БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СБОРНИК ЗАДАЧ ПО ТЕОРИИ АЛГОРИТМОВ. СТРУКТУРЫ ДАННЫХ

Рекомендовано
Учебно-методическим объединением
по естественно-научному образованию
в качестве учебно-методического пособия для студентов
учреждений высшего образования, обучающихся по специальностям
1-31 03 03 «Прикладная математика (по направлениям)»,
1-31 03 04 «Информатика», 1-31 03 05 «Актуарная математика»,
1-31 03 06 «Экономическая кибернетика (по направлениям)»,
направление специальности 1-31 03 06-01 «Экономическая кибернетика
(математические методы и компьютерное моделирование в экономике)»,
1-31 03 07 «Прикладная информатика (по направлениям)»,
направление специальности 1-31 03 07-01 «Прикладная информатика
(программное обеспечение компьютерных систем)»,
1-31 03 07-02 «Прикладная информатика (информационные
технологии телекоммуникационных систем)»,
1-98 01 01 «Компьютерная безопасность (по направлениям)»,
направление специальности 1-98 01 01-01 «Компьютерная
безопасность (математические методы и программные системы)»

МИНСК
БГУ
2020

УДК 510.51(075.8)
ББК 22.12я73-1
С23

А в т о р ы :
С. А. Соболь, К. Ю. Вильчевский, В. М. Котов, Е. П. Соболевская

Р е ц е н з е н т ы :
кафедра информатики и методики преподавания информатики физико-математического факультета Белорусского государственного педагогического университета им. Максима Танка (заведующий кафедрой кандидат педагогических наук, доцент *С. В. Вабищевич*);
профессор кафедры информационных технологий в культуре Белорусского государственного университета культуры и искусства кандидат физико-математических наук, доцент *П. В. Гляков*

Сборник задач по теории алгоритмов. Структуры данных : С23 учеб.-метод. пособие / С. А. Соболь [и др.]. — Минск : БГУ, 2020. — 159 с.

ISBN 978-985-566-842-9.

Учебно-методическое пособие кроме теоретического материала включает задачи для самостоятельного решения по теме «Структуры данных», большинство из которых имеют творческий характер и предлагались на международных олимпиадах по программированию, а также указания к их решению.

УДК 510.51(075.8)
ББК 22.12я73-1

ISBN 978-985-566-842-9

ПРЕДИСЛОВИЕ

Учебно-методическое пособие посвящено одному из разделов, изучаемых в рамках дисциплин по теории алгоритмов и программированию, — «Структуры данных». При разработке алгоритма решения задачи выбор структур данных является важным звеном, от него напрямую зависит эффективность разрабатываемого алгоритма.

В книге приводится теоретический материал, описываются основные структуры данных, используемые при решении практических задач. Для каждой структуры данных указаны способы её представления в памяти компьютера, базовые операции и их время работы.

Для закрепления навыка выбора структур данных, позволяющих решить задачу за указанное время с учётом заданных ограничений, приводится список задач для самостоятельного решения. Даются указания к решению задач.

Для проверки работоспособности программ на факультете прикладной математики и информатики (ФПМИ) БГУ применяется образовательная платформа iRunner, которая доступна в интернете по адресу `https://acm.bsu.by`. Все представленные задачи размещены в iRunner и доступны для тестирования. Один из авторов учебно-методического пособия С. А. Соболь является разработчиком и администратором iRunner. Более чем 15-летний опыт использования образовательной платформы в вузе позволяет уверенно утверждать о её высокой эффективности на практике. Организовано дистанционное обучение, самостоятельная и контролируемая работа учащихся. Повысилась заинтересованность в практических занятиях как со стороны студентов, так и со стороны преподавателей. Возможность круглосуточной самостоятельной работы (из дома, общежития) способствует получению высоких результатов в будущей профессиональной деятельности, а

также на различных интеллектуальных соревнованиях в области алгоритмизации и спортивного программирования. Студенты ФПМИ БГУ ежегодно представляют нашу страну на международных соревнованиях ICPC и завоёвывают дипломы разной степени. Всё это, несомненно, способствует повышению позиции БГУ в мировых рейтингах [6; 7].

Авторы учебно-методического пособия имеют многолетний опыт преподавания в вузах Республики Беларусь. Профессор В. М. Котов и доцент Е. П. Соболевская — лауреаты премии имени А. Н. Севченко в номинации «Образование» за цикл пособий по дискретной математике, проектированию и анализу алгоритмов. Магистры математики и информационных технологий С. А. Соболь и К. Ю. Вильчевский имеют большой опыт участия в престижных международных соревнованиях по программированию, лауреаты специального фонда Президента Республики Беларусь по социальной поддержке одарённых учащихся и студентов.

Издание будет интересно всем, кто стремится углубить свои знания в области алгоритмизации.

Часть 1

ПРОСТЕЙШИЕ СТРУКТУРЫ ДАННЫХ

Структура данных представляет собой набор некоторым образом сгруппированных данных. Для каждой структуры определяется, каким образом данные хранятся в памяти компьютера, какие базовые операции можно выполнять над этими данными и за какое время.

Для того чтобы разработать эффективный алгоритм решения поставленной задачи, нередко необходимо проанализировать несколько различных структур данных и выбрать наиболее подходящую. Может оказаться, что каждая из структур данных позволяет нам выполнять одну из операций легко, а другие — с большим трудом, поэтому часто выбирают ту структуру, которая лучше всего подходит для решения всей задачи в целом (не делает максимально лёгким выполнение ни одной операции, но позволяет выполнить всю работу лучше, чем при любом очевидном подходе). Поэтому разработка эффективного алгоритма напрямую связана с выбором хорошей структуры данных.

В данном разделе рассмотрим такие простейшие структуры данных, как массив фиксированного размера, динамический массив и связный список [3].

В дальнейшем, если не оговорено иное, то будем предполагать, что в рассматриваемой структуре данных хранится n элементов.

1.1. МАССИВ

Наиболее известной и распространённой структурой данных является массив.

Массив (англ. *array*) — это структура данных с *произвольным доступом* (англ. *random access*) к элементу, т. е. доступ к любому элементу

по индексу осуществляется за время O(1) вне зависимости от того, где в массиве располагается элемент (в отличие от *последовательного доступа*, когда время доступа к элементу зависит от места его расположения в структуре).

Эта структура однородна, так как все компоненты имеют один и тот же тип. Под массив в памяти компьютера выделяется непрерывный блок памяти. Элементы массива в памяти располагаются один за другим и являются равнодоступными. *Индексами* являются последовательные целые числа (рис. 1.1). Математическим аналогом массива является вектор или матрица:

$$A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix}.$$

При написании программ на псевдокоде будем использовать для i-го элемента массива A обозначение `a[i]`, в формулах будем писать a_i.

Поддержка массивов (свой синтаксис объявления, функции для работы с элементами и т. д.) есть в большинстве высокоуровневых языков программирования.

Как правило, нумерация элементов начинается с нуля (0-индексация). Это удобно, так как адрес расположения i-го элемента в памяти можно быстро вычислить, прибавив к адресу начала массива размер одного элемента, умноженный на i.

Рис. 1.1. Устройство массива

Кроме одномерных (линейных) массивов, нередко в задачах удобно использовать многомерные массивы, в которых обращение к элементу идёт по нескольким индексам. Доступ к произвольному элементу такого массива тоже выполняется за константное время.

В массивах трудно осуществлять поиск элемента (для этого в общем случае необходим проход по всему массиву), трудно производить включение или исключение отдельных элементов (необходимо выполнять сдвиг других элементов вправо или влево).

1.2. ДИНАМИЧЕСКИЙ МАССИВ

Размер массива в простейшем случае фиксирован и должен быть известен заранее. Однако на практике часто удобно использовать динамический массив, который можно расширять по мере надобности.

Под *динамическим массивом* (англ. *dynamic array*) понимается структура данных, которая обеспечивает произвольный доступ и позволяет добавлять или удалять элементы.

Рассмотрим классический сценарий использования динамического массива: пусть изначально массив пуст, затем в него последовательно добавляют n элементов, при этом каждый раз новый элемент добавляется в конец. Как можно организовать динамический массив на базе статического?

1.2.1. Наивная реализация

Каждый раз при необходимости изменения размера будем делать *реаллокацию* (англ. *reallocation*), т. е. выделять новый массив и перемещать все элементы из старого массива в новый.

Подсчитаем общее число «лишних» операций по перемещению данных. Так, для добавления i-го по счёту ($1 \leqslant i \leqslant n$) элемента потребуется перенести все $i - 1$ элементов на новое место и записать в конец один новый элемент. Всего на выполнение n перекладываний уйдёт

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

операций. Это слишком медленно для использования на практике.

Попробуем уменьшить число реаллокаций. Очевидная идея — расширять массив «с запасом», оставляя пустые ячейки, которые можно будет использовать на следующих шагах. Число реально занятых ячеек памяти будем называть *логическим размером* (*size*) динамического массива. Общее число зарезервированных ячеек будем называть *ёмкостью* (*capacity*) (рис. 1.2).

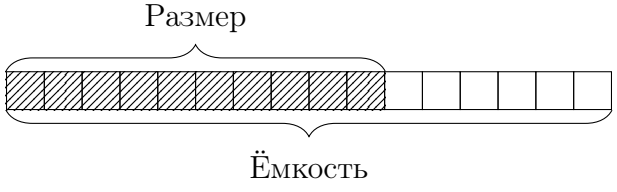


Рис. 1.2. Размер и ёмкость

1.2.2. Линейный рост

Будем каждый раз расширять массив не на один элемент, а сразу на Δ элементов ($\Delta \geqslant 1$). Последовательность изменения ёмкости будет иметь вид 0, Δ, 2Δ, 3Δ, ...

Так, при добавлении первого элемента сразу будет выделен массив ёмкости Δ, в его начало будет сохранён первый элемент, а остальные $\Delta - 1$ ячеек пока останутся пустыми.

Следующие $\Delta - 1$ добавлений будут выполнены легко и быстро, так как перевыделение памяти не требуется. На каждом шаге будет увеличиваться логический размер, а ёмкость изменяться не будет.

Как только поступит значение под номером $\Delta + 1$, потребуется создать новый массив ёмкости 2Δ и перенести все данные в него, затем уже сохранить новый элемент (рис. 1.3).

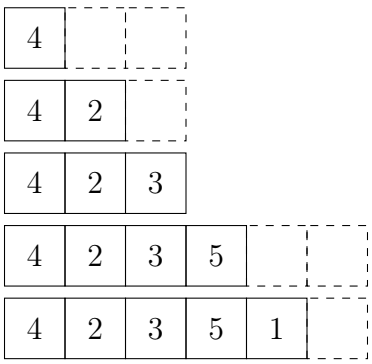


Рис. 1.3. Этапы линейного расширения массива при $\Delta = 3$

Нетрудно заметить, что в ходе добавления n элементов реаллокация выполняется $k = \lceil n/\Delta \rceil$ раз, каждый раз перемещается 0, Δ, 2Δ, ... значений, поэтому общее число «лишних» операций вычисляется по формуле

$$T(n) = \sum_{i=1}^{k} (i-1)\Delta = \Delta \cdot \sum_{i=0}^{k-1} i = \Delta \cdot \frac{k(k-1)}{2}.$$

Пользуясь свойством $\lceil x \rceil \geqslant x$ функции «потолок», получим оценку снизу:

$$T(n) \geqslant \Delta \cdot \frac{1}{2} \cdot \frac{n}{\Delta} \cdot \left(\frac{n}{\Delta} - 1 \right) \geqslant \frac{n^2}{2\Delta} = \Omega(n^2). \tag{1.1}$$

Получается, что алгоритм по-прежнему квадратичный. Нетрудно построить аналогичным образом оценку сверху и доказать, что в формуле (1.1) букву Ω можно смело заменить на Θ.

Казалось бы, выбрав в качестве Δ число порядка n (скажем, $n/10$), можно добиться линейного времени работы. Но так рассуждать нельзя: число n заранее неизвестно, а константа Δ должна быть зафиксирована и не может зависеть от входных данных.

Практическое преимущество подхода «рост с шагом Δ» по сравнению с простейшим — он работает быстрее в константное число (в Δ) раз. Недостаток — перерасход памяти: в каждый момент времени некоторое количество (до $\Delta - 1$) ячеек в массиве реально не используется.

1.2.3. Экспоненциальный рост

Следующей идеей будет использовать другую стратегию изменения ёмкости: не «расширяем на Δ единиц», а «расширяем в α раз» ($\alpha > 1$) (рис. 1.4). Если начать с массива ёмкости 1, то получим последовательность ёмкостей 1, $\lfloor\alpha\rfloor$, $\lfloor\alpha^2\rfloor$, $\lfloor\alpha^3\rfloor$, ... Отметим, что в общем случае число α может быть дробным (например, можно увеличивать размер на каждом шаге в полтора раза). Ёмкость массива — всегда целое число, поэтому в формулах появляется операция округления, которая их заметно усложняет.

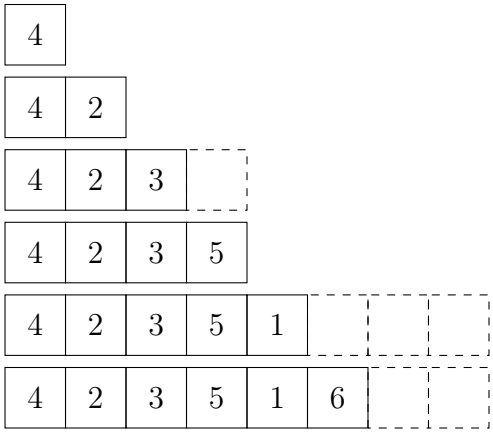


Рис. 1.4. Этапы экспоненциального расширения массива при $\alpha = 2$

Определим число k — количество шагов увеличения ёмкости массива, необходимое для сохранения n элементов. Нетрудно увидеть, что раз выполнить $k - 1$ шаг ещё недостаточно, а k шагов — достаточно, то

$$\lfloor\alpha^{k-1}\rfloor < n \leqslant \lfloor\alpha^k\rfloor.$$

Рассмотрим левую часть неравенства. Исходя из свойства $x - 1 < \lfloor x\rfloor$ операции «пол», имеем

$$\alpha^{k-1} - 1 < \lfloor\alpha^{k-1}\rfloor < n,$$

отсюда получим оценку:

$$\alpha^{k-1} < n + 1.$$

Всего при выполнении этих k реаллокаций нужно будет выполнить

$$T(n) = \sum_{i=1}^{k} \lfloor \alpha^{i-1} \rfloor$$

элементарных операций переноса значений из старого массива в новый. Оценивая слагаемые сверху и пользуясь формулой суммы геометрической прогрессии, имеем:

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

или

$$T(n) = \mathrm{O}(n).$$

Таким образом, можно сделать вывод, что при фиксированной константе $\alpha > 1$ общее число операций по перемещению данных в памяти, которые выполняются при последовательном добавлении n элементов, растёт линейно с ростом n.

Пример 1.1. Часто на практике используется значение $\alpha = 2$. Это значит, что ёмкость динамического массива изменяется следующим образом: 1, 2, 4, 8, 16, ...

1.2.4. Амортизированная константа

Итак, если следовать стратегии удвоения размера, на добавление в динамический массив n элементов требуется затратить время $\mathrm{O}(n)$. Значит, каждая вставка выполняется *в среднем* за время $\mathrm{O}(1)$.

На самом деле конкретная операция вставки каждого элемента осуществляется или за константное время (когда в массиве есть свободная ёмкость), или за линейное (когда свободного места нет, выполняется реаллокация). Но усреднённо время вставки одного элемента получается константным. В этом случае говорят, что $\mathrm{O}(1)$ — *амортизированная* оценка для операции вставки.

На практике в системах реального времени такие непредсказуемые задержки при выполнении отдельной операции могут представлять проблему. Если известно

примерное число элементов, которые будут в итоге добавлены в массив, можно заранее *зарезервировать* (англ. *reserve*) нужную ёмкость массива и уменьшить число реаллокаций.

1.2.5. Пример реализации

Опишем на псевдокоде класс, который организует динамический массив на основе статического с использованием стратегии удвоения.

```
class DynamicArray:
    def __init__(self):
        self.data = array(1)
        self.size = 0
        self.capacity = 1

    def append(self, x):
        if self.size == self.capacity:
            new_capacity = self.capacity * 2
            new_data = array(new_capacity)

            for i in range(self.capacity):
                new_data[i] = self.data[i]

            self.data = new_data
            self.capacity = new_capacity

        self.data[self.size] = x
        self.size += 1
```

С целью оптимизации память для пустого динамического массива (нулевого размера) можно не выделять, но потребуется этот случай рассмотреть отдельно.

При необходимости можно описать и другие операции: удаление последнего элемента, установка размера массива в фиксированное значение, предварительное резервирование заданной ёмкости и др.

1.2.6. Применение на практике

Динамические массивы очень удобны и широко используются на практике в прикладных задачах. С точки зрения скорости доступа к элементам они эквивалентны статическим массивам. Готовые реализации динамических массивов предоставляются стандартными библиотеками всех основных современных языков программирования.

В языке C++ динамический массив реализован в классе `std::vector`. Значение множителя роста не зафиксировано стандартом языка и различается в зависимости

от конкретной реализации (так, в libc++ традиционно применяют число 2, в версии от Microsoft — число 1,5).

При создании программ на Java широко используется класс `ArrayList` (множитель роста 1,5).

В языке программирования Python применяется тип `list`. Если обратиться к реализации CPython 3.7, то можно увидеть, что при расширении действует оригинальная стратегия: старый размер умножается на 1,125, затем к нему прибавляется константа 3 или 6. Получается такая последовательность ёмкостей: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...

1.3. СВЯЗНЫЙ СПИСОК

Связный список (англ. *linked list*) — некоторая последовательность элементов, которые связаны друг с другом логически. Логический порядок прохождения элементов определяется с помощью ссылок, при этом он может не совпадать с физическим порядком размещения элементов в памяти компьютера. Доступ к элементам списка осуществляется *последовательно*, т. е. чем дальше в структуре расположен элемент, тем дольше к нему по времени будет осуществляться доступ.

Список состоит из *узлов* (англ. *nodes*). Каждый узел включает две части: информационную (непосредственные данные, принадлежащие элементу) и ссылочную (указатель/ссылка на следующий и/или предыдущий узел).

В *односвязном*, или *однонаправленном связном*, списке (англ. *singly linked list*) каждый узел содержит ссылку на следующий узел (рис. 1.5). Для последнего узла эта ссылка обычно является нулевой. По односвязному списку можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

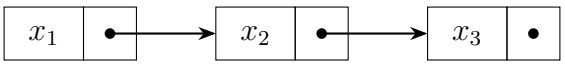


Рис. 1.5. Односвязный список

В *двусвязном*, или *двунаправленном связном*, списке (англ. *doubly linked list*) ссылки в каждом узле указывают на предыдущий и на последующий узел (рис. 1.6). Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом даёт возможность перемещения в обе стороны. В таком списке проще производить удаление и перестановку элементов, так как легко получить

доступ ко всем элементам списка, ссылки которых направлены на изменяемый элемент.

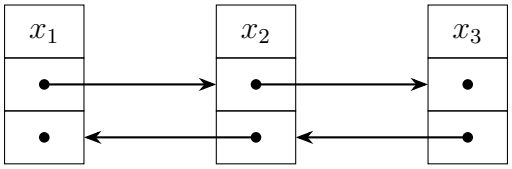


Рис. 1.6. Двусвязный список

При работе со списком вводятся дополнительные ссылки на первый и последний элемент списка. Будем называть их `head` («голова») и `tail` («хвост»).

1.3.1. Введение буферного элемента

Часто для списка вводят *буферный элемент* (англ. *sentinel*), т. е. такой вспомогательный узел, на который ссылается логически последний узел списка. На рис. 1.7 показан пример такого списка.

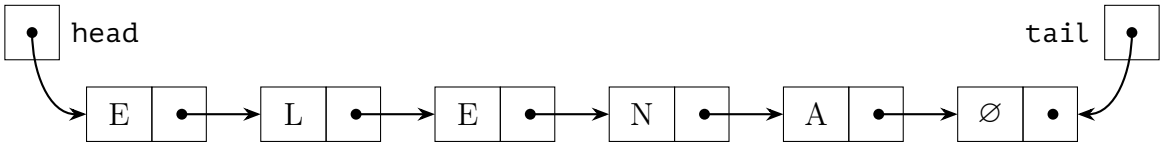


Рис. 1.7. Односвязный список с буферным элементом

Этот приём даёт возможность упростить реализацию структуры данных. Так, теперь ссылки `head` и `tail` всегда ненулевые, для пустого списка они ссылаются на буферный элемент (рис. 1.8).

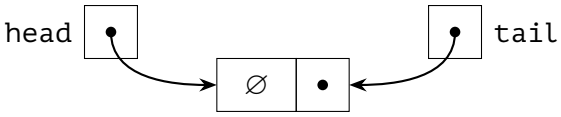


Рис. 1.8. Пустой односвязный список с буферным элементом

Для однонаправленного списка буферный элемент позволяет выполнять операцию удаления элемента за константное время, если удаляемый элемент не нужно искать, а сразу задана ссылка на него. Случай, когда удаляемый элемент стоит в списке последним, проиллюстрирован на рис. 1.9. При удалении узла мы берём узел, следующий за ним, и переносим всю информацию (данные и ссылку) в текущий узел, затем

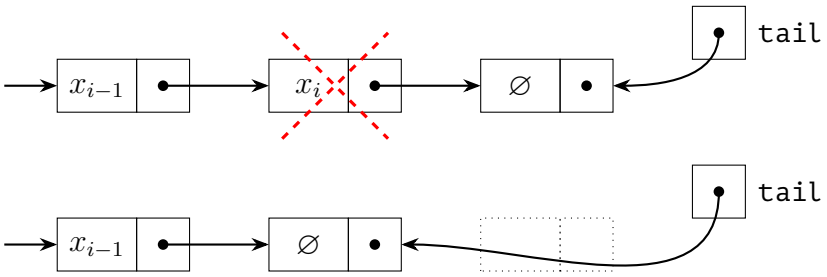


Рис. 1.9. Удаление элемента x_i из односвязного списка с буферным элементом

следующий узел удаляем. После этого при необходимости обновляем ссылку `tail`, которая всегда указывает на буферный элемент.

Если бы буферного элемента не было, то в аналогичной ситуации в ходе удаления требовалось бы модифицировать узел x_{i-1} (установить нулевую ссылку). При наличии только ссылки на узел x_i нет возможности эффективно перейти к предыдущему узлу.

1.3.2. Реализация на массивах

Чаще всего узлы списка размещают в динамической памяти, при этом в качестве значений ссылок используются адреса узлов. Альтернативный способ — использовать для хранения информации обычные массивы, тогда в качестве значений ссылок будут выступать индексы (порядковые номера элементов массива). На рис. 1.10 приведём пример представления списка, представленного на рис. 1.7 (без буферного элемента), в виде двух массивов.

i	0	1	2	3	4
list[i]	A	E	L	N	E
next[i]	−1	2	4	0	3

Рис. 1.10. Представление связного списка на базе двух массивов

В данном представлении `list[i]` содержит значение элемента списка (латинскую букву), а `next[i]` определяет позицию (индекс) следующего за ним элемента. Индекс первого элемента списка `head = 1`, а индекс последнего элемента списка — `tail = 0`. Иногда для представления списка вместо двух отдельных массивов используют один массив, элементами которого являются структуры, состоящие из полей.

1.3.3. Сравнение связных списков и динамических массивов

Связные списки имеют несколько преимуществ перед динамическими массивами.

• **Быстрая вставка и удаление.** Операции вставки в конкретное место списка и удаления определённого элемента списка выполняются за O(1) при условии, что на вход даётся ссылка на узел (идущий перед точкой вставки или предшествующий узлу, который будет удалён). Заметим, что если такая ссылка не предоставлена, то операции работают за O(n). В то же время вставка в произвольное место динамического массива требует перемещения в среднем половины элементов, а в худшем случае — всех элементов. Хотя можно «удалить» элемент из массива за константное время, пометив его ячейку как «свободную», это вызовет фрагментацию, которая будет негативно влиять на скорость прохода по массиву.

• **Нет реаллокаций.** В связный список может быть вставлено произвольное количество элементов, ограниченное только доступной памятью. Ранее вставленные элементы никуда не перемещаются, их адреса в памяти не меняются. В динамических массивах при вставке иногда происходит реаллокация; это дорогостоящая операция, которая может оказаться невозможной при высокой фрагментированности памяти (не удастся найти непрерывный блок памяти нужного размера, хотя небольшие свободные блоки будут доступны в достаточном количестве).

С другой стороны, у списков есть и существенные недостатки.

• **Нет произвольного доступа.** Динамические массивы обеспечивают произвольный доступ к любому элементу по индексу за константное время, в то время как связные списки допускают лишь последовательный доступ к элементам. По односвязному списку можно пройти только в одном направлении. Это делает связные списки непригодными для алгоритмов, в которых нужно быстро получать элемент по его индексу (например, к такому типу относятся многие алгоритмы сортировки).

• **Медленный последовательный доступ.** Линейный проход по элементам массива на реальных машинах выполняется гораздо быстрее, чем по элементам связного списка. Это связано с тем, что элементы массива хранятся в памяти один за одним, поэтому не требуется выполнять на каждом шаге переход по указателю. За счёт локальности хранения данных эффективно работает кеширование на уровне процессора.

• **Перерасход памяти.** На хранение ссылок в узлах связного списка расходуется дополнительная память. Эта проблема особенно актуальна, если полезные данные имеют небольшой размер. Накладные расходы на хранение ссылок могут превышать размер данных в восемь или более раз.

1.3.4. Применение на практике

В реальной практике прикладного программирования связные списки в чистом виде используются крайне редко. Динамические массивы обычно оказываются удобнее и эффективнее.

Так, если в задаче требуется хранить граф в виде списков смежности (для каждой вершины храним набор вершин, смежных с ней), эти абстрактные «списки смежности» не обязательно размещать именно в связных списках. С тем же успехом можно применить динамические массивы.

Однако есть ряд алгоритмов, при разработке которых не обойтись без классических связных списков (например, к ним относятся многие механизмы кеширования). Связные списки находят применение в системном программировании: в ядре операционной системы в связных списках хранятся активные процессы, потоки и другие динамические объекты, в менеджерах памяти (аллокаторах) в связных списках хранятся готовые к использованию блоки свободной памяти и т. д.

Двусвязный список представлен в стандартной библиотеке языка C++ классом `std::list`, в библиотеке языка Java — классом `LinkedList`. В языке Python встроенной реализации нет.

Часть 2

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

Абстрактный тип данных (англ. *abstract data type*) — это математическая модель, где тип данных характеризуется поведением (семантикой) с точки зрения пользователя данных. Для абстрактного типа определяется *интерфейс* — набор операций, которые могут быть выполнены. Пользователь абстрактного типа, используя эти операции, может работать с данными, не вдаваясь во внутренние детали механизма хранения информации.

Реализациями абстрактных типов данных являются конкретные структуры данных. Реализация определяет, как именно представлены в памяти данные и как функционирует та или иная операция. На практике один абстрактный тип обычно может быть реализован несколькими альтернативными способами, которые различаются по времени работы и объёму используемой памяти.

Если алгоритм работает с данными исключительно через интерфейс, то он продолжит функционировать, если одну реализацию интерфейса заменить на другую. В этом и заключается суть абстракции: реализация скрыта за интерфейсом. Такой подход позволяет создавать универсальные алгоритмы и строить программное обеспечение из отдельных модулей.

2.1. СПИСОК

Список (англ. *list*) — это абстрактный тип данных, представляющий собой набор элементов, которые следуют в определённом порядке. Список является компьютерной реализацией математического понятия конечной последовательности.

Реализация абстрактного списка может предусматривать некоторые из следующих операций:

1) создание пустого списка;

2) проверку, является ли список пустым;

3) операцию по добавлению объекта в начало или конец списка;

4) операцию по получению ссылки на первый элемент («голову») или последний элемент («хвост») списка;

5) операцию перехода от одного элемента к следующему или предыдущему элементу;

6) операцию для доступа к элементу по заданному индексу.

Абстрактный тип данных «список» обычно реализуется на практике либо как массив (чаще всего динамический), либо как связный список (односвязный или двусвязный).

В стандартной библиотеке C++ нет специального контейнера, представляющего абстрактный список. Следует использовать либо `std::vector` (динамический массив), либо `std::list` (связный список). В языке Java существует интерфейс `List`, реализациями которого являются классы `ArrayList` (динамический массив) и `LinkedList` (связный список). В программах на Python широко используется тип данных `list`, внутри являющийся динамическим массивом.

2.2. СТЕК

Иногда при работе со списковой структурой возникает потребность включать и исключать элементы в определённом порядке. Если реализуется принцип «последним пришёл — первым вышел» (англ. LIFO — last in first out), то такую структуру данных называют *стеком* (англ. *stack*).

Основные операции, которые выполняются над стеком (рис. 2.1):

1) Init() — создание пустого стека;

2) IsEmpty() — проверка стека на пустоту; возвращается значение «истина», если стек пуст, и «ложь» в противном случае;

3) Push(x) — добавление элемента x; заданный элемент добавляется *на вершину* стека;

4) Pop() — удаление элемента из стека; выполняется при условии, что стек не пуст, поэтому сначала надо убедиться в этом, а затем — извлечь с вершины стека последний занесённый в него элемент.

Существуют различные способы реализации интерфейса стека. Наиболее распространёнными являются моделирование стека на динамическом массиве и на связном списке. Если наибольшее число элементов, которые будут одновременно находиться в стеке, заранее известно, то можно использовать статический массив.

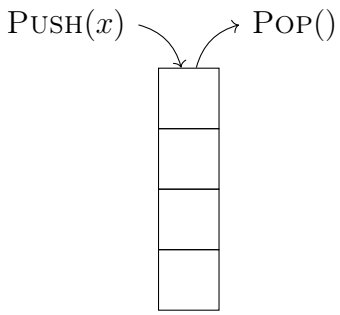


Рис. 2.1. Схема работы стека

Все основные операции над стеком выполняются за время O(1).

В стандартной библиотеке C++ доступен контейнер-адаптер `std::stack`, реализующий интерфейс стека. По умолчанию `std::stack` функционирует на базе контейнера `std::deque` (см. раздел 2.4). В языке Java существует класс `Stack`. В языке Python специального класса для создания стека нет, предлагается использовать объект типа `list` (методы `append` и `pop`).

2.3. ОЧЕРЕДЬ

Если при работе со списковой структурой реализуется принцип «первым пришёл — первым вышел» (англ. FIFO — first in first out), то такую структуру данных называют *очередью* (англ. *queue*).

Основные операции, которые выполняются над очередью (рис. 2.2):

1) Init() — создание пустой очереди;

2) IsEmpty() — проверка очереди на пустоту;

3) Enqueue(x) — добавление элемента x; заданный элемент добавляется *в конец* очереди;

4) Dequeue() — удаление элемента из очереди; элемент удаляется *из начала* очереди; операция выполняется при условии, что очередь не пуста, поэтому сначала надо убедиться в этом, а затем — извлечь элемент.

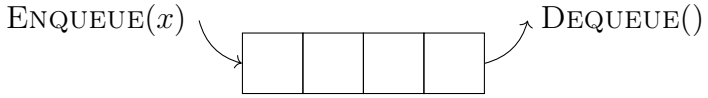


Рис. 2.2. Схема работы очереди

Наиболее простыми способами реализации интерфейса очереди являются моделирование очереди на обычном массиве и на связном списке.

Если очередь моделируется на статическом массиве, то очередь делают *кольцевой* (рис. 2.3). Кроме самого массива, достаточно хранить два индекса: индекс «головы» и индекс «хвоста». Максимальное число элементов очереди будет ограничено размером этого массива. При добавлении элемента сначала нужно проверить, есть ли в массиве свободное место для его размещения.

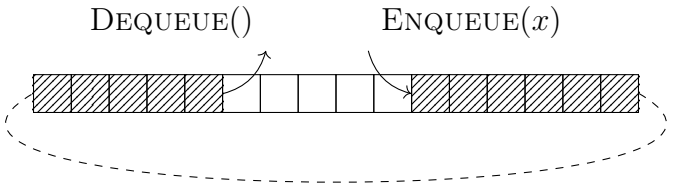


Рис. 2.3. Очередь на базе кольцевого буфера

Динамический массив не подходит для создания очереди, поскольку он, хоть и даёт возможность эффективно добавлять элементы в конец, не позволяет быстро удалять элементы из начала. Однако эту структуру данных можно доработать, предусмотрев наличие зарезервированных, но не занятых ячеек не только в конце, но и в начале блока памяти (рис. 2.4).

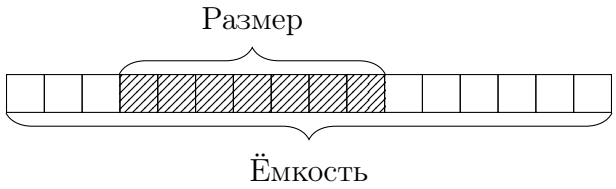


Рис. 2.4. Динамический массив, расширяющийся в обе стороны

Нетрудно понять, что для реализации интерфейса очереди можно также использовать односвязный список, элементы которого следуют в порядке от начала очереди к концу.

Все основные операции над очередью выполняются за время $O(1)$ (при условии, что выбрана подходящая реализация).

В стандартной библиотеке C++ доступен контейнер-адаптер `std::queue`, реализующий интерфейс очереди. По умолчанию он работает на основе контейнера `std::deque` (см. раздел 2.4). В языке Java существует интерфейс `Queue`. В языке Python предлагается использовать более общий контейнер типа `collections.deque`.

2.4. ДВУХСТОРОННЯЯ ОЧЕРЕДЬ

Абстрактный тип данных *двухсторонняя очередь* (англ. *double ended queue*, или *deque*) — обобщение очереди, где добавление и удаление элементов возможно с обоих концов. Таким образом, интерфейсы стека и очереди являются частным случаем интерфейса двухсторонней очереди (рис. 2.5).

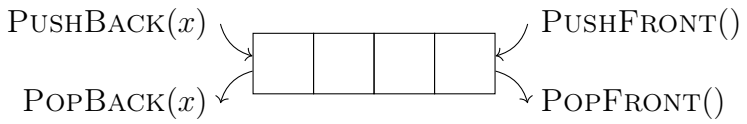


Рис. 2.5. Схема работы двухсторонней очереди

Роль двухсторонней очереди в стандартной библиотеке C++ играет контейнер `std::deque`. Отметим, что этот контейнер обеспечивает доступ к любому элементу по индексу за O(1), как и вектор, но не гарантирует, что все элементы будут лежать в памяти последовательно. В Java есть интерфейс `Deque`, и он реализуется, в частности, классами `ArrayDeque` и `LinkedList`. В языке Python в модуле `collections` предлагается контейнер `deque`.

2.5. ПРИОРИТЕТНАЯ ОЧЕРЕДЬ

Предположим, что для каждого элемента определён некоторый *приоритет*. В простейшем случае значение приоритета может совпадать со значением элемента. В общем случае соотношение элемента и приоритета может быть произвольным.

Приоритетной очередью (англ. *priority queue*) называется такая абстрактная структура данных, интерфейс которой включает в себя следующие операции:

1) PullHighestPriorityElement() — поиск и удаление элемента с самым высоким приоритетом;

2) InsertWithPriority($x, \text{prior}(x)$) — добавление элемента x с указанным приоритетом.

Интерфейс структуры данных «приоритетная очередь» может быть реализован на основе различных структур данных. Хотя приоритетные очереди часто ассоциируются с кучами, они концептуально отличаются от них. Приоритетная очередь — это абстрактное понятие. По аналогии с тем, как список может быть реализован с помощью связного списка или массива, приоритетная очередь может быть реализована с помощью кучи или другими способами.

Стек, очередь

Заметим, что стек и очередь могут рассматриваться как частный случай приоритетной очереди.

1. Предположим, что на вход алгоритма поступают заявки и самый высокий приоритет имеет та заявка, которая поступила раньше. В этом случае для моделирования процесса обслуживания заявок интерфейс приоритетной очереди можно реализовать, используя структуру данных очередь (FIFO).

2. Предположим, что на вход алгоритма поступают заявки и самый высокий приоритет имеет та заявка, которая поступила позже. В этом случае для моделирования процесса обслуживания заявок интерфейс приоритетной очереди можно реализовать, используя структуру данных стек (LIFO).

Кучи

Предположим, что на вход алгоритма поступают заявки, но теперь приоритет заявки никак не связан с моментом её поступления. В этом случае для моделирования процесса обслуживания заявок интерфейс приоритетной очереди можно реализовать, используя такую специальную структуру данных, как *куча* (англ. *heap*).

Существуют различные способы реализации структуры данных куча. В [3] подробно рассмотрены следующие реализации: с помощью полного бинарного дерева — *бинарная куча*; с помощью семейства биномиальных деревьев — *биномиальная куча*; с помощью семейства корневых деревьев — *куча Фибоначчи*. В каждой куче вершины деревьев упорядочены в соответствии со свойством неубывающей пирамиды: приоритет предка не ниже приоритета его потомков. На практике в силу своей простоты наиболее часто используется бинарная куча, поэтому на этой специальной структуре данных мы остановимся подробнее в разделе 5.1.

Две очереди

Следующий пример демонстрирует реализацию интерфейса приоритетной очереди на двух очередях (FIFO).

На практике хорошо известен алгоритм Э. Дейкстры, который находит во взвешенном графе кратчайшие маршруты между вершиной s и всеми вершинами, достижимыми из неё. Алгоритм работает только

для графов без рёбер отрицательного веса. На каждом шаге алгоритма Дейкстры в качестве текущей вершины выбирается та вершина, которая ещё не просмотрена и расстояние до которой минимально. Значит, для хранения набора вершин можно применить приоритетную очередь, при этом та вершина более приоритетна, расстояние до которой меньше, т. е. в качестве значения приоритета можно использовать минус расстояние до вершины. Подчеркнём, что использование противоположного знака связано с тем, что интерфейс приоритетной очереди выдаёт элемент с максимальным приоритетом, а для алгоритма требуется вершина с минимальным расстоянием.

В зависимости от того, как именно реализована приоритетная очередь (на базе обычного массива, на базе бинарной кучи, на базе кучи Фибоначчи), время работы алгоритма Дейкстры для взвешенного (n, m)-графа будет различным [4]. Оказывается, для графов специального вида можно применить особую реализацию и достичь оптимального времени работы.

Так, если веса рёбер графа $c_{u,v}$ принимают одно их двух значений a или b, то в алгоритме Дейкстры интерфейс приоритетной очереди можно реализовать на двух обычных очередях Q_a и Q_b, в которых хранятся пары (вершина, её приоритет). На каждом шаге будет поддерживаться такое свойство: элементы каждой очереди упорядочены по убыванию приоритетов (в начале очереди стоит самая приоритетная вершина). Алгоритм работает следующим образом.

На начальном этапе стартовую вершину s добавляем в любую из очередей (Q_a или Q_b) с приоритетом 0.

На текущей итерации алгоритма Дейкстры для выбора текущей вершины v возьмём первые элементы каждой из двух очередей и выберем из них тот, приоритет которого больше. Предположим, что это вершина u с приоритетом $-d_u$ (данная вершина среди всех вершин обеих очередей обладает самым высоким приоритетом). Значит, длина кратчайшего маршрута из стартовой вершины s до вершины u равна d_u. Удаляем вершину u из очереди, полагаем её просмотренной и выполняем релаксацию, т. е. добавляем все смежные с ней непросмотренные вершины v в одну из двух очередей (рис. 2.6):

- если $c_{u,v} = a$, то добавляем вершину v с приоритетом $-(d_u + a)$ в очередь Q_a;
- если $c_{u,v} = b$, то добавляем вершину v с приоритетом $-(d_u + b)$ в очередь Q_b.

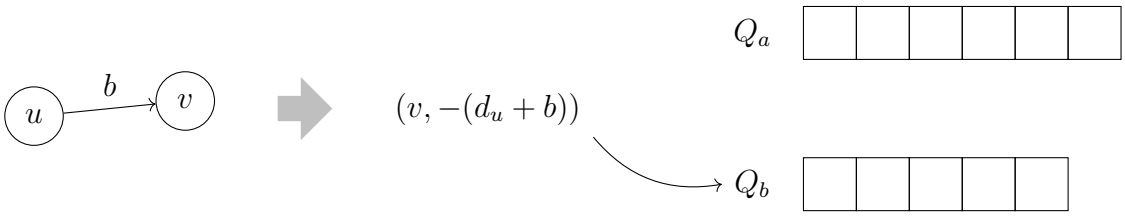


Рис. 2.6. Вершина v добавляется в очередь в зависимости от веса ребра $c_{u,v}$

Алгоритм продолжает свою работу до тех пор, пока обе очереди не станут пустыми. Приоритет, с которым вершина u удаляется из очереди, равен длине кратчайшего маршрута из стартовой вершины s в вершину u, взятой с противоположным знаком.

Так как добавление и удаление элемента из очереди выполняется за время $\mathrm{O}(1)$, а количество элементов этих очередей ограничено m, то время работы алгоритма Дейкстры для заданного графа есть $\mathrm{O}(n+m)$.

Таким образом, для взвешенных графов специального вида (в которых все веса принимают значение a или b) получен эффективный алгоритм нахождения кратчайших путей.

Двухсторонняя очередь

Если в графе веса рёбер принимают одно из двух значений 0 или 1, то в алгоритме Дейкстры интерфейс приоритетной очереди можно реализовать на структуре данных *двухсторонняя очередь* (англ. *deque — double ended queue*). Для неё добавление и удаление элементов допустимо как с начала очереди, так и с конца. Рассуждая аналогично предыдущему случаю, при релаксации по ребру веса 0 добавляем элемент в начало двухсторонней очереди, а при релаксации по ребру веса 1 — в конец. В качестве текущего элемента на итерациях алгоритма Дейкстры всегда выбирается первый элемент очереди, а алгоритм продолжает свою работу, пока двухсторонняя очередь не станет пустой. Так как добавление и удаление элемента из двухсторонней очереди выполняется за $\mathrm{O}(1)$, а количество элементов очереди ограничено m, то время работы алгоритма Дейкстры для даного класса графов равно $\mathrm{O}(n+m)$.

В стандартной библиотеке C++ имеется контейнер-адаптер под названием `std::priority_queue`, представляющий приоритетную очередь, основанную на бинарной куче. В языке Java существует класс `PriorityQueue`, также содержащий внутри бинарную кучу. В языке Python нет абстрактного интерфейса приоритетной очереди, есть лишь модуль `heapq`, в котором реализована бинарная куча.

2.6. МНОЖЕСТВО

Множество (англ. *set*) — абстрактная структура данных, которая хранит набор попарно различных объектов без определённого порядка. Эта структура данных является компьютерной реализацией математического понятия конечного множества. Подчеркнём важные отличия множества от списка:

- в списке одинаковые элементы могут храниться несколько раз, а в множестве все элементы уникальны;
- в списке порядок следования элементов сохраняется, а в множестве — нет.

Интерфейс множества включает три основные операции:

1) INSERT(x) — добавить в множество ключ x;
2) CONTAINS(x) — проверить, содержится ли в множестве ключ x;
3) REMOVE(x) — удалить ключ x из множества.

Для реализации интерфейса множества обычно используются такие структуры данных:

- сбалансированные поисковые деревья: например, красно-чёрные деревья [1], AVL-деревья [3], 2-3-деревья [3], декартовы деревья (см. раздел 5.2);
- хеш-таблицы (см. часть 3).

В стандартной библиотеке C++ есть контейнер `std::set`, который реализует множество на основе сбалансированного дерева (обычно красно-чёрного), и контейнер `std::unordered_set`, построенный на базе хеш-таблицы. В языке Java определён интерфейс `Set`, у которого есть несколько реализаций, среди которых классы `TreeSet` (работает на основе красно-чёрного дерева) и `HashSet` (на основе хеш-таблицы). В языке Python есть только встроенный тип `set`, использующий хеширование, но нет готового класса множества, построенного на сбалансированных деревьях.

2.7. АССОЦИАТИВНЫЙ МАССИВ

Ассоциативный массив (англ. *associative array*), или *отображение* (англ. *map*), или *словарь* (англ. *dictionary*), — абстрактная структура данных, которая хранит пары вида (ключ, значение), при этом каждый ключ встречается не более одного раза.

Название «ассоциативный» происходит от того, что значения ассоциируются с ключами. Такой массив удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов (например строки).

Интерфейс ассоциативного массива включает операции:

1) INSERT(k, v) — добавить пару, состоящую из ключа k и значения v;

2) FIND(k) — найти значение, ассоциированное с ключом k, или сообщить, что значения, связанного с заданным ключом, нет;

3) REMOVE(k) — удалить пару, ключ в которой равен k.

Данный интерфейс реализуется на практике теми же способами, что и интерфейс множества. Реализация ассоциативного массива технически немного сложнее, чем множества, но использует те же идеи.

Для языка программирования C++ в стандартной библиотеке доступен контейнер `std::map`, работающий на основе сбалансированного дерева (обычно красно-чёрного), и контейнер `std::unordered_map`, работающий на основе хеш-таблицы. В языке Java определён интерфейс `Map`, который реализуется несколькими классами, в частности классом `TreeMap` (базируется на красно-чёрном дереве) и `HashMap` (базируется на хеш-таблице). В языке Python широко используется встроенный тип `dict`. Этот словарь использует внутри хеширование.

Часть 3

ХЕШИРОВАНИЕ

Потребность в структуре данных, реализующей интерфейс множества (см. раздел 2.6), возникает при решении многих задач.

Естественным способом реализации такого интерфейса является дерево поиска. Если использовать сбалансированные бинарные поисковые деревья (например AVL-деревья или красно-чёрные деревья), то для множества из n элементов время выполнения каждой из указанных операций будет величиной $O(\log n)$.

Другим способом эффективной реализации интерфейса множества является хеш-таблица. На практике множества, построенные на хеш-таблицах, обычно работают быстрее, чем множества на основе деревьев, и все основные операции выполняются за $O(1)$. Но для конкретных реализаций на конкретных входных данных ситуация может быть прямо противоположной. Время выполнения основных операций с хеш-таблицей может варьироваться от $O(1)$ до $O(n)$.

В этой части книги рассмотрен принцип построения хеш-таблиц, а также затронуты теоретические аспекты, под ними лежащие. Умения строить хеш-таблицы на практике часто недостаточно для того, чтобы понимать, какие гарантии на время выполнения операций даёт такая структура данных. Оказывается, хеш-таблицы могут быть быстрее деревьев в некотором доказуемом смысле при выполнении ряда условий.

3.1. УСТРОЙСТВО ХЕШ-ТАБЛИЦЫ

Будем строить структуру данных типа «множество», которая поддерживает базовые операции: добавление ключа, проверка наличия ключа, удаление ключа. Для простоты будем считать, что ключи являются целыми числами из диапазона $[0, N)$.

Обозначим через K множество возможных ключей:

$$K = \{0, 1, 2, \ldots, N-1\}.$$

На практике это множество обычно довольно большое. Часто в качестве ключей в промышленном программировании применяются 32-битные или 64-битные целые числа, т. е. $N = 2^{32} \approx 4{,}2 \cdot 10^9$ или $N = 2^{64} \approx 1{,}8 \cdot 10^{19}$.

3.1.1. Прямая адресация

Если у нас достаточно памяти для массива, число элементов которого равно числу всех возможных ключей, то для каждого возможного ключа можно отвести ячейку в этом массиве и тем самым иметь возможность добраться до любой записи за время $O(1)$.

Таким образом, для хранения множества используется булев массив T размера N, называемый *таблицей с прямой адресацией*. Элемент t_i массива содержит истинное значение, если ключ i входит в множество, и ложное значение, если ключ i в множестве отсутствует. Нетрудно заметить, что все три указанные операции легко выполняются за константное время.

Прямая адресация обладает очевидным недостатком: если множество K всевозможных ключей велико, то хранить в памяти массив T размера N непрактично, а то и невозможно. Кроме того, если число реально присутствующих в таблице записей мало по сравнению с N, то много памяти тратится зря. Размер таблицы с прямой адресацией не зависит от того, сколько элементов реально содержится в множестве.

Минимальным адресуемым набором данных в современных компьютерах является один байт, состоящий из восьми битов. Не представляет трудности реализовать таблицу с прямой адресацией так, чтобы каждый бит был использован для хранения одной ячейки. Если N — мощность множества возможных ключей, то для прямой адресации требуется выделить последовательный блок из как минимум N бит памяти. Так, для размеров множества K в 10^9 элементов таблица займёт около 120 МБ памяти. Во многих случаях такой расход памяти неприемлем, особенно когда есть необходимость создавать несколько таблиц. Тем не менее при сравнительно небольших N метод прямой адресации успешно используется на практике.

3.1.2. Хеш-функция

Хеш-таблицу можно рассматривать как обобщение обычного массива с прямой адресацией.

Мы задумываем некоторую функцию, называемую *хеш-функцией* (англ. *hash function*), которая отображает множество ключей в некоторое гораздо более узкое множество:

$$h : K \to \{0, 1, \ldots, M-1\}, \tag{3.1}$$

$$x \mapsto h(x).$$

Величина $h(x)$ называется *хеш-значением* (*hash value*) ключа x.

Далее, вместо того чтобы работать с ключами, мы работаем с хеш-значениями. При этом возникают так называемые *коллизии* (*collisions*). Это ситуации, когда разные ключи получают одинаковые хеш-значения:

$$x \neq y, \quad h(x) = h(y).$$

Хотелось бы выбрать хеш-функцию так, чтобы коллизии были невозможны. Но в общем случае при $M < N$ это неосуществимо: согласно принципу Дирихле, нельзя построить инъективное отображение из большего множества в меньшее.

Пример 3.1. Приведём примеры некоторых функций. Так, константа $h(x) \equiv 0$ — хеш-функция, для которой любая пара различных ключей будет давать коллизию. Безусловно, такая хеш-функция бесполезна несмотря на то, что она простая и быстро вычисляется. Функция $h(x) = \mathrm{rand}(M)$, всякий раз возвращающая случайное число от 0 до $M-1$ включительно, выбранное равновероятно независимо от x, не может быть использована как хеш-функция, потому что хеш-функция обязана для равных ключей возвращать одинаковые значения. Функция $h(x) = x \bmod M$, возвращающая остаток от деления ключа x на M, является вполне годной для практики хеш-функцией и часто применяется.

Коллизии практически неизбежны, когда требуется осуществлять хеширование случайных подмножеств большого множества K допустимых ключей. Попробуем оценить вероятность коллизии с точки зрения комбинаторики. Рассуждения аналогичны тем, что используются для объяснения парадокса дней рождения в теории вероятностей. Предположим, что хеш-значения ключей независимы и распределены идеально

равномерно от 0 до $M-1$. Пусть мы осуществляем хеширование для n различных ключей ($n \leqslant M$). Когда мы назначаем всем ключам их хеш-значения, мы по сути строим некоторый вектор длины n, каждый элемент которого принимает одно из M значений. Всего существует M^n таких векторов (размещения с повторениями из M по n). Число векторов, в которых все элементы различны, равно $\frac{M!}{(M-n)!}$ (размещения без повторений из M по n). Разделив вторую величину на первую, мы получим вероятность того, что все элементы вектора различны. Значит, вероятность того, что в векторе найдутся хотя бы два одинаковых элемента, т. е. случится коллизия, равна

$$p(M,n) = 1 - \frac{M!}{(M-n)! \times M^n}.$$

Такое выражение неудобно для вычислений, но при помощи формулы Стирлинга для факториала можно вывести приближённые формулы, дающие неплохую точность, когда M велико и n много меньше M, например

$$p(M,n) \approx 1 - e^{\frac{-n^2}{2M}}.$$

Пример 3.2. Допустим, что число M возможных значений хеш-функции равно одному миллиону. Если мы осуществляем хеширование для $n = 2450$ уникальных ключей, то с вероятностью 95 % найдутся такие два ключа, что их хеш-значения будут одинаковыми, т. е. будет иметь место коллизия. Интуитивно кажется, что коллизии очень маловероятны, но это не так.

Если бы коллизий не было, хеш-таблицу можно было бы организовать как массив, в котором в качестве индексов использовались бы хеш-значения. Но из-за того, что возникают коллизии, структура хеш-таблицы более сложная. Разработано несколько стратегий разрешения коллизий.

3.2. РАЗРЕШЕНИЕ КОЛЛИЗИЙ МЕТОДОМ ЦЕПОЧЕК

Образно говоря, хеш-функция раскладывает исходные ключи по *корзинам* (англ. *bins*, *buckets*) или *слотам* (англ. *slots*). Ключ попадает в корзину с номером, равным хеш-значению.

Для хранения элементов с одинаковыми хеш-значениями внутри одной корзины можно использовать связные списки. На верхнем уровне

организуется массив размера M — по числу различных значений хеш-функции, каждый элемент которого — это односвязный список, состоящий из ключей, имеющих конкретное хеш-значение. Возникают цепочки ключей, из-за чего метод и получил название *метода цепочек* (англ. *separate chaining*).

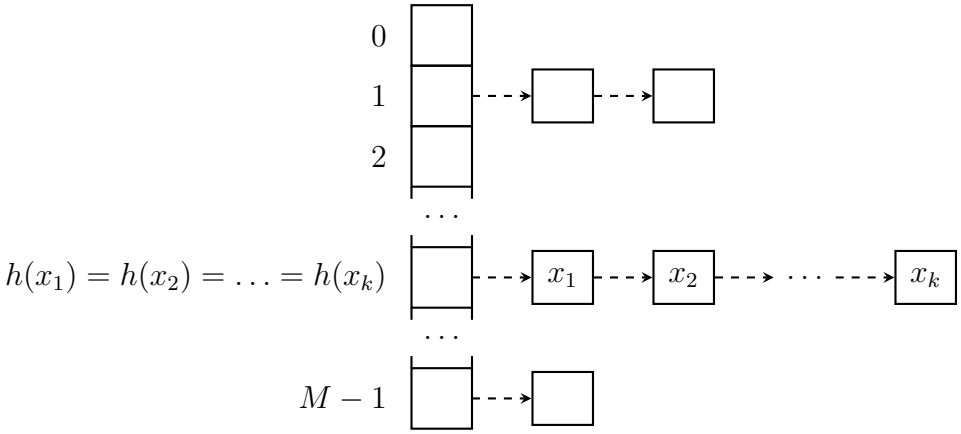


Рис. 3.1. Устройство хеш-таблицы с цепочками

Способ разрешения коллизий с помощью цепочек является одним из довольно часто применяемых на практике, хотя он и не единственный. Определим скорость работы такой хеш-таблицы.

Посмотрим на выполнение операций вставки. Сначала вычисляется хеш-значение $h(x)$ для ключа x, затем происходит обращение к соответствующему связному списку. Если не стоит задача проверять, присутствует элемент x в таблице или нет, то операция вставки может быть реализована за константное время: всегда можно добавить элемент в начало списка, и не нужно идти по всему связному списку. Однако обычно имеет смысл перед вставкой проверить, есть элемент x в таблице или нет, и добавлять только уникальные элементы. Это удобно в силу ряда причин: можно легко отвечать на запросы о числе элементов в множестве, меньше расход памяти (нередко на практике вставок выполняется много, но среди ключей мало различных), проще организовать удаление ключа. Поэтому операция вставки вначале выполняет проход по списку, и на это расходуется время, пропорциональное длине соответствующей цепочки.

Удаление ключа x также требует от нас выполнить прохождение списка в поиске элемента x. Отметим следующий факт: в общем случае из односвязного списка удалить элемент из середины сложно. Однако в рассматриваемом случае, несмотря на то, что список односвязный,

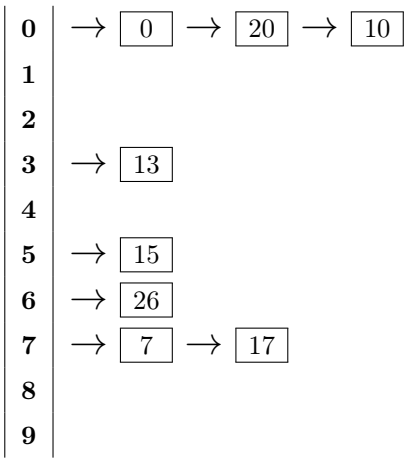


Рис. 3.2. Пример хеш-таблицы для заданного набора ключей

удалять из него нетрудно, потому что мы движемся слева направо и можем поддерживать указатель на текущий элемент и на предыдущий. При удалении указатель у предыдущего элемента перенаправляется на следующий элемент, а память из-под текущего элемента освобождается.

Таким образом, производительность всей конструкции связана с таким параметром, как длина цепочки. Для дальнейшего анализа введём обозначения для длин цепочек: $l_0, \ldots, l_{M-1}$. Для каждого хеш-значения длина своя.

Каждая из трёх рассмотренных операций с ключом x требует времени $\mathrm{O}(1 + l_i)$, где l_i — длина цепочки, в которую попадает ключ x. Отметим важность слагаемого 1 в асимптотике: даже если цепочка имеет нулевую длину, требуется время на то, чтобы вычислить хеш-значение (мы полагаем, что хеш-функция от ключа вычисляется за константу) и обратиться к соответствующей цепочке.

Пример 3.3. Используя хеш-функцию $h(x) = x \bmod 10$, сформируем хеш-таблицу размера 10 из последовательности ключей 0, 20, 15, 13, 26, 7, 17, 10. Результат показан на рис. 3.2.

3.3. РАЗРЕШЕНИЕ КОЛЛИЗИЙ МЕТОДОМ ОТКРЫТОЙ АДРЕСАЦИИ

Альтернативный подход называется *открытой адресацией* (англ. *open addressing*). В линейном массиве хранятся непосредственно ключи, а не заголовки связных списков. Когда выполняется операция вставки, ячейки массива проверяются, начиная с того места, в которое указы-

вает хеш-функция, в соответствии с некоторой *последовательностью проб* (англ. *probe sequence*), пока не будет найдено свободное место. Для осуществления поиска ключа массив проходится в той же последовательности, пока либо не будет найден искомый ключ, либо не будет обнаружена пустая ячейка (в таком случае утверждается, что искомого ключа нет). Операция удаления будет рассмотрена в разделе 3.3.2.

Название «открытая адресация» связано с тем фактом, что положение (адрес) элемента не определяется полностью его хеш-значением. Такой способ также называют *закрытым хешированием* (англ. *closed hashing*).

3.3.1. Последовательность проб

Обозначим через $h(x, i)$ номер ячейки в массиве, к которой следует обращаться на i-й попытке при выполнении операций с ключом x. Чтобы формулы были проще, удобно нумеровать попытки с нуля. Последовательность проб для ключа x получается такой:

$$h(x, 0),\ h(x, 1),\ h(x, 2), \ldots\ .$$

Для успешной работы алгоритмов поиска последовательность проб должна быть такой, чтобы в результате M проб все ячейки хеш-таблицы оказались просмотренными ровно по одному разу в каком-либо порядке:

$$\forall x \in K \quad \{h(x, i) \mid i = 0, 1, \ldots, M-1\} = \{0, 1, \ldots, M-1\}.$$

Широко используются три вида последовательностей проб: линейная, квадратичная и двойное хеширование.

Линейное пробирование. В этом случае ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом c между ячейками:

$$h(x, i) = (h'(x) + c \cdot i) \bmod M, \tag{3.2}$$

где $h'(x)$ — некоторая хеш-функция. Можно заметить, что не всякое значение c является подходящим.

Теорема 3.1. *Для того чтобы в ходе M проб все ячейки таблицы оказались просмотренными по одному разу, необходимо и достаточно, чтобы число c в формуле* (3.2) *было взаимно простым с размером хеш-таблицы M.*

Доказательство. Воспользуемся сведениями из элементарной теории чисел.

Докажем необходимость. Пусть $h(x, i)$ пробегает все значения от 0 до $M - 1$. Значит, для любого t найдётся такой индекс i, что $c \cdot i \equiv t \pmod{M}$. В частности, это верно для $t = 1$. Следовательно, есть такое i, что $c \cdot i \equiv 1 \pmod{M}$, или, другими словами, $c \cdot i - 1$ делится на M. Пусть d — общий делитель чисел c и M. Тогда число 1 также вынуждено делиться на d. Значит, $d = \pm 1$, т. е. числа c и M взаимно просты и не могут иметь других общих делителей.

Докажем достаточность. Пусть числа c и M взаимно просты. Предположим от противного, что при i-й и j-й пробах получаются одинаковые индексы: $h(x, i) = h(x, j)$. Но это значит, что $ci \equiv cj \pmod{M}$, или $c(i - j) \equiv 0 \pmod{M}$. Отсюда следует, что разность $i - j$ делится на M без остатка. Но раз номера попыток i и j лежат на отрезке от 0 до $M - 1$, то это возможно лишь в случае, когда $i = j$. Противоречие. Значит, при всех попытках пробы получаются разными. Поскольку попыток всего M, каждая ячейка будет учтена по одному разу. □

Пример 3.4. Пусть $M = 10$. Формула $h(x, i) = (x + 2i) \bmod 10$ не подходит в качестве последовательности проб. Например, при $x = 5$, подставляя i от 0 до 9, будем получать индексы

$$5, 7, 9, 1, 3, 5, 7, 9, 1, 3,$$

в результате чётные позиции оказываются не посещены. В то же время функция $h(x, i) = (x + 3i) \bmod 10$ работает правильно и возвращает каждый индекс один раз:

$$5, 8, 1, 4, 7, 0, 3, 6, 9, 2.$$

В простейшем случае можно взять единицу в качестве константы c в (3.2). Тогда ячейки просматриваются подряд слева направо, за последней ячейкой просматривается первая. Недостаток такой последовательности проб проявляется в том, что на реальных данных часто образуются кластеры из занятых ячеек (длинные последовательности ячеек, идущих подряд). При непрерывном расположении заполненных ячеек увеличивается время добавления нового элемента и других операций. Таким образом, линейная последовательность проб довольно далека от равномерного хеширования.

Пример 3.5. Пусть $M = 10$, тогда хеш-таблица представляет собой массив из десяти элементов. При осуществлении последовательности проб используется самая простая формула $h(x, i) = (x + i) \bmod 10$. Пусть в хеш-таблицу добавляются ключи

$$7, 29, 67, 38, 25, 27, 18$$

в указанном порядке. Тогда массив будет заполнен следующим образом:

0	1	2	3	4	5	6	7	8	9
38	27	18			25		7	67	29

Квадратичное пробирование. Интервал между ячейками с каждым шагом увеличивается на константу:

$$h(x, i) = (h'(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod M,$$

где числа c_1 и c_2 фиксированы. Значения c_1 и c_2 должны быть тщательно подобраны, чтобы в результате M попыток все ячейки были посещены.

На практике такой метод часто работает лучше линейного, разбрасывая ключи более равномерно по массиву. Тенденции к образованию кластеров нет, но аналогичный эффект проявляется в форме образования вторичных кластеров.

Двойное хеширование. Интервал между проверяемыми ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей:

$$h(x, i) = (h'(x) + h''(x) \cdot i) \bmod M.$$

Последовательность проб в этой ситуации при работе с ключом x представляет собой арифметическую прогрессию (по модулю M) с первым членом $h'(x)$ и шагом $h''(x)$.

Чтобы последовательность проб покрыла всю таблицу, значение $h''(x)$ должно быть ненулевым и взаимно простым с M.

Для этого можно поступить следующим образом:

• выбрать в качестве M степень двойки, а функцию $h''(x)$ взять такую, чтобы она принимала только нечётные значения;

• выбрать в качестве M простое число и потребовать, чтобы вспомогательная хеш-функция $h''(x)$ принимала значения от 1 до $M - 1$.

3.3.2. Поддержка операции удаления

Удаление элементов в схеме с открытой адресацией несколько затруднено, и все операции немного усложняются. Рассмотрим один из способов. Для ячейки вводятся три состояния:

- EMPTY — ячейка пуста;
- KEY(x) — ячейка содержит ключ x;
- DELETED — ячейка ранее содержала ключ, но он был удалён.

В любой момент каждая из M ячеек массива находится в одном из трёх состояний. Для удобства обычно поддерживают отдельную целочисленную величину — счётчик ячеек, заполненных актуальными ключами (ячеек в состоянии KEY). Например, это позволяет быстро отвечать на вопрос об общем числе хранящихся элементов. Иначе для этого всякий раз нужно было бы сканировать весь массив за время $O(M)$.

Итак, изначально все ячейки пусты (состояние EMPTY), счётчик выставлен в нуль.

При поиске ключа x необходимо просмотреть все возможные местоположения этого ключа: проверяются ячейки $h(x, 0)$, $h(x, 1)$, ..., пока не найдётся либо ячейка KEY(x) (говорим, что ключ найден), либо ячейка EMPTY (говорим, что искомого ключа в таблице нет). Если встречается ячейка DELETED, процесс пробирования её игнорирует и идёт дальше.

Перед вставкой ключа x сначала выполняется поиск этого ключа. Если ключ уже есть, вставка не требуется. Иначе проверяется принципиальное наличие ячеек, не занятых ключами (используем счётчик занятых ячеек). Если все M ячеек имеют состояние KEY, вставка невозможна: таблица переполнена. Иначе вновь пробуются позиции $h(x, 0)$, $h(x, 1)$, ..., пока не будет найдена либо свободная ячейка EMPTY, либо ячейка с удалённым ключом DELETED (удалённые ячейки при вставке приравниваются к свободным, а при поиске нет). Ячейка переводится в состояние KEY(x), счётчик занятых ячеек увеличиваем на единицу.

При удалении ключа x вначале выполняем поиск ключа x. Если такая ячейка найдена, переводим её в состояние DELETED и счётчик занятых ячеек уменьшаем на единицу.

Нетрудно видеть, что наличие большого числа DELETED-ячеек отрицательно сказывается на времени выполнения операции поиска, а значит и других операций. Чтобы исправить ситуацию, после ряда удалений можно перестраивать хеш-таблицу заново, уничтожая удалённые ячейки.

3.3.3. Плюсы и минусы открытой адресации

Недостаток систем с открытой адресацией состоит в том, что число хранимых ключей не может превышать размер хеш-массива. По факту, даже при использовании хороших хеш-функций, производительность резко падает, когда хеш-таблица оказывается заполненной на 70 % и более. Во многих приложениях это приводит к обязательному использованию динамического расширения таблицы, о котором будет идти речь в разделе 3.4.

Схема с открытой адресацией предъявляет более строгие требования к хеш-функции, чтобы механизм работал хорошо. Кроме того, чтобы распределять значения максимально равномерно по корзинам, функция должна минимизировать кластеризацию хеш-значений, которые стоят рядом в последовательности проб. Так, в методе цепочек одна забота — чтобы много ключей не получило одно и то же хеш-значение, и если хеш-значения получились разные, совершенно всё равно, рядом ли они стоят, отличаются ли на единицу и т. д. Если при использовании метода открытой адресации образовались большие кластеры, время выполнения всех операций может стать неприемлемым даже при том, что заполненность таблицы в среднем невысокая и коллизии редки.

Открытая адресация позволяет существенно экономить память, если размер ключа невелик по сравнению с размером указателя. В методе цепочек приходится хранить в массиве указатели на начала списков, а каждый элемент списка хранит, кроме ключа, указатель на следующий элемент, поэтому на все эти указатели расходуется память.

Открытая адресация не требует затрат времени на выделение памяти на каждую новую запись и может быть реализована даже на миниатюрных встраиваемых системах, где полноценный аллокатор недоступен. Также в открытой адресации нет лишней операции обращения по указателю (*indirection*) при доступе к элементу. Открытая адресация обеспечивает лучшую локальность хранения, особенно с линейной функцией проб. Когда размеры ключей небольшие, это даёт лучшую производительность за счёт хорошей работы кеша процессора, который ускоряет обращения к оперативной памяти. Однако когда ключи «тяжёлые» (не целые числа, а составные объекты), они забивают все кеш-линии процессора, к тому же много места в кеше тратится на хранение незанятых ячеек. Как вариант, можно в массиве с открытой адресацией хранить не сами ключи, а указатели на них. Очевидно, часть преимуществ при этом будет утрачена.

Так или иначе, любой подход к реализации хеш-таблицы может работать достаточно быстро на реальных нагрузках. Время, которое занимают операции с хеш-таблицами, обычно составляет малую долю от общего времени работы программы. Расход памяти редко играет решающую роль. Часто выбор между той или иной реализацией хеш-таблицы делается на основании других факторов в зависимости от ситуации.

3.4. КОЭФФИЦИЕНТ ЗАПОЛНЕНИЯ

Критически важным показателем для хеш-таблицы является *коэффициент заполнения* (англ. *load factor*) — отношение числа ключей, которые хранятся в хеш-таблице, к размеру хеш-таблицы:

$$\alpha = \frac{n}{M}.$$

Коэффициент заполнения может быть как меньше, так и больше единицы (не для всех способов разрешения коллизий такое возможно: например, это недопустимо для разрешения коллизий методом открытой адресации).

Пусть для разрешения коллизий используется метод цепочек. На первый взгляд очевидно, что чем больше коэффициент заполнения, тем медленнее работает хеш-таблица. Однако коэффициент заполнения не показывает различия между заполненностью отдельных корзин. Например, пусть есть две хеш-таблицы, в каждой используется 1000 корзин и хранится всего 1000 ключей, поэтому коэффициент заполнения в обоих случаях равен единице. Однако в первой таблице в каждой цепочке по одному ключу, а во второй все ключи лежат в одной длинной цепочке. Очевидно, что вторая хеш-таблица будет работать очень медленно.

Низкий коэффициент заполнения не является абсолютным благом. Если коэффициент близок к нулю, это говорит о том, что большая часть таблицы не используется и память тратится впустую.

Для оптимального использования хеш-таблицы желательно, чтобы её размер был примерно пропорционален числу ключей, которые нужно хранить. На практике редко случается, что число ключей фиксировано и можно заранее выставить хорошее значение параметра M. Если ставить его заведомо больше, то много памяти будет потрачено зря (особенно если нужно организовать много хеш-таблиц с небольшим числом ключей в каждой).

Реализация хеш-таблицы общего назначения обязана поддерживать операцию изменения размера.

Часто используемым приёмом является автоматическое изменение размера, когда коэффициент заполнения превышает некоторый порог $\alpha_{\max}$. Выделяется память под новую, бóльшую таблицу, все элементы из старой таблицы перемещаются в новую, затем память из-под старой хеш-таблицы освобождается. Аналогично, если коэффициент заполненности опускается ниже другого порога $\alpha_{\min}$, элементы перемещаются в хеш-таблицу меньшего размера.

Чисто технически в языках программирования под хеш-функцией понимается функция, отображающая ключ в произвольное целое число без ограничения на величину (не требуется попадания в промежуток $[0, M)$, как в определении (3.1)). В этом есть здравый смысл. Хеш-значение — это свойство ключей, и оно не зависит от размера хеш-таблицы. Тот факт, что мы далее берём его по модулю M, — это исключительно особенность его использования внутри конкретной хеш-таблицы конкретного размера. Не нужно предварительно брать остаток. Часто на практике хеш-кодами выступают всевозможные беззнаковые целые числа, и остаток берётся по нужному модулю непосредственно в месте использования.

3.5. ОЦЕНКИ ВРЕМЕНИ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ

Оценим время выполнения операций над хеш-таблицей в случае разрешения коллизий методом цепочек. Метод открытой адресации кажется существенно более сложным для теоретического анализа.

Нетрудно понять, что от выбора хеш-функции многое зависит. Если она будет «плохая» (например константная), то будут иметь место сплошные коллизии (хеш-таблица превратится в связный список). Как определить меру качества хеш-функции и научиться получать «хорошие» хеши?

За годы развития вычислительной техники сформировалась наука о том, как практически строить хеш-функции. Разработаны всевозможные правила на тему того, какие биты ключа x стоит взять, на сколько позиций выполнить поразрядный сдвиг, как применить операцию исключающего «или», на что умножить, чтобы получить хеш, который бы выглядел как случайный. Чтобы найти примеры практических хеш-функций, можно взять какую-нибудь стандартную библиотеку

языка программирования (см. раздел 3.8). Эти функции часто ничего не гарантируют: коллизии, конечно, есть. Отметим, что коллизии неизбежны. Любую фиксированную хеш-функцию можно всегда «сломать» — придумать ситуацию, когда там будут одни сплошные коллизии.

Как же тогда оценивать трудоёмкость? Когда длины цепочек «хорошие», всё работает быстро. Однако на практике длины цепочек бывают «плохими». Хотелось бы мыслить в терминах оценок сверху, однако в худшем случае время работы оказывается неприемлемым. Теоретически есть два направления возможной деятельности.

Первое направление заключается в том, чтобы добавить рандомизацию: внести элемент случайности, чтобы не было фиксированного «плохого» случая. Есть идея не брать какую-либо одну конкретную хеш-функцию, а, например, организовать программу так, чтобы при каждом запуске хеш-функция выбиралась заново. Тогда одним и тем же входом «сломать» программу не получится, и можно будет рассуждать о длине цепочки с точки зрения теории вероятностей: говорить о том, какая длина цепочки в среднем, какая у неё дисперсия... Такой подход называют *универсальным хешированием* (англ. *universal hashing*).

Второе направление состоит в следующем. Несмотря на то, что потенциальное множество ключей велико, в каждый момент времени мы храним ключей меньше, чем число корзин в хеш-таблице. Пусть S — фактическое множество ключей в хеш-таблице, и у этого множества размер n, где $n \leqslant M$. Тогда понятно, что существует хеш-функция, которая не даёт коллизий: если элементов меньше, чем слотов, то их всегда можно отобразить без коллизий. Это хорошая хеш-функция, её называют *совершенной* (англ. *perfect hash*). В чём проблема? Она всегда есть и зависит от набора ключей. Даже если предположить, что набор ключей константный и известен заранее, есть трудность в том, как эффективно задать эту хеш-функцию.

Использовать для задания функции таблицу значений непрактично, потому что множество K образов для этой функции слишком большое. Если применять какую-либо ассоциативную структуру, это ровно та же задача, которую мы пытаемся решать (если для вычисления хеш-функции делать запрос в хеш-таблицу, то какую хеш-функцию использовать в этой новой хеш-таблице?). Другое решение: например, можно все значения S отсортировать по возрастанию и в качестве хеш-функции элемента использовать его порядковый номер. Это правильно с точки зрения избавления от коллизий, но как тогда определять номер

элемента во время работы программы? Мы не можем использовать бинарный поиск (см. часть 6), потому что он требует логарифмического времени и сведёт на нет смысл построения хеш-таблицы. Удобно, когда хеш-функция — небольшое выражение, которое может быть вычислено за константное время.

Итак, возникает задача построения хеш-функции, которая является

- простой, т. е. достаточно константного объёма памяти, чтобы её хранить;
- быстрой, т. е. требуется константное время на вычисление;
- совершенной, т. е. коллизии отсутствуют.

Оказывается, сделать это можно, но рассмотрение данного вопроса выходит за рамки учебно-методического пособия.

3.6. ИДЕАЛЬНОЕ ХЕШИРОВАНИЕ

Первая идея — взять в качестве h случайную функцию. Это так называемое *идеальное хеширование* в том смысле, что оно нереализуемо на практике и существует лишь в теории.

Что значит «h — случайная функция»? Функцию можно представить как таблицу, в которой для каждого значения аргумента написано, чему функция равна. Под случайной функцией понимают функцию, у которой каждое значение выбирается случайно равновероятно на множестве $\{0, 1, \ldots, M-1\}$ и все значения выбираются независимо.

Мы пытаемся заполнить таблицу числами от 0 до $M-1$, причём числа могут повторяться, количество элементов в этой таблице равно N. Каждая клетка выбирается независимо в соответствии с равномерным распределением.

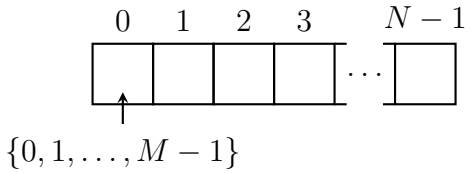


Рис. 3.3. Задание идеальной хеш-функции с помощью таблицы

Идеальная хеш-функция позволяет получить хорошую оценку для длины цепочки.

3.6.1. Оценка длины цепочки

Пусть $S = \{x_1, \ldots, x_n\}$. Для фиксированного хеш-значения t длина цепочки l_t будет случайной величиной. Определим её математическое ожидание:

$$\mathbf{E}\{l_t\} = \mathbf{E}\left\{\sum_{i=1}^{n} \mathbb{1}_{\{h(x_i)=t\}}\right\},$$

где $\mathbb{1}_{\{h(x_i)=t\}}$ — индикатор того события, что i-й ключ попал в список, соответствующий хеш-значению t, т. е.

$$\mathbb{1}_{\{h(x_i)=t\}} = \begin{cases} 1, & \text{если } h(x_i) = t, \\ 0, & \text{иначе.} \end{cases}$$

Нетрудно видеть, что

$$\mathbf{E}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = 1 \cdot \mathbf{P}\{h(x_i) = t\} + 0 \cdot \mathbf{P}\{h(x_i) \neq t\} = \frac{1}{M},$$

поскольку $h(x_i)$ равновероятно принимает некоторое значение из M возможных, одно из которых t, и не зависит от i.

Из линейности математического ожидания следует, что

$$\mathbf{E}\{l_t\} = \sum_{i=1}^{n} \mathbf{E}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = \frac{n}{M} = \alpha,$$

где α — введённый ранее в разделе 3.4 коэффициент заполнения хеш-таблицы, равный отношению количества хранящихся в хеш-таблице элементов к количеству цепочек.

Аналогично можно вычислить дисперсию длины цепочки. Действительно, каждый индикатор является дискретной случайной величиной, имеющей распределение Бернулли с вероятностью успеха $1/M$, поэтому

$$\mathbf{V}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = \frac{1}{M}\left(1 - \frac{1}{M}\right).$$

Поскольку случайные величины независимы, дисперсия суммы равна сумме дисперсий:

$$\mathbf{V}\{l_t\} = \sum_{i=1}^{n} \mathbf{V}\left\{\mathbb{1}_{\{h(x_i)=t\}}\right\} = \frac{n}{M}\left(1 - \frac{1}{M}\right) < \frac{n}{M} = \alpha.$$

Получается, что длина цепочки имеет константное матожидание и константную дисперсию. В среднем элементы располагаются по корзинам равномерно и длины цепочек небольшие.

3.6.2. Сложность идеальной хеш-функции

С точки зрения теории информации хеш-функция сложна, потому что количество информации, содержащейся в этой функции, велико.

Представлять, что хеш-функция — это таблица из N ячеек, непрактично, потому что N обычно очень велико. Если бы можно было создать такую таблицу, не было бы смысла использовать хеширование.

Таблицу можно строить не сразу, а постепенно. А именно, когда спрашивают значение $h(x)$, можно сгенерировать случайное значение и выдать его. Но для одного и того же x хеш-функция должна возвращать один и тот же результат, поэтому необходимо хранить ранее выданные значения, а эта задача аналогична той, которую мы решаем.

На практике идеальных хеш-функций не бывает, но они обладают рядом интересных свойств. Во-первых, этих свойств достаточно для того, чтобы получать некоторые оценки. Во-вторых, существуют способы построения функций h, обладающих теми же свойствами, но построение этих функций менее затратно.

Нужно отказаться от сильного свойства полной покоординатной случайности и независимости и заменить на что-то более простое, но так, чтобы сохранить оценку матожидания (дисперсию сохранить, увы, не удастся).

Заметим, что для получения оценки матожидания мы мало чем пользовались. Так, полученная оценка верна, даже если хеш-функция на всех аргументах принимает одно и то же случайно выбранное значение. Вообще, в анализе алгоритмов хорошее матожидание мало что значит, важнее оценивать матожидание в совокупности с дисперсией, а ещё лучше оценивать худший случай с высокой вероятностью, т. е. оценку, которая верна «почти всегда».

3.7. УНИВЕРСАЛЬНОЕ ХЕШИРОВАНИЕ

Универсальное хеширование (англ. *universal hashing*) — это хеширование, при котором хеш-функция выбирается случайным образом из определённого семейства хеш-функций

$$H = \{h : K \to \{0, 1, \ldots, M-1\}\}.$$

Семейство хеш-функций H называется *универсальным* (англ. *universal*), если для любых двух различных значений ключа x и y вероятность получить коллизию ведёт себя ожидаемым образом, т. е.

$$\mathbf{P}_{h \in H} \{h(x) = h(y)\} = \mathrm{O}\left(\frac{1}{M}\right). \tag{3.3}$$

Когда константа, скрытая в асимптотике, равна единице, семейство называют *сильно универсальным*. Часто для теоретических рассуждений величина константы особой роли не играет.

Можно пытаться усилить универсальность и ввести понятие k-независимости. Семейство функций H называют *k-независимым*, если для любых различных ключей $x_1, x_2, \ldots, x_k$ и для любых возможных значений хеш-функций $t_1, t_2, \ldots, t_k$ имеет место условие

$$\mathbf{P}_{h \in H} \{h(x_1) = t_1,\ h(x_2) = t_2,\ \ldots,\ h(x_k) = t_k\} = \mathrm{O}\left(\frac{1}{M^k}\right). \tag{3.4}$$

Формально у нас есть некоторое семейство хеш-функций H с некоторым распределением вероятностей на нём, и $h \in H$ выбирается в соответствии с этим распределением. Важно понимать, что в формулах (3.3) и (3.4) вероятности берутся не по значениям ключей (все x_i фиксируются), а по h. Поэтому на множестве всех h есть некоторое распределение (обычно равномерное, но не обязательно). Итак, есть семейство H, и из этого семейства мы выбираем h.

В эту схему, когда хеш-функция выбирается из семейства, отлично вкладывается концепция идеальной хеш-функции. В этом случае H — множество всех функций с равномерным распределением. Свойство универсальности, конечно, всегда справедливо для идеальной хеш-функции. Более того, оно справедливо с жёсткой константой, равной единице.

3.7.1. Построение универсального семейства хеш-функций для целочисленных ключей

Рассмотрим один способ построения универсального семейства хеш-функций для целочисленных ключей. Метод был предложен Дж. Л. Картером (J. L. Carter) и М. Н. Вегманом (M. N. Wegman) в 1977 г. Фактически семейство будет даже сильно универсальным. Способ линейно-алгебраический по содержанию и состоит из двух шагов.

На первом шаге возьмём простое число p, которое больше или равно N. Например, можно взять ближайшее простое не меньше N. По

разным свойствам распределений простых чисел мы знаем, что p будет, вообще говоря, недалеко от N. Так, постулат Бертрана гласит, что существует достаточно близкое к N простое число: $N \leqslant p < 2N$.

Пример 3.6. Для $N = 1000$ можно взять $p = 1009$.

Через $\mathbb{Z}_p$ будем обозначать множество наименьших неотрицательных вычетов по модулю p:

$$\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}.$$

На втором шаге мы независимо формируем два случайных вычета по модулю p, один из которых ненулевой:

$$a \in \mathbb{Z}_p \setminus \{0\}, \quad b \in \mathbb{Z}_p.$$

После того как эти два значения выбраны, возникает функция, задаваемая выражением

$$(ax + b) \bmod p. \tag{3.5}$$

Заметим, что раз $N \leqslant p$, то ключ x, как a и b, является вычетом по модулю p, но старшая часть вычетов запрещена (от N до $p-1$ включительно).

Формула (3.5) для любого ключа x даёт на выходе вычет по модулю p, который в качестве хеш-значения не годится, потому что p может быть велико. Есть отличный способ сделать его меньше — взять по модулю M:

$$h_{a,b}(x) = [(ax + b) \bmod p] \bmod M. \tag{3.6}$$

Полученная функция $h_{a,b}(x)$ может служить хеш-функцией для построения хеш-таблиц размера M.

Пример 3.7. Пусть $M = 10$ и $N = 1000$. Зафиксируем простое число $p = 1009$. Тогда коэффициент a можно выбирать от 1 до 1008 (всего 1008 способов), коэффициент b можно выбирать от 0 до 1008 (всего 1009 способов). Итого можно получить $1008 \times 1009 = 1\,017\,072$ разных функции, например

$$\begin{aligned} h_{3,4}(x) &= ((3x + 4) \bmod 1009) \bmod 10, \\ h_{670,905}(x) &= ((670x + 905) \bmod 1009) \bmod 10, \\ h_{101,0}(x) &= ((101x \bmod 1009) \bmod 10, \\ &\ldots \end{aligned} \tag{3.7}$$

Заметим, что в общем случае нельзя «для простоты счёта» опустить первое взятие остатка, например

$$((1010 \bmod 1009) \bmod 10) = 1 \neq 0 = (1010 \bmod 10).$$

Конструкция (3.6) выглядит странно: часто один вычет по модулю другого вычета не имеет особого смысла, только если один модуль не является делителем другого. Но в данном случае h как раз оказывается хеш-функцией, выбранной равномерно из универсального семейства. Справделива следующая теорема.

Теорема 3.2. *Семейство функций*

$$H = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\},$$

где $h_{a,b}(x)$ определяется по формуле (3.6)*, число p простое, является сильно универсальным.*

Пример 3.8. Пусть $N = 1000$, $M = 10$. Выберем p равным 1009, как в примере 3.7. Ранее отмечалось, что $|H| = 1\,017\,072$.

Проверим теорему 3.2 на практике. Зафиксируем два различных ключа x и y, например $x = 1$ и $y = 2$. Затем промоделируем процесс выбора хеш-функции на компьютере: будем перебирать в двух вложенных циклах a от 1 до 1008 и b от 0 до 1008 включительно, каждый раз будем вычислять хеши $h(x)$ и $h(y)$, сравнивать их и подсчитывать число случаев, когда случается коллизия. Оказывается, что это число равно 100 800 и не зависит от x и y. Значит, вероятность того, что на данной паре ключей при случайном выборе хеш-функции из универсального семейства будет коллизия, равна $\frac{100\,800}{1\,017\,072} \approx 0{,}0991 < 0{,}1 = \frac{1}{M}$.

Можно рассмотреть хеш-функции более простого вида:

$$h_a(x) = (ax \bmod p) \bmod M. \tag{3.8}$$

Теорема 3.3. *Семейство функций*

$$H = \{h_a \mid a \in \mathbb{Z}_p, a \neq 0\},$$

где $h_a(x)$ определяется по формуле (3.8)*, число p простое, обладает свойством универсальности, но не является сильно универсальным.*

3.7.2. Универсальное хеширование векторов

Если в качестве ключей выступают не просто целые числа, а объекты других типов, то нужно использовать соответствующие универсальные семейства хеш-функций.

Например, пусть ключ представляет собой вектор фиксированной длины

$$x = (x_1, x_2, \ldots, x_r),$$

составленный из целых чисел, каждое из которых лежит в отрезке от 0 до $N - 1$, при этом N небольшое. Размер хеш-таблицы M выберем простым и таким, чтобы выполнялось условие $M \geqslant N$.

Хеш-функцию будем строить по формуле

$$h_a(x) = \sum_{i=1}^{r} a_i x_i \bmod M, \tag{3.9}$$

где $a = (a_1, a_2, \ldots, a_r)$ — случайно выбираемый вектор из вычетов по модулю M. Нетрудно видеть, что всего существует M^r таких векторов.

Теорема 3.4. *Семейство H, составленное из всех функций вида* (3.9)*, является универсальным семейством хеш-функций.*

Пример 3.9. IP-адрес представляет собой 32-битное целое число, которое часто записывают как четвёрку 8-битных целых чисел, например 192.168.1.2. Пусть нужно организовать хранение примерно 250 IP-адресов в хеш-таблице.

Каждый адрес можно рассматривать как четырёхкомпонентный вектор. Тогда получается, что $N = 256$. Размер таблицы M предлагается взять равным 257 (это простое число).

Для выбора хеш-функции из универсального семейства нужно сгенерировать четыре случайных целых числа из отрезка от 0 до 256 включительно, а потом подставить их в формулу (3.9). Например, если выбрано $a = (87, 23, 125, 4)$, то получается такая хеш-функция:

$$h(x) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \bmod 257.$$

3.8. ХЕШ-ТАБЛИЦЫ НА ПРАКТИКЕ

Структуры данных на основе хеш-таблиц реализованы в стандартных библиотеках всех широко используемых языков программирования. В большинстве случаев библиотеки предоставляют как множество, так и ассоциативный массив.

3.8.1. Требования к ключам

Зачастую при программировании ключами в множествах и ассоциативных массивах выступают не просто целые числа, а какие-либо более сложные объекты, например строки, пары чисел, структуры из нескольких полей и прочие объекты произвольной природы.

Чтобы объект некоторого типа мог выступать ключом в хеш-таблице, необходимо выполнение двух условий:

- должна быть задана хеш-функция, которая ставит в соответствие объекту его хеш-значение (целое число, причём диапазон допустимых значений оговаривается в спецификациях);
- должна быть определена функция, которая для пары объектов отвечает, равны они или нет.

Формально от хеш-функции требуется лишь, чтобы для равных объектов хеш-значения были равными. Функция проверки на равенство должна задавать на множестве объектов отношение эквивалентности, т. е. бинарное отношение, для которого выполнены следующие условия:

1) $a = a$ для любого a (свойство рефлексивности);
2) если $a = b$, то $b = a$ (свойство симметричности);
3) если $a = b$ и $b = c$, то $a = c$ (свойство транзитивности).

Для того чтобы объект мог служить ключом в бинарном дереве поиска, требование выдвигается совсем иное: нужно уметь сравнивать любые два элемента. Строго говоря, множество объектов должно быть линейно упорядочено. Напомним, что линейным порядком на множестве называют бинарное отношение $\leqslant$, которое удовлетворяет следующим условиям:

1) любые два объекта сравнимы между собой, т. е. верно хотя бы одно из неравенств $a \leqslant b$ или $b \leqslant a$ (свойство полноты);
2) если $a \leqslant b$ и $b \leqslant a$, то $a = b$ (свойство антисимметричности);
3) если $a \leqslant b$ и $b \leqslant c$, то $a \leqslant c$ (свойство транзитивности).

Такой порядок также называют *полным порядком* (англ. *total order*). Линейно упорядоченные объекты могут быть размещены на прямой линии один за другим. Первое свойство влечёт за собой свойство рефлексивности, т. е. $a \leqslant a$. Следовательно, линейный порядок является также частичным порядком. Понятие частично упорядоченного множества слабее (в определении частичного порядка в первом пункте требуется лишь рефлексивность).

От языка программирования зависит способ, при помощи которого можно определять те или иные операции для объектов произвольного

типа (подсчёт хеш-функции, проверка на равенство или неравенство). При этом на программиста возлагается контроль того, что реализация операции удовлетворяет всем требованиям.

Например, если функция проверки двух объектов на равенство будет возвращать каждый раз случайное значение (истинное или ложное), естественно ожидать, что хеш-таблица будет работать неправильно. Если функция сравнения для множества на основе дерева не обеспечивает линейного упорядочивания, программа будет вести себя непредсказуемо.

3.8.2. Объединение хеш-значений

Нередко на практике требуется реализовать хеш-функцию от структуры, содержащей два поля, для каждого из которых по отдельности хеш-функции определены. Например, координаты точек на плоскости хранятся в виде пар целых чисел (x, y), и нужно создать множество точек с использованием хеш-таблицы, а для этого необходимо вычислять хеш-значения от точек. Пусть получены хеш-значения двух координат $h(x)$ и $h(y)$. Как их объединить, чтобы получить хеш от пары? Пусть для простоты верхняя граница возможных значений хеш-функции не фиксирована (где-то дальше в реализации хеш будет взят по нужному модулю M).

Часто на практике программисты для соединения хешей пишут тривиальные функции, например через операцию побитового исключающего или (xor):

```
def combine(hx, hy):
    return hx ^ hy
```

Такой вариант часто работает на практике приемлемо, но не лишён очевидных недостатков. Например, для всех точек с равными координатами x и y хеш-функция будет принимать нулевое значение, и если точек на прямой $y = x$ во входных данных окажется много, производительность будет низкой из-за коллизий. Также очевидно, что разные точки (x, y) и (y, x), симметричные относительно той же прямой, получат одинаковые хеш-значения.

Чтобы подобрать пары, дающие коллизию, было труднее, для объединения хешей используют более сложные функции с обилием «магических» констант и странных операций. Например, в C++-библиотеке boost используется примерно такая формула:

```
def combine(hx, hy):
    return hx ^ (hy + 0x9e3779b9 + (hx << 6) + (hx >> 2))
```

Часто берут линейную комбинацию двух хеш-значений с, например, большими взаимно простыми коэффициентами. Как вариант:

```
def combine(hx, hy):
    return hx + 1000000007 * hy
```

Основной смысл таких манипуляций — сделать так, чтобы на реально встречающихся в жизни данных коллизии были более редкими. Но контрпример при желании можно подобрать. Лучшего универсального решения в этом деле нет.

Отметим, что если научиться объединять хеши двух элементов, то можно последовательно объединить хеши любого числа элементов.

3.8.3. Проход по содержимому хеш-таблицы

В процессе программирования может возникнуть необходимость выполнить обход всех элементов структуры данных и, например, распечатать их. Предположим, требуется вывести все ключи, которые содержатся в заданном множестве.

Функция для итерации по содержимому структуры данных не вводилась в разделе 2.6, но является полезной, поэтому обычно поддерживается в реализациях хеш-контейнеров, с которыми ведётся работа на практике.

В большинстве реализаций проход по хеш-множествам выполняется в произвольном порядке, не гарантируется какой-либо отсортированности ключей. В случае, если внутренняя реализация хеш-таблицы использует метод цепочек, обычно функция обхода выдаёт сначала все элементы первой корзины (с хеш-значением 0) в порядке их следования в цепочке, затем все элементы второй корзины (с хеш-значением 1), и т. д. Для внешнего наблюдателя может казаться, что ключи выходят в случайном порядке.

Более того, если распечатать элементы хеш-множества, добавить новый ключ, сразу удалить его, вновь распечатать элементы, то порядок может получиться другим. Такое может случиться, если добавление нового ключа привело к перестроению хеш-таблицы с изменением числа M корзин и элементы были перераспределены по корзинам вновь.

Не стоит нигде в коде закладываться на порядок итерации по хеш-контейнерам: большинство реализаций в разных языках программирования могут гарантировать только то, что посещены будут все элементы, не важно в каком порядке.

Наоборот, средства итерации по ключам множества, которое построено на базе бинарного поискового дерева, обычно возвращают ключи в порядке возрастания (выполняется внутренний обход дерева). Порядок фиксирован и каждый раз одинаковый. Часто предсказуемость результата удобна, например, для написания модульных тестов к частям программы.

Таким образом, если порядок итерации важен, возможно, стоит использовать «древесные» структуры данных.

3.8.4. Хеш-таблицы в C++

Долгое время в языке C++ не было стандартных реализаций структур данных на основе хеш-таблиц. Контейнеры `std::set` и `std::map` из STL строятся на основе сбалансированных бинарных поисковых деревьев (во всех популярных реализациях применяются красно-чёрные деревья). Хеш-таблицы существовали в виде нестандартных расширений (например `stdext::hash_set` в Visual Studio) или внешних библиотек (например boost).

Наконец, в стандарте C++11 в STL официально были добавлены хеш-таблицы. Стандарт предусматривает четыре контейнера на основе хеш-таблиц, которые отличаются от своих аналогов на основе деревьев наличием префикса `unordered_` в названии. Так, `std::unordered_set` представляет собой динамическое множество, `std::unordered_map` — ассоциативный массив. Существует также два `multi`-контейнера, которые допускают хранение одинаковых ключей.

Стандарт требует, чтобы в построении этих структур данных авторы компиляторов использовали разрешение коллизий методом цепочек. Метод открытой адресации не был стандартизирован из-за внутренних трудностей при удалении элементов. Однако детали реализации хеш-таблиц стандартом не регламентируются.

В качестве хеш-значения в C++ используется число типа `size_t`. Все хеш-контейнеры предоставляют метод `rehash()`, который позволяет установить размер хеш-таблицы (число корзин M). Метод под названием `load_factor()` возвращает текущий коэффициент заполнения.

Рассмотрим более подробно реализацию в компиляторе GCC. Пусть ключи добавляются в `std::unordered_set` по одному. Когда коэффициент заполнения достигает значения 1, происходит перестроение хеш-таблицы: в качестве нового числа корзин берётся первое простое

число из заранее составленного списка, не меньшее удвоенного старого числа корзин (таким образом, размер таблицы как минимум удваивается и является простым числом). Длины отдельных цепочек никак не анализируются (появление одной длинной цепочки не повлечёт за собой операцию перестроения).

3.8.5. Хеш-таблицы в Java

Предполагаем, что речь идёт о языке Java актуальных на момент написания этих строк версий.

Коллекции `HashSet` и `HashMap` реализуются как хеш-таблицы, для разрешения коллизий используется метод цепочек.

Для хеширования целых чисел применяется функция следующего вида:

```
int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

(здесь операция `>>>` — беззнаковый сдвиг вправо: биты смещаются вправо, число слева дополняется нулями, операция `^` — поразрядное сложение по модулю 2, исключающее «или»). Затем в классе коллекции результат функции `hash` берётся по модулю числа корзин, по которым раскладываются элементы. Число корзин, оно же число различных значений хеш-функции M, в Java всегда выбирается как некоторая степень числа 2, чтобы деление на M можно было заменить операцией битового сдвига вправо (современные процессоры выполняют инструкцию деления целых чисел существенно медленнее, чем битовые операции).

В версии Java 8 разработчики озаботились вопросом устойчивости коллекций, использующих хеширование, к коллизиям. В исходном коде библиотеки Java можно найти следующую константу:

```
static final int TREEIFY_THRESHOLD = 8;
```

В случае, если новый ключ попадает в корзину, в которой уже лежат как минимум восемь других ключей, библиотека преобразует связный список для данной корзины в бинарное сбалансированное поисковое дерево. Получается гибридная структура: корзины для тех хеш-значений, где ключей мало, хранятся списками, а корзины, где ключей накопилось много, хранятся в виде деревьев.

3.8.6. Хеш-таблицы в Python

Рассмотрим реализацию языка программирования CPython версии 2.7.

Встроенный тип `dict` — ассоциативный массив, словарь — очень широко используется в языке. Он реализован в виде хеш-таблицы, где коллизии разрешаются методом открытой адресации. Разработчики предпочли метод открытой адресации методу цепочек ввиду того, что он позволяет значительно сэкономить память на хранении указателей, которые используются в хеш-таблицах с цепочками.

Подробное описание принципов устройства словаря в Python на русском языке можно найти в интернете.

Интерпретатором CPython поддерживается опция командной строки `-R`, которая активирует на старте случайный выбор начального значения (англ. *seed*), которое затем используется для вычисления хеш-значений от строк и массивов байт.

3.8.7. Атаки против стандартных хеш-функций

Для целочисленной хеш-функции, которая используется в Java, оказалось нетрудно придумать обратную, поскольку функция выполняет линейное преобразование битов исходного ключа. С использованием этого результата можно строить наборы различных ключей, которые при добавлении в `HashSet` попадают в одну корзину.

Более того, на одних и тех же входных данных может наблюдаться деградация производительности сразу нескольких реализаций хеш-таблиц. Так, в рамках соревнования IPSC 2014 была предложена следующая задача. Дано две программы. Первая программа на языке C++ просто читает числа на входе и добавляет их по одному в множество на основе `std::unordered_set`. Вторая программа на языке Java полностью аналогична, выполняет чтение чисел и занесение их в `HashSet`. Требовалось построить такую последовательность из не более чем 50 тысяч 64-битных целых чисел, чтобы время работы обоих программ превысило 10 секунд на сервере организаторов (указывались особенности архитектуры и версии компиляторов обоих языков программирования). Задача была решена многими участниками, т. е. им удалось подобрать такой набор входных данных, что построение обеих хеш-таблиц заняло квадратичное время из-за коллизий.

Существует целый класс атак на серверы, называемый *hash-flooding DoS* — отказ в обслуживании, вызванный заполнением хеша. Злоумыш-

ленники формируют специальные запросы, которые вызывают на сервере большое число хеш-коллизий и оттого медленно обрабатываются. В результате вычислительные ресурсы тратятся впустую, легальные пользователи системы не могут получить доступ к ресурсам либо этот доступ затруднён.

3.8.8. Криптографические хеш-функции

Для приложений в области криптографии разработаны специальные хеш-функции. Можно считать, что в криптографии множество K возможных ключей бесконечно, и любой блок данных является ключом (в принципе, произвольный массив байт можно рассматривать как двоичную запись некоторого числа). Хеш-функция $h(x)$ называется криптографической, если она удовлетворяет следующим требованиям:

- *необратимость*: для заданного значения хеш-функции c должно быть сложно определить такой ключ x, для которого $h(x) = c$;
- *стойкость к коллизиям первого рода*: для заданного ключа x должно быть вычислительно невозможно подобрать другой ключ y, для которого $h(x) = h(y)$;
- *стойкость к коллизиям второго рода*: должно быть вычислительно невозможно подобрать пару ключей x и y, имеющих одинаковый хеш.

Криптографические хеш-функции обычно не используются в хеш-таблицах, потому что они сравнительно медленно вычисляются и имеют большое множество значений. Зато такие хеш-функции широко применяются в системах контроля версий, системах электронной подписи, во многих системах передачи данных для контроля целостности.

Примерами криптографических хеш-функций являются алгоритмы MD5, SHA-1, SHA-256. Так, метод SHA-1 ставит в соответствие произвольному входному сообщению некоторую 20-байтную величину, т. е. результат вычисления SHA-1 принимает одно из 2^{160} различных значений. Вот пример вычисления SHA-1 от ASCII-строки, где результат записан в шестнадцатеричной системе счисления:

```
SHA-1("The quick brown fox jumps over the lazy dog")
  = 0x2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
```

В настоящий момент коллизии для MD5 и SHA-1 обнаружены, поэтому методы постепенно выходят из широкого использования. Более

новые алгоритмы семейства SHA-2 считаются существенно более стойкими к коллизиям. Тем не менее следует понимать, что коллизии есть обязательно, потому что нельзя биективно отобразить бесконечное множество в конечное. Вопрос только в том, насколько трудно эти коллизии отыскать.

3.9. ВЫВОДЫ

Таким образом, рассмотрены подходы к реализации динамических множеств и ассоциативных массивов на основе хеш-таблиц в теории и на практике.

Как уже говорилось, альтернативой хеш-таблицам являются структуры на основе сбалансированных бинарных поисковых деревьев. Возникает вопрос, как теоретический, так и практический, о том, насколько эти способы отличаются, какой из них лучше или хуже. Ответ здесь такой: хеш-таблицы в некотором доказуемом смысле могут быть быстрее.

Интуитивно более-менее понятно, почему так: в случае хеш-таблицы мы работаем с ключами, которые являются просто числами, причём на этих числах не предполагается никакого порядка, при этом интерфейс не может данный порядок воспроизвести. Например, нельзя найти все ключи от a до b, просто потому что на ключах порядка как такового нет. Конечно, если ключи — это числа, то порядок возникает, но никакой семантики не несёт. Мы отказываемся от требования упорядоченности, потому можем сделать структуру данных более эффективной.

Так, если к исходному интерфейсу множества (см. раздел 2.6) добавить операцию типа LowerBound(x) — найти первый ключ, больший либо равный x, то получится уже не просто множество (англ. *set*), а упорядоченное множество (англ. *ordered set*). Новая операция осуществляет поиск в окрестности значения x. На первый взгляд единственный доступный нам приём все эти операции поддерживать — это использовать деревья поиска. Оказывается, нет. Существуют интересные гибриды, находящиеся посередине между деревьями поиска и хеш-таблицами, они в частности реализуют концепцию упорядоченного множества. Это всевозможные деревья Ван Эмде Боасса (Van Emde Boas tree), X-fast-, Y-fast- и Fusion-деревья, у которых в оценках временной сложности появляется двойной логарифм.

Часть 4

СТРУКТУРЫ ДАННЫХ ДЛЯ РЕШЕНИЯ ЗАДАЧ НА ИНТЕРВАЛАХ

Когда на практике требуется хранить в памяти последовательность элементов и обращаться к одиночным элементам этой последовательности по индексу, обычно используют массив фиксированного размера или динамический массив. Эти структуры данных просты и эффективны.

Если нужно выполнить какую-либо произвольную операцию над k элементами (например, проверить, сколько из них удовлетворяют некоторому свойству), то в программе очевидным образом появляется цикл, пробегающий по k индексам и обращающийся к каждому элементу за $O(1)$, в итоге такая операция работает суммарно за время $\Omega(k)$ — требует выполнения не менее чем k обращений, плюс ещё время обработки самих элементов. В самом деле, это фундаментальное ограничение: не имея дополнительной информации, мы вынуждены просмотреть все k элементов и обработать каждый из них в отдельности, чтобы полностью выполнить операцию в общем случае.

Однако для конкретных задач иногда возможно разработать специальные структуры данных, которые позволяют выполнять групповые операции быстрее, т. е. обрабатывать один запрос, который сразу затрагивает k элементов, за время меньшее, чем $\Omega(k)$. В этой части мы рассмотрим примеры таких структур данных.

4.1. ПОСТАНОВКА ЗАДАЧИ О СУММЕ

Для простоты ограничимся случаем, когда k элементов, участвующих в операции, стоят в исходной последовательности рядом, т. е. образуют подпоследовательность подряд идущих элементов. Для обозначения

такого объекта будем использовать понятие *интервала* (англ. *interval*). Интервал задаётся двумя индексами его концов (в зависимости от типа интервала концы включаются или не включаются). Длина интервала может быть произвольной: от пустого интервала до интервала, захватывающего всю последовательность.

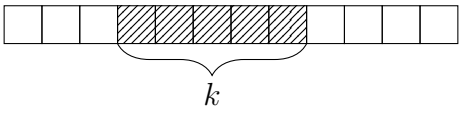


Рис. 4.1. Интервал длины k

Отметим, что вокруг понятия интервала в математике есть заметная терминологическая путаница (как в русскоязычной, так и в иностранной литературе). Некоторые авторы под интервалом подразумевают только открытый интервал (без включения концов). Другие авторы используют термин «промежуток» вместо слова «интервал». Кроме того, в некоторых книгах вводится понятие «отрезка» или «сегмента» для обозначения замкнутого интервала (концы включены). Чтобы не возникало неоднозначности, говоря об элементах массива, мы будем использовать понятие «интервал» и явно уточнять, включаются ли его концы в рассмотрение. Термин «отрезок» применять не будем, так как он имеет скорее геометрический оттенок.

Часто при программировании удобно работать с полуинтервалами, у которых левый конец включается, а правый исключается. Будем использовать следующее обозначение:

$$[l, r) = \{l, l + 1, ..., r - 1\}.$$

В отличие от ранее рассмотренных популярных структур, структуры данных для решения задач на интервалах обычно сильно завязаны на предметную область, плохо обобщаются и потому не представлены в стандартных библиотеках. Это создаёт дополнительные трудности для программиста. Раз использовать готовые реализации не получается, очень важно изучить подходы, идеи и алгоритмы, лежащие в основе построения таких структур данных, и уметь применить их при необходимости в конкретном практическом случае.

Рассмотрим следующую модельную задачу. Изначально дана последовательность чисел A длины n (индексация с нуля):

$$a_0, a_1, a_2, \ldots, a_{n-1}.$$

Поступают запросы двух типов.

• *Запрос модификации.* Задан индекс i и число x. Нужно прибавить к i-му элементу число x.

• *Запрос суммы.* Задана пара индексов l и r. Нужно вычислить сумму элементов на полуинтервале $[l, r)$, т. е. $a_l + a_{l+1} + \ldots + a_{r-1}$, и вернуть результат.

В этом случае запрос суммы называют *интервальным*, поскольку он затрагивает целый интервал значений.

4.2. НАИВНЫЙ ПОДХОД. ПОДСЧЁТ ПРЕФИКСНЫХ СУММ

Для хранения элементов будем использовать обычный массив. Запрос модификации выполняется за время $O(1)$, а вот запрос суммы требует линейного прохода по элементам. Поскольку длина интервала ограничена лишь размером массива, время выполнения запроса можно оценить как $O(n)$.

Практическая применимость такого решения обуславливается особенностями конкретной задачи. Если запросов первого типа много, а запросы второго типа редки, наивный подход будет работать хорошо.

Если посмотреть на проблему с другой стороны, можно получить альтернативное решение, которое даёт нам быстрое суммирование, но медленную модификацию.

Введём понятие *частичной суммы*, или *суммы на префиксе длины* k:

$$s_k = \sum_{i=0}^{k-1} a_i = a_0 + a_1 + \ldots + a_{k-1}.$$

По данной последовательности A массив префиксных сумм S можно построить за время $O(n)$, используя нехитрое рекуррентное соотношение:

$$\begin{aligned} s_0 &:= 0, \\ s_i &:= s_{i-1} + a_{i-1}, \quad i = 1, \ldots, n. \end{aligned}$$

Нетрудно видеть, что, исходя из свойств алгебраической операции сложения, сумма элементов массива a на полуинтервале $[l, r)$ равна разности двух префиксных сумм s_r и s_l.

Пример 4.1. Рассмотрим последовательность $(2, 1, 9, 7, 5, 2, 6)$.

a_i	2	1	9	7	5	2	6	
	0	1	2	3	4	5	6	7
s_i	0	2	3	12	19	24	26	32

Для суммы на полуинтервале $[2, 6)$ справедливы равенства:

$$\Sigma_{[2,6)} = a_2 + a_3 + a_4 + a_5 = 9 + 7 + 5 + 2 = 23;$$

$$\Sigma_{[2,6)} = s_6 - s_2 = 26 - 3 = 23.$$

Таким образом, при наличии массива S выполнение любого запроса суммы на полуинтервале сводится к вычислению разности двух чисел — это $O(1)$. Запрос модификации выполняется сложнее: раз i-й элемент участвует в префиксных суммах s_{i+1}, s_{i+2}, …, s_n, то к каждой из этих сумм придётся прибавить число x. Значит, такой запрос выполняется за время $O(n)$. Отметим, что такое простое решение прекрасно работает в случае, когда запросов модификации мало или нет вовсе.

Итак, используя лишь структуру данных «массив», мы можем решать задачу таким образом, что одна из операций выполняется за константу, а другая — за линейное время. Возникает логичный вопрос: можно ли разработать алгоритм, который даёт время лучше линейного сразу для обеих операций? Первоначально кажется, что линейного прохода не избежать, но на самом деле это не так, ответ на вопрос положительный. Мы далее рассмотрим две интересные идеи решения поставленной задачи. Идеального метода не существует, каждое решение предполагает определённый компромисс между разными аспектами (табл. 4.1).

Таблица 4.1

Сравнение различных методов решения

Подход	Время на запрос модификации	Время на запрос суммы	Время на предпросчёт	Объём доп. памяти
Обычный массив	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Префиксные суммы	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Sqrt-декомпозиция	$O(1)$	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n})$
Дерево отрезков	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$

4.3. SQRT-ДЕКОМПОЗИЦИЯ

В наивном подходе узким местом является операция подсчёта суммы: выполняется проход по всем элементам интервала. Возникает такая идея: можно разбить весь массив на блоки, посчитать заранее суммы для целых блоков; затем при суммировании в цикле можно будет «перепрыгивать» блоки, сумма для которых уже известна (рис. 4.2).

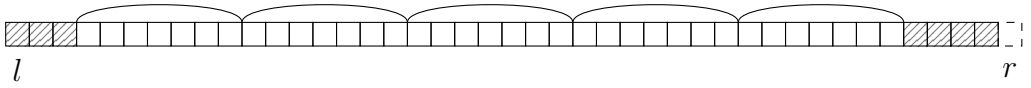


Рис. 4.2. Обработка запроса суммы на $[l, r)$

Пусть исходный массив A имеет размер n, индексы начинаются с нуля. Этот массив естественным образом разбивается на $\lceil n/k \rceil$ блоков размера k (последний блок, возможно, будет неполным). Блоки также удобно нумеровать с нуля, при этом элемент i массива относится к блоку с индексом $\lfloor i/k \rfloor$. Для хранения сумм по блокам нам понадобится дополнительный массив B. Элемент b_j вычисляется так:

$$b_j = a_{j \cdot k} + a_{j \cdot k+1} + \ldots + a_{j \cdot k+k-1} \tag{4.1}$$

(если какой-либо индекс выходит за границы массива A, полагаем слагаемое равным нулю).

Изначально, когда структура данных инициализируется, значения массива B вычисляются по формуле (4.1) исходя из значений в массиве A (рис. 4.3).

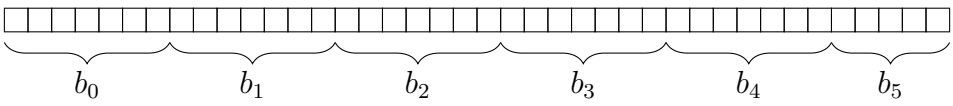


Рис. 4.3. Расчёт сумм по блокам при $n = 40$ и $k = 7$

Затем, когда приходит запрос модификации и нужно прибавить к i-му элементу число x, мы делаем две операции: увеличиваем a_i на x, увеличиваем $b_{\lfloor i/k \rfloor}$ также на x. Таким образом поддерживается согласованность массивов A и B. Операции выполняются за время $O(1)$.

Запрос суммы на полуинтервале $[l, r)$ можно обрабатывать так.

Для начала определим индексы блоков, к которым соответствуют границы интервала:

$$j_l := \lfloor l/k \rfloor,$$
$$j_r := \lfloor r/k \rfloor.$$

Если эти числа оказались равными, то обе границы попали в один блок, а значит, сумму можно вычислить напрямую, используя только массив A. На это будет затрачено время $\mathrm{O}(k)$. Если же $j_l < j_r$, то итоговая сумма будет складываться из трёх частей.

1. «Хвост» j_l-го блока: слагаемые от a_l (включительно) до $a_{(j_l+1)\cdot k}$ (не включительно). Вычисляется циклом по массиву A за время $\mathrm{O}(k)$.

2. Промежуточные блоки, которые попадают в интервал целиком: слагаемые от b_{j_l+1} (включительно) до b_{j_r} (не включительно). Вычисляется циклом по массиву B за время $\mathrm{O}(n/k)$.

3. «Голова» j_r-го блока: слагаемые от $a_{j_r \cdot k}$ (включительно) до a_r (не включительно). Вычисляется циклом по массиву A за время $\mathrm{O}(k)$.

Итого общее время выполнения такого запроса будет равно

$$T(n) = \mathrm{O}(k) + \mathrm{O}\left(\frac{n}{k}\right) + \mathrm{O}(k) = \mathrm{O}\left(k + \frac{n}{k}\right).$$

Каким лучше выбрать размер блока k? Точный ответ зависит от конкретных констант, скрытых в асимптотике, но можно попробовать исследовать функцию, стоящую под O, на экстремум средствами математического анализа.

Действительно, у функции

$$f(k) = k + \frac{n}{k}$$

производная имеет вид

$$f'(k) = 1 - \frac{n}{k^2}.$$

Приравнивая это выражение к нулю, получаем, что в точке $k = \sqrt{n}$ достигается минимум, равный $2\sqrt{n}$ (рис. 4.4).

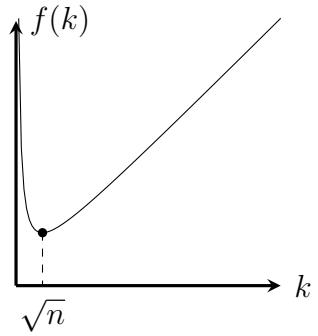


Рис. 4.4. График функции

Таким образом, оптимально разбивать на блоки размера примерно по корню квадратному из n каждый. При этом время обработки запроса тоже будет величиной порядка $\sqrt{n}$. Поэтому такой приём называют *Sqrt-декомпозицией*, или *корневой эвристикой*.

Реализуем в псевдокоде описанную структуру данных.

```
class Summator:
    def __init__(self, a):
        self.a = a
        self.k = floor(sqrt(len(a)))

        self.b = []
        for i in range(0, len(a), self.k):
            bsum = sum(self.a[i:(i + self.k)])
            self.b.append(bsum)

    def Add(self, i, x):
        self.a[i] += x
        self.b[i // self.k] += x

    def FindSum(self, l, r):
        jl = l // self.k
        jr = r // self.k
        if jl == jr:  # same block
            return sum(self.a[l:r])
        else:
            return (
                sum(self.a[l:((jl + 1) * self.k)]) +
                sum(self.b[(jl + 1):jr]) +
                sum(self.a[(jr * self.k):r])
            )
```

4.4. ДЕРЕВО ОТРЕЗКОВ

Дальнейшим развитием идеи разбиения на блоки будет следующее: можно использовать блоки разной длины и организовать их в виде бинарного дерева.

Пусть каждая вершина дерева представляет некоторый отрезок (или, для удобства, полуинтервал) элементов исходного массива A. Корень дерева соответствует всему массиву — полуинтервалу $[0, n)$. Далее, если какая-либо вершина связана с полуинтервалом $[t_l, t_r)$, причём $t_r - t_l > 1$, то у этой вершины будет два потомка: один отвечает за часть $[t_l, m)$, второй — за часть $[m, t_r)$, где $m = \lfloor (t_l + t_r)/2 \rfloor$ (рис. 4.5). Листьями дерева будут вершины, для которых полуинтервал имеет вид $[x, x + 1)$ и содержит всего один элемент.

Такое дерево и называют *деревом отрезков*. В англоязычной традиции нет устоявшегося термина для этой структуры данных. Можно встретить вариант *segment tree*, но в литературе это словосочетание

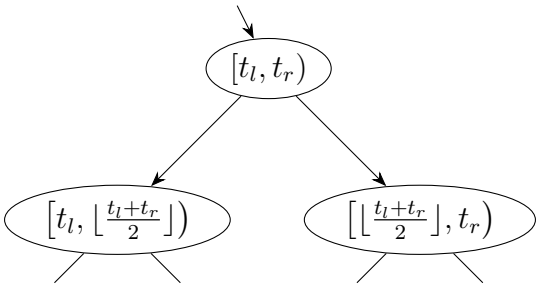


Рис. 4.5. Фрагмент дерева отрезков

часто употребляется в другом смысле: для обозначения структуры данных, которая работает с геометрическими отрезками, а вовсе не с отрезками массива, и позволяет для точки определять, какие отрезки её содержат.

Существует также понятие «дерево интервалов» (англ. *interval tree*): обычно оно относится к совсем другой структуре данных, которая здесь рассматриваться не будет.

4.4.1. Организация

Пример дерева отрезков показан на рис. 4.6. Из рисунка нетрудно видеть, что дерево отрезков в общем случае не является полным (уровни могут быть заполнены не целиком). Однако, тем не менее, высота этого дерева имеет порядок $\mathrm{O}(\log n)$, число листьев равно n, каждый лист однозначно соответствует элементу исходного массива. Более того, общее число вершин в дереве есть величина порядка $\mathrm{O}(n)$.

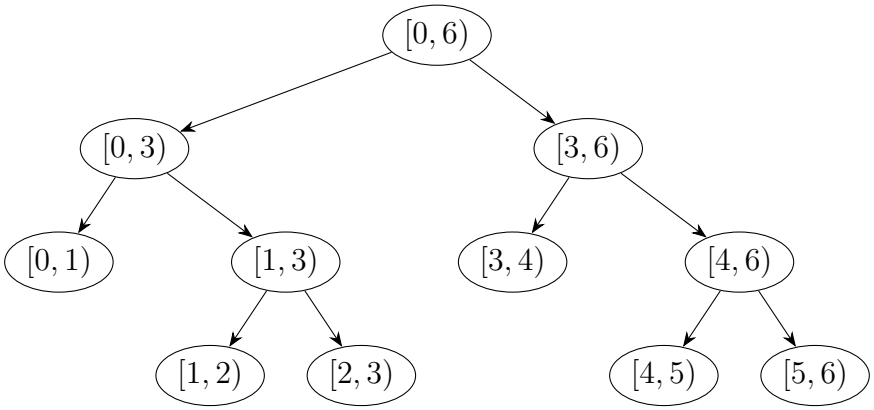


Рис. 4.6. Дерево отрезков, построенное для шести элементов

Теорема 4.1. *В дереве отрезков, которое построено для последовательности из n элементов, число вершин равно $2n - 1$.*

Доказательство. Применим метод математической индукции.

Если $n = 1$, то по формуле получим $2n - 1 = 1$. Действительно, дерево отрезков состоит из одной вершины, поэтому утверждение верно.

Если $n > 1$, то полуинтервал $[0, n)$ разобьётся на два подинтервала $[0, m)$ и $[m, n)$. Для каждого подинтервала будет независимо построено дерево отрезков, затем будет добавлена вершина для $[0, n)$, объединяющая два этих дерева. Общее число вершин будет равно

$$(2m - 1) + (2(n - m) - 1) + 1 = 2n - 1.$$

Итого, в дереве ровно n листьев и $n - 1$ внутренняя вершина. □

В каждой вершине дерева отрезков содержится сумма элементов исходного массива, индексы которых принадлежат соответствующему отрезку. Так, в корне дерева лежит общая сумма всех элементов массива. В листьях дерева находятся сами элементы.

4.4.2. Хранение в памяти

Дерево отрезков является частным случаем бинарного дерева, поэтому можно организовать хранение дерева отрезков в памяти компьютера с помощью указателей или ссылок. Однако более рационально хранить все элементы дерева отрезков в массиве и задавать структуру дерева неявно через индексы, как это делается для бинарной кучи (см. раздел 5.1.1): индексация в массиве начинается с единицы, у вершины с индексом v левый сын имеет индекс $2v$, правый сын — $2v + 1$. Таким образом, каждая вершина хранится в виде одного числа.

В случае, когда n не степень двойки, бинарное дерево не является полным, из-за чего в массиве могут быть неиспользуемые ячейки. Чтобы места гарантированно хватило, можно выставить размер массива равным $4n$, или же оценить нужный размер более точными формулами.

Пример 4.2. Если дерево отрезков строится для последовательности длины $n = 1025$, то в массиве, в котором хранятся вершины дерева, будут использоваться индексы вплоть до $v = 4095$.

Пример 4.3. Для массива из $n = 6$ элементов $(9, 2, 5, 3, 6, 1)$ дерево отрезков, показанное на рис. 4.7, в виде массива будет храниться так:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
26	16	10	9	7	3	7	—	—	2	5	—	—	6	1

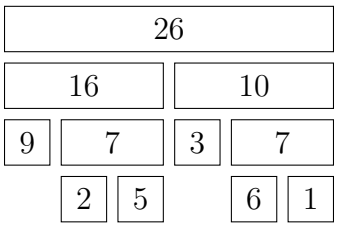


Рис. 4.7. Условное изображение дерева отрезков для массива $(9, 2, 5, 3, 6, 1)$

Обратите внимание, что в примере при $n = 6$ количество реально занятых ячеек равно 11 (т. е. $2n - 1$), но лишняя память расходуется.

4.4.3. Построение дерева отрезков

Процесс построения дерева отрезков по заданному массиву A удобно описывать рекурсивно.

Пусть мы находимся в вершине дерева отрезков с индексом v, она соответствует некоторому полуинтервалу $[t_l, t_r)$. Если вершина является листом, т. е. $t_r = t_l+1$, то в вершину дерева заносим сам элемент массива $A[t_l]$. Иначе определяем границу m, вызываем функцию рекурсивно на $[t_l, m)$ для левого поддерева (его корень имеет индекс $2v$) и на $[m, t_r)$ для правого поддерева (его корень с индексом $2v+1$), агрегируем суммы потомков и получаем финальное значение для текущей вершины.

```
def DoBuild(a, t, v, tl, tr):
    if tr - tl == 1:
        t[v] = a[tl]
    else:
        m = (tl + tr) // 2
        DoBuild(a, t, v=2*v, tl=tl, tr=m)
        DoBuild(a, t, v=2*v+1, tl=m, tr=tr)
        t[v] = t[2*v] + t[2*v+1]
```

Процедура запускается от корня дерева следующим образом:

```
def Build(a, n):
    t = [None] * 4*n  # array of size 4n
    DoBuild(a, t, v=1, tl=0, tr=n)
    return t
```

Теорема 4.2. *По массиву из n элементов дерево отрезков строится за* $\mathrm{O}(n)$.

Доказательство. Действительно, число вершин в дереве линейное, благодаря стеку рекурсии значения вычисляются снизу вверх, для каждой вершины значение определяется за $\mathrm{O}(1)$ на основании значений вершин-потомков. □

4.4.4. Запрос модификации

Реализуем рекурсивную функцию, которая будет выполнять прибавление числа x к элементу с индексом i. Предположим, что мы находимся в вершине дерева отрезков с индексом v, она соответствует некоторому полуинтервалу $[t_l, t_r)$.

Если текущая вершина — лист, то достаточно прибавить x к числу, которое хранится в этой вершине. Иначе определяем границы полуинтервалов, которые соответствуют двум потомкам текущей вершины: $[t_l, m)$ и $[m, t_r)$. Если $i < m$, то выполняем рекурсивный вызов для левого сына, иначе — для правого сына. Наконец, пересчитаем значение суммы в текущей вершине точно таким же образом, как делали это при первоначальном построении дерева отрезков.

```
def DoAdd(t, v, tl, tr, i, x):
    if tr - tl == 1:
        t[v] += x
        return
    m = (tl + tr) // 2
    if i < m:
        DoAdd(t, v=2*v, tl=tl, tr=m, i=i, x=x)
    else:
        DoAdd(t, v=2*v+1, tl=m, tr=tr, i=i, x=x)
    t[v] = t[2*v] + t[2*v+1]
```

Стартовый вызов рекурсивной функции делаем так:

```
def Add(t, n, i, x):
    DoAdd(t, v=1, tl=0, tr=n, i=i, x=x)
```

Теорема 4.3. *Время работы операции модификации одиночного элемента в дереве отрезков есть* $\mathrm{O}(\log n)$.

Доказательство. Заметим, что раз отрезки на каждом уровне не пересекаются (по построению), то i-й элемент входит в $\mathrm{O}(\log n)$ отрезков (не более чем в один отрезок на уровне). Поэтому для выполнения запроса модификации требуется обновить не более чем $\mathrm{O}(\log n)$ значений в дереве отрезков. □

4.4.5. Запрос суммы на интервале

Эта задача более сложная. Реализуем рекурсивную функцию, которая будет её решать. Пусть нужно вычислить сумму на полуинтервале $[l, r)$. Мы находимся в вершине дерева отрезков с индексом v, она соответствует некоторому полуинтервалу $[t_l, t_r)$.

Если $[l, r) = [t_l, t_r)$, то в качестве ответа сразу возвращаем предварительно вычисленное значение суммы, записанное в дереве.

В противном случае определяем границы полуинтервалов, соответствующих двум потомкам: $[t_l, m)$ и $[m, t_r)$. Если $[l, r) \subseteq [t_l, m)$, то рекурсивно вызываем функцию для левого потомка (эта вершина дерева имеет индекс $2v$), не изменяя границы запроса. Аналогичную проверку $[l, r) \subseteq [m, t_r)$ делаем для правого потомка (он имеет индекс $2v + 1$).

Если же запрос $[l, r)$ пересекается с полуинтервалами обоих потомков, то нет другого выхода, кроме как перейти в левого потомка и вычислить ответ для фрагмента $[l, m)$, перейти в правого потомка и вычислить ответ для фрагмента $[m, r)$, затем просуммировать оба ответа.

```
def DoFindSum(t, v, tl, tr, l, r):
    if l == tl and r == tr:
        return t[v]
    m = (tl + tr) // 2
    if r <= m:
        return DoFindSum(t, v=2*v, tl=tl, tr=m, l=l, r=r)
    if m <= l:
        return DoFindSum(t, v=2*v+1, tl=m, tr=tr, l=l, r=r)
    return (
        DoFindSum(t, v=2*v, tl=tl, tr=m, l=l, r=m) +
        DoFindSum(t, v=2*v+1, tl=m, tr=tr, l=m, r=r)
    )
```

Таким образом, алгоритм представляет собой спуск по дереву отрезков, который посещает нужные ветви и для быстрой работы использует уже посчитанные суммы, хранящиеся в дереве.

Чтобы вычислить ответ для полуинтервала $[l, r)$, нужно начать путь из корня дерева, вызывая функцию `DoFindSum` с правильными аргументами:

```
def FindSum(t, n, l, r):
    return DoFindSum(t, v=1, tl=0, tr=n, l=l, r=r)
```

Возникает вопрос о том, каково время работы описанного алгоритма. Если каждый раз рекурсия будет уходить в обе ветви, то общее время будет линейным. Однако на практике этого не происходит, и вот почему.

Теорема 4.4. *Время работы операции нахождения суммы на интервале есть* $O(\log n)$.

Доказательство. Предположим вначале, что левая граница полуинтервала, на котором нужно искать сумму, равна нулю, т. е. полу-

интервал захватывает начало массива и имеет вид $[0, r)$. На каждом шаге алгоритма возможна одна из трёх конфигураций, показанных на рис. 4.8.

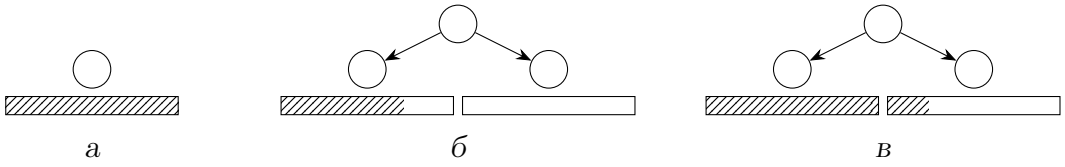


Рис. 4.8. Возможные ситуации на очередном шаге рекурсии для $[0, r)$: *а* — остановка; *б* — спуск только влево; *в* — спуск влево и вправо

В случае *а* отрезок, который представляется текущей вершиной, полностью совпадает с искомым, поэтому алгоритм прекращает работу и возвращает значение, записанное в вершине дерева для этого отрезка, за $O(1)$.

В случае *б* выполняется рекурсивный спуск только в левое поддерево, поскольку в правом поддереве нет интересующих нас элементов.

Наибольший интерес представляет случай *в*. Здесь рекурсия разветвляется, делаются два рекурсивных вызова. Однако левая рекурсивная цепочка остановится на следующем же шаге из-за того, что придёт к случаю *а*. Продолжит выполняться только правая цепочка.

Таким образом, можно заметить, что активно работает только одна ветвь рекурсии на каждом уровне. Раз число уровней имеет порядок логарифма, то общее время обработки запроса есть $O(\log n)$.

Аналогичные рассуждения можно провести для случая, когда интервал захватывает конец массива и имеет вид $[l, n)$.

Пусть теперь интервал, на котором ищется сумма, произвольный — $[l, r)$. Начиная рекурсивный спуск от корня, алгоритм всякий раз приходит к одной из конфигураций, показанных на рис. 4.9.

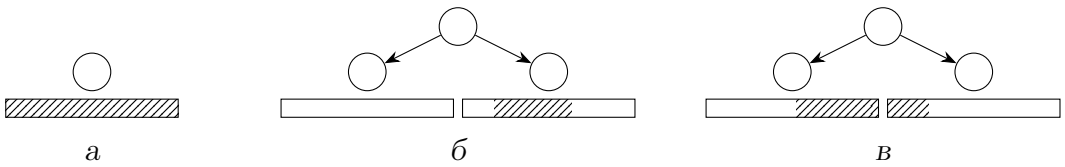


Рис. 4.9. Возможные ситуации на очередном шаге рекурсии для $[l, r)$: *а* — остановка; *б* — спуск в одну сторону; *в* — спуск в обе стороны

В случае *а*, как и ранее, работа прекращается. В случае *б* спуск продолжается только по одной из ветвей. Наконец, если через какое-то

время (не более чем через $O(\log n)$ шагов) достигнута конфигурация $в$, то дальнейшая задача сводится к двум независимым подзадачам описанного ранее вида. Эти подзадачи решаются двумя активно работающими рекурсивными вызовами. □

4.4.6. Обобщения

Можно также построить нерекурсивные реализации описанных операций. На практике код без рекурсии обычно работает быстрее. Кроме того, можно выполнять операции не «сверху вниз», а «снизу вверх», как описано в [5].

Выше рассматривалась задача, когда запрос модификации затрагивает единственный элемент массива. На самом деле дерево отрезков позволяет делать запросы, которые применяются к целым отрезкам подряд идущих элементов, причём выполнять эти запросы за то же время $O(\log n)$.

Концепция интервальных запросов очень гибкая и может быть распространена на многие другие задачи.

4.5. ЗАДАЧА RMQ. SPARSE TABLE

Операцию сложения в разделе 4.1 можно заменить на какую-либо иную операцию: взятие минимума, максимума и пр. Отдельное название в литературе получила задача поиска минимального значения на интервале — RMQ (англ. *range mininum query*) — в связи с её практической важностью. К этой задаче при определённых условиях сводится, например, задача поиска наименьшего общего предка в дереве — LCA (англ. *lowest common ancestor*).

Нетрудно заметить, что для решения задачи RMQ можно применить корневую эвристику или дерево отрезков.

4.5.1. Статическая версия RMQ

Предположим, что запросов модификации нет, все данные статичны. Есть некоторый постоянный массив A, состоящий из n чисел. Поставим задачу следующим образом: поступают запросы вида $[l, r)$, ответом на запрос является минимальное значение среди элементов массива $a_l, a_{l+1}, \ldots, a_{r-1}$. Нужно уметь отвечать на такие запросы за $O(1)$.

Очевидное решение — посчитать заранее ответы на всевозможные запросы и сохранить их в двумерном массиве, чтобы затем извлекать

за $O(1)$. Полная таблица T, где элемент $t_{i,j}$ содержит минимум на полуинтервале $[i, j)$, строится за время $\Theta(n^2)$ и занимает $\Theta(n^2)$ памяти. Эти расходы слишком велики. Оказывается, можно построить таблицу меньшего размера — «разрежённую» — так, что она будет занимать $O(n \log n)$ памяти, а время ответа на каждый запрос будет по-прежнему константным.

Отметим, что простой префиксный метод здесь не работает: если известен минимум в массиве на полуинтервале $[0, l)$ и минимум на полуинтервале $[0, r)$, о минимуме на $[l, r)$ в общем случае ничего сказать нельзя.

4.5.2. Sparse table

Sparse table (рус. *разрежённая таблица*, но этот термин пока не устоявшийся) представляет собой двумерную структуру, в которой хранятся минимумы на всех интервалах, длины у которых являются некоторыми степенями двойки. Другими словами, для каждой позиции i массива A будут подсчитаны минимумы на всех интервалах длины $1, 2, 4, \ldots$ вправо от i. Элемент $t_{i,k}$ таблицы определяется формулой

$$t_{i,k} = \min\{a_i, a_{i+1}, \ldots, a_{i+2^k-1}\}.$$

Нетрудно заметить, что нет смысла рассматривать значения k больше, чем $\lfloor \log_2 n \rfloor$. Для фиксированного k заполняются лишь ячейки, для которых $i + 2^k \leqslant n$. Тем не менее общий размер таблицы — $O(n \log n)$.

Для эффективного вычисления $t_{i,k}$ можно использовать следующее рекуррентное соотношение:

$$\begin{aligned} t_{i,0} &= a_i; \\ t_{i,k} &= \min\{t_{i,k-1}, t_{i+2^{k-1},k-1}\}, \quad k \geqslant 1. \end{aligned}$$

Пример 4.4. Для массива $(2, 9, 1, 9, 6, 7, 5, 2)$ из восьми элементов разрежённая таблица имеет следующий вид (для наглядности каждому k соответствует одна строка, каждому i — столбец):

k	2^k	0	1	2	3	4	5	6	7
0	1	2	9	1	9	6	7	5	2
1	2	2	1	1	6	6	5	2	
2	4	1	1	1	5	2			
3	8	1							

Допустим, таблица сформирована, тогда ответ на запрос минимума в массиве на полуинтервале $[l, r)$ будет вычисляться следующим образом.

Найдём максимальную степень двойки p, которая не превосходит длины заданного полуинтервала:

$$2^p \leqslant r - l < 2^{p+1}.$$

Для этого можно использовать следующую формулу:

$$p = \lfloor \log_2(r - l) \rfloor.$$

На практике, чтобы избежать операций с плавающей точкой при вычислении логарифма, можно заранее просчитать значения p для всевозможных длин при помощи простого рекуррентного соотношения и сохранить их в таблицу. Так или иначе, число p можно найти за $O(1)$.

Пример 4.5. Полуинтервал $[3, 8)$ имеет длину 5, значит, $p = 2$.

Далее, нетрудно показать, что полуинтервал $[l, r)$ может быть покрыт не более чем двумя полуинтервалами, длина которых равна 2^p. Возможно, эти интервалы будут пересекаться.

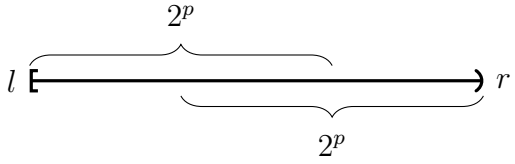


Рис. 4.10. Покрытие интервала $[l, r)$ двумя интервалами длины 2^p

Искомый минимум на $[l, r)$ равен минимуму из двух минимумов: на $[l, l + 2^p)$ и $[r - 2^p, r)$. Эти два минимума посчитаны заранее и хранятся в sparse table, значит, можно получить ответ на запрос за время $O(1)$:

$$\min\{t_{l,p}, t_{r-2^p,p}\}.$$

Отметим, что описанный подход можно применить к операции максимума, но нельзя, например, к операции сложения, так как сумма сумм на двух перекрывающихся подынтервалах не равна сумме на целом интервале.

Часть 5

СПЕЦИАЛЬНЫЕ СТРУКТУРЫ ДАННЫХ

5.1. БИНАРНАЯ КУЧА

Куча (англ. *heap*) — специализированная древовидная структура данных, которая удовлетворяет *свойству кучи*. В вершинах древовидной структуры хранятся ключи. Различают два варианта куч: *min-heap* и *max-heap*. Для min-heap свойство кучи формулируется следующим образом: если вершина с ключом y является потомком вершины с ключом x, то $x \leqslant y$. Когда рассматривается max-heap, знак в неравенстве меняется на противоположный. Для определённости в этом разделе будем работать с первым вариантом — min-heap.

Для куч определён следующий базовый набор операций:

1) GetMin() — поиск минимального ключа;
2) ExtractMin() — удаление минимального ключа;
3) Insert(x) — добавление ключа x.

Расширенный набор операций включает также следующие:

• IncreaseKey и DecreaseKey — модификация ключа вершины на заданную величину (предполагается, что известна позиция вершины внутри структуры данных);

• Heapify — построение кучи для последовательности из n ключей.

Существует много способов реализации структуры данных «куча» с помощью корневых деревьев. Наиболее простым способом является реализация с помощью *полного бинарного дерева*. Такая куча называется *бинарной кучей* (англ. *binary heap*), или *пирамидой*.

Напомним, что полное бинарное дерево — это такое корневое дерево, в котором каждая вершина имеет не более двух сыновей, а заполнение вершин осуществляется в порядке от верхних уровней к нижним, причём на одном уровне заполнение вершинами производится слева

направо. Пока уровень полностью не заполнен, к следующему уровню не переходят. Последний уровень может быть заполнен не полностью. Высота полного бинарного дерева, содержащего n вершин, — $O(\log n)$.

5.1.1. Представление в памяти

В памяти компьютера полное бинарное дерево легко реализуется с помощью массива (индексы массива начинаются с единицы). Для элемента с индексом i сыновьями являются элементы с индексами $2i$ и $2i + 1$, а родителем является элемент массива по индексу $\lfloor i/2 \rfloor$.

Пример 5.1. На рис. 5.1 приведён пример полного бинарного дерева из десяти вершин.

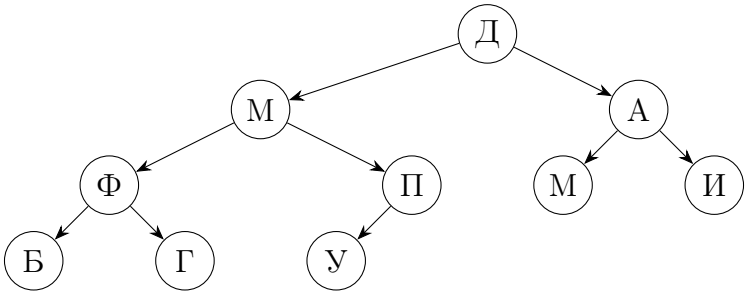


Рис. 5.1. Полное бинарное дерево

В памяти компьютера указанное полное бинарное дерево будет храниться в массиве следующим образом:

1	2	3	4	5	6	7	8	9	10
Д	М	А	Ф	П	М	И	Б	Г	У

Пример 5.2. На рис. 5.2 приведён пример бинарной кучи.

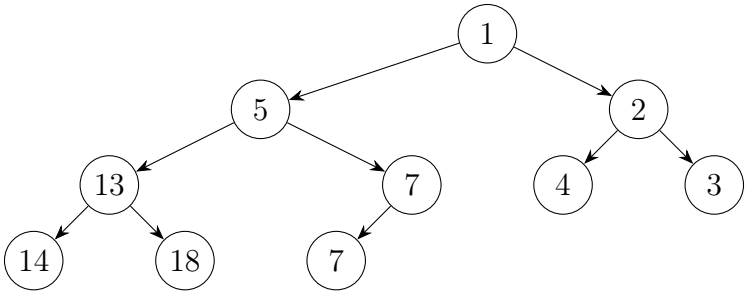


Рис. 5.2. Бинарная куча (min-heap)

В памяти компьютера эта бинарная куча будет храниться в массиве следующим образом:

1	2	3	4	5	6	7	8	9	10
1	5	2	13	7	4	3	14	18	7

Кроме массива, который используется для хранения ключей бинарной кучи, вводится переменная целого типа, определяющая количество элементов в куче.

5.1.2. Основные операции

Время выполнения базовых операций для бинарной кучи, содержащей n вершин, следующее:

- GetMin() — $O(1)$;
- ExtractMin() — $O(\log n)$;
- Insert(x) — $O(\log n)$.
- IncreaseKey и DecreaseKey — $O(\log n)$;
- Heapify — $O(n)$.

Более подробно с описанием процедур, реализующих интерфейс структуры данных «бинарная куча», можно ознакомиться, например, в [3].

5.1.3. Псевдокод

Опишем на псевдокоде основные операции, которые можно выполнять с бинарной кучей (для определённости min-heap). Массивы в псевдокоде индексируются с нуля, как во всех распространённых языках программирования, поэтому требуется определить, как отношения между элементами связаны с их индексами. Для перехода от 1-индексации к 0-индексации в формулы, указанные в начале раздела 5.1.1, вместо i подставим $i' = i + 1$, затем из результата вычтем 1.

- Сыновьями элемента i являются элементы с индексами

$$2(i+1) - 1 = 2i + 1, \quad 2(i+1) + 1 - 1 = 2i + 2.$$

- Родителем элемента i является элемент

$$\lfloor (i+1)/2 \rfloor - 1 = \lfloor (i-1)/2 \rfloor.$$

Пусть в коде `a` — динамический массив, который используется для хранения кучи (индексы начинаются с 0). Тогда `len(a)` — число элементов в этом массиве. Изначально массив пуст.

Операция Insert(x) работает следующим образом. Ключ добавляется в конец массива. Возможно, при этом свойство кучи нарушится, поэтому нужно восстановить это свойство, выполнив проталкивание элемента вверх. Каждый раз ключ текущего элемента с индексом i сравнивается с ключом его родителя с индексом j, при необходимости выполняется обмен ключей.

```
def Insert(a, x):
    a.append(x)

    i = len(a) - 1
    while i > 0:
        j = (i - 1) // 2  # a[j] is the parent of a[i]
        if a[j] <= a[i]:
            break
        a[i], a[j] = a[j], a[i]  # swap
        i = j
```

Операция получения минимума GetMin() тривиальна: минимум находится в корне дерева, нужно просто вернуть его.

```
def GetMin(a):
    return a[0]
```

Операция извлечения минимума ExtractMin() работает так. Берём последний элемент (крайний правый элемент на нижнем уровне дерева) и ставим его в корень. Возможно, при этом свойство кучи нарушится, поэтому нужно восстановить это свойство, выполнив проталкивание элемента вниз по дереву. Каждый раз для элемента с ключом i определяется индекс потомка j, ключ которого наименьший, и при необходимости выполняется обмен.

```
def ExtractMin(a):
    a[0] = a[len(a) - 1]
    a.pop()

    i = 0
    while 2 * i + 1 < len(a):
        if (2 * i + 2 == len(a)) or (a[2 * i + 1] < a[2 * i + 2]):
            j = 2 * i + 1  # left child
        else:
            j = 2 * i + 2  # right child
        if a[i] <= a[j]:
            break
        a[i], a[j] = a[j], a[i]  # swap
        i = j
```

На практике бинарную кучу редко приходится реализовывать самостоятельно, поскольку готовые решения есть в стандартных библиотеках многих языков программирования. Однако важно понимать, как именно устроена эта структура данных.

Классы из разных языков программирования, внутри которых скрыты бинарные кучи, перечислены в разделе 2.5. Кроме того, в C++ STL доступна серия алгоритмов `std::make_heap`, `std::push_heap`, `std::pop_heap` и др. Эти функции позволяют построить кучу на базе любой последовательности элементов.

5.2. ДЕКАРТОВО ДЕРЕВО

Декартовым деревом (англ. *treap*) принято называть структуру данных, которая сочетает в себе свойства бинарной кучи (см. раздел 5.1) и бинарного поискового дерева [4]. У данной структуры существует несколько названий, каждое из которых объединяет понятия дерева и кучи (например, дерамида).

Другими словами, это такое бинарное дерево, в котором в каждой вершине хранятся пары значений $(key, priority)$, и по значениям key бинарное дерево является двоичным деревом поиска, а по значениям $priority$ является бинарной кучей (max-heap). Будем считать, что все ключи имеют попарно различные значения. Тогда, если рассмотреть вершину дерева $(key, priority)$, то у всех элементов в левом поддереве значение ключа будет строго меньше key, в правом — строго больше. В то же время, значение всех приоритетов в левом и правом поддереве будет не больше $priority$ (рис. 5.3).

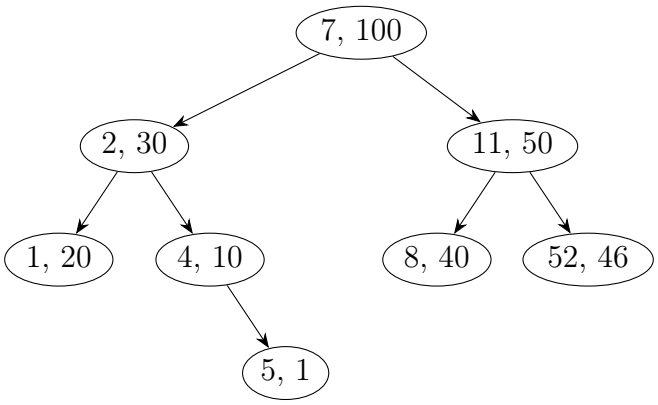


Рис. 5.3. Декартово дерево

Существует несколько основных операций в декартовом дереве: Split (разрезание дерева), Merge (слияние деревьев), Insert (добав-

ление новой вершины), REMOVE (удаление вершины из дерева). Каждая операция в данной структуре выполняется за время $O(h)$, где h — высота дерева. Далее рассмотрим каждую операцию по отдельности.

5.2.1. Операция разрезания

Операция SPLIT позволяет разделить начальное дерево T на два дерева L, R по некоторому ключу k так, что в первом дереве будут лежать все вершины с ключом меньше k, а во втором все остальные.

Рассмотрим случай, при котором необходимо разрезание по ключу, который больше, чем ключ корня (рис. 5.4). Определим, каким образом будут выглядеть два итоговых дерева (после разрезания).

Дерево L будет иметь в качестве своего левого поддерева левое поддерево дерева T. Для формирования правого поддерева сначала разобьём правое поддерево дерева T с помощью рекурсивного вызова функции SPLIT на два дерева L' и R' по ключу k. Тогда правое поддерево дерева L будет совпадать с L', а дерево R будет совпадать с деревом R'.

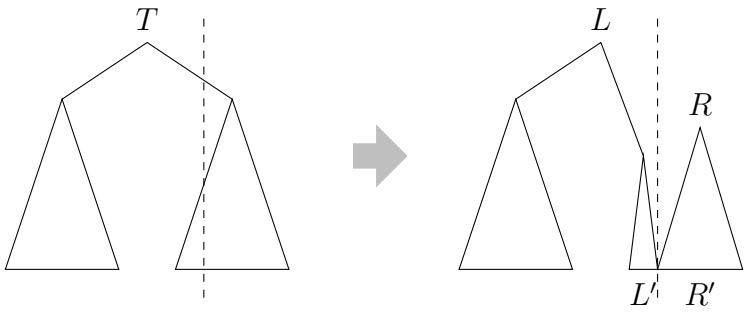


Рис. 5.4. Операция SPLIT

Аналогично рассматривается случай, когда значение в корне дерева T не превышает значения k.

```
def Split(T, k):
    if T is None:
        return (None, None)
    elif k >= T.key:
        T.right, R = Split(T.right, k)
        L = T
        return (L, R)
    else:
        L, T.left = Split(T.left, k)
        R = T
        return (L, R)
```

Во время выполнения операции Split рекурсивно вызывается разрезание для дерева высотой хотя бы на один меньше, поэтому в худшем случае будет сделано $O(h)$ вызовов, где h — высота дерева.

5.2.2. Операция слияния

Операция Merge позволяет слить два дерева L, R в одно декартово дерево T. Причём предполагается, что перед началом операции все ключи в первом дереве строго меньше ключей во втором дереве.

Так как в T корневая вершина должна обладать наибольшим приоритетом, то нетрудно заметить, что этой вершиной является или корневая вершина дерева L, или корневая вершина дерева R, поскольку они обладали наибольшими приоритетами в своих деревьях по определению. Рассмотрим случай, когда значение приоритета в корневой вершине L больше, чем значение приоритета в корневой вершине R (рис. 5.5). Тогда левое поддерево T будет совпадать с левым поддеревом L. Правым поддеревом T станет дерево, полученное в результате объединения правого поддерева дерева L и дерева R. Обратный случай рассматривается аналогично.

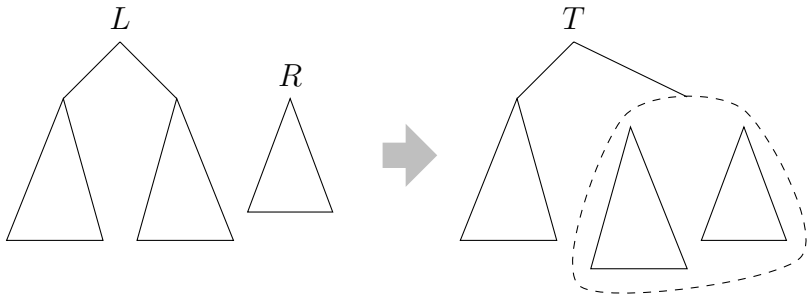


Рис. 5.5. Операция Merge

Приведём реализацию описанного алгоритма на псевдокоде.

```
def Merge(L, R):
    if L is None:
        return R
    elif R is None:
        return L
    elif L.priority > R.priority:
        L.right = Merge(L.right, R)
        return L
    else:
        R.left = Merge(L, R.left)
        return R
```

Аналогично методу Split доказывается, что в худшем случае будет сделано $O(h)$ вызовов функции Merge, где h — высота дерева.

5.2.3. Операция добавления новой вершины

Операция Insert позволяет вставить новую вершину в дерево, чтобы оно осталось корректным.

```
def Insert(T, key, priority):
    temporary = TreapNode(key, priority)
    if T is None:
        return temporary
    L, R = Split(T, key)
    result = Merge(L, temporary)
    result = Merge(result, R)
    return result
```

В ходе выполнения добавления нового элемента будут осуществлены одна операция Split и две операции Merge, поэтому операция Insert выполняется за время $O(h)$, где h — высота дерева.

5.2.4. Операция удаления вершины

Операция Delete позволяет удалить из дерева T вершину с заданным ключом k.

```
def Delete(T, key):
    if T is None:
        return None
    T1, T2 = Split(T, key)
    T2, T3 = Split(T2, key + 1)
    return Merge(T1, T3)
```

В ходе выполнения удаления элемента будут выполнены одна операция Merge и две операции Split, поэтому операция Delete выполняется за время $O(h)$, где h — высота дерева.

5.2.5. Применение

Можно доказать, что если назначать вершинам декартова дерева значения приоритетов случайными величинами с равновероятным распределением, то средняя высота дерева будет равна $O(\log n)$. Тогда любая из вышеописанных операций в среднем будет выполняться за логарифмическое время.

Фактически, если рассматривать декартово дерево как ассоциативный массив (см. раздел 2.7), то можно осуществлять операции добавле-

ния и удаления элементов за логарифмическое время. Если ключами выступают целые числа от 0 до $n-1$, то такая структура может служить заменой обычному массиву.

Более того, ключи можно не хранить в вершинах явно. Заметим, что фактически в данном случае ключ для какой-то вершины — это количество вершин, лежащих левее неё (такие вершины находятся не только в левом поддереве данной вершины, но и, возможно, в левых поддеревьях её предков). Поддерживая в каждой вершине число вершин в её поддереве, можно отказаться от хранения ключей. Таким образом получится *неявное декартово дерево*.

С помощью нетрудных рассуждений можно реализовать следующие операции, каждая из которых будет выполняться также за $O(\log n)$:

1) поиск максимума, минимума, суммы и т. д. на отрезке;
2) прибавление на отрезке, установка значения на отрезке;
3) перестановка некоторого отрезка подряд идущих элементов в обратном порядке.

Декартово дерево не является самобалансирующимся в обычном смысле, но применяется на практике по следующим весомым причинам.

1. Реализация проста по сравнению с самобалансирующимися деревьями (например по сравнению с красно-чёрным деревом).

2. Самая стандартная операция сортирующего дерева (разделение на два поддерева по определённому ключу) выполняется за время $O(h)$, где h — высота дерева. В более труднореализуемых структурах, например во всё том же красно-чёрном дереве, необходимо также восстанавливать балансировку после выполнения каждой операции.

Основным недостатком декартова дерева являются большие расходы на хранение информации: вместе с каждым элементом необходимо хранить несколько указателей и значение приоритета.

5.3. СИСТЕМА НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ

В некоторых задачах необходимо хранить разбиение какого-то набора уникальных объектов на непересекающиеся *динамические множества* (англ. *dynamic sets*). В каждом множестве выделен один из его элементов, который называют *представителем* (англ. *representative*). Представитель множества определяет данное множество, иногда в литературе вместо понятия «представитель» встречается *лидер*. В процессе перераспределения объектов по множествам представители могут ме-

няться, но важно, чтобы при выполнении операций представитель множества оставался одним и тем же, если множество за этот промежуток времени не менялось.

Ниже приведён пример системы непересекающихся множеств (для каждого множества представитель выделен полужирным шрифтом):

$$\{1, \mathbf{2}, 3, 4, 8\}, \{5, \mathbf{6}\}, \{\mathbf{7}\}. \tag{5.1}$$

Пусть изначально каждый объект находится в собственном одноэлементном множестве. С множествами нужно уметь выполнять следующие операции:

1) FindSet(x) — выдать указатель на представителя множества, которому принадлежит элемент x;

2) Union(x, y) — объединить два непересекающихся множества, которые содержат элементы x и y.

Структуру данных, поддерживающую такой интерфейс, называют *системой непересекающихся множеств* (СНМ) (англ. *disjoint set union* (DSU)).

Предположим, что набор объектов, для которого выполнялось разбиение на непересекающиеся множества, — целые числа от 1 до n. Рассмотрим несколько способов реализации описанного выше интерфейса.

5.3.1. Реализация на массиве

Реализация с использованием структуры данных *массив* предполагает, что элемент массива с индексом i (нумерация с 1) содержит представителя множества, которому принадлежит элемент i.

Для системы непересекающихся множеств (5.1) массив, который используется для работы с множествами, будет иметь следующий вид:

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

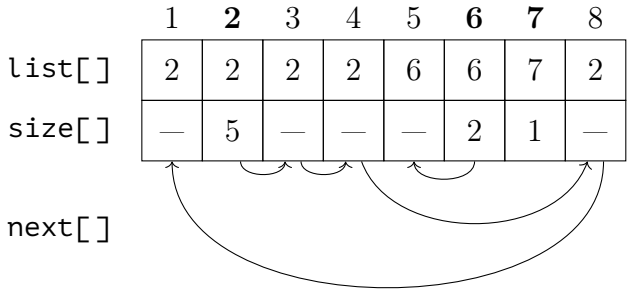
При такой реализации время выполнения базовых операций следующее:

- FindSet(x) выполняется за время $O(1)$ — достаточно одного обращения к элементу массива по индексу;
- Union(x, y) выполняется за время $O(n)$ — необходим проход по всему массиву.

5.3.2. Реализация на массиве и списке

Реализация с использованием структуры данных *список с указателем на представителя* предполагает, что элементы каждого множества связаны в отдельный список. Представителем множества является первый элемент списка. Каждый элемент списка содержит указатель на представителя и на следующий за ним элемент множества. Для каждого списка поддерживают два указателя: на первый и последний элементы списка. Кроме того, для каждого элемента множества необходимо в массиве дополнительно поддерживать указатель на его место в списковой структуре.

Вместо того чтобы организовывать связные списки на указателях, можно использовать для хранения системы непересекающихся множеств несколько массивов. Система множеств 5.1 может быть представлена следующим образом в виде трёх массивов:



Здесь массив `list` для каждого i содержит представителя множества, к которому принадлежит i. Массив `size` для каждого представителя хранит размер его множества, остальные элементы этого массива не используются (размеры множеств будут учитываться далее в ходе операции объединения). Массив `next` (на рисунке изображён условно в виде стрелок) связывает элементы каждого множества в цепочку, первым элементом которой является представитель.

Такая сложная структура даёт возможность эффективно перечислить элементы любого множества (за время, пропорциональное размеру этого множества). Однако в худшем случае время выполнения базовых операций остаётся прежним:

- FindSet(x) выполняется за время $O(1)$;
- Union(x, y) выполняется за время $O(n)$.

Размер множества выступает его весовой характеристикой. Оказывается, такие веса можно использовать, чтобы сделать выполнение операции объединения более эффективными.

Введём правило «*меньшее к большему*»: при выполнении операции UNION(x, y) ссылки на нового представителя изменяются у всех элементов меньшего множества. Данное правило позволяет получить следующую оценку [1]:

Теорема 5.1. *Последовательность из m операций* FINDSET(x), UNION(x, y) *потребует времени* $\mathrm{O}(m + n \log n)$.

Доказательство. Каждый раз, когда какой-либо элемент перемещается из одного множества в другое с изменением представителя, размер множества, содержащего этот элемент, увеличивается не менее чем вдвое. Поскольку множество может вырасти лишь до размера n, каждый элемент может подвергнуться перемещению не более чем $\log_2 n$ раз. Значит, все операции объединения потребуют в худшем случае времени $\mathrm{O}(n \log n)$. □

5.3.3. Реализация с помощью корневых деревьев

Другой подход предполагает использование *семейства корневых деревьев*. Каждому множеству поставим в соответствие своё корневое дерево. Каждой вершине дерева соответствует ровно один элемент множества. Корень дерева содержит представителя множества.

Семейство корневых деревьев в памяти компьютера хранится в каноническом виде — как массив `parent`, где `parent[i]` — отец вершины i в корневом дереве. Если вершина i является корнем дерева, то верно равенство `parent[i] == i`, т. е. представитель множества ссылается на самого себя.

На рис. 5.6 система непересекающихся множеств (5.1) представлена в виде семейства корневых деревьев.

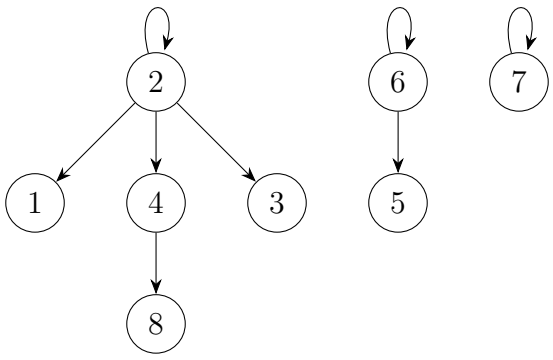


Рис. 5.6. Представление системы непересекающихся множеств в виде семейства корневых деревьев

В памяти компьютера семейство корневых деревьев хранится в следующем виде:

	1	2	3	4	5	6	7	8
`parent[]`	2	2	2	2	6	6	7	4

При простой реализации последовательность из $n-1$ операции UNION может привести к тому, что будет построено корневое дерево высоты $n-1$, а время выполнения базовых операций будет следующим:

- FINDSET(x) выполняется за время $\mathrm{O}(n)$;
- UNION(x, y) выполняется за время $\mathrm{O}(n)$.

Эвристика объединения по размеру или рангу

Время выполнения операций можно улучшить, если у каждого корневого дерева поддерживать весовую характеристику, в качестве которой может выступать или число вершин в дереве (размер), или высота дерева (ранг). Тогда при выполнении операции UNION корень дерева с меньшей весовой характеристикой будет ссылаться на корень дерева с большей весовой характеристикой. Можно доказать, что время выполнения базовых операций будет следующим:

- FINDSET(x) выполняется за время $\mathrm{O}(\log n)$;
- UNION(x, y) выполняется за время $\mathrm{O}(\log n)$.

Описанное правило (эвристика) называется *объединение по размеру или рангу* (англ. *union by size or rank*).

Рассмотрим случай, когда в качестве весовой характеристики используется размер дерева. В памяти компьютера семейство корневых деревьев с дополнительной весовой характеристикой `size` хранится следующим образом:

	1	2	3	4	5	6	7	8
`size[]`	1	5	1	2	1	2	1	1
`parent[]`	2	2	2	2	6	6	7	4

Иногда не вводят дополнительное поле (`size`), а в массиве `parent` для представителя множества хранят со знаком минус весовую характеристику множества, т. е. знак минус идентифицирует представителя множества [3].

	1	2	3	4	5	6	7	8
parent[]	2	−5	2	2	6	−2	−1	4

На рис. 5.7 приведён результат выполнения операции UNION(5, 8) для системы непересекающихся множеств, приведённой на рис. 5.6.

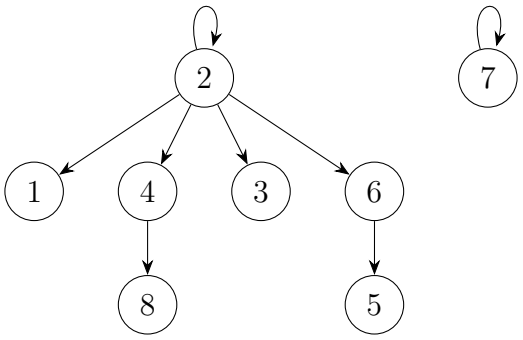


Рис. 5.7. Результат выполнения UNION(5, 8) с эвристикой объединения по размеру

В памяти компьютера после выполнения операции UNION(5, 8) система непересекающихся множеств будет иметь следующий вид:

	1	2	3	4	5	6	7	8
size[]	1	7	1	2	1	2	1	1
parent[]	2	2	2	2	6	2	7	4

Эвристика сжатия пути

Ещё одно усовершенствование, которое позволяет получить практически линейное время работы серии операций FINDSET и UNION, носит название *сжатие пути* (англ. *path compression*). Предположим, что выполняется операция FINDSET(x), которая выполняется простым подъёмом от вершины x к корню дерева, используя массив предков `parent`. Все посещённые при этом подъёме вершины составляют *путь поиска*. Процедура сжатия пути всем вершинам, лежащим на пути поиска, в качестве предка присваивает ссылку на корень данного дерева (сжатие пути не изменяет ранги вершин). Теперь массив `parent` правильнее называть *сжатым массивом предков*.

На рис. 5.8 приведён результат выполнения операции FINDSET(8) со сжатием пути для системы непересекающихся множеств, приведённой на рис. 5.7.

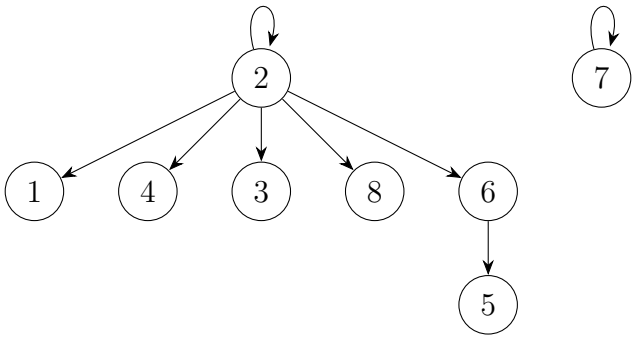


Рис. 5.8. Результат выполнения FindSet(8) с эвристикой сжатия пути

Справедлива следующая теорема (доказательство теоремы можно найти в [1]).

Теорема 5.2. *Последовательность из m операций* FindSet(x) *и* Union(x, y) *может быть выполнена с использованием эвристик объединения по размеру и сжатия пути за время* $\mathrm{O}(m\alpha(n))$ *в худшем случае.*

В формулировке теоремы функция $\alpha(n)$ — обратная функция для функции Аккермана. Функция $\alpha(n)$ растёт очень медленно, она становится больше числа 4 только для очень больших значений $n \gg 10^{80}$. Поэтому для практических приложений полагают, что $\alpha(n) \leqslant 4$.

Функция Аккермана:

$$A(m,n) = \begin{cases} n+1, & m = 0; \\ A(m-1,1), & m > 0, n = 0; \\ A(m-1, A(m,n-1)), & m > 0, n > 0. \end{cases}$$

Обратная функция для функции Аккермана определяется следующим образом:

$$\alpha(n) = \min\{k \geqslant 1 : A(k,k) \geqslant n\}.$$

5.3.4. Псевдокод

Приведём итоговую реализацию системы непересекающихся множеств на псевдокоде, которая хранит множества в виде семейства корневых деревьев и применяет обе описанные эвристики (объединение по размеру и сжатие пути).

```
class DisjointSetUnion:
    def __init__(self, n):
        self.size = [1 for i in range(n)]
        self.parent = [i for i in range(n)]
```

```
    def FindSet(self, x):
        if x == self.parent[x]:
            return x
        # Path compression
        self.parent[x] = self.FindSet(self.parent[x])
        return self.parent[x]

    def Union(self, x, y):
        x = self.FindSet(x)
        y = self.FindSet(y)
        if x != y:
            # Union by size
            if self.size[x] < self.size[y]:
                # Swap x and y
                x, y = y, x
            # Now size[x] >= size[y]
            self.parent[y] = x
            self.size[x] += self.size[y]
```

Заметим, что в результате применения сжатия пути размеры поддеревьев, хранящиеся в массиве `size`, для некоторых вершин становятся неактуальными. Однако это не влияет на корректность алгоритма: при выполнении объединения играют роль только значения размера, которые хранятся в корнях деревьев — у представителей множеств.

Поскольку система непересекающихся множеств является узкоспециализированной структурой данных и используется в прикладном программировании крайне редко, она не включена в стандартные библиотеки популярных языков программирования. Тем не менее реализация DSU есть в известной библиотеке boost для языка C++.

Часть 6

БИНАРНЫЙ ПОИСК

Операция поиска некоторого элемента x в произвольном массиве выполняется за время $\Theta(n)$ (в худшем случае нужно просмотреть последовательно все элементы). Однако если массив уже отсортирован, поиск элемента можно выполнить за время $O(\log n)$. Этот метод известен как бинарный поиск (аналогичные названия: поиск делением пополам, дихотомия) и часто используется на практике.

6.1. ОПИСАНИЕ МЕТОДА

Предположим, что есть некоторая упорядоченная последовательность элементов A, где $a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n-1}$. Необходимо проверить, есть ли среди них заданный элемент x. Первоначально определяем границы $[l, r)$ области поиска как $l = 0$, $r = n$. Определяем индекс центрального элемента области поиска. Предположим, что это число $k = \lfloor \frac{l+r}{2} \rfloor$. Сравниваем a_k — элемент последовательности — и число x. Если элементы совпадают, то поиск завершён. Если $x < a_k$, то продолжаем аналогичные действия, изменяя правую границу области поиска на k. Если $x > a_k$, то продолжаем аналогичные действия, изменяя левую границу области поиска на $k+1$. Алгоритм прекращает работу, как только будет найден требуемый элемент либо станет верным равенство $l = r$ (эта ситуация говорит о том, что элемента в последовательности нет).

```
def BinarySearch(a, x):
    l = 0, r = len(a)
    while l < r:
        k = (l + r) // 2
        if x == a[k]:
            return True
        else if x < a[k]:
```

```
            r = k
        else: # x > a[k]
            l = k + 1
    return False
```

В псевдокоде предполагается, что оператор `//` выполняет целочисленное деление с округлением вниз. Вместо выражения $\lfloor \frac{l+r}{2} \rfloor$ можно использовать эквивалентное $l + \lfloor \frac{r-l}{2} \rfloor$, чтобы избежать целочисленного переполнения в момент вычисления суммы.

Можно показать, что каждая новая итерация этого алгоритма приводит к уменьшению оставшейся части последовательности поиска не менее чем в два раза, поэтому алгоритм гарантированно завершается за $O(\log n)$ шагов. Частая проблема в реализации — вместо строки кода

```
l = k + 1
```

ошибочно пишут строку

```
l = k
```

При этом наблюдается такой эффект: как только полуинтервал $[l, r)$ сходится к единичной длине, т. е. становится верно $r = l + 1$, мы всякий раз получаем $k = \lfloor \frac{l+r}{2} \rfloor = l$ и выполняем тождественное присваивание $l = l$ — алгоритм зацикливается.

6.2. БИНАРНЫЙ ПОИСК САМОГО ЛЕВОГО ЭЛЕМЕНТА

Нетрудно модифицировать описанный выше алгоритм бинарного поиска, чтобы он выдавал индекс элемента x в случае, если такой элемент есть. Однако при наличии в массиве равных элементов непонятно, индекс какого из них будет выдан. Иногда требуется получить самое левое или самое правое вхождение. Обобщением этой задачи является задача поиска индекса первого элемента, большего, чем x, либо равного ему. Реализуем операцию LowerBound, которая решает эту задачу. В случае отсутствия в массиве подходящих элементов договоримся, что возвращаемое значение будет равно n.

```
def LowerBound(a, x):
    l = 0, r = len(a)
    while l < r:
        k = (l + r) // 2
        if x <= a[k]:
            r = k
```

```
        else: # x > a[k]
            l = k + 1
    return l
```

Кроме этого, может потребоваться найти индекс первого элемента, строго большего, чем x. Такая операция называется UpperBound, её реализация отличается от предыдущей только одним знаком неравенства.

```
def UpperBound(a, x):
    l = 0, r = len(a)
    while l < r:
        k = (l + r) // 2
        if x < a[k]:
            r = k
        else: # x >= a[k]
            l = k + 1
    return l
```

Заметим, что для любого x полуинтервал от LowerBound(A, x) (включительно) до UpperBound(A, x) (не включительно) определяет участок массива, содержащий элементы x. Разность между этими двумя величинами равна количеству элементов массива A, которые равны x.

Пример 6.1. Рассмотрим следующий массив длины 9:

0	1	2	3	4	5	6	7	8
2	4	**5**	**5**	**5**	27	67	91	97

Предположим, что искомый элемент равен 5. В этом случае функция LowerBound вернёт значение индекса 2, функция UpperBound вернёт 5.

В стандартной библиотеке языка C++ есть функция `std::binary_search`, которая выполняет бинарный поиск и возвращает логическое значение (есть элемент или нет). Кроме того, определены функции `std::lower_bound` и `std::upper_bound`, которые действуют аналогично рассмотренным и возвращают итераторы. В языке Java для классов `Arrays` и `Collections` определён статический метод `binarySearch`, который совмещает в себе описанные выше функции `BinarySearch` и `LowerBound`, однако является менее гибким (при наличии в массиве нескольких элементов, равных искомому, метод может вернуть индекс любого). В языке Python бинарный поиск реализован в стандартном модуле `bisect`.

Часть 7

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

7.1. УСЛОВИЯ

Задача 1. Считалка

Вокруг считающего стоят n человек, из которых выделен первый, а остальные занумерованы по часовой стрелке числами от 2 до n. Считающий, начиная с некоторого человека, ведёт счёт до m. Человек, на котором остановился счёт, выходит из круга. Счёт продолжается со следующего человека аналогичным образом до тех пор, пока не останется один человек.

Необходимо определить:

1) номер оставшегося человека, если известно m и то, что счёт начинался с первого человека;

2) номер человека, с которого начинался счёт, если известны m и номер оставшегося человека k.

Формат входных данных

На входе три целых числа: n ($2 \leqslant n \leqslant 10\,000$), m ($1 \leqslant m \leqslant 10^9$) и k ($1 \leqslant k \leqslant n$).

Формат выходных данных

В первой строке выведите номер оставшегося человека, во второй — номер человека, с которого должен был начаться счёт.

Пример

входной файл	*выходной файл*
5 5 5	2 4

Задача 2. Полоска

Задана полоска длины 2^k клеток и шириной в одну клетку. Полоску сгибают пополам так, чтобы правая половинка оказалась под левой. Сгибание продолжают до тех пор, пока сверху находится больше одной клетки.

Необходимо пронумеровать клетки таким образом, чтобы после окончания сгибания полосы номера клеток в получившейся колонке были расположены в порядке $1, 2, 3, 4, \ldots, 2^k$.

Формат входных данных

Единственная строка содержит целое число k ($1 \leqslant k \leqslant 13$).

Формат выходных данных

Выведите 2^k чисел.

Пример

входной файл	*выходной файл*
5	1 32 17 16 9 24 25 8 5 28 21 12 13 20 29 4 3 30 19 14 11 22 27 6 7 26 23 10 15 18 31 2

Задача 3. Квадрат

Квадрат разбит на 4^k равновеликих квадратных клеток. Квадрат перегибается поочерёдно относительно вертикальной (правая половина подкладывается под левую) и горизонтальной (нижняя половина подкладывается под верхнюю) осей симметрии до тех пор, пока все клетки не будут расположены друг под другом.

Необходимо занумеровать клетки исходного квадрата таким образом, чтобы в результате выполнения операций перегиба номера клеток, расположенных друг под другом, образовали числовую последовательность 1, 2, 3, …, 4^k, начиная с верхней клетки.

Формат входных данных

Единственная строка содержит целое число k ($1 \leqslant k \leqslant 6$).

Формат выходных данных

Выведите 4^k чисел.

Пример

входной файл	*выходной файл*
2	1 8 7 2 16 9 10 15 13 12 11 14 4 5 6 3

Задача 4. Таблица

В последовательности из n чисел за один просмотр необходимо каждый элемент заменить на ближайший следующий за ним элемент, который больше его, если такой существует, или на ноль в противном случае. Можно использовать дополнительную память.

Формат входных данных

На входе задано n ($1 \leqslant n \leqslant 10^7$) целых чисел, каждое из которых лежит на интервале от 1 до 10^9 включительно.

Формат выходных данных

Выведите элементы новой последовательности.

Пример

входной файл	*выходной файл*
`1 3 2 5 3 4`	`3 5 5 0 4 0`

Задача 5. Остановки

В городе имеется n автобусных остановок, обозначенных числами из множества $\{1, 2, \ldots, n\}$. Имеется r автобусных маршрутов, заданных последовательностями соседних остановок при движении автобуса в одном направлении.

Необходимо по заданным номерам остановок s и f определить наиболее быстрый маршрут перемещения пассажира от остановки s к остановке f с использованием имеющихся маршрутов автобусов при условии, что время движения между соседними остановками у всех маршрутов одинаково и в три раза меньше времени изменения маршрута (выход из автобуса также считается изменением маршрута). Известно, что автобусы могут двигаться в обоих направлениях.

Формат входных данных

В первой строке записаны два целых числа: количество остановок n ($1 \leqslant n \leqslant 1\,000\,000$) и количество маршрутов r ($1 \leqslant r \leqslant 1000$).

Вторая строка содержит два числа: f (конечная остановка) и s (начальная остановка).

Далее идут r строк, каждая строка содержит информацию об одном маршруте: сначала указывается число остановок в маршруте, а затем перечислены соседние остановки при движении автобуса в одном направлении. Число пунктов в каждом маршруте не превосходит 1000.

Формат выходных данных

Если маршрута между заданными остановками не существует, то выведите одну строку `NoWay`.

В случае существования маршрута между заданными остановками, в первую строку выводится минимальное время, за которое можно проехать от остановки s к остановке f. Последующие строки содержат информацию о способе самого быстрого проезда от s к f. Каждая строка содержит два числа, задающие номер остановки и номер автобуса, на котором осуществлялся выезд *из* неё. Маршруты занумерованы целыми числами от 1 до r в порядке их следования на входе. Последняя строка содержит два числа: f и номер автобуса, на котором прибыли *в* конечный пункт f.

Замечания

• Маршрут может через некоторую остановку проходить несколько раз: например,

$$1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 4 \leftrightarrow 5 \leftrightarrow 6 \leftrightarrow 3 \leftrightarrow 7 \leftrightarrow 8.$$

• Если начальная остановка совпадает с конечной, то выдаётся только минимальная стоимость маршрута (0).

• Если вы на некоторой остановке x вышли из автобуса некоторого маршрута y, а затем опять сели на тот же маршрут y (на той же остановке x), то за это берётся дополнительный взнос в три единицы.

Примеры

входной файл	*выходной файл*	*пояснение*
7 3 7 1 6 1 2 3 4 5 6 2 2 6 2 6 7	9 1 1 2 2 6 3 7 3	
8 1 8 1 9 1 2 3 4 5 6 3 7 8	7 1 1 2 1 3 1 7 1 8 1	*рис. 7.1*

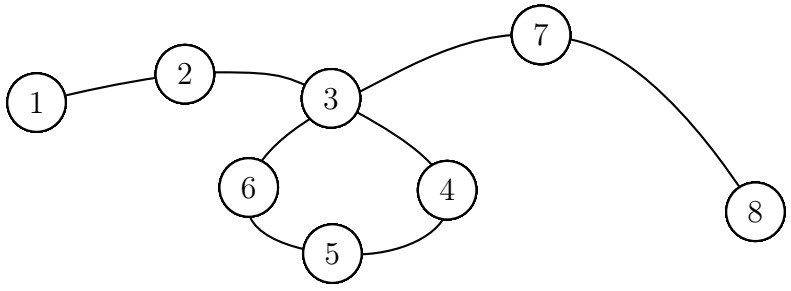


Рис. 7.1. Иллюстрация ко второму примеру

Задача 6. Король

Одинокий король долго ходил по бесконечной шахматной доске. Известна последовательность из n его ходов. Необходимо определить, побывал ли король дважды на одном и том же месте, за минимально возможное при заданном n число операций.

Формат входных данных

В первой строке задано число n ходов короля ($1 \leqslant n \leqslant 200\,000$).

Вторая строка содержит последовательность из n чисел, задающих последовательность ходов короля:

0 — вверх,	4 — вниз,
1 — вправо—вверх,	5 — влево—вниз,
2 — вправо,	6 — влево,
3 — вправо—вниз,	7 — влево—вверх.

Формат выходных данных

Выведите сообщение `Yes`, если король побывал на одной клетке дважды, и `No` в противном случае.

Пример

входной файл	*выходной файл*
8 0 1 2 3 4 5 6 7	Yes

Задача 7. Карточки

Имеется n чёрных и белых карточек, сложенных в стопку. Карточки раскладываются на стол в одну линию следующим образом: первая кладётся на стол; вторая — под низ стопки; затем третья — на стол; четвёртая — под низ стопки и т. д., пока все карточки не будут выложены на стол.

Необходимо определить, каким должно быть исходное расположение карточек в стопке, чтобы разложенные на столе карточки чередовались по цвету: белая, чёрная, белая, чёрная и т. д.

Формат входных данных

На входе задано количество карточек n ($1 \leqslant n \leqslant 10^6$).

Формат выходных данных

Выведите последовательность из нулей и единиц, где 0 — белая карточка, а 1 — чёрная.

Пример

входной файл	*выходной файл*
10	0 1 1 1 0 0 1 0 0 1

Задача 8. Форма с погружением

Форма тела задана матрицей A размера $n \times m$. Элемент a_{ij} соответствует высоте горизонтальной квадратной площадки размера 1×1 относительно нижнего основания. Основание формы горизонтально.

Необходимо определить объём невытекшей воды, если тело опускается полностью в воду, затем поднимается.

Формат входных данных

В первой строке задаются два числа — n и m ($1 \leqslant n, m \leqslant 1000$).

Начиная со второй строки идут $n \cdot m$ целых неотрицательных чисел — элементы матрицы A.

Формат выходных данных

Выведите объём невытекшей воды.

Пример

входной файл	*выходной файл*
3 3 5 3 1 5 2 5 2 5 5	1

Задача 9. Шланги

Два длинных шланга разных цветов запутаны между собой (рис. 7.2). При следовании вдоль каждого из шлангов точки их пересечения идут последовательно. Заданы координаты этих точек (в одном из двух возможных порядков) и для каждой точки указано, какой из шлангов находится сверху.

Имеются две стенки, расположенные одна напротив другой. Каждый из запутанных шлангов одним концом закрепляется на одной стенке, а другим — на противоположной. Необходимо определить, можно ли распутать шланги, не освобождая их концы.

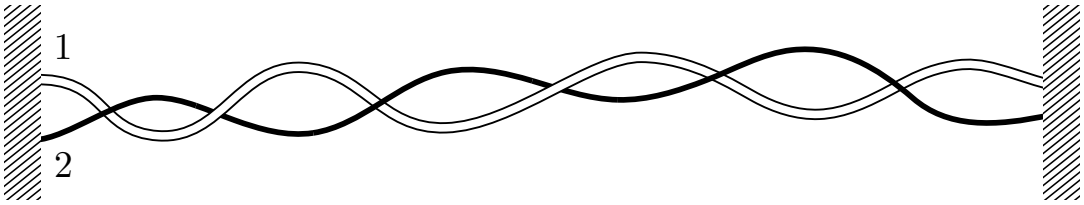


Рис. 7.2. Схема расположения шлангов

Формат входных данных

Первая строка содержит число n точек ($1 \leqslant n \leqslant 10^6$), в которых перепутаны шланги. Каждая из следующих n строк содержит коорди-

наты точки, в которой шланги перепутаны, и номер шланга (1 или 2), который находится сверху (точки указываются в порядке следования вдоль одного из шлангов).

Формат выходных данных

Выведите `Yes`, если шланги можно распутать, иначе `No`.

Примеры

входной файл	*выходной файл*
6 1 1 2 2 1.2 1 4 1 2 5.3 4 1 7.1 4 2 9 3 2	No
6 4 9.5 2 6 9 2 7 8 1 7.8 6 1 7.5 4 2 6 2.3 2	Yes

Задача 10. Лабиринт с конкретным входом-выходом

Матрица размера $n \times m$ определяет некоторый лабиринт (нумерация строк и столбцов матрицы начинается с 1). В матрице элемент 1 обозначает стену, а элемент 0 — свободное место. В первой строке матрицы определяются входы x_i, а в последней — выходы y_i $(i = 1, \ldots, k)$, которые должны быть нулевыми элементами матрицы (все элементы первой и последней строк, отличные от входов и выходов, равны 1).

Имеется k человек, которые занумерованы целыми числами от 1 до k. Человек с номером i начинает движение от входа x_i и заканчивает движение, выходя через выход y_i $(i = 1, \ldots, k)$. Движение в лабиринте осуществляется только по вертикали или горизонтали.

Необходимо определить, можно ли провести всех людей таким образом, чтобы каждое свободное место посещалось не более одного раза.

Формат входных данных

В первой строке задаются размеры n и m $(1 \leqslant n \times m \leqslant 10^6)$ матрицы.

Следующие n строк задают матрицу лабиринта (входы и выходы задаются нулями в первой и последней строках).

Формат выходных данных

Выведите сообщение `Possible`, если можно пройти, и `Impossible`, если нельзя.

Пример

входной файл	*выходной файл*
7 6 110011 000000 000100 000000 110110 000000 111010	Possible

Задача 11. Лабиринт и произвольный выход

Матрица размера $n \times m$ определяет некоторый лабиринт (нумерация строк и столбцов матрицы начинается с 1). В матрице элемент, равный 1, обозначает клетку, проход через которую запрещён, а равный 0 — свободное место. В первой строке матрицы определяются входы x_i, а в последней выходы — y_i $(i = 1, \ldots, k)$, которые должны быть нулевыми элементами (но не каждый нулевой элемент первой (последней) строки является входом (выходом)). Возле каждой клетки, которая помечена как вход, в стене расположена дверь; за дверью стоит один человек с номером i. В каждой клетке, которая помечена как выход, в полу расположен люк, через который могут спускаться люди.

Необходимо определить, какое максимальное число людей можно провести по лабиринту. Каждую свободную клетку лабиринта разрешено посещать не более одного раза (существует опасность разрушения несущих конструкций пола после прохода человека, поэтому второму на него ступать опасно). Однако через один выход (в люк) могут выйти несколько человек. Движение в лабиринте осуществляется только по вертикали или горизонтали.

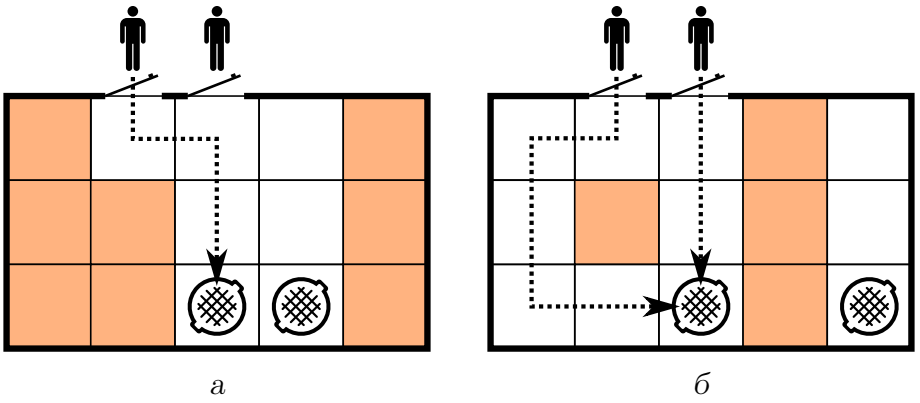


Рис. 7.3. Схемы лабиринтов: *а* — первый пример; *б* — второй пример

Формат входных данных

В первой строке находятся три числа: n, m и k ($1 \leqslant n \times m \leqslant 10^6$, $1 \leqslant k \leqslant m$).

Во второй строке находятся числа x_i для $i = 1, \ldots, k$, которые соответствуют номерам столбцов первой строки матрицы лабиринта, являющимся входами для i-го человека (первый столбец имеет номер 1).

В третьей строке находятся числа y_i для $i = 1, \ldots, k$, которые соответствуют номерам столбцов последней строки матрицы лабиринта, являющимся выходами.

В следующих n строках задаются строки полей матрицы лабиринта: 0 — свободное место, 1 — стена.

Формат выходных данных

Первая строка содержит максимальное количество людей, которых удалось провести.

Следующие n строк по m элементов в каждой задают матрицу лабиринта, содержащую пути людей, которых удалось провести: позиции лабиринта, по которым должен идти к выходу человек с номером i, помечаются числом $i+1$ (так как через один выход могут выйти несколько человек, то клетка выхода помечается номером $(i + 1)$ последнего человека (i), который вышел через люк).

Примеры

входной файл	*выходной файл*	*пояснение*
3 5 2 2 3 3 4 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1	1 1 2 2 0 1 1 1 2 0 1 1 1 2 0 1	*рис. 7.3, а*
3 5 2 2 3 3 5 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0	2 2 2 3 1 0 2 1 3 1 0 2 2 3 1 0	*рис. 7.3, б*

Задача 12. Чиновники

Есть министерство из n чиновников. У каждого из чиновников могут быть как подчинённые, так и начальники, причём справедливы правила:

- подчинённые моего подчинённого — мои подчинённые;
- мой начальник не есть мой подчинённый;
- у каждого чиновника не более одного непосредственного начальника.

Для того чтобы получить лицензию на вывоз меди, необходимо получить подпись первого чиновника — начальника всех чиновников (главный чиновник имеет всегда номер 1). Проблема осложняется тем, что каждый чиновник, вообще говоря, может потребовать визы, т. е. подписи некоторых своих непосредственных подчинённых, и взятки — известное число долларов дополнительно к соответствующей визе. Для каждого чиновника известен набор возможных виз и соответствующая каждой визе взятка. Пустой набор для некоторого чиновника означает, что он не требует виз, чтобы поставить свою подпись (т. е. ставит свою подпись бесплатно). Чиновник ставит свою подпись только после того, как ему представлена хотя бы одна подпись из набора требуемых им виз (либо список виз для него пустой).

Необходимо определить, какое минимальное количество денег предпринимателю нужно заплатить, чтобы получить лицензию у главного чиновника, и указать соответствующий этой сумме порядок получения подписей.

Формат входных данных

Первая строка содержит число n ($1 \leqslant n \leqslant 100\,000$).

Далее следуют n строк следующего формата: k — номер чиновника, m — возможное количество требуемых данным чиновником виз, далее следуют m пар чисел: номер чиновника-подчинённого и стоимость соответствующей взятки.

Формат выходных данных

Первая строка содержит минимальное количество денег, которое предпринимателю нужно заплатить, чтобы получить лицензию у главного чиновника.

Вторая строка — номера чиновников в последовательности, противоположной той, в которой следует получать подписи для получения лицензии. Список начинается с главного чиновника (т. е. с номера 1).

Пример

входной файл	*выходной файл*
8 1 2 2 4 3 4 2 3 4 3 5 6 6 2 3 2 7 1 8 2 4 0 5 0 6 0 7 0 8 0	5 1 3 7

Задача 13. Караван с движением по меньшей высоте

Имеется план местности, разбитый на квадраты, заданный матрицей размера $n \times n$. Каждый квадрат с номером (i, j) имеет высоту относительно уровня моря, значение которой определяется числом a_{ij}.

Необходимо определить маршрут каравана из позиции (x_1, y_1) в позицию (x_2, y_2). Караван может двигаться только по местности параллельно осям O_x и O_y между центрами квадратов и только в соседний квадрат с меньшей высотой.

Формат входных данных

Первая строка содержит число n ($1 \leqslant n \leqslant 1\,000$), а также числа x_1, y_1, x_2, y_2 ($1 \leqslant x_i, y_i \leqslant n$).

Каждая из следующих n строк содержит информацию о строке матрицы высот. Все высоты задаются положительными целыми числами, не превосходящими 10^9.

Формат выходных данных

Выведите n строк по n элементов. В i-й строке на j-м месте выведите элемент a_{ij} матрицы, если в квадрат (i, j) местности из точки старта потенциально может пройти караван, и число 0 в противном случае.

Пример

входной файл	*выходной файл*
5 1 1 4 5 15 12 11 13 9 16 11 10 9 8 7 10 11 8 9 6 9 10 7 10 5 4 3 2 1	15 12 11 0 0 0 11 10 9 8 7 10 0 8 0 6 9 0 7 0 5 4 3 2 1

Задача 14. Караван с движением по крутизне не больше k

Имеется план местности размера $n \times m$, разбитый на квадраты. Каждый квадрат с координатами (i, j) имеет высоту относительно уровня моря, значение которой определяется числом a_{ij}. Караван может двигаться только по местности и только параллельно осям O_x и O_y между центрами квадратов. При переходе в соседний квадрат крутизна подъёма (спуска) равна модулю разности высот квадратов.

Необходимо определить маршрут каравана из позиции (x_1, y_1) в позицию (x_2, y_2) минимальной длины (по количеству переходов из квадрата в квадрат), при котором крутизна его подъёмов и спусков на каждом переходе из квадрата в квадрат не превышает величины k.

Формат входных данных

Первая строка содержит два целых числа n и m ($1 \leqslant n, m \leqslant 1000$).

Следующие n строк содержат информацию о матрице высот. Высоты задаются положительными целыми числами, не превосходящими 10^9.

Следующая строка содержит число k ($0 \leqslant k \leqslant 10^9$).

Последняя строка содержит числа x_1, y_1, x_2 и y_2 ($1 \leqslant x_i, y_i \leqslant n$).

Формат выходных данных

Выведите в первой строке сообщение `Yes`, если движение в точку (x_2, y_2) возможно, а во второй строке — минимальное количество требуемых для этого перемещения ходов.

Если движение невозможно, то выведите сообщение `No`.

Пример

входной файл	*выходной файл*
5 5 50 14 18 12 17 19 12 10 21 12 15 9 7 13 11 11 8 5 11 9 5 4 3 2 1 13 1 2 5 4	Yes 6

Задача 15. Минимальная длина маршрута робота

Имеется план местности, разбитой на квадраты, заданный матрицей размера $n \times m$. Каждый квадрат с координатами (i, j) имеет высоту относительно уровня моря, значение которой определяется натуральным числом a_{ij} (нумерация строк и столбцов матрицы A начинается с 1, верхний левый квадрат матрицы имеет координаты $(1, 1)$). Робот может двигаться только по местности и только параллельно осям O_x и O_y между центрами квадратов. При переходе в соседний квадрат длина подъёма (спуска) равна модулю разности высот квадратов, а длина перемещения из квадрата в квадрат равна величине k.

Необходимо найти среди маршрутов робота из позиции (x_1, y_1) в позицию (x_2, y_2) маршрут минимальной длины. Длина маршрута определяется как суммарная длина подъёмов и спусков плюс суммарная длина перемещений из квадрата в квадрат.

Формат входных данных

Первая строка содержит числа n и m ($1 \leqslant n, m \leqslant 1000$) — размеры поля, по которому двигается робот.

Следующие n строк содержат информацию о матрице высот каждого квадрата (строка входа соответствует строке матрицы высот; высоты — целые числа от 1 до 10^9).

Следующая строка — число k ($0 \leqslant k \leqslant 10^9$).

В последней строке задаются координаты начальной и конечной точек движения робота: x_1, y_1, x_2, y_2.

Формат выходных данных

Выведите длину кратчайшего маршрута робота.

Пример

входной файл	*выходной файл*
4 4 5 3 2 6 1 8 4 2 3 2 5 4 2 2 2 2 1 1 1 4 4	13

Задача 16. Минимальное число ходов коня

Имеется шахматная доска $n \times m$ клеток (верхний левый квадрат доски имеет координаты $(1, 1)$). Некоторые поля на шахматной доске заняты белыми фигурами и пешками (эти поля недоступны для хода белого коня), при этом каждое занятое поле определяется числом -1.

Необходимо определить минимальное число ходов белого коня, которое потребуется для его перемещения из позиции (x_1, y_1) в позицию (x_2, y_2).

Формат входных данных

В первой строке задаются размеры поля n и m ($1 \leqslant n, m \leqslant 1000$).

Следующие n строк задают данные поля (по m чисел в строке).

В последней строке находятся числа x_1, y_1, x_2, y_2, задающие координаты начальной и конечной позиций коня.

Формат выходных данных

В единственной строке выведите минимальное число ходов коня или сообщение `No`, если пути не существует.

Пример

входной файл	*выходной файл*
8 8 0 0 -1 0 0 0 0 0 -1 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 -1 0 0 0 0 -1 0 0 0 0 0 0 -1 0 0 0 0 0 -1 -1 0 0 -1 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 6 8 1 7	4

Задача 17. Минимальное число ходов белого коня со взятием чёрных фигур

Имеется шахматная доска $n \times m$ клеток (верхний левый квадрат доски имеет координаты $(1, 1)$, правый нижний — (n, m)). Некоторые поля на ней заняты белыми и чёрными фигурами (ладья, слон, конь, король и ферзь) и пешками. Каждое поле, занятое белыми фигурами и белыми пешками, определяется отрицательным числом -1, а чёрными фигурами и чёрными пешками — числом 1.

Необходимо определить маршрут белого коня из поля (x_1, y_1) в поле (x_2, y_2), при котором число его ходов минимально. Взятие белым конём чёрной фигуры или пешки считается дополнительным ходом (т. е. ход на поле, занятое чёрной фигурой или чёрной пешкой, стоит два хода).

Формат входных данных

В первой строке задаются размеры поля n и m ($1 \leqslant n, m \leqslant 1000$).

Следующие n строк задают поле (по m чисел в строке).

В последней строке заданы числа x_1, y_1, x_2 и y_2.

Формат выходных данных

В единственной строке выведите минимальное число ходов или сообщение `No`, если пути не существует.

Пример

входной файл	*выходной файл*
5 5 -1 0 -1 0 0 0 0 0 1 0 -1 1 1 0 0 1 1 -1 0 0 0 -1 1 -1 0 1 2 4 4	4

Задача 18. Минимальное число белых коней

Имеется шахматная доска $n \times m$ клеток (верхний левый квадрат доски имеет координаты $(1, 1)$). Некоторые поля на ней заняты белыми фигурами, но не конями (ладья, слон, король и ферзь) и белыми пешками. Каждое занятое поле определяется отрицательным числом -1, а свободное — числом 0.

Необходимо определить минимальное число белых коней, которое необходимо расставить на доске, чтобы при постановке чёрной фигуры в любое оставшееся свободное поле она могла быть взята одним из этих коней за некоторое число ходов.

Формат входных данных

В первой строке задаются размеры поля n и m ($1 \leqslant n, m \leqslant 1000$). Следующие n строк задают поле (по m чисел в строке).

Формат выходных данных

В единственной строке выведите минимальное число белых коней.

Пример

входной файл	*выходной файл*
5 3 0 0 0 0 0 0 -1 0 -1 -1 -1 0 -1 0 0	4

Задача 19. Минимальное число белых ладей

Имеется шахматная доска $n \times m$ клеток. Некоторые поля на ней заняты белыми фигурами, но не ладьями (конь, слон, король, ферзь), и белыми пешками. Каждое занятое поле определяется отрицательным числом -1, а свободное — числом 0.

Необходимо определить минимальное число белых ладей, которое необходимо расставить на доске, чтобы при постановке чёрной фигуры в любое оставшееся свободное поле она могла быть взята одной из этих ладей за некоторое число ходов.

Формат входных данных

В первой строке задаются размеры поля n и m ($1 \leqslant n, m \leqslant 1000$). Следующие n строк задают данные поля (по m чисел в строке).

Формат выходных данных

Выведите минимальное число белых ладей.

Пример

входной файл	*выходной файл*
5 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 -1 0 0 0 0	2

Задача 20. Минимальное число белых слонов

Имеется шахматная доска $n \times m$ клеток (верхний левый квадрат доски имеет координаты $(1, 1)$). Некоторые поля на ней заняты белыми

фигурами, но не слонами (конь, ладья, король, ферзь) и белыми пешками. Каждое занятое поле определяется отрицательным числом -1, а свободное — числом 0.

Необходимо определить минимальное число белых слонов, которое необходимо расставить на доске, чтобы при постановке чёрной фигуры в любое оставшееся свободное поле она могла быть взята одним из этих слонов за некоторое число ходов.

Формат входных данных

В первой строке задаются размеры поля n и m ($1 \leqslant n, m \leqslant 1000$). Следующие n строк задают позицию на поле (по m чисел в строке).

Формат выходных данных

В единственной строке выведите минимальное число белых слонов.

Пример

входной файл	*выходной файл*
5 3 0 0 0 0 0 0 -1 0 -1 -1 -1 0 -1 0 0	3

Задача 21. Цилиндр

Лист бумаги в клетку размера $n \times m$ сначала склеили в цилиндр высоты m (рис. 7.4), а затем из него вырезали некоторые из клеток.

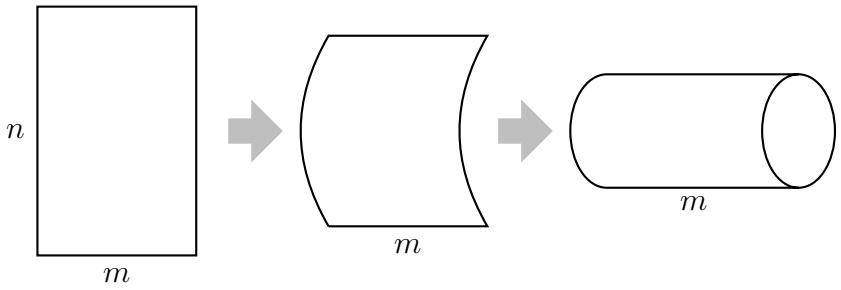


Рис. 7.4. Техника изготовления цилиндра из бумаги

Необходимо определить, на какое число кусков распадётся цилиндр. Две невырезанные смежные клетки принадлежат одному куску, если у них есть общая сторона.

Формат входных данных

В первой строке заданы числа n и m ($1 \leqslant n, m \leqslant 1000$). Следующие n строк (по m чисел в каждой) содержат данные клеток листа, где 1 означает, что клетку вырезали, а 0 — что клетку не вырезали.

Формат выходных данных

Выведите число кусков, на которое распалась оставшаяся часть листа бумаги.

Пример

входной файл	*выходной файл*
2 4 0 1 1 0 0 1 0 0	2

Задача 22. Прямоугольники

Прямоугольники различных цветов располагаются на белом прямоугольном листе бумаги размера $h \times w$. Стороны прямоугольников параллельны краям листа, а сами прямоугольники не выходят за пределы листа. В результате образуются различные одноцветные фигуры. Если два прямоугольника одного цвета имеют хотя бы одну общую точку, то они являются частями одной одноцветной фигуры.

Необходимо для каждого цвета вычислить площадь каждой из видимых одноцветных фигур.

Начало системы координат находится в центре листа, а оси параллельны краям листа (рис. 7.5). Ось O_x идёт слева направо, ось O_y проходит снизу вверх. Левый нижний угол имеет координаты $(-w/2, -h/2)$, а правый верхний угол — координаты $(w/2, h/2)$.

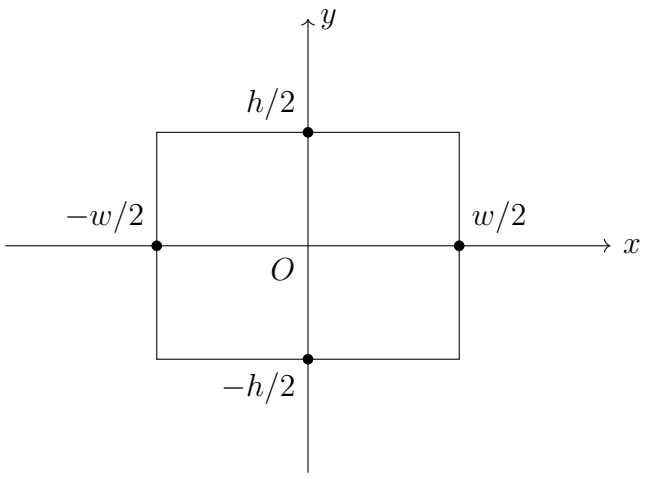


Рис. 7.5. Схема расположения листа бумаги

Формат входных данных

Первая строка содержит три целых числа h, w и n ($2 \leqslant h, w \leqslant 500$, числа h и w — чётные, $1 \leqslant n \leqslant 10\,000$). Далее идут n строк, содержащих следующую информацию:

• целочисленные координаты точки (x_1, y_1), в которую помещена левая нижняя вершина прямоугольника;

• целочисленные координаты точки (x_2, y_2), в которую помещена правая верхняя вершина прямоугольника;

• цвет прямоугольника, заданный целым числом от 1 до 64; белый цвет представлен числом 1.

Прямоугольником называется множество точек (x, y), удовлетворяющих неравенствам $x_1 \leqslant x \leqslant x_2$ и $y_1 \leqslant y \leqslant y_2$. Гарантируется, что для каждого прямоугольника верны неравенства $-w/2 \leqslant x_1 < x_2 \leqslant w/2$ и $-h/2 \leqslant y_1 < y_2 \leqslant h/2$.

Формат выходных данных

Для каждой из одноцветных фигур выведите в отдельной строке её цвет и площадь. Эти строки расположите в порядке возрастания номера цвета. Если две фигуры имеют одинаковый цвет, то выведите результат в порядке возрастания площади соответствующих одноцветных фигур.

Замечания

1. Следует учитывать, что информацию о фигурах белого цвета также следует выводить.

2. Порядок строк на входе соответствует тому порядку, в котором прямоугольники размещались на листе (от первого прямоугольника до последнего).

Пример

входной файл	*выходной файл*
12 20 5 -7 -5 -3 -1 4 -3 -3 5 3 2 -4 -2 -2 2 4 2 -2 3 -1 12 3 1 7 5 1	1 177 2 39 4 23 12 1

Задача 23. За́мок

Имеется план замка, который разделён на комнаты. Условно замок разделён на $m \times n$ клеток. Комната — связная область, ограниченная стенками и границами замка. План замка задаётся в виде последовательности чисел, характеризующих эти клетки.

Необходимо определить:

1) число комнат в замке;

2) площадь наибольшей комнаты;

3) какую стену в замке следует удалить, чтобы получить комнату наибольшей возможной площади.

Формат входных данных

В первой строке записаны два числа m и n — число клеток в направлении с севера на юг и с запада на восток ($1 \leqslant m, n \leqslant 1000$).

Каждая из следующих m строк задаёт информацию о клетках, при этом каждая клетка описывается числом p ($0 \leqslant p \leqslant 15$). Это число p является суммой следующих чисел (рис. 7.6):

- 1, если клетка имеет западную стену;
- 2 — северную;
- 4 — восточную;
- 8 — южную.

Формат выходных данных

В первой строке выведите число комнат в замке. Во второй — площадь наибольшей комнаты. В третьей — наибольшую возможную площадь комнаты, полученной после удаления одной стены.

Пример

входной файл	*выходной файл*	*пояснение*
3 3 15 11 6 3 14 5 9 10 12	2 8 9	*рис. 7.7*

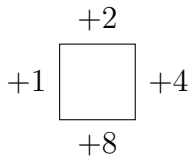


Рис. 7.6. Принцип кодирования стен

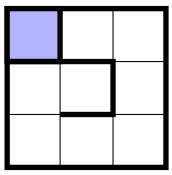


Рис. 7.7. Схема замка из примера

Задача 24. Компании

Некоторые компании являются совладельцами других компаний, так как приобрели часть их акций. Пусть компания A владеет α % акций компании B. Говорят, что компания A контролирует компанию B, если имеет место по меньшей мере одно из следующих двух условий:

- A владеет более чем 50 % акций B, т. е. $\alpha > 50$;
- A контролирует k ($k > 0$) компаний $C_1, \ldots, C_k$, из которых компания C_i владеет соответственно γ_i % акций компании B ($i = 1, \ldots, k$), и $\alpha + \gamma_1 + \gamma_2 + \ldots + \gamma_k > 50$.

Необходимо определить все пары (i, j) чисел, при которых компания i контролирует компанию j.

Формат входных данных

На входе задаётся последовательность троек (i, j, p), которые означают, что компания i владеет $p\,\%$ акций компании j. Общее число компаний не превышает 100.

Формат выходных данных

В каждой строке выведите одну найденную пару (i, j). Пары требуется упорядочить лексикографически.

Пример

входной файл	*выходной файл*
2 3 25 1 4 36 4 5 63 2 1 48 3 4 30 4 2 52 5 3 30	4 2 4 3 4 5

Задача 25. Правильная скобочная последовательность

Правильная скобочная последовательность (ПСП) — последовательность скобочных символов, определяющаяся следующими образом.

- Пустая строка есть правильная скобочная последовательность.
- Пусть S — правильная скобочная последовательность, тогда (S), $[S]$ и $\{S\}$ — правильные скобочные последовательности.
- Пусть S_1, S_2 — правильные скобочные последовательности, тогда $S_1 S_2$ есть правильная скобочная последовательность.

Проверьте, является ли заданная строка правильной скобочной последовательностью.

Формат входных данных

На входе задана непустая строка, состоящая из круглых, квадратных и фигурных скобок. Длина строки не превосходит 1 000 000.

Формат выходных данных

Если строка удовлетворяет определению правильной скобочной последовательности, выведите единственную строку `YES`. В противном случае в первой строке выведите `NO`, а во второй строке укажите длину наибольшего префикса исходной строки, который можно дополнить справа до правильной скобочной последовательности.

Примеры

входной файл	*выходной файл*
([{}])	YES
{}(()])	NO 5

Задача 26. Чёрный ящик

Чёрный ящик организован наподобие примитивной базы данных. Он может хранить набор целых чисел и имеет выделенную переменную i. В начальный момент времени чёрный ящик пуст, а значение переменной i равно нулю. Чёрный ящик обрабатывает последовательность поступающих команд (запросов). Существуют два вида запросов:

- Добавить(x) — положить в чёрный ящик элемент x;
- Получить() — увеличить значение переменной i на 1 и выдать копию i-го по величине элемента чёрного ящика (напомним, что i-м по величине элементом называется число, стоящее на i-м месте в отсортированной по неубыванию последовательности элементов чёрного ящика).

Необходимо разработать алгоритм, обрабатывающий заданную последовательность поступающих команд (запросов) за время $O(m \log m)$, где m — число запросов Добавить.

Формат входных данных

В первой строке задано число m запросов Добавить и число n запросов Получить ($1 \leqslant n \leqslant m \leqslant 2\,000\,000$).

Во второй строке — последовательность $a_1, a_2, \ldots, a_m$ включаемых в чёрный ящик элементов — целых чисел, не превосходящих по абсолютной величине $2\,000\,000\,000$.

В третьей строке — последовательность $u_1, u_2, \ldots, u_n$, задающая число содержащихся в чёрном ящике элементов в момент выполнения первой, второй, ..., n-й команды Получить.

Схема работы чёрного ящика предполагает, что последовательность $u_1, u_2, \ldots, u_n$ целых чисел упорядочена по неубыванию, $n \leqslant m$ и для всех p ($1 \leqslant p \leqslant n$) выполняется соотношение $p \leqslant u_p \leqslant m$. Последнее следует из того, что для p-го элемента последовательности u мы выполняем запрос Получить, выдающий p-е по величине число из набора $a_1, a_2, \ldots, a_{u_p}$.

Формат выходных данных

Выведите полученную последовательность ответов чёрного ящика для заданной последовательности запросов (т. е. результаты работы команды Получить).

Пример

входной файл	*выходной файл*
6 4 3 1 -4 2 8 -1000 1 2 6 6	3 3 1 2

Задача 27. Межшкольная сеть

Некоторые школы связаны компьютерной сетью. Между школами заключены соглашения: каждая школа имеет список школ-получателей, которым она рассылает программное обеспечение всякий раз, получив новое бесплатное программное обеспечение (извне сети или из другой школы). При этом если школа b есть в списке получателей школы a, то школа a может и не быть в списке получателей школы b.

Необходимо решить следующие две подзадачи.

1. Определить минимальное число школ, которым надо передать по экземпляру нового программного обеспечения, чтобы распространить его по всем школам сети в соответствии с соглашениями.

2. Обеспечить возможность рассылки нового программного обеспечения из любой школы по всем остальным школам. Для этого можно расширять списки получателей некоторых школ, добавляя в них новые школы. Требуется найти минимальное суммарное число расширений списков, при которых программное обеспечение из любой школы достигло бы всех остальных школ. Под расширением понимается добавление одной новой школы-получателя в список получателей одной из школ.

Формат входных данных

Первая строка содержит число n школ в сети ($2 \leqslant n \leqslant 100$). Школы нумеруются первыми n положительными целыми числами.

Каждая из следующих n строк задаёт список получателей. Строка $i + 1$ содержит номера получателей i-й школы. Каждый такой список заканчивается нулём. Пустой список содержит только ноль.

Формат выходных данных

В первой строке выведите решение первой подзадачи.

Во второй строке — минимальное суммарное число k расширений списков.

В следующих k строках выведите пары (a_i, b_i) чисел, где b_i — номер школы, которая добавляется в список получателей школы a_i. Если решение задачи неоднозначно, выдайте любое подходящее расширение списка получателей.

Пример

входной файл	*выходной файл*
5 2 4 3 0 4 5 0 0 0 1 0	1 2 3 1 4 1

Задача 28. Расписание

Имеется одна машина и n работ, которые должны быть выполнены на этой машине. Некоторые из работ должны предшествовать другим. Правило предшествования работ задаётся ациклическим орграфом: дуга (i, j) означает, что работа i должна предшествовать работе j. Для каждой работы i определены две величины: длительность выполнения p_i и директивный срок завершения d_i.

Если дана некоторая последовательность X работ, то через $c(x_i)$ будем обозначать время завершения работы i при выполнении её в последовательности работ X. Директивный срок d_i — это срок (от момента начала выполнения всех работ), к которому работа i должна быть выполнена, чтобы не платить штраф.

Штраф для работы i при выполнении её в последовательности X работ определяется как величина $\max\{\, c(x_i) - d_i, 0 \,\}$.

Необходимо определить последовательность Y выполнения всех работ, чтобы минимизировать функцию

$$\max_{i=1,\ldots,n} \{c(y_i) - d_i, 0\}.$$

Определите для найденной последовательности Y работ номер работы с максимальным штрафом и сам максимальный штраф.

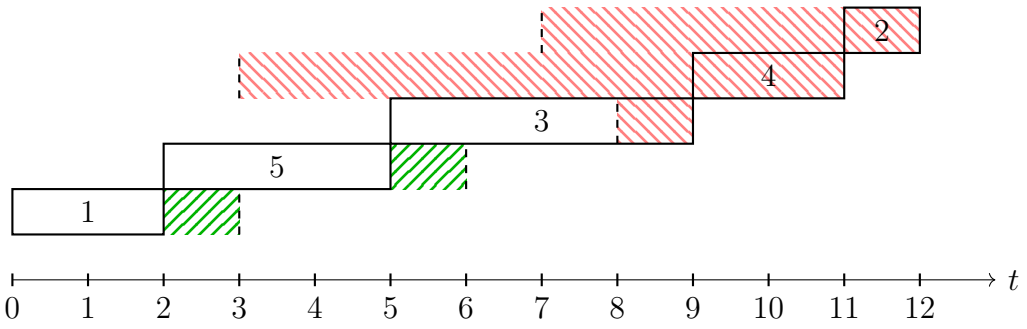


Рис. 7.8. Иллюстрация к примеру

Формат входных данных

Первая строка содержит число n работ ($1 \leqslant n \leqslant 200\,000$).

Каждая из следующих n строк содержит по два числа: p_i и d_i ($1 \leqslant p_i, d_i \leqslant 10^9$).

Строка $n + 2$ содержит число m дуг орграфа ($1 \leqslant m \leqslant 700\,000$).

Каждая из последующих m строк содержит два числа i и j, задающих дугу орграфа ($1 \leqslant i, j \leqslant n$, $i \neq j$).

Формат выходных данных

В первой строке выведите номер работы с максимальным штрафом и сам максимальный штраф.

В каждой из следующих n строк выведите номер работы в оптимальной последовательности (т. е. работы выполнятся в том порядке, в котором они следуют в оптимальной последовательности).

В случае неоднозначности решения выдайте любое из них.

Пример

входной файл	*выходной файл*	*пояснение*
5 2 3 3 7 4 8 2 3 3 6 4 1 2 1 5 5 4 3 4	4 8 1 5 3 4 2	*рис. 7.8*

Задача 29. Пересечение реки

На реке имеется n островков, пронумерованных от 1 до n слева направо поперёк реки. Жители должны пересечь реку, начиная с её левого берега и используя некоторые островки для достижения правого берега. Левый берег расположен на одну позицию левее первого островка, а правый берег расположен на одну позицию правее островка с номером n.

В момент времени $t = 0$ житель находится на левом берегу реки и имеет целью добраться до правого берега за минимальное время. В каждый момент времени каждый из островков поднят или опущен и житель стоит на островке или на берегу. Житель может стоять на островке только тогда, когда он поднят. Такой островок называется доступным. В начальный момент времени каждый островок опущен, затем островок поднят a моментов времени, затем опущен b моментов времени, поднят a моментов, опущен b и т. д. Константы a и b задаются отдельно для каждого островка. Например, островок, характеризующийся параметрами $a = 2$ и $b = 3$, опущен в момент времени 0, поднят в моменты времени 1 и 2, опущен в моменты времени 3, 4 и 5 и т. д. В момент времени $t + 1$ житель может выбрать островок или берег в пределах пяти ближайших с каждой стороны от его местоположения в момент времени t или остаться на месте (если возможно).

Необходимо вычислить минимальный момент достижения правого берега, если такое достижение возможно.

Формат входных данных

Первая строка содержит число островков ($5 \leqslant n \leqslant 1000$).

Каждая из последующих n строк содержит два натуральных числа a и b ($1 \leqslant a, b \leqslant 5$). Числа в $(i+1)$-й строке описывают поведение i-го островка.

Формат выходных данных

Выведите минимальный момент достижения правого берега или сообщение No, если пересечение реки невозможно.

Примеры

входной файл	*выходной файл*
10 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	No
10 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1	4

Задача 30. Flood it!

«Flood it!» — это простая, но крайне увлекательная игра. Правила её не очень затейливы: имеется прямоугольное поле, которое разбито на n строк и m столбцов. Перед началом игры каждая из $n \times m$ клеток покрашена в один из k цветов. Прежде чем продолжать изложение условий игры, дадим несколько определений. Две клетки назовём *соседними*, если они имеют общую сторону. Несколько клеток назовём *областью*, если выполняются следующие условия:

• между любыми двумя клетками области существует маршрут, проходящий по соседним клеткам этой области;

- все клетки области окрашены в один цвет — цвет области;
- ни одна клетка, не принадлежащая области, но соседствующая с ней, не окрашена в цвет области.

Цель игры состоит в захвате максимально возможного числа клеток игрового поля. Изначально во владение игрока отдаётся область, содержащая верхнюю левую клетку. На каждом ходу игрок выбирает какой-либо цвет из k заданных и красит свои владения в этот цвет. Если перед ходом с одной их захваченных ранее клеток соседствует область, имеющая выбранный цвет, то игрок захватывает эту область.

На основании исходного состояния игрового поля и информации о первых t ходах игрока нарисуйте игровое поле, получившееся после этих ходов.

Формат входных данных

В первой строке заданы четыре целых положительных числа: n и m ($1 \leqslant n, m \leqslant 1000$) — размеры поля, k ($1 \leqslant k \leqslant 1\,000\,000$) — число различных цветов и t ($1 \leqslant t \leqslant 100\,000$) — длина последовательности ходов.

Далее следует n строк, в каждой из которых записано по m чисел в диапазоне от 1 до k, где числа обозначают цвета начальной раскраски.

В последней строке даны t чисел в диапазоне от 1 до t — последовательность ходов.

Формат выходных данных

Необходимо вывести n строк, в каждой из которых записано m чисел в диапазоне от 1 до k — состояние игрового поля по прошествии t заданных ходов.

Примеры

входной файл	*выходной файл*
3 4 5 3 1 2 3 5 2 3 1 1 2 1 3 2 2 3 2	2 2 2 5 2 2 1 1 2 1 3 2
2 5 6 2 1 3 3 3 6 2 4 6 3 4 3 1	1 1 1 1 6 2 4 6 1 4

Задача 31. Химическая реакция

Некоторая матрица A размера $n \times n$ задаёт результаты химической реакции веществ. Все вещества имеют номера от 1 до n. Элемент a_{ij} матрицы равен номеру того вещества, которое получается в результате

химической реакции вещества i с веществом j (если вещества не вступают в реакцию, то $a_{ij} = 0$). Задана пробирка, в которую последовательно добавляются некоторые химические вещества. Вещество i вступает в химическую реакцию с веществом j, если находится в пробирке непосредственно над ним. Если вещества не вступают в реакцию, то они не смешиваются в пробирке.

Необходимо определить, какие вещества и в какой последовательности будут находиться в пробирке после того, как все вещества будут добавлены в пробирку.

Формат входных данных

Первая строка содержит число n различных веществ ($1 \leqslant n \leqslant 300$) и число m ($1 \leqslant m \leqslant 1\,000\,000$) добавляемых веществ.

Следующие n строк содержат элементы матрицы, каждая строка содержит элементы соответствующей строки матрицы. Гарантируется, что матрица является симметричной ($a_{ij} = a_{ji}$ для всех i, j) и одинаковые вещества между собой не реагируют, а смешиваются ($a_{ii} = i$ для всех i).

Последняя строка содержит m чисел, которые соответствуют номерам веществ. Номера химических веществ следуют в строке в той последовательности, в которой они добавлялись в пробирку.

Формат выходных данных

Выведите номера химических веществ, которые получились в пробирке, начиная от верхнего уровня и заканчивая нижним уровнем.

Пример

входной файл	*выходной файл*
3 4 1 3 2 3 2 1 2 1 3 1 3 1 2	1

Задача 32. Равенства и неравенства

Задачи, похожие на эту, возникают в области анализа текстов программ, автоматического доказательства теорем и искусственного интеллекта.

Для набора переменных x_1, x_2, …, x_n вам дан список ограничений-равенств вида $x_i = x_j$ и ограничений-неравенств вида $x_i \neq x_j$. Возможно ли подобрать такие целочисленные значения переменных x_1, x_2, …, x_n, чтобы все ограничения выполнялись одновременно?

Формат входных данных

В первой строке заданы число n переменных ($1 \leqslant n \leqslant 100\,000$) и число m ограничений ($1 \leqslant m \leqslant 100\,000$).

Затем в m строках заданы сами ограничения в формате «переменная», «оператор», «переменная». Переменные задаются так: буква `x`, затем номер — целое число от 1 до n включительно. Оператор — это либо `==` (равно), либо `!=` (не равно).

Формат выходных данных

Выведите ответ на вопрос задачи — `Yes` или `No`.

Примеры

входной файл	*выходной файл*
3 2 x1 == x2 x2 != x3	Yes
3 3 x1 == x2 x2 == x3 x3 != x1	No

Задача 33. Парк

Парк имеет форму прямоугольника с вершинами $(0, 0)$, $(w, 0)$, (w, h), $(0, h)$. Внутри парка расположены n деревьев. Создатели парка явно жалеют времени и денег на поливку деревьев и удобрение почвы. Поэтому стволы деревьев настолько тонки, что сами деревья мы будем считать точками. Недавно горисполком высказал претензию дирекции парка. Дело в том, что в парке, кроме деревьев, ничего нет — ни аттракционов, ни даже детской площадки. Директор решил не испытывать судьбу и построить хотя бы детскую площадку. Чтобы чиновники из горисполкома остались довольны, должны выполняться три условия.

1. Площадка имеет форму прямоугольника со сторонами, параллельными осям координат.

2. Внутри площадки нет деревьев (хотя они могут быть на её границе) и площадка не выходит за границы парка.

3. Площадь площадки максимальна.

Формат входных данных

Первая строка содержит три целых числа n, w и h ($0 \leqslant n \leqslant 5000$, $1 \leqslant w, h \leqslant 30\,000$).

Следующие n строк описывают расположение деревьев. Каждое дерево описано в отдельной строке, содержащей его координаты x_i и y_i ($0 \leqslant x_i \leqslant w$, $0 \leqslant y_i \leqslant h$).

Формат выходных данных

Выведите максимально возможную площадь детской площадки.

Пример

входной файл	выходной файл	пояснение
3 5 5 1 3 2 2 4 4	12	рис. 7.9

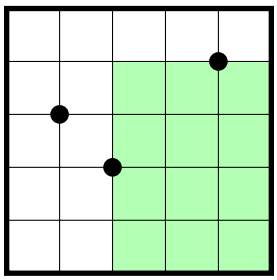


Рис. 7.9. Иллюстрация к примеру: площадка 3×4

Задача 34. Урок математики

Как-то раз на уроке по занимательной математике Мария Ивановна решила задать классу интересную задачку. Для этого на левой части доски она выписала n целых чисел $a_1, a_2, \ldots, a_n$. На правой части доски она решила выписать в произвольном порядке всевозможные парные суммы этих чисел, т. е. для каждого i и для каждого j $(1 \leqslant i, j \leqslant n)$ были выписаны числа $a_i + a_j$. Получилось, что на левой половине было n чисел, а на правой — ровно n^2 чисел. Но только Мария Ивановна дописала последнее число на правой половине доски, как прозвенел звонок. После перемены, когда все уже собрались в классе, ученики заметили, что какой-то злодей стёр все n чисел, которые находились на левой половине доски. Эта новость очень возмутила и расстроила учительницу, вследствие чего урок оказался на грани срыва. Ваша же задача не допустить, чтобы урок сорвался, т. е. вам необходимо по n^2 числам, которые остались на правой половине, восстановить исходные n чисел.

Формат входных данных

В первой строке записано одно целое число n $(1 \leqslant n \leqslant 1500)$. Во второй строке — n^2 чисел, все возможные попарные суммы.

Все числа целые положительные и не превосходят $2 \cdot 10^9$.

Формат выходных данных

Выведите n чисел a_i, по одному в строке. Числа должны быть выведены в неубывающем порядке.

Примеры

входной файл	*выходной файл*
2 5 4 5 6	2 3
3 8 6 10 8 6 8 6 6 8	3 3 5

Задача 35. Космическая экспедиция

В 2004 году обитатели планеты Кремонид организовали космическую экспедицию для полёта в соседнюю галактику, где по их расчётам существует планета, пригодная для жизни. На космическом корабле был сконструирован жилой комплекс, куда заселили множество учёных.

Жилой комплекс имеет форму прямоугольного параллелепипеда размера $n \times m \times k$. Комплекс разбит на кубические отсеки размера $1 \times 1 \times 1$, всего $n \times m \times k$ отсеков.

Каждый отсек имеет координаты (x, y, z), соответствующие положению отсека в комплексе, где $1 \leqslant x \leqslant n$, $1 \leqslant y \leqslant m$, $1 \leqslant z \leqslant k$.

Расстоянием между двумя отсеками с координатами (x_1, y_1, z_1) и (x_2, y_2, z_2) назовём число $|x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$.

Два отсека находятся *в одном ряду*, если их координаты отличаются ровно одной компонентой (например, $(2, 4, 3)$ и $(2, 6, 3)$ находятся в одном ряду).

Два отсека являются *соседними*, если расстояние между ними равно 1.

В каждый отсек был установлен персональный компьютер. После взлёта жители комплекса решили объединить свои компьютеры в сеть. Был разработан план прокладывания сети, который представляет собой следующую процедуру.

• Выбираются два отсека, находящихся в одном ряду. Первый отсек назовём начальным, второй — конечным. Робот, прокладывающий сеть, стартует в начальном отсеке.

• На каждом шаге робот передвигается в тот соседний отсек, расстояние от которого до конечного минимально.

• При этом он соединяет пары компьютеров в соседних отсеках, через которые он проходит, если это не приводит к образованию цикла. (*Циклом* называется последовательность компьютеров, в которой

первый связан со вторым, второй с третьим и т. д., а последний — с первым.) Если же соединение приводит к образованию цикла, то робот запоминает координаты этой пары соседних отсеков и не соединяет компьютеры в них между собой.

• Робот перемещается, пока не достигнет конечного отсека.

Указанная процедура повторяется t раз. Необходимо определить, какие пары отсеков запомнил робот.

Формат входных данных

Первая строка содержит четыре числа n, m, k и t ($2 \leqslant n, m, k \leqslant 100$, $1 \leqslant t \leqslant 20\,000$).

Далее следует t строк, из которых $(i+1)$-я описывает пару отсеков, между которыми продвигается робот, и содержит шесть чисел: первые три числа — координаты начального отсека, оставшиеся три числа — координаты конечного отсека.

Формат выходных данных

Выведите последовательность строк, описывающих пары отсеков, которые запомнил робот в хронологическом порядке.

В каждой строке выведите шесть чисел — координаты отсеков в паре в порядке прохождения их роботом.

Пример

входной файл	*выходной файл*
4 5 6 6 1 1 1 1 1 3 1 1 4 1 4 4 1 1 2 1 1 4 4 1 1 4 1 4 1 1 1 4 1 1 1 1 4 4 1 4	1 1 2 1 1 3 3 1 4 4 1 4

Задача 36. Звёздочки

Астрономы часто изображают карту звёздного неба на бумаге, где каждая звезда имеет декартовы координаты. Пусть уровнем звезды будет число других звёзд, которые расположены на карте не выше и не правее данной звезды (т. е. каждой звезде можно присвоить какой-то определённый уровень). И вот астрономы решили узнать уровни всех звёзд. Но за этим они обратились к вам.

Вам требуется определить распределение звёзд по уровням, т. е. определить, сколько звёзд каждого уровня имеется на карте.

Формат входных данных

В первой строке записано число n звёзд на карте ($1 \leqslant n \leqslant 300\,000$).

Далее следуют n строк, в каждой из которых расположено по два числа x и y ($0 \leqslant x, y \leqslant 500\,000$) — координаты i-й звезды.

Звёзды упорядочены по возрастанию y-координаты. Звёзды с одинаковой y-координатой упорядочены по возрастанию x-координаты. Никакие две звезды не имеют одинаковые координаты.

Формат выходных данных

Выведите n строк: в первой — число звёзд нулевого уровня, во второй — первого и т. д.

Пример

входной файл	*выходной файл*	*пояснение*
6 1 1 3 1 5 1 3 2 5 2 2 3	1 2 2 0 1 0	*рис. 7.10*

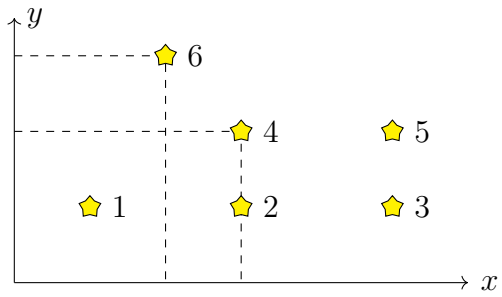


Рис. 7.10. Иллюстрация к первому примеру:
уровень звезды 6 равен 1, уровень звезды 4 равен 2

Задача 37. Дороги

В Байтландии n городов. Некоторые пары городов соединены между собой двусторонними дорогами. Сеть дорог позволяет из любого города проехать в любой другой. К сожалению, из-за недостатка финансирования дороги год за годом приходят в упадок и выводятся из эксплуатации. Известен порядок, в котором будут закрываться все дороги. Определите, в какой момент при закрытии очередной дороги жители Байтландии потеряют возможность проехать из какого-либо одного города в другой город.

Формат входных данных

В первой строке через пробел записаны числа n и m ($1 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 300\,000$) — число городов и число дорог соответственно.

В последующих m строках заданы все дороги Байтландии в хронологическом порядке их закрытия. Дорога описывается парой чисел u и v ($1 \leqslant u, v \leqslant n$, $u \neq v$). Между парой городов может быть построено несколько дорог.

Формат выходных данных

Выведите одно число — номер дороги, после закрытия которой дорожная сеть утратит связность. Дороги нумеруются начиная с единицы в соответствии с порядком следования на входе.

Пример

входной файл	*выходной файл*
3 3 1 2 2 3 3 1	2

Задача 38. Корпоративная сеть

Очень большая корпорация решила разработать свою собственную сеть. Вначале каждое из n предприятий этой корпорации, пронумерованных целыми числами от 1 до n, имеет свой вычислительный и телекоммуникационный центр.

Для повышения качества сервиса корпорация начала собирать некоторые предприятия в кластеры, каждый из которых обслуживается одним телекоммуникационным центром. Для этой цели корпорация выбирает один из существующих центров (пусть это будет центр i, который обслуживает кластер A) и одно из предприятий j, находящееся в каком-то другом кластере B (j не обязательно является центром). Затем i и j связываются телекоммуникационной линией. Длина линии между предприятиями i и j вычисляется по формуле $|i - j| \bmod 1000$. Таким образом, два старых кластера объединяются в один новый, причём телекоммуникационным центром нового кластера объявляется телекоммуникационный центр кластера B (рис. 7.11).

Если телекоммуникационный сервер j обслуживает кластер A и предприятие i находится в кластере A, то будем говорить, что j является обслуживающим центром для i.

Но, к сожалению, после каждого такого объединения длина телекоммуникационной линии между предприятием и обслуживающим его центром может меняться, и руководство корпорации хочет знать эту длину. Напишите программу, которая позволила бы корпорации узнавать расстояния от предприятия до центра в любой момент времени.

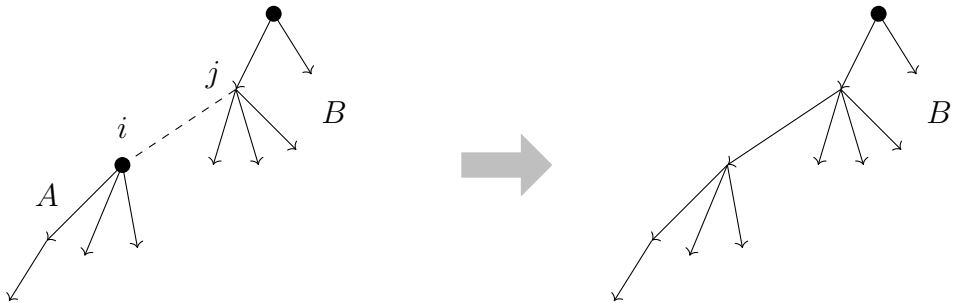


Рис. 7.11. Схема объединения кластеров

Формат входных данных

Первая строка содержит число n предприятий ($4 \leqslant n \leqslant 200\,000$).

Следующие строки описывают команды, подаваемые корпорацией. Число команд не превосходит 200 000.

Формат команд:

- `E` i — запрос длины пути между предприятием i и обслуживающим его телекоммуникационным центром;
- `I` i j — информирует о том, что обслуживающий центр i присоединяется к предприятию j;
- `0` — говорит о том, что корпорация заканчивает свои команды (конец входных данных).

Формат выходных данных

Выведите по одной строке на каждую команду `E` во входных данных. Каждая такая строка должна содержать единственное число — запрашиваемую длину между предприятием и обслуживающим его телекоммуникационным центром.

Пример

входной файл	*выходной файл*
4 E 3 I 3 1 E 3 I 1 2 E 3 I 2 4 E 3 0	0 2 3 5

Задача 39. Мобильники

Предположим, что в регионе Тампере базовые станции обеспечения мобильной телефонной связи четвёртого поколения действуют следующим образом.

Регион поделён на квадраты. Квадраты образуют таблицу размера $s \times s$, строки и столбцы которой пронумерованы от 0 до $s - 1$.

В каждом квадрате находится базовая станция. Число работающих мобильных телефонов внутри квадрата может меняться, так как телефоны могут перемещаться из одного квадрата в другой, а также включаться или выключаться. В некоторые моменты времени каждая базовая станция передаёт головной базовой станции отчёт об изменении числа работающих телефонов и свои координаты (номер строки и номер столбца соответственно).

Напишите программу, которая получает эти отчёты и отвечает на запросы о текущем общем числе работающих мобильных телефонов в некоторой прямоугольной области.

Формат входных данных

Каждая строка содержит одну команду. Команда состоит из кода и набора параметров (целых чисел) в соответствии со следующим списком.

• Команда 0, параметр s ($1 \leqslant s \leqslant 10^9$). Инициализирует таблицу размера $s \times s$ нулями. Эта команда выдаётся только один раз и должна быть первой командой.

• Команда 1, параметры x, y, a ($0 \leqslant x, y < s$, $-10^9 \leqslant a \leqslant 10^9$). Увеличивает число работающих мобильных телефонов в квадрате (x, y) на a. Число a может быть как положительным, так и отрицательным.

• Команда 2, параметры x_1, y_1, x_2 и y_2 (где $0 \leqslant x_1 \leqslant x_2 < s$ и $0 \leqslant y_1 \leqslant y_2 < s$). Запрашивает общее число работающих мобильных телефонов во всех квадратах (x, y), где $x_1 \leqslant x \leqslant x_2$, $y_1 \leqslant y \leqslant y_2$.

• Команда 3. Завершает программу. Эта команда выдаётся только один раз и должна быть последней.

Индексы в таблице начинаются от 0. Число команд на входе не превосходит 200 000.

Формат выходных данных

Выдайте ответы на все команды с кодом 2 в порядке их поступления, по одному в строке.

Пример

входной файл	*выходной файл*
0 4 1 1 2 3 2 0 0 2 2 1 1 1 2 1 1 2 -1 2 1 1 2 3 3	3 4

Задача 40. Ленивый программист

Недавно образовавшаяся студия веб-дизайна SMART (Simply Masters of ART) имеет в своём активе пока только двух работников: дизайнера, по совместительству занимающего должность директора студии, и программиста. Благодаря расторопности директора студия сразу после своего образования получила n заказов на разработку веб-сайтов. Для i-го заказа известен крайний срок d_i, до которого студия должна выполнить его (измеряемый в некоторых условных единицах времени с начала образования студии).

Известно, что программист очень ленив и не торопится выполнять заказы. На выполнение i-го заказа ему необходимо b_i единиц времени. К счастью, программист согласен ускорить темп своей работы за дополнительную плату. В частности, если директор заплатит ему x_i долларов, то он выполнит i-й заказ за $b_i - a_i \cdot x_i$ единиц времени (время выполнения остальных заказов не изменится — за каждый заказ нужно доплачивать отдельно).

Как видите, программист настолько любит деньги, что способен выполнить заказ даже мгновенно, если ему заплатят достаточную сумму. Но выполнять заказы за отрицательное время он пока ещё не научился, поэтому не имеет смысла доплачивать за выполнение i-го заказа более чем b_i/a_i долларов.

Теперь перед директором студии стоит непростая задача: организовать рабочее время и поощрение программиста так, чтобы он выполнил все заказы вовремя, и при этом доплатить программисту минимальную сумму денег.

Формат входных данных

Первая строка содержит целое число n заказов ($1 \leqslant n \leqslant 100\,000$).

Следующие n строк описывают заказы: i-я строка описывает i-й заказ и содержит целые числа a_i, b_i, d_i ($1 \leqslant a_i, b_i \leqslant 10\,000$; $1 \leqslant d_i \leqslant 10^9$).

Формат выходных данных

В единственной строке выведите минимальную сумму денег, которую нужно доплатить программисту, чтобы он, при правильной организации своего рабочего времени, выполнил все заказы вовремя.

Абсолютная погрешность ответа не должна превосходить 10^{-2}.

Пример

входной файл	*выходной файл*
1 8398 8019 7290	0.09

Задача 41. Тройное доминирование

В командной олимпиаде ФПМИ по программированию участвуют n команд. Каждая команда состоит из трёх участников. Так как олимпиада является командной, то, даже располагая сведениями о силах каждого из участников команды, довольно сложно делать какие-либо выводы о силе команды. Поэтому затруднительно делать какие-нибудь прогнозы относительно того, кто станет победителем олимпиады. Однако буквально за несколько дней до командной олимпиады ФПМИ прошла личная олимпиада ФПМИ, и её результаты уже известны. Таким образом, для каждого участника командной олимпиады известно место, которое он занял в личной. Имея эту информацию, можно попытаться сузить круг кандидатов в победители командной олимпиады.

Пронумеруем команды от 1 до n и обозначим через x_i, y_i и z_i места, которые заняли участники i-й команды в личной олимпиаде. Переставлять участников в команде нельзя, т. е. (x_i, y_i, z_i) — это не множество, а упорядоченная тройка.

Будем говорить, что команда i *доминирует над командой* j, если выполняются неравенства $x_i < x_j$, $y_i < y_j$ и $z_i < z_j$. Понятно, что если команда i доминирует над командой j, то у команды j вряд ли есть шанс стать победителем олимпиады. Поэтому имеет смысл назвать команду i кандидатом в победители, если никакая другая команда не доминирует над ней.

Напишите программу, которая по результатам личной олимпиады определяет число команд – кандидатов в победители.

Формат входных данных

Первая строка содержит число n команд ($1 \leqslant n \leqslant 100\,000$).

Следующие n строк описывают места, которые заняли в личной олимпиаде участники каждой из команд. Каждая из этих строк содержит три числа x_i, y_i и z_i ($1 \leqslant x_i, y_i, z_i \leqslant 3n$). Никакие два участника личной олимпиады не разделили место между собой.

Формат выходных данных

Выведите искомое число команд-кандидатов.

Пример

входной файл	*выходной файл*
5 11 13 10 5 6 1 4 15 3 8 14 9 7 12 2	2

Задача 42. Соревнования

Для выявления лучшего программиста в стране решили провести соревнования по программированию. В соревнованиях было зарегистрировано n человек. Было проведено три соревнования, в каждом из которых участвовали все зарегистрировавшиеся (известно, что никакие два участника не имели одинаковые результаты в одном и том же соревновании). И вот перед жюри конкурса встал один вопрос: как же, всё-таки, выбрать лучшего?

Будем говорить, что участник a лучше участника b, если участник a во всех соревнованиях занял места выше, чем участник b. Назовём участника c лучшим, если нет такого участника, который лучше участника c. С целью выявления числа лучших участников жюри конкурса наняло вас. Помогите жюри найти число лучших среди всех зарегистрированных участников.

Формат входных данных

Первая строка содержит число n зарегистрированных участников ($1 \leqslant n \leqslant 100\,000$).

Далее, в последующих n строках записаны по три числа a_i, b_i, c_i ($1 \leqslant a_i, b_i, c_i \leqslant n$) — места в каждом из трёх соревнований i-го участника.

Формат выходных данных

Выведите число лучших участников.

Пример

входной файл	*выходной файл*
3 1 2 3 2 3 1 3 1 2	3

Задача 43. Кодирование Хаффмана

Кодирование Хаффмана (D. A. Huffman) относится к префиксному кодированию, позволяющему минимизировать длину текста за счёт того, что различные символы кодируются различным числом битов.

Напомним процесс построения кода. Вначале строится дерево кода Хаффмана. Пусть исходный алфавит состоит из n символов, i-й из которых встречается p_i раз во входном тексте. Изначально все символы считаются активными вершинами будущего дерева, i-я вершина помечена значением p_i. На каждом шаге мы берём две активных вершины с наименьшими метками, создаём новую вершину, помечая её

суммой меток этих вершин, и делаем её их родителем. Новая вершина становится активной, а двое её сыновей из списка активных вершин удаляются. Процесс многократно повторяется, пока не останется только одна активная вершина, которая полагается корнем дерева.

Заметим, что символы алфавита представлены листьями этого дерева. Для каждого листа (символа) длина его кода Хаффмана равна длине пути от корня дерева до него. Сам код строится следующим образом: для каждой внутренней вершины дерева рассмотрим две дуги, идущие от неё к сыновьям. Одной из дуг присвоим метку 0, другой — 1. Код каждого символа — последовательность из нулей и единиц на пути от корня к листу.

Задача состоит в том, чтобы вычислить длину текста после его кодирования методом Хаффмана. Сам текст не дан, известно лишь, сколько раз каждый символ встречается в тексте. Этого достаточно для решения задачи, поскольку длина кода зависит только от частоты появления символов.

Формат входных данных

Первая строка содержит целое число n ($2 \leqslant n \leqslant 500\,000$).

Вторая строка содержит n чисел p_i — частоты появления символов в тексте ($1 \leqslant p_i \leqslant 10^9$, $p_i \leqslant p_{i+1}$ для каждого i от 1 до $n-1$).

Формат выходных данных

Выведите единственное число — длину (в битах) закодированного текста.

Пример

входной файл	*выходной файл*	*пояснение*
6 1 1 2 2 5 8	42	*рис. 7.12*

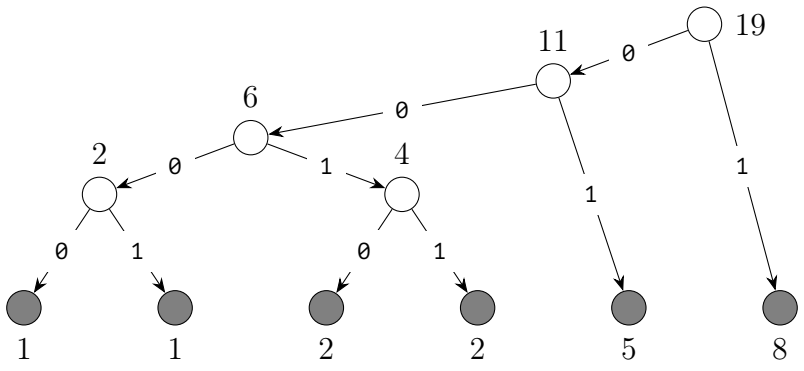


Рис. 7.12. Один из вариантов дерева кодирования Хаффмана

Задача 44. Слияние возрастающих последовательностей

Дано n упорядоченных по возрастанию последовательностей целых чисел, каждая из которых имеет длину m.

Необходимо слить их в одну последовательность длины $m \cdot n$, также упорядоченную по возрастанию.

Сложность алгоритма не должна превышать $O(nm \log n)$, затраты памяти — $O(nm)$.

Формат входных данных

В первой строке заданы два целых числа n и m ($1 \leqslant n, m \leqslant 1000$).

В следующих n строках — по m упорядоченных по возрастанию целых чисел в каждой. Числа не превосходят по модулю 10^9.

Формат выходных данных

Выведите итоговую последовательность.

Примеры

входной файл	*выходной файл*
2 5 1 3 5 7 9 2 4 6 8 10	1 2 3 4 5 6 7 8 9 10
4 2 1 4 2 8 3 7 5 6	1 2 3 4 5 6 7 8

Задача 45. Мегаинверсии

Мегаинверсией в перестановке $p_1, p_2, \ldots, p_n$ назовём тройку (i, j, k), для которой $i < j < k$ и $p_i > p_j > p_k$.

Формат входных данных

Первая строка содержит целое число n ($1 \leqslant n \leqslant 300\,000$).

Следующие n строк описывают перестановку: i-я строка содержит одно целое число p_i ($1 \leqslant p_i \leqslant n$; все p_i попарно различны).

Формат выходных данных

Выведите число мегаинверсий в перестановке $p_1, p_2, \ldots, p_n$.

Пример

входной файл	*выходной файл*
4 4 3 2 1	4

Задача 46. Отрицательные скидки

Во многих магазинах действует принцип: чем больше вы покупаете, тем меньше стоит единица товара. Но если товар дефицитный и его предложение ограниченно, то может происходить обратное.

Представьте, что вам необходимо приобрести ровно k единиц некоторого редкого товара. Вы выяснили, что он есть в наличии в n магазинах (для простоты пронумеруем их числами от 1 до n). Цена товара в i-м магазине увеличивается с ростом объёма покупки: первая единица товара стоит a_i у. е., вторая — $a_i + b_i$ у. е., третья — $a_i + 2b_i$ у. е. и т. д.

Магазин сможет предоставить вам любое количество товара, но это может оказаться весьма дорого. При этом будем считать, что магазин ведёт учёт покупателей, и поэтому нельзя сделать несколько маленьких покупок подряд вместо одной большой, а также нельзя попросить сделать покупку другого человека.

Ваша задача — решить, сколько единиц товара купить в каждом магазине, чтобы в итоге оказалось куплено ровно k и при этом потрачено как можно меньше денег.

Формат входных данных

В первой строке записаны число n магазинов и число k единиц товара, при этом $1 \leqslant n \leqslant 200\,000$ и $1 \leqslant k \leqslant 100\,000$.

Затем следуют n строк, каждая описывает один магазин и содержит два целых числа a_i и b_i ($1 \leqslant a_i, b_i \leqslant 1000$) — коэффициенты, участвующие в ценообразовании (в у. е.).

Формат выходных данных

Выведите одно число — минимальную сумму в у. е., необходимую для осуществления закупок.

Пример

входной файл	*выходной файл*
4 4 10 2 9 3 5 2 4 10	25

Замечание

Такой результат можно получить, купив одну единицу товара во втором магазине (9 у. е.), две единицы товара в третьем ($5 + 7 = 12$ у. е.) и одну единицу в последнем (4 у. е.).

Задача 47. Хорды

На окружности отмечено $2n$ различных точек, пронумерованных от 1 до $2n$ против часовой стрелки. Петя нарисовал n хорд, i-я из которых соединяет точки с номерами a_i и b_i. При этом каждая точка является концом ровно одной хорды. Теперь Петя заинтересовался, сколько пар хорд пересекаются. Помогите ему найти это число.

Формат входных данных

Первая строка содержит число n ($1 \leqslant n \leqslant 300\,000$) проведённых хорд.

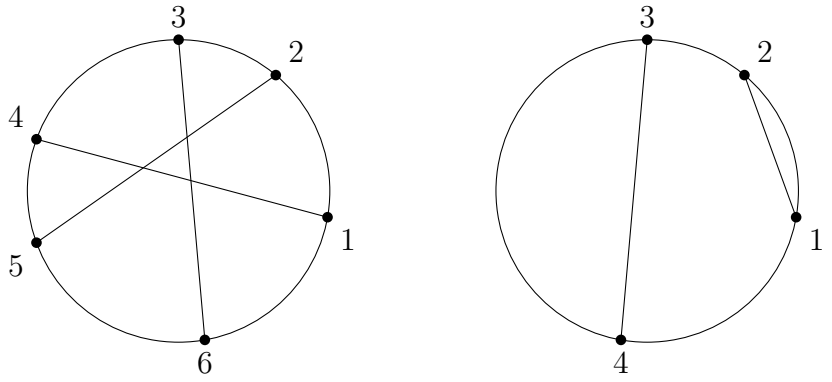
Следующие n строк содержат по два числа a_i и b_i ($1 \leqslant a_i, b_i \leqslant 2n$, $a_i \neq b_i$).

Формат выходных данных

Выведите число пар хорд, которые пересекаются.

Примеры

входной файл	*выходной файл*	*пояснение*
3 1 4 2 5 3 6	3	*рис. 7.13, а*
2 1 2 3 4	0	*рис. 7.13, б*



а *б*

Рис. 7.13. Схемы расположения хорд: *а* — первый пример; *б* — второй пример

Задача 48. Архив

В историческом архиве недавно случился аврал: поступило воистину чудовищное число заявок от различных исследовательских институтов, и чтобы успеть все их удовлетворить, работники архива начали использовать сильно упрощённую процедуру выдачи документов.

На примере одного шкафа с документами новая стратегия может быть описана следующим образом: в шкафу изначально лежит стопка из n документов, пронумерованных от 1 до n в порядке их следования от вершины стопки. Кроме того, каждый документ имеет учётный номер, который в начале аврала странным образом совпадал с номером документа в стопке. Когда приходит запрос на получение некоторого документа, архивариус, полагаясь на свой богатый опыт работы, извлекает из стопки пачку из подряд идущих документов с номерами от a до b включительно, ищет в ней нужный документ, снимает с него копию, после чего кладёт вынутый документ в то же место в пачке, из которого он его достал, а саму пачку кладёт на верх стопки. Таким образом, документы с позициями от a до b включительно перемещаются на позиции от 1 до $b - a + 1$ включительно.

Такая стратегия позволила пережить аврал, однако порядок документов в стопке был нарушен, и потому следовало бы восстановить первоначальный порядок документов, когда документ на позиции i имеет учётный номер i. На самих документах регистрационный номер не значится (их ведь нельзя повреждать), а потому придётся расположение документов в стопке определять исходя из записей архивариуса, работавшего с этой стопкой.

Формат входных данных

В первой строке находятся числа n и q — число документов в стопке и число раз, когда архивариус вытягивал пачку документов и клал её на вершину стопки ($1 \leqslant n, q \leqslant 300\,000$).

В строках со второй по $(q + 1)$-ю включительно находятся пары (a_i, b_i) натуральных чисел ($1 \leqslant a_i \leqslant b_i \leqslant n$), означающих номер первого и последнего документа в пачке, считая от вершины стопки.

Запросы следуют в порядке поступления.

Формат выходных данных

Выведите n натуральных чисел от 1 до n — учётные номера документов в порядке следования от вершины стопки в конце аврала.

Пример

входной файл	*выходной файл*
6 3 2 4 3 5 2 2	1 4 5 2 3 6

Задача 49. Продажа билетов

Вас наняла в качестве консультанта Главная Железнодорожная Компания Абсурдляндии (ГЖКА). Железнодорожная сеть этой замечательной страны представляет собой цепочку из n перегонов между $n+1$ городом. В самом скором времени из города 1 (первого города цепочки) отправится поезд в город $n+1$ (последний город цепочки). Компания предоставила вам данные о том, сколько мест будет свободно в поезде на каждом перегоне. Ваша задача — ответить на q запросов вида «какое максимальное число билетов от города u_j до города v_j можно продать». Разумеется, нельзя продавать билеты k пассажирам, если на каком-то перегоне между городами u_j и v_j свободно менее чем k мест.

Формат входных данных

Первая строка входа содержит n — число перегонов в железнодорожной сети ($1 \leqslant n \leqslant 300\,000$).

Следующая строка содержит n чисел a_i ($1 \leqslant a_i \leqslant 10^9$). Число a_i определяет, сколько будет свободных мест в поезде на перегоне между городами i и $i+1$.

Далее следует строка, содержащая q ($1 \leqslant q \leqslant 1\,000\,000$) — число запросов.

После этого следуют q запросов вида u_j, v_j (где $1 \leqslant u_j < v_j \leqslant n+1$).

Формат выходных данных

Выведите q строк, каждая из строк должна содержать одно число. j-я строка должна содержать ответ на j-й запрос — максимальное число билетов, которое можно продать.

Пример

входной файл	*выходной файл*	*пояснение*
4 5 6 3 7 6 2 5 4 5 1 2 4 5 3 5 1 3	3 7 5 7 3 5	*рис. 7.14*

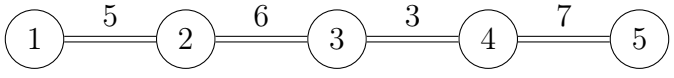


Рис. 7.14. Схема железнодорожной сети

Задача 50. Скользящая медиана

Медиана в математической статистике — число, характеризующее выборку (например, набор чисел). Медиану можно найти, упорядочив элементы выборки по неубыванию и взяв средний элемент. Например, выборка $(11, 9, 3, 5, 5)$ после упорядочивания превращается в $(3, 5, 5, 9, 11)$, и её медианой является число 5. Если в выборке чётное число элементов, в качестве медианы будем использовать полусумму двух соседних средних значений (т. е. медиану набора $(1, 3, 5, 7)$ считаем равной 4).

Поступают запросы двух видов: на добавление одного числа в выборку и на вычисление медианы. Нужно научиться эффективно их обрабатывать.

Формат входных данных

Целые числа (от 0 до 10^{18}) записаны в одну строку.

Если число положительное, то его нужно добавить в выборку.

Если число равно нулю, то нужно вычислить медиану выборки, которая была накоплена к данному моменту. Общее количество чисел на входе не превосходит 300 000. Гарантируется, что первое число не равно нулю.

Формат выходных данных

Для каждого числа 0 на входе выведите в отдельной строке полученное значение медианы с абсолютной погрешностью не более 0,1.

Примеры

входной файл	*выходной файл*
1 0 2 0 3 0	1 1.5 2
11 9 3 5 5 0	5
1 3 5 7 0	4
1 0 3 0 2 0 7 0 8 0 10 0 3 0 6 0 8 0 1 0 1 0 1 0	1 2 2 2.5 3 5 3 4.5 6 4.5 3 3

7.2. УКАЗАНИЯ К РЕШЕНИЮ ЗАДАЧ

1. Считалка. Моделирование действий считающего можно провести на массиве или двунаправленном списке. При этом следует учитывать, что если на некотором этапе число m больше, чем число человек n', которые к этому моменту остались в круге, то нужно предварительно взять остаток от деления m на n'. Время работы алгоритма — $O(n \cdot \min(n, m))$.

2. Полоска. Смоделируем сложение полоски, используя структуру данных «стек». Пронумеруем клетки полоски слева направо целыми числами от 1 до 2^k. Создадим массив размера 2^k, каждый элемент которого ассоциируется со стопкой клеток полоски (их номеров) после очередного сгибания. Для моделирования процесса сгибания удобно задавать каждый элемент массива, используя структуру данных «стек». В этом случае одно сгибание потребует выполнить ряд операций, при которых элементы одного стека добавляются к другому (при этом порядок следования номеров клеток в стеке будет противоположен тому, который получается при сложении полоски). После того как все 2^k клеток полоски окажутся в одном стеке, останется в соответствии с порядком следования их номеров в стеке занумеровать соответствующие клетки целыми числами от 2^k до 1. Время работы алгоритма — $O(k \cdot 2^k)$.

3. Квадрат. Смоделируем сложение квадрата, используя структуру данных «стек». Пронумеруем клетки квадрата слева направо и сверху вниз целыми числами от 1 до 4^k. Создадим матрицу размера $2^k \times 2^k$, каждый элемент которой ассоциируется со стопкой клеток квадрата (их номеров) после очередного сгибания (первоначально каждый элемент матрицы состоит из единственного элемента). Для моделирования процесса сгибания удобно задавать каждый элемент матрицы, используя структуру данных «стек». В этом случае одно сгибание потребует выполнить ряд операций, при которых элементы одного стека добавляются к другому (при этом порядок следования номеров клеток в стеке будет противоположен тому, который получается при сложении квадрата). После того как все 4^k клеток квадрата окажутся в одном стеке, останется в соответствии с порядком следования их номеров в стеке занумеровать соответствующие клетки целыми числами от 4^k до 1. Время работы алгоритма — $O(k \cdot 4^k)$.

4. Таблица. Для решения данной задачи можно использовать структуру данных «стек». Предположим, что на некотором этапе алгоритма в

стеке располагается информация о просмотренных элементах таблицы (в стеке будем хранить не сами элементы, а их индексы в таблице), причём справедливо, что элемент, который был занесён в стек раньше, больше того элемента, который был занесён в стек позже. Тогда если текущий элемент x таблицы меньше элемента y, который был занесён в стек последним, то информация об элементе x добавляется в стек. В противном случае из стека удаляются все элементы, которые меньше x, и их значения в таблице заменяются на x, после чего информация об элементе x добавляется в стек. На заключительном этапе, когда все элементы таблицы просмотрены, из стека удаляются все элементы, а соответствующие им элементы таблицы заменяются на 0. Время работы алгоритма — $O(n)$.

5. Остановки. Построим по входным данным задачи ориентированный граф. Если в некотором маршруте остановка встречается несколько раз, то столько раз данная вершина будет продублирована в орграфе. По условию задачи в орграфе веса дуг могут принимать одно из двух значений (1 или 3), поэтому для решения задачи можно использовать реализацию алгоритма Дейкстры с использованием структуры данных «приоритетная очередь», реализованной на двух очередях. Так как в оптимальном решении через одну остановку можно проехать несколько раз, то в алгоритме Дейкстры блокировать нужно не вершины, которые соответствуют остановкам, а дуги.

Ещё один, креативный, подход решения задачи заключается в том, что можно использовать базовый алгоритм поиска в ширину (структура данных «очередь»). Для этого нужно так перестроить орграф путём введения $2n$ фиктивных вершин, чтобы веса всех дуг были равны 1, и тогда кратчайший по стоимости маршрут совпадёт с маршрутом, наименьшим по количеству дуг. Время работы — $O(n + m)$, где m — суммарное количество остановок во всех маршрутах.

6. Король. Если бы фрагмент шахматной доски можно было представить в виде булевой матрицы и явно отмечать посещённые королём ячейки, то мы бы получили решение со временем работы $O(n)$. Однако поскольку король ходит в произвольных направлениях, нам может потребоваться матрица размера порядка $n \times n$. Такая матрица занимает слишком много места в памяти, а её инициализация будет требовать времени $O(n^2)$.

Для хранения координат ячеек, которые были посещены, будем

использовать хеш-таблицу. Получим алгоритм с временем работы в среднем $O(n)$.

7. Карточки. Решение задачи можно смоделировать на двух списках. Первый список соответствует карточкам, которые ещё не выложены на стол, второй — выложенным на стол. Сначала в первом списке n элементов — карточки с номерами 1, 2, ..., n. Если карточка кладётся на стол, то соответствующий элемент удаляется из первого списка и добавляется во второй. Если карточка кладётся под низ стопки, то соответствующий элемент первого списка удаляется из него и добавляется в конец этого списка. Действия продолжаются до тех пор, пока первый список не станет пустым. Теперь остаётся пройтись по второму списку, присваивая карточкам попеременно цвет: белая, чёрная, белая, чёрная и т. д. Время работы — $O(n)$.

8. Форма с погружением. Пусть матрица B (размера $n \times m$) содержит уровень воды над соответствующей клеткой (уровень воды равен высоте клетки плюс количество воды над ней). Поскольку вода всегда вытекает за края формы, то уровень воды над окаймляющими клетками формы равен их высоте. Для всех внутренних клеток формы уровень воды пока неизвестен, поэтому полагаем его равным либо бесконечности, либо наибольшему элементу матрицы высот A. Для определения уровня клеток формы можно использовать структуру данных «приоритетная очередь» (построенную на базе кучи). Элемент приоритетной очереди — тройка чисел: $(i, j, l_{i,j})$, где i и j — координаты клетки формы, а $l_{i,j}$ — её текущий уровень. Самой приоритетной считается тройка, у которой элемент $l_{i,j}$ наименьший.

Алгоритм

1. Заносим в кучу все окаймляющие клетки формы с уровнем, равным высоте $a_{i,j}$ соответствующих клеток формы. Все клетки формы считаем непросмотренными.

2. Пока куча не станет пустой, выполняем следующую последовательность шагов:

2.1 — удаляем из кучи элемент (с наименьшим значением $l_{i,j}$); предположим, что это элемент $(i, j, l_{i,j})$:

• если клетка (i, j) уже просмотрена, то её итоговый уровень уже был ранее определён, поэтому игнорируем клетку (i, j) и возвращаемся к шагу 2 алгоритма;

• если клетка (i, j) не просмотрена, то полагаем $b_{i,j} = l_{i,j}$ (итоговый уровень клетки); считаем клетку (i, j) просмотренной и переходим к шагу 2.2 алгоритма;

2.2 — рассматриваем все соседние с (i, j) непросмотренные клетки: $(i, j-1)$, $(i, j+1)$, $(i-1, j)$, $(i+1, j)$ (если они существуют):

• если текущий уровень соседней клетки, например $b_{i,j-1}$, больше, чем уровень клетки $b_{i,j}$ (вода из клетки $(i, j-1)$ будет выливаться, а текущий уровень воды станет равным максимальному значению из уровня клетки (i, j) и высоты клетки $(i, j-1)$, так как уровень воды не может быть меньше, чем высота самой клетки), то заносим клетку $(i, j-1)$ в приоритетную очередь:

$$(i, j-1, l_{i,j-1} = \max\{b_{i,j}, a_{i,j-1}\})\,;$$

• если текущий уровень соседней клетки меньше либо равен уровню клетки $b_{i,j}$, то ничего с такой клеткой не делаем.

Матрица B задаёт уровни воды клеток формы. Объём невытекшей воды может быть определён по формуле

$$\sum_{i=1}^{n}\sum_{j=1}^{m}(b_{i,j} - a_{i,j}).$$

Из описания алгоритма следует, что если клетка (i, j) удаляется из кучи первый раз, то её итоговый уровень становится определённым. Действительно, после удаления из приоритетной очереди клетка (i, j) может содержаться в приоритетной очереди ещё несколько раз, но с таким же или с большим значением уровня (уровни вновь добавляемых элементов не меньше уровня удалённого из кучи элемента). Поэтому после первого удаления клетки (i, j) из приоритетной очереди её полагают просмотренной, и в дальнейшем она в алгоритме игнорируется. Количество операций добавления (удаления) элементов в приоритетную очередь не превосходит величины $4 \cdot n \cdot m$, так как каждая клетка формы (i, j) может добавляться в кучу не более четырёх раз. Действительно, клетка (i, j) может быть занесена в приоритетную очередь из соседних клеток: $(i, j-1)$, $(i, j+1)$, $(i-1, j)$, $(i+1, j)$ (если они существуют), причём из каждой соседней клетки клетка (i, j) может быть занесена в приоритетную очередь не более одного раза. Число арифметических операций предложенного алгоритма — $\mathrm{O}(nm \log(nm))$.

Для примера в результате выполнения алгоритма получим такие матрицы:

$$A = \begin{pmatrix} 5 & 3 & 1 \\ 5 & 2 & 5 \\ 2 & 5 & 5 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 3 & 1 \\ 5 & 3 & 5 \\ 2 & 5 & 5 \end{pmatrix}.$$

Объём невытекшей воды равен 1.

9. Шланги. Для решения задачи можно использовать структуру данных «стек», в который будем заносить номера шлангов. Последовательно двигаемся вдоль одного из шлангов. Для первой точки пересечения шлангов заносим в стек номер того шланга, который находится сверху. Предположим, что в следующей точке пересечения сверху находится шланг 1 (2). Тогда, если на вершине стека находится 1 (2), то удаляем элемент из стека (распутываем шланги в этом месте), в противном случае добавляем число 1 (2) в стек. Если все точки пересечения просмотрены и стек пустой, то шланги можно распутать, в противном случае — нет. Время работы — $O(n)$.

10. Лабиринт с конкретным входом-выходом. Несложно заметить, что для существования решения необходимо проводить людей от самого левого входа к самому левому выходу, а от самого правого входа — к самому правому выходу.

Для решения задачи можно использовать структуру данных «стек» (алгоритм «поиск в глубину» с приоритетами, где приоритет определяется по правилу правой руки: человек должен двигаться, держась правой рукой за «стенку» лабиринта, в те позиции, где он ещё не был и куда ещё может пойти). При выборе направления движения из текущей позиции (i, j) лабиринта необходимо учитывать ту позицию, из которой вы пришли в позицию (i, j). Предположим, что $(1, j_0)$ — один из входов, тогда можно считать, что в этот вход движение было осуществлено из фиктивной позиции $(0, j_0)$. В общем случае, если в позицию (i, j) вы пришли из позиции лабиринта $(i-1, j)$, сначала нужно попытаться пойти в позицию $(i, j-1)$ (если $(i, j-1)$ существует и не заблокирована), иначе — в позицию $(i+1, j)$ (если $(i+1, j)$ существует и не заблокирована), в противном случае — в позицию $(i, j+1)$ (если $(i, j+1)$ существует и не заблокирована). Аналогично можно определить наилучшее направление для других случаев. При переходе в новую позицию она сразу блокируется для всех людей. Время работы алгоритма — $O(nm)$.

11. Лабиринт и произвольный выход. Решение задачи аналогично задаче 10 «Лабиринт с конкретным входом-выходом». Отличие заключается лишь в том, что при переходе в новую позицию, которая помечена как выход, она не блокируется для всех людей (через выход по условию можно провести несколько людей), а все остальные позиции при их проходе должны быть заблокированы. Время работы алгоритма — $O(nm)$.

12. Чиновники. Построим корневое дерево. Вершины корневого дерева соответствуют чиновникам (корень дерева — самый главный чиновник). Дуга корневого дерева (i, j) означает, что у чиновника j непосредственным начальником является чиновник i. Припишем каждой дуге вес $c(i, j)$, который равен сумме денег, которую потребует чиновник j, если к нему обратиться за подписью. Так как дерево является корневым, то из корня дерева в каждую вершину существует единственный путь. Для решения задачи необходимо найти кратчайший маршрут из корня в лист дерева. Данную задачу можно решить за время $O(n)$, используя структуры данных «стек» или «очередь». Время работы алгоритма — $O(n)$.

13. Караван с движением по меньшей высоте. Для решения задачи можно использовать структуры данных «стек» (алгоритм поиска в глубину) или «очередь» (алгоритм поиска в ширину). Время работы алгоритма — $O(n^2)$.

14. Караван с движением по крутизне не больше *k*. Так как необходимо найти минимальный по количеству переходов из квадрата в квадрат маршрут каравана, то для решения задачи нужно использовать структуру данных «очередь» (алгоритм поиска в ширину). Время работы алгоритма — $O(nm)$.

15. Минимальная длина маршрута робота. Так как в задаче требуется найти кратчайший маршрут, то для решения задачи можно использовать одну из реализаций алгоритма Дейкстры, использующую структуру данных «приоритетная очередь». Время работы алгоритма — $O(nm \log(nm))$.

16. Минимальное число ходов коня. Так как в задаче требуется найти минимальный по количеству переходов маршрут коня между заданными клетками поля, то для решения нужно использовать струк-

туру данных «очередь» (алгоритм поиска в ширину). Время работы алгоритма — $O(nm)$.

17. Минимальное число ходов белого коня со взятием чёрных фигур. Один из подходов к решению задачи заключается в поиске кратчайшего по стоимости маршрута шахматного коня между заданными клетками шахматной доски. В данной интерпретации ход коня на свободную клетку стоит 1 единицу, а ход белым конём на клетку, занятую чёрной фигурой или пешкой, стоит 2 единицы. Для решения данной задачи можно использовать реализацию алгоритма Дейкстры с использованием структуры данных «приоритетная очередь», при этом приоритетную очередь можно реализовать на двух очередях (при ходе коня на свободное поле добавление клетки с соответствующим приоритетом идёт в первую очередь, а при ходе коня на клетку, занятую чёрной фигуры или пешкой, — во вторую очередь). Легко доказать, что приоритеты элементов каждой из очередей упорядочены, что позволяет на каждой итерации алгоритма Дейкстры элемент с наибольшим приоритетом выбирать за время $O(1)$, а новые элементы добавлять в соответствующую очередь (один элемент добавляется в очередь за время $O(1)$). Время работы алгоритма — $O(nm)$.

18. Минимальное число белых коней. Для решения данной задачи можно использовать структуры данных «стек» или «очередь». Эти структуры данных используются для того, чтобы для текущей свободной позиции, куда ставится конь, можно было пометить все позиции шахматной доски, которые пробиваются этим конём за некоторое число ходов. Время работы алгоритма — $O(nm)$.

19. Минимальное число белых ладей. Для решения данной задачи можно использовать структуры данных «стек» или «очередь». Эти структуры данных используются для того, чтобы для текущей свободной позиции, куда ставится ладья, можно было пометить все позиции шахматной доски, которые пробиваются этой ладьёй за некоторое число ходов. При этом для текущей позиции (i, j), где стоит ладья, за один ход обследуются позиции $(i+1, j)$, $(i-1, j)$, $(i, j+1)$ и $(i, j-1)$, если они существуют, не заняты фигурами и не были ранее посещены. Согласно правилам шахмат, ладья может перемещаться на несколько клеток по горизонтали или вертикали, однако такой ход можно заменить несколькими маленькими ходами (из клетки в соседнюю клетку). Время работы алгоритма — $O(nm)$.

20. Минимальное число белых слонов. Для решения данной задачи можно использовать структуры данных «стек» или «очередь». Эти структуры данных используются для того, чтобы для текущей свободной позиции, куда ставится слон, можно было пометить все позиции шахматной доски, которые пробиваются этим слоном за некоторое число ходов. При этом для текущей позиции (i, j), где стоит слон, за один ход обследуются позиции $(i+1, j+1)$, $(i-1, j-1)$, $(i-1, j+1)$ и $(i+1, j-1)$, если они существуют, не заняты фигурами и не были ранее посещены. Согласно правилам шахмат, слон может перемещаться на несколько клеток по диагонали, однако такой ход можно заменить несколькими маленькими ходами (из клетки в соседнюю клетку). Время работы алгоритма — $O(nm)$.

21. Цилиндр. Для выделения связных областей можно применить структуры «стек» или «очередь». При этом следует учитывать, что цилиндр склеили таким образом, что первая строка матрицы, задающей цилиндр, связана не только со второй строкой, но и с m-й. Время работы алгоритма — $O(nm)$.

22. Прямоугольники. Разделим прямоугольник на клетки размера 1×1. Изначально необходимо для каждой такой клетки посчитать, в какой цвет она будет покрашена после всех перекрашиваний. Для каждой строки прямоугольника необходимо построить дерево отрезков, поддерживающее операции присвоения на интервале. Перекрашивание подпрямоугольника начального прямоугольника можно разбить на перекрашивание некоторых полосок размера $1 \times w$. Для каждой такой полоски будет применена одна операция присвоения на интервале. Для выделения одноцветной фигуры можно использовать стек или очередь. При этом следует учитывать, что для некоторой клетки (i, j) смежные с ней клетки того же цвета, что и (i, j) (смежность проверяется по восьми направлениям) не всегда принадлежат одной одноцветной фигуре. Так, смежные клетки (i, j) и $(i-1, j)$ принадлежат одной одноцветной фигуре, если их цвет совпадает, но в то же время для смежных клеток (i, j) и $(i-1, j+1)$ только лишь совпадения цвета не всегда достаточно для того, чтобы они принадлежали одной одноцветной фигуре. Сложность решения — $O(nh \log w + hw)$.

23. За́мок. Для того чтобы найти число комнат в замке и площадь наибольшей комнаты, можно каждой клетке поставить в соответствие

номер комнаты, которой она принадлежит. Для этого можно использовать такие структуры данных, как «стек» или «очередь».

После того как решены первые две подзадачи, несложно определить наибольшую возможную площадь комнаты, полученную после удаления одной из стенок. Для решения этой подзадачи достаточно просмотреть всю матрицу, задающую замок, по строкам и, «разрушая» стенки, расположенные снизу и справа от текущей клетки (i, j), перевычислять площадь новой максимальной по площади комнаты (перевычисление происходит в том случае, если смежные клетки принадлежат разным комнатам). Время работы алгоритма — $O(nm)$.

24. Компании. Для решения задачи можно использовать структуру данных «очередь». Последовательно просматриваем все компании. Предположим, что A — текущая компания. Просмотрим все акции этой компании и добавим в очередь номера тех компаний, которыми она владеет (более чем 50 % акций). Пока очередь не станет пустой, удаляем компанию из очереди, суммируем акции этой компании с акциями компании A (акции тех компаний, про которые уже известно, что ими владеет компания A, игнорируем), если при суммировании процент акций становится более 50 %, то сразу добавляем эту компанию в очередь. Время работы алгоритма — $O(n^3)$.

25. Правильная скобочная последовательность. Для решения задачи нужно использовать структуру данных «стек». Время работы алгоритма — $O(|S|)$, где S — исходная строка.

26. Чёрный ящик. Для решения задачи можно использовать структуру данных «бинарная куча». В процессе поступления запросов будем формировать две кучи: min-heap и max-heap. Все поступающие по запросам Добавить элементы будем распределять по двум кучам таким образом, чтобы в max-heap оказывались первые p наименьших элементов из всех, поступивших на данный момент, где p — количество запросов Получить, поступивших на данный момент. Тогда каждый из запросов Добавить или Получить потребует выполнить несколько операций удаления и добавления элемента из кучи.

Второй структурой данных, которую можно использовать для решения задачи, является сбалансированное поисковое дерево, в котором для каждой вершины v нужно ввести дополнительную метку — число вершин в поддереве с корнем в вершине v. Время работы алгоритма — $O(m \log m)$, где m — число запросов Добавить.

27. Межшкольная сеть. По входным данным задачи построим ориентированный граф G. Для решения первой подзадачи сначала найдём все сильносвязные компоненты ориентированного графа и стянем все вершины каждой компоненты сильной связности в одну вершину, получая при этом ациклический ориентированный граф G'. Для выделения сильносвязных компонент можно использовать, например, алгоритм Косарайю (Kosaraju) и Шарира (Sharir), основанный на двукратном запуске поиска в глубину [3]. Теперь решение первой подзадачи — число вершин G' без входящих дуг.

Для ориентированного графа G' обозначим через A множество вершин без входящих дуг, а через B — множество вершин без выходящих дуг (каждую изолированную вершину v предварительно дробим на две вершины v' и v'', которые соединяем дугой (v', v'')). Теперь для решения второй подзадачи, используя поиск в глубину, каждой вершине a из множества A попробуем поставить в соответствие такую вершину b из множества B, которая достижима из a (после поиска очередной пары (a_i, b_i), пометки посещения вершин не снимаем). Предположим, что получено следующее разбиение на пары: (a_i, b_i), $i = 1, \ldots, k$. Соединяя последовательно дугами вершины b_i и a_{i+1}, $i = 1, \ldots, k-1$ и, добавляя дугу (b_k, a_1), получим контур. У нас могут остаться непарными некоторые вершины множества A, из которых обязательно идёт путь в контур, а также вершины множества B, в которые обязательно идёт путь из контура. Остаётся добавить дуги между этими непарными вершинами, чтобы получить одну сильносвязную компоненту. Время работы алгоритма — $O(n^3)$.

28. Расписание. Первый подход к решению задачи. Сначала нужно доказать следующее утверждение: для независимого множества работ последней должна быть выполнена та из них, директивный срок которой наибольший. На начальном этапе в независимое множество работ входят работы без выходящих дуг. Используя структуру данных «приоритетная очередь» (max-heap), среди независимого множества работ находим работу с максимальным директивным сроком, удаляем её и дополняем множество независимых работ.

Второй подход к решению задачи. Заменим дуги исходного орграфа на противоположно направленные (т. е. инвертируем орграф). Используя структуру данных «приоритетная очередь» (min-heap), упорядочиваем всё множество работ в порядке неубывания директивного срока. Выбираем работу с минимальным директивным сроком и находим все

достижимые из неё вершины в инвертированном графе. Все найденные вершины необходимо добавить в итоговую последовательность выполнения работ в топологическом порядке.

Время работы алгоритма — $O(n \log n)$.

29. Пересечение реки. Очевидно, что острова возвращаются в начальное состояние в моменты времени, кратные

$$T = \text{НОК}(a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n).$$

Так как

$$2 \leqslant a_i + b_i \leqslant 10,$$

то

$$T \leqslant \text{НОК}(2, 3, \ldots, 10) = 2^3 \cdot 3^2 \cdot 5 \cdot 7 = 2520.$$

Таким образом, различных взаимных расположений всех островов может быть не более T.

Назовём состоянием пару (i, t), где t — момент времени (по модулю T), в который житель находится на острове i. Тогда мы можем перейти в состояния $(i-5, t'), \ldots, (i+5, t')$, где $t' = (t+1) \bmod T$, при условии, что соответствующие острова доступны в следующий момент времени.

Состояние однозначным образом характеризует взаимное расположение островов и их расположение в дальнейшем, поэтому для решения задачи имеет значение лишь минимальное время, за которое мы сможем попасть в каждое из состояний. Но тогда время, за которое мы сможем попасть в любое из состояний, не превышает $T \cdot n$.

Для нахождения минимального времени будем использовать поиск в ширину по всем возможным состояниям. Ясно, что если мы пришли в состояние впервые, то соответствующее время будет минимальным. Для каждого состояния будем проверять все состояния, в которые мы можем перейти (в случае, если остров поднят) и добавлять в очередь в случае, если они не были добавлены ранее. Таким образом, мы рассмотрим каждое из состояний не более одного раза, поэтому сложность алгоритма — $O(T \cdot n)$

30. Flood it! Нетрудно заметить, что если некоторая клетка после проведённых перекрашиваний находится в одной области с левой верхней клеткой матрицы, то это условие не нарушится никогда. Любое перекрашивание может только увеличить размер области, в которой

расположена верхняя левая клетка матрицы. Из этого следует вывод, что все клетки, для которых все их соседи по стороне находятся в одной области, не имеют никакого принципиального значения и цвет в них может быть определён через соседей. Поэтому для решения задачи достаточно поддерживать только контур всей области, достижимой из левой верхней клетки.

Во время выполнения операций необходимо поддерживать следующую информацию. Для каждого из k цветов необходимо поддерживать все смежные с контуром области клетки этого цвета. Тогда при очередном перекрашивании сразу известно, какие клетки будут добавлены в область, достижимую из левой верхней. Одновременно с этим будут пересчитаны все соседи у уже нового образованного контура.

Время работы решения — $O(nm + k + t)$.

31. Химическая реакция. Процесс добавления веществ в пробирку можно смоделировать на структуре данных «стек». При этом нужно доказать, что суммарное число операций работы со стеком (PUSH, POP) есть $O(m)$, где m — количество добавляемых веществ. Время работы алгоритма — $O(n^2 + m)$, где слагаемое n^2 связано с вводом данных.

32. Равенства и неравенства. Для решения задачи можно использовать структуру данных «система непересекающихся множеств» (она задаётся в виде семейства корневых деревьев; при объединении двух множеств используют стратегию «меньшее к большему»). Сначала обрабатываются переменные, которые связаны оператором (`==`): все переменные, которые равны между собой, должны быть отнесены к одному и тому же множеству. Затем обрабатываются переменные, которые связаны оператором `!=`; если эти переменные принадлежат одному и тому же множеству, то ответом решения задачи является `No`. Время работы алгоритма — $O(m \log n + n)$.

33. Парк. Предположим, что зафиксированы нижняя и верхняя границы детской площадки $y_{\min}$ и $y_{\max}$. Если выписать различные x-координаты всех деревьев, лежащих строго внутри рассматриваемой области, и упорядочить их по возрастанию ($x_1 < x_2 < \ldots < x_k$), то наибольшая площадь детской площадки в данной области определяется формулой

$$(y_{\max} - y_{\min}) \times \max\{x_1 - 0, x_2 - x_1, \ldots, x_k - x_{k-1}, w - x_k\}.$$

Допустим, нижняя граница неподвижна, а верхняя граница смещается вниз. При этом некоторые деревья оказываются за пределами рассматриваемой области. Научимся эффективно обновлять указанный максимум. Будем хранить все значения x_i в двусвязном списке. Также для каждого x_i будем хранить число деревьев, имеющих такую x-координату. Тогда в момент исчезновения последнего дерева с координатой x_i можно за $O(1)$ определить соседние элементы x_{i-1} и x_{x+1}, при необходимости скорректировать значение максимума и удалить элемент x_i из списка. Однако для выполнения этой операции необходимо поддерживать число деревьев, имеющих одну x-координату, и быстро получать доступ к элементам внутри списка. Для этого можно создать массив, индексами которого являются x-координаты, а значениями — указатели на элементы связного списка.

Однако при использовании описанного подхода время работы алгоритма и объём используемой памяти зависят от абсолютных величин координат. Чтобы этого избежать, можно изначально выполнить *сжатие координат* по оси абсцисс. Выпишем все x-координаты, упорядочим их, уберём повторяющиеся значения, затем каждую x-координату заменим её порядковым номером в полученной последовательности. Например, если изначально деревья имеют координаты $(303, 1)$, $(202, 5)$, $(101, 3)$, $(303, 6)$, то набор уникальных x-координат имеет вид $(101, 202, 303)$, после сжатия получим $(3, 1)$, $(2, 5)$, $(1, 3)$, $(3, 6)$.

Итак, для решения задачи упорядочим деревья по убыванию y-координат. Будем передвигать нижнюю границу сверху вниз, перемещаясь по уникальным значениям y-координат деревьев (отдельно рассматривается случай, когда нижняя граница совпадает с осью абсцисс). Для каждой позиции нижней границы области устанавливаем верхнюю границу области в соответствии с верхней границей парка и двигаем её вниз. Общее время работы решения — $O(n^2)$.

34. Урок математики. Допустим, что нам известны все n чисел, которые надо найти. Упорядочим данные числа по неубыванию:

$$a_1 \leqslant a_2 \leqslant a_3 \leqslant \ldots \leqslant a_n.$$

Предположим, что по данным числам построили матрицу S размера $n \times n$ (всех возможных парных сумм), где

$$s_{i,j} = a_i + a_j.$$

Например, для $n = 4$ матрица будет такой:

$a_1 + a_1$	$a_1 + a_2$	$a_1 + a_3$	$a_1 + a_4$
$a_2 + a_1$	$a_2 + a_2$	$a_2 + a_3$	$a_2 + a_4$
$a_3 + a_1$	$a_3 + a_2$	$a_3 + a_3$	$a_3 + a_4$
$a_4 + a_1$	$a_4 + a_2$	$a_4 + a_3$	$a_4 + a_4$

Матрица S является симметричной относительно главной диагонали, а элементы каждой строки (столбца) упорядочены по неубыванию. Заметим, что элементы матрицы S, записанные в произвольном порядке, и являются входными данными задачи.

Очевидно, что в матрице наименьший элемент стоит в верхнем левом углу. Таким образом, взяв из входных данных самое маленькое число, мы узнаем $s_{1,1}$. Затем можно однозначно определить a_1 по формуле $a_1 = s_{1,1}/2$. Вычёркиваем число $s_{1,1}$ из входных данных. Из оставшихся элементов самое маленькое число будет соответствовать элементу матрицы $s_{1,2} = s_{2,1} = a_1 + a_2$, поэтому находим следующее число a_2 по формуле $a_2 = s_{1,2} - a_1$, вычёркиваем из входа элементы $s_{1,2}$, $s_{2,1}$ и $s_{2,2}$, которые являются парными суммами элементов a_1 и a_2. Повторяя описанные действия, однозначно определим все числа $a_1, a_2, \ldots, a_n$.

Реализовать описанные действия можно, используя структуру данных «ассоциативный массив» (map), который работает по принципу «ключ — значение». В качестве ключей выступают парные суммы, а в качестве значений — соответствующих данным суммам количества их повторений. Добавление, удаление и обращение к элементам map происходит за $O(\log m)$, где m — число элементов в ассоциативном контейнере. Время работы алгоритма — $O(n^2 \log n)$.

35. Космическая экспедиция. Для решения задачи можно использовать систему непересекающихся множеств. Изначально каждый отсек соответствует одноэлементному множеству. Каждый раз, когда робот переходит из отсека в соседний, нужно проверять, принадлежат ли эти отсеки одному множеству, и при необходимости объединять множества. Время работы решения — $O(nmk \log nmk + t \cdot \max\{n, m, k\})$.

36. Звёздочки. В условии задачи указано, что точки отсортированы в порядке возрастания y-координаты, а при равенстве — в порядке возрастания x-координаты, поэтому для любой пары индексов $i < j$ выполняется $y_i \leqslant y_j$. Тогда можно перефразировать условие задачи

следующим образом: для очередной звезды необходимо посчитать количество звёзд, расположенных до неё и имеющих x-координату, не превышающую текущую.

Для решения задачи можно построить дерево отрезков по x-координатам. Будем рассматривать все звёзды по очереди, начиная с первой. Для каждого листа дерева отрезков будем хранить количество уже рассмотренных звёзд с фиксированной x-координатой. Тогда задачу можно решать с помощью простых операций суммы на интервале и изменения одиночного значения в дереве отрезков.

Время работы решения — $O(n \log n)$.

37. Дороги. Будем рассматривать дороги в порядке, обратном порядку разрушения. Изначально между городами нет дорог, дороги строятся по одной, нужно определить момент, когда сеть станет связной. Для решения такой задачи можно использовать систему непересекающихся множеств. Итоговое время работы (при построении СНМ с использованием эвристик сжатия пути и объединения по рангу) — $O(n + m\alpha(n))$, где $\alpha(n)$ — медленно растущая обратная функция Аккермана.

38. Корпоративная сеть. Предприятия, входящие в кластеры, и телекоммуникационные линии между ними можно задать корневыми деревьями. Каждой дуге (i, j) корневого дерева ставится в соответствие длина телекоммуникационной линии между предприятиями i и j. Корневое дерево удобно представить в памяти компьютера в каноническом виде. В этом случае команду `I` можно выполнить за время $O(1)$. При выполнении команды `E` нужно не только осуществлять подъём по корневому дереву от предприятия i к её обслуживающему центру, но и сохранять все пройденные при этом вершины (предприятия), чтобы в последующем выполнить сжатие пути (всем этим предприятиям в корневом дереве назначить в качестве предка их обслуживающий центр и скорректировать веса дуг). Данные действия аналогичны процедуре сжатия пути при работе со структурой данных «система непересекающихся множеств» (DSU). Время работы алгоритма — $O(n \log n)$ [3].

39. Мобильники. Проанализируем такую же задачу, только для одномерного случая. Нетрудно заметить, что тогда задачу можно решать с помощью дерева отрезков, которое умеет выполнять простые операции суммы и прибавления на интервале.

Для двумерного случая решение принципиально не меняется. Если рассмотреть принцип работы дерева отрезков, то при его построении или выполнении определённых операций рассматриваемый интервал разделяется на две равные части и задача решается рекурсивно. Далее ответ для частей объединяется и получается результат для рассматриваемого интервала. Для одномерного случая интервал разбивается на две равные части, для двумерного будем делать аналогично — разбивать матрицу на подматрицы с одинаковым количеством строк или разделять по столбцам, если количество строк равно 1.

Также стоит отметить, что необходимо реализовать «ленивое» дерево отрезков, в котором каждая вершина, отвечающая за какую-либо подматрицу изначальной матрицы, не разбивается на части без необходимости. Разбиение будет производиться только в том случае, если для обработки запроса недостаточно информации, содержащейся в текущей вершине.

Время работы решения — $O(t \log^2 s)$, где t — количество команд на входе.

40. Ленивый программист. Заказы имеет смысл выполнять только в порядке неубывания d_i, а заказы с равными значениями d_i — в произвольном порядке. Далее будем считать, что

$$d_1 \leqslant d_2 \leqslant \ldots \leqslant d_n.$$

Предположим, что нам известен оптимальный способ выполнения первых i заказов, при этом последний заказ заканчивает выполняться в момент времени $t_i \leqslant d_i$. Рассмотрим, как построить оптимальный способ выполнения первых $i+1$ заказов. Вначале положим $x_{i+1} = 0$. При этом получим некоторый способ выполнения первых $i+1$ заказов, со временем выполнения последнего заказа $t_{i+1} = t_i + b_{i+1}$.

Если $t_{i+1} \leqslant d_{i+1}$, то построен оптимальный способ для первых $i+1$ заказов. Иначе необходимо уменьшить время выполнения уже рассмотренных заказов (путём увеличения одного из x_k, $1 \leqslant k \leqslant i$). Очевидно, что в первую очередь нужно увеличивать x_k для заказов с максимальным значением a_k, что позволит уменьшить время выполнения, доплатив меньше денег.

Таким образом, пока $t_{i+1} \geqslant d_{i+1}$, будем искать заказ с максимальным значением a_k среди заказов, у которых $x_k \leqslant b_k/a_k$ (данное требование продиктовано тем, что заказ не может быть выполнен за отрицательное время).

Предположим, что это заказ с номером l. Тогда для погашения всей нужной разницы $(t_{i+1} - d_{i+1})$ потребуется, чтобы

$$b_l - a_l(x_l + \varepsilon) = (b_l - a_l \cdot x_l) - (t_{i+1} - d_{i+1}),$$

откуда следует, что

$$a_l \cdot \varepsilon = t_{i+1} - d_{i+1}.$$

Поэтому

$$\varepsilon = \frac{t_{i+1} - d_{i+1}}{a_l}.$$

С другой стороны, учитывая, что выполнять заказы за отрицательное время нельзя, получаем, что максимально допустимое значение ε для заказа l равно

$$\varepsilon = \frac{b_l}{a_l} - x_l.$$

Таким образом,

$$x_{l+1} = x_l + \min\left\{\frac{b_l}{a_l} - x_l, \frac{t_{i+1} - d_{i+1}}{a_l}\right\}.$$

Для эффективного выполнения описанных действий можно использовать структуру данных «приоритетная очередь» (max-heap). Время работы алгоритма — $O(n \log n)$.

41. Тройное доминирование. Сначала упорядочим команды по возрастанию места, которое в личной олимпиаде заняли первые игроки каждой из них. Очевидно, что над командой A в отсортированном списке команд теперь могут доминировать только те, которые в этом списке встречаются раньше, и именно среди них и будем искать такую команду.

В процессе работы алгоритма будем последовательно формировать массив P (от англ. *place*), индексы которого изменяются от 1 до n (первоначально данный массив заполнен нулями). В массиве P элемент p_i равен тому месту, которое занял в личных соревнованиях третий игрок команды, в которой второй игрок занял место i. Например, если участники некоторой команды заняли места (x, y, z), при этом $y = 3$ и $z = 8$, то $p_3 = 8$.

Итак, предположим, что, последовательно просматривая отсортированный список команд, мы дошли до команды A, участники которой

заняли места (x, y, z). Для того чтобы определить, есть ли команды, которые доминируют над командой A, необходимо в массиве P на отрезке от 1 до y найти минимум, и если этот минимум больше, чем z, то нет команд, которые могут доминировать над командой A. После того как рассмотрена команда A, полагаем $p_y = z$ и переходим к рассмотрению следующей команды из отсортированного списка команд.

Для выполнения описанных действий можно использовать структуру данных «дерево отрезков». Эта структура данных для последовательности из n элементов позволяет за время $O(\log n)$ выполнять следующие две операции: находить элемент с минимальным ключом на интервале от 1 до r и выполнять модификацию ключа элемента по его индексу. Время работы алгоритма — $O(n \log n)$.

42. Соревнования. Подход к решению задачи аналогичен тому, который использовался в предыдущей задаче. Упорядочим участников по возрастанию места, которое они заняли в первом соревновании. Очевидно, что лучше, чем участник A, в отсортированном списке участников могут быть лишь те, кто в этом списке встречаются раньше него.

В процессе работы алгоритма будем последовательно формировать массив P (от англ. *place*), индексы которого изменяются от 1 до n (первоначально данный массив заполнен нулями). Если некоторый участник во втором соревновании занял 4-е место, а в третьем соревновании — 7-е, то $p_4 = 7$. Данный массив будет сформирован корректно, так как по условию задачи никакие два участника не имели одинаковые результаты в одном и том же соревновании.

Итак, предположим, что, последовательно просматривая отсортированный список, мы дошли до участника A, который занял в трёх соревнованиях соответственно места (x, y, z). Для того чтобы определить, есть ли участники, которые лучше него, необходимо в массиве P на интервале от 1 до y найти минимум. Если этот минимум больше, чем z, то нет никого, кто лучше, чем A (т. е. A — лучший). После того как рассмотрен участник A, полагаем $p_y = z$ и переходим к рассмотрению следующего участника из отсортированного списка.

Для выполнения описанных действий можно использовать структуру данных «дерево отрезков», которая для последовательности из n элементов позволяет за время $O(\log n)$ выполнять следующие две операции: находить элемент с минимальным ключом на интервале от 1 до r и выполнять модификацию ключа элемента по его индексу. Время работы алгоритма — $O(n \log n)$.

43. Кодирование Хаффмана. Анализируя то, как подготовлены данные на входе и как вычисляются метки внутренних вершин дерева кода Хаффмана, можно сделать вывод, что для решения задачи не обязательно непосредственно строить дерево кода Хаффмана и определять глубину каждого листа этого дерева. Для вычисления длины закодированного текста достаточно, последовательно считывая входные данные, использовать структуру данных «приоритетная очередь», интерфейс которой для нашей задачи можно реализовать на двух очередях FIFO. Время работы алгоритма — $O(n)$.

44. Слияние возрастающих последовательностей. Для решения задачи можно использовать структуру данных «приоритетная очередь», реализованную на базе бинарной кучи, в которой будем поддерживать не более чем n элементов. Первоначально в кучу (min-heap) добавим первые элементы каждой из n упорядоченных по возрастанию последовательностей. Пока куча не станет пустой, удаляем из неё элемент и добавляем вместо него элемент из той последовательности, которой он принадлежал, следующий непосредственно за ним.

Удаление и добавление одного элемента в кучу выполняется за время $O(\log n)$, число таких операций равно общему числу всех элементов $O(nm)$. Итого, время работы алгоритма — $O(nm \log n)$.

45. Мегаинверсии. Для решения задачи последовательно просматриваем элементы перестановки слева направо. Предположим, что j — индекс текущего элемента в перестановке. Если мы научимся эффективно определять, сколько уже встретилось элементов, которые больше, чем p_j, то, учитывая тот факт, что все элементы перестановки — попарно различные целые числа от 1 до n, сразу сможем вычислить число мегаинверсий, для которых $i < j < k$ и $p_i > p_j > p_k$. Для решения этой подзадачи можно использовать структуру данных «дерево отрезков». Время работы алгоритма — $O(n \log n)$.

46. Отрицательные скидки. Для решения задачи используется приоритетная очередь, интерфейс которой реализован на основе бинарной кучи (min-heap). В куче храним пару (стоимость товара, номер магазина). Наибольший приоритет имеет товар с наименьшей стоимостью. Время работы алгоритма — $O(k \log n)$.

47. Хорды. Пусть для определённости $a_i < b_i$ для любого i, хорды имеют направление от a_i к b_i. Создадим массив A размера $2n$, заполненный нулями. Будем рассматривать точки в порядке 1, 2, ..., $2n$.

Если точка p является началом i-й хорды (т. е. $p = a_i$), то увеличим на единицу значение $A[a_i]$ (хорда «открыта»). Если точка p является концом i-й хорды (т. е. $p = b_i$), то уменьшим значение $A[a_i]$ на единицу (хорда «закрыта») и прибавим к ответу сумму $A[a_i]$, $A[a_i] + 1$, ..., $A[b_i]$ (это будет число хорд, пересекающих i-ю хорду, начало которых лежит между a_i и b_i).

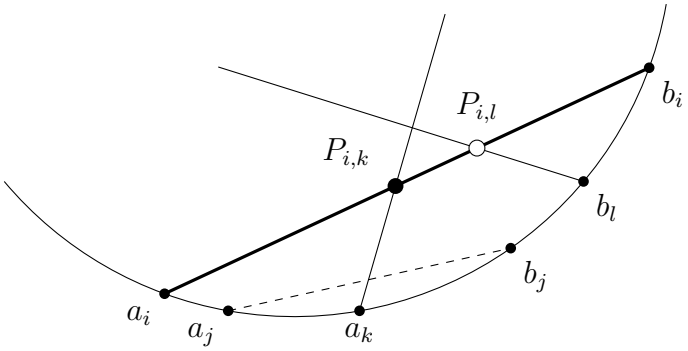


Рис. 7.15. Пример расположения хорд

Можно показать, что каждая точка пересечения будет учтена ровно один раз. Так, на рис. 7.15 анализируется хорда с началом a_i и концом b_i. В момент рассмотрения точки b_i хорда с началом a_k находится в «открытом» состоянии, поэтому пересечение с этой хордой (точка $P_{i,k}$) будет посчитано и к ответу прибавится единица. В то же время хорда с концом b_l уже находится в «закрытом» состоянии, поэтому точка $P_{i,l}$ не будет посчитана (она уже внесла свой вклад в ответ, когда рассматривался конец хорды b_l).

Для быстрого вычисления сумм на отрезке нужно использовать соответствующую структуру данных. Время работы алгоритма — $O(n \log n)$.

48. Архив. Если расположить все документы в стопке и представить это как некоторую полоску, то переставление любого подотрезка документов — это не что иное, как разрезание полоски в двух местах и её склейка в нужном порядке (сначала вырезанная часть, затем левая и правая от образованных разрезов соответственно). Подобные операции позволяет делать декартово дерево по неявному ключу. Время работы решения задачи — $O((q + n) \log n)$.

49. Продажа билетов. Для решения задачи можно использовать структуру данных «разрежённая таблица». За время $O(n \log n)$ подсчи-

таем минимумы на всех отрезках, длины которых являются степенями двойки. Для запроса $[l, r]$ найдём такое максимальное x, что 2^x не превышает $r - l + 1$ (длину отрезка). Тогда отрезок $[l, r]$ можно покрыть двумя отрезками длиной 2^x (возможно, накладывающимися друг на друга) и вычислить минимальное значение на отрезке за время $O(1)$. Общее время работы — $O(n \log n + q)$.

50. Скользящая медиана. Выборку размера n будем хранить в виде двух половин, используя две кучи: первая куча устроена по принципу max-heap и содержит $\lceil n/2 \rceil$ чисел, вторая куча организована как min-heap и содержит $\lfloor n/2 \rfloor$ чисел, при этом максимальное значение в первой куче не превосходит минимального значения во второй куче.

Нетрудно заметить, что при таком способе хранения выборки медиана либо лежит в корне первой кучи, либо равна среднему арифметическому корней обеих куч (в зависимости от чётности n). При добавлении в выборку нового числа оно размещается в той или иной куче в зависимости от его величины, затем при необходимости производится балансировка размеров куч: максимум из первой кучи перебрасывается во вторую или минимум из второй кучи направляется в первую.

Общее время работы решения — $O(n \log n + q)$, где n — число запросов добавления, q — число запросов расчёта медианы.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — М. : Вильямс, 2005. — 1296 с.

2. *Котов В. М.*, *Мельников О. И.* Информатика. Методы алгоритмизации : учеб. пособие для 10 – 11 кл. общеобразоват. шк. с углубл. изучением информатики. — Минск : Нар. асвета, 2000. — 221 с.

3. *Котов В. М.*, *Соболевская Е. П.*, *Толстиков А. А.* Алгоритмы и структуры данных : учеб. пособие. — Минск : БГУ, 2011. — 267 с. — (Классическое университетское издание).

4. Сборник задач по теории алгоритмов : учеб.-метод. пособие/ В. М. Котов [и др.]. — Минск : БГУ, 2017. — 183 с.

5. Теория алгоритмов : учеб. пособие / П. А. Иржавский [и др.]. — Минск : БГУ, 2013. — 159 с.

6. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Опыт использования образовательной платформы Insight Runner на факультете прикладной математики и информатики Белорусского государственного университета // Роль университетского образования и науки в современном обществе : материалы междунар. науч. конф., Минск, 26–27 февр. 2019 г. / Белорус. гос. ун-т ; редкол.: А. Д. Король (пред.) [и др.]. — Минск : БГУ, 2019. — С. 263-267.

7. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Методика преподавания дисциплин по теории алгоритмов с использованием образовательной платформы iRunner [Электронный ресурс] // Судьбы классического университета: национальный контекст и мировые тренды : материалы XIII Респ. междисциплинар. науч.-теорет. семинара «Инновационные стратегии в современной социальной философии» и междисциплинар. летней школы молодых ученых «Экология культуры», Минск, 9 апр. 2019 г. / Белорус. гос. ун-т ; сост.: В. В. Анохина, В. С. Сайганова ; редкол.: А. И. Зеленков (отв. ред.) [и др.]. — Минск : БГУ, 2019. — С. 346–355.

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ 3

Часть 1. ПРОСТЕЙШИЕ СТРУКТУРЫ ДАННЫХ

1.1. Массив 5
1.2. Динамический массив 7
1.3. Связный список 12

Часть 2. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

2.1. Список 17
2.2. Стек 18
2.3. Очередь 19
2.4. Двухсторонняя очередь 21
2.5. Приоритетная очередь 21
2.6. Множество 25
2.7. Ассоциативный массив 25

Часть 3. ХЕШИРОВАНИЕ

3.1. Устройство хеш-таблицы 27
3.2. Разрешение коллизий методом цепочек 30
3.3. Разрешение коллизий методом открытой адресации 32
3.4. Коэффициент заполнения 38
3.5. Оценки времени выполнения операций 39
3.6. Идеальное хеширование 41
3.7. Универсальное хеширование 43
3.8. Хеш-таблицы на практике 47
3.9. Выводы 55

Часть 4. СТРУКТУРЫ ДАННЫХ ДЛЯ РЕШЕНИЯ ЗАДАЧ НА ИНТЕРВАЛАХ

4.1. Постановка задачи о сумме 56
4.2. Наивный подход. Подсчёт префиксных сумм 58
4.3. Sqrt-декомпозиция 60
4.4. Дерево отрезков 62
4.5. Задача RMQ. Sparse table 69

Часть 5. СПЕЦИАЛЬНЫЕ СТРУКТУРЫ ДАННЫХ

5.1. Бинарная куча 72
5.2. Декартово дерево 76
5.3. Система непересекающихся множеств 80

Часть 6. БИНАРНЫЙ ПОИСК

6.1. Описание метода 88
6.2. Бинарный поиск самого левого элемента 89

Часть 7. ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

7.1. Условия 91
7.2. Указания к решению задач 136

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ 157

Учебное издание

Соболь Сергей Александрович
Вильчевский Константин Юрьевич
Котов Владимир Михайлович и др.

СБОРНИК ЗАДАЧ ПО ТЕОРИИ АЛГОРИТМОВ. СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Ответственный за выпуск *Т. М. Турчиняк*
Художник обложки *Т. Ю. Таран*
Технический редактор *Л. В. Жабаровская*
Компьютерная вёрстка *С. А. Соболя*
Корректор *Е. В. Бобрович*

Подписано в печать 28.02.2020. Формат 60×84/16. Бумага офсетная.
Печать офсетная. Усл. печ. л. 10,69. Уч.-изд. л. 9,06.
Тираж 80 экз. Заказ 2715.

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/270 от 03.04.2014.
Пр. Независимости, 4, 220030, Минск.