «плохих» входных данных.

Итак, в среднем дуча имеет оптимальную глубину, т. е. хорошо сбалансирована. Следовательно, поиск ключа в ней выполним в среднем за время $O(\log n)$. Но по-прежнему неясно, как добавлять ключи в дучу и удалять из нее. Делать это удобнее всего, предварительно введя две дополнительные (и весьма полезные) операции: Split и Merge.

Операция $\textsc{Split}(T, \lambda)$ принимает на вход декартово дерево T и граничное значение λ. В результате ее работы исходное дерево T разрушается, зато появляются два новых: T_1 и T_2. В дерево T_1 попадают все пары из T, ключи которых не превышают λ. Оставшиеся пары (с ключами, бóльшими λ) оказываются в дереве T_2.

Операция $\textsc{Merge}(T_1, T_2)$ выполняет в некоторым смысле противоположное действие: получая на вход два декартовых дерева T_1 и T_2, она разрушает их, изготавливая новое дерево T, содержащее пары из T_1 и T_2. При этом, однако, требуется, чтобы любой ключ в T_1 не превосходил любого ключа из T_2.

Выразить Insert и Remove через Split и Merge несложно. Для того чтобы удалить ключ x из дерева T, произведем вызов $\textsc{Split}(T, x)$, получив на выходе деревья T_1 и T_2. В результате ключ x, если он присутствовал в T, окажется максимальным ключом в T_1, т. е. будет содержаться на конце правого пути из корня. Удалить такую вершину v из T_1 можно непосредственно, при этом ее левый сын становится на ее место. После этого T_1 и T_2 нужно слить вызовом $\textsc{Merge}(T_1, T_2)$.

Добавление ключа x к дереву T производится аналогично, но с небольшими изменениями. Вначале вызовом Split(T, x) разделим T на части T_1 и T_2. Затем создадим дерево T_3 с единственной вершиной, содержащей ключ x (в этот момент необходимо сгенерировать для x случайный приоритет). Дело завершают два вызова Merge, сливающие деревья T_1, T_3 и T_2.

Итак, осталось объяснить принцип работы алгоритмов Split и Merge и оценить их сложность. Пусть на вход Split подано дерево T. Если T пусто, то результатом работы также будут пустые деревья. Иначе пусть v — корень дерева T. Будем считать, что $pri(v) \leqslant \lambda$, другой случай симметричен. Удалим связь между вершиной v и ее правым сыном $right(v)$. Вызовем для $right(v)$ рекурсивно алгоритм Split. Пусть в результате образовались деревья T_{21} и T_{22}. Положим $T_2 := T_{22}$. Для получения T_1 возьмем дерево T (в котором, напомним, отрезана правая ветвь из вершины v) и объявим T_{21} правым сыном вершины v. Результатом и будет T_1. Поскольку на каждом шаге своей работы Split спускается вниз по дереву, время работы оценивается как $O(h)$, где h — глубина дерева.

Теперь рассмотрим вызов Merge(T_1, T_2). Пусть v_1 (соответственно v_2) обозначает корень дерева T_1 (соответственно T_2). Будем считать, что $pri(v_1) \leqslant pri(v_2)$, другой случай симметричен. Возьмем v_1 в качестве корня дерева T, $left(v_1)$ в качестве левого сына корня, а в качестве правого сына — результат слияния поддерева с корнем $right(v_1)$ и T_2. Проверка корректности такого способа построения оставляется читателю в качестве упражнения. Сложность операции Merge оценивается как $O(h)$.

С учетом логарифмической оценки на математическое ожидание глубины h дучи заключаем, что операции Insert и Remove работают в среднем за время $O(\log n)$.

4.10. Задачи

Задача 1. Дано k файлов с целыми числами. Числа в каждом файле отсортированы по неубыванию. Необходимо создать один общий файл, содержащий все числа, которые были в этих файлах, в порядке неубывания, $2 \leqslant k \leqslant 1\,000\,000$.

Считается, что файлы представлены в виде потоков, которые можно читать лишь последовательно.

Решение. В каждый момент времени каждый файл будет частично прочитан, и у нас будет текущее число, прочитанное последним из данного файла.

Создадим кучу, в которой будут храниться эти числа. При этом будем хранить вместе с числом также информацию о том, из какого файла данное число было прочитано. Далее повторяем следующие шаги, до тех пор пока куча непустая.

1. Достаем из кучи минимальное число и записываем его в выходной файл.
2. Читаем очередное число из того файла, в котором было минимальное. Если не дошли до конца файла, то добавляем новое число в кучу.

Таким образом, в куче в каждый момент времени будут все минимальные числа из входных файлов, еще не записанные в выходной файл, и на каждом шаге мы будем доставать минимальное из них и записывать его в выходной файл, а значит, получим правильный выходной файл.

Сложность этого алгоритма $O(N \log k)$, где N — общее количество элементов во всех входных файлах, так как на каждую из операций уходит $O(\log k)$ времени. Затраты памяти — $O(k)$.

Задача 2. Выведите в порядке неубывания первые M сумм вида $a^3 + b^3$, где a и b — натуральные числа, не превосходящие n ($1 \leqslant M \leqslant 1\,000\,000$, $1 \leqslant n \leqslant 1\,000$, $M \leqslant n^2$). Если одно и то же число представляется несколькими способами в виде суммы кубов натуральных чисел, выводите его столько раз, сколькими способами оно так представляется.

В единственной строке входа записаны числа M и n. На выход выдайте все M сумм в одну строку в порядке неубывания.

Пример входа	Пример выхода
4 2	2 9 9 16
7 3	2 9 9 16 28 28 35

Решение. Будем хранить в виде кучи множество пар (a, b) — кандидатов на то, чтобы быть очередной минимальной суммой вида $a^3 + b^3$. Изначально в кучу положим $\min(M, n)$ пар вида $(i, 1)$, $1 \leqslant i \leqslant \min(M, n)$. Приоритетом пары (a, b) в куче будет значение $a^3 + b^3$.

Далее, на каждом шаге будем доставать из кучи минимальную пару (a, b), выводить число $a^3 + b^3$ и добавлять в кучу пару $(a, b + 1)$, если $b + 1 \leqslant n$. Таким образом, у нас в каждый момент времени в куче для каждого a будет находиться пара (a, b) с минимальным еще не рассмотренным значением b.

Так сделаем M шагов и напечатаем все M нужных чисел. Пары, для которых $a > n$, нам не понадобятся по условию, а пары, для которых $a > M$ при $M < n$, не понадобятся, так как есть не менее M пар, сумма кубов которых меньше (а именно те, которые мы изначально добавили в кучу).

В каждый момент в куче будет не более $\min(M, n)$ элементов, так как после удаления элемента из кучи мы добавляем не более одного нового элемента. Всего будет выполнено M шагов, значит, сложность алгоритма составит $O(M \log \min(M, n))$. Затраты памяти равны $O(\min(M, n))$. Это существенно лучше, чем очевидный алгоритм, который сначала берет все n^2 возможных пар, сортирует их и затем выводит первые M. В худшем случае, когда $M = n^2$, сложность алгоритмов совпадает, но у предложенного алгоритма затраты памяти будут линейны, а у сортировки — квадратичны.

Глава 5. Хеширование

5.1. Прямая адресация

Ранее при изучении алгоритмов поиска предполагалось, что ключи образуют линейно упорядоченное множество, так что единственной допустимой операцией для них является сравнение. Попробуем добиться улучшения в случае, когда ключи — обычные целые числа.

Для начала предположим, что ключами в нашем контейнере будут целые числа из отрезка $[0, m-1]$. Если значение m невелико (например, не сильно превосходит n), то можно применить таблицу с *прямой адресацией* (*direct-address table*). Данная таблица представляет собой массив T размера m, ячейки которого индексируются всевозможными значениями ключей. Если от нас требуется реализовать интерфейс множества, то в ячейках таблицы следует помнить булево значение, определяющее, присутствует ли соответствующий ключ в множестве или нет. Если же мы реализуем словарь, то в ячейках следует сохранять данные, отвечающие ключам, либо специальное значение NULL, обозначающее отсутствие ключа в словаре.

Сложность операций поиска, добавления и удаления здесь составляет $O(1)$. Таким образом, данное решение оказывается оптимальным. Однако за подобную эффективность приходится расплачиваться большим расходом памяти. Действительно, несмотря на то что всего в структуре данных хранится n элементов, она состоит из m ячеек. Скажем, если ключи — это целые 32-битные числа ($m = 2^{32}$), то данный метод малоприменим. В следующем разделе мы рассмотрим его вариант, при котором расход памяти может быть сокращен ценой некоторого снижения быстродействия.

5.2. Хеш-функции

Итак, недостатком метода прямой адресации является то, что при малом отношении n/m бóльшая часть таблицы оказывается пустую-

щей. Этого можно избежать следующим образом. Предположим, что на множестве ключей K выбрана *хеш-функция* (*hash function*)

$$h\colon K \to \{0, \ldots, m-1\}.$$

Значение $h(k)$ для $k \in K$ будем называть *хеш-кодом* (*hash code*) ключа k.

Для работоспособности алгоритма никаких специальных свойств функции h не требуется, но для достижения практической эффективности важно, чтобы эта функция обладала хорошим «перемешивающим эффектом». Кроме того, хеш-функция должна быть быстро вычислима по ключу; будем предполагать, что это возможно за константное время.

Формальные определения «хорошей» хеш-функции будут отложены до раздела 5.4. Скажем лишь, что желательно, чтобы для каждого возможного значения j количество ключей, содержащихся в структуре данных и имеющих заданный хеш-код j, было не намного больше $\alpha = n/m$. Значение α называют *коэффициентом заполнения* (*load factor*). Таким образом, хотелось бы добиться своеобразной равномерности распределения хеш-кодов по набору ключей.

После того как хеш-функция выбрана, можно реализовать следующий способ хранения множества A. Заведем таблицу T размера m, элементами которой будут ключи или специальные значения NULL. Теперь ключ k следует искать в ячейке $T[h(k)]$.

Важно, что ячейки таблицы хранят не просто булевы значения (как ранее, при использовании прямой адресации), а сами ключи. Действительно, если при поиске ключа k ячейка $h(k)$ оказалась занятой, то в общем случае неверно, что занята она именно ключом k, а не другим ключом k', имеющим тот же хеш-код ($h(k') = h(k)$). Поэтому необходимо сравнить $T[h(k)]$ с k.

Ситуация совпадения хеш-кодов различных ключей называется *коллизией*. Хотелось бы подобрать такую хеш-функцию, чтобы коллизии были невозможны, но при $n > m$ сделать это невозможно по принципу Дирихле. Пусть $j = h(k_1) = h(k_2)$ при $k_1 \neq k_2$. Какой же ключ тогда следует помнить в ячейке j таблицы? Решить эту проблему призваны *методы разрешения коллизий*. Мы рассмотрим лишь некоторые из них.

В *методе цепочек* (*chaining*) все ключи, имеющие одинаковый хеш-код, попадают в одну ячейку таблицы, но сами ячейки при этом представляют собой односвязные списки. Таким образом, время поиска ключа в таблице составляет $O(l)$, где l обозначает длину списка, который нужно просмотреть. Например, если использовать наше предположение о равномерности хеш-функции, то длина списка будет $O(1 + n/m)$. Аналогичный просмотр списка потребуется при выполнении удаления. Важно понимать, что, несмотря на то что в методе цепочек используются односвязные списки, это не мешает удалению из них, поскольку, перед тем как удалить ключ, алгоритм просматривает список с самого начала. Операция добавления не требует просмотра и выполняется за время $O(1)$.

Метод цепочек является асимптотически наилучшим с точки зрения времени работы, другие методы разрешения коллизий уступают ему по этому параметру. Однако они более экономно используют память: в методе цепочек помимо хранения самих ключей мы также помним указатели, связывающие эти ключи в виде списков.

Общая схема, которая является альтернативой для метода цепочек, — это *открытая адресация*. При ее использовании $n \leqslant m$, и каждая ячейка таблицы T хранит лишь единственный ключ (либо специальное значение NULL, обозначающее отсутствие ключа). Вместо старой хеш-функции $h\colon K \to \{0, \ldots, k-1\}$ введем отображение

$$h\colon K \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}.$$

Здесь первый его аргумент имеет прежний смысл, а второй обозначает номер попытки (нумеруемой с нуля) при просмотре таблицы. Требуется, чтобы при любом фиксированном ключе k значения $h(k, 0), \ldots, h(k, m-1)$ образовывали перестановку индексов от 0 до $m-1$.

Для того чтобы найти ключ k в методе открытой адресации, обращаемся к ячейке $T[h(k, 1)]$. Если она свободна, то искомого ключа в множестве нет. Если она занята, то сравниваем $T[h(k, 1)]$ с k. При положительном результате сравнения поиск завершен. Иначе снова изучаем таблицу, но на этот раз обращаемся к ячейке $T[h(k, 2)]$ и т. д.

Тем самым возникает последовательность просмотренных позиций:

$$h(k,0), h(k,1), \ldots, h(k,j).$$

Поиск завершается, если либо $T[h(k,j)] = \text{NULL}$, либо $T[h(k,j)] = k$, либо $j = m-1$ и $T[h(k,j)] \neq k$.

Таким образом, метод открытой адресации также представляет собой своеобразный просмотр таблицы, как и линейный поиск. Важнейшее отличие состоит в том, что при использовании открытой адресации последовательность проверяемых позиций зависит от ключа.

Операция добавления нового ключа аналогична операции поиска, только теперь нужно останавливаться при первом обнаружении пустой ячейки. А вот с операцией удаления возникают трудности: после того как ключ k обнаружен в таблице при j-м просмотре ($T[h(k,j)] = k$), мы не можем очистить соответствующую ячейку, поскольку это разрушит цепочку начиная с j-й позиции (так что при поиске ключей с тем же хеш-кодом мы не обратимся к ячейкам $T[h(k,j')]$ при $j' > j$).

С этой трудностью можно бороться, если отмечать, что ячейка $T[h(k,j)]$ свободна (записав туда специальное значение DELETED). Поиск не следует обрывать при обнаружении таких освобожденных ячеек, а вот для операции добавления процедуры NULL и DELETED эквивалентны. После удаления длина просматриваемой при поиске цепочки не сокращается, так что время поиска может оказаться большим даже при небольшом коэффициенте заполнения. Это свойство, безусловно, составляет недостаток открытой адресации.

Какую именно функцию от двух переменных взять в качестве h? Оказывается, неплохих результатов можно добиться, если рассматривать функции из достаточно узкого класса. А именно, пусть $h_0 \colon K \to \{0, \ldots, m-1\}$ обозначает хеш-функцию (от одного переменного). Тогда можно положить

$$h(k,j) := (h_0(k) + j) \mod m.$$

Таким образом, при поиске алгоритм будет последовательно просматривать все ячейки таблицы начиная с $h_0(k)$. Такой способ называется *линейным* (*linear*).

Другой, более сложный вариант состоит в том, чтобы выбрать пару хеш-функций $h_0, h_1 \colon K \to \{0, \ldots, m-1\}$ и положить

$$h(k, j) := (h_0(k) + j \cdot h_1(k)) \mod m,$$

получив так называемый метод *двойного хеширования* (*double hashing*). Иными словами, ячейки таблицы просматриваются последовательно, но с шагом, зависящим от ключа k. Напомним, что при фиксированном ключе k среди значений $h(k, j)$ не должно быть повторяющихся. В противном случае возможна ситуация, когда даже при наличии свободных мест в таблице не удается найти подходящую позицию для вставки. Чтобы обеспечить это свойство для указанной функции h, нужно потребовать, чтобы все значения h_1 были взаимно просты с m. Простой способ обеспечить это — взять в качестве размера таблицы m простое число и запретить функции h_1 равняться нулю.

5.3. Примеры хеш-функций

Пока что мы совсем не касались важного вопроса о том, как выбирать хеш-функцию. Конечно, ответ здесь зависит от типа ключей и специфики задачи. Тем не менее, существуют достаточно общие рецепты, позволяющие достигать неплохих результатов на практике.

Например, если необходимо хешировать значения, представляющие собой вещественные (на самом деле рациональные) числа из полуинтервала $[0, 1)$, то подходящей может оказаться функция $h(x) = \lfloor mx \rfloor$. Иными словами, полуинтервал $[0, 1)$ последовательно разбивается на m промежутков, получающих хеш-коды $0, \ldots, m-1$.

Аналогичные соображения работают и для случая целых ключей. Тогда можно попробовать взять хеш-функцию $h(x) = x \mod m$. Такой метод хеширования иногда называют *методом деления с остатком*. Более сложный пример — *мультипликативный метод*, которому отвечает хеш-функция $h(x) = \lfloor m(cx \mod 1) \rfloor$ (зависящая от выбранной вещественной константы c).

Наконец, предположим, что хешировать надо не отдельные числа, а их последовательности $a = (a_0, \ldots, a_{n-1})$. Для такой ситуации подходящим может быть *полиномиальный метод*. Для подсчета значения хеш-функции следует предварительно фиксировать *основание* x,

представляющее собой вычет по модулю m. После этого хеш-кодом последовательности a объявляется значение полинома с коэффициентами a в точке x, вычисленное по модулю m:

$$h(a) := (a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}) \mod m.$$

5.4. Вероятностный анализ алгоритмов хеширования

В данном разделе мы попытаемся дать уточнение понятия «хорошей» хеш-функции. Как уже было сказано ранее, желательным требованием является равномерность распределения хеш-кодов. Чтобы формализовать эту идею, предположим, что значение хеш-функции от ключа k является случайной величиной, равномерно распределенной на множестве $\{0, \ldots, m-1\}$. Кроме того, предположим, что для различных ключей k_1 и k_2 хеш-коды $h(k_1)$ и $h(k_2)$ независимы. Если оба эти предположения выполнены, то скажем, что справедлива гипотеза *простого равномерного хеширования*.

Оказывается, введенного предположения достаточно для доказательства ряда полезных фактов о поведении ранее изложенных методов хеширования в среднем. Рассмотрим, например, метод цепочек для разрешения коллизий.

Теорема 5.4.1. *В предположении гипотезы простого равномерного хеширования средняя длина цепочки в хеш-таблице равна* $n/m = \alpha$.

Доказательство. Обозначим через $k_1, \ldots, k_n$ все ключи, присутствующие в хеш-таблице. Фиксируем произвольную ячейку с номером j. Для каждого ключа k_i обозначим через X_{ij} случайную величину, равную 1, если $h(k_i) = j$, и 0 в противном случае. В силу гипотезы простого равномерного хеширования $E[X_{ij}] = \frac{1}{m}$. Очевидно, длина цепочки, лежащей в j-й ячейке, равна $\sum\limits_i X_{ij}$, следовательно, ее математическое ожидание равно $E\left[\sum\limits_i X_{ij}\right] = \sum\limits_i E[X_{ij}] = \sum\limits_i \frac{1}{m} = \frac{n}{m}$. □

Отметим, что в приведенном выше доказательстве не использовался факт независимости хеш-кодов различных ключей.

Теорема 5.4.2. *В предположении гипотезы простого равномерного хеширования среднее время безуспешного поиска ключа равно* $\Theta(\alpha + 1)$.

Доказательство. Поскольку поиск завершается безуспешно, в его процессе будет полностью просмотрена соответствующая цепочка. Средняя длина цепочки равна α, откуда и следует доказываемое утверждение. □

Теорема 5.4.3. *В предположении гипотезы простого равномерного хеширования среднее время успешного поиска ключа равно* $\Theta(\alpha+1)$.

Отметим важный момент: в формулировке предполагается усреднение не только по вероятностному пространству, отвечающему хеш-функции, но также и по множеству ключей, уже добавленных в хеш-таблицу.

Доказательство. Обозначим через $k_1, \ldots, k_n$ ключи, присутствующие в хеш-таблице. Более того, будем считать, что выбранная нумерация отражает порядок, в котором эти ключи добавлялись в хеш-таблицу.

Фиксируем ключ k_i и оценим время, которое потребуется на то, чтобы (успешно) найти его в хеш-таблице. Поскольку при составлении цепочек новые ключи добавляются в их начало, количество элементов, которые придется просмотреть при поиске k_i, равно количеству ключей среди $k_i, \ldots, k_n$, которые попали в одну ячейку с k_i. Обозначим через X_{ij} случайную величину, равную 1 в случае, когда $h(k_i) = h(k_j)$, и 0 в противном случае. Отметим, что $E[X_{ii}] = 1$ и $E[X_{ij}] = \dfrac{1}{m}$ при $i \neq j$

Тогда время поиска ключа k_i равно $X_{ii} + \ldots + X_{in}$, а среднее (по всем ключам k_i) время поиска составит

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} E\left[\sum_{j=i}^{n} X_{ij}\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E[X_{ij}]\right) = \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = 1+\frac{1}{mn}\sum_{i=1}^{n}(n-i) = \\ &= 1+\frac{1}{mn}\left(n^2-\frac{n(n+1)}{2}\right) = \Theta(1+\alpha).\end{aligned}$$

□

Рассмотренная выше гипотеза простого равномерного хеширования выглядит слегка неправдоподобно. Действительно, мы ведь

прекрасно понимаем, что хеш-функция является детерминированной, а следовательно, «случайность» ее значений подразумевает ту или иную «случайность» самих ключей, подвергаемых хешированию. В случае, если противник знает устройство используемой хеш-функции, он всегда может подобрать ключи так, чтобы количество коллизий (а следовательно, и расходуемое время) было намного больше математического ожидания.

С данной трудностью можно бороться, если использовать не одну хеш-функцию, а целое семейство таковых. При этом перед началом работы выберем хеш-функцию из семейства случайно в соответствии с равномерным распределением. Такой подход называется *универсальным хешированием* (*universal hashing*). Его преимущество состоит в том, что он гарантирует хорошую в среднем производительность, причем усреднение производится не по распределению на входных ключах, а по внутреннему датчику случайности.

Дадим более строгое определение. Пусть $\mathcal{H}$ обозначает некоторое множество хеш-функций, отображающих множество ключей в $\{0, \ldots, m-1\}$. Тогда $\mathcal{H}$ называется *универсальным*, если для любой пары различных ключей k_1 и k_2 количество функций $h \in \mathcal{H}$, которые склеивают ключи k_1 и k_2, не превосходит $|\mathcal{H}|/m$. Иными словами, вероятность выбрать хеш-функцию из $\mathcal{H}$, на которой возникнет коллизия, не выше $\frac{1}{m}$. Аналогичный факт мы видели раньше для простого равномерного хеширования, однако в нем вероятность оценивалась относительно иной меры.

Оказывается, для модели универсального хеширования справедливы аналоги теорем 5.4.2 и 5.4.3.

Теорема 5.4.4. *В случае универсального хеширования среднее время безуспешного поиска ключа равно* $\Theta(\alpha+1)$.

Доказательство. Для ключей k и l обозначим через X_{kl} случайную величину, равную 1, если $h(h) = h(l)$, и 0 в противном случае. Пусть $k_1, \ldots, k_n$ обозначает множество всех ключей в хеш-таблице. Рассмотрим произвольный ключ l. Тогда длина цепочки, лежащей в ячейке $h(k)$, равна $X_{k_1,l} + \ldots + X_{k_n,l}$. Математическое ожидание каждого слагаемого в данной сумме не превышает $\frac{1}{m}$ (согласно опре-

делению универсального хеширования). Следовательно, математическое ожидание суммы не превосходит $\frac{n}{m} = \alpha$, как и утверждалось. (Дополнительная единица появилась в оценке из-за того, что любая процедура требует не менее $\Theta(1)$ единиц времени.) □

Теорема 5.4.5. *В случае универсального хеширования среднее время успешного поиска ключа равно* $\Theta(\alpha + 1)$.

Доказательство. Рассуждения аналогичны предыдущей теореме, однако теперь список $k_1, \ldots, k_n$ включает ключ l. Количество шагов при поиске оценим сверху той же суммой $X_{k_1,l} + \ldots + X_{k_n,l}$, выражающей длину цепочки. Математическое ожидание всех слагаемых здесь не превосходит $\frac{1}{m}$, кроме одного, а именно X_{ll}; для него оно равно 1. Следовательно, получаем оценку $\frac{n-1}{m} + 1 \leqslant \alpha + 1$. □

Согласно приведенным выше свойствам универсальное хеширование выглядит весьма привлекательно. Однако сразу не ясно, существуют ли вообще в природе универсальные семейства хеш-функций и насколько сложно вычислять их значения.

Оказывается, такие семейства есть, причем задаются они весьма несложно. Мы опишем универсальное семейство хеш-функций для чисел. Вначале выберем простое число p, большее m. Функции из семейства будут задаваться парой чисел a и b. Здесь a представляет собой ненулевой вычет по модулю p (т. е. фактически число из множества $\{1, \ldots, p-1\}$), а b — произвольный вычет по тому же модулю (т. е. элемент множества $\{0, \ldots, p-1\}$). Значение хеш-функции h_{ab} вычисляется так:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

Иными словами, мы вначале действуем на ключ линейным отображением $x \mapsto ax + b$, причем производим вычисления по модулю p, а затем переходим к арифметике по модулю m.

Теорема 5.4.6. *Описанное семейство хеш-функций* $\mathcal{H} = \{h_{ab}\}$ *универсально.*

Доказательство. Рассмотрим пару различных ключей k_1 и k_2. Вначале подействуем «линейным отображением по модулю p», т. е. построим значения $t_i = (ak_i + b) \mod p$, где $i = 1, 2$. Утверждается,

что $t_1 \neq t_2$. Действительно, вышеуказанное линейное преобразование является инъективным (в силу условия $a \neq 0$).

Покажем теперь, что различные пары (a, b) приводят к возникновению различных пар (t_1, t_2). Действительно, если k_1, k_2, t_1, t_2 заданы, то значения a и b выражаются однозначно (проделайте необходимые выкладки самостоятельно). Далее, имеется ровно $p(p-1)$ способов выбрать a и b и ровно столько же способов выбрать пару различных вычетов t_1 и t_2. Следовательно, каждая пара (t_1, t_2) (при $t_1 \neq t_2$) реализуема, причем ровно одним способом, так что если выбирать хеш-функцию h_{ab} случайно и равномерно распределенно из $\mathcal{H}$, то для нее пара (t_1, t_2) окажется также равномерно распределена на множестве пар с различающимися координатами.

Теперь вероятность того, что $h_{ab}(k_1) = h_{ab}(k_2)$, равна вероятности, с которой различные равномерно распределенные вычеты t_1, t_2 по модулю p совпадут по модулю m (т. е. $t_1 \equiv t_2 \pmod m$). Если значение t_1 фиксировать, то количество отличающихся от него значений t_2, дающих тот же остаток по модулю m, не превосходит

$$\left\lceil \frac{p}{m} \right\rceil - 1 \leqslant \frac{p+m-1}{m} - 1 = \frac{p-1}{m}.$$

Следовательно, вероятность получить коллизию не превосходит

$$\frac{1}{p-1} \cdot \frac{p-1}{m} = \frac{1}{m},$$

что и требовалось доказать. □

5.5. Совершенная хеш-функция

Все изученные нами до сих пор типы хеш-функций допускали коллизии. Конечно, если множество ключей, с которыми нам придется иметь дело, заранее неизвестно, то коллизии неизбежны (если только множество возможных ключей не оказывается по размеру не больше m).

Однако если известно, с какими ключами предстоит иметь дело, то можно предложить схему хеширования, свободную от коллизий. Более точно, наш план таков. Схема хеширования будет двухуровневой. На первом уровне коллизии возможны, но затем в каждой ячейке хеш-таблицы используется хеширование второго уровня, которое уже

не допускает коллизий. На обоих уровнях мы используем подходящим образом сконструированные хеш-функции из универсального семейства. При этом общий размер памяти, который требуется в данном методе, линеен по количеству ключей n.

Перейдем к более подробному описанию идеи. Начнем с хеширования на втором уровне. От него требуется полное отсутствие коллизий. Каким образом можно гарантировать это свойство? Оказывается, для этого достаточно выбрать размер хеш-таблицы квадратичным по количеству ключей.

Теорема 5.5.1. *Если использовать универсальное семейство хеш-функций для хранения n ключей, то при размере хеш-таблицы $m = n^2$ вероятность получить хотя бы одну коллизию не превосходит $1/2$.*

Доказательство. Всего существует $\binom{n}{2}$ пар ключей, хеш-значения которых могут совпасть. Каждая пара ключей дает коллизию с вероятностью не выше $1/m$. Тогда математическое ожидание количества коллизий X равно

$$E[X] = \frac{1}{m}\binom{n}{2} = \frac{1}{m} \cdot \frac{n(n-1)}{2} < \frac{1}{2}.$$

Применим теперь неравенство Маркова $P[X \geqslant t] \leqslant \frac{1}{t}E[X]$ для $t = 1$ и получим требуемое утверждение. □

Практически доказанная выше теорема означает, что хеш-функция, не дающая коллизий, не просто существует, но также может быть обнаружена в среднем за $O(1)$ проб.

Итак, используя универсальное семейство хеш-функций, можно добиться отсутствия коллизий. Почему такой метод нельзя применить на первом уровне схемы (тем самым второй уровень был бы вовсе лишним)? Причина очевидна: размер хеш-таблицы пришлось бы сделать квадратичным по n, что неэффективно.

Отсюда также следует свойство хеширования на первом уровне, которое нам потребуется. А именно, будем использовать на первом уровне хеш-таблицу размера $m = n$. Пусть n_i обозначает количество ключей, получивших (на первом уровне) хеш-значение i $(0 \leqslant i < m)$. Числа n_i зависят от выбора хеш-функции, а потому представляют

собой случайные величины. Тогда если использовать в каждой ячейке первого уровня вышеописанную схему, свободную от коллизий, то всего дополнительно потребуется $O(\sum_i n_i^2)$ памяти. Наша задача — оценить этот параметр.

Теорема 5.5.2. *Если хеш-функция на первом уровне выбрана случайно и равномерно из универсального семейства, то*

$$E\left[\sum_i n_i^2\right] < 2n.$$

Доказательство. Вначале преобразуем выражение:

$$E\left[\sum_i n_i^2\right] = E\left[\sum_i n_i + 2\binom{n_i}{2}\right] =$$

$$= E\left[\sum_i n_i\right] + 2E\left[\binom{n_i}{2}\right] = n + 2E\left[\sum_i \binom{n_i}{2}\right].$$

Сумма $\sum_i \binom{n_i}{2}$ имеет простой комбинаторный смысл — она равна общему количеству пар ключей, дающих коллизию. Поэтому для ее оценки можно применить тот же метод, что и в доказательстве предыдущей теоремы. Вероятность, с которой произвольная пара различных ключей получит одинаковый хеш-код, не превосходит $1/m = 1/n$. Следовательно, математическое ожидание общего числа коллизий не превосходит $\frac{1}{n}\binom{n}{2} = \frac{n-1}{2}$. Тогда

$$E\left[\sum_i n_i^2\right] \leqslant n + 2\frac{n-1}{2} < 2n.$$

Доказательство завершено. □

Теперь осталось применить неравенство Маркова:

$$P\left[\sum_i n_i^2 \geqslant 4n\right] \leqslant E\left[\sum_i n_i^2\right] \cdot \frac{1}{4n} \leqslant \frac{1}{2}.$$

Следовательно, за $O(1)$ проб можно найти такую хеш-функцию для первого уровня схемы, что ее применение даст линейный (по n) суммарный размер хеш-таблиц второго уровня.

Упражнение 5.5.1. Реализуйте описанный метод совершенного хеширования.

5.6. Фильтр Блюма

Интересных результатов можно добиться, если рассмотреть задачу поиска в множестве A, при котором разрешается односторонняя ошибка. А именно, будем считать допустимой ситуацию, когда алгоритм ошибочно сообщает о присутствии ключа, которого в действительности в A нет. Данная ситуация называется *ложноположительным* ответом (*false positive*). Одновременно запрещается выдавать *ложноотрицательные* ответы (*false negative*), т. е. объявлять об отсутствии в A ключа, который там содержится.

Естественно, нужно, чтобы ложноположительные срабатывания происходили достаточно редко и при этом их частоту можно было заранее оценить.

Возможное применение такой структуры данных — это простейшая система проверки орфографии по словарю. При этом системе разрешается не замечать опечатки в словах (т. е. иметь ложноположительные срабатывания), но если слово не проходит тест на принадлежность, то его действительно не должно быть в словаре (что, впрочем, не исключает ситуацию, что слово правильное, а словарь неполон).

Сейчас мы опишем одно из возможных решений такой задачи, известное как *фильтр Блюма* (*Bloom filter*), см. [3]. Конструктивно данный фильтр представляет собой булев массив T, индексируемый числами из отрезка $[0, m-1]$. В отличие от обычных хеш-таблиц, нам потребуется не одна, а несколько хеш-функций. Обозначим их $h_1, \ldots, h_s \colon K \to [0, m-1]$.

Изначально множество A пусто, все значения в массиве T равны FALSE. Добавление ключа k выглядит следующим образом: вычисляются его хеш-коды $h_1(k), \ldots, h_s(k)$, после чего в соответствующие ячейки T записывается значение TRUE. Проверка ключа k на принадлежность к A также очень проста: если

$$T[h_1(k)] = T[h_2(k)] = \ldots = T[h_s(k)] = \text{TRUE},$$

то считаем, что $k \in A$, иначе $k \notin A$.

Видно, что при таком подходе возможны ложноположительные срабатывания (требуемая комбинация логических единиц в позициях $h_1(k), \ldots, h_s(k)$ могла быть получена не только добавлением ключа k

в A, но и другими способами). С другой стороны, ложноотрицательных срабатываний случиться не может.

Также видно, что удалять элементы из A не получится, поскольку неизвестно, скольким ключам соответствует логическая истина в какой-либо ячейке таблицы T. Данную трудность, конечно, легко преодолеть, если использовать счетчики вместо булевых значений. Это, однако, увеличит объем занимаемой фильтром памяти.

Упражнение 5.6.1. Реализуйте фильтр Блюма с возможностью удаления и без таковой.

Осталось оценить, насколько велика вероятность ложноположительного ответа фильтра (и, в частности, относительно какого распределения вычисляется данная вероятность). Анализ проведем, предполагая, что для каждого ключа k значения $h_1(k), \ldots, h_s(k)$ представляют собой независимые равномерно распределенные случайные величины. Пусть в фильтр были добавлены n ключей. Фиксируем произвольный бит i в таблице. Одна хеш-функция выдает значение, отличное от i, с вероятностью $1 - 1/m$. Вероятность того, что после вставки n ключей данный бит все еще будет равен FALSE, составляет $(1 - 1/m)^{sn}$ (с учетом независимости соответствующих событий). Следовательно, для произвольного бита вероятность того, что он равен TRUE, составляет $1 - (1 - 1/m)^{sn}$. Предположим, что мы проверяем на вхождение в A ключ k, который там отсутствует. Тогда случайные величины $h_1(k), \ldots, h_s(k)$ независимы (в совокупности) с $h_i(a_j)$ $(1 \leqslant i \leqslant s,\ 1 \leqslant j \leqslant n)$, откуда следует, что вероятность ложноположительного срабатывания составляет

$$\left(1 - \left(1 - \frac{1}{m}\right)^{sn}\right)^s.$$

В случае, когда sn/m достаточно велико, можно заменить $(1 - 1/m)^{sn}$ на $e^{-sn/m}$, так что вероятность ложноположительного срабатывания приближенно равна

$$(1 - e^{-sn/m})^s.$$

5.7. Задачи

Задача 1. Даны три множества целых чисел A, B и C. Все числа в каждой из последовательностей различны. Также дано целое чис-

ло S. Необходимо найти все такие тройки (a, b, c), что $a \in A$, $b \in B$, $c \in C$, $a + b + c = S$.

В первой строке вводится число n_a — количество элементов в A. Во второй строке содержится n_a чисел — элементы множества A. В третьей и четвертой строках аналогичным образом закодирована последовательность B, а в пятой и шестой строках — последовательность C. В седьмой строке записано число S. Ограничения: $1 \leqslant n_a, n_b, n_c \leqslant 1\,000$, $-10^9 \leqslant S \leqslant 10^9$, элементы последовательностей от -10^9 до 10^9.

Выведите количество искомых троек.

Пример входа	Пример выхода
3 1 2 3 3 1 2 3 3 1 2 3 6	6
1 3 1 3 1 3 8	0

Решение. Заведем хеш-таблицу и сложим в нее все элементы последовательности C. Затем будем перебирать все такие пары элементов (a, b), что $a \in A, b \in B$. Для каждой такой пары значение c однозначно определяется как $S - a - b$. Осталось определить, содержится ли c в C, — это мы сделаем с помощью хеш-таблицы. Здесь существенно то, что все числа в каждой последовательности различны.

Оценим сложность алгоритма. Будем считать, что длины всех трех последовательностей ограничены одним и тем же числом n. Тогда сложность алгоритма — ожидаемое время $O(n^2)$, так как мы перебираем $O(n^2)$ пар и для каждой из них ищем за ожидаемое время

$O(1)$ значение в хеш-таблице. Заметим, что если бы были разрешены одинаковые числа, то задача стала бы сложнее и пришлось бы запоминать не только числа, но и их кратности, чтобы посчитать ответ к задаче за квадратичное время. На вывод самих троек времени уже точно не хватило бы, так как их может быть $\Theta(n^3)$, если, к примеру, все числа во всех последовательностях равны a, а $S = 3a$.

Эту задачу можно решить также и за «честное» время $O(n^2)$, а не за время $O(n^2)$ в среднем. Для этого необходимо воспользоваться решением задачи 2 из гл. 2 или совершенным хешированием для хранения чисел из третьего массива.

Задача 2. Даны две строки S_1 и S_2. Длины строк не превосходят 1 000 000. Найдите их наибольшую по длине общую подстроку.

В первой строке входа записана строка S_1, состоящая из маленьких латинских букв, во второй строке — S_2.

Выведите наибольшую общую подстроку.

Пример входа	Пример выхода
aabcdeabcdef acdeabf	cdeab

Решение. Воспользуемся полиномиальным хешированием для подстрок S_1 и S_2. Хеш-кодом строки α длины k будет

$$\big(\alpha[0] + x\alpha[1] + x^2\alpha[2] + \ldots + x^{k-1}\alpha[k-1]\big) \mod p,$$

где p — некоторое простое число, x — случайно выбранный остаток по модулю p. Для того чтобы быстро рассчитывать хеш-коды для подстрок строки S, нужно делать некоторый предрасчет. Во-первых, предварительно подсчитаем все степени числа x от $-n+1$ до $n-1$, где n — длина S, и сложим их в массив P. Во-вторых, предварительно подсчитаем хеш-коды всех префиксов строки S и сложим их в массив H. Теперь для того, чтобы посчитать хеш-код подстроки $S[i:j]$, достаточно вычислить за время $O(1)$ значение

$$P[-i](H[j] - H[i-1]) \mod p.$$

Теперь будем решать исходную задачу с помощью бинарного поиска по ответу. Ясно, что если есть общая подстрока длины l, то и для всех $l' < l$ тоже есть общая подстрока такой длины, поэтому бинарный поиск возможен.

При фиксированной длине строки l определим, есть ли у S_1 и S_2 общая подстрока длины ровно l. Для этого рассчитаем хеш-коды всех подстрок S_1 длины l (всего их $O(|S_1|)$) и сложим их в хеш-таблицу: ключом будет значение хеш-функции, а значением — индекс начала подстроки. Теперь пройдемся по всем подстрокам S_2 длины l и каждую попытаемся найти в хеш-таблице. Поиск будет выполнен за время $O(1)$, если воспользоваться совершенным хешированием. Значит, за время $O(n)$ можно определить, есть ли у S_1 и S_2 общая подстрока длины ровно l, и если есть, то какая именно подстрока. Таким образом, сложность решения $O(n \log n)$, памяти необходимо линейное количество на сами строки, вспомогательные массивы для быстрого подсчета хеш-кодов подстрок и для хеш-таблицы на каждом шаге бинарного поиска.

Эта оценка сложности не учитывает того факта, что теоретически могут совпасть хеш-коды у подстрок, даже если сами подстроки различаются. Однако вероятность такого события легко сделать очень маленькой с помощью выбора достаточно большого числа p, а алгоритм, точно выдающий ответ с учетом возможности совпадения хеш-кодов разных подстрок, работает за время $O(n \log n + nK)$, где K — количество неудачных попыток, т. е. таких, когда две разные строки получили один и тот же хеш-код. Для того чтобы добиться этого, просто каждый раз, когда мы находим в хеш-таблице нужное значение, будем сравнивать строки полностью на совпадение. Каждое сравнение строк занимает $O(n)$ времени, поэтому при условии, что все попытки удачные, каждый шаг бинарного поиска будет работать те же $O(n)$ единиц времени, а на каждую неудачную попытку добавляется $O(n)$ лишних операций на сравнение.

Глава 6. Системы непересекающихся множеств

6.1. Постановка задачи

Начав с изучения простейших структур данных (массивы переменного размера) и алгоритмов (сортировки), мы постепенно переходим к описанию задач, которые уже не кажутся столь естественными, по крайней мере на первый взгляд. Однако последующее изучение предмета еще не раз докажет нам их полезность.

Настоящая глава посвящена структуре данных под названием *система непересекающихся множеств* (*disjoint set union*), которая устроена следующим образом. Рассмотрим конечное множество U — *универсум*. Пусть, как это обычно бывает, n обозначает число элементов в нем. В любой момент времени система непересекающихся множеств $\mathcal{D}$ хранит в себе разбиение U на семейство непересекающихся подмножеств. Основной запрос, на который может отвечать $\mathcal{D}$, таков: по заданной паре элементов $x, y \in U$ выяснить, попадают ли они в одно подмножество (относительно разбиения $\mathcal{D}$) или в разные.

Для этой цели в каждом таком подмножестве A один из элементов выбран как *канонический*; обозначим его через $c(A)$. Отметим, что какой именно элемент окажется каноническим — целиком и полностью зависит от реализации структуры данных. Пользователь не должен делать каких-либо предположений о значениях $c(A)$ и не в силах повлиять на их выбор.

Теперь, имея пару элементов $x, y \in U$, можно выяснить, попадают ли они в одно подмножество, путем сравнения канонических элементов $c(X)$ и $c(Y)$ соответствующих подмножеств X и Y ($x \in X$, $y \in Y$). Иными словами, для пользователя системы непересекающихся множеств оказывается полезной операция Get-Root(a), которая выдает канонический элемент (называемый далее *корнем*) в подмножестве, содержащем a.

Приведем теперь полный список операций, поддерживаемых системой $\mathcal{D}$:

1) CREATE — создать структуру, отвечающую пустому универсуму U;
2) INSERT(x) — добавить элемент x к U и создать синглетон $\{x\}$ из него;
3) GET-ROOT(x) — вернуть корень a;
4) UNITE(x, y) — объединить подмножества, содержащие элементы x и y (в случае, если x и y уже лежат в одном подмножестве, запрос игнорируется).

Системы непересекающихся множеств оказываются полезными во многих задачах теории графов, о которых речь пойдет во второй части нашего курса.

Тривиальная реализация вышеуказанного интерфейса такова: с каждым элементом $x \in U$ свяжем его корень, который обозначим $p(x)$. Тогда добавление нового элемента x в универсум сводится к присваиванию $p(x) := x$. Операции CREATE и GET-ROOT тривиальны. Однако выполнить эффективно операцию UNITE(x, y) уже не получится, поскольку потребуется изменить указатели на корни всех элементов в одном из множеств X или Y ($x \in X$, $y \in Y$).

Если всегда действовать единообразно и изменять указатели на корни в множестве X или Y, то сложность операции UNITE составит $\Theta(n)$ (легко придумать соответствующий пример последовательности запросов). Конечно, на ум сразу приходят различные идеи оптимизации. Например, если бы мы знали, какое из двух множеств (X или Y) больше, то можно было бы изменять указатели на корни в меньшем множестве. Такое решение потребует от нас хранения не только указателей p, но и списков элементов в каждом из множеств, а также счетчика, отмечающего количество элементов в них. Можно показать, что учетная сложность операций составит $O(\log n)$. Этот метод, однако, оказывается более громоздким и менее эффективным, чем другой, к изложению которого мы и приступаем.

6.2. Лес непересекающих множеств

Описанный ранее прием с хранением указателей на корни в сущности эквивалентен тому, что мы предварительно подсчитаем ответы на

все возможные запросы GET-ROOT. Поступим теперь иначе. Для каждого множества X разбиения построим некоторое направленное дерево T_X, вершины которого будут отвечать элементам множества X. Для элемента x его родителя обозначим $p(x)$. В случае, когда x — корень дерева T_X, положим $p(x) = \text{NULL}$. Каноническим элементом $c(X)$ объявим корень дерева T_X. Теперь ответ на запрос GET-ROOT(x) уже не столь тривиален, как ранее. Необходимо построить последовательность

$$x,\ p(x),\ p(p(x)),\ \ldots$$

и найти в ней последнее не NULL-значение y. Очевидно, y будет корнем дерева T_X $(x \in X)$.

Описанный метод носит название *леса непересекающихся множеств*. Заметим, что старый способ является его частным случаем, при котором разрешены лишь деревья специального вида, состоящие из корня с присоединенными к нему листьями.

Операция UNITE(x, y) выполняется следующим образом: вначале мы находим корни соответствующих поддеревьев x' и y' (выполняя запросы GET-ROOT(x) и GET-ROOT(y)). В случае, если $x' = y'$, элементы x и y уже содержатся в одном подмножестве, так что никакие дополнительные действия не требуются. Иначе $x \in X$, $y \in Y$, $T_X \neq T_Y$. Для того чтобы объединить деревья T_X и T_Y в одно дерево, добавим дугу между корнями поддеревьев x' и y'.

В каком же направлении добавлять дугу? Как и ранее, резонно было бы подключить меньшее дерево в качестве поддерева к большему. Иными словами, если T_X «меньше» T_Y, то следует выполнить присваивание $p(x') := y'$, а иначе $p(y') := x'$.

Осталось формализовать, что мы понимаем под «бóльшим» деревом. Пусть $r(x)$ обозначает высоту (выражаемую числом дуг на длиннейшем пути до листа) в поддереве с корнем в x. В процессе слияния будем сравнивать множества по высоте отвечающих им деревьев. Значения r будут храниться в процессе работы наряду со значениями p. Обсудим, как знание этих высот отразится на сложности операций, а также как поддерживать корректные значения r в процессе модификации леса.

Во-первых, снова рассмотрим запрос UNITE(x, y), положим $x :=$ $:=$ GET-ROOT(x), $y' :=$ GET-ROOT(y) и предположим, что $x' \neq y'$. Если

$r(x') < r(y')$, то следует присвоить $p(x') := y'$. Если $r(y') < r(x')$, то аналогично $p(y') := x'$. Заметим важное обстоятельство: в обоих указанных случаях значения r не требуют обновления. Действительно, единственная величина, которая могла бы потребовать коррекции, — это значение r в корне вновь образованного дерева ($r(y')$ в первом случае и $r(x')$ во втором). Однако, поскольку мы подключаем дерево строго меньшей высоты, высота большего дерева остается неизменной.

Остался неразобранным случай $r(x') = r(y')$. Поскольку у нас нет причин предпочесть одно из деревьев T_X, T_Y другому, положим $p(x') := y'$. Кроме того, теперь высота образовавшегося дерева на единицу больше высоты деревьев T_X и T_Y (в этом легко убедиться, если добавить к длиннейшему пути от листа до корня в T_X еще и вершину y'), так что следует увеличить значение $r(y')$ на единицу.

Подход к слиянию деревьев, при котором меньшее дерево подключается к большему на основании сравнения чисел, отвечающих их размерам, носит название *ранговой эвристики*. Соответственно, значения высот r мы будем назвать *рангами* деревьев.

Описание реализации полностью завершено, переходим к оценке сложности операций. Заметим, что, не считая двух вызовов процедуры GET-ROOT, сложность алгоритма UNITE составляет $O(1)$. Таким образом, требуется лишь оценить сложность алгоритма GET-ROOT(x). Последняя же равна высоте дерева T_X, содержащего элемент x. Докажем, что ранговая эвристика служит хорошим средством балансировки, так что деревья имеют высоту $O(\log n)$.

Лемма 6.2.1. *Поддерево с корнем в вершине x содержит не менее $2^{r(x)}$ вершин.*

Доказательство. Доказательство проведем по индукции. Сразу после создания синглетона $\{x\}$ ранг $r(x)$ равен 0, что согласуется с утверждением леммы. Пусть теперь происходит объединение деревьев T_X и T_Y с корнями x и y соответственно. Если $r(x) \neq r(y)$, то ранги вершин не меняются, а количество вершин в поддеревьях лишь увеличивается, так что требуемое неравенство по-прежнему выполняется. Осталось рассмотреть случай $r(x) = r(y)$. Тогда x объяв-

ляется сыном y и ранг y увеличивается на единицу. По индуктивному предположению в дереве, полученном после объединения, содержится не менее $2^{r(x)} + 2^{r(y)} = 2^{r(y)+1}$ элементов. Следовательно, увеличение $r(y)$ на единицу не нарушит справедливости неравенства. □

Из доказанной леммы немедленно следует оценка $O(\log n)$ на время работы алгоритма GET-ROOT(x). Действительно, его можно оценить как $O(1 + r(x'))$, где x' — корень дерева T_X, содержащего x. В данном дереве не более n вершин, поэтому $2^{r(x')} \geqslant n$, а значит, $r(x') \leqslant \log n$.

Можно ли далее ускорить алгоритм? Оказывается, ответ положительный. В процессе вычисления GET-ROOT(x) будем подниматься вверх по дереву, пока не дойдем до корня x'. Заметим, что при выполнении следующего подобного запроса GET-ROOT(x) мы будем вынуждены снова подниматься по тому же самому пути P. Этого, однако, легко избежать. После того как корень x' найден, можно снова просмотреть путь P (от x до x') и для каждой вершины y на нем выполнить присвоение $p(y) := x'$. Таким образом, мы перестраиваем дерево, поднимая вершины пути P ближе к корню. Отметим, что единственный требуемый инвариант — каждое из подмножеств в $\mathcal{D}$ представлено отдельным деревом — при таком преобразовании сохраняется.

Преобразование, обновляющее ссылки на родителей в вершинах пути P, называется *эвристикой сжатия путей*. Несложно понять, что добавление ее к ранговой эвристике может лишь уменьшить время работы операции GET-ROOT. Однако при совместном применении этих эвристик значения r уже не носят прежнего простого смысла высот поддеревьев. Анализ сложности алгоритма оказывается весьма нетривиальным, поэтому мы не приводим его в настоящем конспекте, а отсылаем читателя к книге [1].

Сформулируем лишь результат анализа. Оказывается, произвольная последовательность из m операций GET-ROOT и UNITE на n-элементном универсуме требует времени $O(m \cdot \alpha(m, n))$. Здесь $\alpha(m, n)$ обозначает так называемую *обратную функцию Аккермана*. Определение этой функции можно найти по вышеуказанной ссылке, но оно не потребуется нам в дальнейшем. Важно лишь то обстоятельство, что обратная функция Аккермана растет чрезвычайно медленно. К примеру, при $m, n \leqslant 10^{1000}$ имеем $\alpha(m, n) \leqslant 4$.

Упражнение 6.2.1. Реализуйте лес непересекающихся множеств с использованием эвристик ранга и сжатия путей.

6.3. Дополнительные операции

До сих пор не делалось никаких предположений о том, какова природа элементов множества U. Ряд интересных задач возникает, если наделить их дополнительной информацией. К примеру, пусть с каждым элементом $x \in U$ связано числовое значение (*ключ*). Требуется расширить интерфейс системы непересекающихся множеств операцией Get-Min(y), которая бы по заданному элементу y находила элемент x в подмножестве, содержащем y, обладающий минимальным ключом.

Велик соблазн объявить элемент x каноническим в своем подмножестве, тогда операция Get-Min совпадала бы с Get-Root. Однако, как уже отмечалось ранее, пользователь не вправе заказывать структуре данных, какие элементы ей следует делать каноническими. В действительности именно этой степенью свободы — возможностью произвольно выбирать корень дерева при слиянии — мы активно пользовались.

Поэтому следует избрать иной путь. А именно, ответ на запрос Get-Min для элементов подмножества запишем в корне дерева, этому подмножеству отвечающего. Легко понять, что при слиянии двух деревьев эту информацию легко обновлять за константное время.

В общем случае можно предположить, что на элементах множества U определена некоторая ассоциативная и коммутативная бинарная операция $f(\cdot,\cdot)$. Тогда корректно говорить о результате данной операции $f(X)$, примененной ко всем элементам подмножества X (порядок применения роли не играет). Теперь для каждого подмножества X в U значение $f(X)$ можно записать в корне дерева T_X. При слиянии двух множеств X и Y мы пользуемся соотношением $f(X \cup Y) = f(f(X), f(Y))$ и обновляем информацию в корне объединенного дерева.

Упражнение 6.3.1. Реализуйте вариант леса непересекающихся множеств, в котором каждый элемент универсума снабжен числовым ключом, а дополнительной операцией является нахождение второго по величине ключа в подмножестве, содержащем заданный элемент.

Наконец, для некоторых приложений требуется уметь обращать операции слияния UNION. Получаемая структура данных носит название системы непересекающихся множеств *с разделением* (disjoint set union *with deunions*). Более точно, заметим, что каждое множество X, $|X| > 1$, было образовано в результате объединения некоторых двух непустых подмножеств Y и Z. Добавим в интерфейс структуры данных дополнительную операцию DEUNION(x), $x \in X$, которая отменяет объединение Y и Z. Тем самым в $\mathcal{D}$ исчезает множество Z, а вместо него появляются множества X и Y.

Будем в каждой вершине x хранить стек $s(x)$. В этот стек будем складывать корни деревьев, «подцепляемых» к x в качестве сыновей. Иными словами, в процессе слияния деревьев с корнями x и y вместе с присвоением $p(y) := x$ будем также добавлять y в стек $s(x)$. Теперь для того, чтобы отменить последнее слияние, произошедшее в дереве с корнем x, извлечем y из стека $s(x)$ и отделим поддерево с корнем y, полагая $p(y) := y$.

При таком подходе применение эвристики сжатия путей становится затруднительным. Действительно, ее использование может привести к тому, что в момент вызова DEUNION(x) деревья, отвечающие Y и Z, уже не существуют ни в каком естественном виде. Эвристика ранга, однако, полностью сохраняется. Поэтому, в частности, сложность операций при указанной реализации оказывается равной $O(\log n)$.

Упражнение 6.3.2. Реализуйте систему непересекающихся множеств с разделением.

6.4. Задачи

Задача 1. Феодальное государство начинает свое существование в виде n никак не связанных городов-государств, пронумерованных от 1 до n. Периодически один из городов нападает на другой, захватывает его и таким образом присоединяет к своей стране. Ваша задача — определить после каждого такого происшествия самую сильную страну, т. е. страну с самым большим количеством городов. Из стран с одинаковым количеством городов нужно определить ту, у которой наименьший номер города — наименьший из всех.

В первой строке входа записаны два числа n и m — количество городов в стране, а также количество захватов одними городами

других. В каждой из следующих m строк записаны два числа a и b, $1 \leqslant a, b \leqslant n$, $a \neq b$. Это означает, что город a захватил город b. Захваты перечислены в хронологическом порядке.

Выведите количество городов и наименьший номер города в самой сильной стране после каждого захвата.

Пример входа	Пример выхода
4 3 3 4 2 3 1 2	2 3 3 2 4 1

Решение. Будем хранить страны с помощью системы непересекающихся множеств. В корне каждой компоненты системы непересекающихся множеств будем хранить следующую информацию:

1) индекс родительской вершины в системе непересекающихся множеств;
2) ранг для эвристики рангов;
3) количество городов в стране;
4) наименьший индекс города в стране.

В остальных вершинах также будет содержаться индекс родительской вершины в системе непересекающихся множеств, а дополнительная информация (п. 2—4) будет как-то заполнена, но может быть некорректной, к ней никогда не будет обращений.

Изначально есть n множеств из одного города, наименьший индекс совпадает с индексом самого города. Покажем, что такую информацию можно поддерживать. Действительно, при вызовах процедуры Get-Root ничего менять не нужно, так как дополнительная информация в этот момент не меняется. При вызовах процедуры Unite возможны два случая. Если один город захватывает другой город в своей же стране, то ничего делать не нужно. Если же город захватывает город в чужой стране, то количество городов в странах нужно просуммировать, наименьший индекс выбрать как минимум из аналогичных значений для двух стран и записать эту информацию в корень соответствующего множества.

Сложность решения составляет $O(m \cdot \alpha(m, n))$, затраты памяти $O(n)$, так как сами события — захваты городов — можно не хранить, а считывать и выполнять по одному.

Задача 2. В Версале, чтобы ограничить поток туристов, гуляющих по некоторым особо популярным аллеям парка, ввели систему входных билетов разного уровня доступа. Турист может купить билет за c евро, где c — любое положительное целое число, и это позволит ему гулять по всем аллеям, популярность которых не превосходит c. Система аллей парка представляет собой неориентированный граф. Вершинами являются достопримечательности, пронумерованными от 1 до n, а ребрами — аллеи. Каждая аллея соединяет собой две достопримечательности, по аллее можно гулять в обе стороны. Для каждой аллеи известна *популярность* p и *длина* l. Купить билет и начать прогулку по парку можно в любой из достопримечательностей парка. Купленный билет позволяет гулять по любой из доступных аллей произвольное количество раз.

Нескольких *русских* туристов, прибывших в парк, не интересуют популярные аллеи и известные достопримечательности, им просто хочется хорошенько прогуляться в парке. У каждого из туристов есть с собой определенное количество денег. Туристы интересуются, какой наибольшей суммарной длины доступных аллей парка каждый из них сможет достичь при своих ограничениях на количество потраченных денег, при том что начать прогулку он может в любой из достопримечательностей парка.

Ваша задача — помочь туристам, для каждого из них определить наибольшую возможную суммарную длину доступных аллей парка, наименьший номер достопримечательности, с которой ему при этом можно начинать прогулку, а также цену билета, который необходимо приобрести (она может оказаться меньше ограничений по количеству денег для некоторых из наших туристов, так называемых *новых русских*). Каждый турист собирается совершить только одну прогулку по парку.

В первой строке входа записаны три целых числа n, m и k — количество достопримечательностей и аллей в парке, а также количество туристов ($1 \leqslant n, k, m \leqslant 100\,000$). В следующих m строках

описания аллей записано по 4 целых числа a, b, p, l, где a, b — номера достопримечательностей, соединяемых аллеей, $1 \leqslant a, b \leqslant n, a \neq b$; p — популярность аллеи, $1 \leqslant p \leqslant 1000$, l — длина аллеи, $1 \leqslant l \leqslant 1000$. В следующих k строках перечислены суммы денег в евро, имеющиеся у каждого из туристов. У каждого туриста с собой не более 100 000 евро.

Выведите в одну строку для каждого туриста, в порядке перечисления туристов во входе, три целых числа: максимально возможную суммарную длину доступных аллей парка, наименьший номер достопримечательности, начиная с которой можно гулять, так чтобы суммарная длина доступных аллей была максимальной, и оптимальную цену билета, который необходимо приобрести в этих условиях.

Пример входа	Пример выхода
6 7 2 1 2 1 1 2 3 1 1 3 1 1 1 3 4 4 2 4 5 1 2 5 6 1 2 6 4 1 2 2 5	6 4 1 11 1 4

Решение. Считаем сразу все описания аллей и туристов. Отсортируем все аллеи по возрастанию популярности, а туристов — по возрастанию суммы имеющихся у них денег. Выводить ответы для туристов нам необходимо в том же порядке, в котором они были перечислены во входе, поэтому перед сортировкой туристов сохраним для каждого туриста еще его исходный номер от 0 до $k - 1$.

Понятно, что чем больше денег у туриста, тем большее количество аллей ему будет доступно, если купить билет по максимально возможной цене. Найдем для каждого туриста все ребра графа аллей, которые будут ему доступны, если купить билет по максимально возможной цене, разобьем получившийся граф на компоненты связности, найдем в каждой компоненте связности сумму длин аллей,

и таким образом найдем наибольшую возможную сумму длин доступных аллей для этого туриста. Действительно, начав прогулку в какой-либо из достопримечательностей, турист может попасть во все достопримечательности и на все аллеи из той же компоненты связности, но никак не может попасть в другие компоненты связности. Если есть несколько компонент связности с одинаковой суммой длин аллей, то нужно выбрать ту, в которой номер наименьшей участвующей достопримечательности наименьший. Оптимальная цена билета для данного туриста равна популярности самой популярной из дорог в выбранной компоненте связности.

Если для каждого туриста отдельно строить граф и находить все компоненты связности, то сложность алгоритма составит $O(k(m+n))$, что слишком много при заданных ограничениях. Поэтому будем искать компоненты связности и их характеристики постепенно, по возрастанию доступной суммы денег. При этом в граф будут постепенно добавляться новые ребра из списка. Для этого мы и отсортировали туристов по возрастанию доступной суммы денег и аллеи по популярности. Текущие компоненты связности и их характеристики будем хранить с помощью системы непересекающихся множеств. В корне каждой компоненты системы непересекающихся множеств будем хранить следующие характеристики:

1) индекс родительской вершины в системе непересекающихся множеств;
2) ранг для эвристики рангов;
3) общую длину аллей в компоненте связности;
4) наименьший индекс вершины в компоненте связности;
5) наибольшую популярность аллеи в компоненте связности.

В остальных вершинах также будет храниться индекс родительской вершины в системе непересекающихся множеств, а дополнительная информация (п. 2—5) будет как-то заполнена, но может быть некорректной, к ней никогда не будет обращений.

Изначально имеется ровно n множеств, в каждом по одной вершине, а все дополнительные параметры равны нулю.

Покажем, что эту информацию можно поддерживать. В операции Get-Root ничего не нужно изменять, так как при выполнении этой

операции дополнительная информация не меняется. При проведении операции UNITE есть два случая: когда ребро добавляется внутри уже имеющейся компоненты связности и когда ребро добавляется между компонентами связности. В первом случае нужно добавить длину новой аллеи к общей длине аллей, а также обновить, если необходимо, наибольшую популярность аллеи в компоненте. Во втором случае необходимо просуммировать общие длины аллей в корнях компонент, а также добавить длину новой соединительной аллеи, затем взять минимальный из двух наименьших индексов достопримечательностей в компонентах и взять максимум из наибольших популярностей аллей в компонентах и популярности новой аллеи.

Все компоненты будем хранить еще в упорядоченном множестве, ключом в котором будет пара из общей длины аллей в компоненте и наименьшего индекса вершины в компоненте. Считаем, что компонента A больше компоненты B, если суммарная длина аллей в A больше, чем в B, либо если суммарная длина аллей в A такая же, как в B, но наименьший индекс вершины в A меньше, чем в B. Таким образом, чем больше по этому определению компонента связности, тем лучше она подходит для туриста. Упорядоченное множество будем хранить с помощью сбалансированного бинарного дерева (см. гл. 3).

Если происходит объединение двух компонент с помощью процедуры UNITE, то сначала достаем обе компоненты из упорядоченного множества, а потом добавляем новую после объединения. Понятно, что в множестве будут храниться не сами компоненты, а индексы корней этих компонент: имея корень компоненты, можно восстановить по нему любую информацию о самой компоненте.

Дальнейший алгоритм выглядит следующим образом. Добавляя нового туриста по возрастанию суммы денег, также добавляем в систему непересекающихся множеств все аллеи с популярностью, не превосходящей количества денег у этого туриста, при этом поддерживая дополнительную информацию и упорядоченное множество компонент. Как только все необходимые ребра были добавлены, выбираем в упорядоченном множестве наибольшую компоненту — в ней и хранится ответ для туриста. Общая длина аллей компоненты — это и есть наибольшая общая длина доступных для туриста аллей,

наименьший индекс вершины в компоненте — это наименьший индекс достопримечательности, с которого туристу можно начинать прогулку, а наибольшая популярность аллеи в компоненте равна цене билета, который туристу нужно купить, чтобы гулять по всем аллеям компоненты.

В конце, когда известны ответы для всех туристов, остается только вывести их в правильном порядке, что просто, так как у нас сохранены их изначальные номера в списке.

Оценим сложность решения и затраты памяти. На сортировку аллей и туристов уйдет $O(m \log m + k \log k)$ времени. На добавление всех ребер в систему непересекающихся множеств уйдет $O(m\alpha(m, n))$ времени, но нам еще нужно поддерживать упорядоченное множество компонент, поэтому нужно еще $O(\log n)$ времени на каждый шаг. Всего мы сделаем $O(m)$ шагов, значит, общая сложность всех шагов добавления ребер равна $O(m \log n)$. Итого, имеем сложность решения $O(m(\log m + \log n) + k \log k)$. Затраты памяти составляют $O(m + n + k)$ на хранение аллей, компонент связности и туристов.

Глава 7. Задачи RMQ и LCA

7.1. Постановка задачи

В данном разделе будут изучены два малосвязанных, на первый взгляд, вопроса: *задача о минимуме на отрезке* (*range minimum query*, RMQ) и *задача о наименьшем общем предке* (*least common ancestor*, LCA). Хотя их формулировки кажутся совершенно непохожими друг на друга, эти задачи тесно связаны.

Начнем с задачи RMQ. Пусть зафиксирована последовательность из n вещественных чисел $A = (a_1, \ldots, a_n)$. Требуется реализовать структуру данных, которая по заданной паре (i, j) будет находить индекс k на отрезке $[i, j]$, отвечающее которому значение a_k минимально. Конечно, такой индекс можно найти, перебрав все возможные значения (количество которых $j - i + 1$). Такой алгоритм будем называть *прямым вычислением*. Сложность ответа на запрос при таком подходе оценивается как $O(n)$. Оказывается, существуют намного более эффективные решения, к изучению которых мы и переходим.

7.2. Динамическая задача RMQ, деревья отрезков

Помимо запросов на поиск минимума на отрезке можно также разрешить операцию $\textsc{Change}(i, x)$, которая изменяет i-й элемент последовательности A, присваивая ему значение x. Имея в виду возможность менять A, говорят о *динамической* задаче.

Один из стандартных способов получить решение — это использовать специальную структуру данных, называемую *деревом отрезков*. Это дерево является бинарным и строится для массива длины n следующим образом. Каждая вершина дерева v задает некоторый непрерывный отрезок массива, обозначаемый $\Delta(v)$. Корню отвечает отрезок $[1, n]$, покрывающий весь массив. Рассмотрим теперь произвольную вершину v, и пусть $\Delta(v) = [l, r]$. Если $l = r$, то v является

листом. В противном случае v имеет двух сыновей x и y, причем

$$\Delta(x) = [l, m] \quad \text{и} \quad \Delta(y) = [m+1, r], \quad \text{где } m := \lfloor (l+r)/2 \rfloor.$$

Таким образом, листья дерева непосредственно соответствуют элементам массива, а вершины, находящиеся на более высоких уровнях, задают отрезки большей длины. При переходе от вершины к детям текущий отрезок разбивается на две равные (или почти равные) части. Очевидно, что высота построенного дерева составляет $O(\log n)$.

Отрезки массива, отвечающие вершинам дерева, будем называть *каноническими*. Оказывается, справедливо следующее утверждение.

Лемма 7.2.1. *Любой отрезок Δ может быть представлен в виде дизъюнктного объединения непересекающихся канонических отрезков $\Delta_1, \ldots, \Delta_k$, где $k = O(\log n)$.*

Доказательство. Утверждение о том, что любой отрезок можно разбить на канонические, само по себе тривиально: достаточно взять разбиение на одноэлементные множества. Интерес представляет возможность разбить на небольшое количество канонических отрезков.

Для доказательства опишем рекурсивную процедуру DECOMPOSE, которая строит искомое разбиение заданного отрезка Δ. Помимо самого отрезка на вход ей будем также передавать вершину v, для которой будет выполнено условие $\Delta \subseteq \Delta(v)$. Итак, полная сигнатура процедуры имеет вид DECOMPOSE(Δ, v).

Изначально v представляет собой корень дерева. DECOMPOSE поступает следующим образом. Если $\Delta = \varnothing$, то искомое разбиение пусто. Если $\Delta = \Delta(v)$, то разбиение состоит из одного отрезка. Иначе ясно, что v не может быть листом. Пусть, как и ранее, x и y обозначают левого и правого (соответственно) сыновей вершины v. Для построения разложения Δ рассмотрим отрезки $\Delta_x := \Delta \cap \Delta(x)$ и $\Delta_y := \Delta \cap \Delta(y)$. Они вложены в $\Delta(x)$ и $\Delta(y)$, поэтому для построения их разложений произведем рекурсивные вызовы DECOMPOSE(Δ_x, x) и DECOMPOSE(Δ_y, y) и объединим их результаты.

Корректность процедуры очевидна. Оценим теперь количество слагаемых в разложении. Очевидно, количество таковых не превосходит общего числа вызовов процедуры DECOMPOSE, которые нам придется произвести. Докажем, что их количество составляет $O(\log n)$.

Вначале предположим, что произведен вызов *простой* процедуры DECOMPOSE(Δ, v), т. е. у отрезков Δ и $\Delta(v)$ общий левый или правый конец. Тогда несложно видеть, что при разложении Δ мы произведем $O(\log n)$ рекурсивных вызовов. (Каждый раз, когда процесс вычисления разветвляется «налево» и «направо», либо «правый» вызов получает для разложения пустой отрезок, либо «левый» — канонический отрезок.) Аналогичное наблюдение справедливо для случая, когда правые концы отрезков Δ и v совпадают.

Рассмотрим теперь общий случай. Назовем вершину v *существенной*, если для нее был произведен вызов процедуры DECOMPOSE, причем он потребовал использования рекурсии (и тем самым свелся к вызову для левого сына x и правого сына y). Если процедура DECOMPOSE была вызвана для несущественной вершины v, то ее родитель (если v не корень) уже должен быть существенным. Таким образом, достаточно доказать, что в дереве $O(\log n)$ существенных вершин.

Рассмотрим корень дерева. Если он несущественный, то доказательство завершено. Будем спускаться вниз от корня, до тех пора пока у текущей вершины есть ровно один существенный сын.

Мы можем остановиться по двум причинам. Во-первых, пусть у последней вершины v нет существенных сыновей. Тогда все существенные вершины дерева образуют путь от корня до v и, очевидно, он имеет длину $O(\log n)$.

Иначе у вершины v оба сына x и y существенные. Пусть Δ обозначает отрезок, который подлежал разложению при входе в v. Как и ранее, пусть $\Delta_x := \Delta \cap \Delta(x)$ и $\Delta_y := \Delta \cap \Delta(y)$. Тогда, поскольку x и y существенны, $\Delta \cap \Delta_x \neq \varnothing$ и $\Delta \cap \Delta_y \neq \varnothing$. Таким образом, если обозначить $\Delta(v) = [l_v, r_v]$, $\Delta(x) = [l_x, r_x]$, $\Delta(y) = [l_y, r_y]$ и $\Delta = [l, r]$, то окажется справедливым условие

$$l_v = l_x \leqslant l \leqslant r_x = l_y - 1 \leqslant r \leqslant r_y = r_v.$$

Тем самым осталось выполнить два простых вызова процедур: DECOMPOSE(Δ_x, x) и DECOMPOSE(Δ_y, y). Как уже было показано, оба они посетят логарифмическое число вершин. □

Описанный метод построения разложения вполне алгоритмичен и, будучи реализованным, требует $O(\log n)$ времени. Теперь осталось

сделать совсем немногое, чтобы получить решение задачи RMQ. Свяжем с каждой вершиной v число $m(v)$, равное минимуму среди значений массива A по каноническому отрезку $\Delta(v)$.

Если массив A нам задан изначально, то построить числа m можно снизу вверх, произведя обход дерева отрезков. Для листа v справедливо соотношение

$$m(v) = a_i, \quad \text{где } \Delta(v) = [i, i]. \tag{7.1}$$

Для внутренней вершины v имеем

$$m(v) = \min(m(x), m(y)), \quad \text{где } x \text{ и } y - \text{сыновья вершины } v. \tag{7.2}$$

Для того чтобы ответить на RMQ-запрос по отрезку Δ, применим процедуру DECOMPOSE и разложим Δ в набор канонических отрезков. Для каждого канонического отрезка $\Delta(v)$ ответ задачи нам уже известен (он хранится в $m(v)$), а потому, взяв минимум по компонентам разложения, получаем искомый минимум по Δ.

Если содержимое ячейки a_i изменяется, то потребуется обновить числа m. Для этого спустимся от корня по пути $v_1, \ldots, v_k$, перечислив все отрезки, накрывающие точку i. Применим к этим вершинам соотношения (7.1) и (7.2), восстановив корректные значения m. Данная операция потребует времени $O(\log n)$.

Отметим, что для представления дерева в памяти не нужно хранить указатели на левого, правого сына или родителя. Вместо этого можно воспользоваться представлением бинарных деревьев в виде массива (см. гл. 4). При этом, конечно, нужно позаботиться о том, чтобы дерево отрезков было почти полным. Мы оставляем подробности читателю в качестве упражнения.

Кроме того, рассмотренная конструкция является совершенно общей и может быть применена не только для поиска минимума. Несложно видеть, что данный метод без изменения переносится на случай произвольной ассоциативной бинарной операции. Например, с помощью дерева отрезков можно решать задачу RSQ (range sum query), в которой вместо минимума берется сумма элементов.

Упражнение 7.2.1. Реализуйте алгоритмы задач RMQ и RSQ, представив дерево отрезков в виде массива.

7.3. Статическая задача RMQ, предобработка

Итак, мы смогли предложить решение задачи RMQ со сложностью $O(\log n)$ на запрос. Для того чтобы улучшить эту оценку рассмотрим теперь *статическую* версию задачи, в которой массив в процессе работы остается неизменным. В таком случае ускорения можно достичь за счет *предобработки*.

Предобработка представляет собой предварительную стадию, на которой содержимое последовательности уже известно, но никакие вопросы не задаются. Суть предобработки состоит в вычислении некоторой полезной информации, которая затем может помочь ускорить ответы на запросы. В дальнейшем, говоря о сложности решения задачи RMQ, будем иметь в виду пару $(f(n), g(n))$, где $f(n)$ — сложность алгоритма предобработки, а $g(n)$ — сложность алгоритма ответа на запрос. В таких обозначениях прямое вычисление имеет сложность $(O(1), O(n))$.

Статический вариант задачи RSQ тривиален: достаточно вычислить в процессе предобработки суммы по всем начальным отрезкам массива, после чего ответ для произвольного отрезка можно получить вычитанием. В случае задачи RMQ, на которой мы теперь и сконцентрируем внимание, так поступить не получится, так как операция взятия минимума не имеет обратной. Поскольку всего количество различных запросов весьма невелико (их $O(n^2)$), можно заранее вычислить ответы на все возможные запросы. Эти ответы можно представить в виде матрицы $k[i, j]$. Такой метод называется *полным предвычислением*, его сложность $(O(n^2), O(1))$.

Упражнение 7.3.1. Покажите, что можно вычислить все ответы в задаче RMQ за время $O(n^2)$.

Конечно, такой способ вряд ли можно считать удовлетворительным при больших значениях n. Попробуем добиться лучшего баланса между временем предобработки и временем ответа на запрос. Будем называть отрезок $[i, j]$ *стандартным*, если его длина (которая равна $j - i + 1$) представляет собой степень двойки.

Лемма 7.3.1. *Любой отрезок либо является стандартным, либо представляет собой объединение двух различных пересекающихся стандартных отрезков.*

Доказательство. Рассмотрим отрезок $\Delta = [i, j]$, не являющийся стандартным. Пусть k обозначает наибольшую степень двойки, не превосходящую длины $j - i + 1$ данного отрезка. Тогда утверждается, что

$$\Delta = \Delta_1 \cup \Delta_2,$$

где $\Delta_1 = [i, i + k - 1]$, $\Delta_2 = [j - k + 1, j]$. Достаточно проверить, что $i + k - 1 \geqslant j - k + 1$. Данное неравенство эквивалентно неравенству $k \geqslant (j - i + 1)/2$, которое верно в силу выбора k. □

Это наблюдение используется в методе *разреженной таблицы*. На этапе предобработки он вычисляет ответы для всех стандартных отрезков. Таких отрезков всего $O(n \log n)$, столько же потребуется памяти и времени для вычисления.

Упражнение 7.3.2. Покажите, как вычислить ответы для всех стандартных отрезков за время $O(n \log n)$.

Теперь для ответа на запрос про отрезок $\Delta = [i, j]$ проверим, является ли он стандартным. Если да, то ответ для него уже вычислен. Иначе представим Δ в виде объединения двух стандартных отрезков, найдем (предвычисленные) минимумы для каждого из них, а затем выберем из этих минимумов наименьший. Сложность ответа на запрос будет $O(1)$, а значит, сложность всего метода оценивается как $(O(n \log n), O(1))$.

Отметим несколько обстоятельств. Во-первых, мы существенно пользовались спецификой функции «минимум», а именно — ее идемпотентностью (равенством $\min(a, a) = a$), что позволило не беспокоиться о наличии пересечения стандартных отрезков. Для функции «сумма», к примеру, такой подход не применим.

Во-вторых, неявно предполагалось, что для заданного натурального числа a $(1 \leqslant a \leqslant n)$ можно вычислить максимальную степень двойки, не превосходящую a, за время $O(1)$. Строго говоря, для этого придется на этапе предобработки вычислить массив, хранящий ответы для всевозможных значений a. На практике такой массив обычно не используют, а вычисляют искомую степень двойки с помощью битовых операций.

Дальнейшее улучшение в задаче RMQ возможно, но оно требует введения новых понятий, к изучению которых мы и переходим.

7.4. Задача LCA, сведение к задаче RMQ

Еще одна задача, в которой четко выделяется этап предобработки, формулируется в терминах теории графов. А именно, пусть задано не обязательно бинарное дерево T. В этом дереве у каждой вершины, кроме корня, есть единственный родитель, а также некоторое (возможно, пустое) множество сыновей. Для каждой пары вершин u и v рассмотрим пути от u и v вверх по дереву до корня. Пусть w обозначает первую (максимально удаленную от корня) точку пересечения этих путей. Вершину w называют *наименьшим общим предком* вершин u и v (обозначение: $w = lca(u, v)$). Термин «наименьший» связан с тем, что множество вершин VT дерева T можно рассматривать как частично упорядоченное множество, в котором запись $u \preceq v$ означает, что вершина u является потомком вершины v (возможно, $u = v$).

Отметим очевидные свойства функции lca. Например, она симметрична, т. е. $lca(u, v) = lca(v, u)$. В случае, если вершина u является потомком v или совпадает с ней (т. е. $u \leqslant v$), справедливо равенство

$$lca(u, v) = v.$$

Наша цель — разработать метод, способный во время предобработки проанализировать структуру дерева T, а затем быстро вычислять наименьшие общие предки для заданных пар вершин.

Очевидное решение, вовсе не использующее предобработку, имеет сложность $(O(1), O(n))$. (Здесь мы пользуемся ранее введенной «парной» нотацией для обозначения времени предобработки и ответа на запрос.) Существенное улучшение оценок возможно с использованием понятия *эйлерова обхода* дерева. А именно, считаем, что вершины дерева — это числа от 1 до n. Эйлеров обход — это последовательность номеров вершин, получаемая следующим рекурсивным способом. Последовательность $ET(u)$ для вершины u с сыновьями $v_1, \ldots, v_k$ получается так: если $k = 0$ (т. е. u — лист), то положим $ET(u) := (u)$, иначе

$$ET(u) = (u) \circ ET(v_1) \circ (u) \circ ET(v_2) \circ (u) \circ \ldots \circ (u) \circ ET(v_k) \circ (u),$$

где $\alpha \circ \beta$ обозначает конкатенацию последовательностей α и β. Эйлеровым обходом $ET(T)$ дерева T называется последовательность $ET(r)$, где r — корень дерева T.

Упражнение 7.4.1. Докажите, что длина эйлерова обхода дерева T с n вершинами равна $2n-1$.

С каждой вершиной u связана ее *глубина* $d[u]$, равная длине (выраженной в количестве ребер) пути от корня до u. Пусть $first[u]$ обозначает минимальный номер позиции в обходе $ET(T)$, в которой встречается u. Несложно видеть, что эйлеров обход дерева, глубины его вершин d и индексы $first$ можно вычислить за время $O(n)$.

Упражнение 7.4.2. Реализуйте вышеуказанный алгоритм построения $ET(T)$, d и $first$.

Следующее утверждение сводит задачу LCA к задаче RMQ.

Лемма 7.4.1. *Пусть задана пара вершин u и v. Рассмотрим отрезок $\Delta = [first[u], first[v]$ в последовательности $ET(T)$. Тогда минимум глубин d вершин на отрезке Δ достигается в вершине $lca(u,v)$.*

Поскольку задача RMQ имеет решение со сложностью

$$(O(n \log n), O(1)),$$

задача LCA также имеет решение с такой сложностью. Удивительным образом, существует и обратное сведение, причем оно помогает получить решение обеих задач со сложностью $(O(n), O(1))$.

7.5. Декартово дерево, сведение задачи RMQ к задаче LCA

Вернемся снова к задаче RMQ и рассмотрим числовую последовательность $A = (a_1, \ldots, a_n)$. Напомним, что *декартово дерево* представляет собой бинарное дерево T, в котором каждой вершине v назначены *ключ* $key(v)$ и приоритет $pri(v)$, причем относительно ключей T образует дерево поиска, а относительно приоритетов — кучу. Декартово дерево существует для любого набора ключей и приоритетов, предписанных вершинам, однако при совпадающих приоритетах оно может быть определено неоднозначно.

Возьмем в качестве вершин такого дерева индексы $1, \ldots, n$. Ключом вершины i объявим само число i, а приоритетом — значение a_i (при этом считаем, что куча устроена так, что меньший приоритет находится в корне). Поскольку ключи получаются упорядоченными по возрастанию, декартово дерево для набора $\{(1, a_1), \ldots, (n, a_n)\}$ строится за время $O(n)$ с помощью метода, изложенного в разделе 4.9.

Лемма 7.5.1. *Минимум в последовательности A на отрезке $[i, j]$ достигается в позиции k, где $k = lca(i, j)$ относительно вышеописанного декартова дерева.*

Доказательство. Из определения дерева поиска следует, что вершина i находится в левом поддереве вершины k или совпадает с k. Аналогично вершина j находится в правом поддереве вершины i или совпадает с i. Следовательно, $i \leqslant k \leqslant j$. Докажем теперь, что значение a_k не больше любого значения на отрезке $[i, j]$. Фиксируем произвольный индекс $l \in [i, j]$. Тогда при inorder-обходе дерева вершина l должна следовать после i (или совпадать с i), а также предшествовать j (или совпадать с j). Несложно заметить, что l лежит в поддереве с корнем k. Но по определению кучи приоритет в вершине k не больше приоритета в вершине l, что и требовалось доказать. □

Итак, для того чтобы эффективно решать задачу RMQ, достаточно предъявить эффективное решение задачи LCA. Но, поскольку последняя была решена с использованием RMQ, кажется, что пользы от построенного сведения нет.

Однако это не так. Задача RMQ, которая получается при сведении от задачи LCA, обладает специальными свойствами, позволяющими решать ее более эффективно. А именно, заметим, что глубины соседних вершин в эйлеровом обходе различаются ровно на единицу. Этим свойством мы и воспользуемся.

Итак, пусть $|a_i - a_{i+1}| = 1$ для всех i. Такой вариант RMQ будем называть ± 1-RMQ. Зафиксируем параметр k, значение которого выберем впоследствии. Разобьем последовательность A на отрезки длиной k (в случае, если длина последовательности A не делится на k, дополним последовательность подходящими значениями). Каждый из этих отрезков будем называть *блоком*. Тем самым последовательность A представляется в виде $A_1 \circ A_2 \circ \ldots \circ A_{\frac{n}{k}}$. Пусть a'_i обозначает минимум по блоку A_i. Образуем новую последовательность

$$A' = (a'_1, \ldots, a'_{\frac{n}{k}}).$$

Наш план таков: любой отрезок запроса Δ представляется в виде $\Delta_1 \sqcup \Delta_2 \sqcup \Delta_3$, где Δ_1 — конечный отрезок некоторого блока, Δ_2 составлен из целого числа подряд идущих блоков, а Δ_3 — начальный отрезок некоторого блока. Запрос поиска минимума по Δ_2 можно

переадресовать последовательности A', которая имеет меньшую длину. Для ответов на такие запросы воспользуемся методом разреженной таблицы. Тем самым время на предобработку составит $O(\frac{n}{k}\log\frac{n}{k})$. Время ответа на такие запросы будет составлять $O(1)$.

Поиск минимумов по Δ_1 и Δ_3 более сложен. Рассмотрим произвольный блок $B = (b_1, \ldots, b_k)$. Поскольку каждый последующий элемент блока отличается от предыдущего ровно на единицу, B полностью задается указанием своего первого элемента b_1, а также вектора $d \in \{+1, -1\}^{k-1}$, где $d_i = b_{i+1} - b_i$. Будем называть блок *приведенным*, если его первый элемент равен нулю. Обозначим через B^0 приведенный блок, получающийся из B вычитанием b_1 из каждого элемента, т. е. $B^0 = (0, b_2 - b_1, \ldots, b_k - b_1)$. Тогда для произвольной пары индексов i, j $(1 \leqslant i \leqslant j \leqslant k)$ минимум по отрезку $[i, j]$ в блоке B достигается в той же позиции, что и в блоке B^0.

Всего существует 2^{k-1} типов приведенных блоков. Типы можно обозначать целыми неотрицательными $(k-1)$-битными числами. На этапе предобработки для блоков $A_1, \ldots, A_{n/k}$ вычисляются и запоминаются типы соответствующих приведенных блоков $A^0_1, \ldots, A^0_{n/k}$; это требует времени $O(n)$.

Далее для каждого типа приведенного блока, а также для каждого его начального и конечного отрезка вычисляется минимум по данному отрезку. Всего необходимо подготовить и сохранить $O(k2^k)$ значений, а сам процесс предобработки можно организовать так, чтобы он требовал $O(k2^k)$ единиц времени.

Теперь для вычисления минимумов по Δ_1 и Δ_3 достаточно узнать типы приведенных блоков, содержащих эти отрезки, а затем обратиться к предвычисленной таблице, для того чтобы определить, в какой позиции Δ_1 и Δ_3 достигаются эти минимумы.

Тем самым мы по-прежнему имеем сложность $O(1)$ на каждый запрос. Сложность предобработки составляет

$$O\Big(n + k2^k + \frac{n}{k}\log\frac{n}{k}\Big).$$

Выбирая $k = \lceil\sqrt{\log n}\rceil$, получаем оценку $O(n)$. Итак, задача ±1-RMQ может быть решена со сложностью $(O(n), O(1))$. Кроме того, к ней сводится задача LCA. Таким образом, задача LCA также может быть решена со сложностью $(O(n), O(1))$. Наконец, вспомним, что ис-

пользование декартова дерева дает возможность свести задачу RMQ к задаче LCA. Последнее означает, что и общая задача RMQ имеет решение со сложностью $(O(n), O(1))$.

Упражнение 7.5.1. Реализуйте алгоритмы для задач LCA и RMQ со сложностью $(O(n), O(1))$.

7.6. Задачи

Задача 1. На прямой задано множество из n отрезков с целыми координатами концов в пределах от 1 до $2n$. Все концы отрезков находятся в разных точках, т. е. в каждой из точек от 1 до $2n$ находится ровно один из концов всех отрезков. Для каждого отрезка найдите количество остальных отрезков, с которыми он пересекается. При этом отрезки не считаются пересекающимися, если один из них вложен в другой.

В первой строке входа записано число n, $1 \leqslant n \leqslant 100\,000$, а в каждой из следующих n строк — по два целых числа a и b, координаты концов отрезка $(1 \leqslant a, b \leqslant 2n, a < b)$.

Выведите для каждого из отрезков в порядке их задания во входном файле количество других отрезков из множества, которые с ним пересекаются.

Пример входа	Пример выхода
4 1 5 2 3 4 6 7 8	1 0 1 0
4 1 5 2 6 3 7 4 8	3 3 3 3

Решение. Если перебрать все пары отрезков и проверить их на пересечение, получится алгоритм, работающий за время $O(n^2)$, что при данных ограничениях слишком долго.

Для каждой точки от 1 до $2n$ запомним номер отрезка, который в ней начинается или заканчивается, а также то, левый это конец отрезка или правый. Будем перебирать все точки по порядку слева направо. Пересечения отрезка номер i с остальными будем подсчитывать в тот момент, когда мы дошли до его правого конца. Он пересекается с двумя видами отрезков:

- с отрезками, у которых левый конец лежит на i-м отрезке, а правый конец — правее правого конца i-го отрезка;
- с отрезками, у которых правый конец лежит на i-м отрезке, а левый конец — левее левого конца i-го отрезка.

В тот момент, когда рассматривается правый конец i-го отрезка, нужно быстро посчитать количество отрезков обоих видов, пересекающихся с ним. Заведем массив A длины $2n$, в котором каждой точке будет сопоставлено число 0, 1 или -1. Изначально всем точкам сопоставлено число 0. Когда встречается левый конец отрезка (в процессе перебора точек слева направо), сопоставим ему число 1. Когда встречается правый конец отрезка, это означает, что до этого уже встречался его левый конец и ему сопоставлено число 1. В этот момент сопоставим левому концу отрезка число -1, а правому — число 1.

Рассмотрим сумму чисел на i-м отрезке, не считая его концов, в тот момент, когда рассматривается его правый конец. Если оба конца некоторого отрезка лежат вне i-го отрезка, то он не пересекается с i-м отрезком по нашему определению и его концы не дают никакого вклада в сумму. Если оба конца какого-либо отрезка лежат внутри i-го отрезка, то к этому моменту уже просмотрены оба его конца, поэтому в левом конце записано -1, в правом записано 1 и этот отрезок тоже дает нулевой вклад в сумму. Если отрезок принадлежит к первому типу отрезков, пересекающихся с i-м отрезком, то мы успели рассмотреть только его левый конец и в его левом конце записано число 1, которое прибавляется к сумме. Если отрезок принадлежит ко второму типу отрезков, пересекающихся с i-м отрезком, то рассмотрены уже оба его конца и в его правом конце записано число 1, которое прибавляется к сумме. Таким образом, сумма чисел внутри i-го отрезка равна количеству отрезков, с которыми он пересекается.

Остается только научиться эффективно изменять значения массива A в одной точке и считать сумму чисел массива A на отрезке.

Но такую функциональность как раз предоставляет дерево отрезков. Тогда операции по изменению значения в одной ячейке и по подсчету суммы на отрезке будут выполняться за время $O(\log n)$. Общая сложность алгоритма составит $O(n \log n)$ на сортировку плюс $O(n \log n)$ на выполнение всех операций последовательно над точками, итого $O(n \log n)$. Затраты памяти линейны.

Задача 2. Имеется компьютерная сеть из n компьютеров, представляющая из себя дерево: некоторые пары компьютеров соединены между собой проводами так, что между любыми двумя компьютерами существует единственный путь по проводам.

Между компьютерами, соединенными проводом непосредственно, в каждый момент определено время отклика, зависящее от нагруженности канала между ними. Если есть пара компьютеров, не соединенных непосредственно проводами, время отклика между ними равно сумме времен отклика по всем проводам на пути между ними.

Вам даны изначальные времена откликов между соединенными компьютерами, а также запросы двух видов: запрос на изменение времени отклика по конкретному проводу и запрос на измерение времени отклика между двумя заданными компьютерами. Необходимо быстро отвечать на все запросы.

В первой строке входного файла записаны два целых числа n и q — количество компьютеров в сети и количество запросов, которые необходимо обработать. В каждой из следующих $n-1$ строк записаны три целых числа a, b и c — пара компьютеров (a, b) и время отклика c между ними в начальный момент времени. В каждой из следующих q строк находится описание запроса. Описание начинается с идентификатора типа запроса: 1 означает, что изменилось время отклика по какому-то проводу; 2 означает, что нужно измерить время отклика между двумя компьютерами. В случае 1 строка будет выглядеть как $1\,a\,b\,x$, где a и b — компьютеры, соединенные проводом, а x — величина, на которую уменьшилась пропускная способность канала между ними. В случае 2 строка будет выглядеть как $2\,a\,b$, где (a, b) — пара компьютеров, между которыми нужно измерить время отклика. Ограничения: $1 \leqslant n, q \leqslant 100\,000$, $1 \leqslant a, b \leqslant n$, $a \neq b$, $0 \leqslant c \leqslant 1\,000$, $-100 \leqslant x \leqslant 100$. Гарантируется, что в результате изменений времен

отклика между парами, соединенными непосредственно проводом, эти времена никогда не выйдут за пределы отрезка $[0, 1000]$.

Для каждого запроса на измерение времени отклика выведите одну строку с результатом измерения.

Пример входа	Пример выхода
7 12 1 2 6 1 3 5 2 4 4 2 5 3 3 6 2 3 7 1 2 4 5 2 6 7 2 4 7 2 2 3 1 1 2 4 2 2 3 1 2 4 -2 2 4 5	7 3 11 15 5

Решение. Выберем произвольную вершину дерева в качестве корня. Теперь путь от вершины u до вершины v — это всегда путь от u вверх до $w = lca(u, v)$, затем путь вниз от w до v. Поэтому воспользуемся описанными ранее структурами данных для быстрого вычисления значений LCA. Будем считать, что мы умеем находить LCA двух вершин дерева за время $O(\log n)$ с предобработкой за время $O(n)$.

Можно считать, что времена отклика, указанные в задаче, — это длины ребер дерева, они могут меняться. Тогда время отклика между парой компьютеров i и j равно длине пути между вершинами i и j в данный момент.

Если i и j — вершины, между которыми необходимо измерить время отклика, $l := lca(i, j)$, r — корень дерева, а $L(a, b)$ — функция длины пути между вершинами a и b, то

$$L(i, j) = L(r, i) + L(r, j) - 2L(r, l).$$

Таким образом, если мы научимся быстро вычислять расстояние от корня дерева до любой вершины (за время $O(\log n)$), то сможем быстро вычислять и расстояние между любыми двумя вершинами (тоже за время $O(\log n)$, так как нахождение LCA занимает логарифмическое время).

Пусть длина ребра (i, j) изменяется на величину x, i — родитель вершины j в дереве. Тогда на x изменяются все величины $L(r, k)$ для всех вершин k, лежащих в поддереве с корнем j, а для всех остальных вершин k' величина $L(r, k')$ не меняется. Значит, необходима структура данных, позволяющая быстро изменить на одну и ту же величину значения $L(r, k)$ для всех k в поддереве любой вершины j, а также узнать значение $L(r, k)$ в конкретной вершине k. Эти операции напоминают операции обычного дерева отрезков для случая массива и подотрезков в нем вместо дерева и поддеревьев. Построим аналог дерева отрезков для случая дерева и поддеревьев.

Выпишем все вершины дерева в порядке эйлерова обхода, используемого для быстрого нахождения LCA. Заведем параллельный массив X. В $X[i]$, будем хранить текущее расстояние $L(r, i)$ от корня до вершины i. При этом одно и то же значение $L(r, i)$ будет храниться в массиве X несколько раз: столько, сколько раз вершина i входит в эйлеров обход. Изначальные значения $L(r, i)$ легко вычисляются в процессе эйлерова обхода с использованием изначальных времен отклика, заданных в задаче.

Теперь построим дерево отрезков над X. Оно и будет являться деревом отрезков над исходным деревом. Действительно, все вхождения вершин поддерева вершины j в эйлеров обход находятся между первым и последним вхождениями самой вершины j в эйлеров обход. Запомним позиции первого $first(j)$ и последнего $last(j)$ вхождений для каждой вершины j в эйлеров обход дерева. Когда нужно будет изменить на x все значения $L(r, k)$ для всех вершин k в поддереве вершины j, это будет означать то же самое, что изменить на x все значения массива X с индексами от $first(j)$ до $last(j)$ включительно. При этом $L(r, k) = X[first(k)]$.

Таким образом, запрос на изменение времени отклика ребра (a, b) на x выполняем следующим образом: определяем, какая из вершин a и b выше в дереве; пусть это a, тогда прибавляем x ко всем

значениям на отрезке от $first(b)$ до $last(b)$ с помощью дерева отрезков. Запрос на измерение времени отклика между a и b выполняем так: сначала находим LCA для a и b; пусть это l. Затем находим $L(r,a) = X[first(a)]$, $L(r,b) = X[first(b)]$ и $L(r,l) = X[first(l)]$. Далее вычисляем $L(a,b) = L(r,a) + L(r,b) - 2L(r,l)$ и выводим ответ.

Оба типа запроса обрабатываются за время $O(\log n)$. Таким образом, общая сложность решения составляет $O(n)$ на предобработку плюс $O(\log n)$ на каждый из q запросов, т. е. $O(n + q \log n)$. Затраты памяти линейны.

Глава 8. Динамическое программирование

В этой главе мы поговорим о большом разделе теории алгоритмов — *динамическом программировании*. Чтобы лучше понять, что такое динамическое программирование, начнем с примеров двух задач, которые решаются с его помощью. Затем найдем общие признаки, присущие этим двум задачам, и на их примере поговорим об общих принципах и признаках применимости динамического программирования. В завершение рассмотрим практическую задачу, необходимую любому поисковому сервису, которая легко решается с помощью динамического программирования.

В целом метод динамического программирования (далее для краткости — просто «ДП») позволяет решать очень разнообразные задачи, и бывают более сложные способы его применения, чем те, что будут рассмотрены в этой главе. Однако, поняв общие принципы, читатель сможет конструировать и более сложные решения, основанные на ДП, самостоятельно.

8.1. Наибольшая возрастающая подпоследовательность

Рассмотрим классическую задачу нахождения *наибольшей возрастающей подпоследовательности*. А именно, пусть задана последовательность A длины n, элементы которой обозначим через $A[i]$, $0 \leqslant i < n$. Требуется найти наибольшее значение k и последовательность $0 \leqslant i_1 < i_2 < i_3 < \ldots < i_k < n$, удовлетворяющие условию

$$A[i_1] < A[i_2] < \ldots < A[i_k].$$

Если таких последовательностей $i_1, i_2, \ldots, i_k$ для максимально возможного k несколько, требуется выбрать из них лексикографически наименьшую.

Простейший способ решать эту задачу — *полный перебор* (*brute force*). Этот способ состоит в рекурсивном переборе всех возможных

последовательностей $i_1, i_2, \ldots, i_k$, в процессе которого для текущего начала последовательности $i_1, i_2, \ldots, i_l$ поддерживается соотношение $A[i_1] < A[i_2] < \ldots < A[i_l]$. Однако легко понять, что на строго возрастающем массиве такой алгоритм рассмотрит все возможные подпоследовательности, прежде чем выдать ответ, и поэтому будет работать экспоненциальное время.

Если алгоритм немного видоизменить, чтобы он прерывал выполнение, как только определит, что подпоследовательность максимально возможной длины n подходит, то на строго возрастающем массиве A этот алгоритм уже не будет работать экспоненциальное время, однако и против алгоритмов такого типа существует тест. Рассмотрим последовательность A длины $2n$, состоящую из чисел от 1 до $2n$:

$$2,\ 1,\ 4,\ 3,\ 6,\ 5,\ \ldots,\ 2n,\ 2n-1.$$

Тогда ответом будет $k = n$ и $i_1 = 0,\ i_2 = 2,\ i_3 = 4,\ \ldots,\ i_n = 2n-2$. При этом существует всего 2^n возрастающих подпоследовательностей длины n, и нам придется в процессе рекурсии проверить их все, чтобы убедиться, что ни одну из них нельзя удлинить и получить возрастающую подпоследовательность длины $n+1$. Таким образом, полный перебор будет работать экспоненциальное время.

Чтобы решить задачу за полиномиальное время, сначала немного переформулируем ее. Будем рассматривать не только исходную задачу, а гораздо большее количество аналогичных задач. Отметим, что такой прием является стандартным для ДП.

А именно, для каждого индекса t от 0 до $n-1$ найдем лексикографически наименьшую длиннейшую возрастающую подпоследовательность в последовательности $A[0], A[1], \ldots, A[t]$, *заканчивающуюся* на элементе $A[t]$. Иными словами, найдем наибольшее значение $k = k(t)$ и последовательность $0 \leqslant i_1 < i_2 < \ldots < i_k = t$, удовлетворяющие условию $A[i_1] < A[i_2] < \ldots < A[i_k]$ и такие, что при этом последовательность $i_1, i_2, \ldots, i_k$ лексикографически наименьшая из всех последовательностей, отвечающих $k = k(t)$.

Ясно, что если решить все подзадачи, то решение всей задачи можно получить, сравнив решения для всех подзадач и выбрав наилучшее (длиннейшее, а из решений с максимальной длиной — лексикографически наименьшее).

Заметим следующую особенность новых подзадач: пусть $0 \leqslant i_1 < i_2 < \ldots < i_k = t$ — решение t-й подзадачи. Тогда $0 \leqslant i_1 < i_2 < \ldots < i_{k-1}$ будет решением подзадачи номер i_{k-1}. Действительно, подпоследовательность $A[i_1], A[i_2], \ldots, A[i_{k-1}]$, очевидно, возрастает, а если бы существовала последовательность $0 \leqslant j_1 < j_2 < \ldots < j_l = i_{k-1}$, более длинная либо той же длины, но лексикографически меньшая, то последовательность $0 \leqslant j_1 < j_2 < \ldots < j_l = i_{k-1} < i_k = t$ была бы решением t-й подзадачи, лучшим, чем найденное нами решение $i_1, i_2, \ldots, i_k$, — противоречие, так как мы выбрали лучшее решение для t-й подзадачи. Это свойство, называемое *оптимальностью для подзадач*, также является неотъемлемым признаком ДП.

Итак, чтобы теперь решить задачу, будем решать все подзадачи последовательно для t от 0 до $n-1$. Решение нулевой подзадачи тривиально. Свойство оптимальности для подзадач позволяет нам сделать так называемый *переход* динамического программирования за линейное время. Под переходом в данном случае понимается решение очередной подзадачи (t-й), когда все предыдущие подзадачи (с нулевой по $(t-1)$-ю) уже решены, с использованием результатов решения предыдущих подзадач.

Мы предложим два подхода к решению этой задачи. Первый будет состоять в следующем. Мы будем для каждой подзадачи полностью хранить весь ответ к ней, т. е. для t-й подзадачи будем хранить всю последовательность $i_1, i_2, \ldots, i_k = t$. Предположим, что все подзадачи от 0-й до $(t-1)$-й решены. Как найти ответ для t-й подзадачи? Ответ для t-й подзадачи как минимум не хуже, чем последовательность $i_1 = t$ из одного элемента, так как она удовлетворяет всем условиям. Более того, это единственная последовательность из одного элемента, удовлетворяющая всем условиям, так как последовательность должна заканчиваться элементом $A[t]$. Если же ответ лучше, то искомая последовательность состоит более чем из одного элемента. Рассмотрим ее предпоследний элемент i_{k-1}, где k — длина ответа. Тогда $i_{k-1} < t$, значит, мы уже решили подзадачу i_{k-1}. Кроме того, должно выполняться условие $A[i_{k-1}] < A[t]$. Тогда рассмотрим все возможные значения для i_{k-1}. Для каждого из них определена наилучшая последовательность $\overline{i_1}, \overline{i_2}, \ldots, \overline{i_{k-1}} = i_{k-1}$. Выберем из всех этих по-

следовательностей самую длинную, а если последовательностей максимальной длины окажется несколько, выберем лексикографически наименьшую. Припишем к ней t в конце — получим ответ для t-й подзадачи.

Этот алгоритм работает за время $O(n^3)$: для каждого t необходимо в худшем случае рассмотреть $O(t)$ вариантов для i_{k-1} и в каждом из вариантов сравнить последовательность длины $O(t)$ с текущей наилучшей последовательностью, что делается за время $O(t)$, т. е. для каждого t выполняется $O(t^2)$ действий, а значит, всего выполнится $O(n^3)$ действий.

Теперь опишем второй подход к решению этой задачи. Чтобы сократить сложность алгоритма до $O(n^2)$, воспользуемся другим стандартным приемом, относящимся к динамическому программированию. Будем запоминать ответы для подзадач не целиком, а только часть, по которой впоследствии можно будет восстановить ответ для всей задачи. За счет того, что храниться будет только часть информации, память будет использоваться экономнее, и алгоритм будет работать быстрее, но при этом все еще возможно будет восстановить полный ответ для любой подзадачи. Реально будем выводить только ответ для изначальной задачи.

Для того чтобы найти *лексикографически наименьшую* подпоследовательность, воспользуемся следующим общим приемом. Изменим тип подзадач: для каждого t потребуем найти лексикографически наименьшую длиннейшую возрастающую подпоследовательность A, *начинающуюся* в элементе $A[t]$. Такие задачи симметричны исходным, поэтому переход делается аналогично. Кроме того, не будем запоминать саму подпоследовательность, начинающуюся в элементе $A[t]$, а запомним только ее длину.

Предположим, что мы решили все подзадачи и записали длины соответствующих подпоследовательностей в массив *length*, — теперь необходимо восстановить ответ, используя только длины, а не сами подпоследовательности. Пусть L — максимальное число в массиве *length*. Тогда длина искомой подпоследовательности равна L. Сначала найдем первый элемент искомой подпоследовательности. Первым элементом будет наименьшее такое значение t, что $length[t] = L$. Очевид-

но, что с t-го элемента будет начинаться некая подпоследовательность максимальной длины. Так как ищется лексикографически наименьшая подпоследовательность (по индексам), ее первый элемент должен быть расположен как можно левее. Теперь, когда определен первый элемент, нужно найти следующий. Следующий элемент должен быть, во-первых, правее $A[t]$, во-вторых, строго больше его. Среди таких элементов нужно выбрать первое такое j, что $length[j] = length[t] - 1$. Действительно, с j-го элемента начинается возрастающая подпоследовательность длины $length[t] - 1$, значит, если добавить к ней в начало элемент $A[t]$, то получится подпоследовательность нужной длины $length[t]$, кроме того, нужно выбрать самое первое такое j, чтобы подпоследовательность была лексикографически наименьшей по индексам. И так далее, по порядку находим все L членов наилучшей подпоследовательности.

Общая схема нахождения лексикографически минимальной подпоследовательности, удовлетворяющей некоторым условиям и при этом максимизирующей некоторый функционал F, обычно сводится к тому, чтобы сначала решить аналогичную подзадачу, начиная с каждой позиции. Далее нужно найти самую первую возможную позицию, для которой получается максимально возможное значение F. Затем необходимо на каждом шаге находить самый первый элемент, который можно приписать к текущему префиксу искомой подпоследовательности, чтобы не нарушить никаких накладываемых на подпоследовательность условий и чтобы при этом существовала подпоследовательность с указанным префиксом, максимизирующая F. А именно, каждый раз добавляем к ответу первый элемент, после выбора которого еще возможно достроить последовательность до наилучшей возможной в смысле значения функционала F и при этом удовлетворяющей всем наложенным условиям.

Алгоритм теперь будет работать за время $O(n^2)$. Для каждого t от $n-1$ до 0 он просмотрит все следующие j, удовлетворяющие условиям $j > t$, $A[j] > A[t]$ (для них подзадача уже решена), найдет из них j с наибольшим значением $length[j]$ и присвоит $length[t] = length[j] + 1$. Если же ни одного подходящего j не найдется, то он присвоит $length[t] = 1$. Таким образом, массив $length$ будет построен за время $O(n^2)$, так как на каждое t уходит $O(n - t)$ действий. Теперь по

массиву *length* решение можно восстановить указанным выше образом за время $O(n)$.

8.2. Перемножение последовательности матриц

Рассмотрим другую классическую задачу, на первый взгляд ничем не связанную с предыдущей. Пусть задана последовательность матриц $A_1, A_2, \ldots, A_n$. Обозначим размерность матрицы A_i за $r_i \times s_i$. Размерности этих матриц согласованы в том смысле, что определено произведение $A_1 A_2 \ldots A_n$. Иными словами, $s_i = r_{i+1}$ для всех $i = 1, \ldots, n-1$. Произведение матриц — ассоциативная операция, поэтому порядок вычисления произведений не влияет на результат. Необходимо выбрать такой порядок перемножения (т. е. расставить скобки в выражении $A_1 A_2 \ldots A_n$), чтобы общее количество выполненных операций для перемножения всех матриц было наименьшим. Количество операций для перемножения матриц размера $p \times q$ и $q \times r$ считается при этом для простоты равным pqr (впрочем, и другие формулы, зависящие только от p, q и r, не повлияют на суть решения).

Пример: пусть $A = A_1 A_2 A_3$; A_1 имеет размер 2×50, $A_2 - 50 \times 2$, $A_3 - 2 \times 50$. Тогда если перемножать в порядке $(A_1 A_2) A_3$, то получим $2 \times 50 \times 2 + 2 \times 2 \times 50 = 200 + 200 = 400$ действий. Если же перемножать в порядке $A_1 (A_2 A_3)$, получим $50 \times 2 \times 50 + 2 \times 50 \times 50 = 5000 + {} + 5000 = 10000$ действий, т. е. в 25 раз больше.

Если решать эту задачу полным перебором, то снова получится экспоненциальный алгоритм: количество способов расставить скобки в выражении длины n равно $\frac{1}{n}\binom{2n-2}{n-1}$. Любой способ расстановки скобок теоретически может быть наилучшим, поэтому перебор всех вариантов нам не подходит.

Однако посмотрим, как бы мы могли все-таки хотя бы устроить этот перебор. В любом способе перемножения существует последнее выполняющееся умножение. В последнем умножении мы находим результат перемножения матриц $B = A_1 A_2 \ldots A_i$ и $C = A_{i+1} A_{i+2} \ldots A_n$ для некоторого i $(1 \leqslant i < n)$. Если рассмотреть произведения, производящие B и C и далее, то становится понятно, что в общем случае решается задача о наилучшем способе перемножения матриц $A_i A_{i+1} \ldots A_j$. Получаем разбиение задачи на аналогичные меньшие

подзадачи, как и в случае предыдущей задачи про наибольшую возрастающую подпоследовательность.

Чтобы решить такую подзадачу, нужно выбрать такое k от i до $j-1$, что последнее умножение применяется к матрицам $D = A_i \dots A_k$ и $E = A_{k+1} \dots A_j$. Заметим, что здесь опять возникает понятие оптимальности для подзадач: произведения в D и в E нужно вычислять независимо и наилучшим возможным способом. Действительно, иначе можно было бы заменить вычисления в D или E на лучший вариант, после чего сделать все остальное точно так же, и общее решение улучшилось бы: независимо от того, каким образом были вычислены матрицы D и E, их размеры будут равны соответственно $r_i \times s_k$ и $s_k \times s_j$. Значит, на их перемножение уйдет $O(r_i s_k s_j)$ действий независимо от того, как они были вычислены. Итак, подзадача определяется парой индексов (i, j), решается с помощью перебора всех k от i до $j-1$, решения подзадач для пар (i, k) и $(k+1, j)$, построения ответа исходя из того, что последнее умножение было между k-й и $(k+1)$-й матрицами, и выбора лучшего из всех этих решений.

Один из способов реализации такого алгоритма использует рекурсивную функцию с двумя аргументами i и j, задающими подзадачу, возвращающую наименьшее возможное количество действий для перемножения всех матриц в произведении $A_i \dots A_j$. Функция работает следующим образом: если $i = j$, вернуть 0. Иначе рассмотреть все k от i до $j-1$ и решить задачу рекурсивно для (i, k) и $(k+1, j)$. Пусть получены ответы a_k и b_k (количества действий для выполнения умножений в левой и правой частях соответственно). Тогда возможный ответ для задачи (i, j) выглядит как $a_k + b_k + r_i s_k s_j$. Выбираем наименьшее из таких чисел и возвращаем его.

При указанной реализации время все еще будет экспоненциальным, поскольку рассматриваются все возможные последовательности перемножений. Но здесь уже можно заметить, что одна и та же подзадача (i, j) решается много раз. Такая проблема решается с помощью *мемоизации* — свойственного динамическому программированию приема реализации. Заведем матрицу с ответами X, в которой на позиции X_{ij} будет храниться ответ для подзадачи (i, j). Изначально заполним ее какими-нибудь невозможными значениями, например -1. Далее, вначале в функции решения проверяем, не является ли

X_{ij} уже неотрицательным. Если является, то нужно вернуть его: это означает, что подзадача (i, j) уже однажды решена и вычисленный ответ лежит в соответствующей ячейке матрицы. Если же $X_{ij} = -1$, значит, подзадача решается в первый раз — нужно решить ее описанным выше способом, а потом, прежде чем возвращать ответ, записать его в ячейку X_{ij} матрицы с ответами. Таким образом, каждая подзадача будет решена не более одного раза. Всего подзадач столько же, сколько пар (i, j), т. е. $O(n^2)$. При этом каждая подзадача решается за время $O(j - i + 1)$, а значит, суммарная сложность алгоритма будет равна $O(n^3)$, т. е. окажется полиномиальной.

Встает вопрос, что делать, если нужно определить не только наименьшее возможное количество операций, но и конкретный способ перемножения матриц. Если полностью запоминать расстановку скобок, то нужное количество памяти и сложность алгоритма возрастут в n раз. Чтобы этого избежать, поступим, как и в предыдущей задаче: будем запоминать только часть ответа. А именно, во-первых, запомним само наименьшее количество операций, а во-вторых, — для каждой пары (i, j) запомним еще индекс $k(i, j)$, на котором был достигнут минимум. Тогда в конце можно будет восстановить и правильную расстановку скобок для исходного произведения: находим $k_0 = k(1, n)$ — получаем, что последнее умножение было между k_0 и $k_0 + 1$. Отдельно восстанавливаем расстановку скобок для отрезка матриц от A_1 до A_{k_0}, отдельно — для отрезка от A_{k_0+1} до A_n. Далее аналогично рекурсивно восстанавливаем весь ответ.

8.3. Общие принципы

Приведем список общих принципов динамического программирования, которые можно заметить по двум только что рассмотренным задачам.

- Вместо исходной задачи решаем целый набор аналогичных подзадач.
- Переход динамического программирования: подзадачи можно решать в определенном порядке таким образом, чтобы ответ для очередной подзадачи можно было полностью восстановить по ответам для уже решенных подзадач.

- Оптимальность для подзадач: если ответ для подзадачи X зависит от ответов для других подзадач $X_1, \ldots, X_k$, то ответ для каждой из подзадач X_i должен быть оптимальным, чтобы общий ответ для X стал оптимальным.
- Большое перекрытие подзадач: если бы решались все подзадачи, однако решения не запоминались бы, а каждый раз считались с нуля с помощью рекурсивного вызова функции решения для нужных подзадач, то каждая конкретная подзадача решалась бы много раз одним и тем же способом. За счет запоминания ответов для подзадач каждая из подзадач решается не более одного раза
- Для восстановления ответа часто необходимо хранить не полностью наилучшие ответы для подзадач, а только часть наилучшего ответа, но в итоге по этим частям можно полностью восстановить наилучший ответ для всей задачи. Эта часть алгоритма называется обратным ходом динамического программирования. Та часть, в которой решаются все подзадачи исходной задачи, называется прямым ходом динамического программирования.

Итак, исходная задача разбивается на множество подзадач. Каждая подзадача определяется неким *состоянием* (в случае первой задачи это был индекс i, во второй задаче — пара (i, j)). Все состояния образуют *пространство состояний* динамического программирования S (в первой задаче это множество чисел $[1, n]$, во второй — множество подотрезков отрезка $[1, n]$).

Между некоторыми парами состояний есть *переходы* динамического программирования (в первой задаче переход между состояниями $j < i$ существовал, если $A[j] < A[i]$). Переходы образуют направленный *граф переходов* динамического программирования, множество вершин которого совпадает с S, а множество ребер определяется множеством переходов T, каждый из которых представляет собой ребро. Граф переходов не должен содержать циклов, чтобы существовал порядок перебора всех состояний (подзадач), при котором решение очередной подзадачи зависело бы только от решений уже рассмотренных подзадач.

Прямой ход динамического программирования заключается в последовательном вычислении ответов для всех подзадач в порядке, при котором ребра переходов могут быть направлены только от преды-

дущих состояний к следующим. Прямой ход может быть реализован в программе одним из трех принципиально различных способов: динамическое программирование вперед, динамическое программирование назад или рекурсивная реализация («ленивое динамическое программирование»). В любом случае хранится таблица ответов динамического программирования, ячейки которой взаимно однозначно соответствуют состояниям. Для каждого состояния в соответствующей ячейке хранится ответ для соответствующей подзадачи.

Динамическое программирование назад было использовано нами при решении первой задачи. Состояния рассматриваются по порядку, и к моменту рассмотрения состояния s уже решены все предыдущие подзадачи, из которых выходят переходы в s. Рассматриваются все такие состояния s', по таблице ответов динамического программирования находятся ответы для соответствующих подзадач, и по ним строится ответ для s.

Динамическое программирование вперед рассматривает состояния по порядку, и к моменту, когда рассматривается очередное состояние s, в таблице динамического программирования уже сохранен ответ для s. Тогда рассматриваются все переходы, *выходящие* из состояния s, и для каждого из состояний s', в которые возможен переход, делается попытка улучшить ответ для s' с помощью ответа для s.

Рекурсивная реализация соответствует нашему решению второй задачи. Вызывается функция с параметрами, определяющими основное состояние динамического программирования, которому соответствует исходная задача. Внутри функции в самом начале проверяется, не сохранен ли еще ответ в таблице ДП. Если сохранен, то это значение и возвращается, иначе все-таки считается ответ. Таким образом, каждая подзадача решается не более одного раза. Затем для тех состояний s', из которых есть переход в текущее состояние s, рекурсивно вызывается та же функция. По полученным решениям строится решение для s. Ответ записывается в таблицу и возвращается. Этот подход также называется «ленивым динамическим программированием», так как в случае, когда некоторые состояния не нужны для того, чтобы найти ответ для основного состояния, для них ответ так и не будет посчитан. Ответы вычисляются только по мере надобности. За счет этого в некоторых задачах может быть достигнуто существенное

ускорение. Иначе говоря, ответ вычисляется для тех состояний, из которых в графе переходов достижимо основное состояние, и только для них.

По сути все три подхода делают одно и то же, но в разных задачах различные подходы более удобны для реализации.

Кроме таблицы ответов могут понадобиться дополнительные таблицы, необходимые для восстановления ответа в конце. Обычно в них для каждого состояния s сохраняются ссылки на те состояния s', из которых был получен наилучший ответ в s.

При *обратном ходе* мы начинаем с основного состояния. Для текущего состояния определяем, из ответов для каких подзадач получился наилучший ответ, и переходим (рекурсивно) к восстановлению ответов для этих подзадач. После того как ответы для них восстановлены, восстанавливаем ответ для текущей подзадачи. Этот процесс может быть реализован как рекурсивно, так и итеративно, в зависимости от особенности задачи разные подходы могут быть более удобными.

Сложность решения с помощью ДП обычно оценивается следующим образом: нужно посчитать сложность $O(f)$ рассмотрения одного перехода $s' \to s$. Тогда общая сложность решения равна $O(f\,|T|)$. Чаще всего количества переходов из разных состояний оцениваются сверху одним и тем же числом k, и тогда можно оценить $|T|$ как $O(k\,|S|)$. Памяти обычно уходит $O(|S|)$ на некоторое количество таблиц, ячейки которых взаимно однозначно соответствуют состояниям ДП.

В первой задаче $|S| = n$, из каждого состояния выходит $O(n)$ переходов — $|T| = O(n^2)$, и сложность решения $O(n^2)$. Во второй задаче $|S| = O(n^2)$, переходов из каждого $O(n)$ — $|T| = O(n^3)$, и общая сложность $O(n^3)$.

8.4. Сегментация запросов

Рассмотрим такую задачу: на вход поисковой системы поступила строка на некотором известном языке, например `new york times square`, и необходимо разделить эту строку на словосочетания, из которых она состоит. Возможны различные разбиения на словосочетания, но необходимо определить то, которое наиболее вероятно имел в виду пользователь.

В данном случае есть много вариантов словосочетаний, которые теоретически могли присутствовать в строке: слово `new`(новый), город `New York`, город `York`, газета `New York Times`, площадь в городе `square` и центральная площадь Нью-Йорка `Times Square`. Скорее всего, здесь пользователь имел в виду именно площадь `Times Square` в Нью-Йорке, так как остальные варианты плохо сочетаются вместе.

Однако как же может машина определить это? Обычно считается, что у нас уже есть предварительно рассчитанная *модель языка*, упрощенно состоящая из таблицы, в которой для всех существующих слов и словосочетаний длины не более K приводится вероятность того, что они встретятся в тексте. Например,

$$\begin{aligned} P(\texttt{new}) &= 0{,}3, \\ P(\texttt{new york}) &= 0{,}1, \\ P(\texttt{new york times}) &= 0{,}02, \\ P(\texttt{times square}) &= 0{,}05, \\ P(\texttt{square}) &= 0{,}09, \\ P(\texttt{york}) &= 0{,}05. \end{aligned}$$

Если в таблице нет ключа, соответствующего какому-либо словосочетанию, то вероятность этого словосочетания считается нулевой. Модель расчета предлагается следующей: пусть мы разделили исходный запрос q на словосочетания $w_1, w_2, \ldots, w_n$. Тогда вероятность такого разделения определяется как $P(w_1)P(w_2)\ldots P(w_n)$, и мы выбираем сегментацию, максимизирующую эту вероятность.

Пусть запрос q состоит из последовательности слов $s_1, s_2, \ldots, s_m$. Для каждой позиции i от 1 до m решим задачу о сегментации строки $s_1, s_2, \ldots, s_i$ — запомним, во-первых, ответ (наибольшую возможную вероятность $P(w_1)P(w_2)\ldots P(w_k)$ разбиения этой строки на словосочетания), а во-вторых, количество слов в последнем использованном словосочетании w_k.

Таким образом, мы определили набор из m подзадач. Переход между подзадачами i и j есть, если $j - i \leqslant K$. Так как нам нужно разбить всю строку $s_1 s_2 \ldots s_i$ на словосочетания, в частности, последнее слово s_i входит в некоторое словосочетание. Переберем количество слов в этом словосочетании от 1 до $\min(i, K)$. Пусть мы зафикси-

ровали его длину l. Тогда замечаем, что в этой задаче действует принцип оптимальности для подзадач, ответ для подзадачи i строится из ответа для подзадачи $i-l$ и получается вероятность

$$ans[i] = ans[i-l] \cdot P(s_{i-l+1}s_{i-l+2}\dots s_i).$$

Теперь из всех l выбираем то, для которого получается максимальный ответ $ans[i]$, а в $prev[i]$ записываем $i-l$. Таким образом, получаем ответ для i-й подзадачи исходя из ответов для предыдущих подзадач. В обратном ходе динамического программирования мы посмотрим на значение $prev[m]$ для $i=m$, поймем, что последнее словосочетание в ответе состоит из слов

$$s_{prev[m]+1}, \quad s_{prev[m]+2}, \quad \dots, \quad s_m,$$

и перейдем к восстановлению ответа на начальном участке $s_1, s_2, \dots$ $\dots, s_{prev[m]}$, т. е. к подзадаче номер $prev[m]$, и т. д.

В нашем примере получится следующее.

- Для пустой подзадачи $ans[0]=1$, $prev[0]$ не определен.
- Для задачи из одного слова `new`:

$$\begin{aligned} ans[1] &= ans[0] \cdot P(\texttt{new}) = 1 \cdot 0{,}3 = 0{,}3, \\ prev[1] &= 0. \end{aligned}$$

- Для задачи из двух слов `new york`:

$$\begin{aligned} ans[2] &= \max(ans[0] \cdot P(\texttt{new york}), ans[1] \cdot P(\texttt{york})) = \\ &= \max(1 \cdot 0{,}1; 0{,}3 \cdot 0{,}05) = 0{,}1, \\ prev[2] &= 0. \end{aligned}$$

- Для задачи из трех слов `new york times`:

$$\begin{aligned} ans[3] &= \max(ans[0] \cdot P(\texttt{new york times}), \\ &\qquad ans[1] \cdot P(\texttt{york times}), ans[2] \cdot P(\texttt{times})) = \\ &= \max(1 \cdot 0{,}02; 0{,}3 \cdot 0; 0{,}1 \cdot 0) = 0{,}02, \\ prev[3] &= 0. \end{aligned}$$

- Для задачи из четырех слов `new york times square`:

$$\begin{aligned} ans[4] = \max(&ans[0] \cdot P(\texttt{new york times square});\\ &ans[1] \cdot P(\texttt{york times square});\\ &ans[2] \cdot P(\texttt{times square});\\ &ans[3] \cdot P(\texttt{square})) = \\ = \max(&1 \cdot 0;\\ &0{,}3 \cdot 0;\\ &0{,}1 \cdot 0{,}05;\\ &0{,}02 \cdot 0{,}09) = 0{,}005, \\ prev[4] = 2.& \end{aligned}$$

Таким образом, наилучшая сегментация состоит в разбиении исходной строки на два словосочетания: `new york` и `times square`.

Сложность алгоритма составляет $O(|T|) = O(|S| \cdot K) = O(mK)$, где S — множество состояний динамического программирования, T — множество переходов, K — ограничение на длину словосочетания. Если бы ограничения на длину словосочетания не было, то сложность составила бы $O(|T|) = O(m^2)$. Затраты по памяти равны $O(|S|) = O(m)$.

Список литературы

1. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. М.: Вильямс, 2007.
2. *Грэхем Р.* Конкретная математика. Математические основы информатики / Р. Грэхем, Д. Кнут, О. Паташник. М.: Вильямс, 2010.
3. *Кнут Д.* Искусство программирования. Том 3. Сортировка и поиск / Д. Кнут. М.: Вильямс, 2008.
4. *Tarjan R.* Data structures and network algorithms / R. Tarjan. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1983.

Магазин «Математическая книга»

Книги издательства МЦНМО можно приобрести в магазине «Математическая книга» в Москве по адресу: Б. Власьевский пер., д. 11; тел. (499) 241-72-85; biblio.mccme.ru

Книга — почтой: http://biblio.mccme.ru/shop/order

Книги в электронном виде: http://www.litres.ru/mcnmo/

Мы сотрудничаем с интернет-магазинами

- Книготорговая компания «Абрис»; тел. (495) 229-67-59, (812) 327-04-50; www.umlit.ru, www.textbook.ru, абрис.рф
- Интернет-магазин «Книга.ру»; тел. (495) 744-09-09; www.kniga.ru

Наши партнеры в Москве и Подмосковье

- Московский Дом Книги и его филиалы (работает интернет-магазин); тел. (495) 789-35-91; www.mdk-arbat.ru
- Магазин «Молодая Гвардия» (работает интернет-магазин): ул. Б. Полянка, д. 28; тел. (499) 238-50-01, (495) 780-33-70; www.bookmg.ru
- Магазин «Библио-Глобус» (работает интернет-магазин): ул. Мясницкая, д. 6/3, стр. 1; тел. (495) 781-19-00; www.biblio-globus.ru
- Спорткомплекс «Олимпийский», 5-й этаж, точка 62; тел. (903) 970-34-46
- Сеть киосков «Аргумент» в МГУ; тел. (495) 939-21-76, (495) 939-22-06; www.arg.ru
- Сеть магазинов «Мир школьника» (работает интернет-магазин); тел. (495) 715-31-36, (495) 715-59-63, (499) 182-67-07, (499) 179-57-17; www.uchebnik.com
- Сеть магазинов «Шаг к пятерке»; тел. (495) 728-33-09, (495) 346-00-10; www.shkolkniga.ru
- Издательская группа URSS, Нахимовский проспект, д. 56, Выставочный зал «Науку — Всем», тел. (499) 724-25-45, www.urss.ru
- Книжный магазин издательского дома «Интеллект» в г. Долгопрудный: МФТИ (новый корпус); тел. (495) 408-73-55

Наши партнеры в Санкт-Петербурге

- Санкт-Петербургский Дом книги: Невский пр-т, д. 62; тел. (812) 314-58-88
- Магазин кМир науки и медициныь: Литейный пр-т, д. 64; тел. (812) 273-50-12
- Магазин «Новая техническая книга»: Измайловский пр-т, д. 29; тел. (812) 251-41-10
- Информационно-книготорговый центр «Академическая литература»: Васильевский остров, Менделеевская линия, д. 5
- Киоск в здании физического факультета СПбГУ в Петергофе; тел. (812) 328-96-91, (812) 329-24-70, (812) 329-24-71
- Издательство «Петроглиф»: Фарфоровская, 18, к. 1; тел. (812) 560-05-98, (812) 943-80-76; k_i_@bk.ru, k_i_@petroglyph.ru
- Сеть магазинов «Учебная литература»; тел. (812) 746-82-42, тел. (812) 764-94-88, тел. (812) 235-73-88 (доб. 223)

Наши партнеры в Челябинске

- Магазин «Библио-Глобус», ул. Молдавская, д. 16, www.biblio-globus.ru

Наши партнеры в Украине

- Александр Елисаветский. Рассылка книг наложенным платежом по Украине: тел. 067-136-37-35; df-al-el@bk.ru