МАКСИМ ИВАНОВ

АЛГОРИТМИЧЕСКИЙ ТРЕНИНГ.

Решения практических задач на PYTHON и C++

bhv®

МАКСИМ ИВАНОВ

АЛГОРИТМИЧЕСКИЙ ТРЕНИНГ.
Решения практических задач на PYTHON и C++

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.43
ББК 32.973.26-018.2
И20

Иванов М. К.

И20 Алгоритмический тренинг. Решения практических задач на Python и C++. — СПб.: БХВ-Петербург, 2023. — 416 с.: ил.

ISBN 978-5-9775-1168-1

Опираясь на богатый соревновательный и эвристический опыт, автор предлагает оригинальные реализации классических алгоритмов Computer Science на языках Python и C++. Особое внимание уделено математическим и геометрическим алгоритмам, графовым алгоритмам, структурам данных (в особенности различным деревьям), комбинаторике и работе со строками. Книга поможет заложить и расширить алгоритмическую подготовку, познакомит с эффективными решениями вычислительных задач, а для обучающихся станет настольной. Поможет подготовиться к экзаменам, сертификации, олимпиадам по программированию.

УДК 004.43
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Анна Брезман*
Оформление обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-1168-1

Содержание

Введение

Глава 1.
Для кого эта книга

Эта книга задумана как вводный курс в спортивное программирование.

Мы приглашаем всех читателей, желающих научиться решать алгоритмические задачи по программированию или уже имеющих начальный опыт участия в олимпиадах, проследовать за нашим путеводителем. Книга будет интересна как тем, кто желает попробовать себя в спортивном программировании, так и тем, кто хочет дополнительно подготовиться к собеседованиям на должность разработчика.

Бóльшая часть материала книги предполагает знание уровня школьной программы по математике. Мы ожидаем, что читатель имеет базовые навыки написания программ на языке Python или языке C++.

Глава 2.
Чему обучит эта книга

Книга начинается с рассмотрения более или менее простых задач, не требующих специальной подготовки. Затем мы переходим к основополагающим алгоритмам и приемам, таким, как двоичный поиск и динамическое программирование. Ближе к концу книги мы знакомимся с некоторыми графовыми и геометрическими алгоритмами, а также важными структурами данных.

Книга организована в порядке нарастания уровня сложности. Каждая тема сопровождается примерами решений задач, поскольку главной целью является научиться применять изученные алгоритмы на практике. Параллельно с изложением алгоритмов излагается теоретическая база на минимально необходимом уровне.

Чего нет в данной книге

Читатель не найдет описаний алгоритмов сортировки или сбалансированных деревьев — это чрезвычайно объемные темы, в то время как в современных реалиях эти алгоритмы практически никогда не требуется реализовывать самостоятельно, равно как и не требуется детальное знание их устройства.

Кроме того, не следует искать в нашей книге математически строгих доказательств используемых алгоритмов. Наша цель — не опубликовать снова давно известные безупречные (и сухие) доказательства, а помочь читателю разобраться на интуитивном уровне в предпосылках, из которых выводятся эти популярные алгоритмы.

Глава 3. Спортивное и промышленное программирование

Не утихают споры на тему того, какова роль спортивного программирования по отношению к индустрии разработки программного обеспечения.

Помогает ли опыт участия в соревнованиях по программированию в последующей работе программистом? Следует ли расценивать спортивное программирование как чисто интеллектуальную игру, вроде шахмат, имеющую мало точек соприкосновения с реальным миром? Или даже, хуже того, правда ли, что олимпиадники пишут только нечитаемый код и не умеют работать в команде?

Можно встретить самые различные, нередко диаметрально противоположные мнения на эти темы. На наш взгляд, ответ на все эти вопросы — «и да, и нет», т. е. истина лежит посередине.

- Алгоритмическое спортивное программирование основано на фундаменте многих ключевых дисциплин информатики, таких, как теория алгоритмов, вычислительная сложность, структуры данных, архитектура компьютера. Успешное участие в олимпиадах невозможно без уверенных **знаний из фундаментальных областей** и, в свою очередь, укрепляет эти знания.
- Хотя многие алгоритмы уже реализованы в виде известных библиотек, которые в реалиях промышленной разработки ПО просто переиспользуются как готовые абстракции, рано или поздно возникают задачи, требующие **более глубокого понимания используемых «под капотом» операций** — например, в случае проблем производительности, вызванных плохо масштабирующимся алгоритмом.
- Кроме того, участие в соревнованиях дает и чисто **практические навыки**, такие, как безошибочное написание кода с первой попытки (конечно, по мере возможности), умение отлаживать программы, способность быстро обнаруживать типичные ошибки и проблемные места в коде.
- Наконец, прагматический аргумент: многие IT-компании используют задачи из области спортивного программирования для **собеседований разработчиков**. Хотя этот подход, в каком-то смысле, дает фору опытным олимпиадникам при собеседовании в такие компании, последние, вероятно, руководствуются схожими с описанными выше соображениями и предполагают ненулевую корреляцию с успешностью кандидата в будущей роли сотрудника.

- Но, разумеется, для того чтобы быть хорошим профессиональным разработчиком, требуются и многие другие навыки помимо вышеперечисленных: навыки написания поддерживаемого кода, навыки проектирования архитектуры, знание фреймворков, владение soft skills («гибкими навыками») и т. п. Участник даже с выдающимися результатами в спортивном программировании будет начинать в этих областях **с чистого листа** (если, конечно, он не успел получить соответствующий опыт).

Резюмируя, можно сказать, что есть некоторая область, в которой спортивное программирование и промышленное программирование перекрываются; размер этой области — дискуссионный вопрос.

Спортивное программирование за пределами этой области — это, возможно, «лишь» интеллектуальная игра с некоторыми элементами computer science. Впрочем, это не делает участие в соревнованиях по программированию менее интересным!

Глава 4.
Как пользоваться книгой

Данная книга предлагает определенный порядок изучения тем. По нашей задумке читатель сначала познакомится с основополагающими темами и простыми, но в то же время часто применяемыми алгоритмами. По мере продвижения к концу книги уровень сложности материала будет возрастать.

> Разумеется, читатель волен выбирать и любой другой порядок изучения тем. Большинство глав являются в этом смысле полностью независимыми друг от друга.

Особое внимание уделяется **решению задач**. Можно сказать, что лишь 20% книги составляют описания алгоритмов, а остальные 80% посвящены рассмотрению различных задач на эти алгоритмы. Это неслучайно: умение «свести» нестандартную задачу к стандартному алгоритму — это один из самых важных навыков в спортивном программировании, и приведенные примеры помогают его развить.

Задачи разделены на три группы:

1. Примеры, решения которых даны сразу же в тексте главы. Тем не менее мы рекомендуем читателю вначале **пытаться решить каждый пример самостоятельно**, хотя бы в течение нескольких минут, и только затем разбирать предложенное решение.
2. Задачи для самостоятельного решения, подсказки к которым даны в конце книги. Мы рекомендуем читателю не просто решать эти задачи в уме, но и **написать код** для решения каждой из задач.
3. Ссылки на задачи на онлайн-ресурсах, которые позволяют отправить собственное решение на автоматическую проверку.

Ступень I.
Разминка

Глава 5.
Вводные задачи

5.1. A+B

Задача. Даны два целых числа, записанных через пробел. Числа находятся в диапазоне от −1000 до 1000. Требуется вывести их сумму.

Пример

Входные данные	Требуемый результат
12 -5	7

Решение

Это одна из традиционных «разминочных» задач. Она нацелена на отработку процесса решения спортивных задач и их проверки в соответствии с требованиями конкретной тестирующей системы.

Реализация

Python

```
a, b = input().split()
print(int(a) + int(b))
```

C++

```
#include <iostream>
using namespace std;
int main() {
  int a, b;
  cin >> a >> b;
  cout << a + b << endl;
}
```

Примечание по формату ввода-вывода

Решение предполагает, что чтение входных данных производится через стандартный поток ввода, а вывод выходных — через стандартный поток вывода. Это наиболее распространенный формат в современных тестирующих системах. При ручном запуске в терминале входные данные надо будет вводить с клавиатуры, а результат будет печататься на экране. Для удобства можно использовать перенаправление ввода-вывода и другие техники.

Примечание о проверке данных на корректность

Мы не проверяем корректность входных данных (например, то, что они не содержат букв), поскольку, как это принято в спортивном программировании, все входные наборы данных гарантируются на соответствие условию. В то же время при выводе результата мы завершаем его переводом строки, согласно общепринятому формату текстовых файлов (хотя большинство тестирующих систем принимают результат и без завершающего перевода строки).

5.2. Пример решения задачи. Числа Фибоначчи

Последовательность Фибоначчи определяется следующим образом: каждое число равно сумме двух предыдущих. Первые два члена последовательности — числа 0 и 1. Ряд чисел Фибоначчи начинается так:

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, F_8 = 13, \ldots$$

Эта последовательность удивительно часто встречается при решении разнообразных задач. Несмотря на простоту определения, с числами Фибоначчи связано множество нетривиальных математических соотношений, а для их быстрого вычисления существует элегантный, хотя далеко и не очевидный, алгоритм. Эти темы будут рассмотрены в *главе 22*.

Историческая справка

Эта последовательность названа в честь итальянского математика Леонардо Пизанского, также известного как Фибоначчи, который в 1202 г. привел ее в своем труде «Книга абака». Эта последовательность дана как результат решения задачи о кроликах: предполагая, что изначально у нас есть одна пара кроликов и что пара дает потомство в виде новой пары каждый месяц начиная со второго, требуется посчитать общее число пар (разумеется, эта задача использует идеализированную модель, имеющую мало общего с реальным биологическим миром).

Впрочем, теперь установлено, что эта же последовательность была известна индийским математикам задолго до Фибоначчи: в частности, древнеиндийский математик II в. до н. э. Пингала пришел к ней, исследуя число возможных стихов заданной длины, состоящих только из слогов длины 1 и 2.

Задача. По данному n (n — целое число в отрезке [0; 90]) вывести n-е число Фибоначчи.

Примеры

Входные данные	**Требуемый результат**
7	13
Входные данные	**Требуемый результат**
8	21

Решение

При заданных ограничениях можно просто реализовать формулу так, как она описана в условии.

Python

```
n = int(input())
f = [0, 1]
while len(f) <= n:
  f.append(f[-1] + f[-2])
print(f[n])
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int64_t> f = {0, 1};
  while ((int)f.size() <= n) {
    f.push_back(f[f.size() - 1] +
                f[f.size() - 2]);
  }
  cout << f[n] << endl;
}
```

Примечание о размерах типов данных в C++

Числа Фибоначчи растут очень быстро, поэтому при использовании C++ следует учитывать возможное переполнение стандартных целочисленных типов. При заданных выше ограничениях числа Фибоначчи будут переполнять 32-битные типы, однако типа int64_t будет достаточно. Для справки, уже 47-е число Фибоначчи выходит за пределы знакового 32-битного типа, а 93-е число — переполнит 64-битные типы.

Примечание о типичной ошибке в рекурсии

Одна из типичных ошибок — реализация «в лоб» в виде рекурсивной функции, дважды вызывающей себя и не делающей никакой мемоизации (запоминания результатов). Такая реализация будет работать чрезвычайно долго, поскольку фактически число рекурсивных вызовов будет соответствовать величине ответа. Вычисление, например, 90-го числа Фибоначчи таким ошибочным алгоритмом потребует большего времени, чем возраст Вселенной.

5.3. Пример решения задачи. Манхэттенское расстояние

Задача. Рассмотрим идеализированную модель Манхэттена (одного из районов Нью-Йорка): все улицы идут от одного края острова до другого, все пересечения улиц происходят под прямым углом, и все кварталы имеют одинаковую длину. Иными словами, карта Манхэттена представляет собой прямоугольную решетку некоторого размера с фиксированным шагом, где горизонтальные и вертикальные линии на карте — улицы, а их пересечения — перекрестки, на которых можно переходить с одной улицы на другую. Пронумеруем горизонтальные улицы сверху

вниз начиная с единицы и аналогично пронумеруем слева направо вертикальные улицы. Тогда каждый перекресток описывается двумя координатами (x, y) — номерами горизонтальной и вертикальной улиц (рис. 5.1).

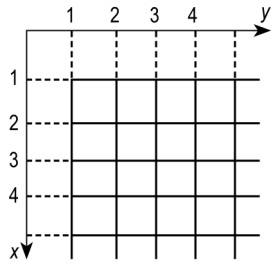

Рис. 5.1. Карта

Наша задача — помочь таксисту найти кратчайший путь между двумя заданными точками. Известно, что таксист преодолевает каждый квартал за одну и ту же единицу времени, а повороты совершает мгновенно.

В первой строке входных данных даны координаты (x_1, y_1) стартового перекрестка, во второй — координаты (x_2, y_2) конечного перекрестка. Координаты — натуральные числа, не превосходящие 10^4. Вывести требуется длину кратчайшего пути.

ПРИМЕРЫ

Входные данные	**Требуемый результат**
1 2 2 1	2
Входные данные	**Требуемый результат**
1 1 11 21	30

Решение

Заметим, что в данной модели таксисту всегда выгодно двигаться в сторону конечной точки, сокращая каждым шагом либо разницу по одной координате, либо по другой. Если следовать этой стратегии, то путь обязательно будет оптимальным (рис. 5.2).

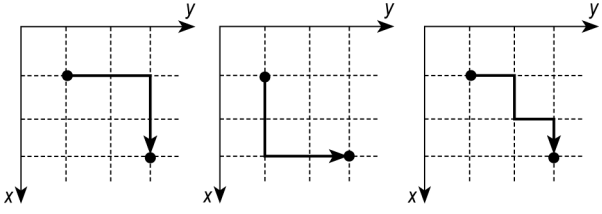

Рис. 5.2. Кратчайшие пути

Все такие пути имеют одинаковую длину, равную:

$$|x_1 - x_2| + |y_1 - y_2|.$$

Такое расстояние называется манхэттенским расстоянием, или «расстоянием городских кварталов». Оно встречается и во многих других задачах, как практических, так и чисто математических.

Реализация

Python

```
x1, y1 = input().split()
x2, y2 = input().split()
dx = abs(int(x1) - int(x2))
dy = abs(int(y1) - int(y2))
print(dx + dy)
```

C++

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
  int x1, y1, x2, y2;
  cin >> x1 >> y1 >> x2 >> y2;
  int dx = abs(x1 - x2);
  int dy = abs(y1 - y2);
  cout << dx + dy << endl;
}
```

Примечание о заголовочных файлах на C++

В языке C++ функция `std::abs()` присутствует в стандартной библиотеке в нескольких вариантах, с различными типами входного аргумента и результата. Предоставляемый заголовочным файлом cstdlib (он же stdlib.h) вариант подходит для целочисленных операций, в то время как в заголовочном файле cmath содержатся варианты для работы с числами с плавающей запятой. Это важно учитывать и не забывать подключать нужный заголовочный файл, поскольку при неправильном выборе могут возникать ошибки компиляции (когда компилятор не может выбрать среди целочисленных вариантов при компиляции вызова с дробным аргументом) или даже неверные результаты (из-за ошибок округления при вызове дробного варианта с большим целым числом).

5.4. Пример решения задачи. Путь до ксерокса

Как правило, задачи по спортивному программированию даются не в абстрактном виде, как это было с приведенными выше задачами, а в виде некой истории, повышающей реалистичность условия задачи. Эта история — часто называемая легендой — может как помогать быстрее уяснить суть задачи, так и, наоборот, маскировать какие-то важные для решения задачи детали, сделав их менее заметными в новом контексте. Так или иначе, но навык чтения условий и «отделения зерен от плевел» является очень важным для участника.

Задача. Студент Вася, ожидая в университете начала занятий по программированию, внезапно осознал, что сегодня состоится сдача и проверка домашнего задания, которого он по личным обстоятельствам делать не начинал. Вася умеет быстро ориентироваться в подобных ситуациях, поэтому он уже нашел прилежного со-

курсника с распечаткой готового решения того же, что и у Васи, варианта. Осталось лишь скопировать эту распечатку на одном из копировальных аппаратов, которые находятся в здании университета на каждом k-м этаже (т. е. на этажах с номерами k, $2k$, $3k$ и т. д.). Вася находится на этаже с номером n, и он со свойственной ему предусмотрительностью понял, что главное — как можно меньше бегать по лестницам, ведь в противном случае он сильно запыхается и не сможет выспросить у сокурсника объяснений его решения. По данным n и k (натуральным числам, не превосходящим 10^9) выведите минимальное число этажей, которое потребуется пройти Васе до ксерокса.

ПРИМЕРЫ

Входные данные	**Требуемый результат**
4 10	6
Входные данные	**Требуемый результат**
105 4	1

Решение

После выделения существенной информации из условия становится понятно, что задача заключается в минимизации $|n - ki|$ по всевозможным натуральным i.

Вместо перебора всех вариантов, который может работать продолжительное время, заметим, что достаточно рассматривать только $i = \lfloor n / k \rfloor$ и на единицу большую величину; в терминах условия эти два варианта соответствуют ближайшему ксероксу этажом ниже и этажом выше.

Единственная тонкость — отбросить вариант с $i = 0$, поскольку по условию на нулевом этаже ксерокса нет.

Реализация

Python

```
n, k = map(int, input().split())
i = max(n // k, 1)
ans1 = abs(n - k * i)
i += 1
ans2 = abs(n - k * i)
print(min(ans1, ans2))
```

C++

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  int i = max(n / k, 1);
  int ans1 = abs(n - k * i);
  ++i;
  int ans2 = abs(n - k * i);
  cout << min(ans1, ans2) << endl;
}
```

5.5. Пример решения задачи. Проверка перестановки

Определение. Перестановкой длины *n* называется набор чисел 1, 2, …, *n*, выписанных в некоторой последовательности.

Задача. Дана последовательность натуральных чисел. Требуется вывести «OK», если она является перестановкой, и «BAD» в противном случае.

Входной набор состоит из единственной строки, в которой записано одно или несколько натуральных чисел, разделенных пробелом.

ПРИМЕРЫ

Входные данные	**Требуемый результат**
1 3 2	OK
Входные данные	**Требуемый результат**
2 3 4	BAD

Решение

Обозначим через *n* количество чисел во входном наборе. Тогда достаточно проверить, что числа не превосходят *n* и что никакое число не встречается несколько раз.

Реализация

Python

```
from collections import *
a = list(map(int, input().split()))
cnt = Counter(a)
ans = (max(a) == len(a) and
       max(cnt.values()) == 1)
print("OK" if ans else "BAD")
```

C++

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;
int main() {
  string s;
  getline(cin, s);
  istringstream stream(s);
  vector<int> a;
  int x;
  while (stream >> x)
    a.push_back(x);
  int n = (int)a.size();
  vector<int> cnt(n);
  bool ok = true;
  for (auto x : a) {
    if (x > n || cnt[x - 1])
      ok = false;
```

```
    else
      ++cnt[x - 1];
  }
  cout << (ok ? "OK" : "BAD")
       << endl;
}
```

5.6. Пример решения задачи. Циклы перестановки

Определение. Пусть дана некоторая перестановка длины n. Выберем стартовый элемент — целое число от 1 до n — и будем двигаться, каждый раз переходя от числа i к числу, записанному в перестановке на i-м месте. Утверждается, что рано или поздно мы вернемся к стартовому элементу; назовем циклом пройденный путь от стартовой точки вплоть до возврата в нее.

Задача. Дана перестановка. Требуется вывести количество циклов в ней.

Примеры

Входные данные	Требуемый результат
1 3 2	2
Входные данные	**Требуемый результат**
2 3 4 1	1

Пояснения к примерам

В первом примере есть два цикла: (1) и (32), поскольку единица переходит в единицу, тройка — в двойку, а двойка — в тройку.

Во втором примере единственный цикл — (1234), поскольку единица переходит в двойку, двойка — в тройку, тройка — в четверку, четверка — в единицу.

Решение

Для начала заметим, что никакие два цикла не пересекаются друг с другом, т. е. любая перестановка распадается на набор отдельных циклов. В самом деле, если бы два цикла пересекались, то существовал бы общий элемент x, в который переходят два различных элемента. Однако тогда в двух позициях перестановки должно было быть записано одно и то же число x, что невозможно по определению перестановки.

Отсюда следует простой алгоритм поиска циклов: будем поочередно брать каждый непосещенный элемент и строить из него цикл, помечая все элементы цикла как посещенные. При таком обходе каждый цикл будет просмотрен ровно один раз, поэтому ответ будет равен числу совершенных проходов.

Реализация

Python

```
perm = map(int, input().split())
perm = [x - 1 for x in perm]
n = len(perm)
ans = 0
visited = [False] * n
for start in range(n):
  if visited[start]:
    continue
  ans += 1
  current = start
  while True:
    visited[current] = True
    current = perm[current]
    if current == start:
      break
print(ans)
```

C++

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;
int main() {
  string s;
  getline(cin, s);
  istringstream stream(s);
  vector<int> perm;
  int x;
  while (stream >> x)
    perm.push_back(x - 1);
  int n = (int)perm.size();
  int ans = 0;
  vector<char> visited(n);
  for (int start = 0; start < n;
       ++start) {
    if (visited[start])
      continue;
    ++ans;
    int current = start;
    do {
      visited[current] = true;
      current = perm[current];
    } while (current != start);
  }
  cout << ans << endl;
}
```

Примечание по укороченной реализации

В случае если массив самой перестановки больше ни для чего не нужен, эти реализации можно немного сократить за счет перезаписывания пройденных в перестановке элементов значением «−1». В такой реализации массив `visited` не понадобится.

Глава 6. Разминочные конструктивные задачи

Конструктивными задачами в спортивном программировании называют такие задачи, в которых основная сложность заключается в нахождении требуемой ключевой идеи, а реализация решения зачастую проста. Искомая ключевая идея может быть какой-либо формулой либо некой конструкцией, гарантированно дающей правильный ответ.

6.1. Пример решения задачи. Пара с минимальным произведением

Задача. Дан массив из n целых чисел; $2 \le n \le 10^5$, и числа по модулю не превосходят 10^5. Требуется выбрать два числа из массива так, чтобы получить в результате их перемножения минимальное возможное значение. Входные данные состоят из числа n, после которого в отдельной строке даны n разделенных пробелом чисел. Выведите искомое наименьшее произведение.

Примеры

Входные данные	**Требуемый результат**
4 1 2 3 4	2
Входные данные	**Требуемый результат**
4 -1 2 3 4	-4

Решение

Если массив состоит только из положительных чисел, то ответ очевиден: произведение двух минимумов этого массива.

Более интересен случай, когда в массиве есть отрицательные числа. В таком случае комбинацией отрицательного и положительного числа мы получим отрицательное число, которое будет тем меньше, чем больше по модулю выбранные числа. Иными словами, при наличии как отрицательных, так и положительных чисел искомым будет наименьшее из отрицательных чисел, умноженное на наибольшее из положительных.

Однако здесь есть и каверзный особый случай: это случай, когда все числа в массиве отрицательны. В таком случае произведение будет положительным, и для его минимизации потребуется взять два наибольших отрицательных числа.

Что касается нулей — которые мы временно убрали из рассмотрения для упрощения рассуждений — то в приведенных выше выкладках ничего не меняется, если заменить слово «положительный» на «неотрицательный».

> ***Примечание о разнице между понятиями «положительный» и «неотрицательный»***
>
> Описанный выше порядок решения — сначала рассмотреть задачу для строго положительных чисел и только затем перейти к более общему случаю неотрицательных чисел — соответствует обычному процессу размышления над какой-либо задачей. Мы специально избегаем стиля типичных математических доказательств, когда автор текста знает наперед все тонкости и сложности и тщательно учитывает граничные случаи, начиная с самой первой строчки доказательства. Этот стиль был бы оправдан в математических трудах, излагающих некоторые результаты и обосновывающих их истинность. Наша же задача совершенно другая: мы пытаемся помочь читателю проследовать путь придумывания решения и его доказательства, с тем чтобы он мог самостоятельно решать похожие задачи в будущем.

Реализация

Заметим, что реализовать это решение можно различными подходами, однако один из самых простых способов — это посчитать три ответа тремя описанными выше способами (два минимума, два максимума и минимум с максимумом) и выбрать наилучший среди них. При этом реализацию поиска минимумов и максимумов проще всего выполнить путем сортировки всего массива.

Python

```
n = int(input())
a = list(map(int, input().split()))
a.sort()
ans1 = a[0] * a[1]
ans2 = a[0] * a[-1]
ans3 = a[-2] * a[-1]
print(min(ans1, ans2, ans3))
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int64_t> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  sort(a.begin(), a.end());
  int64_t ans1 = a[0] * a[1];
  int64_t ans2 = a[0] * a[n - 1];
  int64_t ans3 = a[n - 2] *
                 a[n - 1];
  int64_t ans = min(
      ans1, min(ans2, ans3));
  cout << ans << endl;
}
```

Примечание по типам данных в C++

Следует учитывать, что, хотя множители помещаются в 32-битный тип данных, для произведения требуется уже 64-битный тип. Для простоты реализации — чтобы не производить приведение типа при каждом перемножении — мы использовали 64-битный тип и для хранения множителей тоже.

6.2. Пример решения задачи. Тайная жизнь деревьев

Задача. Лесничий Петя раскрыл очередную тайну растительного мира. Оказывается, деревья некоторых видов начинают цвести не в случайное время после наступления подходящих погодных условий. Петя обнаружил, что деревья неким образом «договариваются» друг с другом о начале цветения, чтобы цвести как можно дольше вместе с сородичами и тем самым повысить шансы на удачный обмен генами.

Принцип достаточно прост: как только дерево обнаруживает (с помощью специального механизма), что достаточное число сородичей уже цветут, то оно начинает цвести и само. Однако это число — пороговое число сородичей — не общее для всего этого вида, а является характеристикой конкретного дерева. Для каждого из n деревьев в его лесу Петя определил это пороговое число a_i (которое является неотрицательным целым числом). Деревья, у которых $a_i = 0$, начинают цвести весной самостоятельно, независимо от состояния других.

Петя задумался: сколько деревьев надо ему подсадить в лес, чтобы весь лес зацвел следующей весной? Отметим, что у Пети в запасе много саженцев, так что добавляемые деревья могут иметь произвольные параметры по его усмотрению.

Первая строка входного набора содержит число n ($1 \le n \le 1000$). Во второй строке записаны n разделенных пробелом чисел a_i ($1 \le a_i \le 1000$). Нужно вывести наименьшее число деревьев, которое требуется добавить.

Примеры

Входные данные	**Требуемый результат**
4 0 0 3 3	1
Входные данные	**Требуемый результат**
3 2 0 1	0

Решение

Сразу заметим, что подсаживать имеет смысл только деревья с пороговым числом $a = 0$, потому что использование деревьев с бóльшими числами позволит достичь либо такого же, либо худшего результата.

Для решения задачи будем рассматривать деревья в порядке увеличения (или, если быть точным, неубывания) их пороговых величин — попросту говоря, отсортируем массив a; обозначим отсортированный массив через b. Чтобы зацвело первое дерево в этом порядке, очевидно, недостает b_1 дополнительных деревьев. Чтобы зацвело второе дерево, недостает $b_2 - 1$ дополнительных деревьев (считая, что цветение первого дерева мы обеспечили). Продолжая эту логику, получаем, что для цветения i-го дерева требуется подсадить $b_i - i + 1$ дополнительных деревьев. Таким образом, ответом ко всей задаче является следующая величина:

$$\max_{i=1\ldots n}\{b_i - i + 1\}.$$

(Как легко убедиться, эта формула остается верной и при наличии повторяющихся элементов в массиве b. Кроме того, среди чисел $b_i - i + 1$ могут быть «фиктивные» отрицательные — в случае, когда уже цветущих деревьев достаточно с избытком — однако в конечном итоге формула даст неотрицательную величину, поскольку как минимум для $i = 1$ значение является неотрицательным.)

Реализация

Python

```
n = int(input())
a = map(int, input().split())
b = sorted(a)
ans = max(cur - i for i, cur
          in enumerate(b))
print(ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  sort(a.begin(), a.end());
  int ans = 0;
  for (int i = 0; i < n; ++i)
    ans = max(ans, a[i] - i);
  cout << ans << endl;
}
```

Примечание об индексации массивов

Формулы в приведенных выше программах не содержат компонента «+1», поскольку индексация массивов в Python и C++ производится начиная с нуля.

6.3. Пример решения задачи. Подписывание открыток

Задача. Маша и Петя подписывают пригласительные открытки для своих друзей. Чтобы открытки получились оригинальными, помимо текста на каждой открытке

рисуется маленькая картинка. Подписывание текста занимает 10 минут, рисование картинки — 20 минут. Если Маша подписывает текст на какой-либо открытке, то рисовать картинку на ней должен Петя, и наоборот. И Маша, и Петя одинаково хорошо справляются с обеими задачами, а при необходимости они даже могут одновременно работать над одной и той же открыткой: в то время как один или одна подписывает текст, другой или другая может рисовать картинку. Какое наибольшее число открыток могут изготовить Маша и Петя, если у Маши есть a минут свободного времени, а у Пети — b минут?

Входные данные состоят из двух чисел a и b, разделенных пробелом; оба числа — неотрицательные целые, не превосходящие 10^{18}. Вывести требуется единственное число — максимально возможное количество изготовленных открыток.

Примеры

Входные данные	**Требуемый результат**
`41 80`	`4`
Входные данные	**Требуемый результат**
`100 90`	`6`

Пояснения к примерам

В первом примере Маша должна подписать все четыре открытки, потратив на это $4 \cdot 10 = 40$ минут, а Петя — нарисовать картинки на всех них, потратив $4 \cdot 20 = 80$ минут.

Во втором примере Маша и Петя должны поровну разделить работу: Маша должна подписать текст на трех открытках и нарисовать картинку на трех оставшихся открытках; аналогично и Петя.

Решение

Если бы ограничения на входные числа были меньше, то найти решение можно было бы, перебрав количество заданий «подписать текст», сделанных Машей, и вычислив по оставшимся временам количество остальных заданий, которые могут сделать Маша и Петя, и, тем самым, возможное число открыток. Однако это решение — слишком медленное при заданных ограничениях: потребовалось бы перебрать 10^{18} вариантов.

Вместо этого постараемся вывести взаимосвязь между величинами входных чисел a и b и искомым результатом x. Поскольку каждая открытка требует минимум 10 минут времени как от Маши, так и от Пети, то можно утверждать:

$$10x \le a,$$
$$10x \le b.$$

Теперь осталось учесть дополнительные 10 минут времени, которые требует каждая открытка у одного из них. Для этого посмотрим на свободное время, оставшееся у Маши и Пети вместе — это $a - 10x$ и $b - 10x$ соответственно; поскольку дробить задачи нельзя, то каждую из этих величин следует поделить на 10 и округлить

вниз, чтобы получить число открыток, над которыми они могут поработать эти дополнительные 10 минут времени. Таким образом, в сумме получаем ограничение:

$$x \le \left\lfloor \frac{a-10x}{10} \right\rfloor + \left\lfloor \frac{b-10x}{10} \right\rfloor,$$

что можно упростить до:

$$3x \le \lfloor a/10 \rfloor + \lfloor b/10 \rfloor.$$

Комбинируя это ограничение с двумя другими ограничениями, выведенными нами в начале решения, мы получаем ответ как минимум из трех величин:

$$x_{\max} = \min\left(\lfloor a/10 \rfloor, \lfloor b/10 \rfloor, \left\lfloor \frac{\lfloor a/10 \rfloor + \lfloor b/10 \rfloor}{3} \right\rfloor \right).$$

Реализация

Python

```
a, b = map(int, input().split())
a //= 10
b //= 10
c = (a + b) // 3
print(min(a, b, c))
```

C++

```
#include <algorithm>
#include <cstdint>
#include <iostream>
using namespace std;
int main() {
  int64_t a, b;
  cin >> a >> b;
  a /= 10;
  b /= 10;
  cout << min({a, b, (a + b) / 3})
       << endl;
}
```

Примечание о типе данных в решении на C++

Типичный баг в подобной задаче — забыть, что входные числа могут достигать величины 10^{18}; при использовании типа данных int, который обычно является 32-разрядным, такие большие числа не будут помещаться в переменные, что приведет к ошибке выполнения. Следует всегда внимательно следить за ограничениями и выбирать подходящие типы данных, в данном случае — использовать 64-разрядный тип.

6.4. Пример решения задачи. Исправление перестановки

Задача. Дана последовательность натуральных чисел. Требуется за минимальное число операций превратить ее в перестановку. Операция заключается в изменении одного элемента последовательности на другое значение (по нашему усмотрению).

Входной набор состоит из двух строк: в первой строке дана длина последовательности, а во второй строке дана сама последовательность в виде разделенных пробе-

лами натуральных чисел. Все числа входного набора меньше либо равны 100. Вывести требуется искомое наименьшее количество операций (в первой строке) и полученную перестановку (во второй строке, разделяя элементы перестановки пробелом). Если возможны несколько решений, разрешается вывести произвольное.

Примеры

Входные данные	**Требуемый результат**
4 1 100 3 3	2 1 4 3 2
Входные данные	**Требуемый результат**
3 3 2 1	0 3 2 1

Пояснения к примерам

В первом примере достаточно двух замен, чтобы из данной последовательности получить перестановку; при этом есть несколько вариантов решения: в приведенном примере число 100 было заменено на 4, а второе число 3 — на 2.

Во втором примере входная последовательность уже является перестановкой, поэтому никаких замен не требуется.

Решение

Обозначим через n длину входной последовательности. Во-первых, заметим, что числа, превосходящие n, потребуется заменить в любом случае, поскольку они не могут входить в корректную перестановку. Во-вторых, то же относится к повторяющимся числам из диапазона $[1; n]$: потребуется оставить только по одному экземпляру каждого из чисел, а все остальные — заменить. Поскольку и те, и другие замены — неизбежные, то ответ не может быть меньше, чем суммарное количество таких «плохих» чисел.

Попытаемся придумать стратегию, которая достигает решения путем замены только «плохих» чисел. С учетом вышесказанного эта стратегия будет оптимальной, поскольку обойтись меньшим числом операций невозможно.

Для создания стратегии посмотрим на «недостающие» числа, т. е. такие натуральные числа, не превосходящие n, которые не встречаются в исходной последовательности. Можно заметить, что их количество в точности равно количеству «плохих» чисел. Это можно концептуализировать следующим образом: предположим, что есть n корзин (по одной для каждого возможного значения от 1 до n), и i-я корзина содержит элементы, равные i. Тогда в корректной перестановке каждая корзина содержит ровно один элемент, в то время как при наличии «плохих» чисел каждое такое число вместо занятия своей собственной корзины либо лежит в какой-то уже занятой корзине, либо не лежит ни в одной корзине.

Таким образом, стратегию решения можно сформулировать следующим образом. Будем по очереди брать каждое «плохое» число и исправлять его, заменяя на любое

из «недостающих» чисел. Поскольку, по условию, никаких дополнительных целей (как, например, лексикографической минимизации) не ставится, порядок выбора «плохих» и «недостающих» чисел не важен — ответ будет в любом случае оптимален.

Реализация

Python

```
n = int(input())
a = list(map(int, input().split()))
free = set(range(1, n + 1))
free -= set(a)
print(len(free))
used = set()
for i in range(n):
  if a[i] > n or a[i] in used:
    a[i] = free.pop()
  used.add(a[i])
print(*a)
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<int> cnt(n + 1);
  for (auto x : a) {
    if (x <= n)
      ++cnt[x];
  }
  vector<int> free;
  for (int i = 1; i <= n; ++i) {
    if (!cnt[i])
      free.push_back(i);
  }
  cout << free.size() << endl;
  for (auto x : a) {
    if (x <= n)
      --cnt[x];
    if (x > n || cnt[x]) {
      x = free.back();
      free.pop_back();
    }
    cout << x << ' ';
  }
  cout << endl;
}
```

Примечание по формату вывода решения на C++

Внимательный читатель заметил, что решение на C++ выводит лишний пробел: это пробел после последнего элемента перестановки. Хотя это было бы легко поправить (например, с помощью дополнительной проверки), обычно подобные усложнения не требуются в спортивном программировании, поскольку типичные тестирующие системы настроены на игнорирование подобных заключительных пробелов. Впрочем, изредка встречаются и задачи, где лишний пробел приведет к незачтенному решению — например, такое может быть в задачах, где требуется выводить текст.

6.5. Пример решения задачи. Минимальный палиндром

Определение. Палиндромом называется такая строка, которая одинаково читается в обоих направлениях. Например level и anna являются палиндромами, а label и coco — нет.

Задача. Дана строка. Из строки можно удалять буквы и переставлять их местами. Требуется путем таких преобразований построить палиндром с наибольшей возможной длиной. Среди всех решений с одинаковой длиной требуется найти лексикографически минимальное.

Лексикографическое сравнение определяется следующим образом. Одна строка считается лексикографически меньше другой, если в позиции их первого различия (т. е. в позиции, следующей после наибольшего общего префикса) буква первой строки имеет меньший порядковый номер в алфавите, чем буква второй строки. Частный случай: если одна строка начинается с другой, то она считается лексикографически большей.

Входной набор состоит из единственной строки, состоящей из строчных букв латинского алфавита. Длина строки не превосходит 100. Требуется вывести строку — искомый палиндром.

Примеры

Входные данные	**Требуемый результат**
nana	anna
Входные данные	**Требуемый результат**
aabbcd	abcba

Решение

Сначала решим задачу нахождения наибольшего по длине палиндрома. Для этого заметим, что в любом палиндроме все буквы, кроме, возможно, одной, встречаются четное число раз. И, наоборот, если у нас есть некоторый набор букв, в котором каждая буква содержится четное число раз, мы всегда можем построить из него палиндром и затем по желанию добавить еще одну букву в середине. Например, из букв aaaabb можно построить палиндромы abaaba, aabbaa и baaaab, а из букв aaaabbc — палиндромы abacaba, aabcbaa и baacaab.

Таким образом, алгоритм определения наибольшей длины следующий: подсчитаем количество вхождений каждой буквы в исходной строке, затем просуммируем эти количества, отнимая единицу от всех нечетных чисел, кроме одного.

Теперь перейдем к решению второй части задачи: лексикографической минимизации. Заметим, что в любом палиндроме правая половина однозначно определяется левой, таким образом, для лексикографической минимизации правая половина роли не играет. Из решения первой части задачи мы знаем, какие и сколько символов

должны стоять в половине палиндрома. Очевидно, что для минимизации следует их выстроить в алфавитном порядке. Например, для букв aaaabb наименьший возможный палиндром — aabbaa.

Единственный нетривиальный момент — средняя позиция в случае, когда палиндром имеет нечетную длину. У нас может быть несколько букв-кандидатов для этой позиции (это все буквы, встречающиеся нечетное число раз в исходной строке). Тогда, очевидно, выгодно взять наименьшую из доступных букв и поставить ее в середину палиндрома. Например, для букв aaaabbcde наименьшим палиндромом будет aabcbaa.

Реализация

Python

```
from collections import *
s = input()
cnt = Counter(s)
half = ""
mid = ""
for c in sorted(cnt.keys()):
  half += c * (cnt[c] // 2)
  if cnt[c] % 2 and not mid:
    mid = c
ans = half + mid + half[::-1]
print(ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;
int main() {
  string s;
  cin >> s;
  int cnt[26] = {0};
  for (auto c : s)
    ++cnt[c - 'a'];
  string half, mid;
  for (int i = 0; i < 26; ++i) {
    char c = 'a' + i;
    half += string(cnt[i] / 2, c);
    if (cnt[i] % 2 && mid.empty())
      mid += c;
  }
  string half2 = half;
  reverse(half2.begin(),
          half2.end());
  string ans = half + mid + half2;
  cout << ans << endl;
}
```

6.6. Пример решения задачи. Исключающее ИЛИ от 1 до n

Определение. Логическая операция «исключающее ИЛИ» (*англ.* exclusive or, сокращенно — XOR) определяется следующим образом: результат равен «истине», когда один аргумент истинен, а другой — ложен. Это определение можно пояснить

следующей таблицей истинности (где символ ⊕ обозначает XOR, ноль — истинное значение, единица — ложное значение):

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	0
1	1	1

Битовой операцией XOR для двух чисел называется результат применения логической операции XOR к соответствующим битам этих чисел попарно. Например:

a	b	$a \oplus b$
$1 = 1_2$	$1 = 1_2$	$0 = 0_2$
$2 = 10_2$	$3 = 11_2$	$1 = 1_2$
$6 = 110_2$	$11 = 1011_2$	$13 = 1101_2$

Для наглядности в таблице выше помимо десятичной системы счисления мы использовали также двоичную систему.

Задача. Дано число n. Требуется посчитать $1 \oplus 2 \oplus \ldots \oplus n$, т. е. результат применения битовой операции XOR ко всем числам от 1 до n. Известно, что n — натуральное число, не превосходящее 10^{18}.

Примеры

Входные данные	**Требуемый результат**
2	3
Входные данные	**Требуемый результат**
5	1

Решение

Входные ограничения достаточно большие, чтобы результат нельзя было посчитать методом «брутфорс», т. е. проходом по всем числам от 1 до n. Это несложно проверить, написав такую тривиальную программу и запустив ее на числах порядка 10^{10} и более.

Постараемся найти более быстрое решение. Для этого заметим, что бóльшая часть битов в этой XOR-сумме взаимно уничтожается. Ключевой факт — для любого четного x верно следующее:

$$x \oplus (x+1) = 1.$$

В самом деле, четное число x отличается от числа $x + 1$ только в самом младшем бите: в то время как в числе x младший бит равен нулю, в числе $x + 1$ он равен единице.

Используя этот факт, мы можем существенно упростить искомую XOR-сумму, сгруппировав попарно ее элементы и заменив каждую пару единицей:

$$1 \oplus 2 \oplus 3 \oplus \ldots = 1 \oplus (2 \oplus 3) \oplus (4 \oplus 5) \oplus \ldots = 1 \oplus 1 \oplus 1 \oplus \ldots .$$

В зависимости от n эта сумма состоит либо только из единиц, либо из единиц и числа n в конце. При этом если единиц четное число, то в XOR-сумме они взаимно уничтожают друг друга с нулевым результатом; если же единиц нечетное число, то их XOR-сумма равна единице. Объединяя эти результаты, получаем закономерность, повторяющуюся каждые четыре элемента. Если обозначим через k величину n mod 4 (т. е. остаток от деления n на четыре), то:

- если $k = 1$, то ответ равен 1;
- если $k = 2$, то ответ равен $1 \oplus n$ (или, что в данном случае то же самое, $1 + n$);
- если $k = 3$, то ответ равен 0;
- если $k = 0$, то ответ равен n.

Эта закономерность позволяет мгновенно вычислить ответ на задачу для любого n.

Альтернативный способ вывода закономерности

Вместо аналитического вывода закономерности можно было бы прийти к этому же результату и «эмпирическим» путем, а именно, написав тривиальную программу, вычисляющую результат «брутфорсом». Распечатав на экран таблицу результатов для первых пары десятков чисел, было бы легко заметить паттерны в этих результатах. Это распространенный прием в спортивном программировании: если задача такова, что решение «в лоб» написать очень легко и есть надежда на существование некой закономерности, то целесообразно потратить на это несколько минут и попытаться обнаружить ее наличие эмпирически. В отличие от математических олимпиад или промышленного программирования, в спортивном программировании не требуется иметь доказательство решения, для того чтобы успешно сдать задачу.

Реализация

Python

```
n = int(input())
k = n % 4
if k == 1:
  ans = 1
elif k == 2:
  ans = n + 1
elif k == 3:
  ans = 0
else:
  ans = n
print(ans)
```

C++

```
#include <iostream>
using namespace std;
int main() {
  int n;
  cin >> n;
  int ans;
  switch (n % 4) {
    case 1: ans = 1; break;
    case 2: ans = n + 1; break;
    case 3: ans = 0; break;
    case 0: ans = n; break;
  }
  cout << ans << endl;
}
```

6.7. Пример решения задачи. Врачи и посетители

Задача. В маленькой клинике работают всего два врача. Несколько пациентов записались на прием в клинику в определенные часы; изначально неизвестно, к которому из врачей попадет посетитель. Ваша задача — спланировать работу врачей, т. е. определить, кто должен принять первого пациента, кто — второго и т. д. Разумеется, каждый врач может работать только с одним пациентом в каждый момент времени (однако допускается принять нового пациента в тот же момент времени, когда посещение другим пациентом завершается).

Входные данные начинаются с числа n — натурального числа, не превосходящего 100 и задающего количество посетителей. Затем следуют описания запланированных посещений: два целых числа l_i и r_i ($0 < l_i < r_i < 1000$). Вывести требуется строку из n символов, в которой i-й символ должен быть «A» или «B» в зависимости от того, к первому или ко второму врачу должен пойти i-й пациент. Если существует несколько возможных решений, разрешается вывести любое из них. Если решения нет, то следует вывести «No solution».

Примеры

Входные данные	Требуемый результат
3 1 2 2 3 1 3	AAB
Входные данные	**Требуемый результат**
3 1 3 1 4 2 4	No solution

Пояснения к примерам

В первом примере приведенное решение «AAB» обозначает, что в момент времени 1 пациент №1 заходит на прием к первому врачу, а пациент №3 — ко второму врачу. В момент времени 2 пациент №1 уходит и вместо него на прием к первому врачу заходит пациент №2. В момент времени 3 пациенты №2 и 3 уходят.

Решение

Попробуем строить решение, постепенно добавляя пациентов по одному в порядке роста времени начала приема. Стартовая ситуация — пациентов нет, и у каждого из врачей текущий план приема пуст. При добавлении очередного пациента отправим

его к тому врачу, который может принять его согласно текущему плану приема. Если оба врача могут принять добавляемого пациента, то отправим его к любому из врачей — например, к первому. Если же ни один из врачей не может этого сделать, то выведем «No solution» и завершим решение.

На первый взгляд кажется, что такое решение может привести к тупику и ошибочному «No solution» из-за неправильно сделанного на предыдущем шаге выбора. Однако так ли это на самом деле?

Жадные алгоритмы

Описанный выше алгоритм можно назвать «жадным алгоритмом». Не вдаваясь в детали терминологии, «жадными» называются такие алгоритмы, которые строят решение постепенно и при этом на каждом шаге ориентируются лишь на достижение локального оптимума для этого шага. Например, в описанном выше алгоритме при распределении очередного пациента мы никогда не меняем построенный для предыдущих пациентов план и также не учитываем последующих пациентов.

Доказательство решения

Вначале попытаемся понять, в каких случаях описанный выше алгоритм завершается с результатом «No solution». Обозначим через k номер пациента, на котором алгоритм остановился из-за невозможности подобрать ему врача. То, что оба врача заняты, означает, учитывая рассмотрение пациентов в порядке увеличения l_i, что есть два других пациента x и y таких, что $r_x > l_k$ и $r_y > l_k$. Попросту говоря, «No solution» выводится алгоритмом только тогда, когда во входном наборе есть три пациента, периоды приема которых накладываются.

Однако легко понять, что во всех таких случаях никакого решения существовать и не может: невозможно распределить три накладывающихся посещения по двум врачам.

Таким образом, приведенный алгоритм верен: он находит решение во всех случаях, кроме тех, когда решения существовать и не может.

Важность порядка рассмотрения пациентов

То, что приведенный выше алгоритм рассматривает пациентов в порядке увеличения l_i — критически важная деталь. Если бы мы рассматривали пациентов в произвольном порядке, то алгоритм мог бы зайти в тупик, когда очередного пациента добавить нельзя из-за пересечения с пациентами обоих врачей, однако «тройного» пересечения нет, и, как следует из вышеприведенного доказательства, решение на самом деле существует. Пример входных данных, на котором возникнет эта проблема: $l_1 = 1$, $r_1 = 2$, $l_2 = 9$, $r_2 = 10$, $l_3 = 2$, $r_3 = 10$, $l_4 = 1$, $r_4 = 3$. Исправить такую тупиковую ситуацию простыми способами невозможно, поскольку для этого потребовалось бы переделать планы приема предшествующих пациентов, что может вызвать новые конфликты. Это поучительный пример того, как нехватка даже одной небольшой детали делает жадный алгоритм неверным и практически неисправимым.

Реализация

Python

```
import sys
n = int(input())
a = []
for index in range(n):
  left, right = input().split()
  a.append([int(left), int(right),
            index])
doc_free_time = [0] * 2
ans = [None] * n
for left, right, index in \
    sorted(a):
  for doc in range(2):
    if doc_free_time[doc] <= left:
      doc_free_time[doc] = right
      ans[index] = "AB"[doc]
      break
  else:
    print("No solution")
    sys.exit(0)
print("".join(ans))
```

C++

```
#include <algorithm>
#include <iostream>
#include <string>
#include <tuple>
#include <vector>
using namespace std;
struct item {
  int left, right, index;
  bool operator<(
    const item& other) const {
    return std::tie(left, right) <
           std::tie(other.left,
                    other.right);
  }
};
int main() {
  int n;
  cin >> n;
  vector<item> a(n);
  for (int i = 0; i < n; ++i) {
    cin >> a[i].left >> a[i].right;
    a[i].index = i;
  }
  sort(a.begin(), a.end());
  int doc_free_time[2] = {0};
  string ans(n, 0);
  for (int i = 0; i < n; ++i) {
    for (int doc = 0; doc < 2;
         ++doc) {
      if (doc_free_time[doc] <=
          a[i].left) {
        doc_free_time[doc] =
          a[i].right;
        ans[a[i].index] =
          'A' + doc;
        break;
      }
    }
    if (!ans[a[i].index]) {
      cout << "No solution"
           << endl;
      return 0;
    }
  }
  cout << ans << endl;
}
```

6.8. Пример решения задачи. Минимизация перепадов

Задача. Вокруг бассейна на курорте стоят n пальм. По соображениям эстетики владелец курорта приказал переставить пальмы (они растут в больших горшках) так, чтобы стоящие рядом пальмы не сильно отличались друг от друга по высоте. Формально, если a_i — высота i-й пальмы ($i = 0 \ldots n - 1$), то требуется найти такую перестановку p_i, чтобы следующая величина — наибольший перепад — принимала минимально возможное значение:

$$\max_{i=0\ldots n-1} \left| a_{p_i} - a_{p_{(i+1) \bmod n}} \right|.$$

Входные данные состоят из количества n пальм в первой строке и их высот a_i, записанных через пробел во второй строке. Все числа натуральные; n не превосходит 10^5, a_i — не превосходят 10^9. Вывести требуется оптимальную перестановку p_i, т. е. n разделенных пробелами различных целых чисел из отрезка $[0; n - 1]$. Если существует несколько решений, минимизирующих наибольший перепад, то вывести разрешается любое.

Примеры

Входные данные	**Требуемый результат**
4 1 2 3 4	0 2 3 1
Входные данные	**Требуемый результат**
1 123	0

Пояснения к примерам

В первом примере ответ предлагает образовать следующий круг из пальм: 1, 3, 4, 2. Перепады в этом круге равны соответственно: 2, 1, 2, 1. Таким образом, искомый наибольший перепад равен 2.

Во втором примере всего одна пальма, поэтому существует единственно возможный порядок, который и является ответом.

Решение

Если проанализировать примеры из условия, то видно, что простая сортировка входного массива не даст правильного решения: перепад в этом случае составит $\max\{a_i\} - \min\{a_i\}$, что хуже, чем приведенный ответ в первом примере. Какие есть другие подходы к решению этой задачи?

Постараемся понять, как можно было бы построить наилучший ответ. Рассмотрим, например, самую высокую пальму: с какими пальмами она должна соседствовать для минимизации ее собственных перепадов? Это вторая и третья по высоте пальмы. Предположим, что мы действительно ставим эти три пальмы в этом

порядке: вторая по высоте пальма, затем самая высокая пальма, затем третья по высоте пальма.

Попытаемся продолжить это решение: какую пальму надо поставить рядом со второй пальмой, а какую — рядом с третьей? Учитывая, что первая пальма уже занята, ближайшие по высоте — четвертая и пятая пальмы. Тогда у нас есть два варианта: вторая+четвертая и третья+пятая, либо вторая+пятая и третья+четвертая. Первый вариант лучше или, по крайней мере, не хуже, поскольку второй вариант менее сбалансирован из-за прыжка от второй к пятой пальме.

Продолжая аналогичные рассуждения, мы придем к следующей стратегии. Для удобства обозначим через b_i отсортированный массив a_i. Тогда стратегия будет выглядеть следующим образом:

$$b_1, b_3, b_5, \ldots, b_n, \ldots, b_6, b_4, b_2,$$

иными словами — сначала нужно взять все элементы с нечетными индексами в порядке возрастания, затем — все элементы с четными индексами в порядке убывания.

В зависимости от правил конкретного соревнования и желаемой степени риска на этом рассуждении можно было бы остановиться и попытаться реализовать (и сдать) эту стратегию. Однако приведем и доказательство этого решения.

Доказательство

Покажем, что это решение — оптимальное. Доказательство требуется только для случая $n \geq 3$, поскольку при $n \leq 2$ любой порядок дает одинаковый перепад.

Заметим, что предложенный в решении вариант дает перепад, равный $\max_{i=0\ldots n-3}\{b_{i+2} - b_i\}$. Докажем оптимальность найденного решения от противного: предположим, что есть решение, перепад в котором меньше $b_{k+2} - b_k$ для какого-либо k. Это означает, что рядом с пальмой b_{k+1} должны стоять пальмы b_k и b_{k+2}. Однако поскольку пальмы стоят в виде круга, то помимо этой последовательности (b_k, b_{k+1}, b_{k+2}), «набирающей» высоту от b_k до b_{k+2}, должна быть также и другая последовательность пальм, «опускающая» высоту обратно. Мы пришли к противоречию, поскольку без уже задействованной пальмы b_{k+1} для этого потребуется прыжок как минимум на $b_{k+2} - b_k$.

Таким образом, предложенное выше конструктивное решение — верно. Его реализация требует $O(n \log n)$ времени для сортировки.

Реализация

Python

```
n = int(input())
a = map(int, input().split())
b = list(sorted((h, idx) for idx, h
        in enumerate(a)))
ans = b[::2] + b[1::2][::-1]
ans = list(idx for _, idx in ans)
print(*ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;
int main() {
  int n;
```

```
  cin >> n;
  vector<pair<int, int>> a(n);
  for (int i = 0; i < n; ++i) {
    cin >> a[i].first;
    a[i].second = i;
  }
  sort(a.begin(), a.end());
  for (int i = 0; i < n; i += 2)
    cout << a[i].second << ' ';
  for (int i = n - 1 - n % 2;
       i >= 1; i -= 2)
    cout << a[i].second << ' ';
}
```

6.9. Пример решения задачи. Одномерный геометрический центр

Задача. Даны n точек на прямой; i-я точка имеет координату x_i. Требуется найти их геометрический центр, т. е. такую точку c, которая минимизирует следующую величину:

$$\sum_{i=1}^{n} |c - x_i|.$$

Входные данные состоят из натурального числа n в первой строке и разделенных пробелами целых чисел x_i во второй. Гарантируется, что $n \le 10^5$, $-10^9 \le x_i \le 10^9$. Вывести требуется единственное число c — координату искомого центра; если точек, дающих минимальную сумму расстояний, несколько, то разрешается вывести любую.

Примеры

Входные данные	**Требуемый результат**
4 1 4 3 2	2
Входные данные	**Требуемый результат**
2 -10 10	0

Решение

В задачах, подобной этой, всегда есть смысл разобрать вручную несколько тестовых примеров, чтобы попытаться заметить закономерности и идеи для решения задачи.

Понятно, что когда $n = 1$, то ответ совпадает со входной точкой, т. е. $c = x_1$, поскольку это единственное решение, дающее нулевое расстояние.

Случай, когда $n = 2$ и $x_1 \neq x_2$, — уже более интересен; не теряя общности, положим $x_1 < x_2$. Нарисуем для наглядности несколько потенциально возможных позиций c (рис. 6.1).

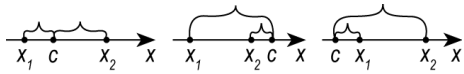

Рис. 6.1. Позиция с двумя точками и тремя положениями c

Как можно заметить по рис. 6.1, оптимальный ответ достигается, когда $x_1 \leq c \leq x_2$, и этот ответ более не зависит от конкретного положения внутри этого отрезка. (При желании это можно доказать и формально, рассмотрев все три возможных способа раскрыть оба модуля в сумме $|x_1 - c| + |x_2 - c|$.) То же самое верно, когда $x_1 = x_2$.

Случай $n = 3$ решить таким наглядным способом уже сложно, однако можно попытаться свести его к рассмотренным ранее. В самом деле, пусть, не теряя общности, $x_1 \leq x_2 \leq x_3$, тогда сумма для крайних точек x_1 и x_3 имеет, как описано выше, оптимальное решение $c \in [x_1; x_3]$, а задача для средней точки x_2 имеет оптимальное решение $c = x_2$. Поскольку $x_2 \in [x_1; x_3]$, то получается, что оптимальные решения этих двух подзадач имеют общую точку $c = x_2$, что и является ответом для случая $n = 3$.

Случай $n = 4$ сводится аналогичным образом к случаю $n = 2$ для двух средних точек.

Продолжая эти рассуждения на бесконечность, получаем следующее решение: ответом является точка-медиана входных точек; а если n четное, то ответом является любая точка между двух медиан.

Более того, мы «случайно» заодно получили и доказательство этого решения по индукции, если мы положим случаи $n = 1$ и $n = 2$ базой индукции и посмотрим, что происходит с ответом при добавлении двух точек — одной с краю слева и одной с краю справа.

Реализация

Искать медиану можно различными способами; самый простой и в то же время приемлемо быстрый — отсортировать входной массив и взять его средний элемент. Хотя есть и более быстрые алгоритмы поиска медиан, разницы при решении большинства задач спортивного программирования нет.

Python

```
n = int(input())
x = map(int, input().split())
```

C++

```
#include <algorithm>
#include <iostream>
```

```
ans = list(sorted(x))[n // 2]
print(ans)
```

```
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> x(n);
  for (int i = 0; i < n; ++i)
    cin >> x[i];
  sort(x.begin(), x.end());
  int ans = x[n / 2];
  cout << ans << endl;
}
```

Глава 7.
Разминочные реализационные задачи

Задачами на реализацию в спортивном программировании называют такие задачи, для решения которых не требуется ничего изобретать, а надо лишь правильно реализовать описанную в условии последовательность действий. Фокус здесь делается на том, как аккуратно, без ошибок и как можно быстрее запрограммировать требуемые операции.

Упрощая, можно сказать, что реализационные задачи — противоположный полюс относительно конструктивных задач.

7.1. Пример решения задачи. Морской бой

Задача. Дано расположение кораблей для игры в «Морской бой». В этой игре корабли расставляются на клетчатом поле некоторого размера, и каждый корабль представляет собой горизонтальный или вертикальный отрезок последовательно расположенных клеток. Требуется определить, корректно ли расставлены корабли, т. е. что корабли не пересекаются и никакие два корабля не соприкасаются по ребру или углу. Ограничений на количество кораблей или их размеры в данной задаче нет (рис. 7.1).

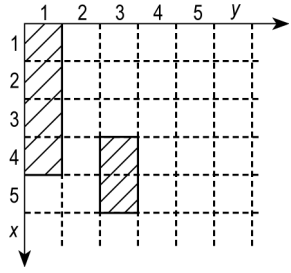

Рис. 7.1. Примеры из условий

В первой строке входного набора дается число кораблей (это натуральное число, не превосходящее 100). В каждой из последующих строк содержится описание одного корабля: координаты (x_1, y_1, x_2, y_2) клеток — его концов. Все координаты — натуральные числа, не превосходящие 100. Вывести требуется «OK», если расстановка кораблей корректна, или «BAD» в противном случае.

Примеры

Входные данные	Требуемый результат
2 1 1 4 1 5 2 5 2	BAD
Входные данные	**Требуемый результат**
2 1 1 4 1 5 3 4 3	OK

Решение

Реализовать требуемую проверку можно различными способами. Один из способов, простых для реализации, следующий.

Будем «рисовать» поле в памяти компьютера, для чего заведем двумерный массив, изначально заполненный нулями. Затем будем по одному добавлять на поле корабли, заполняя клетки каждого корабля числами 1. Наконец, для обнаружения соприкосновений будем «обводить» каждый корабль, т. е. помечать клетки, касающиеся его, числами 2. Тогда критерием корректности будет следующая проверка: ни одна клетка, уже помеченная как «1», не должна быть помечена еще раз.

В самом деле, попытка поставить «1» поверх существующей «1» означает, что два корабля пересекаются в этой клетке. А попытка поставить «2» поверх существующей «1» означает, что добавляемый корабль касается одного из уже добавленных. Заметим, что многократные пометки «2» в одной и той же клетке не являются ошибкой, поскольку они лишь говорят о том, что есть два корабля, расстояние между которыми составляет одну пустую клетку.

Реализация

Python

```
import sys
FIELD = 102
a = [[0] * FIELD] * FIELD
n = int(input())
for _ in range(n):
  x1, y1, x2, y2 = map(
    int, input().split())
  if x1 > x2:
    x1, x2 = x2, x1
  if y1 > y2:
    y1, y2 = y2, y1
  for x in range(x1 - 1, x2 + 2):
```

C++

```
#include <iostream>
using namespace std;
const int FIELD = 102;
int main() {
  int n;
  cin >> n;
  int a[FIELD][FIELD] = {0};
  for (int i = 0; i < n; ++i) {
    int x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;
    for (int x = x1 - 1;
         x <= x2 + 1; ++x) {
```

```
    for y in range(y1 - 1, y2 + 2):
      if a[x][y] == 1:
        print("BAD")
        sys.exit(0)
      a[x][y] = 2
  for x in range(x1, x2 + 1):
    for y in range(y1, y2 + 1):
      a[x][y] = 1
print("OK")
```

```
      for (int y = y1 - 1;
           y <= y2 + 1; ++y) {
        if (a[x][y] == 1) {
          cout << "BAD" << endl;
          return 0;
        }
        a[x][y] = 2;
      }
    }
    for (int x = x1;
         x <= x2; ++x) {
      for (int y = y1;
           y <= y2; ++y)
        a[x][y] = 1;
    }
  }
  cout << "OK" << endl;
}
```

Примечание о константе 102

Эта константа — размер поля, рисуемого программой в памяти. Она получается из входных ограничений «100» на координаты после учета того, что рисование обводки добавляет по одной клетке в каждом направлении. Хотя можно было бы ограничиться полем размера 100×100, это потребовало бы реализации проверок на выход за пределы поля. Стандартный прием в спортивном программировании — избежать усложнения кода путем добавления «запаса» к выделяемым массивам.

Примечание о синтаксисе = {0} в C++

Данный синтаксис — краткая форма записи для заполнения массива нулями. Ее удобно применять в спортивном программировании благодаря ее лаконичности. Следует помнить, однако, что эта форма работает **только с нулем**: если написать `= {1}`, то единицей будет проинициализирован только самый первый элемент массива, а все последующие будут проинициализированы значением по умолчанию, т. е. нулем!

7.2. Пример решения задачи. Стоимость интернет-связи

Задача. Интернет-провайдер ввел тариф с переменной ценой. Например, в некоторые дни недели цена одного мегабайта интернет-трафика ночью ниже, чем днем. Тариф полностью описывается списком записей вида «день недели, час, цена», обозначающих, что указанная цена действует с начала указанного часа и вплоть до следующей записи. Тариф, действующий на момент окончания воскресенья, по умолчанию продолжает действовать в утро понедельника. Требуется посчитать стоимость сеанса интернет-связи, описанного списком записей «день недели, час, число мегабайт».

Первая строка входного файла содержит натуральное число n — число записей в описании тарифа. Следующие n строк содержат записи описания тарифа, каждая состоящая из названия дня недели («Mon», «Tue», «Wed», «Thu», «Fri», «Sat» или «Sun»), часа (целого числа от 0 до 23) и цены (неотрицательного целого числа, не превосходящего 1000), разделенных пробелами. Характеристики тарифа даны в хронологическом порядке, т. е. в порядке увеличения дней недели или в порядке увеличения номера часа внутри одного дня. Гарантируется, что никакие две записи не совпадают по времени. В следующей строке записано число k — число записей в описании сеанса связи. Следующие k строк содержат записи описания сеанса связи, каждая состоящая из дня недели, часа и числа мегабайт, разделенных пробелами. Число мегабайт в каждой записи — натуральное, не превосходящее 1000. Описания сеанса связи могут иметь совпадающие день и час и заданы в произвольном порядке. Вывести требуется единственное число — стоимость сеанса связи.

Примеры

Входные данные	**Требуемый результат**
3 Mon 0 10 Mon 21 5 Tue 0 10 2 Mon 20 1 Mon 21 1	15
Входные данные	**Требуемый результат**
1 Wed 12 10 1 Mon 0 1	10

Решение

Можно выделить два возможных подхода к решению этой задачи. Первый вариант — рассматривать каждую запись сеанса связи по отдельности и определять тариф, действующий во время этой записи. Второй вариант — посчитать заранее для каждого времени суток каждого дня его цену Интернета и затем просуммировать эти величины по всем записям сеанса связи.

Первый вариант представляется чуть более простым по реализации. Второй вариант будет работать быстрее при большом числе записей, однако при заданных ограничениях это не является критическим фактором. Поэтому приведем ниже реализацию первого варианта.

Единственная тонкость в любом из решений — не забыть учесть замкнутость тарифа, т. е. то, что тариф в начале суток понедельника равен тарифу в конце суток вос-

кресенья. Простой способ решить эту проблему — добавить фиктивную первую запись в ноль часов понедельника.

Реализация

Python

```
DAYS = ["Mon", "Tue", "Wed", "Thu",
        "Fri", "Sat", "Sun"]

n = int(input())
tariff = [None] * (n + 1)
for i in range(n):
  day, hour, price = input().split()
  day = DAYS.index(day)
  hour = int(hour)
  price = int(price)
  tariff[i + 1] = (day, hour,
                   price)
tariff[0] = (0, 0, tariff[-1][2])

k = int(input())
cost = 0
for i in range(k):
  day, hour, mb = input().split()
  day = DAYS.index(day)
  hour = int(hour)
  mb = int(mb)
  for j in range(n + 1):
    if tariff[j][0] > day:
      break
    if (tariff[j][0] == day and
        tariff[j][1] > hour):
      break
  cost += mb * tariff[j - 1][2]
print(cost)
```

C++

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const string DAYS[7] = {
  "Mon", "Tue", "Wed", "Thu",
  "Fri", "Sat", "Sun"};
int day_number(string s) {
  return int(find(
    DAYS, DAYS + 7, s) - DAYS);
}
int main() {
  int n;
  cin >> n;
  vector<int> tday(n + 1),
              thour(n + 1),
              tprice(n + 1);
  for (int i = 1; i <= n; ++i) {
    string day_s;
    cin >> day_s >> thour[i]
        >> tprice[i];
    tday[i] = day_number(day_s);
  }
  tprice[0] = tprice[n];
  int k, cost = 0;
  cin >> k;
  for (int i = 0; i < k; ++i) {
    string day_s;
    int hour, mb;
    cin >> day_s >> hour >> mb;
    int day = day_number(day_s);
    int j = 0;
    while (tday[j + 1] < day ||
           tday[j + 1] == day &&
           thour[j + 1] <= hour)
      ++j;
    cost += mb * tprice[j];
  }
  cout << cost << endl;
}
```

7.3. Пример решения задачи. Пересечение двух прямоугольников

Задача. На стене висит экран для проектора. Известно, что проектор был настроен до установки экрана, поэтому освещаемая им область может не совпадать с экраном. И экран, и освещенная область представляют собой прямоугольники со сторонами, параллельными осям координат. Требуется вычислить площадь освещенной части экрана.

Оба прямоугольника задаются координатами двух точек — противоположных углов прямоугольника. Координаты записаны через пробел в порядке x_1, y_1, x_2, y_2 и являются целыми числами, не превосходящими 1000 по модулю. Оба входных прямоугольника имеют ненулевую площадь.

Примеры

Входные данные	**Требуемый результат**
`0 0 10 10` `5 15 15 6`	`20`
Входные данные	**Требуемый результат**
`0 0 1 1` `1 1 2 2`	`0`

Решение

Эту задачу можно решать различными способами. Приведем здесь распространенный прием решения подобных задач.

Первым делом нормализуем входные данные, чтобы первая точка указывала на левый нижний угол (т. е. угол с минимальными абсциссой и ординатой), а вторая — на правый верхний. Для этого сравним и при необходимости обменяем местами абсциссы и ординаты входных точек.

Далее вычислим две точки: точку с координатами, равными максимумам из координат левых нижних углов, и точку с координатами, равными минимумам из координат правых верхних углов. Обозначим эти две точки через (r_{x1}, r_{y1}) и (r_{x2}, r_{y2}) соответственно.

Можно заметить, что если входные прямоугольники пересекаются, эти две точки (r_{x1}, r_{y1}) и (r_{x2}, r_{y2}) как раз задают координаты углов общей части. Если же входные прямоугольники не пересекаются, то либо $r_{x1} > r_{x2}$, либо $r_{y1} > r_{y2}$, либо и то и другое одновременно (рис. 7.2).

Для доказательства можно отметить, что задача пересечения прямоугольников распадается на две одномерные подзадачи: одна задача в проекции на ось абсцисс, другая — на ось ординат. При решении одномерной задачи подход «взять максимум левых концов и минимум правых концов» верен, поскольку общая часть двух

отрезков не может начинаться левее ни одного из левых концов и не может заканчиваться правее ни одного из правых концов. Таким образом, и решение двумерной задачи также верно.

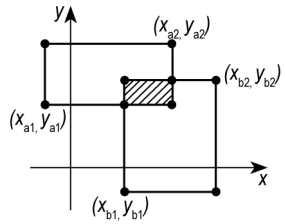

Рис. 7.2. Общая часть двух прямоугольников

Реализация

Python

```
def read_rect():
  x1, y1, x2, y2 = map(
    int, input().split())
  return (min(x1, x2), min(y1, y2),
          max(x1, x2), max(y1, y2))

ax1, ay1, ax2, ay2 = read_rect()
bx1, by1, bx2, by2 = read_rect()
rx1 = max(ax1, bx1)
ry1 = max(ay1, by1)
rx2 = min(ax2, bx2)
ry2 = min(ay2, by2)
s = (max(0, rx2 - rx1) *
     max(0, ry2 - ry1))
print(s)
```

C++

```
#include <algorithm>
#include <iostream>
#include <utility>
using namespace std;
void read_rect(int& x1, int& y1,
               int& x2, int& y2) {
  cin >> x1 >> y1 >> x2 >> y2;
  if (x1 > x2)
    swap(x1, x2);
  if (y1 > y2)
    swap(y1, y2);
}
int main() {
  int ax1, ay1, ax2, ay2;
  read_rect(ax1, ay1, ax2, ay2);
  int bx1, by1, bx2, by2;
  read_rect(bx1, by1, bx2, by2);
  int rx1 = max(ax1, bx1);
  int ry1 = max(ay1, by1);
  int rx2 = min(ax2, bx2);
  int ry2 = min(ay2, by2);
  int s = (max(0, rx2 - rx1) *
           max(0, ry2 - ry1));
  cout << s << endl;
}
```

Примечание к конструкции «max(0, …)»

Как можно заметить, при вычислении итоговой площади мы заменили явные сравнения координат выражением $\max(0, x)$. Результата эта замена не изменяет: нулевой ответ по-прежнему будет получаться в случае, когда уменьшаемое меньше вычитаемого.

Данный прием — это стандартный способ упростить обработку особых случаев и сделать запись более компактной; чем сложнее выражение, тем больше оправдывает себя такая лаконичная форма.

Примечание о вспомогательных типах данных

Приведенные реализации не вводят структур данных наподобие «прямоугольник» или «точка», а вместо этого оперируют отдельными числовыми переменными. Разумеется, ввести подходящие типы данных было бы возможно (и даже нужно, если речь шла бы о промышленном программировании), однако в спортивном программировании, да еще и в настолько короткой реализации, предпочтение зачастую отдается лаконичным конструкциям, не обремененным абстракциями.

7.4. Пример решения задачи. Проверка скобочной последовательности

Задача. Дана строка, состоящая только из круглых скобок, т. е. «(» и «)». Строка не пуста и не превосходит по длине 10^5. Требуется проверить ее на правильность, т. е. могла ли она получиться из корректного арифметического выражения после удаления из него цифр и знаков операций. Ответ на задачу требуется вывести в виде строки «YES» или строки «NO».

Формально все правильные скобочные последовательности образуются следующим образом: это либо пустая строка, либо строка вида «(X)», либо строка вида «XY», где X и Y — правильные скобочные последовательности.

Примеры

Входные данные	**Требуемый результат**
(()())()	YES
Входные данные	**Требуемый результат**
)(	NO
Входные данные	**Требуемый результат**
(()	NO

Решение

Будем двигаться по строке слева направо и поддерживать текущий баланс, т. е. отслеживать число открывающих скобок, для которых пока не было найдено пары. Если очередной символ строки — открывающая скобка, то баланс нужно увеличить на единицу; если же текущий символ — закрывающая скобка, то баланс нужно уменьшить. Тогда после обработки всей строки мы можем определить ответ: строка правильна тогда и только тогда, когда итоговый баланс равен нулю и ни разу по ходу обработки строки не становился отрицательным.

Проиллюстрируем работу алгоритма на примере из условия — «(()())()» (рис. 7.3)

```
(          :1
((         :2
(()        :1
(()(       :2
(()()      :1
(()())     :0
(()())(    :1
(()())()   :0
```

Рис. 7.3. Пошаговая обработка тестового примера

Пояснение

Корректность этого решения можно доказать строго, однако ограничимся здесь интуитивным рассуждением. По сути, алгоритм неявно пытается найти для каждой закрывающей скобки соответствующую ей открывающую. Когда мы стоим на символе закрывающей скобки и текущий баланс равен нулю, то ни одной подходящей открывающей скобки нет и ответ — «NO». Когда же баланс положителен, то правильно взять в пару последнюю из незакрытых открывающих скобок (рис. 7.4). После обработки всей строки сравнение баланса с нулем проверяет, что не осталось «лишних» открывающих скобок.

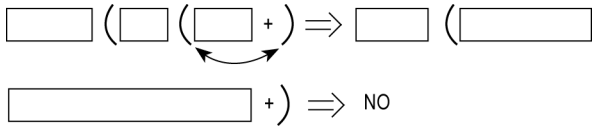

Рис. 7.4. Обработка закрывающей скобки с балансом, равным 4

Реализация

Python

```
def valid(s):
  balance = 0
  for c in s:
    if c == '(':
      balance += 1
    else:
      balance -= 1
      if balance < 0:
        return False
  return balance == 0

s = input()
print('YES' if valid(s) else 'NO')
```

C++

```
#include <iostream>
#include <string>
using namespace std;
bool Valid(const string& s) {
  int balance = 0;
  for (auto c : s) {
    if (c == '(') {
      ++balance;
    } else {
      --balance;
      if (balance < 0)
        return false;
    }
  }
  return balance == 0;
```

```
}
int main() {
  string s;
  cin >> s;
  cout << (valid(s) ? "YES" : "NO")
       << endl;
}
```

7.5. Пример решения задачи. Проверка скобочной последовательности с двумя типами

Задача. Дана непустая строка длины до 10^5, состоящая только из символов «(», «)», «[», «]». Требуется вывести «YES» или «NO» в зависимости от того, правильная это скобочная последовательность или нет.

Примеры

Входные данные	**Требуемый результат**
([]())[([])]	YES
Входные данные	**Требуемый результат**
()	NO
Входные данные	**Требуемый результат**
[](	NO

Решение

Как и при решении предыдущей задачи, будем двигаться по строке слева направо и поддерживать состояние, описывающее незакрытые на данный момент скобки. Только теперь мы будем поддерживать не числовой баланс, а **стек**, содержащий символы, — не закрытые пока скобки.

Когда текущий символ — открывающая скобка, то будем добавлять ее в стек. Когда текущий символ — закрывающая скобка, то будем проверять, на самом ли деле на вершине стека лежит соответствующая открывающая скобка. Если вместо этого там будет скобка другого типа либо же стек будет пустым, то скобочная последовательность некорректна. По завершении обработки строки останется только убедиться, что стек пуст.

Реализация

Python

```
def valid(s):
  stack = []
  for c in s:
```

C++

```
#include <iostream>
#include <string>
using namespace std;
```

```
    if c in ('(', '['):
      stack.append(c)
      continue
    if not stack:
      return False
    need = '(' if c == ')' else '['
    if stack.pop() != need:
      return False
  return not stack

s = input()
print('YES' if valid(s) else 'NO')
```

```
bool Valid(const string& s) {
  string stack;
  for (auto c : s) {
    if (c == '(' || c == '[') {
      stack += c;
      continue;
    }
    if (stack.empty())
      return false;
    char need = (c == ')') ?
      '(' : '[';
    if (stack.back() != need)
      return false;
    stack.pop_back();
  }
  return stack.empty();
}
int main() {
  string s;
  cin >> s;
  cout << (valid(s) ? "YES" : "NO")
       << endl;
}
```

Примечание к типу данных «стек»

В приведенных выше реализациях мы не использовали структуру данных «стек» из стандартных библиотек, а использовали обычный список в Python и обычную строку в C++. Отчасти это связано с тем, что интерфейс стека является подмножеством интерфейсов этих типов данных; также это продиктовано соображениями лаконичности. Кроме того, в языке C++ тип std::stack по умолчанию основан на std::deque, который влечет неоправданные накладные расходы. Впрочем, при заданных в этой задаче ограничениях это уже становится вопросом предпочтений.

7.6. Пример решения задачи. Зеркальный лабиринт

Задача. Дано описание лабиринта размером $n \times m$ клеток. Каждая клетка (i, j) $(i = 1...n, j = 1...m)$ описывается одним из трех символов:

- символ «.» (точка) обозначает пустую клетку;
- символ «/» (слеш) обозначает зеркало, повернутое по часовой стрелке на 45 градусов;
- символ «\» (обратный слеш) обозначает зеркало, повернутое против часовой стрелки на 45 градусов.

Задача — смоделировать траекторию лазерного луча в этом двумерном поле, считая, что луч идеально отражается от встречаемых по пути зеркал без рассеяния, ис-

кривлений и потерь энергии. Например, направленный слева направо луч при встрече с зеркалом типа «/» отразится и направится наверх. Изначально луч попадает в лабиринт в клетку (1, 1) с левой стороны в середине ребра (рис. 7.5).

Входные данные состоят из чисел n и m ($1 \le n, m \le 50$), записанных в первой строке, за которыми следует описание лабиринта: n строк по m символов в каждой. Требуется вывести два числа, разделенных пробелом: координаты последней клетки, в которой луч побывает, перед тем как покинуть пределы лабиринта.

Примеры

Входные данные	**Требуемый результат**
2 2 \. \/	1 2
Входные данные	**Требуемый результат**
2 3 ..\ ///	2 2

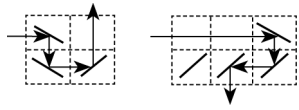

Рис. 7.5. Примеры

Решение

Это чисто реализационная задача; есть много различных способов решить ее. Здесь мы опишем один из подходов, который основан на нумерации четырех возможных направлений движения и их кодировании в виде «дельт», — это прием, который пригождается во многих подобных задачах.

Итак, выпишем четыре возможных направления движения. Пронумеруем их в порядке, например, обхода по часовой стрелке. Для каждого направления укажем «дельты», т. е. то, как меняются координаты при соответствующем движении:

- №0: вправо ($\partial_x = 0$, $\partial_y = 1$);
- №1: вниз ($\partial_x = 1$, $\partial_y = 0$);
- №2: влево ($\partial_x = 0$, $\partial_y = -1$);
- №3: вверх ($\partial_x = -1$, $\partial_y = 0$).

Теперь будем моделировать траекторию луча, двигаясь по полю клетка за клеткой. Когда текущая клетка пуста, то для совершения одного шага достаточно прибавить к текущим координатам ∂_x и ∂_y.

Более интересный случай — когда текущая клетка содержит зеркало. Тогда перед совершением шага в следующую клетку требуется изменить направление. Вместо того чтобы кодировать вручную восемь различных случаев — рискуя допустить ошибку, не говоря уже о том, насколько это долгая работа, — попытаемся вывести простую формулу с использованием операции XOR («исключающее ИЛИ»). Если мы обозначим через d текущее направление (от нуля до трех), через d' — направление после отражения и через $\oplus$ — операцию XOR, то справедливо:

- зеркало «/»: $d' = d \oplus 3$;
- зеркало «\»: $d' = d \oplus 1$.

> ***Как вывести подобную формулу***
>
> Готового «рецепта» не существует. В данном случае, когда значений всего четыре и каждое зеркало наклонено под углом ровно 45 градусов по отношению к лучу, интуиция подсказывает, что наверняка должна быть простая формула. Причем формулы вида «прибавить константу» или «отнять от константы» явно не годятся. Быстрая проверка операции XOR на направлении «вправо» дает константы 3 и 1, и проверка на остальных направлениях подтверждает, что такая формула работает.

Таким образом, мы получили достаточно простые правила для пошагового моделирования движения луча в лабиринте.

Реализация

Python

```
DELTA = ((0, 1), (1, 0), (0, -1),
  (-1, 0))
n, m = map(int, input().split())
a = [input() for _ in range(n)]

def ray():
  x, y = 0, 0
  d = 0
  while True:
    if a[x][y] == '/':
      d ^= 3
    elif a[x][y] == '\\':
      d ^= 1
    nx = x + DELTA[d][0]
    ny = y + DELTA[d][1]
    if (nx < 0 or ny < 0 or
        nx >= n or ny >= m):
      return x, y
    x, y = nx, ny

x, y = ray()
print(x + 1, y + 1)
```

C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const int DELTA[4][2] = {
  {0, 1}, {1, 0}, {0, -1}, {-1, 0}
};
int main() {
  int n, m;
  cin >> n >> m;
  vector<string> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  int x = 0, y = 0, d = 0;
  for (;;) {
    if (a[x][y] == '/')
      d ^= 3;
    else if (a[x][y] == '\\')
      d ^= 1;
    int nx = x + DELTA[d][0];
    int ny = y + DELTA[d][1];
    if (nx < 0 || ny < 0 ||
```

```
        nx >= n || ny >= m) {
      cout << x + 1 << ' ' << y + 1
           << endl;
      return 0;
    }
    x = nx;
    y = ny;
  }
}
```

7.6. Пример решения задачи. Троичная сбалансированная система счисления

Определение. Троичной сбалансированной системой счисления называется система с основанием 3 и цифрами «0», «+» (обозначает единицу), «−» (обозначает минус единицу). Примеры записи различных чисел в этой системе счисления:

$$0_{10} = 0_{bal3},$$
$$1_{10} = +_{bal3},$$
$$2_{10} = +-_{bal3},$$
$$3_{10} = +0_{bal3},$$
$$4_{10} = ++_{bal3},$$
$$5_{10} = +--_{bal3}.$$

Свойства и применения

Троичная сбалансированная система счисления использовалась в некоторых ранних компьютерах, например, в ЭВМ «Сетунь». Эта система обладает интересными свойствами, например, естественностью представления отрицательных чисел и большей компактностью по сравнению с двоичной системой. Округление в этой системе счисления эквивалентно отбрасыванию лишних разрядов. Эта система также пригождается при решении некоторых задач о гирях и взвешиваниях. Впрочем, в современном мире эта система счисления почти не используется на практике.

Задача. Дано целое неотрицательное число n, записанное в десятичной системе счисления. Требуется перевести его в троичную сбалансированную систему счисления.

Входное число не превосходит 10^9. Выводить результат требуется с помощью символов «0», «+» и «−» без лишних лидирующих нулей.

Примеры

Входные данные	**Требуемый результат**
5	+--
Входные данные	**Требуемый результат**
0	0

Решение

Эту задачу можно решать различными способами.

Один из возможных вариантов — заметить, что эта система счисления во многом схожа с обычной троичной системой, однако с запретом цифры 2. Неформально можно сказать, что цифра 2 заменена в этой системе комбинацией «+ −», т. е. числом 3, уменьшенным на единицу.

Тогда можно строить ответ постепенно, начиная с младших разрядов, следующим образом. Поделим входное число на 3 и посчитаем остаток от деления. Если он равен 0 или 1, то просто припишем соответствующую цифру к ответу. Если же остаток оказался равен 2, то припишем к ответу «−», а затем увеличим текущее число на единицу. Будем повторять эту процедуру до тех пор, пока текущее число не станет равным нулю.

Альтернативное описание

Более изящно описанную выше процедуру можно сформулировать следующим образом: переведем входное число в обычную троичную систему счисления, затем прибавим к нему бесконечное число «…111» в троичной системе счисления и после отнимем от каждого разряда по единице.

Реализация

Python

```
n = int(input())
ans = "" if n else "0"
while n:
  cur = n % 3
  n //= 3
  ans = "0+-"[cur] + ans
  if cur == 2:
    n += 1
print(ans)
```

C++

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  int n;
  cin >> n;
  string ans = n ? "" : "0";
  while (n) {
    int cur = n % 3;
    n /= 3;
    ans = "0+-"[cur] + ans;
    if (cur == 2)
      ++n;
  }
  cout << ans << endl;
}
```

Глава 8.
Задачи для самостоятельного решения

Для закрепления пройденного материала настоятельно рекомендуем читателю самостоятельно прорешать задачи этой главы.

8.1. Примеры задач

Подсказки к приведенным ниже задачам даны в конце книги.

8.1.1. Задача. Фибоначчилистник

Растение Фибоначчилистник растет следующим образом. В первый день из семечки вырастает ветвь. На второй день на ветви появляется лист и почка. На третий и последующие дни появляется по одному новому листу, а из почки вырастает по одной новой ветви; эти новые ветви развиваются аналогичным образом. Требуется вывести общее число листьев на момент окончания n-го дня ($1 \le n \le 50$).

Примеры

Входные данные	**Требуемый результат**
4	4
Входные данные	**Требуемый результат**
1	0

Пояснения к примерам

В первом примере в конце четвертого дня имеется 3 ветви, на первой из которых 2 листа, на второй — 1, на третьей — ни одного.

8.1.2. Задача. Картина на кафеле

На стену, выложенную кафельной плиткой, повесили картину. Требуется посчитать, сколько плиток полностью покрыты картиной и сколько — частично. Известно, что все плитки имеют равный размер, ориентированы одинаковым образом и каждая плитка выложена целиком и без наложений с другими плитками. Начало координат находится в левом нижнем углу стены.

В первой строке входных данных содержатся два целых числа — размеры плитки (по осям абсцисс и ординат соответственно). Во второй и третьей строках даны координаты двух противоположных углов картины (в виде пары абсциссы и ордина-

ты); гарантируется, что обе размерности картины ненулевые. Все числа во входном наборе — неотрицательные целые, не превосходящие 10^3.

Вывести требуется два числа, разделенных пробелом: количество полностью покрытых плиток и количество частично покрытых плиток.

Примеры

Входные данные	**Требуемый результат**
3 5 0 0 10 10	6 2
Входные данные	**Требуемый результат**
50 50 40 50 50 40	0 1

8.1.3. Задача. Лексикографически минимальное исправление перестановки.

Как и в задаче «Исправление перестановки» *(см. разд. 6.4)*, требуется превратить заданную последовательность чисел в перестановку. Отличие этой задачи в том, что помимо минимизации числа операций требуется также найти лексикографически минимальную перестановку, возможную при этом числе операций.

Примеры

Входные данные	**Требуемый результат**
4 1 100 3 3	2 1 2 3 4
Входные данные	**Требуемый результат**
3 3 2 1	0 3 2 1

Пояснение к примерам

Хотя в первом примере двумя операциями можно получить несколько различных перестановок («1 2 3 4», «1 2 4 3», «1 4 2 3», «1 4 3 2»), вывести требуется лексикографически наименьшую из них — коей является «1 2 3 4».

8.1.4. Задача. XOR нечетных чисел от 1 до *n*

Дано нечетное целое число n, не превосходящее 10^{18}. Требуется вывести XOR-сумму нечетных чисел от 1 до n, т. е. следующую величину:

$$1 \oplus 3 \oplus \ldots \oplus (n-2) \oplus n.$$

Примеры

Входные данные	Требуемый результат
3	2
Входные данные	**Требуемый результат**
5	7

8.1.5. Задача. Манхэттенский центр

Даны координаты n точек, заданных своими координатами x_i и y_i. Это локации мест в Манхэттене, и таксист хочет найти такое расположение своей базы, которое бы минимизировало сумму расстояний от базы до всех этих точек.

В идеализированной модели Манхэттена, используемой в этой задаче, таксист может передвигаться только по линиям целочисленной решетки, параллельной координатным осям, и скорость его движения постоянна и не зависит от направления движения. Таксист может выбирать произвольные улицы, а повороты совершает мгновенно.

Входные данные состоят из натурального числа n в первой строке ($1 \le n \le 10^5$) и n последующих строк, содержащих по паре координат x_i и y_i, разделенных пробелом. Все координаты целые и по модулю не превосходят 10 000. Вывести требуется два разделенных пробелом целых числа — координаты искомой базы. Если оптимальных решений несколько, разрешается вывести любое.

Примеры

Входные данные	Требуемый результат
3 0 0 1 1 2 2	1 1
Входные данные	**Требуемый результат**
2 1 1 -1 1	0 1

8.1.6. Задача. Превращение в одинаковые числа

Дана последовательность a_i целых чисел длины n. Требуется сделать все числа в последовательности одинаковыми за наименьшее число операций, где операция заключается в увеличении или уменьшении одного из элементов на один.

Входные данные состоят из натурального числа $n \le 10^5$ в первой строке и последовательности из n целых чисел a_i ($|a_i| \le 10^9$) во второй. Вывести требуется единственное число — минимальное возможное число операций.

Примеры

Входные данные	Требуемый результат
5 1 2 3 4 5	6
Входные данные	**Требуемый результат**
1 123	0

Пояснение к примерам

В первом примере можно превратить все числа в число 3, дважды увеличив единицу, один раз двойку, один раз уменьшив четверку и дважды — пятерку; итого шесть операций. Во втором примере ни одной операции не требуется, поскольку входная последовательность — единственное число — уже находится в требуемом состоянии.

8.2. Задачи в онлайн-системах

- UVA, задача 100: «The 3*n* + 1 problem».
- Codeforces, раунд 30, задача A: «Бухгалтерский учет».
- Codeforces, раунд 31, задача B: «Сисадмин Вася».
- Codeforces, раунд 34, задача C: «Список».
- Codeforces, раунд 752, задача A: «Раз-делимое замешательство».
- Codeforces, раунд 588, задача A: «Марчин и сборы».
- Codeforces, раунд 752, задача B: «Мягкая модульная медиана».
- Codeforces, раунд 694, задача A: «Странный массив».

Итоги ступени I

На протяжении *ступени I* читатель получил опыт решения таких задач спортивного программирования, которые не требуют никаких специальных познаний.

Главный вывод — знание большого количества алгоритмов совершенно **не гарантирует** помощь в решении задач спортивного программирования на практике. Есть большой класс задач (кратко представленных в предыдущих главах), для успешного и быстрого решения которых не требуется ничего, кроме хорошего владения языком программирования и практического опыта.

Кроме того, на страницах *ступени I* читатель познакомился с такими важными понятиями, как:

- числа Фибоначчи;
- манхэттенская метрика;
- перестановки;
- палиндромы;
- правильные скобочные последовательности;
- операция XOR.

Все они будут встречаться читателю и далее — как на дальнейшем протяжении книги, так и при решении практических задач.

Ступень II.
Базовые алгоритмы

Глава 9.
Оценка скорости работы алгоритмов

Умение **приблизительно** оценивать скорость работы того или иного алгоритма — один из критически важных навыков в спортивном программировании. Как правило, каждое задание имеет несколько способов решения, отличающихся скоростью работы и сложностью реализации. Задача участника — найти по возможности самое простое для реализации решение, которое работает достаточно быстро при поставленных входных ограничениях.

У этой задачи есть две стороны:

- теоретическая — какое число абстрактных операций выполняет алгоритм «на бумаге»;
- и эмпирическая — сколько операций успеет выполнить реальный компьютер за единицу времени.

9.1. Эмпирическая скорость процессора

9.1.1. Тактовая частота процессора

Многие знакомы с понятием «тактовая частота» — это один из основных параметров, на которые обращают внимание при покупке процессора. Например, самый продаваемый процессор 2020 г. имеет частоту 3,6 ГГц. Однако что именно обозначает это число?

На самом деле это низкоуровневый параметр, который выражает количество тактовых импульсов, поступающих на процессор и синхронизирующих его работу. В указанном выше примере процессор совершает более трех миллиардов тактов в секунду.

Однако тактовая частота не связана напрямую с числом команд, которые процессор может выполнить в единицу времени. С одной стороны, многие команды требуют нескольких тактов; с другой — современные процессоры умеют группировать и исполнять несколько инструкций одновременно (даже в пределах одного ядра). Более того, реальная скорость будет зависеть от многих других факторов, таких, как:

- производительность остальных компонентов компьютера (в первую очередь, материнской платы и оперативной памяти);
- расположение данных в памяти (для успешного кэширования и достижения локальности данных);
- история предыдущих исполнений (что влияет на предсказания переходов);

- зависимость по данным между инструкциями программы (что влияет на возможность их внеочередного исполнения);
- версия и параметры компилятора или интерпретатора;
- тип и настройки операционной системы.

Поэтому, вообще говоря, **невозможно точно предсказать на основе тактовой частоты скорость выполнения операций**. А процессор с более высокой тактовой частотой не обязательно окажется быстрее при исполнении конкретной программы.

9.1.2. Многоядерные процессоры

Абсолютное большинство современных ПК построены на основе многоядерных процессоров. Такие процессоры физически позволяют исполнять несколько программных потоков параллельно. Благодаря этому несколько программ могут работать одновременно с минимальными задержками, а специально написанные программы могут совершать вычисления в разы быстрее (при условии, что они реализуют распределение задач по нескольким потокам). В то же время написание многопоточных программ требует специальных навыков и сопряжено с риском совершения различных ошибок.

Однако в спортивном программировании многопоточность **практически не используется**.

- Большинство соревнований и архивов задач явно или неявно требуют, чтобы решения были однопоточными.
  - Даже если участник попытается отправить многопоточное решение, системы проверки, как правило, настроены таким образом, что исполняться оно будет на единственном ядре.
- Есть небольшое число соревнований, на которых таких ограничений нет. Например, на некоторых соревнованиях участник должен просто исполнить свою программу на собственном компьютере в заданных временны́х рамках.
  - Как правило, ограничения даже на таких соревнованиях подобраны достаточно лояльно, чтобы адекватные однопоточные решения укладывались во временны́е ограничения.
  - Если все-таки возникает желание получить преимущество за счет многоядерности (например, при попытке сдать задачу явно недостаточно быстрым решением), то обычно этого легче достичь путем запуска нескольких экземпляров программы-решения, каждый из которых обрабатывает свою порцию входных тестов. Таким образом, и здесь многопоточность крайне редко бывает необходимой.

9.1.3. Эмпирические показатели

Как было объяснено выше, точное предсказание того, сколько времени будет исполняться та или иная программа на конкретном компьютере, — практически не-

выполнимая задача. Однако можно начать с некоторых базовых ориентиров, а постепенно, с опытом, научиться «чувствовать» скорость и узкие места программ.

Примечание

Разумеется, приведенные ниже числа — лишь результат конкретных измерений на конкретных компьютерах, использованных автором книги. На других компьютерах, в том числе на тестирующих системах каких-либо соревнований, или в других условиях те же самые программы могут оказаться существенно быстрее или, наоборот, медленнее. Повторимся: эти числа даны лишь в качестве **начальных ориентиров**.

Операция	Язык программирования	Число выполнений в секунду
Сложение 32-битных чисел	C++	5–6 млрд
	Python (PyPy)	1,8–1,9 млрд
	Python (CPython)	0,02–0,03 млрд
Перемножение 32-битных чисел	C++	0,8–0,9 млрд
	Python (PyPy)	0,6 млрд
	Python (CPython)	0,03 млрд

Из этой таблицы видно, почему при использовании языка программирования Python его реализация PyPy пользуется гораздо большей популярностью в среде спортивного программирования, чем CPython. Более высокая скорость PyPy объясняется тем, что он содержит JIT-компилятор, преобразующий код на Python в машинный код во время его выполнения. По этой же причине на протяжении всей книги при анализе практической скорости выполнения мы будем ссылаться только на PyPy.

9.2. Асимптотическая оценка. Основы

Асимптотическая оценка — это способ оценивать вычислительную сложность алгоритмов, который абстрагируется от деталей архитектуры компьютера, от особенностей конкретной реализации и от конкретных входных данных.

Ключевая идея этого подхода заключается в том, чтобы определить, с какой скоростью число операций, совершаемых конкретным алгоритмом, растет с увеличением входных данных. Этот подход не дает информации о том, **сколько именно** операций потребуется на конкретном входном наборе, зато он позволяет понять, какой алгоритм лучше остальных при **достаточно больших** входных данных.

9.2.1. Графическая иллюстрация метода асимптотических оценок

Проще всего понять суть этого метода с помощью наглядной визуализации.

Предположим, что мы хотим сравнить друг с другом два алгоритма. Допустим, мы знаем, что графики времени работы каждого из этих алгоритмов выглядят **приблизительно** следующим образом (рис. 9.1).

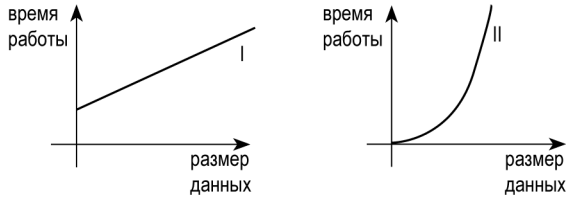

Рис. 9.1. Графики времени работы двух гипотетических алгоритмов в зависимости от величины входных данных: график первого алгоритма — прямая линия, а график второго — парабола

Тогда можно утверждать, что первый алгоритм **лучше** второго, по крайней мере, **при достаточно больших** входных данных. Это не зависит от точного вида графиков, от угла наклона прямой, коэффициента при параболе и т. п. — достаточно лишь знания **общего тренда** (рис. 9.2).

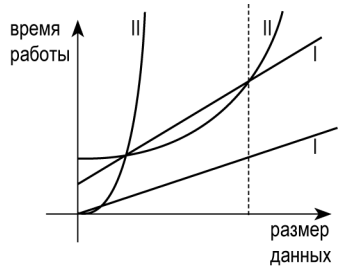

Рис. 9.2. Вне зависимости от вида прямой и параболы рано или поздно парабола обгонит прямую, и первый алгоритм станет работать быстрее второго

9.2.2. Асимптотическая оценка с одним параметром

Начнем с задач, сложность которых оценивается только одним входным параметром; обозначим его n. Тогда, например, какой-либо алгоритм может иметь асимптотическую сложность $O(n)$; какой-либо другой алгоритм может иметь сложность $O(n^2)$.

Что именно значат эти обозначения? Обозначение $O(n)$ говорит о том, что алгоритм работает за линейное время, т. е. совершает порядка n операций. Обозначение $O(n^2)$ значит, что алгоритм имеет квадратичное время работы. Приведем строгую формулировку.

Определение. Асимптотическая сложность алгоритма есть $O(f(n))$, где $f(n)$ — некоторая заданная функция, если существует такая положительная константа c, что при любых n алгоритм совершит не более $cf(n)$ операций.

Обозначение «О большое»

Это стандартное обозначение, широко используемое в различных областях математики. Эти обозначения читаются следующим образом: $O(n)$ — «о от эн», $O(1)$ — «о от единицы» и т. д. Иногда при этом уточняют «о большое», поскольку в математике также существует обозначение «о малое»; впрочем, в контексте асимптотической сложности алгоритмов последнее практически не употребляется.

Отличие от стандартного математического определения

Используемое в математике понятие «о большого», изначально введенное Бахманом в теории чисел, а затем распространившееся и по другим разделам математики, несколько отличается от приведенного выше. Стандартное математическое определение использует чуть более слабый критерий, поскольку помимо константы c оно также вводит константу n_0 и рассматривает неравенство на значения функции только при $n \geq n_0$. Несмотря на кажущуюся малозначительность этого отличия, известно, что к теории алгоритмов такое определение приложимо не очень хорошо, и особенно при работе с функциями нескольких переменных может приводить к неожиданным результатам (простой пример такого парадокса построил Хоуэлл в 2008 г.). Используемое в нашей книге определение такими недостатками не обладает (подробный сравнительный анализ разных определений был опубликован Рутаненом в 2016 г.).

Из определения вытекает несколько свойств:

- анализируется поведение алгоритма **в худшем случае**, т. е. для каждого n подразумеваются такие входные данные, для обработки которых алгоритму потребуется наибольшее число операций;
- оценка производится **с точностью до фиксированного множителя**. Например, $O(5n)$ эквивалентно $O(n)$. Благодаря этому мы абстрагируемся от таких деталей, как «процессор A выполняет операцию X быстрее процессора B» или «в языке программирования A операция X требует бóльших накладных расходов, чем в языке B»;
- акцент делается на **основном тренде** при стремлении n к бесконечности, а несущественные слагаемые могут быть отброшены. Например, $O(3n^2 + 2n + 1)$ — то же самое, что $O(n^2)$, поскольку достаточно выбора другой константы c, чтобы вторая функция была больше первой при всех значениях n;
- из двух предыдущих замечаний вытекает, что всегда разумно использовать самую **краткую** форму записи, как, например:
  - $O(n)$, а не $O(5n)$ или $O(n + 10)$;
  - $O(n^2)$, а не $O(3n^2 + 2n + 1)$;
  - $O(n \log n)$, а не $O(n \log n + n)$;
- асимптотическая оценка может быть неоптимальной; например, про алгоритм, на самом деле работающий за $O(n)$, можно также сказать, что он работает за $O(n^2)$ или $O(n^n)$. Разумеется, на практике стараются искать по возможности **оптимальные** оценки.

9.2.3. Примеры асимптотических оценок

Приведенные выше определения станут более понятными после рассмотрения конкретных примеров:

Программа на Python	**Программа на C++**	**Асимптотическая оценка**

`def sum(n):`
`  sum = 0`
`  for i in range(n):`
`  sum += i`
`  return sum`
	`int Sum(int n) {`
	`  int sum = 0;`
	`  for (int i = 0;`
	`  i < n; ++i) {`
	`  sum += i;`
	`  }`
	`  return sum;`
	`}`
		$O(n)$.
		Программа состоит из n итераций, каждая из которых совершает некоторое фиксированное число операций

`def sum2(n):`
`  return (n * (n + 1) /`
`  2)`
	`int Sum2(int n) {`
	`  return n * (n + 1) /`
	`  2;`
	`}`
		$O(1)$.
		Программа состоит из фиксированного числа операций, не зависящего от n

`def find(numbers, x):`
`  return numbers.index(`
`  x)`
	`int Find(`
	`  const vector<int>&`
	`  numbers,`
	`  int x) {`
	`  return find(`
	`  numbers.begin(),`
	`  numbers.end(),`
	`  x) -`
	`  numbers.begin();`
	`}`
		$O(n)$, где n — размер массива `numbers`.
		Скорость поиска значения в массиве зависит от того, насколько близко к началу массива находится искомый элемент. Однако в худшем случае придется просмотреть все элементы, поэтому оценка — линейная относительно числа элементов

`def calc(n):`
`  res = 0`
`  while n:`
`  n //= 2`
`  res += 1`
`  return res`
	`int Calc(int n) {`
	`  int res = 0;`
	`  for (; n; n /= 2)`
	`  ++res;`
	`  return res;`
	`}`
		$O(\log n)$.
		Цикл совершит $\lfloor \log_2 n \rfloor + 1$ итераций. Учитывая, что логарифмы по разным основаниям отличаются лишь константным множителем, в оценке сложности основание логарифма не указывают

`def pairs(n):`
`  res = 0`
`  for i in range(n):`
`  for j in range(i):`
`  res += 1`
`  return res`
	`int Pairs(int n) {`
	`  int res = 0;`
	`  for (int i = 0;`
	`  i < n; ++i) {`
	`  for (int j = 0;`
	`  j < i; ++j) {`
	`  ++res;`
	`  }`
	`  }`
	`}`
		$O(n^2)$.
		Несмотря на то что точное число итераций цикла равно $n \cdot (n - 1) / 2$, для асимптотической оценки важна только старшая степень — n^2

9.2.4. Асимптотическая оценка с несколькими параметрами

Данное в предыдущем разделе определение для асимптотических оценок с единственным параметром естественным образом обобщается на функции нескольких переменных. Например, о каком-либо алгоритме с двумя параметрами n и m можно говорить, что он работает за время $O(n \cdot m)$. Формально смысл этого понятия в следующем.

Определение. Асимптотическая сложность алгоритма есть $O(f(x_1, x_2, \ldots, x_k))$, где $f(x_1, x_2, \ldots, x_k)$ — некоторая заданная функция, если существует такая положительная константа c, что при любых допустимых наборах $(x_1, x_2, \ldots, x_k)$ алгоритм совершит не более $cf(x_1, x_2, \ldots, x_k)$ операций.

9.2.5. Примеры асимптотических оценок с несколькими переменными

Программа на Python	Программа на C++	Асимптотическая оценка

`def sum_matr(matr):`
`  sum = 0`
`  for row in matr:`
`  for elem in row:`
`  sum += elem`
`  return sum`
	`int SumMatr(`
	`  const`
	`  vector<vector<int>>&`
	`  matr) {`
	`  int sum = 0;`
	`  for (const auto& row:`
	`  matr) {`
	`  for (int elem :`
	`  row) {`
	`  sum += elem;`
	`  }`
	`  }`
	`  return sum;`
	`}`
		$O(n \cdot m)$, где n и m — размерности входной матрицы.
		Программа проходит по всем элементам матрицы, которых $n \cdot m$, совершая некоторое фиксированное число операций для каждого из них

`def diff(a, b):`
`  return (sum(a) -`
`  sum(b))`
	`int Diff(`
	`  const vector<int>& a,`
	`  const vector<int>& b)`
	`  {`
	`  return`
	`  accumulate(`
	`  a.begin(),`
	`  a.end())`
	`  - accumulate(`
	`  b.begin(),`
	`  b.end());`
	`}`
		$O(n + m)$, где n и m — размеры входных массивов.
		Программа состоит из двух блоков, один из которых работает за время $O(n)$, другой — $O(m)$

9.2.6. Асимптотическая оценка в среднем

В большинстве случаев интересны оценки времени работы в худшем случае, поскольку они позволяют выбирать алгоритмы, которые не имеют «слабых мест» и не работают чрезмерно долго ни на каких входных наборах.

Однако есть отдельные примеры алгоритмов, которые могут иметь относительно плохую производительность на конкретных входных наборах, и тем не менее в целом, в «среднестатистическом» случае, работать быстрее многих других алгоритмов. Здесь подразделяются два случая в зависимости от того, требуется ли учитывать вероятности различных входных наборов или нет.

- **Амортизационная** оценка верна всегда, независимо от распределения вероятностей входных наборов.

  Пример. Добавление элементов в массив

  Операция добавления элементов в конец массива является простым примером, когда имеет смысл использовать амортизационную оценку, потому что при правильной реализации добавление одного элемента будет требовать $O(1)$ в среднем, несмотря на то что в худшем случае сложность будет $O(n)$. Эффективная реализация заключается в том, что массив выделяется с некоторым запасом, и при каждом исчерпании этого запаса его новый размер удваивается (или, вообще говоря, растет со скоростью геометрической прогрессии). Таким образом, отдельные операции добавления могут быть «дорогими» за счет необходимости перевыделения памяти и копирования всех элементов, однако со временем эти дорогие операции становятся все реже. **В сумме** вставка n элементов потребует порядка $n + n / 2 + n / 4 + \ldots + 1$ операций, что можно оценить сверху как $2n$, что и приводит к амортизационной оценке $O(1)$ на одну операцию. Такие эффективные реализации используются во всех популярных версиях стандартных библиотек C++ и Python; отметим, что в Python аналогичный подход используется и для операции удаления из массива.

- **Оценка сложности в среднем**, напротив, полагается на знание о вероятностях возникновения конкретных входных наборов.

  Пример. Хеш-таблица

  Хеш-таблицы — это обширная тема, однако в основе их лежат вариации следующей идеи. Задачей хеш-таблицы является эффективная обработка добавления, удаления и поиска элементов, например, чисел. Хеш-таблица содержит только относительно небольшое число ячеек, и каждый элемент хранится в ячейке с номером, равным **хеш-функции** элемента (простейший, хотя и далеко не лучший, пример хеш-функции: взятие числа по модулю размера таблицы). При этом могут возникать **коллизии**, когда разные элементы имеют одинаковое значение хеш-функции, из-за чего в одной ячейке хеш-таблицы придется хранить сразу несколько элементов (например, в виде связанного списка). Производительность хеш-таблицы напрямую зависит от числа коллизий, поскольку при поиске элемента потребуется обойти все содержимое выбранной ячейки. Поэтому при анализе скорости работы в худшем случае — когда число коллизий очень высоко — производительность каждой операции плоха: лишь $O(n)$. Однако при условии если входные элементы являются случайными числами, вероятность коллизий низка и производительность хеш-таблиц получается очень высокой: $O(1)$ в среднем.

Как и любая другая тема в теории алгоритмов, амортизационный анализ и оценка сложности в среднем — это огромное поле для исследования. Однако в сфере спортивного программирования базового понимания основополагающих принципов, описанных выше, на наш взгляд, более чем достаточно.

9.2.7. Оценки потребления памяти

До сих пор мы фокусировались лишь на оценках времени работы алгоритмов, однако тот же самый метод асимптотических оценок применим и к оценкам требуемой памяти.

Особый интерес это представляет в тех случаях, когда есть выбор между несколькими алгоритмами с близкими оценками времени работы, однако различным потреблением памяти. Поскольку на практике операции чтения и записи в оперативной памяти работают существенно медленнее, чем операции с процессорными регистрами, алгоритм с лучшей оценкой потребления памяти будет предпочтительнее.

Рассмотрим несколько примеров:

Программа на Python	**Программа на C++**	**Асимптотическая оценка дополнительной памяти**

`def palindrome(string):`
`  return (string ==`
`  string[::-1])`
	`bool Palindrome(`
	`  const string& str) {`
	`  string rev = str;`
	`  reverse(rev.begin(),`
	`  rev.end());`
	`  return str == rev;`
	`}`
		$O(n)$, где n — длина строки.
		Функция конструирует копию входной строки, записанную в обратном порядке

`def palindrome2(`
`  string):`
`  n = len(string)`
`  for i in range(n):`
`  j = n - 1 - i`
`  if (string[i] !=`
`  string[j]):`
`  return False`
`  return True`
	`bool Palindrome2(`
	`  const string& str) {`
	`  int n =`
	`  (int)str.length();`
	`  for (int i = 0;`
	`  i < n; ++i) {`
	`  int j = n - 1 - i;`
	`  if (str[i] !=`
	`  str[j]) {`
	`  return false;`
	`  }`
	`  }`
	`  return true;`
	`}`
		$O(1)$.
		Функция создает только переменные фиксированного размера, который не зависит от длины входной строки

Потребление памяти и оптимизация

Приведенные выше примеры иллюстративны. На практике может оказаться, что компилятор (в случае C++) или интерпретатор (в случае PyPy) произведет оптимизацию, избавляющуюся от выделения ненужной памяти, и тем самым превратит оба варианта реализации в одинаковый машинный код. Однако в более сложных случаях, рассмотренных в последующих главах книги, оптимизация вручную будет давать практическое преимущество.

9.3. Практическая применимость асимптотических оценок

Может показаться, что асимптотические оценки «оторваны от реальности», ведь они по своей сути лишь описывают поведение алгоритма на бесконечности и игнорируют точное число операций в конкретных случаях. Условный алгоритм с асимптотикой $O(n)$ может требовать на практике $10^{100} \cdot n$ операций, что при любых реальных входных данных будет гораздо медленнее, чем другой алгоритм с асимптотикой $O(n^2)$ и точным числом операций n^2.

Однако на практике все не так плохо и реально используемых алгоритмов со скрытыми константами 10^{100} нет. Типичные алгоритмы, которые мы будем рассматривать в этой книге, имеют скрытые константы в диапазоне от, скажем, единиц до тысяч (в зависимости от того, что считать отдельной операцией). Поэтому **асимптотическая оценка дает первое приближение**, по которому можно определить, имеет рассматриваемый алгоритм шансы в данной задаче или нет.

Дальше начинается искусство: способность интуитивно оценить скрытую константу и тем самым уточнить предполагаемое время работы на конкретной задаче. Это навык, который развивается в процессе решения задач и проверки решений в тестирующих системах.

Впрочем, в спортивном программировании зачастую можно **«угадать» требуемую асимптотику решения** на основе входных ограничений и дальше предположить, что подойдет любое адекватное решение с этой асимптотикой. Этот подход основан на том, что задачи в спортивном программировании обычно строятся так, чтобы предполагаемые решения правильным алгоритмом укладывались в лимит времени с некоторым запасом, а нежелательные — работали сильно дольше этого лимита. На основе этого, а также эмпирических знаний о временах работы типичных алгоритмов можно прийти к интуитивным схемам подобного плана:

Входные ограничения	**Предположительная асимптотика решения**
$n \le 500$	не хуже $O(n^3)$
$n \le 10^4$	не хуже $O(n^2)$
$n \le 5 \cdot 10^5$	не хуже $O(n \log n)$
$n \le 10^7$	не хуже $O(n)$

Разумеется, приведенные числа — лишь гипотетические оценки; реальные показатели зависят от сложности алгоритма (в смысле величины его скрытой константы), производительности компьютера и скорости работы языка программирования. Эту таблицу следует рассматривать лишь как отправную точку.

9.4. Библиотечные реализации алгоритмов и их скорость

Стандартные библиотеки многих современных языков программирования содержат реализации различных структур данных и алгоритмов. При их применении в спортивном программировании важно, как минимум, знать асимптотическую сложность операций над ними.

Детальное знание алгоритмов стандартной библиотеки

На наш взгляд, знать досконально устройство алгоритмов, используемых в стандартной библиотеке, **не требуется**. Это относится к таким алгоритмам, как красно-черные деревья (которые часто используются для реализации std::map и std::set в C++), сортировка quick sort (к тому же, обычно она представлена в усложненных вариантах, чтобы избежать медленной работы на специальных тестах) и т. п. Знания асимптотики и приблизительного представления о времени работы на практике более чем достаточно в подавляющем большинстве задач. Наша рекомендация читателю: инвестируйте больше времени в решение задач, чем в изучение тех алгоритмов, которые реализовывать самостоятельно не нужно либо же бесперспективно в условиях спортивного программирования.

Приведем асимптотические оценки для некоторых наиболее часто используемых примитивов:

Операция	**Язык**	**Название в стандартной библиотеке**	**Асимптотическая сложность**
Массив: добавление элемента в конец	C++	`std::vector::push_back()`	$O(1)$ в среднем
	Python	`list.append()`	
Массив: удаление элемента из конца	C++	`std::vector::pop_back()`	$O(1)$
	Python	`list.pop()`	$O(1)$ в среднем
Массив: добавление элемента в произвольное место	C++	`std::vector::insert()`	$O(n)$
	Python	`list.insert()`	
Массив: удаление элемента из произвольного места	C++	`std::vector::erase()`	$O(n)$
	Python	`list.pop()`	
Словарь: добавление элемента	C++	`std::map::insert()`, `std::set::insert()`	$O(\log n)$
	Python	`dict[key] = value`, `set.add()`	$O(1)$ в среднем для случайных значений, $O(n)$ в худшем (*)
Словарь: поиск элемента	C++	`std::map::find()`, `std::map::count()` `std::set::find()`, `std::set::count()`	$O(\log n)$
	Python	`dict[key]`, `value in set`	$O(1)$ в среднем для случайных значений, $O(n)$ в худшем (*)

(окончание)

Операция	Язык	Название в стандартной библиотеке	Асимптотическая сложность
Словарь: удаление элемента	C++	`std::map::erase()`, `std::set::erase()`	$O(\log n)$
	Python	`del dict[key]`, `set.remove()`	$O(1)$ в среднем для случайных значений, $O(n)$ в худшем (*)
Сортировка	C++	`std::sort`	$O(n \log n)$
	Python	`sorted()`, `list.sort()`	

(*) Асимптотика операций со словарем в Python.

Реализации CPython и PyPy используют хеш-таблицы для хранения объектов dict и set. Как было описано в предыдущем разделе, производительность хеш-таблиц напрямую зависит от того, насколько часто случаются коллизии хеша. На случайных данных стоимость операций составляет O(1) в среднем. Однако надо понимать, что теоретически авторы задачи могли сконструировать входные наборы, вызывающие большое число коллизий, что приведет к катастрофическому падению производительности до O(n) на каждую операцию. Отметим, что начиная с версии Python 3.4 и PyPy3 используется улучшенная рандомизированная хеш-функция, что делает специальное создание «сложных» входных наборов гораздо более сложным, хотя и по-прежнему возможным.

Некоторые комментарии:

- Массивы.
  - В C++ буфер для std::vector можно заранее выделить под нужный размер с помощью вызова `std::vector::reserve()`, что позволит избежать лишних копирований и выделений памяти по ходу добавления элементов. Однако следует помнить, что каждый вызов этой функции может работать линейное время, поэтому злоупотребление ей (например, вызов ее в цикле) может ухудшить асимптотику до квадратичной!
  - Мы не рассматривали подробно такую структуру данных в C++, как std::list — двусвязный список. Она имеет весьма низкую производительность, из-за того что добавление каждого элемента требует отдельного выделения памяти, и в реалиях спортивного программирования ее применение практически никогда не оправдано.
- Строки.
  - В C++ класс std::string для работы со строками имеет точно такие же показатели асимптотики, что и std::vector.
  - В Python есть проблема, связанная с тем, что строковые объекты — неизменяемые по своей семантике. Это приводит к плохой (линейной) асимптотике операций над ними; стандартный пример — конкатенация строк в цикле, которая работает квадратичное время за исключением случаев, когда интерпре-

татор оптимизирует эту операцию (примечание: CPython умеет делать эту оптимизацию, но PyPy — нет; это является редким примером более плохой производительности PyPy).

- Словари.
  - В то время как в C++ структуры данных типа «словарь» имеют гарантированно хорошую (логарифмическую) асимптотику операций, в Python такой гарантии нет. Впрочем, в большинстве задач это проблем не создает — но только до тех пор, пока авторы специально не попытались создать специальный тест против таких реализаций на Python.
  - В C++ есть также реализация структур данных типа «словарь» на основе хеш-таблиц: std::unordered_set, std::unordered_map. Асимптотика операций над ними схожа с приведенной выше асимптотикой этих реализаций в Python, включая их преимущества (высокая производительность при работе со случайными данными) и недостатки (риск крайне медленной работы на специально сконструированных данных).
  - В C++ есть структуры данных «мультисловарь», позволяющие хранить по нескольку элементов с одинаковым ключом: std::multimap, std::multiset. Их интерфейс и асимптотика операций аналогичны std::map/std::set. Однако есть и важная деталь: `std::multimap::count()` и `std::multiset::count()` работают за время $O(\log n + k)$, где k — число элементов с заданным ключом; неучет этого фактора может привести к квадратичному решению!
- Стеки и очереди.
  - В C++ они представлены типами std::stack, std::queue, std::deque. Операции над ними — добавление и удаление элементов на конце — занимают константное или константное в среднем время в зависимости от реализации.
  - В Python они реализованы классом collections.deque. Добавление и удаление элемента требует константного числа операций в среднем.

Наконец, приведем результаты эмпирических замеров скорости работы этих стандартных реализаций. Как и в предыдущем разделе, это лишь ориентировочные числа, полученные автором в определенных условиях; их можно использовать лишь для сравнения друг с другом или же для приблизительных оценок:

Операция	**Число элементов**	**Язык**	**Время выполнения**
Добавление в пустой список n элементов (C++: n вызовов `std::vector<int>::push_back()`, Python: n вызовов `list.append()`)	$n = 10^5$	C++	1 мс
		Python	1 мс
	$n = 10^6$	C++	10 мс
		Python	20 мс
	$n = 10^7$	C++	100 мс
		Python	500 мс

(окончание)

Операция	**Число элементов**	**Язык**	**Время выполнения**
Добавление в пустой словарь n элементов (C++: n вызовов `std::set<int>::insert()`, Python: n вызовов `set.add()`)	$n = 10^5$	C++	25 мс
		Python	5 мс
	$n = 10^6$	C++	350 мс
		Python	100 мс
Сортировка n чисел	$n = 10^5$	C++	5 мс
		Python	15 мс
	$n = 10^6$	C++	100 мс
		Python	300 мс
	$n = 10^7$	C++	1000 мс
		Python	2000 мс

Глава 10. Наибольший общий делитель. Алгоритм Евклида

10.1. Постановка задачи

Даны два натуральных числа. Требуется найти их наибольший общий делитель (сокращенно — НОД), т. е. максимальное из натуральных чисел, которые делят их оба нацело.

Например, НОД для 12 и 18 равен 6:

- делители 12: 1, 2, 3, 4, **6**, 12;
- делители 18: 1, 2, 3, **6**, 9, 18.

По-английски наибольший общий делитель называется greatest common divisor, сокращенно — gcd; это обозначение будет использоваться нами далее в формулах и в программном коде.

Приведем еще несколько примеров: gcd(49, 35) = 7, gcd(6, 6) = 6, gcd(1, 10) = 1.

10.2. Тривиальный алгоритм

Легко можно реализовать решение «в лоб», которое будет перебирать все числа, начиная с единицы, и проверять каждое из них путем взятия остатка от обоих входных чисел.

Такое решение — чрезвычайно простое, однако в то же время и весьма неэффективное. Его время работы составит $O(n)$, где n — верхняя граница для входных чисел. Хотя это решение можно оптимизировать до некоторой степени (путем отсеивания заведомо ненужных чисел-кандидатов), далее мы рассмотрим принципиально другой алгоритм, работающий гораздо быстрее при столь же простой реализации.

10.3. Алгоритм Евклида

Алгоритм Евклида кратко описывается следующим образом:

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0, \\ \gcd(b, a \bmod b) & \text{if } b \neq 0, \end{cases}$$

где операция mod обозначает взятие остатка от деления.

Таким образом, алгоритм представляет собой набор итераций: каждая итерация состоит из деления по модулю и обмена местами двух аргументов. Алгоритм завершается, когда один из аргументов становится равным нулю, и тогда второй аргумент возвращается в качестве результата.

Историческая справка

Алгоритм назван в честь древнегреческого математика, в труде которого «Начала» (книга VII; написана приблизительно в 300 г. до н. э.) была описана эта процедура — разумеется, в несколько иной формулировке, чем данное выше современное определение. Стоит отметить, что, по некоторым предположениям, алгоритм был открыт другим, возможно, жившим ранее древнегреческим математиком, а Евклид лишь зафиксировал его результат в книге, дошедшей до современности. Так или иначе, вероятно, алгоритм Евклида — старейший алгоритм, который продолжает быть активно применяемым и по сей день.

10.4. Доказательство алгоритма Евклида

Вначале докажем, что алгоритм Евклида всегда завершается. Для этого достаточно заметить, что второй аргумент на каждой итерации строго убывает — поэтому, учитывая, что он неотрицательный, число итераций алгоритма обязано быть конечным.

Теперь докажем корректность алгоритма, т. е. то, что величина в правой части определения действительно равна искомому НОД. Заметим, что для этого достаточно показать, что общие делители не меняются при переходе от пары (a, b) к паре $(a - b, b)$, потому что операция взятия остатка эквивалентна операции вычитания, повторенной некоторое количество раз. Поскольку всякий общий делитель двух чисел также делит их сумму и их разность, то и утверждение про неизменность делителей при переходе между (a, b) и $(a - b, b)$ тоже верно в обе стороны. Для завершения доказательства остается только рассмотреть случай $b = 0$; в этом случае a является правильным ответом, поскольку ноль делится на любое натуральное число.

10.5. Реализация алгоритма Евклида

Рассмотрим способы реализации алгоритма Евклида.

10.5.1. Рекурсивная реализация

Эта реализация в точности следует описанию алгоритма, данному в *разд. 10.3*.

Python

```
def gcd(a, b):
  if b == 0:
    return a
  return gcd(b, a % b)
```

C++

```
int Gcd(int a, int b) {
  if (b == 0)
    return a;
  return Gcd(b, a % b);
}
```

Отметим, что эта реализация работает корректно также и в случаях, когда один или оба аргумента равны нулю.

10.5.2. Краткая рекурсивная реализация

Эта реализация аналогична приведенной выше, однако записана с использованием тернарного оператора. Такая краткая реализация особенно популярна в спортивном программировании.

Python

```
def gcd(a, b):
  return gcd(b, a % b) if b else a
```

C++

```
int Gcd(int a, int b) {
  return b ? Gcd(b, a % b) : a;
}
```

10.5.3. Нерекурсивная реализация

В этой реализации рекурсивные вызовы (т. е. вызовы функцией самой себя же) заменены на итерацию.

Python

```
def gcd(a, b):
  while b:
    a, b = b, a % b
  return a
```

C++

```
int Gcd(int a, int b) {
  while (b) {
    a %= b;
    swap(a, b);
  }
  return a;
}
```

10.5.4. Выбор между рекурсивной и нерекурсивной реализациями

- На языке C++ типичный современный компилятор (GCC, Clang, MSVC и т. п.) умеет оптимизировать хвостовую рекурсию, разворачивая ее в итерацию. Поэтому разница в скорости работы между рекурсивной и нерекурсивной реализациями алгоритма Евклида очень мала или отсутствует. В связи с этим рекурсивная реализация, как более лаконичная, — стандартный выбор в спортивном программировании.
- В стандартном CPython такого рода автоматические оптимизации отсутствуют, и поэтому нерекурсивная реализация будет заметно (до двух и более раз) быстрее. В PyPy разница в скорости по-прежнему будет, хотя и не такой сильной (на компьютере автора она составила 25%).

10.5.5. Библиотечные реализации

Отметим, что во многих языках программирования алгоритм Евклида уже реализован в том или ином виде в стандартной библиотеке:

- в Python есть библиотечная функция `math.gcd()` (до версии 3.5 — `fractions.gcd()`);
- в C++ начиная с версии C++17 заголовочным файлом <numeric> предоставляется функция `std::gcd()`;
- при использовании более ранних версий C++ и компиляторов GCC или Clang, заголовочным файлом <algorithm> предоставляется функция `std::__gcd()`. **Внимание**: эта функция является непереносимым и плохо документированным расширением; некоторые из ее реализаций некорректно обрабатывают нули во входных аргументах, приводя к «падению» или зависанию программы на таких вызовах!

Разумеется, при наличии доступной (и надежной — см. выше) библиотечной реализации имеет смысл пользоваться ей вместо реализации алгоритма вручную. Однако ознакомиться со способами самостоятельной реализации этого алгоритма стоит в любом случае. Кроме того, это знание пригодится при изучении так называемого расширенного алгоритма Евклида, представленного в последующих главах этой книги и который обычно не реализован в стандартных библиотеках.

10.6. Время работы алгоритма Евклида

Покажем, что алгоритм Евклида завершается за время $O(\log n)$, где $n = \min(a, b)$.

Заметим следующий факт: для любых двух натуральных $x \le y$ верно, что $y \bmod x < y / 2$. В самом деле:

- если $x \le y / 2$, то, так как результат взятия остатка всегда строго меньше делителя, он строго меньше $y / 2$;
- если же $y / 2 < x \le y$, то, так как при взятии остатка при этих ограничениях делимое уменьшается на делитель ровно один раз, результат опять же строго меньше $y / 2$.

Используя этот факт, получаем, что при условии $a \ge b$ каждая итерация алгоритма Евклида уменьшает второй аргумент, как минимум, вдвое. Если же $a < b$, то алгоритм совершит одну «холостую» итерацию, в результате которой аргументы обменяются местами, и мы опять придем к рассмотренному только что случаю. Таким образом, всего понадобится не более $\lceil \log_2 n \rceil + 1$ итераций, прежде чем второй аргумент примет значение не более единицы, что и доказывает асимптотическую оценку $O(\log n)$.

Можно построить пример, на котором эта асимптотическая оценка достигается. А именно, если применить алгоритм Евклида к соседним членам **последовательности Фибоначчи** — т. е. запустить $\gcd(F_k, F_{k+1})$, — то понадобится порядка k итераций, так как в результате взятия остатка будет получаться предыдущее число Фибоначчи. Поскольку числа Фибоначчи растут с экспоненциальной скоростью, то требуемое число итераций на тесте такого вида составит $O(\log n)$.

Впрочем, эту оценку можно немного **улучшить** до следующей: $O(\min(a, b) / \gcd(a, b))$. Для этого достаточно заметить, что все шаги алгоритма Евклида останутся без изменений, если оба входных аргумента домножить на одно и то же число.

Примечание

Приведенные асимптотические оценки предполагают, что операция деления выполняется за $O(1)$. Это справедливо при использовании встроенных типов данных (таких, как int в C++) или при наличии фиксированной верхней границы на входные параметры. В противном случае асимптотическая оценка будет зависеть от сложности отдельных операций, что, вообще говоря, зависит от выбранной модели вычислений.

10.7. Пример решения задачи. Сокращение дроби

Задача. Даны два целых числа — числитель и знаменатель некоторой дроби (знаменатель положителен). Требуется вывести числитель и знаменатель сокращенной дроби.

Примеры

Входные данные	**Требуемый результат**
12 18	2 3
Входные данные	**Требуемый результат**
3 5	3 5

Решение

Посчитаем НОД от числителя и знаменателя с помощью алгоритма Евклида. Предварительно возьмем числитель по модулю во избежание проблем с отрицательными числами (это может быть необходимо в зависимости от реализации). Ответом будут исходные числитель и знаменатель, деленные на НОД.

10.8. Пример решения задачи. Наименьшее общее кратное

Задача. Даны два натуральных числа n и m. Требуется найти их наименьшее общее кратное (НОК), т. е. минимальное из всех чисел, которые делятся и на n, и на m.

Примеры

Входные данные	**Требуемый результат**
12 18	36
Входные данные	**Требуемый результат**
20 10	20

Решение

Обозначим НОК чисел n и m через $\mathrm{lcm}(n, m)$, следуя английской терминологии least common multiple. Тогда утверждается, что:

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

Доказательство

Интуитивно эта формула «удаляет» «лишние» делители из произведения $n \cdot m$, превращая его из некоего общего кратного в наименьшее общее кратное.

Однако эту формулу можно обосновать и более строго. Представим оба числа n и m в виде разложений по степеням простых делителей, т. е. факторизуем их:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k},$$
$$m = p_1^{\beta_1} p_2^{\beta_2} \ldots p_k^{\beta_k}.$$

Можно заметить, что в таком представлении НОД и НОК определяются через, соответственно, минимумы и максимумы по степеням каждого из простых:

$$\gcd(n,m) = p_1^{\min(\alpha_1,\beta_1)} p_2^{\min(\alpha_2,\beta_2)} \ldots p_k^{\min(\alpha_k,\beta_k)},$$
$$\mathrm{lcm}(n,m) = p_1^{\max(\alpha_1,\beta_1)} p_2^{\max(\alpha_2,\beta_2)} \ldots p_k^{\max(\alpha_k,\beta_k)}.$$

Следовательно:

$$\frac{nm}{\gcd(n,m)} = p_1^{\alpha_1+\beta_1-\min(\alpha_1,\beta_1)} p_2^{\alpha_2+\beta_2-\min(\alpha_2,\beta_2)} \ldots p_k^{\alpha_k+\beta_k-\min(\alpha_k,\beta_k)}.$$

Поскольку для любых двух чисел x и y верно, что $\max(x, y) = x + y - \min(x, y)$, то и выражение выше эквивалентно выражению для $\mathrm{lcm}(n, m)$, что и требовалось доказать.

Реализация

При реализации с использованием встроенных типов данных (таких, как int в C++), имеет смысл сначала выполнить деление и только затем умножение, чтобы избежать по мере возможности целочисленного переполнения (в случаях, когда и сами числа, и их НОД достаточно велики):

C++

```
int Lcm(int n, int m) {
  return n / Gcd(n, m) * m;
}
```

Отметим, что если возможна ситуация $n = m = 0$, то в реализацию придется добавить проверку на этот случай, чтобы избежать деления на ноль.

10.9. Пример решения задачи. НОД нескольких чисел

Задача. Даны n натуральных чисел a_1, a_2, …, a_n. Требуется вывести их общий НОД, т. е. наибольшее из чисел, которые делят все a_i нацело одновременно.

Примеры

Входные данные	**Требуемый результат**
4 9 6 27 12	3
Входные данные	**Требуемый результат**
1 10	10

Решение

Сведем задачу к набору вычислений НОД от двух чисел; последнее мы умеем делать с помощью алгоритма Евклида. Утверждается, что следующая формула верна:

$$\gcd(a_1, a_2, a_3) = \gcd(\gcd(a_1, a_2), a_3).$$

Таким образом, ответ для всего набора можно посчитать, добавляя числа по одному и обновляя ответ как НОД от текущего ответа и добавляемого числа.

Асимптотическая сложность этого решения — $O(n \log c)$, где c — максимум из входных чисел.

Доказательство

Как и в предыдущей задаче, воспользуемся представлением чисел в виде их факторизаций:

$$a_1 = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k},$$
$$a_2 = p_1^{\beta_1} p_2^{\beta_2} \ldots p_k^{\beta_k},$$
$$a_3 = p_1^{\gamma_1} p_2^{\gamma_2} \ldots p_k^{\gamma_k}.$$

Тогда, с одной стороны, НОД этих трех чисел определяется через минимумы по степеням каждого из простых:

$$\gcd(a_1, a_2, a_3) = p_1^{\min(\alpha_1, \beta_1, \gamma_1)} p_2^{\min(\alpha_2, \beta_2, \gamma_2)} \ldots p_k^{\min(\alpha_k, \beta_k, \gamma_k)}.$$

С другой стороны, два последовательных применения НОД дают следующую формулу:

$$\gcd(\gcd(a_1, a_2), a_3) = p_1^{\min(\min(\alpha_1, \beta_1), \gamma_1)} p_2^{\min(\min(\alpha_2, \beta_2), \gamma_2)} \ldots p_k^{\min(\min(\alpha_k, \beta_k), \gamma_k)}.$$

Поскольку для любых трех чисел x, y, z верно $\min(x, y, z) = \min(\min(x, y), z)$, то два выражения выше эквивалентны друг другу, что завершает доказательство.

Реализация

Для упрощения реализации мы пользуемся тем фактом, что $\gcd(0, x) = x$; таким образом, мы избавляемся от особого случая с первоначальным вычислением НОД от первых двух чисел, а также от особого случая при $n = 1$.

Python

```
def gcd_list(a):
  result = 0
  for item in a:
    result = gcd(result, item)
  return result
```

C++

```
int GcdList(const vector<int>& a) {
  int result = 0;
  for (auto item : a)
    result = Gcd(result, item);
  return result;
}
```

10.10. Пример решения задачи. Увеличение НОД массива

Задача. Даны n ($2 \le n \le 10^6$) натуральных чисел a_1, a_2, …, a_n; числа не превосходят 10^{18}. Разрешается заменить одно из чисел по нашему усмотрению. Требуется найти такую замену, которая сделает НОД всего массива наибольшим из всех возможных, и вывести получившийся НОД.

Примеры

Входные данные	**Требуемый результат**
4 9 18 30 12	6
Входные данные	**Требуемый результат**
2 100 200	200

Примечание к первому примеру: этот результат можно достичь, заменив число девять на любое число, кратное шести.

Решение

Можно было бы попытаться перебирать всевозможные замены, заменяя одно число на каждое из остальных чисел массива (можно показать, что ответ всегда получится сконструировать таким образом); однако это было бы решением за $O(n^2)$, что явно слишком медленно.

Другой подход к решению: изменим условие задачи, а именно, вместо смены числа будем удалять выбранное число. Отличается ли ответ на эту задачу от исходной задачи? На самом деле — нет. При замене какого-либо числа наилучший ответ, которого можно достичь, — это НОД оставшихся чисел, что совпадает с ответом при удалении числа.

Для решения новой задачи будем перебирать позицию удаляемого числа, и для каждой такой позиции научимся эффективно вычислять НОД остальных чисел. Это значение сводится к значениям НОД от текущего префикса и НОД от текущего суффикса:

$$\text{answer}_i = \gcd(\gcd(a_1, ..., a_{i-1}), \gcd(a_{i+1}, ..., a_n)).$$

Обе этих величины можно определять эффективно. НОД текущего префикса можно поддерживать в дополнительной переменной в процессе перебора. НОД каждого из суффиксов можно посчитать и сохранить предварительно дополнительным проходом по массиву в обратном порядке.

Асимптотическая сложность получившегося решения — $O(n \log c)$, где c — максимум из входных чисел.

Реализация

Python

```
def solve(a):
  suffix = [0] * (len(a) + 1)
  for i, item in reversed(
      list(enumerate(a))):
    suffix[i] = gcd(
      suffix[i + 1], item)
  answer = 0
  prefix = 0
  for i, item in enumerate(a):
    answer = max(
      answer,
      gcd(prefix, suffix[i + 1]))
    prefix = gcd(prefix, item)
  return answer
```

C++

```
int Solve(const vector<int>& a) {
  int n = (int)a.size();
  vector<int> suffix(n + 1);
  for (int i = n - 1; i >= 0; --i)
    suffix[i] = Gcd(
      suffix[i + 1], a[i]);
  int answer = 0;
  int prefix = 0;
  for (int i = 0; i < n; ++i) {
    answer = max(
      answer,
      Gcd(prefix, suffix[i + 1]));
    prefix = Gcd(prefix, a[i]);
  }
  return answer;
}
```

10.11. Упражнения для самостоятельного решения

Настоятельно рекомендуется прорешать приведенные ниже задачи с целью закрепления пройденного материала. Подсказки и ответы для самоконтроля приведены в *приложении*.

10.11.1. Квадратный кафель

Задача. Даны два натуральных числа l и w — размеры пола (не превосходящие 10^{18}). Требуется замостить пол одинаковыми квадратными плитками кафеля, причем плитки нельзя обрезать или накладывать друг на друга. Нужно найти максимально возможный размер плитки.

Примеры

Входные данные	**Требуемый результат**
150 200	50
Входные данные	**Требуемый результат**
200 100	100

10.11.2. НОД диапазона натуральных чисел

Задача. Даны два числа l и r ($1 \le l \le r \le 10^{18}$). Требуется вывести НОД всех натуральных чисел, принадлежащих отрезку $[l; r]$.

Примеры

Входные данные	**Требуемый результат**
1 10	1
Входные данные	**Требуемый результат**
5 5	5

10.11.3. Подмножество с заданным НОД

Задача. Даны n ($1 \le n \le 10^3$) натуральных чисел a_1, …, a_n и натуральное число k (все a_i и k не превосходят 10^9). Требуется выбрать среди чисел a_i такой набор, чтобы его НОД был равен k. Если такого набора не существует, то следует вывести «−1».

Примеры

Входные данные	**Требуемый результат**
6 1 2 3 40 50 60 20	40 60
Входные данные	**Требуемый результат**
1 10 2	-1

10.11.4. НОК дробей

Задача. Даны n дробей ($1 \le n \le 10^3$), заданных своими числителями a_i и знаменателями b_i (которые являются натуральными числами, не превосходящими 10^9). Гарантируется, что заданные дроби — несократимые. Требуется найти НОК этих дробей, т. е. наименьшую несократимую дробь такую, что при делении ее на каждую из входных дробей получается целое число. Если такой дроби не существует или ее числитель или знаменатель превосходят 10^{18}, то следует вывести «−1».

Примеры

Входные данные	Требуемый результат
2 2 15 3 10	6 5
Входные данные	**Требуемый результат**
1 7 8	7 8

Пояснение к первому примеру: дробь 6 / 5 при делении на 2 / 15 дает 9, а при делении на 3 / 10 дает 4.

10.11.5. Целые точки на отрезке

Задача. Даны координаты двух точек — концов отрезка (координаты целые, не превосходящие 10^{18} по модулю). Требуется посчитать, сколько всего точек с целочисленными координатами лежат на отрезке.

Примеры

Входные данные	Требуемый результат
1 2 3 0	3
Входные данные	**Требуемый результат**
1 0 1 0	1

Пояснение к первому примеру: отрезку принадлежат следующие целочисленные точки: (1; 2), (2; 1), (3; 0).

10.11.6. НОД многочленов

Задача. Даны два многочлена. Требуется найти их НОД, т. е. такой многочлен наибольшей степени, что оба входных многочлена можно получить из него путем домножения на какой-либо многочлен.

Входные многочлены заданы своей степенью (от 0 до 6) и коэффициентами в порядке от старшего к младшему (лежащими в отрезке [−10; 10]).

Из всех возможных решений следует вывести то, у которого все коэффициенты целые, взаимно простые (т. е. их НОД равен единице) и старший коэффициент положителен.

Пример

Входные данные	Требуемый результат
4 1 0 1 2 4 2 2 -1 -1	2 1 1

Пояснение к примеру: исходные многочлены можно представить следующим образом: x3 + x + 2 = (x + 1)(x2 – x + 2), 2x3 + 2x2 – x – 1 = (x + 1)(2x2 − 1).

Глава 11.
Простые задачи на учет асимптотики

В этой главе мы рассмотрим задачи, решение которых надо выбирать исходя из асимптотической сложности алгоритмов.

11.1. Пример решения задачи. Префиксы перестановки

Задача. Дана перестановка p_i длины n. Требуется для каждого ее префикса определить, является ли он перестановкой.

Входные данные состоят из числа n в первой строке и n разделенных пробелом чисел p_i во второй. Гарантируется, что все числа натуральные, $n \le 5 \cdot 10^5$, все p_i различны и не превосходят n. Вывести требуется n чисел, и k-е число должно быть равно 1, если $[p_1, \ldots, p_k]$ является перестановкой, и 0 в противном случае.

Примеры

Входные данные	**Требуемый результат**
4 1 3 2 4	1 0 1 1
Входные данные	**Требуемый результат**
3 3 2 1	0 0 1

Решение

Наивный алгоритм решения был бы следующим: перебрать k и для каждого значения запустить обычный алгоритм проверки перестановки на корректность: проверить, что никакое из чисел $[p_1, \ldots, p_k]$ не превосходит k и не встречается дважды. Однако это решение имело бы асимптотику $O(n^2)$, поскольку состояло бы из двух вложенных циклов по n операций в каждом; это решение было бы слишком медленным при заданных ограничениях.

Для поиска более эффективного решения заметим, что входной массив $[p_1, \ldots, p_n]$ — уже перестановка. Следовательно, проверять, один или много раз встречается каждое число, не требуется. Единственное, что нам осталось проверить, — это то,

что никакое из чисел [p_1, …, p_n] не превосходит k. Для этого достаточно просто поддерживать текущий максимум в отдельной переменной.

Это решение будет работать за $O(n)$, поскольку оно будет состоять из одного цикла, проходящего по перестановке ровно один раз.

Реализация

Python

```
n = int(input())
p = list(map(int, input().split()))
ans = []
mx = 0
for i, cur in enumerate(p):
  mx = max(mx, cur)
  ans.append(1 if mx == i + 1
             else 0)
print(*ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> p(n);
  for (int i = 0; i < n; ++i)
    cin >> p[i];
  vector<int> ans(n);
  for (int i = 0, mx = 0; i < n;
       ++i) {
    mx = max(mx, p[i]);
    ans[i] = mx == i + 1;
  }
  for (int i = 0; i < n; ++i)
    cout << ans[i] << ' ';
}
```

11.2. Пример решения задачи. Парковочные места

Задача. Парковочные места перед супермаркетом пронумерованы по порядку натуральными числами. Будка охранника находится возле места №1; чтобы ему было проще держать все под своим наблюдением, он отправляет каждый приезжающий автомобиль к свободному парковочному месту с наименьшим номером. Помогите охраннику справиться с этим заданием, если про каждый автомобиль известно, в какой момент времени он приезжает и в какой — уезжает. Примечание: на место, освобождаемое автомобилем в некоторый момент времени x, следующий автомобиль может приехать в момент времени $x + 1$.

Первая строка входных данных содержит число n автомобилей ($1 \le n \le 5 \cdot 10^5$). Каждая из следующих n строк содержит описание очередного автомобиля: время приезда a_i и отъезда b_i, записанные через пробел. Все числа — натуральные, и $1 \le a_i < b_i \le 10^9$. Вывести требуется n разделенных пробелом чисел — номер парковочного места для каждого автомобиля в порядке их следования во входных данных.

Примеры

Входные данные	**Требуемый результат**
4 5 8 1 10 9 20 11 20	2 1 2 1
Входные данные	**Требуемый результат**
6 1 2 1 2 2 3 3 4 3 4 3 4	1 2 3 1 2 4

Решение

В первую очередь, заметим, что нам нужно упорядочить события приезда и отъезда машин по времени, чтобы выполнять эти операции в правильном порядке. Для этого достаточно поместить все n событий приезда и n событий отъезда в список и отсортировать его за $O(n \log n)$. При этом мы должны обеспечить, чтобы при равенстве времен события приезда шли до событий отъезда, причем события приезда должны идти в порядке увеличения номеров автомобилей. Например, этого можно достичь, если сортировать тройки *(время, тип события, номер автомобиля)*.

Далее, единственная сложность этой задачи — в эффективном моделировании описанного в условии процесса. Наивное решение — искать для каждой машины свободное место перебором позиций начиная с единицы — требовало бы в худшем случае $O(n^2)$ операций, что слишком медленно при заданных ограничениях на n.

Для оптимизации нужна эффективная структура данных, которая поддерживает следующие операции: добавление элемента (числа), удаление элемента и поиск элемента с минимальным значением. Стандартные библиотеки многих языков программирования предоставляют готовые реализации структур данных, поддерживающих эти операции за время $O(\log n)$: std::set и std::priority_queue в C++, модуль heapq в Python. С использованием одной из таких структур данных мы сможем обрабатывать каждое событие (приезд машины или отъезд машины) за время $O(\log n)$.

Итоговая асимптотика решения составит $O(n \log n)$.

Реализация

Python

```
from heapq import *
n = int(input())
events = []
for car in range(n):
  tadd, tremove = map(
      int, input().split())
  events.append((tadd, 0, car))
  events.append((tremove, 1, car))
ans = [None] * n
free_slots = list(range(1, n + 1))
heapify(free_slots)
for t, type, car in sorted(events):
  if type == 0:
    ans[car] = heappop(free_slots)
  else:
    heappush(free_slots, ans[car])
print(*ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <set>
#include <tuple>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<tuple<int, int, int>> events;
  for (int car = 0; car < n; ++car) {
    int tadd, tremove;
    cin >> tadd >> tremove;
    events.emplace_back(tadd, 0, car);
    events.emplace_back(
        tremove, 1, car);
  }
  sort(events.begin(), events.end());
  set<int> free_slots;
  for (int i = 1; i <= n; ++i)
    free_slots.insert(i);
  vector<int> ans(n);
  for (auto e : events) {
    int t = get<0>(e),
        type = get<1>(e),
        car = get<2>(e);
    if (type == 0) {
      ans[car] = *free_slots.begin();
      free_slots.erase(
          free_slots.begin());
    } else {
      free_slots.insert(ans[car]);
    }
  }
  for (int i = 0; i < n; ++i)
    cout << ans[i] << ' ';
}
```

Глава 12.
Объединение одномерных отрезков

12.1. Постановка задачи

Даны n одномерных отрезков, заданных координатами своих концов l_i и r_i. Требуется построить их объединение, т. е. склеить вместе перекрывающиеся и касающиеся отрезки.

12.2. Алгоритм объединения отрезков

Для решения этой задачи отсортируем отрезки по координате левого конца и будем рассматривать отрезки по одному в этом порядке. Будем при этом накапливать результирующие объединенные отрезки. Тогда для добавления нового входного отрезка достаточно сравнить его с самым правым из отрезков текущего результата: если они пересекаются, то обновить самый правый отрезок текущего результата, иначе — добавить текущий отрезок в виде нового отрезка в результат.

Почему этот алгоритм верен? Единственная оптимизация, совершаемая им по сравнению с тривиальным решением, — это сравнение добавляемого отрезка не со всеми отрезками текущего результата, а только с самым правым из них. Корректность этой оптимизации базируется на том факте, что мы рассматриваем входные отрезки в порядке роста координат левых концов. Благодаря этому добавляемый отрезок не может начинаться левее, чем самый правый отрезок текущего результата, а, значит, не может иметь общих точек с предыдущими отрезками текущего результата.

Временная сложность этого алгоритма складывается из сортировки и последующего прохода за линейное время и поэтому равна $O(n \log n)$.

12.3. Реализация алгоритма объединения отрезков

Как уже упоминалось при решении других задач, мы не будем вводить специальных типов данных «отрезок», поскольку в спортивном программировании это имеет смысл разве лишь в длинных и сложных программах.

Python

```
def merge_segm(segms):
  res = []
  for left, right in sorted(segms):
    if res and left <= res[-1][1]:
      res[-1][1] = max(
        res[-1][1], right)
    else:
      res.append([left, right])
  return res
```

C++

```
vector<pair<int, int>> MergeSegm(
    const vector<pair<int, int>>&
    segms) {
  vector<pair<int, int>> res;
  for (auto segm : segms) {
    if (res.empty() || segm.first
          > res.back().second) {
      res.push_back(segm);
      continue;
    }
    res.back().second = max(
      res.back().second,
      segm.second);
  }
  return res;
}
```

12.4. Пример решения задачи. Часы приема

Задача. Несколько пациентов записались на прием в клинику. Для каждого пациента известно время l_i начала и r_i конца его визита. Требуется определить, в какие отрезки времени в клинике будет хотя бы один пациент, и выдать результат в виде минимального по размеру множества отрезков.

Входные данные состоят из числа n запланированных визитов (натурального, не превосходящего 10^5) в первой строке и n строк, содержащих описания визитов: пары l_i и r_i, разделенные пробелом (гарантируется, что $1 \le l_i < r_i \le 10^9$). Вывести требуется число отрезков в итоговом объединении и затем описания отрезков в том же формате.

Примеры

Входные данные	**Требуемый результат**
4 5 10 2 6 11 12 12 13	2 2 10 11 13
Входные данные	**Требуемый результат**
2 10 20 15 16	1 10 20

Решение

Эта задача — на непосредственное применение описанного выше алгоритма, лишь немного завуалированное «легендой». Поскольку никаких модификаций решения не требуется, приводить его здесь еще раз мы не будем.

12.5. Пример решения задачи. Стрельба по отрезкам

Задача. Даны n отрезков, лежащих на прямой. Требуется наименьшим числом выстрелов попасть во все эти отрезки. Выстрел в какую-либо точку пробивает насквозь все отрезки, содержащие эту точку.

В первой строке входных данных дано число n (натуральное, не превосходящее 10^5). Следующие n строк содержат описания каждого из отрезков в виде двух разделенных пробелом целых чисел l_i и r_i ($-10^9 \le l_i \le r_i \le 10^9$). Вывести требуется единственное число — искомое минимальное число выстрелов.

Примеры

Входные данные	Требуемый результат
4 5 10 2 6 11 12 12 13	2
Входные данные	**Требуемый результат**
2 10 20 15 16	1

Пояснения к примерам

В первом примере можно попасть во все четыре отрезка двумя выстрелами. Например, выстрел в точку с координатой 5 попадает в отрезки №1 и №2, а выстрел в точку с координатой 12 попадает в отрезки №3 и №4.

Во втором примере один из возможных выстрелов — в точку с координатой 15.

Решение

Первое ключевое наблюдение: достаточно рассматривать в качестве точек-кандидатов для выстрелов только точки — правые концы входных отрезков. В самом деле, рассмотрим любое оптимальное решение нашей задачи, в котором есть выстрел, не совпадающий ни с одним из правых концов. Тогда будем модифицировать это решение, постепенно сдвигая этот выстрел вправо. Рано или поздно мы дойдем таким образом до ближайшего из правых концов, при этом оптимальность ответа мы

не нарушили, поскольку все изначально простреленные отрезки по-прежнему остались таковыми.

Второе ключевое наблюдение: если мы уже совершили несколько выстрелов и при этом какие-то отрезки остались непростреленными, то мы обязаны выстрелить в самый левый из правых концов оставшихся отрезков. Действительно, выстрелить позже мы не можем, потому что этот самый «ранее всего кончающийся» отрезок останется непростреленным. А стрелять раньше не требуется согласно первому наблюдению.

Таким образом, для решения задачи нам надо лишь эффективно моделировать следующий процесс: до тех пор пока все отрезки не стали простреленными, определить самый «ранее всего кончающийся» отрезок из оставшихся, выстрелить в его правый конец и удалить из рассмотрения все простреленные при этом отрезки. Существует несколько вариантов реализации этого алгоритма.

Приведем здесь вариант на основе универсального приема рассмотрения процесса как цепочки **событий**.

Введем события двух видов: «в точке x левый конец отрезка i» и «в точке x правый конец отрезка i». Отсортируем эти события по возрастанию координаты, а при равенстве — так, чтобы все события левого конца шли до всех событий правого. Заведем список активных отрезков и массив флагов для каждого отрезка, был ли он простремлен или нет.

Будем просматривать события по одному в порядке сортировки. Если текущее событие — левый конец, то просто добавим этот отрезок в список активных. Если текущее событие — правый конец, то проверим флаг, был ли этот отрезок уже простремлен. Если нет, то выстрелим в его правый конец; при этом извлечем все отрезки из списка активных и пометим их как простреленные.

Временная сложность такой реализации складывается из сортировки $2n$ событий и последующей обработки за линейное время и, таким образом, равна $O(n \log n)$.

Реализация

Python

```
n = int(input())
events = []
for i in range(n):
  l, r = map(int, input().split())
  events.append((l, -1, i))
  events.append((r, +1, i))
ans = 0
is_shot = [False] * n
active = []
for x, tp, idx in sorted(events):
  if tp == -1:
    active.append(idx)
```

C++

```
#include <algorithm>
#include <iostream>
#include <tuple>
#include <vector>
using namespace std;
struct event {
  int x, tp, idx;
  bool operator<(const event& e)
      const {
    return std::tie(x, tp, idx) <
      std::tie(e.x, e.tp, e.idx);
  }
```

```
    continue
  if is_shot[idx]:
    continue
  ans += 1
  for i in active:
    is_shot[i] = True
  active = []
print(ans)
```

```
};
int main() {
  int n;
  cin >> n;
  vector<event> events;
  for (int i = 0; i < n; ++i) {
    int l, r;
    cin >> l >> r;
    events.push_back({l, -1, i});
    events.push_back({r, +1, i});
  }
  sort(events.begin(),
       events.end());
  int ans = 0;
  vector<char> is_shot(n);
  vector<int> active;
  for (auto& e : events) {
    if (e.tp == -1) {
      active.push_back(e.idx);
      continue;
    }
    if (is_shot[e.idx])
      continue;
    ++ans;
    for (int i : active)
      is_shot[i] = true;
    active.clear();
  }
  cout << ans << endl;
}
```

12.6. Пример решения задачи. Многослойная покраска

Задача. Несколько маляров работали над покраской забора; i-й маляр красил доски начиная с l_i-й и заканчивая r_i-й (доски пронумерованы слева направо, начиная с единицы). Из-за невысокого качества краски она хорошо держится на доске, только если она нанесена в минимум k слоев. Требуется посчитать число досок забора, покрашенных надежно, если каждый маляр наносит свою краску только в один слой.

В первой строке указаны через пробел натуральные числа n и k. Следующие n строк содержат по паре натуральных чисел l_i и r_i. Гарантируется, что $n \le 10^5$, $k \le n$ и $l_i \le r_i \le 10^9$. Вывести требуется единственное число — количество надежно покрашенных досок.

Примеры

Входные данные	Требуемый результат
4 2 4 5 3 4 1 1 1 2	2
Входные данные	**Требуемый результат**
3 3 10 20 10 20 10 20	11

Пояснения к примерам

В первом примере доски №1 и №4 покрашены в два слоя, а остальные доски покрашены лишь в один слой (№2, №3, №5) или не покрашены вовсе.

Во втором примере все 11 упомянутых во входном наборе досок (№10–20) покрашены в три слоя.

Решение

Воспользуемся методом «событий», описанным в предыдущей задаче. В текущей задаче нам достаточно событий следующего типа: «у доски с номером x число слоев покраски изменяется на ∂». Для каждого маляра мы должны добавить два события: событие $x = l_i$, $\partial = 1$ и событие $x = r_i + 1$, $\partial = -1$.

Отсортируем эти события и будем просматривать их в порядке увеличения x. При этом будем поддерживать текущий баланс числа покрашенных слоев: изначально баланс равен нулю, а при обработке очередного события его надо изменить на очередное значение ∂. После обработки всех событий с одинаковым значением x текущий баланс равен числу слоев краски на доске с номером x. Кроме того, то же самое верно и для всех последующих досок, вплоть до номера следующего события — это наблюдение позволяет нам «перепрыгивать» через длинные отрезки с постоянным числом краски и тем самым эффективно обрабатывать даже огромные входные координаты.

Таким образом, после обработки всех событий с определенной координатой мы должны сравнить текущий баланс с k и, если он оказался большим либо равным, прибавить к ответу разность между координатой следующего события и текущей координатой.

Временная сложность решения складывается из сортировки $2n$ событий и последующей обработки за линейное время и, таким образом, равна $O(n \log n)$.

Реализация

Python

```
n, k = map(int, input().split())
events = []
for _ in range(n):
  left, right = map(
      int, input().split())
  events.append((left, 1))
  events.append((right + 1, -1))
events.sort()
bal = 0
ans = 0
for idx, (x, delta) in enumerate(
    events):
  bal += delta
  if bal >= k:
    ans += events[idx + 1][0] - x
print(ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  vector<pair<int, int>> events;
  for (int i = 0; i < n; ++i) {
    int left, right;
    cin >> left >> right;
    events.emplace_back(left, 1);
    events.emplace_back(
        right + 1, -1);
  }
  sort(events.begin(),
       events.end());
  int ans = 0;
  int bal = 0;
  for (int i = 0; i < n * 2; ++i) {
    bal += events[i].second;
    if (bal >= k) {
      ans += events[i + 1].first -
             events[i].first;
    }
  }
  cout << ans << endl;
}
```

Примечание к проверкам баланса внутри цикла

В реализациях выше мы использовали лаконичную проверку `bal >= k` без дополнительных проверок того, что текущее событие — не последнее, и того, что следующее событие имеет строго бóльшую координату. Хотя эти дополнительные проверки нужны, если буквально следовать данному выше описанию алгоритма, на практике они не требуются. Во-первых, после обработки самого последнего события баланс равен нулю, а, значит, условие `bal >= k` не может выполниться на последней итерации цикла. Во-вторых, при обработке группы событий с одинаковой координатой разность «координата следующего события минус координата текущего события» равна нулю, а, значит, ошибочных изменений ответа происходить не будет.

Глава 13.
Метод двух указателей

Этот метод позволяет кардинально ускорить решение некоторых задач, наивное решение которых имеет квадратичную сложность. Проще всего усвоить этот метод на примере решения конкретных задач.

13.1. Пример решения задачи. Пары фиксированной суммы

Задача. Дана последовательность **различных** целых чисел a_i ($i = 1...n$). Требуется посчитать количество пар с заданной суммой k, т. е. число способов выбрать i и j так, что $a_i + a_j = k$ (где $1 \le i < j \le n$).

Входные данные состоят из чисел n и k в первой строке и последовательности a_i из n чисел во второй. Все числа — целые, $1 \le n \le 5 \cdot 10^5$, $-10^9 \le k \le 10^9$, $-10^9 \le a_i \le 10^9$. Вывести требуется единственное число — искомое количество пар.

Примеры

Входные данные	**Требуемый результат**
`4 5` `1 3 4 2`	`2`
Входные данные	**Требуемый результат**
`4 10` `4 3 2 1`	`0`

Пояснения к примерам

В первом примере существуют две пары с требуемой суммой: 1 + 4 и 2 + 3. Во втором примере нет ни одной пары с суммой 10, поскольку максимальная сумма, которую можно достичь, равна 5.

Решение

Наивное решение было бы следующим: перебрать всевозможные пары i и j и проверить получившуюся сумму $a_i + a_j$. Однако это решение было бы слишком медленным: оно потребовало бы $O(n^2)$ итераций, что при заданных ограничениях составило бы примерно 10^{11} шагов.

Метод двух указателей позволяет кардинально ускорить это решение следующим образом. Предварительно отсортируем входной массив. Тогда заметим, что при переборе всевозможных индексов i в порядке увеличения ($i = 1...n$) соответствующие индексы j строго убывают. В самом деле, если сумма двух чисел фиксированна, то увеличение одного слагаемого влечет за собой уменьшение другого.

Таким образом, быстрое решение будет выглядеть следующим образом: заведем два индекса i и j, один для движения по массиву слева направо, другой — в обратную сторону. Далее совершим n итераций, по одной для каждого из значений $i = 1...n$, и внутри каждой итерации будем уменьшать индекс j до тех пор, пока не встретим значение $k - a_i$ или меньшее его. Если это значение элемента оказалось в точности равным нужному и при этом $i < j$, то увеличиваем ответ на единицу.

Корректность этого решения следует из приведенных выше рассуждений; при этом мы полагаемся на то, что все входные числа различны, поэтому учитывать повторяющиеся пары не требуется. Временная сложность, помимо сортировки за $O(n \log n)$, составляет $O(n)$, поскольку каждый индекс проходит сквозь весь массив только один раз.

> ***Примечание о наивной и точной оценках асимптотики***
>
> Алгоритм двух указателей, такой, как описанный выше, — это тот случай, когда «наивная» оценка сложности далека от реальной временно́й сложности. Два вложенных цикла, каждый из которых совершает n операций в худшем случае, могут создать ошибочное впечатление, что весь алгоритм будет работать за квадратичное время. В реальности, как было показано выше, время работы этих двух вложенных циклов — линейное.

Реализация

Python

```
n, k = map(int, input().split())
a = list(map(int, input().split()))
a.sort()
j = n - 1
ans = 0
for ai in a:
  while j > 0 and a[j] > k - ai:
    j -= 1
  if ai + a[j] == k and ai < a[j]:
    ans += 1
print(ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  sort(a.begin(), a.end());
  int ans = 0;
  for (int i = 0, j = n - 1; i < n;
       ++i) {
    while (j > 0 &&
           a[j] > k - a[i]) {
      --j;
```

```
    }
    if (a[i] + a[j] == k && i < j)
      ++ans;
  }
  cout << ans << endl;
}
```

13.2. Пример решения задачи. Длиннейший подотрезок без повторов

Задача. Дана последовательность целых чисел a_i ($i = 1...n$). Требуется найти его подотрезок наибольшей длины такой, что все элементы в нем — различны.

Входные данные состоят из числа n в первой строке и n разделенных пробелами чисел a_i. Все числа целые, $1 \le n \le 5 \cdot 10^5$, $-10^9 \le a_i \le 10^9$. Вывести требуется единственное число — длину искомого подотрезка.

Примеры

Входные данные	**Требуемый результат**
10 1 3 4 2 1 5 4 4 4 2	5
Входные данные	**Требуемый результат**
4 4 4 4 4	1

Пояснения к примерам

В первом примере длиннейший подотрезок без повторов — начиная со второго и по шестой элемент, т. е. [3, 4, 2, 1, 5], поэтому искомая длина равна 5. Во втором примере искомая длина равна 1, поскольку любой подотрезок большей длины содержит повторяющиеся числа.

Решение

Наивное решение было бы следующим: перебрать стартовый индекс $i = 1...n$ и для каждого значения i перебирать конечный индекс $j = i...n$, поддерживая в какой-либо структуре данных все содержащиеся в подотрезке $[i; j]$ числа и проверяя добавляемое число на наличие дубликата. Это решение было бы слишком медленным, поскольку даже при использовании типичной эффективной структуры данных «множество» (set) итоговая сложность такого наивного алгоритма составила бы $O(n^2 \log n)$.

Воспользуемся методом двух указателей для ускорения этого решения. Для этого заметим, что с ростом стартового индекса i также растут (или, как минимум, не убывают) соответствующие конечные индексы j. В самом деле, если для индекса i отрезок $[i; j]$ не содержал дублей, то не будет их содержать и отрезок $[i + 1; j]$. Таким образом, верно следующее решение: заведем указатель j, изначально указы-

вающий на позицию до первого элемента, а также структуру данных «множество», изначально пустую. Далее совершим n итераций, передвигающих указатель $i = 1...n$, и на каждой итерации будем продвигать вправо индекс j и добавлять в множество число a_j, до тех пор пока не встретится дубль. В конце каждой итерации будем обновлять ответ длиной текущего подотрезка — т. е. $j - i + 1$ — и удалять число a_i из множества.

Асимптотическая сложность данного решения составит $O(n \log n)$, поскольку каждый из указателей пройдет через n различных значений, и столько же будет совершено операций «добавить в множество», «удалить из множества».

Реализация

Python

```
n = int(input())
a = list(map(int, input().split()))
ans = 0
j = -1
segment = set()
for i in range(n):
  while (j + 1 < n and
         a[j + 1] not in segment):
    j += 1
    segment.add(a[j])
  ans = max(ans, j - i + 1)
  segment.remove(a[i])
print(ans)
```

C++

```
#include <algorithm>
#include <iostream>
#include <set>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  set<int> segment;
  for (int i = 0, j = -1; i < n;
       ++i) {
    while (j + 1 < n &&
           !segment.count(
              a[j + 1])) {
      j += 1;
      segment.insert(a[j]);
    }
    ans = max(ans, j - i + 1);
    segment.erase(a[i]);
  }
  cout << ans << endl;
}
```

13.3. Пример решения задачи. Подотрезки со всеми числами

Задача. Дана последовательность a_i ($i = 1...n$) целых чисел, состоящая только из чисел из отрезка $[1; k]$. Назовем подотрезок хорошим, если он содержит каждое из

этих чисел из отрезка [1; k] хотя бы по одному разу. Требуется найти длину самого короткого хорошего подотрезка.

Входной файл состоит из чисел n и k в первой строке и последовательности из n чисел a_i во второй. Все числа — натуральные; $n \le 5 \cdot 10^5$, $k \le n$. Гарантируется, что последовательность a_i содержит каждое из первых k натуральных чисел хотя бы по одному разу и не содержит никаких других чисел. Вывести требуется искомую длину.

Примеры

Входные данные	**Требуемый результат**
9 4 1 2 1 3 1 2 1 4 1	5
Входные данные	**Требуемый результат**
3 1 1 1 1	1

Пояснения к примерам

В первом примере искомый подотрезок — «3 1 2 1 4». Во втором примере — «1» (в любой из трех стартовых позиций).

Решение

Решать будем методом двух указателей: первый указатель будет пробегать всевозможные стартовые позиции, второй — соответствующие конечные позиции. При увеличении стартовой позиции соответствующая конечная позиция увеличивается или не изменяется, что и позволяет получить линейное в сумме число передвижений. Для проверки подотрезков на «хорошесть» будем поддерживать число вхождений каждого числа и текущее количество чисел с ненулевым количеством.

Асимптотическая сложность такого решения составит $O(n)$.

Реализация

Python

```
n, k = map(int, input().split())
a = list(map(int, input().split()))
cnt = [0] * (k + 1)
present = 0
j = -1
ans = n
for i in range(n):
  while j + 1 < n and present < k:
    j += 1
    if not cnt[a[j]]:
      present += 1
    cnt[a[j]] += 1
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i) {
    cin >> a[i];
    --a[i];
  }
```

```
  if present == k:
    ans = min(ans, j - i + 1)
  cnt[a[i]] -= 1
  if not cnt[a[i]]:
    present -= 1
print(ans)
```

```
  int ans = n;
  vector<int> cnt(k);
  int present = 0;
  for (int i = 0, j = -1; i < n;
       ++i) {
    while (j + 1 < n &&
           present < k) {
      ++j;
      if (!cnt[a[j]])
        ++present;
      ++cnt[a[j]];
    }
    if (present == k)
      ans = min(ans, j - i + 1);
    --cnt[a[i]];
    if (!cnt[a[i]])
      --present;
  }
  cout << ans << endl;
}
```

13.4. Пример решения задачи. Трехцветный забор

Задача. Два маляра красили забор, один в красный цвет, другой — в синий. Известно, какие доски забора были покрашены каждым из маляров: эта информация представлена в виде набора отрезков с номерами досок; доски забора пронумерованы слева направо. Требуется узнать, сколько досок получились фиолетового цвета, если доска приобретает фиолетовый цвет при покраске красным и синим цветом.

Входные данные содержат описания работы обоих маляров, где каждое описание представляет собой число отрезков в первой строке и индексы досок — концов отрезков в последующих строках. Гарантируется, что отрезки одного маляра не пересекаются между собой. Число отрезков в каждом из описаний не превосходит $5 \cdot 10^5$, индексы досок лежат в отрезке $[1; 10^9]$, и первый индекс в описании каждого отрезка не превосходит второй.

Примеры

Входные данные	Требуемый результат
2 6 9 1 4 3 2 3 8 9 5 6	5

Входные данные	Требуемый результат
1 1 9 1 21 29	0

Пояснения к примерам:

В первом примере доски, покрашенные обоими малярами, имеют номера №2, 3, 6, 8, 9 — всего 5 штук. Во втором примере пересечений между досками нет.

Решение

Для эффективного — неквадратичного решения этой задачи упорядочим отрезки из описаний каждого из маляров и будем постепенно обрабатывать их в порядке увеличения индексов. Будем перебирать отрезки первого маляра и для каждого искать пересечения с отрезками второго маляра. Заметим, что нам не требуется возвращаться назад: если мы перешли от одного отрезка первого маляра к следующему, то пересечение может быть только с текущим или одним из следующих отрезков второго маляра.

Это ключевое наблюдение позволяет использовать метод двух указателей: один указатель будет двигаться по отрезкам первого маляра, а другой — по отрезкам второго маляра, которые являются кандидатами на наличие пересечения.

Временна́я сложность этого алгоритма составит $O(n \log n + m \log m)$, где n и m — количества отрезков в описаниях работы первого и второго маляров. Эта асимптотическая сложность образуется из времени двух сортировок $O(n \log n)$ и $O(m \log m)$, а также суммарного времени $O(n + m)$ обработки двумя указателями.

Реализация

Python

```
def read_segms():
  n = int(input())
  segms = []
  for _ in range(n):
    l, r = map(
        int, input().split())
    segms.append((l, r))
  segms.sort()
  return segms

a = read_segms()
b = read_segms()
bptr = 0
ans = 0
for acur in a:
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<pair<int, int>> segm[2];
  for (int i = 0; i < 2; ++i) {
    int n;
    cin >> n;
    segm[i].resize(n);
    for (int j = 0; j < n; ++j) {
      cin >> segm[i][j].first
          >> segm[i][j].second;
    }
    sort(segm[i].begin(),
```

```
  bptr = max(0, bptr - 1)
  while (bptr < len(b) and
         b[bptr][0] <= acur[1]):
    bcur = b[bptr]
    inters_left = max(
        acur[0], bcur[0])
    inters_right = min(
        acur[1], bcur[1])
    ans += max(0, inters_right -
                  inters_left + 1)
    bptr += 1
print(ans)
```

```
        segm[i].end());
  }
  auto& a = segm[0];
  auto& b = segm[1];
  int ans = 0;
  int bptr = 0;
  for (auto acur : a) {
    bptr = max(0, bptr - 1);
    while (bptr < (int)b.size() &&
           b[bptr].first <=
               acur.second) {
      int inters_left = max(
          acur.first,
          b[bptr].first);
      int inters_right = min(
          acur.second,
          b[bptr].second);
      ans += max(0, inters_right -
                    inters_left +
                    1);
      ++bptr;
    }
  }
  cout << ans << endl;
}
```

Примечание по обновлению движущихся указателей

В приведенных выше реализациях на каждой итерации мы возвращаем второй указатель на один шаг назад. Это чисто реализационная тонкость: таким образом удобно обработать случай, когда один отрезок второго маляра пересекается сразу с несколькими отрезками первого. В нашей реализации в момент выхода из вложенного цикла второй указатель указывает на отрезок второго маляра, не пересекающийся с текущим отрезком первого. Уменьшение указателя на следующей итерации внешнего цикла позволяет рассмотреть отрезок второго маляра повторно.

Глава 14.
Двоичный поиск

14.1. Базовая задача: поиск в упорядоченном массиве

Базовой задачей, решаемой алгоритмом двоичного поиска (также известном как бинарный поиск или, на жаргоне, «бинпоиск»), является быстрый поиск заданного значения в массиве чисел, упорядоченных по возрастанию (неубыванию).

Двоичный поиск позволяет производить эту операцию без просмотра всего массива. Особое преимущество этого алгоритма проявляется в случаях, когда для одного и того же массива поступает большое количество запросов поиска.

> ***Примечание об альтернативных способах решения***
>
> Эту базовую задачу можно решить и другими способами, например, построив красно-черное дерево и совершая обходы по нему. Однако двоичный поиск имеет преимущества очень простой реализации и малой скрытой константы. Кроме того, мощь двоичного поиска — в возможности его обобщения на множество других задач.

14.2. Алгоритм двоичного поиска

Основная идея алгоритма двоичного поиска базируется на том, что можно вдвое сузить диапазон поиска, если известен результат сравнения со средним элементом.

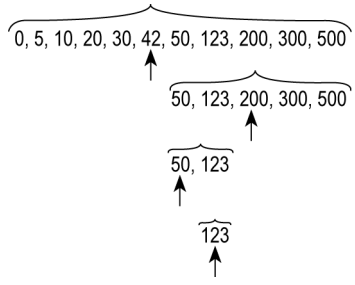

Рис. 14.1. Пример поиска числа 123 в заданном массиве с помощью алгоритма двоичного поиска

Например, если в упорядоченном массиве ищется число 123, а средний элемент массива равен 42, то искомое число может находиться только в правой половине

массива, но не в левой. Если же в том же массиве ищется, например, число 12, то оно может присутствовать только в левой половине (рис. 14.1).

Раз за разом повторяя этот прием, можно очень быстро сократить диапазон поиска до одного элемента, для чего потребуется лишь порядка $O(\log n)$ операций. После этого ответ на задачу определяется тривиальным образом.

14.3. Реализация алгоритма двоичного поиска

Примечание о реализациях в стандартных библиотеках

Большинство современных языков программирования уже включают в состав стандартной библиотеки реализации двоичного поиска. Однако следует уметь реализовывать этот алгоритм и вручную, поскольку зачастую встроенные реализации хорошо годятся только для решения типовых задач, а нестандартные задачи требуют несколько видоизмененной реализации.

Типичные реализации алгоритма двоичного поиска оперируют двумя индексами: левым l и правым r концами поискового диапазона. Изначально этот диапазон покрывает весь массив, а затем каждый шаг алгоритма двоичного поиска уменьшает его примерно вдвое. Алгоритм завершает свою работу, когда l становится равным r.

Python

```
def bin_search(a, x):
  left = 0
  right = len(a) - 1
  while left < right:
    mid = (left + right) // 2
    if a[mid] < x:
      left = mid + 1
    else:
      right = mid
  return (left < len(a) and
          a[left] == x)
```

C++

```
bool BinSearch(
  const vector<int>& a, int x) {
  int left = 0;
  int right = (int)a.size() - 1;
  while (left < right) {
    int mid = (left + right) >> 1;
    if (a[mid] < x)
      left = mid + 1;
    else
      right = mid;
  }
  return left < (int)a.size() &&
         a[left] == x;
}
```

Примечание об ошибках +−1 и зависаниях

Каверзный момент при реализации двоичного поиска — это зависание алгоритма в момент $r = l + 1$ при ошибочной реализации. А именно, если (неправильно) реализовать алгоритм как «if … l = m … else … r = m», то в указанной ситуации этот неверный код будет стоять на одном и том же месте. Существуют и другие популярные способы решения этой проблемы, например: останавливать двоичный поиск, когда $r - l < 2$ (или меньше какой-либо другой небольшой константы).

Примечание о целочисленных переполнениях

Представленная выше реализация на C++ некорректно обработает случай, когда сумма индексов `left` и `right` не помещается в тип данных int. В данной задаче это маловероятный сценарий,

поскольку в случае 32-битного типа для этого потребовался бы массив размером более гигабайта, а в случае 64-битного типа и вовсе невероятно большого размера. Тем не менее это надо будет иметь в виду при обобщениях на другие задачи.

Примечание об использовании битовых операций для деления на 2

В реализации на C++ мы используем операцию битового сдвига вправо (`>> 1`) вместо явного деления на два. В данном случае конечный результат одинаков, однако операция битового сдвига работает быстрее операции деления. По состоянию на 2021 г. типичные компиляторы попрежнему не умеют производить эту оптимизацию в данном коде автоматически.

14.4. Библиотечные реализации

Многие современные языки программирования уже содержат стандартные реализации алгоритма двоичного поиска в своей стандартной библиотеке:

- в языке Python пакет bisect содержит функцию `bisect()`, выполняющую двоичный поиск и возвращающую позицию наименьшего элемента $a_i > x$. Также этот модуль содержит функцию `bisect_left()` для поиска наименьшего элемента $a_i \geq x$ и несколько других вспомогательных функций;
- в языке C++ заголовочный файл <algorithm> предоставляет функцию `binary_search()`, проверяющую наличие указанного элемента x с помощью двоичного поиска. Также он содержит функцию `lower_bound()` для поиска наименьшего элемента $a_i \geq x$ и `upper_bound()` для поиска наименьшего элемента $a_i > x$.

Эти реализации, как правило, надежны и быстры, поэтому при решении задач есть смысл по возможности их и использовать. Однако это не отменяет того факта, что есть множество менее стандартных задач на двоичный поиск, где применение библиотечной реализации будет сложнее написания своей собственной. Поэтому понимать, как работает двоичный поиск, и уметь реализовывать его вручную — очень важный навык.

14.5. Пример решения задачи. Подсчет меньших чисел

Задача. Дан массив целых чисел a_i ($i = 1...n$). Требуется научиться отвечать на запросы «подсчитать количество a_i, меньших заданного x».

Первая строка входных данных содержит n (целое число в диапазоне от 1 до 10^5). Вторая строка содержит разделенные пробелами a_i (целые числа, по модулю не превосходящие 10^9). Третья строка содержит количество m запросов (целое число в диапазоне от 1 до 10^5). Сами запросы даны в четвертой строке (разделенные пробелами целые числа, по модулю не превосходящие 10^9). Вывести требуется последовательность из m разделенных пробелами чисел — ответы на запросы.

Примеры

Входные данные	Требуемый результат
4 1 3 2 4 5 1 2 3 4 5	0 1 2 3 4
Входные данные	**Требуемый результат**
3 1 -1 1 2 0 123	1 3

Решение

Отсортируем массив a_i, тогда ответ на запрос можно искать непосредственным применением двоичного поиска.

Реализация

Python

```
from bisect import *
n = int(input())
a = list(map(int, input().split()))
m = int(input())
q = map(int, input().split())
a.sort()
for query in q:
  print(bisect_left(a, query),
        end=" ")
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  sort(a.begin(), a.end());
  int m;
  cin >> m;
  for (int i = 0; i < m; ++i) {
    int query;
    cin >> query;
    int ans = int(lower_bound(
      a.begin(), a.end(), query) -
      a.begin());
    cout << ans << ' ';
  }
}
```

14.6. Пример решения задачи. Грузовой лифт в отеле

Задача. В огромном отеле постоянно возникают сложные логистические задачи. Одна из задач заключается в планировании работы грузовых лифтов для доставки различных грузов. Перед тем как приступить к планированию, руководство поручило написать вам вспомогательную программу, определяющую номер этажа по поступающему на вход номеру комнаты. Эта задача не так проста, так как из-за стремления руководства к оригинальности нумерация комнат — сквозная и без пропусков. А именно, комнаты на первом этаже имеют номера от 1 до a_1, комнаты на втором этаже — от $a_1 + 1$ до $a_1 + a_2$ и т. д.; таким образом, i-й этаж содержит комнаты с номерами:

$$\left[1+\sum_{j=1}^{i-1} a_j ; \sum_{j=1}^{i} a_j\right].$$

Первая строка входного файла содержит n — число этажей, которое является натуральным числом, не превосходящим 10^5. Вторая строка содержит n чисел a_i, разделенных пробелами (числа не превосходят 10^5). Третья строка содержит m — число запросов $1 \le m \le 10^5$, четвертая строка — сами запросы, т. е. m разделенных пробелами натуральных чисел (числа не превосходят 10^{10}). Вывести требуется m целых чисел через пробел: ответы на запросы, т. е. номер этажа комнаты с номером a_i, либо −1, когда такой комнаты нет.

Примеры

Входные данные	**Требуемый результат**
4 10 10 10 10 6 1 5 10 15 20 25	1 1 1 2 2 3
Входные данные	**Требуемый результат**
2 5 10 4 14 15 16 17	2 2 -1 -1

Решение

Решение «в лоб», с последовательным перебором этажей при обработке каждого запроса будет недостаточно быстрым: в худшем случае оно потребует $O(nm)$ операций.

Посчитаем предварительно для каждого этажа номер первой комнаты на этом этаже. Если мы обозначим через b_i наименьший из номеров комнат на этаже i, то процедуру его вычисления можно описать следующим образом:

$$b_1 = 1,$$
$$b_i = b_{i-1} + a_{i-1} \quad (i = 2...n).$$

Иными словами, последовательность b_i — это префиксные суммы массива a. Тогда поиск ответа на запрос x сводится к тому, чтобы найти наибольшее i такое, что $b_i \leq x$. Поскольку последовательность b_i — строго возрастающая, то здесь можно непосредственно применить алгоритм двоичного поиска. Таким образом, ответ на один запрос будет вычисляться за $O(\log n)$, а вся задача будет решена за $O(m \log n)$.

Альтернативное решение

Эту задачу можно решить и другим способом, без двоичного поиска, если заранее отсортировать запросы по неубыванию и применить метод двух указателей. Впрочем, это решение обладало бы тем недостатком, что все запросы должны были быть известны заранее — в то время как алгоритм на основе двоичного поиска работает с одинаковой эффективностью вне зависимости от знания запросов заранее (т. е. является «онлайн»-алгоритмом). Кроме того, в реализации решения на основе двоичного поиска чуть труднее допустить ошибку благодаря использованию библиотечных функций.

Реализация

Python

```
from bisect import *
from itertools import *
n = int(input())
a = list(map(int, input().split()))
m = int(input())
q = map(int, input().split())
b = list(accumulate([1] + a))
for query in q:
  floor = bisect(b, query)
  print(floor if floor <= n
        else -1, end=" ")
```

C++

```
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<int64_t> b(n + 1);
  b[0] = 1;
  for (int i = 0; i < n; ++i)
    b[i + 1] = b[i] + a[i];
  int m;
  cin >> m;
  for (int i = 0; i < m; ++i) {
    int q;
    cin >> q;
    int floor = int(upper_bound(
```

```
      b.begin(), b.end(), q) -
      b.begin());
    cout << (floor <= n ?
             floor : -1) << ' ';
  }
}
```

14.7. Пример решения задачи. Дисплеи для смартфонов

Задача. Перед фирмой-производителем смартфонов стоит непростая задача. В следующем году запланирован выпуск n новых моделей, однако из-за логистических трудностей имеется лишь ограниченный объем доступных дисплеев. Всего есть k типов дисплеев, и i-го типа имеется лишь c_i штук в доступности. Требуется определить наибольшее число смартфонов, которое можно будет произвести, если должно быть выпущено одинаковое число смартфонов каждой модели и все смартфоны одной модели должны обладать дисплеем одного и того же типа.

Входные данные содержат разделенные пробелом числа n и k в первой строке и разделенные пробелом k чисел c_i — во второй. Все числа — натуральные, n и k не превосходят 10^5, c_i — 10^9. Вывести требуется единственное число — максимально возможное общее число смартфонов.

Примеры

Входные данные	**Требуемый результат**
3 4 3 4 5 6	12
Входные данные	**Требуемый результат**
6 3 4000 8000 12 000	24 000

Пояснения к примерам

В первом примере оптимальное решение — выпустить по 4 смартфона каждой из трех моделей, использовав для каждой по 4 дисплея типов №2, 3, 4 соответственно.

Во втором примере можно использовать все 24 000 дисплеев для выпуска смартфонов, по 4000 смартфонов каждой из 6 моделей. Например, первая модель должна использовать дисплеи типа №1, вторая и третья модели — дисплеи типа №2 и четвертая, пятая и шестая модели — дисплеи типа №3.

Решение

Поскольку смартфонов каждой модели должно быть выпущено одинаковое количество, удобнее искать ответ как максимизацию этого количества; обозначим его через x.

Переформулируем теперь условие задачи в математических терминах. Заметим, что для каждого типа дисплеев i можно определить количество моделей, которое можно обеспечить этими дисплеями, — оно равно $\lfloor c_i / x \rfloor$. Просуммировав эти величины, мы должны получить, как минимум, n — это будет критерием того, что мы можем обеспечить дисплеями все запланированные модели. Таким образом, задача свелась к тому, чтобы найти максимальное x такое, что:

$$\sum_{i=1}^{k} \left\lfloor \frac{c_i}{x} \right\rfloor \geq n.$$

Эту задачу будем решать с помощью двоичного поиска. Изначально диапазон поиска x равен $[1; \max_{i=1\ldots k} c_i]$. Далее мы начинаем итерации двоичного поиска, на каждой из которых мы вычисляем левую часть неравенства для середины текущего диапазона поиска и сравниваем результат с n: если результат меньше n, то поиск нужно продолжить в левой половине поискового диапазона, иначе — в правой.

Временна́я сложность этого решения образуется из числа итераций двоичного поиска — которых не более $\log_2 \max_{i=1\ldots k} c_i$ — и сложности вычисления приведенной выше суммы — которая составляет k. Таким образом, итоговая асимптотическая сложность получается равной $O(k \log \max_{i=1\ldots k} c_i)$.

Реализация

Python

```
n, k = map(int, input().split())
c = list(map(int, input().split()))

def is_good(x):
  if not x:
    return True
  return sum(citem // x
             for citem in c) >= n

left = 0
right = max(c)
while left < right:
  mid = (left + right + 1) // 2
  if is_good(mid):
    left = mid
  else:
    right = mid - 1
print(left * n)
```

C++

```
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> c(k);
  for (int i = 0; i < k; ++i)
    cin >> c[i];
  int left = 0;
  int right = *max_element(
    c.begin(), c.end());
  while (left < right) {
    int mid = (left + right +
               1) >> 1;
    int64_t sum = 0;
    if (mid) {
      for (int i = 0; i < k; ++i)
        sum += c[i] / mid;
    }
    if (!mid || sum >= n)
```

```
      left = mid;
    else
      right = mid - 1;
  }
  cout << int64_t(left) * n
       << endl;
}
```

Примечание о слагаемых +−1 в двоичном поиске

Как и в исходной реализации двоичного поиска, мы должны быть аккуратными в том, как мы обновляем границы поискового интервала, чтобы предотвратить зависание на каких-либо входных данных. Поскольку в этой задаче критерии обновления границ несколько другие — левая граница сдвигается на середину включительно, а правая не включительно — то и индексы вычисляются по-другому: середина сдвигается при нечетной сумме вправо на единицу. Как и в исходной реализации, критерием для самопроверки является то, что код не зависает в случае `left=0`, `right=1`.

Ранее был упомянут и альтернативный способ решения этой проблемы — остановка двоичного поиска на одну-две итерации раньше окончания и выбор лучшего значения простым перебором среди оставшихся элементов. На наш взгляд, преимущество этого альтернативного метода — отсутствие необходимости в аккуратном выборе слагаемых +−1 — перевешивается недостатком большей многословности.

14.8. Пример решения задачи. Прыжки лягушки

Задача. Лягушка находится в точке с координатой 0 и прыгает вдоль координатной оси (в направлении увеличения) на целочисленные расстояния. Она хочет попасть в точку с координатой n не более чем за k прыжков. При этом некоторые точки являются запрещенными для приземления: если $a_i = 1$, то точка с координатой i разрешена для приземления, а если $a_i = 0$ — запрещена ($i = 1...n - 1$). Кроме того, лягушка пытается избегать длинных прыжков, т. е. требуется найти такой маршрут, который бы минимизировал длину самого длинного прыжка.

Входные данные содержат натуральные числа n и k в первой строке ($2 \le n \le 10^5$) и строку из $n - 1$ символов a_i (каждый символ — ноль или единица) во второй. Вывести требуется длину самого длинного прыжка в искомом маршруте.

Примеры

Входные данные	**Требуемый результат**
10 2 001001000	6
Входные данные	**Требуемый результат**
5 5 0000	5

Пояснения к примерам

В первом примере оптимальным является следующий маршрут: из $x = 0$ прыгнуть в $x = 6$ и оттуда прыгнуть в конечную точку $x = 10$. Этот маршрут содержит два прыжка, один длиной 6, другой — 4. Для сравнения, этот маршрут лучше маршрута ($x = 0$, $x = 3$, $x = 10$), поскольку этот маршрут имел бы прыжок длиной 7.

Во втором примере запрещено приземляться в любую из промежуточных точек, поэтому единственное решение — сразу прыгнуть в конечную точку, и ответ равен входному n.

Решение

Начнем с решения «в лоб» — перебирать длину длиннейшего прыжка и проверять каждую из длин на то, можно ли достичь конечной точки не более чем за k прыжков длины, ограниченной текущим числом. Для проверки достижимости будем жадно прыгать каждый раз в самую дальнюю из достижимых разрешенных точек — ведь, в самом деле, мы ничего не выиграем от того, что мы недопрыгнем до какой-нибудь точки. Это наивное решение имело бы асимптотику $O(n^2)$, поскольку есть n возможных длин прыжков, и проверка каждой потребовала бы прохода по всему маршруту.

Для ускорения этого решения заметим, что функция того, можно ли достичь конечной точки с заданным ограничением, монотонна. В самом деле, если прыжки чересчур коротки (короче, чем требуется для достижения конечного пункта за k прыжков), то функция всегда будет говорить «пути нет». И, наоборот, если ограничение больше или равно ответу, то функция всегда будет возвращать «путь есть».

К таким монотонным функциям можно применить метод двоичного поиска (то, что функция может принимать всего два значения, ничего не меняет). Мы начинаем с изначальным поисковым диапазоном $[1; n]$ и на каждой итерации пытаемся достичь конечной точки с ограничением, равным середине текущего поискового диапазона. Если достичь конечную точку удалось, то мы продолжаем поиск в меньшей половине диапазона, иначе — в большей.

Асимптотика этого решения составит $O(n \log n)$, поскольку двоичный поиск совершит примерно $\log_2 n$ итераций, а каждая итерация совершит проход по всему маршруту.

Реализация

Python

```
n, k = map(int, input().split())
a = input()

def is_good(max_jump):
  x = 0
  count = 0
  while x < n:
    count += 1
    for jump in range(
        max_jump, 0, -1):
```

C++

```
#include <iostream>
#include <string>
using namespace std;

int n, k;
string a;

bool IsGood(int max_jump) {
  int x = 0, count = 0;
  while (x < n) {
```

```
      nx = x + jump
      if (nx >= n or
          a[nx - 1] == '1'):
        x = nx
        break
    else:
      return False
  return count <= k

left = 1
right = n
while left < right:
  mid = (left + right) // 2
  if is_good(mid):
    right = mid
  else:
    left = mid + 1
print(left)
```

```
    ++count;
    int nx = x + max_jump;
    while (nx > x) {
      if (nx >= n ||
          a[nx - 1] == '1')
        break;
      --nx;
    }
    if (x == nx)
      return false;
    x = nx;
  }
  return count <= k;
}

int main() {
  cin >> n >> k >> a;
  int left = 1, right = n;
  while (left < right) {
    int mid = (left + right) / 2;
    if (IsGood(mid))
      right = mid;
    else
      left = mid + 1;
  }
  cout << left << endl;
}
```

14.9. Пример решения задачи. Корень уравнения

Задача. Подающая надежды школьница Маша ходит в кружок по математике. Александр, как руководитель кружка по информатике, уверен, что ее ждут бóльшие успехи на поприще спортивного программирования, и хочет переманить ее к себе. Первый шаг уже сделан: Александр рассказал Маше, что математика бессильна перед уравнениями пятой и более степеней, т. е. что они неразрешимы в радикалах (конечно, Александр несколько сгущает краски насчет выводов из этого факта, однако оставим это на его совести). Для окончательной «победы» осталось продемонстрировать Маше, как компьютер может с легкостью решать различные уравнения пятой степени. Помогите Александру — напишите программу, которая ищет один из корней следующего уравнения:

$$x^5 - 10x + a = 0.$$

Входной файл содержит единственное целое число a, по модулю не превосходящее 100. Выведите одно число — корень указанного уравнения; если корней несколько,

вывести можно любой. Ответ должен выводиться с такой точностью, чтобы левая часть уравнения отличалась от нуля не более чем на 10^{-8}.

Примеры

Входные данные	Требуемый результат
0	0
Входные данные	**Требуемый результат**
1	0.100001

Пояснения к примерам

В первом примере уравнение выглядит $x^5 - 10x = 0$, и оно имеет точное решение — один из его корней $x = 0$. Во втором примере все корни иррациональны; приведенный ответ 0.100001 — приблизителен: при подстановке его в уравнения левая часть отличается от нуля приблизительно на $5 \cdot 10^{-10}$.

Решение

Хотя есть целый набор специализированных методов численного решения уравнений, у двоичного поиска есть свои преимущества. Во-первых, он весьма универсален, требуя от функции только непрерывности, а от начальных параметров — только знания двух точек, на которых функция имеет разный знак. Во-вторых, в контексте спортивного программирования двоичный поиск удобен своей простотой и краткостью реализации.

Итак, как применить двоичный поиск к решению уравнений? Пусть уравнение имеет вид $f(x) = 0$, где $f(x)$ — непрерывная функция, и известны две точки l и r, на которых функция $f(x)$ принимает разный знак: $\operatorname{sgn} f(l) \neq \operatorname{sgn} f(r)$. Заметим, что отсюда следует, что на отрезке $[l; r]$ есть, как минимум, один корень; возможно, их несколько, если функция меняет знак несколько раз.

Метод двоичного поиска, или деления пополам, заключается в определении знака функции в середине текущего поискового отрезка, т. е. $m = (l + r) / 2$. В зависимости от знака $f(m)$ можно утверждать, что корень есть в левой или в правой половине поискового отрезка:

- Если $\operatorname{sgn} f(m) = \operatorname{sgn} f(l)$, то корень есть в отрезке $[m; r]$.
- Если $\operatorname{sgn} f(m) = \operatorname{sgn} f(r)$, то корень есть в отрезке $[l; m]$.

Используя этот критерий, мы сужаем поисковый критерий вдвое. Повторяя эти итерации, мы постепенно приближаемся к одному из корней.

Последний вопрос — когда останавливать итерации двоичного поиска? В отличие от предыдущих задач, применявших двоичный поиск к ограниченным множествам целых чисел, в случае вещественных чисел нет никакого естественного критерия для остановки двоичного поиска: при желании его можно выполнять бесконечно, все ближе приближая границы поискового диапазона к истинному корню.

На практике в спортивном программировании обычно хорошо работает следующий подход: совершать определенное число операций, которое жестко закодировано в коде в виде константы. В большинстве задач подходит константа порядка 100. Эта константа обозначает, что стартовый поисковый интервал сужается к концу работы алгоритма в 2^{100} раз, что в десятичной системе счисления составляет приблизительно 10^{30} раз. Этого более чем достаточно в типичных задачах.

> ***Альтернативные подходы к выбору числа итераций***
>
> В редких случаях требуется более тонкая настройка двоичного поиска — увеличение константы (для достижения большей точности), или уменьшение константы (для ускорения программы), или замена константы критериями на основе абсолютной или относительной точности (для ускорения в зависимости от конкретного входного набора). В таких случаях детальной настройки полезно иметь некоторое представление об ошибках округления при работе типов с плавающей запятой — неверно полагать, что погрешность выводимого программой ответа равна длине поискового интервала двоичного поиска.

В заключение вернемся к решаемой нами задаче, в которой $f(x) = x^5 - 10x + a$. Легко убедиться, что двоичный поиск применим к ней, поскольку функция непрерывна, и легко найти стартовый интервал, на концах которого функция имеет разный знак (это видно из того, что функция положительна при $x \to +\infty$ и отрицательна при $x \to -\infty$). В качестве стартового поискового интервала можно задать $[-10; 10]$: функция принимает на концах этого отрезка значения разного знака при любых возможных по условию значениях a.

Реализация

Python

```
ITER = 100
a = int(input())

def f(x):
  return x ** 5 - 10 * x + a

left = -10
right = 10
for _ in range(ITER):
  mid = (left + right) / 2
  if f(mid) < 0:
    left = mid
  else:
    right = mid
ans = (left + right) / 2
print("%.20f" % ans)
```

C++

```
#include <cmath>
#include <iostream>
using namespace std;

const int ITER = 100;
int a;

double f(double x) {
  return pow(x, 5) - 10 * x + a;
}

int main() {
  cout << fixed;
  cout.precision(20);
  cin >> a;
  double left = -10, right = 10;
  for (int iter = 0; iter < ITER;
       ++iter) {
```

```
    double mid =
        (left + right) / 2;
    if (f(mid) < 0)
      left = mid;
    else
      right = mid;
  }
  double ans = (left + right) / 2;
  cout << ans << endl;
}
```

Примечание о взятии среднего арифметического при выводе

Приведенные выше реализации выводят не значение левой или правой границы двоичного поиска, а их среднее арифметическое. В принципе, с учетом большого числа итераций, это может быть излишним — однако в некоторых более сложных задачах эта небольшая поправка может помочь сделать ответ более точным.

14.10. Прочие применения двоичного поиска

Двоичный поиск находит применение и вне сферы спортивного программирования и низкоуровневых операций:

- двоичным поиском можно искать нужную страницу в словаре. Если слова упорядочены в алфавитном порядке, то найти страницу с конкретным словом среди n страниц можно, сначала открыв книгу на середине, затем открыв середину оставшейся части и т. д. — в итоге за $\approx \log_2 n$ шагов;
- другая сфера применений — поиск первого «плохого» состояния в системе контроля версий. Например, тестировщики обнаружили ошибку в поведении программного продукта, и теперь требуется найти, с каким изменением в исходном коде была привнесена эта ошибка. Предполагая, что ошибка была внедрена один раз и ни разу с тех пор не удалена, это можно сделать за $\approx \log_2 n$ проверок, где n — количество изменений исходного кода.

Глава 15.
Проверка на простоту и факторизация

В этой главе речь пойдет о базовых алгоритмах для проверки чисел на простоту и о разложении чисел на простые множители.

15.1. Определения

Простым натуральное число называется тогда, когда оно больше 1 и имеет только два делителя: 1 и самого себя. Первые несколько простых чисел: 2, 3, 5, 7, 11, 13, 17.

Составным называется натуральное число, которое больше 1 и не является простым.

Факторизацией натурального числа называется его разложение в произведение простых. Факторизация существует и единственна для любого из чисел (доказательство этого факта называется основной теоремой арифметики и не приводится здесь). Примеры факторизаций: $5 = 5$, $6 = 2 \cdot 3$, $8 = 2^3$.

15.2. Общие сведения о простых числах и о факторизации

- Простых чисел бесконечно много.
- Единственное четное простое число — 2.
- Число 1 не является ни простым, ни составным.
- Самое большое известное простое число равно $2^{82589933} - 1$ (по состоянию на 2021 г.). Это одно из чисел Мерсенна, т. е. чисел вида $2^n - 1$, и это простое число было обнаружено в 2018 г. с помощью огромного вычислительного кластера.
- Количество простых чисел, не превосходящих n, обозначается как $\pi(n)$ и приблизительно оценивается как $n / \ln(n)$. Таким образом, длина интервала между простым p и следующим простым составляет в среднем $\ln(p)$; в то же время известно, что есть бесконечное множество пар простых чисел, расстояние между которыми не превосходит 246.
- Несмотря на простоту определения и основополагающую важность для многих разделов математики, многие факты касательно простых чисел остаются невыясненными:
  - неизвестны очень эффективные способы проверки числа на простоту или его факторизации;

- неизвестна «формула для простых чисел», т. е. какая-либо математическая конструкция, которая позволила бы относительно легко вычислить *k*-е по счету простое число;
- неизвестно, бесконечно ли много «простых близнецов», т. е. таких *i*, что и *i*, и $i + 2$ являются простыми;
- неизвестно, всякое ли четное число можно представить в виде суммы двух простых (эта задача называется «проблемой Гольдбаха»).

15.3. Проверка числа на простоту. Базовый алгоритм

Задача. Дано натуральное число $n \le 10^{12}$. Требуется определить, простое ли оно, и вывести «YES» или «NO» в зависимости от этого.

Примеры

Входные данные	**Требуемый результат**
19	YES
Входные данные	**Требуемый результат**
1	NO

Решение

Оговоримся: здесь мы рассмотрим базовый алгоритм проверки на простоту; впрочем, этот алгоритм достаточен для абсолютного большинства задач спортивного программирования.

Наивная реализация, непосредственно следующая математическому определению, была бы следующей: перебрать все числа в отрезке $[2; n - 1]$ и вычислить остаток от деления *n* на это число. Такой наивный алгоритм имел бы сложность $O(n)$.

Оптимизируем этот алгоритм, заметив следующий факт: достаточно проверить на делимость лишь числа в следующем отрезке:

$$\left[2; \left\lfloor \sqrt{n} \right\rfloor\right].$$

В самом деле, если *n* делится на какое-либо *i*, то оно делится и на n / i, а, значит, для всякого делителя, превосходящего $\sqrt{n}$, найдется и делитель, меньший $\sqrt{n}$. Эта оптимизация позволяет ускорить алгоритм до $O(\sqrt{n})$.

Потенциальные дальнейшие оптимизации

Помимо произведенного выше асимптотического ускорения можно было бы также произвести и другие оптимизации, например, итерироваться только по нечетным числам за исключением проверки на четность вначале. Другим вариантом было бы итерироваться только по простым числам, если они найдены заранее каким-либо другим ал-

горитмом. Также возможные оптимизации касаются того, каким образом вычисляется граница перебора $\sqrt{n}$: можно либо вычислить ее заранее, либо сравнивать квадрат текущего делителя на каждой итерации с n, либо аппроксимировать эту границу с помощью битовых операций. Впрочем, в большинстве задач спортивного программирования все эти дополнительные оптимизации не требуются, поэтому мы опустим их в нижеприведенной реализации.

Реализация

Python

```
def is_prime(n):
  if n <= 1:
    return False
  i = 2
  while i * i <= n:
    if n % i == 0:
      return False
    i += 1
  return True
```

C++

```
bool IsPrime(int64_t n) {
  if (n <= 1)
    return false;
  for (int i = 2;
       int64_t(i) * i <= n; ++i) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

15.4. Факторизация числа. Базовый алгоритм

Задача. Дано натуральное число $n \leq 10^{12}$. Требуется вывести его факторизацию, т. е. разложение на простые множители.

Входные данные состоят из числа n. Вывести требуется количество различных простых делителей в первой строке и информацию об этих простых делителях в последующих строках: каждая строка должна содержать простой делитель и, через пробел, степень этого делителя. Делители должны быть выведены в порядке увеличения.

Примеры

Входные данные	**Требуемый результат**
120	3 2 3 3 1 5 1
Входные данные	**Требуемый результат**
19	1 19 1

Пояснения к примерам

В первом примере искомая факторизация — $120 = 2^3 \cdot 3^1 \cdot 5^1$. Во втором примере входное число — простое, поэтому его факторизация состоит из него самого же в первой степени.

Решение

Как и в статье о проверке чисел на простоту, мы рассматриваем здесь лишь базовый алгоритм проверки на простоту — в большинстве задач спортивного программирования этого достаточно.

Воспользуемся той же идеей, что и при проверке числа на простоту: будем перебирать числа вплоть до $\sqrt{n}$ и пытаться делить n на каждое из них; при этом в ответ добавляется этот делитель в степени, равной числу раз, на которое удалось поделить без остатка. Единственная тонкость: если после окончания цикла делимое осталось больше единицы, то это оставшееся число — простой множитель, который надо также добавить в ответ, со степенью 1.

Этот алгоритм работает за время $O(\sqrt{n})$.

Реализация

Python

```
def factor(n):
  fact = {}
  i = 2
  while i * i <= n:
    while n % i == 0:
      fact[i] = fact.get(i, 0) + 1
      n //= i
    i += 1
  if n > 1:
    fact[n] = 1
  return fact
```

C++

```
vector<pair<int, int>> Factor(
    int64_t n) {
  vector<pair<int, int>> fact;
  for (int i = 2; int64_t(i) * i <= n;
       ++i) {
    if (n % i)
      continue;
    fact.emplace_back(i, 0);
    while (n % i == 0) {
      ++fact.back().second;
      n /= i;
    }
  }
  if (n > 1)
    fact.emplace_back(n, 1);
  return fact;
}
```

15.5. Пример решения задачи. Подсчет числа делителей

Задача. Дано натуральное число $n \le 10^{12}$. Требуется вывести количество его различных делителей.

Примеры

Входные данные	Требуемый результат
12	6
Входные данные	**Требуемый результат**
19	2

Пояснения к примерам

В первом примере делителей шесть: 1, 2, 3, 4, 6, 12. Во втором примере входное число — простое, поэтому у него ровно два делителя: оно само и единица.

Решение

Понятно, что перебирать всевозможные делители было бы слишком медленно, поскольку даже без учета входного числа его делители могут принимать бóльшие значения, как $n / 2$ в случае четного n.

Оптимизируем это решение, заметив, что если n делится на какое-либо k, то оно делится и на n / k тоже. Таким образом, нет нужды перебирать делители больше $\sqrt{n}$: достаточно лишь добавлять по два делителя для всякого делителя, который меньше $\sqrt{n}$. Единственный нюанс: если обнаруженный делитель в точности равен $\sqrt{n}$, то во избежание двойного его учитывания ответ надо увеличить только на единицу.

Такое решение имеет асимптотическую сложность $O(\sqrt{n})$.

Реализация

Python

```
def count_divisors(n):
  i = 1
  ans = 0
  while i * i <= n:
    if n % i == 0:
      ans += 1 if i * i == n else 2
    i += 1
  return ans
```

C++

```
int CountDivisors(int64_t n) {
  int res = 0;
  for (int i = 1; int64_t(i) * i <= n;
       ++i) {
    if (n % i == 0) {
      res += int64_t(i) * i == n ?
          1 : 2;
    }
  }
  return res;
}
```

15.6. Пример решения задачи. Иррациональный портной

Задача. Иррациональный портной приобрел n квадратных кусков ткани; i-й кусок имеет площадь s_i. Для работы с этими кусками ткани портному требуются соответствующие измерительные ленты. Измерительная лента считается подходящей для

какого-либо куска ткани, если длина стороны этого куска кратна длине измерительной ленты. Портной хочет определить, какое наименьшее количество различных измерительных лент понадобится ему; то, что некоторые из них должны иметь иррациональную длину, не беспокоит портного.

Входные данные состоят из натурального числа n в первой строке и последовательности из n разделенных пробелом натуральных чисел s_i во второй. Ограничения: $n \le 10^4$, $s_i \le 10^8$. Вывести требуется искомое количество лент.

Примеры

Входные данные	**Требуемый результат**
3 1 2 3 4	3
Входные данные	**Требуемый результат**
3 8 2 32	1

Пояснения к примерам

В первом примере стороны кусков ткани имеют длины 1, $\sqrt{2}$, $\sqrt{3}$, 2; портному достаточно, например, следующего набора лент: 1, $\sqrt{2}$, $\sqrt{3}$. Во втором примере длины сторон равны $2\sqrt{2}$, $\sqrt{2}$ и $4\sqrt{2}$; портному достаточно иметь ленту длиной $\sqrt{2}$ (или, вообще говоря, подходящая длина может быть равна $\sqrt{2}/k$, где k — произвольное натуральное число).

Решение

Попытаемся сформулировать, в каких случаях одна измерительная лента подойдет сразу для двух кусков ткани с номерами i и j. Описание из условия можно выразить математически следующим образом:

$$\begin{cases} x \cdot k_1 = \sqrt{s_i}, \\ x \cdot k_2 = \sqrt{s_j}, \end{cases}$$

где x — длина измерительной ленты; k_1 и k_2 — некоторые натуральные числа. Отсюда можно увидеть, что все зависит от того, что можно вынести из-под знака корня с правой стороны, а что остается под знаком корня. В самом деле, если мы выделим в s_i и s_j полные квадраты:

$$s_i = a^2 b,$$
$$s_j = c^2 d,$$

то вышеприведенную систему можно записать следующим образом:

$$\begin{cases} x \cdot k_1 = \sqrt{a^2 b} = a\sqrt{b}, \\ x \cdot k_2 = \sqrt{c^2 d} = c\sqrt{d}, \end{cases}$$

откуда следует $b = d$.

Таким образом, мы нашли критерий, позволяющий узнать число требуемых измерительных лент: оно равно числу различных значений, оставшихся после выделения полных квадратов во входных числах s_i.

Для выделения полного квадрата будем использовать технику, аналогичную алгоритму поиска всех делителей числа: будет перебирать все числа-кандидаты вплоть до квадратного корня из исходного числа. Разница только в том, что вместо простой проверки на делимость мы будем проверять на делимость на квадрат текущего значения.

Это решение имеет асимптотику $O(n\sqrt{m} + n\log m)$, где m — максимум из s_i, поскольку выделение полного квадрата совершит порядка $\sqrt{m}$ операций, и это действие будет совершено n раз, и затем порядка $n \log n$ операций потребуется для подсчета различных чисел.

Быстрый подсчет различных чисел

Если бы в этом была необходимость, можно было бы избавиться от слагаемого $n \log n$ в асимптотике. Например, можно было бы за время $O(m)$ подсчитать различные числа в массиве битовых масок из m / 8 байт. Либо же можно было бы воспользоваться хешированием, которое обеспечит время работы $O(n)$ в среднем (но не в худшем случае). Так или иначе, при данных ограничениях этого не требуется.

Реализация

Python

```
n = int(input())
s = map(int, input().split())
rests = set()
for item in s:
  square = 1
  i = 2
  while i * i <= item:
    if item % (i * i) == 0:
      square = i
    i += 1
  rests.add(item // (square *
square))
print(len(rests))
```

C++

```
#include <iostream>
#include <set>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> s(n);
  for (int i = 0; i < n; ++i)
    cin >> s[i];
  set<int> rests;
  for (int i = 0; i < n; ++i) {
    int square = 1;
    for (int j = 2; j * j <= s[i];
         ++j) {
      if (s[i] % (j * j) == 0)
        square = j;
    }
    rests.insert(s[i] /
                 (square * square));
  }
  cout << rests.size() << endl;
}
```

15.7. Пример решения задачи. Произведения-квадраты

Задача. Даны n натуральных чисел a_i. Требуется посчитать число пар, произведение которых — точный квадрат. Говоря формально, нужно найти количество способов выбрать i и j так, что $1 \le i < j \le n$ и существует такое целое x_{ij}, что:

$$a_i \cdot a_j = x_{ij}^2.$$

Входные данные содержат n в первой строке и последовательность из n разделенных пробелом чисел a_i во второй. Ограничения: $2 \le n \le 10^5$, $a_i \le 10^6$. Вывести требуется единственное число — количество искомых пар.

Примеры

Входные данные	**Требуемый результат**
5 3 8 27 2 4	2
Входные данные	**Требуемый результат**
10 1 2 3 4 5 6 7 8 9 10	4

Пояснения к примерам

В первом примере есть две пары, образующие точный квадрат в произведении: (3, 27) и (2, 8): $3 \cdot 27 = 81$, $2 \cdot 8 = 16$. Во втором примере таких пар четыре: (1, 4), (1, 9), (2, 8) и (4, 9): $1 \cdot 4 = 4$, $1 \cdot 9 = 9$, $2 \cdot 8 = 16$, $4 \cdot 9 = 36$.

Решение

Наивное решение было бы следующим: перебрать все пары чисел из входного набора и проверить каждое произведение на то, является ли он точным квадратом (например, путем вычисления квадратного корня и последующего возведения его в квадрат). Однако такое решение было бы слишком медленным: оно бы потребовало $O(n^2)$ проверок.

Попробуем придумать альтернативное, более эффективное решение. Можно ли по числу a_i понять, с какими числами a_j оно будет образовывать в произведении точный квадрат?..

Если посмотреть на факторизацию точного квадрата, можно заметить, что все простые входят в нее в четной степени. Например, $9 = 3^2$, $4^2 = 16 = 2^4$, $15^2 = 225 = 3^2 \cdot 5^2$. Это следует, например, из того, что при перемножении двух чисел степени простых в их факторизациях складываются, а, значит, при возведении числа в квадрат степени его простых удваиваются.

Таким образом, два числа образуют в произведении точный квадрат, когда для каждого простого делителя сумма его степеней в первом и во втором числах — четна. Перефразируя этот критерий, можно сказать, что после удаления точных квадратов из обеих факторизаций оба числа должны быть равны. Например, 6 и 150 должны давать в произведении точный квадрат, поскольку после удаления точных квадратов они оба превратятся в 6: $150 = 5^2 \cdot 6$. Другой пример: 98 и 18 дают в произведении точный квадрат, поскольку оба превращаются в 2 после удаления точных квадратов: $98 = 2 \cdot 7^2$, $18 = 2 \cdot 3^2$.

Этот критерий дает нам ключ к эффективному решению задачи, ведь теперь мы можем для каждого a_i быстро узнать количество подходящих a_j: если мы будем удалять точный квадрат из каждого числа и подсчитывать количество каждых из результирующих чисел в структуре данных типа «словарь», то подсчет искомого количества подходящих a_j сведется к одному запросу поиска в этой структуре данных.

Оценим асимптотику этого решения. Обозначим $m = \max_{i=1\ldots n}\{a_i\}$. Тогда поиск делителя — точного квадрата займет $O(\sqrt{m})$, а подсчет предыдущих чисел с заданным значением займет $O(\log n)$. Итоговая асимптотика составит $O(n\sqrt{m} + n\log n)$.

Подсчет различных чисел

Как и в предыдущей задаче, можно избавиться от слагаемого $n \log n$ в асимптотике, например выделив массив из m элементов. Эта оптимизация не обязательна при поставленных ограничениях, и приведенные ниже реализации не используют ее.

Реализация

Python

```
n = int(input())
a = map(int, input().split())
cnt = {}
ans = 0
for cur in a:
  square = 1
  i = 1
  while i * i <= cur:
    if cur % (i * i) == 0:
      square = i
    i += 1
  cur //= square * square
  ans += cnt.get(cur, 0)
  cnt[cur] = cnt.get(cur, 0) + 1
print(ans)
```

C++

```
#include <cstdint>
#include <iostream>
#include <map>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  int64_t ans = 0;
  map<int, int> cnt;
  for (auto cur : a) {
    int square = 1;
    for (int i = 1; i * i <= cur;
         ++i) {
```

```
      if (cur % (i * i) == 0)
        square = i;
    }
    cur /= square * square;
    ans += cnt[cur];
    ++cnt[cur];
  }
  cout << ans << endl;
}
```

15.8. Пример решения задачи. Запросы числа делителей

Задача. Даны n натуральных чисел a_i. Вы должны написать программу, обрабатывающую запросы вида (s_j, t_j), где ответом на запрос должно быть количество делителей у числа $a_{s_j} \cdot a_{t_j}$.

Входные данные состоят из числа n в первой строке ($1 \le n \le 10^3$) и последовательности из n разделенных пробелом чисел a_i во второй ($1 \le a_i \le 10^8$). Третья строка содержит количество m запросов ($1 \le m \le 10^5$), и последующие m строк содержат по паре натуральных чисел между 1 и n: значения s_j и t_j для каждого запроса. Вывести требуется m разделенных пробелом чисел — ответы на каждый запрос в порядке их следования.

Примеры

Входные данные	**Требуемый результат**
4 1 2 3 4 5 1 2 1 3 1 4 4 2 4 3	2 2 3 4 6
Входные данные	**Требуемый результат**
1 10 1 1 1	9

Пояснения к примерам

В первом примере имеются запросы числа делителей в следующих числах: 2 (делители 1 и 2), 3 (делители 1 и 3), 4 (делители 1, 2, 4), 8 (делители 1, 2, 4, 8), 12 (делители 1, 2, 3, 4, 6, 12). Во втором примере есть только один запрос: число 100 (делители 1, 2, 4, 5, 10, 20, 25, 50, 100).

Решение

Наивное решение было бы для каждого запроса подсчитать число делителей описанным ранее в этой главе алгоритмом каждое полученное произведение $a_{s_j} \cdot a_{t_j}$.

Однако заданные ограничения делают этот алгоритм слишком медленным: произведение может достигать величины 10^{16}, а, значит, обработка даже одного такого числа обычным алгоритмом будет требовать порядка 10^8 операций; учитывая что запросов может быть 10^5, работать такое решение будет целую вечность.

Первая идея на пути к эффективному решению — заметить, что число делителей числа можно быстро вычислить по его факторизации. В самом деле, если известна факторизация некоторого числа x:

$$x = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k},$$

то число делителей x можно вычислить как:

$$(\alpha_1 + 1)(\alpha_2 + 1)\ldots(\alpha_k + 1).$$

Эту формулу легко вывести самому, если заметить, что любой делитель x может иметь только те же самые простые в своей факторизации, что и x, и только в тех же или меньших степенях. Иными словами, степень простого p_1 может принимать любое целое значение из отрезка $[0; \alpha_1]$, степень простого p_2 может принимать значения из $[0; \alpha_2]$ и т. д. Поскольку все эти варианты независимы друг от друга, общее число комбинаций равно произведению числа вариантов для каждого из простых.

Следующая идея — что для быстрого вычисления факторизации произведения $a_{s_j} \cdot a_{t_j}$ достаточно знать факторизации множителей. В самом деле, при перемножении чисел их факторизации суммируются: степень каждого простого в произведении будет равна сумме степеней этого простого в первом и во втором числе. Эта оптимизация позволяет заменить факторизацию большого числа (до 10^{16}) факторизацией двух небольших чисел (не превосходящих 10^8).

Последняя идея: заранее вычислить факторизации всех чисел a_i: нет нужды вычислять их заново для каждого запроса.

Итого, решение состоит из предварительного подсчета — факторизации всех входных чисел — и обработки запросов, каждая из которых состоит из сложения двух соответствующих факторизаций и вычисления произведения. Предварительный подсчет можно оценить как $O(n\sqrt{k})$, где $k = \max_{i=1\ldots n}\{a_i\}$. Время обработки одного запроса определяется числом элементов в одной факторизации, которое можно оценить как $O(\log k)$, поскольку произведение различных простых чисел растет с экспоненциальной скоростью относительно количества простых. Итоговая асимптотика всего решения получается равной $O(n\sqrt{k} + m\log k)$.

Реализация

Python

```
from collections import *
n = int(input())
a = map(int, input().split())
factor = []
for cur in a:
  curfactor = Counter()
  div = 2
  while div * div <= cur:
    while cur % div == 0:
      curfactor[div] += 1
      cur //= div
    div += 1
  if cur > 1:
    curfactor[cur] = 1
  factor.append(curfactor)
m = int(input())
for _ in range(m):
  s, t = map(int, input().split())
  curfactor = (factor[s - 1] +
               factor[t - 1])
  ans = 1
  for power in curfactor.values():
    ans *= power + 1
  print(ans, end=" ")
```

C++

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;
map<int, int> MergeFactor(
    const map<int, int>& x,
    const map<int, int>& y) {
  auto res = x;
  for (auto item : y)
    res[item.first] += item.second;
  return res;
}
int main() {
  int n;
  cin >> n;
  vector<map<int, int>> factor(n);
  for (int i = 0; i < n; ++i) {
    int cur;
    cin >> cur;
    for (int div = 2;
         div * div <= cur; ++div) {
      while (cur % div == 0) {
        ++factor[i][div];
        cur /= div;
      }
    }
    if (cur > 1)
      ++factor[i][cur];
  }
  int m;
  cin >> m;
  for (int i = 0; i < m; ++i) {
    int s, t;
    cin >> s >> t;
    auto merged = MergeFactor(
        factor[s - 1],
        factor[t - 1]);
    int ans = 1;
    for (auto item : merged)
      ans *= item.second + 1;
    cout << ans << ' ';
  }
}
```

Глава 16. Динамическое программирование. Основы

Динамическое программирование — одна из важнейших тем в спортивном программировании; она встречается в огромном числе задач. Это метод решения, который строит ответ поэтапно, начиная с небольшой подзадачи и постепенно увеличивая размер подзадачи, пока, наконец, не будет решена вся исходная задача целиком.

Поскольку это одна из тем, которые проще всего осваивать на конкретных задачах, сразу перейдем к решению примеров.

16.1. Пример решения задачи. Сумма однообразных чисел

Задача. Однообразным числом называется число, все цифры которого одинаковы (в этой задаче мы имеем в виду десятичную систему счисления). Требуется разложить данное число в сумму наименьшего количества однообразных чисел.

Входные данные состоят из единственного числа n. Гарантируется, что n — натуральное число, не превосходящее 10^5. Вывести требуется искомое наименьшее количество слагаемых.

Примеры

Входные данные	**Требуемый результат**
123	3
Входные данные	**Требуемый результат**
9	1

Пояснения к примерам

В первом примере заданное число можно разложить в сумму трех слагаемых: 111 + 11 + 1 = 123. Во втором примере заданное число уже является однообразным, поэтому ответ равен единице.

Решение

В этой задаче можно попытаться найти какие-либо закономерности — что всегда имеет смысл в качестве первого шага при решении подобных задач — однако ниче-

го очевидного не обнаруживается. Жадный алгоритм — всегда отнимать наибольшее возможное однообразное число — также не всегда верен; контрпримером является число 121 (которое можно разложить в два слагаемых «99+22», но после отнимания 111 можно добиться лишь минимум трех слагаемых в итоге). Перебор всевозможных разбиений на слагаемые — слишком медленный (отсечения перебора могут заметно ускорить его, но трудно предсказать заранее, достаточно ли быстрым будет он при поставленных ограничениях).

Попробуем найти достаточно эффективное решение этой задачи и с понятной асимптотикой. Что, если мы переберем последнее слагаемое в искомой сумме? Обозначим его через k; оно должно быть однообразным числом — обозначим множество всех однообразных чисел через R. Тогда мы можем выразить ответ на задачу для n через ответ на задачу для $n - k$ по всевозможным k:

$$d[n] = 1 + \min_{\substack{k \in R, \\ k \le n}} \{d[n-k]\}.$$

Используя эту формулу, мы можем вычислить $d[1]$, затем $d[2]$ и т. д. вплоть до искомого $d[n]$. В самом начале вычислений мы сразу записываем $d[0] = 0$, поскольку это удобно для вычисления последующих ячеек, которые сами являются однообразными числами.

Это и есть динамическое программирование, в простейшей его форме.

Асимптотика этого решения составит $O(n \log n)$, поскольку мы вычисляем ответ для каждого из n значений, и для вычисления очередного ответа мы проходим по всем однообразным числам, число которых можно оценить как $O(\log n)$ (потому что однообразных чисел заданной длины — ровно 9 штук, а всевозможных длин, не превосходящих длину числа n, ровно $\lfloor \log_{10} n \rfloor + 1$).

Реализация

Python

```
n = int(input())
d = [float('inf')] * (n + 1)
d[0] = 0
for i in range(1, n + 1):
  ones = 1
  while ones <= i:
    for digit in range(1, 10):
      k = ones * digit
      if k > i:
        break
      d[i] = min(d[i],
                 d[i - k] + 1)
    ones = ones * 10 + 1
print(d[n])
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
const int INF = 1E9;
int main() {
  int n;
  cin >> n;
  vector<int> d(n + 1, INF);
  d[0] = 0;
  for (int i = 1; i <= n; ++i) {
    for (int ones = 1; ones <= i;
         ones = ones * 10 + 1) {
      for (int digit = 1;
           digit <= 9; ++digit) {
```

```
            int k = ones * digit;
            if (k > i)
              break;
            d[i] = min(d[i],
                       d[i - k] + 1);
          }
        }
      }
      cout << d[n] << endl;
}
```

16.2. Пример решения задачи. Наидлиннейшая возрастающая подпоследовательность

Задача. Дана последовательность a_i целых чисел длины n. Требуется найти ее наидлиннейшую возрастающую подпоследовательность, т. е. такой набор индексов $1 \le i_1 < i_2 < \ldots < i_k \le n$, что:

$$a_{i_1} < a_{i_2} < \ldots < a_{i_k}$$

и число k — максимально возможное.

Входные данные содержат натуральное n в первой строке ($1 \le n \le 10^4$) и последовательность из n разделенных пробелом чисел a_i во второй ($|a_i| \le 10^9$). Вывести требуется единственное натуральное число — длину наидлиннейшей возрастающей подпоследовательности.

Примеры

Входные данные	**Требуемый результат**
6 1 4 2 3 6 5	4
Входные данные	**Требуемый результат**
4 4 3 2 1	1

Пояснения к примерам

В первом примере существует возрастающая подпоследовательность длины 4: это «1 2 3 6» (или «1 2 3 5»). Во втором примере входная последовательность строго убывающая, поэтому мы можем только взять любой элемент в качестве возрастающей подпоследовательности длины 1.

Решение

Можно попытаться придумать какие-то простые эвристики для решения этой задачи, однако все они будут неверны на каких-то примерах. Концептуальная проблема

здесь в том, что, глядя на какой-то обособленный элемент, мы не знаем, нужен ли он в искомой наидлиннейшей подпоследовательности или нет. Если мы берем какое-то число, то увеличиваем длину на один, но ценой того, что появляются дополнительные ограничения на числа до и после этого числа. Не учитывая всей картины в целом, невозможно принять это локальное решение насчет конкретного элемента.

Постараемся найти такое решение, которое учитывает все необходимые случаи, но при этом и достаточно эффективное. Ключевое наблюдение: мы можем строить несколько ответов-кандидатов, постепенно, пошагово добавляя по одному числу в рассмотрение и обновляя эти кандидаты.

В самом деле, допустим, мы хотим узнать ответ для всех n чисел, после того как мы уже нашли ответ для предыдущих $n - 1$ чисел. Мы рассматриваем a_n и хотим узнать, какой могла бы быть наидлиннейшая возрастающая подпоследовательность, оканчивающаяся на a_n. Что нам надо знать для этого? Единственное ограничение, которое накладывает a_n, — это то, что предыдущее число подпоследовательности должно быть строго меньше него. Мы не знаем, каким должно быть это число — тогда давайте переберем его; обозначим его индекс через k. Теперь нам нужно знать длину наидлиннейшей возрастающей подпоследовательности, оканчивающейся в позиции k; если мы обозначим эту величину через $d[k]$, то справедливо:

$$d[n] = \max_{k<n,\ a_k<a_n} d[k] + 1.$$

Физический смысл этой формулы: мы можем продлить некоторые из уже построенных возрастающих последовательностей — те из них, которые оканчиваются на строго меньших, чем текущее, числах. После продления нам уже не важно, какие числа идут в этой подпоследовательности: единственно ценная для решения информация — это то, что подпоследовательность оканчивается на элементе a_n.

Особый случай — когда a_n больше всех предыдущих элементов либо равен им; в таком случае ответ для него надо положить равным единице — физический смысл чего заключается в том, что мы начинаем новую подпоследовательность — длины 1.

То, что мы получили, — это и есть решение на основе динамического программирования. Мы выразили ответ для каждого индекса через ответы для предыдущих индексов, а, значит, мы можем вычислять их один за другим слева направо.

Асимптотика этого решения составит $O(n^2)$, поскольку мы вычисляем ответ для каждой из n позиций, и каждое из этих вычислений требует просмотра всех предыдущих элементов.

Ускоренное решение

Есть возможность решить эту задачу и за время $O(n \log n)$, используя другой способ динамического программирования или применив специальные структуры данных. Мы рассмотрим эти более эффективные варианты решений в последующих «ступенях» книги, поскольку они выходят за пределы основ динамического программирования.

Реализация

Python

```
n = int(input())
a = list(map(int, input().split()))
d = [1] * n
for i in range(n):
  for k in range(i):
    if a[k] < a[i]:
      d[i] = max(d[i], d[k] + 1)
print(max(*d))
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<int> d(n, 1);
  for (int i = 0; i < n; ++i) {
    for (int k = 0; k < i; ++k) {
      if (a[k] < a[i])
        d[i] = max(d[i], d[k] + 1);
    }
  }
  cout << *max_element(
      d.begin(), d.end()) << endl;
}
```

16.3. Пример решения задачи. Подмножество с заданной суммой

Задача. В кошельке лежат n монет; номинал (достоинство) i-й равен a_i. Требуется узнать, можно ли набрать с их помощью заданную сумму k. Говоря формально, нужно проверить, существует ли подмножество с заданной суммой.

В первой строке входных данных записаны числа n и k, во второй — последовательность из n чисел a_i. Все числа натуральные; $n \le 100$, $k \le 10^6$, $a_i \le 10^4$. Вывести требуется строку «YES» или «NO» — ответ на задачу.

Примеры

Входные данные	**Требуемый результат**
4 15 2 9 6 10	YES
Входные данные	**Требуемый результат**
3 8 1 4 6	NO

Пояснения к примерам

В первом примере сумму 15 можно набрать из элементов 9 и 6. Во втором примере сумму 8 набрать невозможно.

Решение

Эта задача — одна из классических оптимизационных задач. В англоязычной литературе она известна под названием subset sum problem.

Первой попыткой решить эту задачу может быть жадный алгоритм — т. е. всегда брать наибольшую из оставшихся монет. Однако уже на первом примере из условия видно, что этот алгоритм неверен: сумму 15 невозможно будет набрать, если взять монету номиналом 10, хотя это и наибольшая из доступных монет. Попытки решить задачу различными эвристическими алгоритмами будут сталкиваться со схожими трудностями: трудно предсказать, какие монеты взять необходимо, а какие заведут решение в тупик. Перебор всевозможных комбинаций работал бы чрезвычайно медленно: он потребовал бы $O(2^n)$ операций.

Вместо этого попробуем пошагово увеличивать сложность задачи. Какие суммы можно получить одной монетой a_1? Очевидно, только a_1. Какие суммы можно получить двумя монетами a_1 и a_2? Это a_1, a_2, $a_1 + a_2$. И так далее — можно сказать, что добавление очередной монеты позволяет достичь все тех же сумм, что и раньше, плюс все эти же суммы, увеличенные на номинал новой монеты, а также суммы, равной номиналу этой монеты.

На первый взгляд кажется, что мы получили то же самое неэффективное решение с помощью перебора, что и раньше. Однако ключевое наблюдение здесь — что суммы могут начать повторяться; вообще говоря, число различных сумм не превышает k.

Это приводит нас к следующему решению: изначально мы начинаем без единой монеты и с единственной доступной суммой, равной нулю. Далее мы совершаем n итераций, на очередной итерации добавляя монету a_i в рассмотрение. Номинал очередной монеты добавляется к каждой из уже достигнутых ранее сумм, и тем самым строится множество новых достижимых сумм. Реализация должна быть организована таким образом, чтобы дублирующиеся суммы были бы объединены.

Альтернативная интерпретация решения

Это же решение можно описать и по-другому, в терминах состояний (i, j), где i — число уже добавленных в рассмотрение монет, а j — текущая сумма. Тогда из состояния (i, j) можно перейти в состояния $(i + 1, j)$ и $(i + 1, j + a_i)$, что соответствует вариантам, когда мы включаем новую монету в новую сумму или не делаем этого. В такой постановке задача сводится к тому, чтобы проверить, существует ли путь из состояния (0, 0) в состояние (n, k).

Асимптотика этого решения составит $O(n \cdot k)$, поскольку алгоритм совершает n итераций, на каждой из которых может быть просмотрено k различных сумм в худшем случае.

Реализация

При реализации удобно хранить достижимые суммы в булевском массиве размера k, поскольку это автоматически устраняет дублирующиеся значения. Мы обозначим этот массив через d, т. е. d_j равен «истине» или «лжи» в зависимости от того, представима сумма j текущим набором монет или нет.

Тогда при добавлении очередной монеты a_i мы для всех истинных d_j также выставляем значение истины в d_{j+a_i}.

Динамика «вперед» и динамика «назад»

Есть два подхода к динамическому программированию: «вперед» и «назад». Описанное здесь решение — это динамика «вперед», поскольку из каждого состояния мы делаем переход в новое, «следующее», состояние и обновляем его значение. Можно было бы тот же алгоритм представить и в виде динамики «назад», когда мы каждое значение d_j обновляем значением d_{j-a_i}. В данной задаче смена подхода ничего не дает, но в некоторых задачах один подход оказывается удобнее другого.

Python

```
n, k = map(int, input().split())
a = map(int, input().split())
d = [False] * (k + 1)
d[0] = True
for coin in a:
  for oldsum in range(
      k - coin, -1, -1):
    if d[oldsum]:
      d[oldsum + coin] = True
print("YES" if d[k] else "NO")
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<char> d(k + 1);
  d[0] = true;
  for (int coin : a) {
    for (int oldsum = k - coin;
         oldsum >= 0; --oldsum) {
      if (d[oldsum])
        d[oldsum + coin] = true;
    }
  }
  cout << (d[k] ? "YES" : "NO")
       << endl;
}
```

Примечание о порядке обновления массива d

В приведенных выше реализациях важно то, что массив d просматривается и заполняется в порядке от больших индексов к меньшим. Физический смысл этого в том, что одну монету можно использовать в искомой сумме лишь один раз. Если бы мы шли по массиву слева направо, то это было бы нарушено: только что добавленные суммы могли бы быть использованы с той же монетой еще раз.

Примечание о типе vector<char> в реализации на C++

Наша реализация использует тип vector<char>, а не vector<bool>, поскольку последний использует технику битового сжатия, которая экономит память ценой замедления работы. При данных ограничениях потребление памяти — один мегабайт — не является проблемой, поэтому мы использовали более быстрый вариант vector<char>.

Ускорение в 32 и более раз с помощью битовых операций

Можно заметить, что решение можно значительно ускорить с помощью битовых операций. А именно, можно хранить массив d в виде битовой маски, т. е. k / 8 байт, каждый из которых содержит по 8 значений d_i в своих битах. Тогда вложенный цикл решения — проверка и присвоение значений — можно представить как побитовый сдвиг и последующую побитовую операцию «ИЛИ». Эта оптимизация не изменит асимптотику, однако значительно уменьшит скрытую константу (теоретически — в 32 или 64 раз, в зависимости от размера машинного слова; на практике ускорение может немного отличаться в бóльшую или меньшую сторону).

16.4. Пример решения задачи. Минимальное подмножество с заданной суммой

Задача. Аналогично предыдущему заданию в кошельке имеется n монет известных номиналов a_i и требуется набрать этими монетами заданную сумму k. Однако теперь требуется вывести наименьшее число монет, с помощью которых этого можно добиться — либо «−1», если решения не существует.

Примеры

Входные данные	**Требуемый результат**
4 15 2 9 6 10	2
Входные данные	**Требуемый результат**
3 8 1 4 6	-1

Решение

Решать эту задачу будем аналогично предыдущей, только теперь значением динамики будет не булева переменная, обозначающая наличие или отсутствие пути, а число — наименьшее число монет, которым можно достичь заданной суммы.

Более детальное описание алгоритма:

- заведем массив d, каждый элемент d_j которого будет равен наименьшему числу монет, которым можно достичь суммы j;
- мы добавляем монеты в рассмотрение по одной. Изначально нет ни одной доступной монеты, поэтому $d_0 = 0$, а все остальные элементы равны значению «бесконечность» — что обозначает отсутствие решения;

- добавление очередной монеты a_i мы производим как проход по всему массиву d с переходами из каждого d_j в d_{j+a_i}, что обозначает, что к сумме j мы можем добавить текущую монету и получить сумму $j + a_i$. Проход мы совершаем в порядке уменьшения индексов j, чтобы одну и ту же монету нельзя было использовать несколько раз.

Асимптотика решения — по-прежнему $O(n \cdot k)$.

Реализация

Python

```
INF = float('inf')
n, k = map(int, input().split())
a = map(int, input().split())
d = [INF] * (k + 1)
d[0] = 0
for coin in a:
  for oldsum in range(
      k - coin, -1, -1):
    d[oldsum + coin] = min(
        d[oldsum + coin],
        d[oldsum] + 1)
print(-1 if d[k] == INF else d[k])
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
const int INF = (int)1E9;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<int> d(k + 1, INF);
  d[0] = 0;
  for (int coin : a) {
    for (int oldsum = k - coin;
         oldsum >= 0; --oldsum) {
      d[oldsum + coin] = min(
          d[oldsum + coin],
          d[oldsum] + 1);
    }
  }
  cout << (d[k] == INF ? -1 : d[k])
       << endl;
}
```

Константа INF

Приведенные реализации используют константу `INF` для хранения значения «бесконечность», т. е. в данном случае для обозначения отсутствия способа набрать какую-либо сумму. Теоретически вместо использования такой константы, равной некоему большому числу (миллиарду в наших примерах), можно было бы использовать и число $n + 1$, однако константа «бесконечность», на наш взгляд, лучше отражает смысл этой части кода.

16.5. Пример решения задачи. Получение суммы монетами заданных номиналов

Задача. Известны номиналы всех монет некоторой валюты: n натуральных чисел a_i. Требуется проверить, можно ли такими монетами набрать заданную сумму k; можно считать, что имеется неограниченный запас каждой из монет.

Входные данные содержат числа n и k в первой строке и последовательность из n чисел a_i во второй. Ограничения: $1 \le n \le 100$, $1 \le k \le 10^6$, $1 \le a_i \le 10^4$.

Вывести требуется строку «YES» или «NO» — ответ на задачу.

Примеры

Входные данные	**Требуемый результат**
3 8 1 4 6	YES
Входные данные	**Требуемый результат**
4 11 2 8 6 4	NO

Решение

В англоязычной литературе эта задача известна под названием unbounded subset sum problem.

Единственное отличие этой задачи от предыдущих в том, что каждую монету можно использовать неограниченное число раз.

Применяя метод динамического программирования, получаем, что при добавлении в рассмотрение очередной монеты a_i мы из суммы j можем получить суммы $j + a_i$, $j + 2a_i$ и т. д.

При реализации этой динамики можно поступить еще проще и не перебирать коэффициент перед a_i, поскольку достаточно перебирать индексы j **в порядке возрастания**.

Например, если мы добавляем в рассмотрение первую монету с номиналом $a_1 = 2$, то сначала мы обнаружим d_0 = true и пометим d_2 = true, затем мы обнаружим d_2 = true и пометим d_4 = true и т. д.

Таким образом, мы не только упрощаем реализацию, но и сохраняем асимптотику $O(n \cdot k)$, ведь из каждого состояния динамики мы по-прежнему совершаем только по одному переходу для каждой из монет.

Реализация

Python

```
n, k = map(int, input().split())
a = map(int, input().split())
```

C++

```
#include <iostream>
#include <vector>
```

```
d = [False] * (k + 1)
d[0] = True
for coin in a:
  for oldsum in range(
      0, k - coin + 1):
    if d[oldsum]:
      d[oldsum + coin] = True
print("YES" if d[k] else "NO")
```

```
using namespace std;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<char> d(k + 1);
  d[0] = true;
  for (int coin : a) {
    for (int oldsum = 0;
         oldsum <= k - coin;
         ++oldsum) {
      if (d[oldsum])
        d[oldsum + coin] = true;
    }
  }
  cout << (d[k] ? "YES" : "NO")
       << endl;
}
```

16.6. Пример решения задачи. Задача о рюкзаке

Задача. Имеется n предметов, каждый из которых характеризуется своим весом w_i и ценой v_i. Рюкзак может вместить в себя набор предметов, если их суммарный вес не превосходит m. Требуется разместить в рюкзаке такой набор, суммарная стоимость которого была бы максимально возможной.

Входные данные содержат числа n и m в первой строке ($1 \le n \le 100$, $1 \le m \le 10^6$) и набор пар «w_i v_i» в последующих n строках ($1 \le w_i, v_i \le 10^4$).

Вывести требуется одно число — максимально возможную стоимость предметов, размещенных в рюкзаке.

Примеры

Входные данные	Требуемый результат
5 20 10 100 10 5 8 5 9 20 12 110	120

Входные данные	Требуемый результат
1 5 10 10	0

Решение

Английское название этой классической в информатике задачи — knapsack problem. Иногда уточняют: 0–1 knapsack problem, имея в виду, что каждый предмет можно включить в ответ только в одном экземпляре.

Эта задача — очередной пример того, как всевозможные жадные алгоритмы, многообещающие на первый взгляд, будут приводить к неоптимальному ответу на тот или иной тест. Уже в примере из условия продемонстрирована ситуация, когда в ответ невыгодно включать ни самый дорогой, ни самый легкий предмет.

Динамическое программирование позволяет получить надежное решение этой задачи с приемлемым временем работы. Будем, как и в предыдущих задачах, по одному добавлять предметы в рассмотрение.

Введем массив d, каждый элемент d_j которого равен **наибольшей стоимости**, которую можно достичь **с весом *j***.

- Изначально у нас нет ни одного доступного предмета, а потому массив d можно заполнить нулями. Это соответствует тому, что для нуля предметов суммарная их стоимость равна нулю.
- При добавлении предмета a_i мы совершаем переходы из каждого состояния j в состояние $j + w_i$. При этом мы пытаемся улучшить значение d_{j+w_i} значением $d_j + v_i$.
- Как и в задачах с монетами, при совершении переходов важно перебирать индексы j **в порядке убывания**, чтобы очередной предмет был учтен единожды.

Асимптотика решения составит $O(n \cdot m)$, поскольку в динамике есть лишь $m + 1$ состояние и для каждого из n предметов производится проход по всем состояниям.

Альтернативная интерпретация решения

Как и в рассмотренных ранее задачах, это решение можно описать в терминах двумерной динамики: состоянием будет пара (i, j), где i — число включенных в рассмотрение предметов, а j — вес набора, и каждому состоянию будет приписано значение $\partial_{i,j}$ — максимально возможный вес. Тогда из каждого состояния (i, j) можно перейти в состояние $(i + 1, j)$ со значением $\partial_{i,j}$ и в состояние $(i + 1, j + w_i)$ со значением $\partial_{i,j} + v_i$. Для решения задачи при таком подходе надо было бы посчитать значение $\partial_{n,k}$. Такое описание эквивалентно алгоритму, описанному выше, однако при реализации оно было бы более расточительным по памяти: потребовалось бы хранить $O(n \cdot m)$ ячеек двумерного массива ∂.

Реализация

Python

```
n, m = map(int, input().split())
descr = [
    list(map(int, input().split()))
    for _ in range(n)]
d = [0] * (m + 1)
for w, v in descr:
  for oldsum in range(
      m - w, -1, -1):
    d[oldsum + w] = max(
        d[oldsum + w],
        d[oldsum] + v)
print(d[m])
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n, m;
  cin >> n >> m;
  vector<int> w(n), v(n);
  for (int i = 0; i < n; ++i)
    cin >> w[i] >> v[i];
  vector<int> d(m + 1);
  for (int i = 0; i < n; ++i) {
    for (int oldsum = m - w[i];
         oldsum >= 0; --oldsum) {
      d[oldsum + w[i]] = max(
          d[oldsum + w[i]],
          d[oldsum] + v[i]);
    }
  }
  cout << d[m] << endl;
}
```

16.7. Пример решения задачи. Кладоискатель

Задача. Кладоискатель узнал о существовании потенциально богатого на находки места. Итогом подготовительных работ стал план исследований, представляющий собой последовательность из n участков. Кладоискатель начинает в участке №1.

На каждом шаге он исследует текущий участок и, если ничего обнаружить не удалось, переходит дальше. Из-за природной лени он каждый раз при этом бросает монетку: если выпал «орел», то из участка с номером i он переходит в участок $i + 1$, иначе — в участок $i + 2$. Монетка — идеальная, т. е. вероятности выпадения орла и решки в точности равны 50%. Обследование завершается, когда обнаруживается клад или когда кладоискатель должен перейти в участок с номером, бóльшим n.

Вы хотите посчитать, с какой вероятностью кладоискатель обнаружит клад. Вам известна секретная карта местности, из которой вы знаете про каждый участок, содержит он клад или нет. Кроме того, вам известна вероятность p, с которой кладоискатель, обследуя участок с кладом, действительно сумеет его обнаружить.

Входные данные содержат целые числа n и p в первой строке — количество участков и вероятность, заданная в процентах ($1 \le n \le 1000$, $0 \le p \le 100$). Вторая строка содержит секретное описание карты — последовательность из n символов, где ка-

ждый либо нуль (что обозначает пустой участок), либо единица (что обозначает участок с кладом).

Вывести требуется вещественное число — искомую вероятность успеха. Погрешность ответа не должна превышать 10^{-3}.

Примеры

Входные данные	**Требуемый результат**
5 50 01010	0.46875
Входные данные	**Требуемый результат**
3 100 000	0

Решение

Пусть d_i — вероятность пройти ровно i клеток, не найдя клада. Изначально сразу известно, что $d_0 = 0$. Все остальные значения будем подсчитывать постепенно, в порядке возрастания индексов. Пусть для очередного индекса i вероятность d_i уже подсчитана, тогда нам нужно рассмотреть три возможных исхода в позиции i:

1. Возможно, мы обнаружили клад в позиции i:
   - обозначим вероятность этого исхода через f_i;
   - если i-й символ карты — это ноль, то $f_i = 0$;
   - если же i-й символ карты — это единица, то $f_i = d_i \cdot p / 100$, поскольку с вероятностью d_i мы дошли до клетки i и затем с условной вероятностью $p / 100$ мы смогли обнаружить лежащий там клад.
2. Возможно, мы, не найдя клада, перешли в позицию $i + 1$:
   - обозначим через g_i вероятность того, что мы дошли до клетки i и не нашли там клада. Если i-й символ карты — это ноль, то $g_i = d_i$; иначе $g_i = d_i \cdot (1 - p / 100)$, что следует из того, что условная вероятность не найти имеющийся клад равна $1 - p / 100$;
   - вероятность этого события равна $g_i / 2$, поскольку выбор следующей клетки мы производим с условной вероятностью $1 / 2$;
   - прибавим эту величину к d_{i+1}, учтя тем самым этот способ прийти в позицию $i + 1$.
3. Возможно, мы, не найдя клада, перешли в позицию $i + 2$:
   - аналогично предыдущему случаю, вероятность этого события равна $g_i / 2$, и эту величину надо прибавить к d_{i+2}.

Утверждается, что после совершения этих переходов вероятность d_{i+1} уже подсчитана правильно и мы можем перейти к позиции $i + 1$.

После того как мы согласно этому алгоритму подсчитаем все значения d, ответ на задачу будет равен сумме всех f_i.

Время работы этого решения есть величина $O(n)$, поскольку у нас всего n состояний, из каждого из которых всего по три перехода.

Вариант через динамику «назад»

Описанное решение является динамикой «вперед»: зная вероятность прийти в текущее состояние, мы моделируем всевозможные смены состояния, «расщепляем» вероятность по всем ним и прибавляем их к значениям динамики в новых состояниях. Можно было бы решить эту же задачу и динамикой «назад», рассматривая, из каких клеток мы могли прийти в текущую клетку, и суммируя соответствующие вероятности.

Реализация

Python

```
n, p = map(int, input().split())
field = input()
d = [0] * (n + 2)
d[0] = 1
ans = 0
for pos in range(n):
  step_prob = d[pos]
  if field[pos] == '1':
    ans += step_prob * p / 100
    step_prob *= 1 - p / 100
  step_prob /= 2
  d[pos + 1] += step_prob
  d[pos + 2] += step_prob
print('{:.9f}'.format(ans))
```

C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
  int n, p;
  string field;
  cin >> n >> p >> field;
  vector<double> d(n + 2);
  d[0] = 1;
  double ans = 0;
  for (int pos = 0; pos < n;
       ++pos) {
    double step_prob = d[pos];
    if (field[pos] == '1') {
      ans += step_prob * p / 100.0;
      step_prob *= 1 - p / 100.0;
    }
    step_prob /= 2;
    d[pos + 1] += step_prob;
    d[pos + 2] += step_prob;
  }
  cout.setf(ios::fixed);
  cout.precision(9);
  cout << ans << endl;
}
```

16.8. Пример решения задачи. Путь в матрице

Задача. Дана матрица размером $n \times m$. Робот изначально находится в ячейке (1, 1), т. е. в левом верхнем углу. За один шаг робот может передвинуться на одну ячейку вправо или вниз. Требуется найти такой маршрут, сумма элементов которого максимальна.

Входные данные содержат числа n и m в первой строке ($1 \le n, m \le 100$), за которой следуют n строк по m чисел в каждой — элементы матрицы. Все числа целые и по модулю не превосходят 10^4.

Вывести требуется единственное число — сумму чисел вдоль оптимального маршрута.

Примеры

Входные данные	**Требуемый результат**
3 4 1 2 3 4 5 6 7 8 9 1 2 3	30
Входные данные	**Требуемый результат**
3 1 0 -1 0	-1

Решение

Благодаря тому что робот может двигаться только вправо и вниз, любой путь из (1, 1) до любой ячейки (i, j) находится внутри подпрямоугольника $[1...i, 1...j]$. Более того, оптимальный путь до (i, j) всегда представляет собой оптимальный путь до $(i - 1, j)$ или до $(i, j - 1)$, к которому в конец приписали ячейку (i, j).

Следовательно, ответ для каждой ячейки можно выразить через максимум из ответа для ячейки-соседа слева и ответа для ячейки-соседа сверху. Если мы обозначим через $d_{i,j}$ максимальную сумму чисел вдоль пути до (i, j), то эту зависимость можно выразить следующим образом:

$$d_{i,j} = a_{i,j} + \max\left(d_{i-1,j}, d_{i,j-1}\right),$$

где максимум берется среди существующих значений, т. е. для ячеек из первой строки будет взято значение динамики слева, а для ячеек из первого столбца — сверху. Для стартовой ячейки полагаем $d_{0,0} = a_{0,0}$.

Эта формула позволяет последовательно вычислить все значения $d_{i,j}$, например, если идти в порядке увеличения i, при равенстве — в порядке увеличения j.

Асимптотика решения будет $O(n \cdot m)$.

> Отметим, что описанное выше решение является примером динамики «назад», поскольку ответ для текущей ячейки в ней рассчитывается исходя из ответов для предыдущих ячеек.

Реализация

Python

```
INF = float('inf')
n, m = map(int, input().split())
a = [
    list(map(int, input().split()))
    for _ in range(n)]
d = [[-INF] * m for _ in range(n)]
for i in range(n):
  for j in range(m):
    if i == 0 and j == 0:
      d[i][j] = 0
    if i > 0:
      d[i][j] = max(d[i][j],
                    d[i - 1][j])
    if j > 0:
      d[i][j] = max(d[i][j],
                    d[i][j - 1])
    d[i][j] += a[i][j]
print(d[n - 1][m - 1])
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
const int INF = (int)1E9;
int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> a(n);
  for (auto& row : a) {
    row.resize(m);
    for (int& item : row)
      cin >> item;
  }
  vector<vector<int>> d(n);
  for (int i = 0; i < n; ++i) {
    d[i].assign(m, -INF);
    for (int j = 0; j < m; ++j) {
      d[i][j] = -INF;
      if (i == 0 && j == 0)
        d[i][j] = 0;
      if (i > 0) {
        d[i][j] = max(d[i][j],
                      d[i - 1][j]);
      }
      if (j > 0) {
        d[i][j] = max(d[i][j],
                      d[i][j - 1]);
      }
      d[i][j] += a[i][j];
    }
  }
  cout << d[n - 1][m - 1] << endl;
}
```

Константа INF

Отметим, как использование константы «бесконечность» позволило упростить код. Благодаря ей нам не приходится специально разбирать случаи того, берется ли максимум по двум аргументам или по одному. Но надо учитывать, что значение константы должно быть выбрано достаточно большим, чтобы оно превосходило всевозможные суммы путей в матрице.

16.9. Пример решения задачи. Расстояние редактирования

Задача. Даны две строки p и q. Требуется определить наименьшее число операций, которые нужно применить к p, чтобы получить q. Разрешенные операции — это:

- вставка нового символа;
- удаление символа;
- замена символа.

Входные данные состоят из двух строк, содержащих p и q. Как p, так и q не пусты и содержат только символы латинского алфавита. Длины обеих строк не превосходят 1000.

Вывести требуется единственное число — наименьшее число операций.

Примеры

Входные данные	**Требуемый результат**
Samara Saratov	4
Входные данные	**Требуемый результат**
emaxx emacs	2

Пояснения к примерам

В первом примере к строке «Samara» можно применить следующие операции: заменить третий символ на «r», пятый — на «t», шестой — на «o» и вставить символ «v» в конец строки; тем самым получится строка «Saratov». Во втором примере искомые операции — это замена четвертого и пятого символов.

Решение

Будем постепенно добавлять в рассмотрение символы обеих строк. Пусть $d_{i,j}$ — наименьшее число операций для превращения i первых символов строки p в j первых символов строки q.

База динамики, т. е. начальное состояние — это $i = j = 0$, для которого $d_{0,0} = 0$. Все остальные элементы d изначально заполнены значением «бесконечность».

Предположим теперь, что значение $d_{i,j}$ уже подсчитано, и рассмотрим всевозможные переходы из этого состояния в другие:

- если $p_i = q_j$, то мы можем бесплатно перейти в состояние $(i + 1, j + 1)$, т. е. мы обновляем значение $d_{i+1,j+1}$ значением $d_{i,j}$;
- мы можем произвести вставку нового символа в позицию i; при этом подразумевается, что мы вставляем символ, равный q_j, потому что вставлять другой символ в эту позицию нет смысла, учитывая, что мы уже обеспечили совпадение всех символов слева от этой позиции. Таким образом, мы совершаем одной операцией переход в состояние $(i, j + 1)$, для чего мы обновляем значение $d_{i,j+1}$ величиной $1 + d_{i,j}$;
- мы можем произвести удаление символа в позиции i, что означает, что мы переходим в состояние $(i + 1, j)$ с помощью одной операции. Для этого мы обновляем значение $d_{i+1,j}$ величиной $1 + d_{i,j}$;
- мы можем произвести замену символа в позиции i, предполагая, что заменяем на символ, равный q_j. Тем самым мы можем с помощью одной операции перейти в состояние $(i + 1, j + 1)$, для чего мы обновляем значение $d_{i+1,j+1}$ величиной $1 + d_{i,j}$.

Поскольку все переходы совершаются в состояния с бóльшими индексами (либо по i, либо по j, либо по обоим одновременно), то можно, например, перебирать i в порядке возрастания и вложенным циклом перебирать j в порядке возрастания. При таком подходе в момент, когда мы достигаем очередное состояние (i, j), ответ для него уже будет подсчитан в $d_{i,j}$.

Окончательный ответ на задачу будет содержаться в $d_{n,m}$, где n — длина строки p, m — длина q.

Альтернативная динамика «назад»

Описанное выше решение является динамикой «вперед», поскольку мы перебираем из каждого состояния всевозможные операции и пытаемся улучшить значения в новом состоянии. Вместо этого можно представить решение в виде динамики «назад», рассматривая всевозможные способы прийти в текущее состояние — это будут те же самые четыре варианта — и беря наилучший среди них ответ. Сложность такого решения будет примерно такой же, и лишь пропадет необходимость заранее заполнять массив значениями «бесконечность».

Асимптотика решения — $O(n \cdot m)$.

Реализация

Python

```
INF = float('inf')
p = input()
q = input()
n = len(p)
m = len(q)
d = [[INF] * (m + 2)
```

C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const int INF = (int)1E9;
int main() {
```

```
     for _ in range(n + 2)]
d[0][0] = 0
for i in range(n + 1):
  for j in range(m + 1):
    d[i + 1][j] = min(
        d[i + 1][j], d[i][j] + 1)
    d[i][j + 1] = min(
        d[i][j + 1], d[i][j] + 1)
    if i < n and j < m:
      diff = \
          0 if p[i] == q[j] else 1
      d[i + 1][j + 1] = min(
          d[i + 1][j + 1],
          d[i][j] + diff)
print(d[n][m])
```

```
  string p, q;
  cin >> p >> q;
  int n = (int)p.length();
  int m = (int)q.length();
  vector<vector<int>> d(
      n + 2,
      vector<int>(m + 2, INF));
  d[0][0] = 0;
  for (int i = 0; i <= n; ++i) {
    for (int j = 0; j <= m; ++j) {
      d[i][j + 1] = min(
          d[i][j + 1],
          d[i][j] + 1);
      d[i + 1][j] = min(
          d[i + 1][j],
          d[i][j] + 1);
      if (i < n && j < m) {
        int diff = p[i] != q[j];
        d[i + 1][j + 1] = min(
            d[i + 1][j + 1],
            d[i][j] + diff);
      }
    }
  }
  cout << d[n][m] << endl;
}
```

Дополнительная оптимизация потребления памяти

Можно заметить в этом решении, что на каждой итерации цикла по i мы работаем только со строками d[i] и d[i + 1]. Следовательно, достаточно хранить в памяти только две строки — текущую и следующую и тем самым понизить потребление памяти с квадратичного до линейного. С точки зрения реализации это можно закодировать разными способами; один из наиболее простых — это сделать массив d размером $2 \times (n + 2)$, ввести переменную «k = i % 2» (или, с использованием битовых операций, «k = i & 1»), заменить «d[i]» на «d[k]» и «d[i + 1]» на «d[1 – k]». Останется только не забыть заполнить на каждой итерации цикла по i всю строку d[1 – k] значениями «бесконечность».

Глава 17.
Задачи для самостоятельного решения

Для закрепления пройденного материала настоятельно рекомендуем читателю самостоятельно прорешать задачи этой главы.

17.1. Примеры задач

Подсказки к приведенным ниже задачам даны в *приложении*.

17.1.1. Задача. Пары фиксированной суммы, с повторами

> Эта задача аналогична разобранной в *главе 13*, только теперь числа во входной последовательности могут повторяться.

Дана последовательность целых чисел a_i ($i = 1...n$). Требуется посчитать количество пар с заданной суммой k, т. е. число способов выбрать i и j так, что $a_i + a_j = k$ (где $1 \le i < j \le n$).

Входные данные состоят из чисел n и k в первой строке и последовательности a_i из n чисел во второй. Все числа — целые, $1 \le n \le 5 \cdot 10^5$, $-10^9 \le k \le 10^9$, $-10^9 \le a_i \le 10^9$. Вывести требуется единственное число — искомое количество пар.

Примеры

Входные данные	**Требуемый результат**
4 5 1 3 2 3	2
Входные данные	**Требуемый результат**
3 4 1 1 1	3

17.1.2. Задача. Подстроки с заданным числом единиц

Дана строка, состоящая только из символов нуля и единицы, и число k. Требуется найти количество подстрок, содержащих ровно k единиц.

Подстрокой называется непустая последовательность подряд идущих символов исходной строки.

Входные данные содержат целое число k и строку, разделенные пробелом. Строка не пуста, не содержит никаких других символов, кроме нуля и единицы, и ее длина не превосходит 10^6; число k неотрицательно и не превосходит длины строки. Вывести нужно единственное число — искомое количество.

Примеры

Входные данные	**Требуемый результат**
`3 001110101`	`6`
Входные данные	**Требуемый результат**
`0 1010001`	`7`

17.1.3. Задача. Пары с суммой в диапазоне

Дана последовательность из n целых чисел a_i, а также целые числа l и r. Требуется найти количество пар элементов, таких, что их сумма лежит в отрезке $[l; r]$. Говоря формально, нужно подсчитать количество способов выбрать $1 \le p < q \le n$, так чтобы $l \le a_p + a_q \le r$.

Входные данные состоят из целых чисел n, l и r в первой строке и последовательности из n разделенных пробелом целых чисел a_i во второй. Гарантируется, что $1 \le n \le 5 \cdot 10^5$, $-10^9 \le l \le r \le 10^9$, $|a_i| \le 10^9$. Вывести требуется единственное число — искомое количество.

Примеры

Входные данные	**Требуемый результат**
`5 3 5` `1 2 5 4 3`	`4`
Входные данные	**Требуемый результат**
`3 11 11` `4 6 8`	`0`

17.1.4. Задача. Подотрезок с наибольшим минимумом

Даны n целых чисел a_i и число $1 \le k \le n$. Требуется найти подотрезок длины k с наибольшим минимумом из всех возможных. Иными словами, требуется найти такую позицию p, которая максимизирует следующую величину:

$$\min_{p \le j < p+k} a_j.$$

Входные данные состоят из натуральных чисел n и k в первой строке и n целых чисел a_i во второй. Числа разделяются пробелом; $n \le 5 \cdot 10^5$, $|a_i| \le 10^9$. Вывести требуется одно число — позицию начала искомого отрезка; если ответов несколько, вывести можно любой.

Примеры

Входные данные	Требуемый результат
5 3 10 1 20 2 3	3
Входные данные	**Требуемый результат**
3 1 3 2 1	1

Пояснения к примерам

В первом примере искомый подотрезок — с третьего по пятый элемент; минимум на нем равен 2, что лучше, чем любой другой подотрезок длины три. Во втором примере требуется найти подотрезок длины один, и ответом является первый элемент: минимум на нем равен 3.

17.1.5. Задача. Произведения-кубы

Даны n натуральных чисел a_i. Требуется подсчитать количество пар чисел, образующих точный куб, т. е. количество таких $1 \le i < j \le n$, что для них существует натуральное x_{ij} и выполняется:

$$a_i \cdot a_j = x_{ij}^3.$$

Примеры

Входные данные	Требуемый результат
4 1 9 3 27	2
Входные данные	**Требуемый результат**
4 1 2 3 4	1

Пояснения к примерам

В первом примере есть две пары, образующие точный куб в произведении: (1, 27) и (3, 9): $1 \cdot 27 = 27 = 3^3$, $3 \cdot 9 = 27 = 3^3$. Во втором примере такая пара одна: (2, 4): $2 \cdot 4 = 8 = 2^3$.

17.1.6. Задача. Мытье окон

По стене длинного коридора идет ряд из n окон. Про каждое окно известно, требуется ли его помыть: a_i равно единице, если i-е окно требуется помыть, и нулю в противном случае. Уборщик получил задание обеспечить чистоту k подряд идущих окон (неважно, начиная с какого). Определите наименьшее число окон, которое ему потребуется помыть, чтобы достичь этой цели.

Входные данные состоят из разделенных пробелом натуральных чисел n и k в первой строке и последовательности a_i, записанной в виде строки из n символов (нулей или единиц) во второй. Гарантируется, что $n \le 10^6$, $k \le n$. Вывести нужно единственное число — наименьшее количество окон, которое требуется помыть.

Примеры

Входные данные	**Требуемый результат**
10 4 1111001011	1
Входные данные	**Требуемый результат**
5 5 00000	0

Пояснения к примерам

В первом примере достаточно помыть окно №7, чтобы обеспечить чистоту четырех окон: с пятого по восьмое. Во втором примере все окна чистые, поэтому ответ равен нулю.

17.1.7. Задача. Санскритская поэтика

В санскритской поэтике основным структурным элементом является слог. Выделяют слоги двух видов: короткий («лагу») и длинный («гуру»). Их длительность можно сравнить с помощью понятия «моры» — фонетической единицы. Лагу имеет длительность одной моры, а гуру — двух мор. Поэтический метр — это последовательность видов слогов; например, «лагу-гуру» или «лагу-лагу-лагу» являются метрами. Требуется посчитать количество поэтических метров с заданным количеством мор.

Входной файл содержит единственное число — количество мор; это натуральное число, не превосходящее 40. Вывести требуется искомое число поэтических метров.

Примеры

Входные данные	**Требуемый результат**
3	3
Входные данные	**Требуемый результат**
1	1

Пояснения к примерам

В первом примере метров с заданной длиной три: «лагу-лагу-лагу», «лагу-гуру», «гуру-лагу». Во втором примере есть только один метр: «лагу».

17.1.8. Задача. Число способов набрать сумму

Даны n монет, номинал i-й равен a_i. Требуется определить, сколькими способами можно выбрать набор из нескольких монет, так чтобы сумма номиналов в нем была равна заданному k. Все монеты считаются различными с точки зрения сравнения наборов.

Входные данные содержат числа n и k в первой строке и последовательность из n чисел a_i во второй. Ограничения: $1 \le n \le 100$, $1 \le k \le 10^6$, $1 \le a_i \le 10^4$.

Поскольку ответ может быть большим, вывести его требуется по модулю $10^9 + 7$.

Примеры

Входные данные	**Требуемый результат**
4 5 1 2 2 3	3
Входные данные	**Требуемый результат**
2 10 5 15	0

17.1.9. Задача о неограниченном рюкзаке

> Эта разновидность задачи о рюкзаке называется в англоязычной литературе unbounded knapsack problem.

Имеется n разновидностей предметов, каждый из которых характеризуется своим весом w_i и ценой v_i. Рюкзак может вместить в себя набор предметов, если их суммарный вес не превосходит m. Требуется разместить в рюкзаке такой набор, суммарная стоимость которого была бы максимально возможной. Можно считать, что имеется неограниченный запас предметов каждой разновидности.

Входные данные содержат числа n и m в первой строке ($1 \le n \le 100$, $1 \le m \le 10^6$) и набор пар «w_i v_i» в последующих n строках ($1 \le w_i, v_i \le 10^4$).

Вывести требуется в первой строке максимально возможную стоимость предметов, размещенных в рюкзаке, а во второй строке — индексы этих предметов в порядке неубывания (предметы нумеруются начиная с единицы).

Примеры

Входные данные	**Требуемый результат**
3 20 5 1 19 10 7 7	15 1 3 3

Входные данные	Требуемый результат
1 5 10 10	0

17.1.10. Задача об ограниченном рюкзаке

Эта задача известна в англоязычной литературе под названием bounded knapsack problem.

Имеется n разновидностей предметов, каждый из которых характеризуется своим весом w_i, ценой v_i и количеством экземпляров c_i. Рюкзак может вместить в себя набор предметов, если их суммарный вес не превосходит m. Требуется разместить в рюкзаке такой набор, суммарная стоимость которого была бы максимально возможной.

Входные данные содержат числа n и m в первой строке ($1 \le n \le 100$, $1 \le m \le 10^5$) и набор троек «w_i v_i c_i» в последующих n строках ($1 \le w_i, v_i \le 10^4$, $1 \le c_i \le 10$).

Вывести требуется максимально возможную стоимость предметов.

Примеры

Входные данные	Требуемый результат
3 20 5 1 100 19 10 100 7 7 1	9
Входные данные	**Требуемый результат**
1 5 10 10 10	0

17.2. Задачи в онлайн-системах

- Codeforces, раунд 599, задача A: «Покраска плиток».
- Codeforces, раунд 296, задача A: «Игра с бумагой».
- Codeforces, раунд 725, задача C: «Количество пар».
- Codeforces, раунд 505, задача B: «Ослабленный Общий Делитель».

Итоги ступени II

В данном разделе читатель познакомился с базовым инструментарием решения задач спортивного программирования. Ключевые навыки: научиться оценивать асимптотику разных вариантов решения и научиться «видеть» завуалированные применения алгоритмов в конкретных задачах.

Очень редко попадаются задания, где прямым текстом сказано, что надо сделать что-то, в точности совпадающее с определением известного алгоритма. Гораздо чаще требуется прокрутить в голове несколько потенциально возможных подходов, отсеять неприменимые и сравнить оставшиеся по скорости их работы. Мы надеемся, что разобранные примеры решения задач и упражнения для самостоятельного решения помогли читателю развить этот навык.

Повторимся: важно не количество выученных алгоритмов, а хорошо отработанный навык применения алгоритмов из своего инвентаря.

Ступень III.
Расширение базового арсенала

Глава 18.
Техники предварительного подсчета на массивах

Перечислим базовые приемы предварительной обработки массивов и строк, которые помогают быстро обрабатывать какие-либо запросы:

- указатели до ближайших элементов с определенным значением;
- частичные суммы;
- указатели до ближайших меньших/бóльших элементов;
- списки позиций для каждого значения;
- сжатие значений.

18.1. Указатели до ближайших элементов

Пусть есть последовательность a_i ($i = 1...n$) и зафиксировано какое-либо значение x. Тогда введем последовательность указателей p_j ($j = 1...n$) на ближайший слева элемент со значением x, т. е.:

$$p_j = \max_{\substack{k \le j, \\ a_k = x}} k;$$

а в случае если значения с заданным x слева нет, то положим $p_j = 0$.

Например, если последовательность a такова:

$$a_1 = 0,$$
$$a_2 = 1,$$
$$a_3 = 0,$$
$$a_4 = 0,$$
$$a_5 = 1,$$
$$a_6 = 0,$$

и $x = 1$, то последовательность указателей будет следующей:

$$p_1 = 0,$$
$$p_2 = 2,$$
$$p_3 = 2,$$
$$p_4 = 2,$$
$$p_5 = 5,$$
$$p_6 = 5.$$

Вычислить эту последовательность легко одним проходом по последовательности, заметив, что $p_j = j$, когда $a_j = x$, а иначе $p_j = p_{j-1}$ (для удобства можно положить $p_0 = 0$).

Когда пригождается такая последовательность указателей? Она позволяет сэкономить время на поиск нужного элемента в задачах, где требуется большое количество данных операций. Вместо $O(n)$ времени на каждый запрос поиска мы достигаем асимптотики $O(1)$ на один запрос, но ценой предобработки за время $O(n)$.

18.2. Частичные суммы

Пусть есть последовательность чисел a_i ($i = 1...n$). Тогда определим последовательность частичных сумм s_j ($j = 0...n$) следующим образом:

$$s_j = \sum_{k=1}^{j} a_k.$$

Например, если последовательность a такова:

$$a_1 = 1,$$
$$a_2 = 2,$$
$$a_3 = -3,$$
$$a_4 = 1,$$

то последовательность частичных сумм будет следующей:

$$s_0 = 0,$$
$$s_1 = 1,$$
$$s_2 = 3,$$
$$s_3 = 0,$$
$$s_4 = 1.$$

Вычислить все частичные суммы легко одним проходом по входной последовательности, заметив, что $s_j = s_{j-1} + a_j$ для всех $j = 1...n$.

Для чего полезно знание частичных сумм? Оно позволяет за $O(1)$ вычислять сумму на любом подотрезке последовательности ценой предобработки за время $O(n)$.

В самом деле, сумму элементов на произвольном подотрезке $[l; r]$ можно выразить через разность двух частичных сумм:

$$\sum_{k=l}^{r} a_k = s_r - s_{l-1}.$$

18.3. Указатели до ближайших меньших элементов

Это менее очевидный метод предобработки, чем предыдущие два метода. Пусть есть последовательность чисел a_i ($i = 0...n - 1$). Тогда введем последовательность указателей q_j ($j = 0...n - 1$) следующим образом:

$$q_j = \max_{\substack{k<j,\\ a_k<a_j}} k;$$

а в случае если меньшего значения слева нет, то положим $q_j = -1$.

Например, если последовательность a такова:

$$a_0 = 1,$$
$$a_1 = 3,$$
$$a_2 = 2,$$
$$a_3 = 4,$$
$$a_4 = 1,$$

то последовательность указателей будет следующей:

$$q_0 = -1,$$
$$q_1 = 0,$$
$$q_2 = 0,$$
$$q_3 = 2,$$
$$q_4 = -1.$$

Каким образом можно эффективно вычислить эти указатели?

Будем просматривать последовательность a_i слева направо и пытаться параллельно подсчитывать значения q_i. Пусть текущий элемент, на котором мы остановились, имеет индекс u. Какие значения предыдущих элементов нужны для быстрого определения текущего q_u?

Ключевое наблюдение: нет смысла хранить какой-либо из предыдущих элементов, если после него уже встретился элемент с меньшим значением. В самом деле, в этой ситуации никакой указатель не может указывать на более дальний элемент, когда есть элемент ближе по позиции и меньше по значению.

Давайте реализуем эту идею, т. е. будем поддерживать структуру данных, в которой мы будем хранить все «интересные» элементы. Изначально эта структура данных будет пустой. При обработке очередного элемента последовательности — a_u —

мы будем удалять из структуры данных все элементы с бóльшими или равными значениями. После этого самый большой из элементов, оставшийся в структуре данных, даст нам искомое q_u. Последний шаг обработки элемента a_u — добавление его в эту структуру данных.

В качестве структуры данных подойдет стек (или обычный список, или массив), ведь добавляются всегда только элементы с бóльшими значениями, чем уже имеющиеся в структуре данных, а удаляются — только элементы с наибольшими значениями, т. е. последние добавленные элементы.

Это приводит нас к удивительному результату: все q_j можно вычислить за время $O(n)$ в сумме! И это несмотря на то что, формально, в решении есть вложенный цикл — тот, который удаляет лишние элементы из стека; хотя на отдельно взятой итерации этот цикл может совершить много удалений, в сумме по всем итерациям этот цикл не может удалить элементов больше, чем было добавлено — коих ровно n штук.

Реализация

Python

```
def build_closest_smaller(a):
  q = []
  stack = []
  for idx, cur in enumerate(a):
    while (stack and
           a[stack[-1]] >= cur):
      stack.pop()
    q.append(stack[-1] if stack
             else -1)
    stack.append(idx)
  return q
```

C++

```
vector<int> BuildClosestSmaller(
    const vector<int>& a) {
  vector<int> q, stack;
  for (int i = 0;
       i < (int)a.size();
       ++i) {
    while (!stack.empty() &&
           a[stack.back()] >=
              a[i]) {
      stack.pop_back();
    }
    q.push_back(stack.empty() ?
        -1 : stack.back());
    stack.push_back(i);
  }
  return q;
}
```

18.4. Списки позиций

Пусть есть последовательность чисел a_i ($i = 1...n$). Тогда составим множество всех различных значений, встречающихся в последовательности, и для каждого значения сохраним список позиций, где это значение встречается. Формально это можно описать следующим образом:

$$v_x = \{j : a_j = x\}.$$

Например, если последовательность a такова:

$$a_1 = 1,$$
$$a_2 = 2,$$
$$a_3 = 2,$$
$$a_4 = 1,$$
$$a_5 = 2,$$

то списки позиций будут следующими:

$$v_1 = (1, 4),$$
$$v_2 = (2, 3, 5).$$

В каких случаях пригождаются списки позиций? Например, они дают возможность быстро определять, где встречается какой-либо элемент, и, например, подсчитывать число вхождений на заданных подотрезках исходной последовательности.

18.5. Сжатие значений

Пусть есть последовательность чисел a_i ($i = 1...n$). Мы предполагаем, что по условию задачи числа могут принимать большие значения — например, до 10^9 или до 10^{18}.

Прием сжатия значений заключается в том, чтобы заменить эти числа более малыми значениями, сохранив, тем не менее, их относительный порядок.

Стандартный подход — построить список из всех различных значений, встречающихся в a_i, и перейти к решению задачи над последовательностью b_i ($i = 1...n$), где b_i — это индекс значения a_i в списке уникальных значений.

Прием сжатия значений чаще всего полезен в задачах, в которых приходится строить сложные структуры данных, размер которых зависит от величины помещаемых в них чисел.

18.6. Пример решения задачи. Поиск начала слова

Задача. Петя устал ворчать на современные текстовые редакторы, которые чрезвычайно медленно обрабатывают большие файлы, и начал разрабатывать свой собственный «оптидактор» (это слово он придумал для обозначения оптимизированного редактора). Одна из задач, вставшая перед Петей, следующая: дана строка текста, и требуется ответить на m запросов вида «найти позицию начала слова, содержащего символ с индексом a_i». Началом слова считается первый отличный от пробела символ; если a_i указывает на символ пробела, то ответить на запрос надо числом −1.

Входные данные состоят из текста в первой строке, числа m во второй и последовательности из m разделенных пробелом натуральных чисел a_i в третьей. Текст со-

стоит только из пробелов и символов латинского алфавита; индексы нумеруются начиная с единицы. Гарантируется, что длина текста находится в отрезке [1; $5 \cdot 10^5$], m не превосходит $5 \cdot 10^5$ и что ни один a_i не превосходит длины текста. Вывести требуется m разделенных пробелом чисел — ответы на запросы в порядке их следования.

Примеры

Входные данные	**Требуемый результат**
aba Caba 4 1 3 4 8	1 1 -1 5
Входные данные	**Требуемый результат**
X Y 3 3 2 1	3 -1 1

Решение

Если текст содержит очень длинные слова, то наивное решение — двигаться от каждого a_i влево вплоть до первого пробела — будет работать слишком долго, за $O(n \cdot m)$, где n — длина текста.

Оптимизируем это решение, предварительно подсчитав массив указателей: позицию начала слова для каждого индекса текста. Это можно сделать одним проходом за $O(n)$. После этого отвечать на запрос можно будет простым чтением из этого массива. Итоговая асимптотика составит $O(n + m)$.

Альтернативные решения

Эту задачу можно эффективно решать и другими методами, например, поиском по предварительно подсчитанному списку индексов – начал строк, или сортировкой запросов и их обработкой вместе с поддержанием индекса начала текущего слова. Однако решение с помощью указателей представляется наиболее простым.

Реализация

Python

```
text = input()
m = int(input())
a = map(int, input().split())
ptr = -1
p = []
for idx, char in enumerate(text):
  if char == " ":
    ptr = -1
```

C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
  string s;
  getline(cin, s);
  int m;
```

```
  elif ptr == -1:
    ptr = idx + 1
  p.append(ptr)
print(*[p[cur - 1] for cur in a])
```

```
  cin >> m;
  vector<int> a(m);
  for (int& cur : a) {
    cin >> cur;
    --cur;
  }
  vector<int> p;
  for (int ptr = -1, i = 0;
       i < (int)s.length(); ++i) {
    if (s[i] == ' ')
      ptr = -1;
    else if (ptr == -1)
      ptr = i + 1;
    p.push_back(ptr);
  }
  for (int cur : a)
    cout << p[cur] << ' ';
}
```

18.7. Пример решения задачи. Два подотрезка заданной длины с максимальной суммой

Задача. Призовой фонд одной викторины состоит из n призов, про каждый из которых известна его цена a_i. Обычные правила викторины таковы, что победитель получает k призов, причем выданные ему призы должны образовывать отрезок подряд идущих элементов в последовательности a_i (это сделано с целью дополнительного увеличения интриги). Для приближающегося юбилейного выпуска организаторы изменили правила: теперь победитель может выбрать сразу два отрезка, по k призов в каждом!

Спонсоры переживают, не обанкротят ли их такие правила, и просят вас написать программу, определяющую максимальный возможный выигрыш победителя.

Входные данные состоят из чисел n и k в первой строке и последовательности a_i во второй. Все числа натуральные, $2 \le n \le 10^5$, $k \le n / 2$, $|a_i| \le 10^9$. Вывести требуется единственное число — максимальный возможный выигрыш.

Примеры

Входные данные	**Требуемый результат**
5 2 1 1 5 3 4	13
Входные данные	**Требуемый результат**
4 1 10 20 5 30	50

Пояснения к примерам

В первом примере оптимальное для победителя викторины решение — взять подарки №2–3 и №4–5, что дает сумму 13. Во втором примере победителю выгодно просто взять два максимума: 30 и 20.

Решение

Одна из возможных стартовых идей — жадный алгоритм, т. е. сначала выбрать отрезок длины k с максимальной суммой, удалить его (например, заменив его элементы на минус бесконечность) и затем снова найти отрезок длины k с максимальной суммой. Однако уже на примере из условия видно, что этот алгоритм не всегда даст правильный ответ: изначально наилучшим отрезком является №3–4 с суммой 8, однако после его извлечения наилучшим отрезком в остаточной последовательности будет №1–2 с суммой 2 — что в итоге даст лишь сумму, равную 10 вместо 13.

Таким образом, нам нужен алгоритм, который каким-то образом ищет сразу оба оптимальных отрезка. Однако часто есть смысл сначала додумать до конца решение более простой задачи: выбор одного наибольшего отрезка длины k. Сразу вспомним про частичные суммы, ведь мы уже знаем про тесную связь задач такого рода с ними. В терминах частичных сумм упрощенная задача выглядит следующим образом — требуется найти:

$$\max_{u=k\ldots n} s_u - s_{u-k}.$$

В такой формулировке решение за $O(n)$ очевидно: мы перебираем всевозможные u и находим наибольшую из разностей.

Вернемся теперь к решению исходной задачи с двумя отрезками. Применим стандартный подход: будем решать задачу постепенно, добавляя числа в рассмотрение по одному. Пусть сейчас мы на j-м шаге и добавляем число a_j. Нас интересуют новые варианты выбора, появившиеся на текущем шаге, — очевидно, это случаи, когда второй выбранный отрезок оканчивается на a_j. Суммарную стоимость призов в этом отрезке легко определить с помощью частичных сумм: она будет равна $s_j - s_{j-k}$.

Хорошо, мы знаем стоимость второго отрезка на текущем шаге; теперь нам осталось найти соответствующий наилучший первый отрезок. Эту задачу с одним отрезком мы уже решили выше, единственное отличие — здесь мы должны решать ее не на всем массиве, а только среди первых $j - k$ чисел. Осталось добиться хорошей асимптотики, ведь если мы будем искать наилучший первый отрезок для каждого $j = 1\ldots n$, то у нас получится слишком медленное решение за $O(n^2)$.

Для ускорения достаточно одного наблюдения: у нас нет нужды искать наилучший первый отрезок с чистого листа, ведь при переходе от $j - 1$ к j у нас просто добавляется один новый аргумент в выражение с вычислением максимума. Иными словами, если у нас уже был найден оптимальный первый отрезок для шага $j - 1$, то при переходе к шагу j достаточно обновить старый максимум значением $s_{j-k} - s_{j-2k}$.

Итоговая асимптотика решения получается $O(n)$, поскольку решение состоит из предварительного подсчета частичных сумм и последующего одного прохода по

массиву с поддержанием лучшего первого отрезка и вычислением текущего второго отрезка.

Реализация

Python

```
from itertools import *
n, k = map(int, input().split())
a = map(int, input().split())
pref = list(accumulate(
    a, initial=0))
best_first = -float('inf')
ans = -float('inf')
for idx in range(k * 2, n + 1):
  best_first = max(
      best_first,
      pref[idx - k] -
          pref[idx - k * 2])
  second = (pref[idx] -
            pref[idx - k])
  ans = max(ans,
            best_first + second)
print(ans)
```

C++

```
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;
const int64_t INF64 =
    (int64_t)1E18;
int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  vector<int64_t> pref(n + 1);
  for (int i = 0; i < n; ++i)
    pref[i + 1] = pref[i] + a[i];
  int64_t best_first = -INF64;
  int64_t ans = -INF64;
  for (int i = k * 2; i <= n;
       ++i) {
    best_first = max(
        best_first,
        pref[i - k] -
            pref[i - k * 2]);
    auto second =
        pref[i] - pref[i - k];
    ans = max(ans,
              best_first + second);
  }
  cout << ans << endl;
}
```

18.8. Пример решения задачи. Подсчет чисел в подотрезках

Задача. Дана последовательность из n чисел a_i, и поступают запросы вида (x, l, r); ответить на запрос нужно количеством раз, которое число x встречается в подотрезке $[l; r]$ последовательности, т. е. среди $(a_l, a_{l+1}, \ldots, a_{r-1}, a_r)$.

Входные данные содержат два числа n и m в первой строке: длину последовательности и количество запросов ($1 \le n, m \le 5 \cdot 10^5$). Во второй строке следуют n чисел a_i. Последующие m строк содержат описания запросов: три числа x_j, l_j, r_j, где $1 \le l_j \le r_j \le n$. Числа a_i и x_j не превосходят по модулю 10^9. Все числа разделяются пробелом. Вывести требуется m разделенных пробелом чисел — ответы на запросы в порядке их следования.

Примеры

Входные данные	**Требуемый результат**
5 6 1 5 2 1 2 1 1 5 2 2 4 1 2 3 5 2 3 2 5 5 1 1 4	2 1 0 1 1 2
Входные данные	**Требуемый результат**
3 3 1 2 3 3 3 3 4 1 3 5 1 3	1 0 0

Пояснения к примерам

В первом примере первый запрос спрашивает количество единиц в подотрезке с первого по пятый элемент, т. е. во всей последовательности — их две; второй запрос спрашивает количество двоек в подотрезке со второго по четвертый элемент, т. е. в подпоследовательности (5, 2, 1) — двойка там одна; и т. д.

Решение

Разумеется, тривиальное решение — отвечать на каждый запрос, проходя по всем элементам последовательности с l_j-го по r_j-й — будет слишком медленным. Даже если мы его оптимизируем и будем каким-либо образом перескакивать через ненужные числа, все равно его асимптотика будет $O(n \cdot m)$. Примером плохого теста для такого решения является последовательность, целиком состоящая из одних единиц, с набором запросов, спрашивающих количество единиц на всей или почти всей последовательности.

Попытки применить здесь частичные суммы также не увенчаются успехом: поскольку различных чисел в последовательности может быть $O(n)$, то суммарный

объем списков частичных сумм для всех чисел будет $O(n^2)$. Отметим, что если бы задача была несколько другой — например, о подсчете числа букв в строке с небольшим алфавитом — то частичные суммы были бы одним из хороших способов решения этой задачи. Однако в нашей задаче с неограниченным алфавитом это решение не пройдет ни по потреблению памяти, ни по времени.

Воспользуемся методом списков позиций. Напомним, что этот метод заключается в том, чтобы для каждого из чисел, встречающихся во входной последовательности, создать список позиций, в которых оно встречается. Хотя некоторые из списков могут быть большими, суммарный объем их составит ровно n.

Как теперь отвечать на запрос, имея построенные списки позиций? Посмотрим на список позиций для числа x_j из запроса. Задача свелась к следующей: по упорядоченному списку быстро подсчитать количество элементов, значения которых лежат в отрезке $[l_j; r_j]$. Эту задачу мы уже умеем решать с помощью двоичного поиска: достаточно сделать два двоичных поиска, один по значению l_j и другой по значению r_j, и тогда ответом будет разность между найденными позициями (плюс один, в зависимости от реализации).

Общая асимптотика решения складывается из предобработки — построения списков позиций — и ответа на запросы. Предобработку легко реализовать за $O(n \log n)$, если воспользоваться структурой данных «словарь» для отображения чисел в списки позиций. Ответ на один запрос, как было показано выше, выполняется за время $O(\log n)$. Итоговая асимптотика решения составит $O(n \log n + m \log n)$.

Реализация

Python

```
from bisect import *
n, m = map(int, input().split())
a = map(int, input().split())
pos = {}
for idx, cur in enumerate(a):
  pos.setdefault(cur, [])
  pos[cur].append(idx)
for _ in range(m):
  val, left, right = map(
      int, input().split())
  val_pos = pos.get(val, [])
  begin_pos = bisect_left(
      val_pos, left - 1)
  end_pos = bisect_right(
      val_pos, right - 1)
  ans = end_pos - begin_pos
  print(ans, end=" ")
```

C++

```
#include <algorithm>
#include <iostream>
#include <map>
#include <vector>
using namespace std;
int main() {
  int n, m;
  cin >> n >> m;
  map<int, vector<int>> pos;
  for (int i = 0; i < n; ++i) {
    int cur;
    cin >> cur;
    pos[cur].push_back(i);
  }
  for (int i = 0; i < m; ++i) {
    int val, left, right;
    cin >> val >> left >> right;
    auto& val_pos = pos[val];
    auto iter_left =
```

```
        lower_bound(
            val_pos.begin(),
            val_pos.end(),
            left - 1);
    auto iter_right =
        upper_bound(
            val_pos.begin(),
            val_pos.end(),
            right - 1);
    int ans = iter_right -
              iter_left;
    cout << ans << ' ';
  }
}
```

18.9. Пример решения задачи. Подотрезок с максимальной суммой

Задача. Дано n целых чисел a_i. Требуется найти его подотрезок с наибольшей возможной суммой, т. е. такие индексы l и r, которые максимизируют следующую величину:

$$\sum_{j=l}^{r} a_j.$$

Входные данные состоят из числа n в первой строке и разделенных пробелом n чисел a_i во второй. Ограничения: $1 \le n \le 10^5$, $-10^9 \le a_i \le 10^9$. Вывести требуется два разделенных пробелом числа: искомые l и r ($1 \le l \le r \le n$). Если есть несколько вариантов ответа, вывести можно любой.

Примеры

Входные данные	**Требуемый результат**
5 -1 3 -2 5 -1	2 4
Входные данные	**Требуемый результат**
3 1 2 3	1 3

Пояснения к примерам

В первом примере искомый подотрезок — со второго по четвертый элемент, т. е. элементы (3, −2, 5), в сумме дающие 6; любые другие подотрезки имеют меньшую сумму. Во втором примере оптимальным решением является взять весь массив, получив тем самым сумму 1 + 2 + 3 = 6.

Решение №1

Отметим: если все числа входной последовательности положительны, то ответом является вся последовательность целиком. Только если в последовательности есть отрицательные числа, задача становится нетривиальной.

Для поиска решения попробуем типичный метод: добавлять числа в рассмотрение по одному слева направо и соответствующе пересчитывать текущий ответ. Пусть текущий элемент — a_j и ответ для предыдущих элементов уже подсчитан. Понятно, что для обновления ответа нам нужно учесть только все отрезки, оканчивающиеся на a_j. Каким образом сделать это эффективно?

Вспомним, что компактный способ представления сумм подотрезков — это массив частичных сумм. Напомним его определение:

$$s_k = \sum_{u=1}^{k} a_u, \quad k = 0...n.$$

В терминах частичных сумм нам требуется найти такое $l \le j$, что величина $s_j - s_{l-1}$ — максимальна. Перефразировав этот критерий, получаем, что для фиксированного j нам надо просто найти минимальную из предыдущих частичных сумм. Поскольку мы просматриваем j в порядке увеличения, то и минимальную частичную сумму (вместе с соответствующим индексом l) можно просто поддерживать в отдельной переменной.

Проиллюстрируем работу этого алгоритма на примере из условия: $a = (-1, 3, -2, 5, -1)$. Частичные суммы принимают значения: $s_0 = 0$, $s_1 = -1$, $s_2 = 2$, $s_3 = 0$, $s_4 = 5$, $s_5 = 4$. Далее мы перебираем всевозможные j:

- при $j = 1$ мы ищем $s_1 - \min_{k=0...0} s_k$, что дает $-1 - 0 = -1$ при $k = 0$;
- при $j = 2$ мы ищем $s_2 - \min_{k=0...1} s_k$, что дает $2 - (-1) = 3$ при $k = 1$;
- при $j = 3$ мы ищем $s_3 - \min_{k=0...2} s_k$, что дает $0 - (-1) = 1$ при $k = 1$;
- при $j = 4$ мы ищем $s_4 - \min_{k=0...3} s_k$, что дает $5 - (-1) = 6$ при $k = 1$;
- при $j = 5$ мы ищем $s_5 - \min_{k=0...4} s_k$, что дает $4 - (-1) = 5$ при $k = 1$.

Наибольшим значением является 6, и оно достигается при $j = 4$ и $k = 1$, из чего мы получаем ответ [2; 4] с суммой элементов 6.

Таким образом, задача решается за один проход по массиву с итоговой асимптотикой $O(n)$.

Реализация решения №1

При реализации нам даже не требуется явно хранить массив частичных сумм — достаточно лишь поддерживать текущую частичную и наименьшую из предыдущих частичных сумм.

Python

```
n = int(input())
a = map(int, input().split())
```

C++

```
#include <cstdint>
#include <iostream>
```

```
min_pref = (0, -1)
ans = (-float('inf'), -1, -1)
pref = 0
for idx, cur in enumerate(a):
  pref += cur
  cur_ans = (pref - min_pref[0],
             min_pref[1] + 1, idx)
  ans = max(ans, cur_ans)
  cur_pref = (pref, idx)
  min_pref = min(min_pref,
                 cur_pref)
print(ans[1] + 1, ans[2] + 1)
```

```
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  int64_t ans = a[0], pref = 0,
          min_pref = 0;
  int ans_left = 0, ans_right = 0,
      min_pref_pos = -1;
  for (int i = 0; i < n; ++i) {
    pref += a[i];
    int64_t cur_ans = pref -
                      min_pref;
    if (cur_ans > ans) {
      ans = cur_ans;
      ans_left = min_pref_pos + 1;
      ans_right = i;
    }
    if (pref < min_pref) {
      min_pref = pref;
      min_pref_pos = i;
    }
  }
  cout << ans_left + 1 << ' '
       << ans_right + 1 << endl;
}
```

Комментарий к реализации на Python

С целью лаконичности реализации (что в спортивном программировании всегда ценно) мы используем кортежи (*англ.* tuples) для хранения информации о наименьшей частичной сумме и о текущем ответе. В частности, `min_pref` содержит минимальную частичную сумму и позицию ее окончания; эта переменная инициализирована значением (0, −1), что обозначает частичную сумму, равную нулю и оканчивающуюся до самого первого элемента. Переменная `ans` содержит наибольшую из найденных сумм отрезков, позицию его начала и конца; эта переменная инициализирована значением (−∞, −1, −1), что обозначает бесконечно плохой ответ с несуществующими индексами. Внутри цикла мы обновляем оба кортежа `min_pref` и `ans` новыми значениями, если они лучше (меньше и больше, соответственно), чем текущие; эти операции кодируется над кортежами короче, чем при использовании отдельных переменных, как в реализации на C++.

Решение №2

У этой задачи есть и чуть более элегантное решение, которое менее очевидно, но зато чуть проще в реализации. Этот алгоритм был предложен Джеем Кадейном (Jay Kadane) приблизительно в 1977 г.

Предыстория

Интересен контекст, в котором возник этот алгоритм. Изначально Улф Гренандер начал рассматривать задачу о наибольшем подмассиве как упрощенный вариант задачи о наибольшей подматрице, которую он исследовал как один из шагов поиска образцов в изображениях. Первым рассмотренным алгоритмом было решение со скоростью $O(n^3)$, заключавшееся в переборе двух концов подотрезка и суммировании элементов. Гренандер оптимизировал его до $O(n^2)$, заметив, что текущую сумму можно поддерживать вместо вычисления ее каждый раз заново. Когда Гренандер в 1977 г. описал эту задачу Майклу Шэймосу, тот вместе с Джоном Бентли — все трое впоследствии знаменитые ученые в области информатики — за несколько дней изобрели алгоритм с временем работы $O(n \log n)$, основанный на принципе «разделяй-и-властвуй». Некоторое время они полагали, что это быстрейший возможный алгоритм, поскольку многие другие похожие задачи не могут быть решены быстрее $O(n \log n)$. Лишь позже, когда ученый-статистик Джей Кадейн узнал про эту задачу от коллег и про их решения, он буквально за одну минуту предложил алгоритм за $O(n)$, который мы и приводим ниже.

Как и решение №1, алгоритм представляет из себя один проход по массиву с накоплением текущей суммы. Отличие от решения №1 в том, что всякий раз, когда текущая сумма становится отрицательной, мы будем ее обнулять. Утверждается, что ответом на задачу, т. е. максимальной суммой подотрезка, будет наибольшая из полученных таким образом сумм.

Проиллюстрируем работу этого алгоритма на примере из условия: $a = (-1, 3, -2, 5, -1)$. Мы движемся по элементам последовательности $j = 1...5$:

- при $j = 1$ мы получаем сумму $x = -1$, которую, поскольку она отрицательна, мы заменяем на $x = 0$;
- при $j = 2$ мы получаем сумму $x = 3$;
- при $j = 3$ мы получаем сумму $x = 1$;
- при $j = 4$ мы получаем сумму $x = 6$;
- при $j = 5$ мы получаем сумму $x = 5$.

Наибольшим значением является 6, и оно достигается при $j = 4$, а последнее перед этим обнуление суммы происходило при $j = 1$, из чего мы получаем ответ [2; 4] с суммой элементов 6.

Почему этот алгоритм верен? Для краткости назовем позицию в последовательности «критической», если это позиция №1 или это позиция, непосредственно следующая за позицией обнуления суммы. Тогда алгоритм утверждает, что, во-первых, в качестве стартовых позиций достаточно рассматривать только критические позиции. И, во-вторых, в качестве конечных позиций достаточно рассматривать только позиции вплоть до первого обнуления после стартовой позиции.

Второе утверждение легко понять интуитивно: обнуление происходит, когда текущая накопленная сумма отрицательна, а, значит, продолжать увеличивать отрезок нет смысла: его сумма будет заведомо хуже из-за отрицательного префикса, чем новый отрезок, начатый со следующей позиции (рис. 18.1).

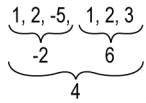

Рис. 18.1. Отрицательный префикс невыгодно брать в ответ: сумма на последующем отрезке всегда будет больше

Осталось доказать первое утверждение; докажем его от противного. Предположим, что есть отрезок $[l; r]$ с бóльшим ответом, начинающийся не в критической позиции. Тогда найдем ближайшую критическую позицию слева от l; обозначим ее через w. Поскольку l — не критическая позиция, мы можем утверждать, что сумма на отрезке $[w; l-1]$ — неотрицательна. Однако это означает, что отрезок $[w; r]$ имеет сумму, не меньшую, чем $[l; r]$; мы пришли к противоречию, что и требовалось доказать.

Этот алгоритм имеет асимптотику $O(n)$.

Реализация решения №2

Python

```
n = int(input())
a = map(int, input().split())
ans = (-float('inf'), -1, -1)
sum = 0
cur_start = 0
for idx, cur in enumerate(a):
  sum += cur
  cur_ans = (sum, cur_start, idx)
  ans = max(ans, cur_ans)
  if sum < 0:
    sum = 0
    cur_start = idx + 1
print(ans[1] + 1, ans[2] + 1)
```

C++

```
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i)
    cin >> a[i];
  int64_t ans = a[0], sum = 0;
  int ans_left = 0, ans_right = 0,
      cur_start = 0;
  for (int i = 0; i < n; ++i) {
    sum += a[i];
    if (sum > ans) {
      ans = sum;
      ans_left = cur_start;
      ans_right = i;
    }
    if (sum < 0) {
      sum = 0;
      cur_start = i + 1;
    }
  }
  cout << ans_left + 1 << ' '
       << ans_right + 1 << endl;
}
```

18.10. Пример решения задачи. Проекционная реклама

Задача. В преддверии большого мероприятия организаторы пытаются разрекламировать его всеми доступными способами, включая проекционную рекламу, т. е. установив мощный проектор, который будет светить на одно или несколько стоящих подряд зданий, отображая на них нужное рекламное изображение. Цель организаторов — создать как можно большее по площади изображение прямоугольной формы.

Вам дано описание улицы, заданное в виде карты высот фасадов зданий, стоящих по одной ее стороне. А именно, на вход подается последовательность h_i ($i = 1...n$), где h_i — высота фрагмента здания между (i −1)-м и i-м метрами, отсчитывая от начала улицы; высота равна нулю, если в этом месте нет здания (рис. 18.2).

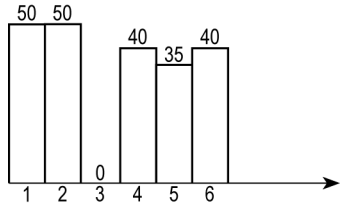

Рис. 18.2. Пример выходных данных: h_1=50, h_2=50, h_3=0, h_4=40, h_5=35, h_6=40

Входные данные содержат число n в первой строке и последовательность из разделенных пробелом n чисел h_i во второй. Гарантируется, что $1 \le n \le 5 \cdot 10^5$ и все h_i — целые неотрицательные числа, не превосходящие 10^9. Требуется вывести искомую наибольшую площадь изображения, считая, что оно целиком должно помещаться на фасадах зданий.

Примеры

Входные данные	**Требуемый результат**
6 50 50 0 40 35 35	105
Входные данные	**Требуемый результат**
1 123	123

Пояснения к примерам

В первом примере наибольшим возможным является прямоугольник шириной 3 и высотой 35 метров, размещенный начиная с третьего метра улицы; он имеет площадь 105 квадратных метров, что лучше, чем прямоугольник площадью 100 метров, возможный при старте с нулевого метра улицы. Во втором примере есть только единственный возможный вариант, он же и является ответом.

Решение

Задачу можно переформулировать следующим образом: найти

$$\max_{l \le r} \left((r - l + 1) \cdot \min_{i=l \ldots r} h_i \right).$$

Тривиальное решение заключалось бы в переборе всевозможных пар l и r и подсчете ответа для каждой из них; если поддерживать текущий минимум вместе с перебором всевозможных r для текущего l, то все решение работало бы за $O(n^2)$. Однако это слишком медленно при заданных ограничениях.

Всевозможные эвристики, такие, как движение от максимального элемента, не всегда верны. Даже на примере из условия видно, что максимальный элемент может даже не входить в оптимальное решение. Попытаемся найти решение без такого рода нечетких эвристик.

Посмотрим на задачу под другим углом: вместо перебора левых и правых границ оптимального прямоугольника рассмотрим его верхнюю границу. Эта верхняя граница «упирается» в крышу одного из зданий — если бы это было не так, то верхнюю границу можно было бы поднять и тем самым увеличить площадь.

Что если мы будем перебирать эту позицию, в которой наш прямоугольник «упирается» в крышу? Пусть текущая рассматриваемая позиция — это k с высотой h_k; теперь нам надо определить наибольший по площади прямоугольник с этой высотой. Для этого мы должны поставить левую границу как можно левее, а правую — как можно правее. Обе границы мы двигаем до тех пор, пока не встретим высоту, меньшую h_k.

Эту подзадачу — поиск ближайшего слева/справа элемента с меньшим значением — мы уже рассматривали в начале главы, и она решается для всех позиций за линейное в сумме время (напомним, решение заключается в поддержании стека из меньших элементов). Используя это решение, мы можем быстро искать для каждой позиции k оптимальную левую и правую границы, а значит, и оптимальный прямоугольник, «упирающийся» в эту позицию.

Итоговая асимптотика решения — $O(n)$.

Реализация

Для упрощения реализации мы добавляем в начало и в конец по одному фиктивному элементу со значением «−1», которое меньше любого возможного значения h_i. Эти фиктивные элементы ограничивают движение указателей влево и вправо, а также обеспечивают то, что стек никогда не становится пустым — тем самым позволяя избежать лишних проверок.

Python

```
n = int(input())
h = list(map(int, input().split()))
h = [-1] + h + [-1]

stack = [n + 1]
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
```

```
smaller_right = [None] * (n + 1)
for i in range(n, 0, -1):
  while h[stack[-1]] >= h[i]:
    stack.pop()
  smaller_right[i] = stack[-1]
  stack.append(i)

stack = [0]
ans = 0
for i in range(1, n + 1):
  while h[stack[-1]] >= h[i]:
    stack.pop()
  smaller_left = stack[-1]
  stack.append(i)
  width = (smaller_right[i] -
           smaller_left - 1)
  cur_ans = h[i] * width
  ans = max(ans, cur_ans)
print(ans)
```

```
  cin >> n;
  vector<int> h(n + 2, -1);
  for (int i = 1; i <= n; ++i)
    cin >> h[i];
  vector<int> smaller_right(n + 1);
  vector<int> stack;
  stack.push_back(n + 1);
  for (int i = n; i >= 1; --i) {
    while (h[stack.back()] >= h[i])
      stack.pop_back();
    smaller_right[i] =
        stack.back();
    stack.push_back(i);
  }
  stack.clear();
  stack.push_back(0);
  int ans = 0;
  for (int i = 1; i <= n; ++i) {
    while (h[stack.back()] >= h[i])
      stack.pop_back();
    int smaller_left =
        stack.back();
    stack.push_back(i);
    int width = smaller_right[i] -
                smaller_left - 1;
    int cur_ans = h[i] * width;
    ans = max(ans, cur_ans);
  }
  cout << ans << endl;
}
```

Примечание по дублированию кода

Приведенные выше реализации содержат дублирующийся код поиска ближайшего соседа с меньшим значением. Хотя этот код можно было бы вынести в отдельную функцию и переиспользовать ее (что и следовало бы сделать, будь это промышленным кодом), пересчет индексов для получения «ближайшего соседа справа» из ответов для отраженной задачи «ближайший сосед слева» был бы не совсем тривиальным. Аккуратная реализация пересчета индексов требует большой внимательности и, следовательно, большего времени, равно как и отладка такого кода. Поэтому, хотя такой подход и имеет право на существование, в нашей реализации выше мы предпочли простое дублирование алгоритма.

18.11. Пример решения задачи. Подотрезок с максимальным средним арифметическим

Задача. Дано n целых чисел a_i. Требуется найти такой подотрезок длины минимум m, чтобы среднее арифметическое выбранных элементов было максимально воз-

можным. Формально, нужно определить такие индексы l и r, которые удовлетворяют условию $r - l + 1 \geq m$ и максимизируют следующую величину:

$$\frac{1}{r-l+1}\sum_{j=l}^{r} a_j.$$

Входные данные состоят из чисел n и m в первой строке и n чисел a_i во второй. Ограничения: $1 \leq m \leq n \leq 10^5$, $-10^9 \leq a_i \leq 10^9$. Вывести требуется два разделенных пробелом числа: искомые l и r ($1 \leq l \leq r \leq n$). Если есть несколько вариантов ответа, вывести можно любой.

Примеры

Входные данные	**Требуемый результат**
5 2 -1 3 -2 5 -1	4 5
Входные данные	**Требуемый результат**
3 2 3 2 3	1 3

Пояснения к примерам

В первом примере искомый подотрезок — с четвертого по пятый элемент, т. е. элементы (5, −1), среднее арифметическое которых равно 2; другой вариант ответа — со второго по четвертый элемент, т. е. элементы (3, −2, 5), среднее арифметическое которых тоже 2; любые другие подотрезки длины, как минимум, два имеют меньшее среднее арифметическое. Во втором примере выгодно взять всю последовательность, поскольку ее среднее арифметическое равно (3 + 2 + 3) / 3 = 8 / 3, что больше, чем у любого подотрезка длины 2.

Решение

Отметим, что если бы на выбор подотрезка не было бы никаких дополнительных ограничений (как в нашем случае ограничение снизу на длину), то ответом всегда был бы подотрезок длины 1 с максимальным элементом.

Хотя эта задача на первый взгляд кажется похожей на предыдущую, можно заметить, что приемы из прошлой задачи, да и вообще простые жадные алгоритмы здесь не работают. Концептуальная проблема тут в том, что мы не знаем заранее коэффициента перед суммой, поскольку этот коэффициент зависит от длины выбранного подотрезка. Перебирать длину и решать задачу для конкретного коэффициента — слишком медленно, так как это решение работало бы $O(n^2)$.

Заметим, что в этой задаче можно выделить монотонную функцию. Обозначим через $f(x)$ функцию, возвращающую 0, если существует подотрезок со средним арифметическим, равным, как минимум, x, и возвращающую 1 в противном случае. Эта функция монотонна: $f(x) = 0$ для всех x, не превосходящих искомого среднего арифметического, и $f(x) = 1$ для всех бóльших x.

Это означает, что для решения задачи, т. е. для поиска значения x, равного искомому среднему арифметическому, мы можем применять двоичный поиск, как это ни неожиданно на первый взгляд. Осталось лишь научиться эффективно вычислять $f(x)$.

Один из простых способов вычисления $f(x)$ — это отнять x от каждого элемента последовательности a_i и проверить, есть ли в получившейся последовательности подотрезок с неотрицательной суммой длины, как минимум, m. Эту проверку можно реализовать, например, аналогично решению №1 предыдущей задачи: будем идти слева направо, поддерживая текущую сумму и минимум среди частичных сумм, индекс которых отстоит, как минимум, на m от текущего элемента.

Асимптотика решения будет складываться из произведения числа итераций двоичного поиска и стоимости вычисления $f(x)$. Число итераций можно ограничить точностью типа данных для операций с плавающей запятой — обозначим ее через W (на практике можно оценить $W = 2^{64}$), а стоимость вычисления $f(x)$ является $O(n)$. Итоговая асимптотика равна $O(n \log W)$.

Быстрое альтернативное решение

У этой задачи есть и более быстрое, хотя и гораздо более сложное, решение. Его детальное изложение выходит за рамки данной книги, но его основную идею можно обрисовать следующим образом. Если рассмотреть задачу в терминах частичных сумм, то ее можно переформулировать как задачу над набором прямых на плоскости, заключающуюся в поиске как можно более правой точки пересечения каждой из прямых с некоторым подмножеством предыдущих прямых. Эту задачу можно решить эффективно, поддерживая нижнюю цепь выпуклой оболочки в виде стека. Итоговая асимптотика этого решения будет $O(n)$.

Реализация

Python

```
from itertools import *

n, m = map(int, input().split())
a = list(map(int, input().split()))

def find_segment(average):
  a_decreased = list(cur - average
                     for cur in a)
  pref = list(accumulate(
      a_decreased, initial=0))
  min_pref = (float('inf'), None)
  ans = (-float('inf'), None, None)
  for idx in range(m, n + 1):
    pref_upd = (pref[idx - m],
                idx - m)
    min_pref = min(min_pref,
                   pref_upd)
```

C++

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

const double DINF = 1E100;
int n, m;
vector<int> a;

pair<int, int> find_segment(
    double average) {
  vector<double> pref(n + 1);
  for (int i = 0; i < n; ++i) {
    pref[i + 1] =
        pref[i] + (a[i] - average);
  }
```

```
    cur_ans = (
        pref[idx] - min_pref[0],
        min_pref[1], idx - 1)
    ans = max(ans, cur_ans)
  if ans[0] < 0:
    return None
  return ans[1], ans[2]

left = min(a)
right = max(a)
for _ in range(64):
  mid = (left + right) / 2
  if find_segment(mid) is None:
    right = mid
  else:
    left = mid
begin, end = find_segment(left)
print(begin + 1, end + 1)
```

```
  double min_pref = DINF,
         ans = -DINF;
  int min_pref_pos = -1;
  pair<int, int> ans_pos;
  for (int i = m; i <= n; ++i) {
    if (pref[i - m] < min_pref) {
      min_pref = pref[i - m];
      min_pref_pos = i - m;
    }
    double cur_ans =
        pref[i] - min_pref;
    if (cur_ans > ans) {
      ans = cur_ans;
      ans_pos = make_pair(
          min_pref_pos, i - 1);
    }
  }
  if (ans < 0)
    return make_pair(-1, -1);
  return ans_pos;
}

int main() {
  cin >> n >> m;
  a.resize(n);
  for (int& cur : a)
    cin >> cur;

  double left = *min_element(
      a.begin(), a.end());
  double right = *max_element(
      a.begin(), a.end());
  for (int iter = 0; iter < 64;
       ++iter) {
    double mid =
        (left + right) / 2;
    auto segm = find_segment(mid);
    if (segm.first == -1)
      right = mid;
    else
      left = mid;
  }
  auto segm = find_segment(left);
  cout << segm.first + 1 << ' '
       << segm.second + 1 << endl;
}
```

Примечание по сравнению вещественных чисел

Приведенные выше решения используют «небезопасные» сравнения вещественных чисел, в частности, в месте сравнения текущего ответа с нулем. Однако в данном случае это корректный и даже более точный подход, поскольку ответ зависит от параметра, подбираемого двоичным поиском. Поэтому даже если при каком-то значении параметра возникнут ошибки округления, дающие ответ, ошибочно чуть больший нуля, двоичный поиск автоматически немного уменьшит параметр для компенсации этих ошибок.

18.12. Пример решения задачи. Сумма в прямоугольнике

Задача. Дана прямоугольная матрица a_{ij} размером $n \times m$. Требуется ответить на q запросов вида (u_k, l_k, d_k, r_k), запрашивающих сумму в прямоугольнике $[u_k \ldots d_k; l_k \ldots r_k]$, т. е.:

$$\sum_{i=u_k}^{d_k} \sum_{j=l_k}^{r_k} a_{ij}.$$

Входные данные состоят из первой строки с числами n, m, q, затем n строк с m числами каждая — элементами a_{ij}, затем q строк с четырьмя числами u_k, l_k, d_k, r_k каждая. Все числа целые; $1 \le n, m \le 100$, $q \le 10^5$, $|a_{ij}| \le 10^9$, $1 \le u_k \le d_k \le n$, $1 \le l_k \le r_k \le m$. Вывести требуется q разделенных пробелом чисел — ответы на запросы.

Примеры

Входные данные	**Требуемый результат**
4 3 4 1 2 3 4 5 6 0 0 0 7 8 9 1 1 4 3 3 1 3 3 2 2 4 2 1 2 3 3	45 0 13 16
Входные данные	**Требуемый результат**
1 1 1 123 1 1 1 1	123

Решение

Понятно, что решение «в лоб» с суммированием элементов для каждого запроса будет работать слишком медленно.

Как это часто бывает полезным, подумаем, как мы решали бы одномерный аналог этой задачи: дан массив и требуется отвечать на запросы суммы на подотрезках. Как мы уже знаем, эту задачу удобно решать с помощью частичных сумм, благодаря чему мы можем вычислять ответ на каждый запрос за $O(1)$.

Вернемся к нашей двумерной задаче; можем ли мы обобщить метод частичных сумм для его применения и здесь? В двумерном случае появляется очевидная трудность в том, что суммирование можно вести в разных направлениях и разными способами и при этом неочевидно, к какому из них можно свести сумму на подпрямоугольнике.

Решением является суммирование «углом», т. е. частичные суммы, определенные следующим образом:

$$s_{xy} = \sum_{i=1}^{x}\sum_{j=1}^{y} a_{ij}, \quad x = 0...n, \quad y = 0...m.$$

Как выразить сумму на подпрямоугольнике $[u_k ... d_k; l_k ... r_k]$ через эти частичные суммы? Это проще всего понять графически (рис. 18.3).

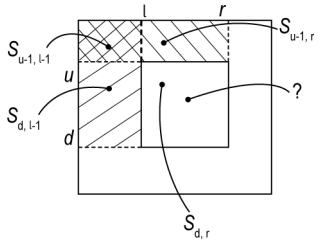

Рис. 18.3. Подрямоугольник $[u \ldots d; L ... r]$ и связанные с ним частичные суммы

Как можно увидеть по рис. 18.3, нам надо отнять от суммы в правом нижнем углу суммы перед левым нижним и правым верхним углах и затем прибавить сумму перед левым верхним углом (из-за того, что она вычтена дважды):

$$\sum_{i=u}^{d}\sum_{j=l}^{r} a_{ij} = s_{d,r} - s_{u-1,r} - s_{d,l-1} + s_{u-1,l-1}.$$

Эта формула дает нам способ отвечать на запросы за $O(1)$.

Последнее, что осталось описать, — это как быстро посчитать матрицу частичных сумм $s_{x,y}$. Это можно делать разными способами, один из них — применить анало-

гичный подход и выразить очередную сумму через значение текущего элемента и три суммы слева, сверху и по диагонали от него:

$$s_{x,y} = a_{xy} + s_{x-1,y} + s_{x,y-1} - s_{x-1,y-1}.$$

Эту формулу можно проиллюстрировать графически следующим образом (рис. 18.4).

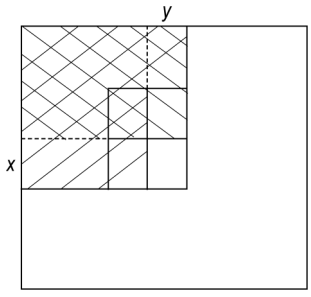

Рис. 18.4. Один элемент матрицы и сопряженные с ним частичные суммы

Реализация

Python

```
n, m, q = map(int, input().split())
a = []
for _ in range(n):
  row = list(map(
      int, input().split()))
  a.append(row)
sum = list([0] * (m + 1)
           for _ in range(n + 1))
for x in range(n):
  for y in range(m):
    sum[x + 1][y + 1] = (
        a[x][y] +
        sum[x][y + 1] +
        sum[x + 1][y] -
        sum[x][y])
for _ in range(q):
  x1, y1, x2, y2 = map(
      int, input().split())
  ans = (sum[x2][y2] -
         sum[x1 - 1][y2] -
         sum[x2][y1 - 1] +
         sum[x1 - 1][y1 - 1])
  print(ans, end=" ")
```

C++

```
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n, m, q;
  cin >> n >> m >> q;
  vector<vector<int>> a(n);
  for (auto& ax : a) {
    ax.resize(m);
    for (int& axy : ax)
      cin >> axy;
  }
  vector<vector<int64_t>> sum(
      n + 1);
  sum[0].resize(m + 1);
  for (int x = 0; x < n; ++x) {
    sum[x + 1].resize(m + 1);
    for (int y = 0; y < m; ++y) {
      sum[x + 1][y + 1] =
          a[x][y] +
          sum[x][y + 1] +
          sum[x + 1][y] -
```

```
            sum[x][y];
      }
    }
    for (int i = 0; i < q; ++i) {
      int x1, y1, x2, y2;
      cin >> x1 >> y1 >> x2 >> y2;
      int64_t ans =
          sum[x2][y2] -
          sum[x1 - 1][y2] -
          sum[x2][y1 - 1] +
          sum[x1 - 1][y1 - 1];
      cout << ans << ' ';
    }
}
```

Глава 19.
Графы. Обход в глубину

19.1. Что такое граф

Предположим, что мы имеем дело со следующей задачей. Есть несколько городов, между некоторыми из них проложена прямая дорога. Требуется узнать, можем ли мы добраться от заданного города до другого заданного города, двигаясь только по этим дорогам. Либо, например, требуется узнать, связана ли эта система городов, т. е. можно ли добраться из любого города до любого другого (рис. 19.1).

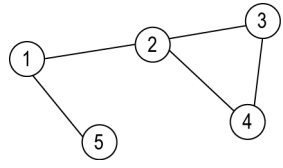

Рис. 19.1. Пример графа дорог, связывающих пять городов

Описание карты — какие города у нас есть и между какими парами городов проложена дорога — и есть граф. Заметим, что в такой постановке мы абстрагируемся от многих излишних деталей: где находятся города, каких они размеров, как именно пролегают дороги на всей их протяженности и т. п.

В широком смысле граф — это набор вершин и набор ребер, связывающих эти вершины. В такой абстрактной формулировке можно смоделировать множество различных задач, не обязательно связанных с географическими картами, например:

- граф может описывать набор людей, где для каждого человека известно, с кем он знаком лично. Целью задачи здесь, например, может быть узнать расстояние между двумя заданными людьми в виде числа «рукопожатий»;
- или же графом может быть описание набора уровней в компьютерной игре и того, между какими уровнями можно перемещаться. Целью задачи, например, может быть поиск кратчайшего маршрута, посещающего все уровни хотя бы по одному разу;
- или же граф может описывать структуру онлайн-энциклопедии, где вершины — это статьи энциклопедии, а ребра — ссылки между статьями. Примером задачи здесь может быть добраться от одной заданной статьи до другой за наименьшее число переходов по ссылке.

Мощь абстракции «граф» в том, что в ее терминах можно выразить множество различных моделей реального или математического мира, абстрагируясь от ненужных деталей и применяя общие графовые алгоритмы и методы анализа.

19.2. Ориентированные и неориентированные графы

Граф называется неориентированным, если по каждому ребру можно пройти в обоих направлениях. В ориентированном же графе (кратко — орграфе) у каждого ребра зафиксировано направление, вдоль которого можно двигаться.

> Можно было бы предположить, что неориентированный граф — частный случай ориентированного графа, в котором между каждой парой вершин либо нет ни одного ребра, либо есть два ребра в каждом из направлений. В некоторых задачах такое обобщение верно, однако есть случаи, когда оно работать не будет. Например, с точки зрения наличия циклов одно неориентированное ребро — не цикл, в то время как пара взаимно противоположных ориентированных ребер — цикл.

В каких задачах возникают ориентированные, а в каких — неориентированные графы? Например, при рассмотрении карты городов, связанных между собой шоссе, естественно рассматривать неориентированный граф, поскольку шоссе в реальном мире двусторонние (обычно). А вот для схемы улиц города может понадобиться ориентированный граф, поскольку нередко встречаются улицы с односторонним движением. Другой пример ориентированного графа — карты ссылок между вебстраницами: по гиперссылке можно перейти с одной страницы на другую, но не в обратном направлении.

19.3. Способы представления графов в компьютере

Перед тем как переходить к первым графовым алгоритмам, обсудим то, каким образом хранить граф в памяти компьютера.

19.3.1. Обозначения вершин

Давайте договоримся обозначать вершины графа (*англ.* vertex) натуральными числами; иными словами, пусть вершины пронумерованы числами от 1 до n, где n — общее число вершин в графе. Никакие две вершины не должны иметь одинаковый номер.

С программной точки зрения этот способ обозначения удобен тем, что набор вершин не требуется хранить явно: это просто диапазон целых чисел.

> Этот прагматичный подход несколько отличается от принятого в математике стиля, в котором вершины графа вводятся через абстрактные элементы некоторого множества вершин.

Единственная тонкость: в программном коде обычно удобна 0-индексация (т. е. нумерация начиная с нуля), в то время как 1-индексация удобнее в словесных описаниях примеров графов и алгоритмов. Кроме того, по традиции форматы входных и выходных данных в задачах спортивного программирования используют 1-

индексацию. Стандартное решение этой несовместимости — это программировать в 0-индексации, при необходимости уменьшая номера на единицу при чтении входных данных и увеличивая их на единицу при печати результата.

19.3.2. Способы представления ребер

Для хранения ребер (*англ.* edge) в памяти компьютера есть несколько вариантов, и оптимальный выбор здесь менее однозначен. Опишем ниже два самых распространенных представления.

> Сразу отметим, что самый прямолинейный способ хранения ребер — в виде одного большого списка всех ребер графа — мы не рассматриваем здесь, поскольку очень мало алгоритмов, для которых это представление удобно и естественно.

Представление №1. Матрица смежности

Этот способ хранения графа заключается в создании квадратной таблицы $n \times n$, где n — число вершин в графе. Каждая ячейка g_{ij} матрицы смежности имеет логический (булев) тип и равна «истине», если между вершинами i и j есть ребро.

Например, рассмотрим следующий ориентированный граф (рис. 19.2).

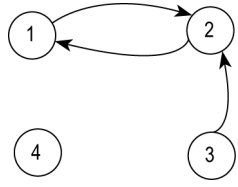

Рис. 19.2. Пример ориентированного графа с 4 вершинами и 3 ребрами

Матрица смежности для него будет выглядеть следующим образом:

$$g = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

где *1* обозначает «истину» (т. е. наличие ребра), *0* — «ложь» (отсутствие ребра). Например, $g_{12} = g_{21} = 1$, поскольку между вершинами 1 и 2 есть ребра в обоих направлениях, а $g_{23} = 0$, поскольку нет ребра из вершины 2 в вершину 3 (оно есть только в обратном направлении, поэтому $g_{32} = 1$).

> ***Наличие ребра или наличие пути?***
>
> Обратите внимание: матрица смежности определяется информацией о ребрах, а не о путях. В примере выше $g_{31} = 0$, несмотря на то что из вершины 3 можно добраться до вершины 1 через вершину 2. Повторимся: ячейки матрицы смежности сообщают о наличии или отсутствии прямого ребра.

Другой пример, с неориентированным графом (рис. 19.3).

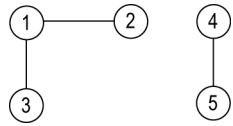

Рис. 19.3. Пример неориентированного графа с 5 вершинами и 3 ребрами

Матрица смежности для него будет выглядеть следующим образом:

$$g = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Два простых свойства матрицы смежности: главная диагональ заполнена нулями (предполагая отсутствие ребер из вершины в нее же, т. е. «петель»), а в случае неориентированного графа матрица смежности симметрична относительно диагонали.

Представление №2. Списки смежности

Списки смежности используют другой подход: для каждой вершины составляется список ее соседей, т. е. таких вершин, в которые можно попасть, пройдя по одному ребру.

Например, для этого ориентированного графа (рис. 19.4) списки смежности будут выглядеть следующим образом:

$$1 : (2,3),$$
$$2 : (1),$$
$$3 : (2),$$
$$4 : ().$$

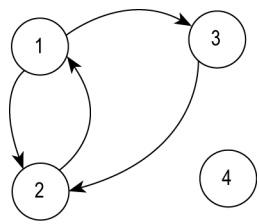

Рис. 19.4. Пример ориентированного графа

Здесь список смежности для вершины 1 содержит вершины 2 и 3, поскольку граф содержит ребра (1, 2) и (1, 3).

Для этого неориентированного графа (рис. 19.5) списки смежности будут выглядеть следующим образом:

1 : (2,3),
2 : (1,4,5),
3 : (1),
4 : (2,5),
4 : (2,4).

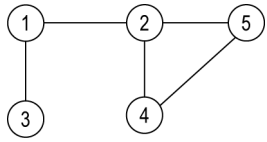

Рис. 19.5. Пример неориентированного графа

Что лучше: матрица или списки смежности?

На этот вопрос нет однозначного ответа: все зависит от задачи и используемого алгоритма.

Ключевым соображением здесь является потребление памяти. Для матрицы смежности всегда нужно $O(n^2)$ байт. Списки смежности занимают лишь $O(n + m)$ байт в сумме, где m — количество ребер. Сравнивая эти два представления, получаем:

- для графов с небольшим числом ребер ($m << n^2$; такие графы называют **разреженными**) списки смежности — это гораздо более компактное представление;
- для графов же с большим числом ребер (такие графы называют **плотными**) эти два способа примерно равноценны. При очень большом числе ребер матрица смежности даже станет более компактной, поскольку ее n^2 ячеек имеют булев тип, а потому она требует ровно n^2 байт либо даже лишь $n^2 / 8$ байт при использовании побитового хранения.

Есть и другой аспект — например, то, насколько быстро можно проверить наличие ребра между заданной парой вершин:

- матрица смежности легко позволяет сделать это за $O(1)$: достаточно лишь обратиться к значению соответствующей ячейки матрицы;
- в случае же списков смежности потребуется поиск по всему списку смежности — что может быть долго, вплоть до $O(n)$. Эту операцию можно ускорить до $O(\log n)$ с помощью двоичного поиска, для чего потребуется предварительная сортировка, либо до $O(1)$ в среднем при использовании хеш-таблиц (но с гораздо большей константой, чем при использовании матрицы смежности).

Наконец, еще один фактор — некоторые алгоритмы удобнее программируются при использовании матрицы смежности, чем при использовании списков смежности; это, однако, скорее частный случай.

Подводя итог: в большинстве задач можно применять списки смежности; они достаточно экономичны по памяти и достаточно быстры. В более редких случаях матрица смежности может быть более целесообразной (чуть более простой для реализации какого-либо алгоритма или более компактной в случае очень плотных графов и т. п.), однако ее применимость ограничена объемом требуемой памяти.

19.3.3. Реализация матрицы смежности

Приведем для наглядности, как реализовать построение матрицы смежности орграфа по заданному списку ребер.

Мы предполагаем, что на вход программе подается число вершин и ребер в первой строке и описания ребер в последующих строках (содержащих по паре натуральных чисел — номера вершин-концов ребер). В целях демонстрации программа будет выводить получившуюся матрицу.

Python

```
vertex_count, edge_count = map(
    int, input().split())
adj_matr = [
    [False] * vertex_count
    for _ in range(vertex_count)]
for _ in range(edge_count):
  vertex_from, vertex_to = map(
      int, input().split())
  vertex_from -= 1
  vertex_to -= 1
  adj_matr[vertex_from][
      vertex_to] = True

print(adj_matr)
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int vertex_count, edge_count;
  cin >> vertex_count
      >> edge_count;
  vector<vector<char>> adj_matr(
      vertex_count,
      vector<char>(vertex_count));
  for (int i = 0;
       i < edge_count;
       ++i) {
    int vertex_from, vertex_to;
    cin >> vertex_from
        >> vertex_to;
    --vertex_from;
    --vertex_to;
    adj_matr[vertex_from][
        vertex_to] = true;
  }

  for (int i = 0;
       i < vertex_count;
       ++i) {
    for (int j = 0;
         j < vertex_count;
         ++j) {
```

```
      cout << (bool)adj_matr[i][j]
           << ' ';
    }
    cout << endl;
  }
}
```

Примечание о типе данных vector<char> в C++

Мы используем этот тип данных вместо vector<bool>, поскольку последний использует технику битового сжатия (в одном байте хранятся сразу 8 элементов, по одному в каждом бите), из-за чего работает существенно медленнее. Хотя при больших *n* техника битового сжатия позволит сильно — в 8 раз — сэкономить память, в остальных случаях использование vector<char> более целесообразно.

Изменения для неориентированных графов

В случае если граф неориентированный, в приведенных реализациях надо просто добавить одну строку: помимо записи в `adj_matr[vertex_from][vertex_to]` также устанавливать значение `adj_matr[vertex_to][vertex_from]`.

Альтернатива — использование массивов в C++

Приведенная выше реализация на C++ использует вложенный vector. Вместо этого можно было бы использовать двумерный массив, т. е. объявить переменную следующим образом: `bool adj_matr[MAX_VERTICES][ MAX_VERTICES]`, где `MAX_VERTICES` — наибольшее возможное в данной задаче число вершин. Это допустимая альтернатива, которая иногда позволяет сделать код чуть лаконичнее. Однако у нее есть и несколько недостатков: зависимость от правильного выбора константы `MAX_VERTICES`, вероятная необходимость использования глобальных переменных (так как слишком большие локальные переменные могут вызвать переполнение стека), необходимость очистки глобальных переменных при работе с несколькими графами. Поэтому мы и в дальнейшем будем использовать вложенный vector.

19.3.4. Реализация списков смежности

Приведем реализацию построения списков смежности для орграфа. Формат входных тот же, что и в предыдущей секции. В целях демонстрации программа выводит получившиеся списки смежности.

Python

```
vertex_count, edge_count = map(
    int, input().split())
adj_list = [[] for _ in
            range(vertex_count)]
for _ in range(edge_count):
  vertex_from, vertex_to = map(
      int, input().split())
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int vertex_count, edge_count;
  cin >> vertex_count
      >> edge_count;
```

```
  vertex_from -= 1
  vertex_to -= 1
  adj_list[vertex_from].append(
      vertex_to)

print(adj_list)
```

```
  vector<vector<int>> adj_list(
      vertex_count);
  for (int i = 0;
       i < edge_count;
       ++i) {
    int vertex_from, vertex_to;
    cin >> vertex_from
        >> vertex_to;
    --vertex_from;
    --vertex_to;
    adj_list[
        vertex_from].push_back(
            vertex_to);
  }

  for (int i = 0;
       i < vertex_count;
       ++i) {
    for (int vertex_to :
         adj_list[i]) {
      cout << vertex_to << ' ';
    }
    cout << endl;
  }
}
```

Изменения для неориентированных графов

В случае если граф неориентированный, в приведенных реализациях надо просто добавить одну строку: помимо добавления `vertex_to` в `adj_list[vertex_from]` также добавлять `vertex_from` в `adj_list[vertex_to]`.

Порядок элементов в списках смежности

Приведенные реализации создают неотсортированные списки смежности: соседи каждой вершины следуют в них в произвольном порядке, зависящем от порядка следования входных данных. Впрочем, для многих графовых алгоритмов никакого определенного порядка и не требуется.

Альтернатива — использование массивов в C++

Приведенная выше реализация на C++ использует вложенный vector. Вместо этого можно было бы использовать массив векторов, т. е. объявить переменную следующим образом: `vector<int> adj_list[MAX_VERTICES]`, где `MAX_VERTICES` — наибольшее возможное в данной задаче число вершин. Как и в предыдущем разделе, такая альтернатива будет чуть более лаконичной в плане заполнения графа, однако взамен приобретет недостатки массивов и глобальных переменных, указанные ранее; кроме того, в отладочном режиме программа может очень сильно замедлиться. Поэтому в книге мы и в дальнейшем будем использовать вложенный vector.

19.4. Алгоритм обхода в глубину

Обход в глубину (*англ.* depth-first search, или, сокращенно, dfs) — самый простой алгоритм обхода графа. Входными параметрами для него являются граф и стартовая вершина. Алгоритм заключается в следующем: перебрать все ребра, исходящие из стартовой вершины, и рекурсивно запустить себя из каждой. По окончании работы алгоритм обойдет все вершины и ребра, достижимые из стартовой вершины.

Ключевая деталь, делающая этот алгоритм быстрым, — пропуск уже посещенных вершин. Для этого вводится дополнительный массив из n булевых переменных, в которых хранится для каждой вершины, посещал ли обход в глубину ее или нет. Рекурсивные запуски будем производить только из тех вершин, которые еще не помечены как посещенные.

Время работы алгоритма обхода в глубину зависит от способа представления графа; асимптотики мы приведем ниже, при разборе конкретных реализаций. Но самое главное то, что обход в глубину посещает каждую вершину не более одного раза и при каждом посещении просматривает список исходящих ребер только один раз.

19.4.1. Демонстрация обхода в глубину

Рассмотрим следующий граф (рис. 19.6).

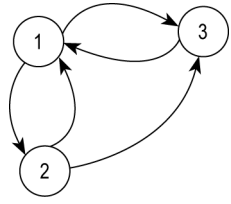

Рис. 19.6. Пример ориентированного графа для демонстрации обхода в глубину

Проследим работу алгоритм обхода в глубину при запуске из вершины №1 (рис. 19.7).

- Начинаем обход в глубину из вершины №1. Помечаем ее как уже посещенную. Начинаем просматривать список исходящих из нее ребер: это ребра в вершины №2 и №3.
- Проходим по первому ребру из вершины №1: оно ведет в вершину №2, которая еще не посещена, поэтому рекурсивно запускаем обход в глубину из вершины №2.
  - Начинаем обход в глубину из вершины №2. Помечаем ее как уже посещенную.
  - Проходим по первому ребру из вершины №2: оно ведет в вершину №1, которая уже посещена, поэтому пропускаем ее.

- Проходим по второму ребру из вершины №2: оно ведет в вершину №3, которая еще не посещена, поэтому рекурсивно запускаемся из вершины №3.
  - Начинаем обход в глубину из вершины №3. Помечаем ее как уже посещенную.
  - Проходим по первому ребру из вершины №3: оно ведет в вершину №1, которая уже посещена, поэтому пропускаем ее.
  - Больше ребер, исходящих из вершины №3, нет, поэтому завершаем обход из вершины №3.
- Больше ребер, исходящих из вершины №2, нет, поэтому завершаем обход из вершины №2.

♦ Проходим по второму ребру из вершины №1: оно ведет в вершину №3, которая уже посещена, поэтому пропускаем ее.

♦ Больше ребер, исходящих из вершины №1, нет, поэтому завершаем обход из вершины №1.

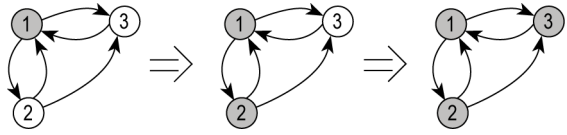

Рис. 19.7. Демонстрация обхода указанного графа в глубину

Почему обход называется обходом «в глубину»

На этом примере уже можно увидеть, почему алгоритм имеет такое название: он как бы «углубляется» внутрь графа вместо равномерного движения по всем направлениям из стартовой вершины. Можно провести и другую аналогию: обход в глубину эквивалентен «правилу одной руки» для поиска выхода из лабиринта — согласно этому правилу, мы на каждой развилке лабиринта должны поворачивать в одну и ту же сторону.

Влияние порядка ребер

Как можно заметить, то, по каким именно путям будет следовать обход в глубину, зависит от порядка просмотра соседей каждой вершины. В примере выше мы просматривали соседей в порядке возрастания их номеров. Однако при другом порядке просмотра обход в глубину мог пойти из вершины №1 сначала в вершину №3 и только затем в вершину №2. На более сложных графах варианты обхода в глубину могут различаться и более кардинально. Однако независимо от порядка обхода главная цель — посещение всех достижимых вершин и ребер — будет достигнута.

19.5. Реализация обхода в глубину

Рассмотрим реализации обхода графа в глубину с использованием матрицы смежности и списка смежности.

19.5.1. Обход в глубину на матрице смежности

Приведем реализацию обхода в глубину, когда граф задан своей матрицей смежности, а стартовая вершина — вершина №1 (что соответствует индексу «0» в программном коде). Реализацию создания матрицы смежности мы не приводим здесь повторно.

Python

```
def dfs(vertex):
  visited[vertex] = True
  for to in range(vertex_count):
    if (adj_matr[vertex][to] and
        not visited[to]):
      dfs(to)

# ... чтение vertex_count,
# adj_list ...

visited = [False] * vertex_count
dfs(0)

print(visited)
```

C++

```
#include <iostream>
#include <vector>
using namespace std;

int vertex_count;
vector<vector<char>> adj_matr;
vector<char> visited;

void Dfs(int vertex) {
  visited[vertex] = true;
  for (int to = 0;
       to < vertex_count;
       ++to) {
    if (adj_matr[vertex][to] &&
        !visited[to]) {
      Dfs(to);
    }
  }
}

int main() {
  // ... чтение vertex_count,
  // adj_list ...

  visited.resize(vertex_count);
  Dfs(0);

  for (int i = 0;
       i < vertex_count;
       ++i) {
    cout << (bool)visited[i]
         << ' ';
  }
}
```

Комментарий об использовании глобальных переменных

Приведенные выше реализации не передают общую информацию (матрицу смежности `g`, число вершин `n`, пометки посещений `visited`) в функцию обхода в глубину, а используют для этой цели глобальные переменные. Это снова тот случай, когда прием, дающий краткость реализации и потому полезный для демонстрации алгоритма и в спортивном программировании вообще, не следует «мерить аршином» промышленного программирования.

Ограничение на глубину рекурсии

Обход в глубину — это алгоритм, требовательный в плане потребления стека рекурсивных вызовов. В худшем случае — если граф связный — глубина рекурсии достигнет n. В типичной задаче (с сотнями тысяч вершин) и стандартными настройками стека (порядка одного или нескольких мегабайт) может произойти аварийное завершение работы из-за переполнения стека. В языке C++ при использовании компилятора MSVC размер стека можно увеличить из программы с помощью директивы `#pragma`, а в остальных случаях придется полагаться на то, что тестирующая система использует специальные флаги для увеличения стека. К сожалению, в PyPy нет простого способа увеличить размер стека, что является серьезным препятствием к использованию обхода в глубину.

Время работы обхода в глубину при использовании матрицы смежности составляет $O(n^2)$, поскольку в худшем случае мы посещаем все n вершин, а для каждой вершины требуется пройти по всей строке матрицы смежности, длина которой равна n.

19.5.2. Обход в глубину на списках смежности

Аналогично предыдущей секции, мы предполагаем, что чтение графа в списки смежности уже реализовано и что стартовой является вершина №1. В целях демонстрации мы выводим массив пометок посещенных вершин.

Python

```
def dfs(vertex):
  visited[vertex] = True
  for to in adj_list[vertex]:
    if not visited[to]:
      dfs(to)

# ... чтение vertex_count,
# adj_list ...

visited = [False] * vertex_count
dfs(0)
```

C++

```
#include <iostream>
#include <vector>
using namespace std;

int vertex_count;
vector<vector<int>> adj_list;
vector<char> visited;

void Dfs(int vertex) {
  visited[vertex] = true;
  for (int to : adj_list[vertex]) {
    if (!visited[to])
      Dfs(to);
  }
}
```

```
int main() {
  // ... чтение vertex_count,
  // adj_list ...

  visited.resize(vertex_count);
  Dfs(0);

  for (int i = 0;
       i < vertex_count;
       ++i) {
    cout << (bool)visited[i] << ' ';
  }
}
```

Комментарий об использовании глобальных переменных

Аналогично реализациям в предыдущей секции, мы используем глобальные переменные с целью достижения компактности кода.

Время работы обхода в глубину при использовании списков смежности составляет $O(n + m)$, где n — число вершин, m — число ребер. Такая асимптотика получается на основе того факта, что обход в глубину не более одного раза проходит каждый из списков смежности, а суммарный размер списков смежности составляет m (или $2m$ в случае неориентированных графов).

Сравнивая эту асимптотику с асимптотикой при использовании матрицы смежности, получаем, что использование списков смежности, как минимум, не хуже, а то и предпочтительнее. В случае разреженных графов, т. е. когда $m << n^2$, преимущество в скорости работы будет становиться существенным.

19.6. Пример решения задачи. Проверка наличия пути

Задача. Дана карта авиаперелетов, которые связывают между собой аэропорты в различных городах; авиаперелеты однонаправленны, т. е. с помощью одного перелета можно добраться из одного города до другого, но не в обратную сторону. Требуется определить, можно ли добраться из города x в город y — с использованием одного или нескольких авиаперелетов.

Входные данные содержат четыре разделенных пробелом числа n, m, x, y: число городов, авиаперелетов, номер стартового города и номер конечного города (города нумеруются от 1 до n) соответственно. Ограничения: $2 \le n \le 10^5$, $0 \le m \le 10^5$, $x \ne y$. Последующие m строк содержат по два разделенных пробелом числа a_i и b_I ($a_i \ne b_i$), задающих город отправления и город прибытия каждого авиаперелета. Требуется вывести «YES» или «NO» в зависимости от того, существует ли требуемый путь.

Примеры

Входные данные	**Требуемый результат**
4 4 3 4 1 2 2 3 3 1 2 4	YES
Входные данные	**Требуемый результат**
4 2 1 3 3 2 2 1	NO

Пояснения к примерам

В первом примере существует следующий путь: 3 → 1 → 2 → 4. Во втором примере пути из города №1 в город №3 нет (хотя есть путь в обратном направлении).

Решение

Для решения этой задачи надо просто запустить обход в глубину из вершины x. Если по его окончании вершина y оказалась помеченной как посещенная, то искомый путь существует, иначе — нет.

Отметим, что при реализации следует использовать списки смежности, поскольку матрица смежности потребует слишком много памяти и времени при заданных ограничениях.

Асимптотика решения — $O(n + m)$.

Реализация

Отметим, что здесь мы будем использовать чуть более лаконичную реализацию алгоритма обхода в глубину: пошаговое объяснение алгоритма было дано в предыдущих разделах, а в контексте спортивного программирования краткие (в основном однобуквенные) названия переменных ускоряют написание кода.

Python

```
def dfs(v):
  u[v] = True
  for to in g[v]:
    if not u[to]:
      dfs(to)

n, m, x, y = map(
    int, input().split())
```

C++

```
#include <iostream>
#include <vector>
using namespace std;

int n;
vector<vector<int>> g;
vector<char> u;
```

```
x -= 1
y -= 1
g = [[] for _ in range(n)]
for _ in range(m):
  v1, v2 = map(
      int, input().split())
  v1 -= 1
  v2 -= 1
  g[v1].append(v2)
u = [False] * n
dfs(x)
print("YES" if u[y] else "NO")
```

```
void dfs(int v) {
  u[v] = true;
  for (int to : g[v]) {
    if (!u[to])
      dfs(to);
  }
}

int main() {
  int m, x, y;
  cin >> n >> m >> x >> y;
  --x;
  --y;
  g.resize(n);
  for (int I = 0; I < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1].push_back(v2);
  }
  u.resize(n);
  dfs(x);
  cout << (u[y] ? "YES" : "NO")
       << endl;
}
```

19.7. Пример решения задачи. Конная прогулка

Задача. Дано поле размером $n \times m$ клеток. В одной из его клеток находится конюшня, а остальные клетки либо пусты, либо содержат болото, т. е. непроходимы. Пустая клетка называется прогулочной, если ее можно достичь, стартовав из конюшни и двигаясь по правилам шахматного коня (т. е. прыгая на расстояние 1 по одной координате и 2 по другой), не выходя за пределы поля и не наступая в клетки с болотом. Подсчитайте количество прогулочных клеток.

Входные данные состоят из пары натуральных чисел n и m в первой строке и описания поля в последующих n строках, с m символов в каждой. В описании поля используются следующие символы:

- точка («.»), обозначающая пустую клетку;
- решетка («#»), обозначающая болото;,
- заглавная буква икс («X»), обозначающая конюшню.

Гарантируется, что $1 \le n, m \le 100$ и описание поля содержит ровно один символ «X».

Вывести требуется единственное число — количество прогулочных клеток.

Примеры

Входные данные	**Требуемый результат**
3 4 ..## X...	7
Входные данные	**Требуемый результат**
3 3 #X# ###	0

Пояснения к примерам

В первом примере можно, стартовав из клетки «X» и делая шаги шахматным конем, достичь всех клеток «.», кроме двух (кроме второй клетки второй строки и четвертой клетки третьей строки); поэтому ответ равен 7. Во втором примере нет ни одной пустой, а значит, и прогулочной, клетки.

Решение

Если бы поле состояло только из пустых клеток, то, если его размер достаточно большой, ответом было бы общее количество клеток: при наличии достаточного пространства для маневра шахматный конь может обойти все клетки. Однако наличие непроходимых клеток все меняет: целые области пустых клеток могут быть окружены «стеной» из непроходимых клеток, или же могут возникать узкие «коридоры», в которых коню негде развернуться, чтобы обойти их целиком. Поэтому требуется решение, которое будет по-настоящему моделировать движение коня и искать все достижимые клетки.

Эту задачу можно свести к графовой: пусть клетки поля будут вершинами графа, а возможные ходы конем из одной клетки в другую — ребрами. В таком графе будет $n \cdot m$ вершин, и из каждой вершины будет выходить от нуля до восьми ребер. Отметим, что ребер, ведущих за пределы поля или в клетку-болото, быть не должно.

Тогда задача сводится к следующей: посчитать число вершин графа, достижимых из стартовой вершины. А для решения этой задачи достаточно запустить обход в глубину и подсчитать число вершин, помеченных как посещенные (ответ будет на единицу меньше, поскольку клетка-конюшня, по условию, не учитывается).

Асимптотика решения составит $O(nm)$, поскольку граф содержит nm вершин и не более $8nm$ ребер.

Реализация

При реализации заметим, что нам нет нужды явно строить и хранить граф, равно как и нумеровать вершины в сквозной нумерации. Лаконичнее и эффективнее реализовать обход в глубину, передавая в него координаты текущей клетки и генерируя возможные переходы в другие клетки внутри функции обхода в глубину.

Один из чисто реализационных вопросов — как закодировать все 8 возможных ходов коня. Наш подход использует стандартную идею с константным массивом «дельт», описывающим каждый из 8 возможных ходов в виде соответствующих ∂_x и ∂_y.

Python

```
DX = [1,2,1,2,-1,-2,-1,-2]
DY = [2,1,-2,-1,2,1,-2,-1]

n, m = map(int, input().split())
field = [input() for _ in range(n)]
visited = [[False] * m for _ in
           range(n)]

def dfs(x, y):
  visited[x][y] = True
  for dir in range(8):
    nx = x + DX[dir]
    ny = y + DY[dir]
    if (0 <= nx < n and
        0 <= ny < m and
        field[nx][ny] == '.' and
        not visited[nx][ny]):
      dfs(nx, ny)

for x, row in enumerate(field):
  for y, cell in enumerate(row):
    if cell == 'X':
      dfs(x, y)
marked = sum(sum(row) for row in
             visited)
print(marked - 1)
```

C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

const int DX[] = {
    1,2,1,2,-1,-2,-1,-2};
const int DY[] = {
    2,1,-2,-1,2,1,-2,-1};

int n, m;
vector<string> field;
vector<vector<char>> visited;

void dfs(int x, int y) {
  visited[x][y] = true;
  for (int dir = 0; dir < 8;
       ++dir) {
    int nx = x + DX[dir];
    int ny = y + DY[dir];
    if (0 <= nx && nx < n &&
        0 <= ny && ny < m &&
        field[nx][ny] == '.' &&
        !visited[nx][ny]) {
      dfs(nx, ny);
    }
  }
}

int main() {
  cin >> n >> m;
  field.resize(n);
  for (auto& row : field)
    cin >> row;
```

```
  visited.assign(
      n, vector<char>(m));
  for (int x = 0; x < n; ++x) {
    for (int y = 0; y < m; ++y) {
      if (field[x][y] == 'X')
        dfs(x, y);
    }
  }
  int marked = 0;
  for (int x = 0; x < n; ++x) {
    for (int y = 0; y < m; ++y)
      marked += visited[x][y];
  }
  cout << marked - 1 << endl;
}
```

19.8. Пример решения задачи. Проверка связности

Определение. Неориентированный граф называется связным, если из любой его вершины можно добраться до всех остальных.

Отметим, что это определение связности применимо только к неориентированным графам. Ориентированные графы используют несколько другую терминологию, которую мы рассмотрим позже.

Задача. Для заданного неориентированного графа определить, является ли он связным или нет.

Входные данные состоят из числа вершин n и ребер m в первой строке ($1 \le n, m \le 10^5$) и описаний ребер в последующих m строках, содержащих по паре чисел — номеров вершин (натуральных чисел, не превосходящих n). Все ребра различны, и никакое ребро не ведет из вершины в нее же. Вывести требуется «YES» или «NO» в зависимости от того, связен граф или нет.

Примеры

Входные данные	**Требуемый результат**
3 2 1 3 2 3	YES
Входные данные	**Требуемый результат**
5 3 1 4 2 3 5 2	NO

Пояснения к примерам:

В первом примере из любой вершины можно добраться до двух остальных. Во втором примере из вершин №1 и 4 нельзя добраться до вершин №2, 3, 5.

Решение

Для решения достаточно произвести обход в глубину из любой вершины (например, вершины №1) и по его окончании проверить, что все вершины графа помечены как посещенные.

Время работы алгоритма составит $O(n + m)$.

Реализация

Python

```
def dfs(v):
  visited[v] = True
  for to in g[v]:
    if not visited[to]:
      dfs(to)

n, m = map(int, input().split())
g = [[] for _ in range(n)]
for _ in range(m):
  v1, v2 = map(int,
               input().split())
  v1 -= 1
  v2 -= 1
  g[v1].append(v2)
  g[v2].append(v1)

visited = [False] * n
dfs(0)
print("YES" if all(visited)
      else "NO")
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int n;
vector<vector<int>> g;
vector<char> visited;

void dfs(int v) {
  visited[v] = true;
  for (int to : g[v]) {
    if (!visited[to])
      dfs(to);
  }
}

int main() {
  int m;
  cin >> n >> m;
  g.resize(n);
  for (int i = 0; i < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1].push_back(v2);
    g[v2].push_back(v1);
  }
  visited.resize(n);
  dfs(0);
  bool connected = !count(
```

```
      visited.begin(),
      visited.end(), false);
  cout << (connected ?
               "YES" : "NO")
       << endl;
}
```

19.9. Пример решения задачи. Проверка двудольного графа

Определение. Граф называется двудольным (*англ.* bipartite), если его вершины можно разбить на две доли (группы) так, что концы каждого ребра лежат в разных долях.

Можно доказать, что граф является двудольным тогда и только тогда, когда в нем нет циклов нечетной длины.

Задача. Дан неориентированный граф. Требуется проверить, является ли он двудольным.

Входные данные состоят из числа вершин и ребер в первой строке и описаний ребер в последующих строках. Каждое ребро задается номерами вершин-концов в 1-индексации. Число вершин положительно, число вершин и ребер не превосходит 10^5, и петли отсутствуют (т. е. нет ребер, ведущих из вершины в нее же). Вывести требуется «0», если граф не двудольный; в противном случае — последовательность из разделенных пробелом чисел «1» или «2», указывающих номер доли для каждой вершины. Если решений несколько, вывести можно любое.

Примеры

Входные данные	**Требуемый результат**
5 5 1 2 1 3 4 5 2 4 3 4	1 2 2 1 2
Входные данные	**Требуемый результат**
4 3 2 3 3 4 2 4	0

Пояснения к примерам

В первом примере вершины №1 и 4 должны принадлежать одной доле, в то время как все остальные — другой доле. Во втором примере решения не существует.

Решение

Как и в любой задаче, начнем с разбора примеров. В графе из первого примера две доли можно выделить следующим образом (рис. 19.8).

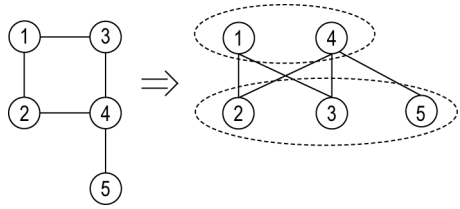

Рис. 19.8. Слева показан входной граф из первого примера, а справа — одно из его возможных разбиений на две доли: с вершинами 1 и 4 в первой доле и остальными вершинами во второй

В графе из второго примера есть цикл нечетной длины, поэтому любые попытки разделить его на две доли будут приводить к тому, что одна из вершин должна будет оказаться ни в одной, ни в другой доле (рис. 19.9).

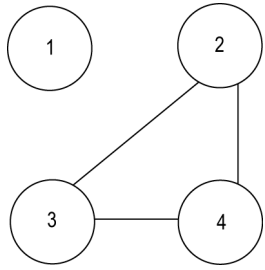

Рис. 19.9. Входной граф из второго примера. Разбить его на две доли невозможно

Каким образом мы строили вручную ответ для первого примера? Мы поместили вершину №1 в одну долю, затем посмотрели на ее соседей — вершины №2 и 3 — и поместили их в другую долю, затем посмотрели на их соседей — вершины №1 и 4 — и поместили вершину №4 снова в первую долю, затем посмотрели на ее соседей — вершины №2, 3 и 5 — и поместили вершину №5 во вторую долю.

По сути, мы использовали правило, вытекающее из определения двудольного графа: если две вершины соединены ребром, то они должны быть в разных долях. Начав со стартовой вершины — которую, не теряя общности, можно поместить в любую из двух долей, — мы последовательно применяли это правило, определяя для каждой из достигнутых вершин номер ее доли.

Всегда ли верно действовать таким образом? Да, поскольку мы базируемся на определении двудольного графа и на обычных логических правилах.

Однако в процессе расстановки долей мы можем натолкнуться на противоречие: одна и та же вершина может требоваться и в первой, и во второй доле. Именно это произойдет во втором примере из условия, когда соседи стартовой вершины будут сначала отнесены ко второй доле, а затем между ними будет обнаружено ребро. Мы знаем, что во втором примере граф не двудольный; на самом деле, из вышесказанного следует, что при обнаружении противоречия граф всегда не двудольный (поскольку противоречие обозначает, что есть две логических цепочки умозаключений «эта вершина должна быть в первой доле» и «эта вершина должна быть во второй доле»).

Осталось только понять, как реализовать этот алгоритм. Фактически нам требуется просто обойти граф, начиная со стартовой вершины и попутно расставляя номера долей — а для этой цели хорошо подходит обход в глубину.

Последний штрих — заметить, что входной граф не обязательно связный, и поэтому при запуске из вершины №1 не все вершины будут посещены. Поэтому требуется произвести серию обходов в глубину, т. е. запустить обход из каждой из не посещенных еще вершин. При этом важно, что отметки посещений сбрасывать не следует, иначе обход будет заново посещать уже посещенные вершины.

Итоговая асимптотика — $O(n + m)$ при использовании обхода в глубину на списках смежности. Несмотря на то что для несвязных графов нам понадобится произвести несколько обходов в глубину, суммарная асимптотика остается такой же, как и у отдельного поиска, ведь мы обработаем каждую вершину только один раз и просмотрим все списки смежности по одному разу.

> ***Жадный алгоритм***
>
> С точки зрения терминологии найденный нами алгоритм — жадный, ведь он постепенно распределяет вершины по долям, никогда не меняя своих решений и руководствуясь каждый раз только локальной информацией о вершинах-соседях. Впрочем, никакой практической роли то, называется ли решение жадным или нет, не имеет.

Реализация

Описанный выше алгоритм представляет собой простую модификацию обхода в глубину: мы дополнительно расставляем вершинам номера доли и проверяем при этом отсутствие конфликтов. Поскольку номер доли у вершины появляется одновременно с ее посещением, можно немного упростить код, избавившись от отдельного массива пометок о посещениях. Удобно рассуждать в терминах «покраски» вершин: изначально все вершины бесцветные, а затем при посещении получают цвет №1 или цвет №2, обозначающий номер доли. Тогда массив этих цветов — единственная дополнительная информация, которую нам надо хранить помимо самого графа.

Python

```
def dfs(vertex):
  global is_bipartite
  for to in adj_list[vertex]:
    if color[to] is None:
      color[to] = 1 - color[vertex]
      dfs(to)
    elif (color[to] ==
          color[vertex]):
      is_bipartite = False

vertex_count, edge_count = map(
    int, input().split())
adj_list = [[] for _ in
            range(vertex_count)]
for _ in range(edge_count):
  vertex_from, vertex_to = map(
      int, input().split())
  vertex_from -= 1
  vertex_to -= 1
  adj_list[vertex_from].append(
      vertex_to)
  adj_list[vertex_to].append(
      vertex_from)

color = [None] * vertex_count
is_bipartite = True
for start in range(vertex_count):
  if color[start] is None:
    color[start] = 0
    dfs(start)
if is_bipartite:
  print(*[i + 1 for i in color])
else:
  print(0)
```

C++

```
#include <iostream>
#include <vector>
using namespace std;

int vertex_count;
vector<vector<int>> adj_list;
vector<char> color;
bool is_bipartite;

void Dfs(int vertex) {
  for (int to : adj_list[vertex]) {
    if (color[to] == -1) {
      color[to] = 1 -
                  color[vertex];
      Dfs(to);
    } else if (color[to] ==
               color[vertex]) {
      is_bipartite = false;
    }
  }
}

int main() {
  int edge_count;
  cin >> vertex_count
      >> edge_count;
  adj_list.resize(vertex_count);
  for (int i = 0;
       i < edge_count;
       ++i) {
    int vertex_from, vertex_to;
    cin >> vertex_from
        >> vertex_to;
    --vertex_from;
    --vertex_to;
    adj_list[
        vertex_from].push_back(
            vertex_to);
    adj_list[vertex_to].push_back(
        vertex_from);
  }

  color.assign(vertex_count, -1);
  is_bipartite = true;
```

```
  for (int start = 0;
       start < vertex_count;
       ++start) {
    if (color[start] == -1) {
      color[start] = 0;
      Dfs(start);
    }
  }

  if (!is_bipartite) {
    cout << 0 << endl;
    return 0;
  }
  for (int i = 0;
       i < vertex_count;
       ++i) {
    cout << color[i] + 1 << ' ';
  }
}
```

19.10. Пример решения задачи. Проверка орграфа на ацикличность

Определение. Граф называется ациклическим, если нельзя, стартовав из какой-либо вершины и пройдя по некоторому количеству ребер без повторов, вернуться в эту же вершину.

> Отметим, что понятие ацикличности имеет смысл в отношении как ориентированных, так и неориентированных графов. Но хотя определение ацикличности и одинаково, есть важное различие между этими двумя случаями: по неориентированному ребру можно ходить в обоих направлениях, однако цикл из одного этого ребра образовать нельзя. Таким образом, неверно считать, что неориентированное ребро — то же самое, что два ориентированных ребра в обоих направлениях.

Простым циклом называется путь в графе, который начинается и заканчивается в одной и той же вершине и в котором все остальные вершины и все ребра различны.

Задача. Определить, является ли заданный ориентированный граф ациклическим, и если нет, то вывести простой цикл, содержащийся в нем.

Входные данные состоят из числа вершин n и ребер m в первой строке и описаний ребер в последующих m строках. Гарантируется, что $1 \le n, m \le 10^5$, концы ребер даны в 1-индексации. Допускаются петли, т. е. ребра, ведущие из вершины в нее же. Вывести требуется длину найденного простого цикла (измеренную в количестве ребер) в первой строке и разделенные пробелом вершины цикла в порядке обхода во второй строке; первая и последняя вершины должны совпадать. Если есть несколько циклов, вывести можно любой из них. Если входной граф ациклический, то вывести нужно просто нуль.

Примеры

Входные данные	Требуемый результат
3 3 1 2 3 1 2 3	3 1 2 3 1
Входные данные	**Требуемый результат**
3 1 1 2	0

Пояснения к примерам

В первом примере весь граф и есть простой цикл: 1 → 2 → 3 → 1. Во втором примере орграф является ациклическим, несмотря на наличие двух различных путей из вершины №1 в вершину №4.

Решение

В этой задаче применение алгоритма обхода в глубину уже менее тривиально.

Помимо хранения отметок посещения будем также хранить для каждой вершины, находится ли она еще в обработке или нет. Для удобства введем такую систему обозначений:

- изначально все вершины графа «белые»;
- когда начинается обработка какой-либо вершины, т. е. обход в глубину приходит в эту вершину и начинает просматривать исходящие из нее ребра, она становится «серой»;
- когда же обработка вершины завершается, т. е. все рекурсивные запуски завершены, вершина становится «черной».

Тогда алгоритм поиска цикла таков: цикл есть тогда и только тогда, когда обход в глубину обнаруживает ребро, ведущее в серую вершину.

Демонстрация работы алгоритма

Продемонстрируем этот алгоритм на примере из условия (рис. 19.10).

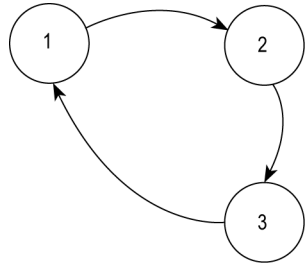

Рис. 19.10. Входной граф из первого примера. Граф содержит цикл

Если мы запустим обход в глубину из вершины №1, то обход сначала пометит ее серой, потом пойдет в вершину №2 и также пометит ее серой, затем из нее пойдет в вершину №3 и пометит ее серой и, наконец, обнаружит ребро, ведущее в серую вершину №1 (рис. 19.11).

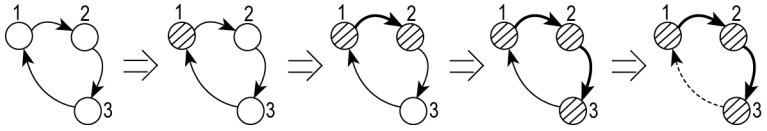

Рис. 19.11. Процесс обхода указанного графа обходом в глубину. Стартуя из вершины 1, алгоритм постепенно посещает все остальные вершины и обнаруживает ребро в вершину 1

Таким образом, алгоритм действительно обнаружил цикл $1 \to 2 \to 3 \to 1$.

Асимптотика алгоритма равна асимптотике алгоритма обхода в глубину и при использовании списков смежности составляет $O(n + m)$.

Доказательство решения

Начнем с доказательства в одну сторону, т. е. что такое ребро в серую вершину — это действительно признак цикла. Докажем это конструктивно, построив простой цикл с этим ребром, которое мы обозначим через $x \to y$, где x — текущая вершина, y — серая. Вспомнив, что поиск в глубину работает рекурсивно и сохраняет у вершины серый цвет только во время рекурсивных запусков из ее соседей, можно понять, что все серые вершины образуют в графе один путь. На конце этого пути в данный момент находится вершина x, а вершина y — либо та же самая вершина, либо лежит ранее в этом пути. Возьмем фрагмент этого пути, начинающийся с вершины y и заканчивающийся вершиной x, и присоединим к нему обнаруженное ребро $x \to y$ — тем самым мы получили требуемый цикл. Графическая иллюстрация этих рассуждений — рис. 19.12.

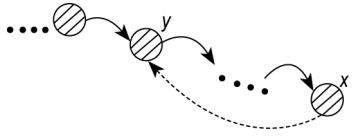

Рис. 19.12. Иллюстрация того факта, что ребро в серую вершину образует цикл из серых вершин

Для доказательства в обратную сторону нам требуется показать, что при наличии в графе цикла ребро в серую вершину будет обязательно обнаружено обходом в глубину. Возьмем любой из простых циклов, содержащихся в графе, и рассмотрим момент времени, когда обход в глубину впервые посетит одну из вершин этого цикла — обозначим эту вершину через s. Обозначим через t ту вершину выбранного нами цикла, которая стоит непосредственно перед s (отметим, что это значит, что в графе есть ребро $t \to s$). Поскольку из любой вершины цикла можно дойти до

любой другой, то и обход в глубину рано или поздно дойдет до вершины t, причем это случится до окончания обработки вершины s. Таким образом, обход в глубину придет в вершину t в тот момент, когда s по-прежнему серая, а, значит, будет обнаружено нужное нам ребро в серую вершину (это будет ребро $t \to s$). Графическая иллюстрация — рис. 19.13.

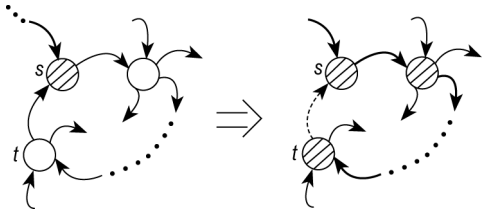

Рис. 19.13. При наличии цикла в графе алгоритм обязательно обнаружит ребро в серую вершину

Это доказывает, что наш алгоритм действительно обнаружит простой цикл, что завершает доказательство.

Поиск конкретного цикла или всех циклов

Заметим, что алгоритм и его доказательство устроены так, что они обнаруживают **какой-либо** из циклов графа, а не обязательно цикл с какими-либо определенными свойствами. Задачи на поиск определенного цикла (например, цикла с конкретной вершиной или самого короткого цикла, или самого длинного цикла), а также на подсчет и поиск всех циклов — существенно сложнее и, как правило, решаются совершенно другими методами.

Реализация

Один из моментов, про которые надо не забыть при реализации, — это то, что недостаточно запустить один обход в глубину из вершины №1: надо запустить его из каждой не посещенной еще вершины (при этом обнулять пометки не следует, так как иначе асимптотика ухудшится до квадратичной).

Другая тонкость — для построения найденного цикла мы храним для каждой вершины ее «родителя» (*англ.* parent) в обходе в глубину, т. е. номер вершины, из которой мы впервые попали в заданную вершину. Тогда построение цикла по обнаруженному серому ребру заключается в движении по этим указателям от одного конца ребра до тех пор, пока не будет встречен другой конец.

Python

```
def dfs(v):
  global cycle_begin, cycle_end
  color[v] = 1
  for to in g[v]:
    if color[to] == 0:
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
```

```
      parent[to] = v
      dfs(to)
    elif color[to] == 1:
      cycle_begin = to
      cycle_end = v
  color[v] = 2

n, m = map(int, input().split())
g = [[] for _ in range(n)]
for _ in range(m):
  v1, v2 = map(int,
               input().split())
  v1 -= 1
  v2 -= 1
  g[v1].append(v2)

color = [0] * n
parent = [None] * n
cycle_begin = cycle_end = None
for start in range(n):
  if color[start] == 0:
    dfs(start)

cycle = []
if cycle_begin is None:
  print(0)
else:
  cycle.append(cycle_begin)
  v = cycle_end
  while v != cycle_begin:
    cycle.append(v)
    v = parent[v]
  cycle.append(cycle_begin)
  cycle = cycle[::-1]
  print(len(cycle) - 1)
  print(*(v + 1 for v in cycle))
```

```
int n;
vector<vector<int>> g;
vector<int> parent;
vector<char> color;
int cycle_begin, cycle_end;

void dfs(int v) {
  color[v] = 1;
  for (int to : g[v]) {
    if (color[to] == 0) {
      parent[to] = v;
      dfs(to);
    } else if (color[to] == 1) {
      cycle_begin = to;
      cycle_end = v;
    }
  }
  color[v] = 2;
}

int main() {
  int m;
  cin >> n >> m;
  g.resize(n);
  for (int i = 0; i < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1].push_back(v2);
  }

  color.assign(n, 0);
  parent.assign(n, -1);
  cycle_begin = -1;
  for (int start = 0; start < n;
       ++start) {
    if (!color[start])
      dfs(start);
  }

  if (cycle_begin == -1) {
    cout << 0 << endl;
    return 0;
  }
  vector<int> cycle;
  cycle.push_back(cycle_begin);
```

```
  for (int v = cycle_end;
       v != cycle_begin;
       v = parent[v]) {
    cycle.push_back(v);
  }
  cycle.push_back(cycle_begin);
  reverse(cycle.begin(),
          cycle.end());
  cout << cycle.size() - 1 << endl;
  for (int v : cycle)
    cout << v + 1 << ' ';
}
```

Реализация для неориентированных графов

Логика алгоритма справедлива и для неориентированных графов. Однако при реализации возникает единственная тонкость: при обычном построении матрицы или списков смежности каждое неориентированное ребро хранится в виде двух ориентированных. Это означает, что приведенная выше реализация неправильно интерпретирует первое же встреченное ребро как цикл (как если бы можно было создать цикл из одного ребра, пройдя по нему в одном направлении и затем вернувшись обратно). Однако это легко поправить с помощью одной дополнительной проверки: игнорировать переход в серую вершину, если она равна `parent[v]`. Последняя тонкость: если в графе могут быть кратные (т. е. повторяющиеся) ребра, то надо либо хранить в массиве `parent` идентификаторы ребер, либо просто обработать случай кратных ребер заранее и вывести ответ без запуска обхода в глубину.

19.11. Пример решения задачи. Топологическая сортировка

Определение. Топологической сортировкой ациклического ориентированного графа называется упорядочение вершин в такой последовательности, что для каждого ребра $a \to b$ вершина a идет в последовательности ранее, чем вершина b.

Если граф содержит циклы, то топологической сортировки для него не существует.

Задача. При планировании работ для очередного квартала руководство компании столкнулось со следующей проблемой. Имеется n задач, которые нужно выполнить. Между некоторыми из задач имеются зависимости: одну задачу нельзя начинать до того, как завершена какая-либо другая задача. Помогите руководству найти такой порядок исполнения задач, который будет удовлетворять всем зависимостям, или сообщите, что такого порядка не существует.

В первой строке входных данных указано число задач n и число зависимостей m. В последующих m строках описаны сами зависимости в виде пар чисел a_i и b_i, обозначающих, что задача номер a_i должна быть выполнена до задачи номер b_i ($1 \le a_i, b_i \le n$, $a_i \ne b_i$). Выведите искомый порядок в виде перестановки чисел от 1 до n или же единственное число -1, в случае если такого порядка не существует. Если ответов несколько, вывести можно любой.

Примеры

Входные данные	**Требуемый результат**
5 4 2 1 3 1 3 4 5 4	2 3 5 1 4
Входные данные	**Требуемый результат**
3 3 1 2 2 3 3 1	-1

Решение

Очевидно, что эта задача — на поиск топологической сортировки в заданном орграфе. И понятно, что ответ может существовать только тогда, когда граф ациклический, ведь иначе мы имеем набор задач, которые зависят друг от друга по кругу. Проверять граф на ацикличность мы уже умеем — с помощью обхода в глубину. Всегда ли существует ответ для ациклического графа?

Мы не будем пытаться доказать этот факт сам по себе, а будем конструктивно искать алгоритм нахождения топологической сортировки. Если посмотреть на зависимости между задачами как на ребра, то вершина в искомом порядке должна следовать до всех своих соседей, а также до соседей этих соседей и т. д. И здесь отчетливо возникает аналогия с обходом в глубину, который тоже сначала посещает какую-либо вершину, затем обходит каждого из ее соседей, каждого из соседей их соседей и т. д.

Осталось понять, как же правильно строить требуемый топологический порядок.

Неверным было бы просто взять порядок обхода вершин в качестве ответа. Например, уже на графе из двух вершин с одним ребром $2 \to 1$ обход в глубину, запускаемый из вершин в порядке возрастания их номеров, сначала посетит вершину №1 (из которой не выходит никаких ребер) и затем вершину №2, в то время как верная топологическая сортировка в этом случае — (2, 1). Главная проблема здесь — в уже посещенных вершинах.

Можно понять, что правильным и в то же время простым для реализации является следующий алгоритм: добавлять вершину в начало списка в момент выхода обхода в глубину из этой вершины. В самом деле, каждый запуск из какой-либо вершины проходит по всем ее соседям, либо рекурсивно запуская себя из каждого соседа, либо пропуская тех соседей, которые уже были посещены ранее. Так или иначе, к моменту окончания работы этого запуска все вершины, которые должны следовать

в искомой сортировке после текущей, уже лежат в списке, поэтому правильно добавить текущую вершину в начало.

Асимптотика этого алгоритма составит $O(n + m)$, поскольку он представляет из себя одну серию обходов в глубину по всему графу.

Реализация

Небольшой трюк: ради простоты и краткости реализации мы будем добавлять посещаемые вершины не в начало, а в конец списка; тогда ответ на задачу будет равен инверсии этого списка.

Python

```
def dfs(v):
  global has_cycle
  color[v] = 1
  for to in g[v]:
    if color[to] == 0:
      dfs(to)
    elif color[to] == 1:
      has_cycle = True
  color[v] = 2
  topo_order.append(v)

n, m = map(int, input().split())
g = [[] for _ in range(n)]
for _ in range(m):
  v1, v2 = map(int,
               input().split())
  v1 -= 1
  v2 -= 1
  g[v1].append(v2)

color = [0] * n
has_cycle = False
topo_order = []
for start in range(n):
  if color[start] == 0:
    dfs(start)
topo_order = topo_order[::-1]

if has_cycle:
  print(-1)
else:
  print(*[v + 1 for v in
          topo_order])
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int n;
vector<vector<int>> g;
vector<char> color;
bool has_cycle;
vector<int> topo_order;

void dfs(int v) {
  color[v] = 1;
  for (int to : g[v]) {
    if (color[to] == 0)
      dfs(to);
    else if (color[to] == 1)
      has_cycle = true;
  }
  color[v] = 2;
  topo_order.push_back(v);
}

int main() {
  int m;
  cin >> n >> m;
  g.resize(n);
  for (int i = 0; i < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1].push_back(v2);
  }
```

```
  color.assign(n, 0);
  has_cycle = false;
  for (int start = 0; start < n;
       ++start) {
    if (!color[start])
      dfs(start);
  }

  if (has_cycle) {
    cout << -1 << endl;
    return 0;
  }
  reverse(topo_order.begin(),
          topo_order.end());
  for (int v : topo_order)
    cout << v + 1 << ' ';
}
```

19.12. Пример решения задачи. Диаметр дерева

Задача. Дано дерево, т. е. связный неориентированный граф без циклов. Требуется найти его диаметр, т. е. расстояние между двумя наиболее удаленными друг от друга вершинами.

Входные данные содержат число вершин n в первой строке ($1 \le n \le 5 \cdot 10^5$) и описание ребер в виде $n - 1$ строк, содержащих по два числа каждая: номера вершин-концов ребра. Вершины нумеруются начиная с единицы.

Вывести требуется три числа — искомый диаметр и номера двух вершин, расстояние между которыми равно диаметру. При наличии нескольких решений вывести можно любое.

Примеры

Входные данные	**Требуемый результат**
6 1 2 1 3 1 6 2 4 3 5	4 4 5
Входные данные	**Требуемый результат**
1	0 1 1

Решение

Есть несколько способов решить эту задачу. Приведем здесь один из них — по-своему изящный алгоритм. Он состоит из двух шагов:

1. Запустить обход в глубину из произвольной вершины, например №1, и найти самую удаленную от нее вершину — обозначим ее через u.
2. Запустить обход в глубину из u и найти самую удаленную от нее вершину — обозначим ее через w. Ответом будет расстояние между u и w.

Мы не будем приводить здесь математически строгого доказательства, однако в качестве неформального аргумента можно действовать от противного: предположим, что выбранная вершина u не является концом никакого из диаметров, тогда рассмотрим любой из диаметров (x, y) и покажем, что либо x, либо y можно заменить на u и получить пару с таким же или бóльшим расстоянием (для этого понадобится рассмотреть несколько возможных конфигураций их взаимного расположения).

> ***Диаметр произвольного графа***
>
> Отметим, что приведенный алгоритм работает **только для деревьев**. Для произвольных графов он, вообще говоря, неверен (что вызвано тем, что между парами вершин в произвольном графе может быть более одного пути).

Асимптотическая сложность этого решения равна сложности обхода в глубину, что в нашем случае дерева дает $O(n)$.

Глава 20.
Графы. Обход в ширину

Алгоритм обхода в ширину (*англ.* breadth-first search, или, сокращенно, bfs) действует таким образом, что он постепенно удаляется от стартовой вершины, двигаясь от нее по всевозможным направлениям.

Интуитивно его можно представить как процесс распространения огня от стартовой точки: сначала огонь перекидывается на все соседствующие со стартовой вершиной, затем на их соседей и т. д. — до тех пор пока огонь не достигнет всех возможных вершин. Как можно заметить, такой обход кардинально отличается от обхода в глубину, который может быстро удалиться от стартовой вершины по первому же ребру из стартовой вершины.

Ключевое свойство обхода в ширину — то, что он определяет **расстояния** от стартовой вершины до всех остальных вершин. Расстояние здесь — это наименьшее количество ребер, которое надо преодолеть, чтобы из одной вершины дойти до другой (т. е. вес каждого ребра равен единице). Возвращаясь к аналогии с пожаром, можно понять, за счет чего происходит это нахождение расстояний:

- на первом шаге огонь переходит на соседей стартовой вершины, т. е. на вершины, расстояние до которых равно единице;
- на втором шаге огонь достигает соседей вершин, достигнутых на первом шаге, — т. е. на вершины, расстояние до которых равно двум;
- и т. д.: на *i*-м шаге огонь достигает вершины, расстояние до которых от стартовой вершины равно *i*.

Преобразуем это интуитивное описание обхода в ширину в более конкретный алгоритм, пригодный к реализации.

20.1. Алгоритм обхода в ширину

Введем **очередь** обработки вершин. Изначально очередь содержит только стартовую вершину. Обход в ширину состоит из набора итераций; на каждой итерации будем извлекать из очереди одну вершину и совершать из нее переходы во всех не посещенных еще соседей. Обход завершается, когда очередь пуста.

Последняя деталь: кратчайшие расстояния можно вычислить следующим образом. Изначально пометим, что расстояние до стартовой вершины равно нулю. А когда обход в ширину совершает переход из одной вершины в другую (до сих пор еще не посещенную), то расстояние до новой вершины будем присваивать единице плюс

расстоянию до старой вершины. Иными словами, если u — текущая вершина, в которой стоит обход в ширину, и есть ребро $u \to v$, где v — еще не посещенная вершина, и $d[]$ — массив расстояний, то:

$$d[v] = d[u] + 1.$$

20.2. Свойства обхода в ширину

- Обход в ширину посетит все вершины, достижимые из стартовой.
- Вершины обходятся в порядке роста (если быть точным, неуменьшения) расстояний от стартовой вершины. Расстояние в данном случае измеряется количеством ребер (этот случай называется невзвешенным графом, в противоположность взвешенным графам, в которых каждое ребро имеет длину или вес).
- Каждая достижимая вершина будет обработана ровно один раз, равно как и каждое достижимое ребро.
- Алгоритм одинаково применим как к ориентированным, так и к неориентированным графам.
- Наличие петель и кратных ребер также не влияет на корректность работы алгоритма.

20.2.1. Демонстрация обхода в ширину

Рассмотрим следующий граф (рис. 20.1).

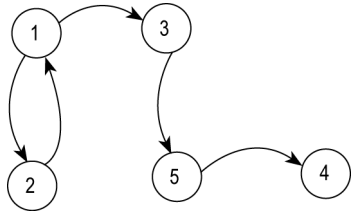

Рис. 20.1. Пример ориентированного графа с 5 вершинами

Проследим работу алгоритма обхода в ширину при запуске из вершины №1 (рис. 20.2).

- Помещаем вершину №1 в очередь. Помечаем ее как уже посещенную, и $d[1] = 0$.
- Достаем очередную вершину из очереди — это вершина №1. Просматриваем исходящие из нее ребра. Одно ребро ведет в вершину №2, которая еще не посещена, поэтому помещаем вершину №2 в очередь, помечаем ее как посещенную и выставляем $d[2] = d[1] + 1 = 1$. Другое ребро ведет в вершину №3, которая тоже добавляется в очередь, помечается посещенной, и $d[3] = 1$.
- Достаем очередную вершину из очереди — это вершина №2. Просматриваем исходящие из нее ребра. Единственное ребро ведет в вершину №1, которая уже посещена, поэтому пропускаем ее.

- Достаем очередную вершину из очереди — это вершина №3. Из нее есть исходящее ребро в еще не посещенную вершину №5, поэтому помещаем №5 в очередь, помечаем ее как посещенную, и $d[5] = d[3] + 1 = 2$.
- Достаем очередную вершину из очереди — это вершина №5. Из нее есть исходящее ребро в еще не посещенную вершину №4, поэтому помещаем №4 в очередь, помечаем ее как посещенную, и $d[4] = d[5] + 1 = 3$.
- Достаем очередную вершину из очереди — это вершина №4. Исходящих из нее ребер нет.
- Очередь пуста — завершаем алгоритм.

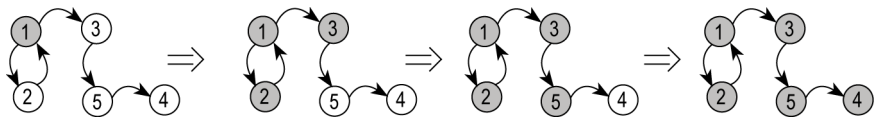

Рис. 20.2. Работа алгоритма обхода в ширину на указанном графе

Почему обход называется обходом «в ширину»

На этом примере видно, что, в отличие от обхода в глубину, алгоритм распространяется с одинаковой скоростью во все стороны, словно волны, расходящиеся от упавшего в воду камня.

Влияние порядка просмотра ребер

Результаты обхода в ширину не зависят от порядка просмотра ребер, т. е. от того, как именно хранятся вершины-соседи у каждой вершины графа. Хотя порядок помещения вершин в очередь может быть различным, влияния на итоговые расстояния это не окажет. Это одно из существенных отличий обхода в ширину от обхода в глубину, пути следования которого по графу могут кардинально изменяться даже от перестановки местами одной пары соседей.

20.3. Реализация обхода в ширину

В отличие от обхода в глубину, в обходе в ширину нет рекурсии. Зато вместо нее появляется очередь, которая нужна для хранения предназначенных к обработке вершин.

Еще одно отличие — в том, что обход в ширину подсчитывает длины кратчайших путей из стартовой вершины до всех остальных вершин. С точки зрения реализации это означает, что хранить отдельный массив с пометками посещенных вершин не требуется: достаточно заранее инициализировать массив расстояний каким-либо особым значением (например, «−1» в C++ или «None» в Python).

20.3.1. Обход в ширину на матрице смежности

Приведем реализацию обхода в ширину, когда граф задан своей матрицей смежности, а стартовая вершина — вершина №1 (что соответствует индексу «0» в про-

граммном коде). Реализацию создания матрицы смежности мы не приводим здесь повторно. В целях демонстрации программа выводит посчитанный массив расстояний.

Python

```
from collections import *

# ... чтение vertex_count,
# adj_matr ...

dist = [None] * vertex_count
start = 0
queue = deque()
queue.append(start)
dist[start] = 0
while queue:
  vertex = queue.popleft()
  for to in range(vertex_count):
    if (adj_matr[vertex][to] and
        dist[to] is None):
      queue.append(to)
      dist[to] = dist[vertex] + 1

print(*dist)
```

C++

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
int main() {
  // ... чтение vertex_count,
  // adj_matr ...

  int start = 0;
  vector<int> dist(
      vertex_count, -1);
  queue<int> q;
  q.push(start);
  dist[start] = 0;
  while (!q.empty()) {
    int vertex = q.front();
    q.pop();
    for (int to = 0;
         to < vertex_count;
         ++to) {
      if (adj_matr[vertex][to] &&
          dist[to] == -1) {
        dist[to] = dist[vertex] +
                   1;
        q.push(to);
      }
    }
  }

  for (int i = 0;
       i < vertex_count;
       ++i) {
    cout << dist[i] << ' ';
  }
}
```

Примечание о кратких наименованиях переменных

В приведенных выше реализациях мы использовали информативные, а значит, и длинные, имена переменных: `vertex_count`, `adj_matr`, `vertex` и т. п. Это было сделано с целью повышения понятности листингов кода в книге. В реальном спортивном программировании более практичными были бы краткие имена, такие, как `n`, `g`, `v` соответственно.

Время работы обхода в ширину при использовании матрицы смежности составляет $O(n^2)$, поскольку в худшем случае мы обрабатываем все n вершин, а для каждой вершины требуется пройти по всей строке матрицы смежности, длина которой равна n.

20.3.2. Обход в ширину на списках смежности

Ниже приведена реализация обхода в ширину в случае, когда граф задан списками смежности.

Python

```
from collections import *

# ... чтение vertex_count,
# adj_list ...

start = 0
dist = [None] * vertex_count
queue = deque()
queue.append(start)
dist[start] = 0
while queue:
  vertex = queue.popleft()
  for to in adj_list[vertex]:
    if dist[to] is None:
      queue.append(to)
      dist[to] = dist[vertex] + 1

print(*dist)
```

C++

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
int main() {
  // ... чтение n, g ...

  int start = 0;
  vector<int> dist(
      vertex_count, -1);
  queue<int> q;
  q.push(start);
  dist[start] = 0;
  while (!q.empty()) {
    int vertex = q.front();
    q.pop();
    for (int to :
         adj_list[vertex]) {
      if (dist[to] == -1) {
        dist[to] = dist[vertex] +
                   1;
        q.push(to);
      }
    }
  }

  for (int i = 0;
       i < vertex_count;
       ++i) {
    cout << dist[i] << ' ';
  }
}
```

Время работы обхода в ширину при использовании списков смежности составляет $O(n + m)$, где n — число вершин, m — число ребер, поскольку в худшем случае обход в ширину обработает каждую вершину и пройдет по всем спискам смежности.

Таким образом, как и в случае с обходом в глубину, использование списков смежности хорошо работает для любых графов (а если граф является разреженным, то списки смежности будут работать существенно быстрее, чем матрицы смежности).

20.4. Пример решения задачи. Кластер компьютеров

Задача. Серверный кластер состоит из n компьютеров, некоторые из них напрямую связаны сетевым кабелем. Говорят, что два компьютера соединены быстрой связью, если можно передать данные с одного на другой, используя не более чем k промежуточных компьютеров. Например, при $k = 0$ только связанные напрямую кабелем компьютеры считаются соединенными быстрой связью. Задача — посчитать, со сколькими компьютерами соединен быстрой связью компьютер №1.

Входные данные состоят из числа n компьютеров в кластере, параметра k и числа m прямых сетевых соединений ($1 \le n \le 10^5$, $0 \le k < n - 1$, $0 \le m \le 10^5$) в первой строке. Последующие m строк содержат по паре чисел a_i и b_i, обозначающих, что компьютеры a_i и b_i связаны прямым сетевым кабелем ($1 \le a_i, b_i \le n$, $a_i \ne b_i$). Вывести требуется одно число — искомое количество компьютеров.

Примеры

Входные данные	**Требуемый результат**
5 1 4 1 2 2 5 3 1 5 4	3
Входные данные	**Требуемый результат**
4 2 2 1 4 2 3	1

Пояснения к примерам

В первом примере компьютер №1 соединен быстрой связью с компьютерами №2, 3, 5; компьютер №4 не подходит, поскольку для передачи данных с компьютера №1 требуется два других компьютера: №2 и 5. Во втором примере компьютер №1 соединен быстрой связью только с компьютером №4.

Решение

Переформулировав условие, получаем, что компьютеры i и j соединены быстрой связью, если в соответствующем неориентированном графе расстояние между ними (измеренное в количестве ребер) не превосходит $k + 1$. Таким образом, решить

задачу можно, просто выполнив обход в ширину из вершины №1 и затем посчитав количество подходящих элементов в массиве расстояний.

Асимптотика решения составит $O(n + m)$ при использовании обхода в ширину на списках смежности.

Реализация

Реализация здесь практически повторяет описанный ранее алгоритм обхода в ширину.

Однако добавим в нее небольшую **оптимизацию**: поскольку нас не интересуют вершины, расстояния до которых больше $k + 1$, то мы будем останавливать обход в ширину, как только дойдем до обработки вершины с расстоянием $k + 1$ в первый раз. В самом деле, расстояния, которые вычисляет обход в ширину, только растут со временем; если алгоритм начал обработку вершины с расстоянием x, то все впоследствии добавленные в очередь вершины будут иметь расстояние $x + 1$, позже $x + 2$ и т. д.

> Эта оптимизация — не асимптотическая, т. е. она никак не улучшает оценку времени работы решения, которая была $O(n + m)$. В текущей задаче с ее ограничениями эта оптимизация не является необходимой. Тем не менее полезно знать об этом способе ускорения алгоритма, поскольку могут быть случаи, когда ранняя остановка обхода поможет значительно сэкономить время работы, особенно если известно, что достижимых до этого порогового расстояния вершин сравнительно немного.

Python

```
from collections import *

n, k, m = map(
    int, input().split())
g = [[] for _ in range(n)]
for _ in range(m):
  v1, v2 = map(
      int, input().split())
  v1 -= 1
  v2 -= 1
  g[v1].append(v2)
  g[v2].append(v1)

dist = [None] * n
q = deque()
q.append(0)
dist[0] = 0
while q:
  v = q.popleft()
  if dist[v] >= k + 1:
```

C++

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
int main() {
  int n, k, m;
  cin >> n >> k >> m;
  vector<vector<int>> g(n);
  for (int i = 0; i < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1].push_back(v2);
    g[v2].push_back(v1);
  }

  vector<int> dist(n, -1);
  queue<int> q;
  q.push(0);
```

```
    break
  for to in g[v]:
    if dist[to] is None:
      q.append(to)
      dist[to] = dist[v] + 1
ans = n - 1 - dist.count(None)
print(ans)
```

```
  dist[0] = 0;
  while (!q.empty()) {
    int v = q.front();
    q.pop();
    if (dist[v] >= k + 1)
      break;
    for (int to : g[v]) {
      if (dist[to] == -1) {
        dist[to] = dist[v] + 1;
        q.push(to);
      }
    }
  }
  int reached = 0;
  for (int i = 0; i < n; ++i) {
    if (dist[i] != -1)
      ++reached;
  }
  cout << reached - 1 << endl;
}
```

20.5. Пример решения задачи. Робот в лабиринте

Задача. Помогите роботу найти кратчайший выход из лабиринта. Описание лабиринта представляет собой карту $n \times m$ клеток, в которой каждая клетка либо пуста, либо непроходима. Поскольку робот ориентируется по установленной на нем камере и она только одна, то из соображений безопасности он может двигаться только вперед, в направлении обзора этой камеры; однако при необходимости робот также может изменить свое направление вправо или влево. За один шаг робот может продвинуться на соседнюю клетку в текущем направлении — при условии, что она пуста — либо повернуться на 90 градусов по или против часовой стрелки. Изначально робот находится в заданной клетке лабиринта и ориентирован наверх (т. е. в сторону первой строки карты). Робот выходит из лабиринта, когда он достигает границ карты, т. е. оказывается в первом или последнем столбце, либо же в первой или последней строке.

Входные данные состоят из размеров карты n и m в первой строке ($1 \le n, m \le 100$) и описания карты в последующих n строках по m символов в каждой. Каждый символ карты — это либо «.» (пустая клетка), либо «#» (непроходимая клетка), либо «S» (начальное положение робота). Гарантируется, что лабиринт содержит ровно одну клетку «S». Вывести требуется одно число — наименьшее число шагов, необходимое роботу для выхода из лабиринта, либо «−1», если решения не существует.

Примеры

Входные данные	**Требуемый результат**
4 5 ##### #...# #.#S# #.###	7
Входные данные	**Требуемый результат**
3 3 ### #S# ##.	-1

Пояснения к примерам

В первом примере следует сделать один ход роботом наверх, затем повернуться налево, затем сделать два хода, затем повернуть налево и сделать еще два хода — таким образом, за семь шагов достигнув нижней границы карты. Во втором примере робот окружен со всех сторон стенкой, поэтому решения не существует.

Решение

В главе об обходе в глубину мы уже видели, как двумерные задачи подобного рода сводятся к графовым: клетки поля становятся вершинами, возможные переходы из одной клетки в другую — ребрами (то, что в прошлой главе ходы совершались по правилам шахматного коня, а здесь — только в четыре соседние клетки, сути не меняет). И поскольку в текущей задаче спрашивают о кратчайшем пути, то сразу понятно, что граф надо пытаться обработать обходом в ширину.

Единственное, что необычно в текущей задаче и что не позволяет решить ее точно тем же способом, что и ту старую задачу — это наличие у робота ориентации, причем такой, что на смену направления тратится отдельный ход.

Учесть эту ориентацию можно следующим образом. Пусть вершинами графа будут не просто клетки поля, а всевозможные состояния «клетка+направление». Поскольку, по условию, направлений всего четыре, то вершин в графе будет $4nm$. А ребра, исходящие из каждой вершины, будут соответствовать всем возможным шагам из этого состояния:

- одно ребро для перехода в соседнюю по текущему направлению (при условии, что эта клетка пуста);
- второе ребро для поворота налево с сохранением координат;
- третье ребро для поворота направо с сохранением координат.

Итого мы получили граф с $4nm$ вершинами и не более чем $12nm$ ребрами. Обход в ширину в таком графе позволит найти длину кратчайшего пути от указанного со-

стояния до всех остальных — в нашем случае до всех состояний, содержащих пограничные клетки поля.

Асимптотика решения составит, при использовании списков смежности, $O(nm)$.

Реализация

Несколько реализационных соображений:

- как и в задаче «Конная прогулка», мы не будем явно строить и хранить граф, а вместо этого будем использовать координаты (номер строки и номер столбца) и генерировать всевозможные ребра-переходы при обработке текущей вершины (без хранения этих ребер где-либо). В каком-то смысле мы будем совершать обход по «виртуальному» графу;
- для хранения вершин в очереди и для хранения массива расстояний будем использовать представление вершин-состояний в виде троек «строка, столбец, направление»;
- все направления занумеруем числами от 0 до 3 в порядке по или против часовой стрелки, чтобы повороты превращались в простое прибавление или вычитание единицы по модулю 4 (что при реализации удобнее всего закодировать как прибавление единицы или тройки по модулю четырех). Для каждого направления зададим в виде констант его «дельты» DX и DY, т. е. то, как меняются координаты при совершении одного шага;
- обход будем немедленно останавливать, когда он приходит в пограничную клетку поля, ведь, согласно свойствам обхода в ширину, никакого более короткого пути уже не будет найдено ни в ту же, ни в какую-либо другую пограничную клетку. Если же обход завершился «естественным» путем, т. е. очередь обработки стала пустой, то решения не существует.

Python

```
from collections import *
import sys

DX = [-1,0,1,0]
DY = [0,1,0,-1]

def handle_edge(x, y, dir,
                nx, ny, ndir):
  if dist[nx][ny][ndir] is None:
    dist[nx][ny][ndir] = \
        dist[x][y][dir] + 1
    q.append((nx, ny, ndir))

n, m = map(int, input().split())
field = [input() for _ in range(n)]
```

C++

```
#include <cstring>
#include <iostream>
#include <queue>
#include <tuple>
#include <vector>
using namespace std;

const int MAXN = 100;
const int DX[] = {-1,0,1,0};
const int DY[] = {0,1,0,-1};

int n, m;
string field[MAXN];
int dist[MAXN][MAXN][4];
queue<tuple<int, int, int>> q;
```

```
q = deque()
dist = [
    [[None] * 4 for _ in range(m)]
    for _ in range(n)]
for x in range(n):
  for y in range(m):
    if field[x][y] == 'S':
      dist[x][y][0] = 0
      q.append((x, y, 0))
while q:
  x, y, dir = q.popleft()
  if (x == 0 or x == n - 1 or
      y == 0 or y == m - 1):
    print(dist[x][y][dir])
    sys.exit(0)
  handle_edge(x, y, dir,
              x, y, (dir + 1) % 4)
  handle_edge(x, y, dir,
              x, y, (dir + 3) % 4)
  nx = x + DX[dir]
  ny = y + DY[dir]
  if (0 <= nx < n and
      0 <= ny < m and
      field[nx][ny] == '.'):
    handle_edge(x, y, dir,
                nx, ny, dir)
print(-1)
```

```
void HandleEdge(
    int x, int y, int dir,
    int nx, int ny, int ndir) {
  if (dist[nx][ny][ndir] == -1) {
    dist[nx][ny][ndir] =
        dist[x][y][dir] + 1;
    q.push(make_tuple(
        nx, ny, ndir));
  }
}

int main() {
  int n, m;
  cin >> n >> m;
  for (int i = 0; i < n; ++i)
    cin >> field[i];

  memset(dist, -1, sizeof dist);
  for (int x = 0; x < n; ++x) {
    for (int y = 0; y < m; ++y) {
      if (field[x][y] == 'S') {
        dist[x][y][0] = 0;
        q.push(make_tuple(
            x, y, 0));
      }
    }
  }
  while (!q.empty()) {
    int x, y, dir;
    tie(x, y, dir) = q.front();
    q.pop();
    if (x == 0 || x == n - 1 ||
        y == 0 || y == m - 1) {
      cout << dist[x][y][dir]
           << endl;
      return 0;
    }
    HandleEdge(
        x, y, dir,
        x, y, (dir + 1) % 4);
    HandleEdge(
        x, y, dir,
        x, y, (dir + 3) % 4);
    int nx = x + DX[dir];
    int ny = y + DY[dir];
    if (0 <= nx && nx < n &&
```

```
        0 <= ny && ny < m &&
        field[nx][ny] == '.') {
      HandleEdge(x, y, dir,
                 nx, ny, dir);
    }
  }
  cout << -1 << endl;
}
```

20.6. Пример решения задачи. Наводнение

Задача. Ученые обнаружили, что уровень воды в определенном водоеме равномерно и непрерывно растет. Помогите им определить, через какое время вся суша скроется под водой. Вам дано схематическое описание участка водоема в виде прямоугольной карты $n \times m$ клеток, где каждая клетка — либо суша (обозначена символом «L»), либо водоем (обозначен символом «W»). За единицу времени вода продвигается вглубь суши на одну клетку; формально клетка суши превращается за единицу времени в клетку водоема, если хотя бы одна из ее восьми соседних клеток является клеткой водоема. В этой задаче время дискретное, т. е. в момент времени $t + 1$ покрываются водой те клетки суши, которые соседствовали с водой в момент времени t, и никаких изменений в промежутке между этими временными точками нет.

Входные данные содержат в первой строке числа n и m. Далее следуют n строк по m символов каждая, описывающих поле. Гарантируется, что описание поля состоит только из символов «L» и «W» и что все пограничные клетки карты — это «W». Размеры поля не превосходят 500. Вывести требуется одно натуральное число — через сколько единиц времени вся суша скроется под водой.

Примеры

Входные данные	**Требуемый результат**
5 7 WWWWWWW WLLLLLW WLLLLLW WWLLLWW WWWWWWW	2
Входные данные	**Требуемый результат**
1 3 WWW	0

Пояснения к примерам

В первом примере за одну единицу времени под водой скроется вся суша, кроме единственной клетки — в третьей строке и четвертом столбце. Во втором примере нет ни одной клетки суши, поэтому ответ равен нулю.

Решение

Переформулировав задачу в терминах графов, где клетки поля — вершины, а ребра связывают соседствующие клетки, мы получаем: требуется найти вершину суши, наиболее удаленную от всех вершин водоема.

Можно было бы решать эту задачу за квадратичное время, запустив по обходу в ширину из каждой вершины суши и тем самым найдя для каждой вершины суши ближайшую вершину моря. Однако при заданных ограничениях решение с асимптотикой $O(n^2m^2)$ — слишком медленное.

Вместо этого постараемся обойтись одним обходом в ширину, а именно, модифицировав его следующим образом. Будем запускать обход не из какой-либо одной стартовой вершины, а сразу из всех вершин водоема. С технической точки зрения это означает, что мы выставим нулевое расстояние сразу у нескольких вершин и все их поместим в очередь. Все остальные части обхода в ширину оставим стандартными. Утверждается, что по окончании работы такого обхода в ширину мы найдем расстояния от каждой клетки суши до ближайшей клетки водоема.

Ответом будет максимум из вычисленных таким обходом в ширину расстояний.

Доказательство

Почему этот модифицированный обход в ширину — с запуском из нескольких стартовых вершин — верен? Вспомним нашу интуитивную аналогию обхода в ширину с процессом распространения огня; теперь мы «поджигаем» не одну вершину графа, а сразу несколько, и огонь начинает расходиться от каждой из этих вершин параллельно. Когда он впервые достигает какую-либо вершину, можно интуитивно понять, что это волна огня, запущенная из ближайшей из стартовых вершин.

Корректность алгоритма можно доказать и более формально. Для этого заметим, что наш модифицированный обход эквивалентен стандартному обходу в ширину на графе с дополнительно введенной стартовой вершиной, из которой есть по одному ребру в каждую вершину водоема. Действительно, при стандартном обходе в ширину этого модифицированного графа после обработки стартовой вершины все вершины водоема будут помещены в очередь с одинаковым расстоянием. Единственное отличие — все расстояния в таком модифицированном графе будут на единицу больше, однако корректности это не меняет.

Ограничения модифицированного алгоритма

Обход в ширину сразу из нескольких стартовых вершин корректен только в том случае, когда расстояние у всех этих стартовых вершин выставлено в ноль. Неверно пытаться обобщить его на случаи, когда, скажем, у одной стартовой вершины расстояние установлено в ноль, а у другой — в единицу.

Глава 21.
Решето Эратосфена

Решето Эратосфена — это алгоритм, позволяющий эффективно найти все простые числа в заданном отрезке [1; n].

Историческая справка

Этот алгоритм — наряду с алгоритмом Евклида один из наиболее древних алгоритмов, активно используемых и в настоящее время. Его изобретение приписывается древнегреческому ученому Эратосфену Киренскому, жившему в III–II вв. до н. э. Впрочем, об этом сохранились лишь косвенные сведения: самое раннее из известных ныне упоминаний содержится в труде «Введение в арифметику» Никомаха Герасского, написанном во II в. н. э. Предполагают, что название «решето» соответствует тому, как этот алгоритм применялся изначально: числа выписывались на дощечке, покрытой воском, и делались проколы в местах, где были записаны составные числа.

21.1. Алгоритм решета Эратосфена

- Изначально выпишем последовательность всех натуральных чисел от 2 до n (число 1 пропущено, поскольку оно не является ни простым, ни составным).
- Затем вычеркнем из последовательности все числа, кратные 2, кроме самого числа 2.
- Затем вычеркнем из последовательности все числа, кратные 3, кроме самого числа 3.
- Затем вычеркнем из последовательности все числа, кратные 5, кроме самого числа 5.
- И т. д.: на каждом шаге мы ищем очередное не вычеркнутое пока число и вычеркиваем все кратные ему числа, кроме него самого.
- После завершения всех шагов оставшиеся невычеркнутыми числа являются искомыми простыми числами.

21.2. Демонстрация работы алгоритма

Проиллюстрируем работу решета Эратосфена при $n = 20$.

- Изначально выписана последовательность всех чисел от 2 до 20:

1. 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

- Затем вычеркиваются числа, кратные 2:

2. 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~

- Затем вычеркиваются числа, кратные 3 (некоторые числа становятся вычеркнутыми дважды):

3. 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

- Затем вычеркиваются числа, кратные 5 (на самом деле все такие числа уже вычеркнуты):

4. 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

- То же самое для 7, 11, 13, 17, 19. Итог:

5. 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

- Простыми являются оставшиеся числа, т. е. следующая последовательность:

6. 2, 3, 5, 7, 11, 13, 17, 19.

21.3. Доказательство корректности решета Эратосфена

С одной стороны, все вычеркнутые числа являются составными, поскольку алгоритм вычеркивает только числа, кратные каким-либо другим числам, большим единицы.

С другой стороны, все составные числа будут вычеркнуты — например, это можно доказать по индукции. При $n = 2$ составных чисел нет, поэтому это утверждение верно. Предположим, что это утверждение верно для $n - 1$ и докажем его для n. В случае когда n простое, истинность утверждения очевидна; в случае же когда n составное, то, если мы обозначим через k наименьший простой делитель n, алгоритм вычеркнет число n при обработке числа k.

21.4. Время работы решета Эратосфена

Решето Эратосфена обладает высокой производительностью: его асимптотическая сложность составляет $O(n \log \log n)$.

> Эта оценка может показаться неожиданной, ведь алгоритм представляет собой два вложенных цикла, оба выполняющихся до достижения n (хотя и с разными шагами).

Доказательство этой асимптотической оценки нетривиально, и, с практической точки зрения, его знание не требуется для успешного применения алгоритма. Однако в целях полноты приведем здесь упрощенное, не до конца строгое доказательство.

Итак, нам нужно оценить суммарное число операций, совершаемых решетом Эратосфена. Заметим, что для каждого простого числа p ($2 \le p \le n$) алгоритм вычерки-

вает $\lfloor n / p \rfloor - 1$ чисел, поэтому суммарная оценка числа операций сводится к оценке следующей суммы:

$$\sum_{\substack{p \in P, \\ p \le n}} \frac{n}{p},$$

где P — множество всех простых чисел. Пользуясь приведенным в *главе 15* фактом о том, что количество простых чисел до n приблизительно равно $n / \ln n$, и приблизительно оценивая k-е простое число как $k \ln k$, эту сумму можно преобразовать следующим образом:

$$\sum_{\substack{p \in P, \\ p \le n}} \frac{n}{p} \approx n\left(\frac{1}{2} + \sum_{k=2}^{n/\ln n} \frac{1}{k \ln k}\right),$$

где первое слагаемое (при $k = 1$) вынесено из-под суммы, поскольку при $k = 1$ знаменатель дроби обратится в нуль. Сумму в правой части выражения можно приблизительно оценить с помощью интеграла, поскольку сумму ряда можно расценивать как приближение интеграла по формуле прямоугольников:

$$\sum_{k=2}^{n/\ln n} \frac{1}{k \ln k} \approx \int_{2}^{n/\ln n} \frac{1}{x \ln x} dx.$$

Этот интеграл вычисляется точно: первообразная функция равна $\ln \ln x$, и, подставляя ее, мы получаем:

$$\int_{2}^{n/\ln n} \frac{1}{x \ln x} dx = \ln \ln \left(\frac{n}{\ln n}\right) - \ln \ln 2 = \ln \ln n - \ln \ln \ln n - \ln \ln 2.$$

Поскольку нас интересует асимптотическая оценка, мы отбрасываем константы и члены меньшего порядка и получаем, что интеграл является величиной $O(\ln \ln n)$. Возвращаясь к исходному выражению для числа операций решета Эратосфена, мы получаем итоговую асимптотическую оценку:

$$O(n \ln \ln n).$$

21.5. Базовые оптимизации решета Эратосфена

Перед тем как перейти к реализации решета Эратосфена, рассмотрим несколько идей для оптимизаций, которые легко реализуются и потому всегда используются на практике.

21.5.1. Просеивание, начиная с квадратов

Идея первой оптимизации звучит следующим образом: будем вычеркивать числа, начиная с квадрата текущего простого числа. Иными словами, на шаге обработки

простого числа k и вычеркивания чисел вида $k \cdot i$ мы будем итерироваться не по $i = 2...\lfloor n/k \rfloor$, а по $i = k...\lfloor n/k \rfloor$.

Почему эта оптимизация корректна? Она основана на утверждении, что все составные числа в промежутке $(k; k^2)$ уже вычеркнуты на предыдущих шагах. В самом деле, у всякого составного числа есть простой делитель, не превосходящий корня из него. Следовательно, у всех чисел, меньших k^2, есть простой делитель, меньший k. Отсюда следует, что все такие числа были вычеркнуты решетом Эратосфена при обработке простых чисел до k.

На практике эта оптимизация дает минимум 30% прироста скорости.

21.5.2. Перебор до квадратного корня

Идея второй оптимизации — выполнять шаги вычеркиваний только для простых чисел, не превосходящих $\sqrt{n}$.

Корректность этой оптимизации вытекает из предыдущего улучшения: поскольку для каждого простого числа k мы вычеркиваем только числа начиная с k^2, то для всех $k > \sqrt{n}$ ни одного вычеркивания произведено не будет.

На практике эта оптимизация дает минимум 15% прироста скорости.

21.6. Реализация решета Эратосфена

Для наглядности приведем программу, считывающую n и выводящую $\pi(n)$, т. е. количество простых чисел в отрезке $[1; n]$.

Python

```
from math import *
n = int(input())
prime = [True] * (n + 1)
prime[0] = prime[1] = False
for i in range(
    2, round(sqrt(n)) + 1):
  if prime[i]:
    for j in range(
        i * i, n + 1, i):
      prime[j] = False
print(sum(prime))
```

C++

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int n;
  cin >> n;
  vector<char> prime(n + 1, 1);
  for (int i = 2; i * i <= n;
       ++i) {
    if (!prime[i])
      continue;
    for (int j = i * i; j <= n;
         j += i) {
      prime[j] = false;
    }
  }
  int count = 0;
```

```
  for (int i = 2; i <= n; ++i) {
    if (prime[i])
      ++count;
  }
  cout << count << endl;
}
```

21.7. Дальнейшие оптимизации решета Эратосфена

Алгоритм решета Эратосфена, даже в его базовой реализации, приведенной выше, весьма быстр на практике. Например, на компьютере автора тест $n = 10^8$ обрабатывается программой на C++ за 1,6 с.

Однако при необходимости есть еще несколько относительно простых оптимизаций, которые позволяют дополнительно ускорить работу решета Эратосфена.

21.7.1. Битовое сжатие

Вместо того чтобы выделять по одному байту для каждого числа (для хранения отметок того, простое это число или нет), можно хранить отметки в отдельных **битах**, достигая тем самым **восьмикратной экономии памяти**.

Влияние этой оптимизации на скорость работы не так очевидно. С одной стороны, выделение памяти происходит быстрее, равно как и операции чтения-записи (за счет того, что данные лучше помещаются в процессорный кэш). С другой стороны, каждая операция чтения и записи будет требовать дополнительных процессорных инструкций из-за выполнения побитовых операций.

Однако практика показывает, что на типичном компьютере первая группа факторов берет верх, и эта оптимизация ускоряет алгоритм на 40% и более по сравнению с базовой реализацией.

Реализация

В то время как при использовании C++ мы можем просто воспользоваться классом `bitset` из стандартной библиотеки, при использовании Python битовые операции приходится реализовывать вручную.

Python

```
from math import *
n = int(input())
size = n // 8 + 1
composite = [0] * size

def get(idx):
  byte = idx >> 3
  bit = idx & 7
```

C++

```
#include <bitset>
#include <iostream>
using namespace std;
const int MAXN = 100 * 1000 * 1000;
bitset<MAXN + 1> composite;
int main() {
  int n;
  cin >> n;
```

```
  return (composite[byte] &
          (1 << bit))
def set(idx):
  byte = idx >> 3
  bit = idx & 7
  composite[byte] |= 1 << bit

for i in range(
    2, round(sqrt(n)) + 1):
  if not get(i):
    for j in range(
        i * i, n + 1, i):
      set(j)
print(sum(not get(i)
          for i in range(
              2, n + 1)))
```

```
  for (int i = 2; i * i <= n;
       ++i) {
    if (composite[i])
      continue;
    for (int j = i * i; j <= n;
         j += i) {
      composite[j] = true;
    }
  }
  cout << n - 1 - composite.count()
       << endl;
}
```

Пояснения к использованным в реализации на Python битовым операциям

Поскольку в одном байте содержится 8 бит, то в сжатом представлении элемент с номером i хранится в байте номер $\lfloor i / 8 \rfloor$ в бите номер $i \bmod 8$. Быстрая реализация деления на степень двойки — это битовый сдвиг вправо на эту степень, что кодируется для деления на 8 в виде «>> 3». А быстрая реализация взятия остатка от деления на степень двойки — это извлечение соответствующего числа младших битов, что с помощью побитового «И» кодируется для деления на 8 в виде «& 7». Для проверки или установки заданного бита используется операция побитового «И» и побитового «ИЛИ» с аргументом, равным соответствующей степени двойки — что кодируется в виде «1 << bit».

Фиксированный размер типа bitset в C++

Небольшим недостатком типа bitset является то, что его размер должен быть указан в момент компиляции. В спортивном программировании не требуется экономить память для малых входных данных, поэтому имеет смысл, как в нашей реализации выше, всегда выделять память под самый большой из возможных n. Другая альтернатива из стандартной библиотеки — vector<bool> — хотя и использует аналогичное сжатое представление, существенно медленнее на практике. Поэтому в случае если потребуется динамическое выделение памяти в сочетании с высокой производительностью, придется реализовывать битовые операции вручную — аналогично приведенной выше реализации на Python.

21.7.2. Рассмотрение только нечетных чисел

Эта оптимизация основана на том факте, что все простые числа, кроме числа «2» — нечетные. Таким образом, нет нужды хранить и поддерживать пометки для четных чисел, что сокращает вдвое потребление памяти и заметно уменьшает число выполняемых операций.

На практике эта оптимизация дает прирост в скорости в 50% и более. Комбинируя это с предыдущей оптимизацией, мы уже получаем в 3–4 раза более быструю, чем базовая, реализацию!

Реализация

Приведем реализацию, комбинирующую эту и все предыдущие оптимизации:

Python

```
from math import *
import sys
n = int(input())
if n == 1:
  print(0)
  sys.exit(0)
size = n // 16 + 1
composite = [0] * size

def get(idx):
  byte = idx >> 3
  bit = idx & 7
  return (composite[byte] &
          (1 << bit))
def set(idx):
  byte = idx >> 3
  bit = idx & 7
  composite[byte] |= 1 << bit

for i in range(
    3, round(sqrt(n)) + 1, 2):
  if not get(i // 2):
    for j in range(
        i * i, n + 1, i * 2):
      set(j // 2)
print((n + 1) // 2 - sum(
    get(i) != 0 for i in range(
        n // 2 + 1)))
```

C++

```
#include <bitset>
#include <iostream>
using namespace std;
const int MAXN = 100 * 1000 * 1000;
bitset<MAXN / 2 + 1> composite;
int main() {
  int n;
  cin >> n;
  if (n == 1) {
    cout << 0 << endl;
    return 0;
  }
  for (int i = 3; i * i <= n;
       i += 2) {
    if (composite[i / 2])
      continue;
    for (int j = i * i;
         j <= n; j += i * 2) {
      composite[j / 2] = true;
    }
  }
  cout << (n + 1) / 2 -
          composite.count()
       << endl;
}
```

Дальнейшая оптимизация

В приведенных выше реализациях можно избавиться от умножений и делений на 2 внутри вложенного цикла, тем самым дополнительно ускорив их на несколько процентов. Для этого потребуется аккуратный учет граничных случаев, поскольку в зависимости от четности n будет меняться правая граница цикла. Так как это сделает код менее наглядным, мы не стали пользоваться этой оптимизацией.

21.7.3. Блочное решето

Эта оптимизация направлена, в первую очередь, на дальнейшее **уменьшение потребления памяти** — в случае когда число n настолько велико, что одномоментное выделение массива соответствующей длины становится невозможным.

Эта оптимизация заключается в следующем. Разобьем отрезок $[1; n]$ на несколько блоков фиксированной длины k; удобно положить длину k равной $\sqrt{n}$ или большему числу. Тогда можно искать простые числа в каждом из этих блоков по отдельности: сначала найти все простые числа в $[1; k]$, затем в $[k+1; 2k]$ и т. д. Если $k \geq \sqrt{n}$, то обработка первого блока будет выглядеть как обычное решето Эратосфена, а обработка всех остальных блоков будет заключаться в их просеивании простыми числами из первого блока.

Таким образом, эта оптимизация снижает потребление памяти с $O(n)$ до $O(k)$ — например, $O(\sqrt{n})$.

Время работы алгоритма сохраняет все ту же асимптотику $O(n \log \log n)$, но на практике алгоритм, скорее всего, будет несколько медленнее предыдущих вариантов из-за более сложного расчета индексов и дополнительных накладных расходов для больших простых чисел. В зависимости от выбранного подхода и качества реализованных низкоуровневых оптимизаций этот вариант алгоритма может работать медленнее на 50–100% при относительно небольших ограничениях, но, начиная с некоторого порога (на компьютере автора это приблизительно $n \geq 10^8$), блочное решето становится быстрее.

Мы не приводим здесь реализацию блочного решета, поскольку она существенно более объемная, чем приведенные ранее примеры кода, и при этом не содержит новых реализационных идей.

21.8. Пример решения задачи. Подсчет простых чисел в отрезке

Задача. Даны два числа L и R таких, что $1 \leq L \leq R \leq 10^{16}$ и $R - L \leq 10^6$. Требуется найти количество простых чисел в отрезке $[L; R]$.

Примеры

Входные данные	**Требуемый результат**
`10 20`	`4`
Входные данные	**Требуемый результат**
`1000000000000 1000000000100`	`4`

Решение

Легко понять, что тривиальная проверка всех чисел в заданном отрезке путем перебора их делителей потребует слишком много времени: $O\left((R-L)\sqrt{R}\right)$, что при заданных ограничениях составило бы порядка 10^{14}.

Воспользуемся решетом Эратосфена, как быстрым способом проверки большого количества чисел на простоту. Стандартный алгоритм, однако, работает только с отрезками вида [1; n], что для нашей задачи не подходит из-за больших ограничений на R.

Заметим, что для проверки на простоту чисел в отрезке [L; R] достаточно знать все простые числа, не превосходящие $\sqrt{R}$. Ограничения позволяют запустить стандартный алгоритм решета Эратосфена для чисел $\left[1;\sqrt{R}\right]$. После этого мы можем запустить обход с «просеиванием» отрезка $R - L$ найденными простыми числами.

Пользуясь теми же рассуждениями, что и при оценке времени работы решета Эратосфена, мы получаем, что это решение будет иметь ту же самую «почти линейную» асимптотику, только уже относительно величин $\sqrt{R}$ и $R - L$.

Реализация

Будем использовать реализацию на основе битовых масок, как достаточно быструю и экономную по памяти.

Python

```
from math import *
left, right = map(
    int, input().split())
left = max(left, 2)
prefix = round(sqrt(right)) + 1
composite_prefix = \
    [0] * (prefix // 8 + 1)

def get(idx):
  byte = idx >> 3
  bit = idx & 7
  return (composite_prefix[byte] &
          (1 << bit))
def set(idx):
  byte = idx >> 3
  bit = idx & 7
  composite_prefix[byte] |= \
      1 << bit

for i in range(
    2, round(sqrt(prefix)) + 1):
  if not get(i):
    for j in range(
```

C++

```
#include <algorithm>
#include <bitset>
#include <iostream>
using namespace std;
const int SQRT = 100 * 1000 * 1000;
const int SEGM = 1000 * 1000;
bitset<SQRT + 1> composite_prefix;
bitset<SEGM + 1> composite;
int main() {
  int64_t left, right;
  cin >> left >> right;
  if (left == 1)
    ++left;
  for (int i = 2; i * i <= SQRT;
       ++i) {
    if (composite_prefix[i])
      continue;
    for (int j = i * i; j <= SQRT;
         j += i) {
      composite_prefix[j] = true;
    }
  }
  for (int64_t i = 2; i <= SQRT;
```

```
        i * i, prefix + 1, i):
      set(j)
composite = \
    [0] * (right - left + 1)
for i in range(2, prefix + 1):
  if not get(i):
    begin = (left + i - 1) // i * i
    begin = max(begin, i * 2)
    for j in range(begin,
                   right + 1, i):
      composite[j - left] = True
print(right - left + 1 -
      sum(composite))
```

```
       ++i) {
    if (composite_prefix[i])
      continue;
    int64_t begin =
        (left + i - 1) / i * i;
    begin = max(begin, i * 2);
    for (int64_t j = begin;
         j <= right;
         j += i) {
      composite[j - left] = true;
    }
  }
  cout << right - left + 1 -
          composite.count()
       << endl;
}
```

Комментарии к реализациям

В начале решений мы избавляемся от случая $L = 1$, поскольку иначе единица была бы неправильно обработана, как если бы она была простым числом. Стартовая позиция `begin` вычисляется как наименьшее из кратных `i` чисел, которые меньше либо равны `left`, для чего мы одно число делим и обратно умножаем на другое с округлением вверх. Последний неочевидный момент: не следует допускать, чтобы `begin` был равен `i`, иначе мы бы вычеркнули само простое число `i` тоже.

Глава 22.
Двоичное возведение в степень

Алгоритм двоичного возведения в степень (или бинарного возведения в степень) позволяет эффективно вычислять большие степени чисел. Как правило, речь при этом идет о возведении в степень **по модулю**, поскольку без этого степени чисел быстро переполняют любые встроенные целочисленные типы.

Рассмотрим несколько примеров:

- если требуется посчитать $2^{20} \bmod 17$, то ответом будет 16, поскольку таков будет остаток от деления $2^{20} = 1\,048\,576$ на 17;
- ответом для $2^{1000} \bmod 11$ будет 1. Например, в этом можно убедиться, зная, что $2^{10} \bmod 11 = 1$, поскольку единица в любой степени будет оставаться единицей.

Кроме того, как мы увидим далее, двоичное возведение в степень можно применять и в некоторых других задачах.

22.1. Ключевая идея

Базовую идею можно проиллюстрировать следующим примером. Допустим, нам требуется эффективно вычислять большие степени числа 3. Тогда можно заметить, что вместо последовательного умножения на 3 можно возводить предыдущий результат в квадрат:

- $3^1 = 3$;
- $3^2 = 9$;
- $3^4 = (3^2)^2 = 9^2 = 81$;
- $3^8 = (3^4)^2 = 81^2 = 6561$;
- и т. д.

Точно так же этот прием можно использовать, когда вычисления производятся по какому-нибудь модулю, например, 17:

- $3^1 \bmod 17 = 3$;
- $3^2 \bmod 17 = 9$;
- $3^4 \bmod 17 = (3^2 \bmod 17)^2 \bmod 17 = 9^2 \bmod 17 = 13$;
- $3^8 \bmod 17 = (3^4 \bmod 17)^2 \bmod 17 = 13^2 \bmod 17 = 16$.

Таким образом, мы можем эффективно вычислять ответ для степеней, которые являются степенью двойки. Однако этот прием можно развить до алгоритма, эффек-

тивно работающего для любых степеней — это и будет алгоритмом двоичного возведения в степень.

22.2. Алгоритм двоичного возведения в степень

Для простоты описания сначала рассмотрим задачу возведения в степень без взятия по модулю. Предположим, что требуется вычислить b-ю степень числа a. Тогда алгоритм будет выглядеть следующим образом:

- если $b = 0$, то ответ равен единице;
- если b четно и больше нуля, то перейдем к возведению a^2 в степень $b / 2$;
- если b нечетно, то домножим ответ на текущее a и перейдем к возведению a в степень $b - 1$.

Справедливость этого алгоритма базируется на том, что для четных степеней искомое выражение можно преобразовать следующим образом:

$$a^b = \left(a^2\right)^{b/2},$$

а для нечетных степеней его можно свести к четному случаю следующим образом:

$$a^b = a \cdot a^{b-1}.$$

Наконец, заметим, что для вычисления степени по модулю те же равенства, а, значит, и тот же алгоритм по-прежнему верны. Алгоритм верен как для простых, так и для составных модулей.

22.3. Иллюстрация работы алгоритма

Рассмотрим работу алгоритма двоичного возведения в степень на примере вычисления $3^{14} \bmod 31$:

- поскольку 14 — четное число, мы переходим к возведению $3^2 = 9$ в степень 7;
- степень 7 нечетна, поэтому домножаем ответ на 9 и переходим к возведению 9 в степень 6;
- степень 6 четна, поэтому переходим к возведению числа 19 ($9^2 = 19 \bmod 31$) в степень 3;
- степень 3 нечетна, поэтому домножаем ответ на 19 и переходим к возведению в степень 2;
- степень 2 четна, поэтому переходим к возведению числа 20 ($19^2 = 20 \bmod 31$) в степень 1;
- степень 1 нечетна, поэтому домножаем ответ на 20 и переходим к возведению в степень 0;
- достигнута степень 0, и алгоритм завершает свою работу. Накопленный за все шаги ответ равен $9 \cdot 19 \cdot 20 = 10 \bmod 31$. Таким образом, результат равен 10.

Таким образом, мы нашли ответ за шесть умножений, что существенно быстрее, чем 14 операций, которые потребовались бы тривиальному алгоритму. Чем больше степень, тем больше будет преимущество быстрого алгоритма.

22.4. Время работы двоичного возведения в степень

Каждый шаг алгоритма уменьшает показатель степени b: уменьшает вдвое, если он был четным, или уменьшает на единицу, если он был нечетным. Поскольку алгоритм останавливается при достижении нуля, то делений пополам не могло быть больше, чем $\lceil \log_2 b \rceil$. А уменьшений на единицу могло быть максимум $1 + \lceil \log_2 b \rceil$, поскольку после каждого уменьшения на один следует деление пополам.

Таким образом, всего алгоритм совершит порядка $\log_2 b$ операций умножения, что дает оценку алгоритма $O(\log b)$. Это существенно лучше тривиального алгоритма, который вычислял бы степень за линейное время.

22.5. Реализация двоичного возведения в степень

При реализации удобно применить следующее небольшое упрощение: объединим шаг деления пополам с предшествующим ему шагом уменьшения на единицу. Таким образом, получается, что каждый шаг алгоритма состоит из двух частей: уменьшение b на единицу в случае его нечетности и затем деление b надвое. Это позволяет избавиться от части ненужных проверок и при этом добиться лаконичного кода.

Обратим внимание: приведенная реализация верна при $p \geq 2$, $0 \leq a < p$, $b \geq 0$. Кроме того, в случае $a = b = 0$ будет возвращена единица (с математической точки зрения результат такой операции не определен).

Python

```
def bin_pow(a, b, p):
  res = 1
  while b:
    if b % 2:
      res = (res * a) % p
    a = (a * a) % p
    b //= 2
  return res
```

C++

```
int64_t BinPow(
    int64_t a, int64_t b,
    int64_t p) {
  int64_t res = 1;
  while (b) {
    if (b % 2)
      res = (res * a) % p;
    a = (a * a) % p;
    b /= 2;
  }
  return res;
}
```

Верхние границы на входные параметры для решения на C++

При больших входных данных возможны целочисленные переполнения при вычислении произведений. Приведенная реализация будет корректно работать для $p \leq 2^{31}$.

Оптимизации с помощью битовых операций

Приведенные реализации можно было бы дополнительно оптимизировать за счет замены операций деления на битовые операции. Вместо деления можно использовать битовый сдвиг вправо на один, а вместо вычисления остатка — операцию «логическое И» с аргументом, равным единице. Вопреки ожиданиям оптимизирующие компиляторы и интерпретаторы не производят этой оптимизации самостоятельно. Впрочем, в большинстве случаев такая оптимизация двоичного возведения даст лишь пренебрежимо малое ускорение, поскольку остальные тяжелые операции — умножение и взятие остатка по заданному модулю — сохранятся.

22.6. Пример решения задачи. Последние цифры степени

Задача. Даны два числа: a и b. Требуется посчитать последние 9 цифр в десятичной записи числа a^b. Ответ следует выводить с лидирующими нулями, в том случае если число a^b имеет более девяти цифр в записи; иначе лидирующих нулей быть не должно.

Входные данные состоят из двух разделенных пробелом целых неотрицательных чисел. Хотя бы одно из чисел отлично от нуля. Числа не превосходят 10^9.

Примеры

Входные данные	**Требуемый результат**
8 10	073741824
Входные данные	**Требуемый результат**
2 5	32

Пояснения к примерам

В первом примере $8^{10} = 1\,073\,741\,824$. Поскольку длина этого числа составляет 10 цифр, при печати ответа выводится ровно 9 цифр, включая лидирующий ноль.

Решение

Эта задача практически совпадает с описанной ранее стандартной задачей для алгоритма двоичного возведения в степень: степень в текущей задаче равна 10^9.

Однако нюанс, связанный с лидирующими нулями, требует специальной обработки, поскольку с точки зрения операции взятия по модулю нет никакой разницы между тем, превосходит результат этот модуль или нет.

Предложим простой способ решения этой проблемы: реализуем тривиальный алгоритм возведения в степень путем умножения с дополнительным критерием остановки по достижению 10^9. Тогда, если наивный алгоритм завершился до достижения этой границы, будет правильно вывести его результат без лидирующих нулей. В противном случае запустим алгоритм двоичного возведения в степень и выведем его результат с лидирующими нулями.

Асимптотика решения составит $O(\log b)$, поскольку добавка, вызванная работой тривиального алгоритма, не превосходит константы.

Реализация

Python

```
P = 10 ** 9

def naive_pow(a, b):
  if a <= 1:
    return a
  res = 1
  for _ in range(b):
    res *= a
    if res >= P:
      return None
  return res

def bin_pow(a, b):
  res = 1
  while b:
    if b % 2:
      res = (res * a) % P
    a = (a * a) % P
    b //= 2
  return res

a, b = map(int, input().split())
res = naive_pow(a, b)
if res is None:
  res = bin_pow(a, b)
  print('{:09d}'.format(res))
else:
  print(res)
```

C++

```
#include <cstdint>
#include <iomanip>
#include <iostream>
using namespace std;

const int64_t P =
    1000 * 1000 * 1000;

int64_t NaivePow(int64_t a,
                 int64_t b) {
  if (a <= 1)
    return a;
  int64_t res = 1;
  for (int i = 0; i < b; ++i) {
    res *= a;
    if (res >= P)
      return -1;
  }
  return res;
}

int64_t BinPow(int64_t a,
               int64_t b) {
  int64_t res = 1;
  while (b) {
    if (b % 2)
      res = (res * a) % P;
    a = (a * a) % P;
    b /= 2;
  }
  return res;
}

int main() {
  int64_t a, b;
  cin >> a >> b;
  int64_t res = NaivePow(a, b);
  if (res != -1) {
    cout << res << endl;
```

```
  } else {
    res = BinPow(a, b);
    cout << setfill('0') << setw(9)
         << res << endl;
  }
}
```

22.7. Пример решения задачи. Обратное по простому модулю. Малая теорема Ферма

Определение. Пусть дано целое число a и модуль p. Тогда число b называется обратным к a, если верно:

$$a \cdot b = 1 \mod p.$$

Можно использовать и обозначение a^{-1}, тогда определение будет выглядеть следующим образом:

$$a \cdot a^{-1} = 1 \mod p.$$

Известна математическая теорема, что обратный элемент **существует тогда и только тогда**, когда a отлично от нуля и взаимно просто с p (напомним, это значит, что наибольший общий делитель этих двух чисел должен быть равен единице).

Задача. Дано натуральное число a и простой модуль p. Требуется найти число, обратное к a по модулю p.

Входные данные состоят из натуральных чисел a и p, разделенных пробелом. Гарантируется, что $a < p \le 10^9$. Вывести требуется одно число — искомый обратный элемент в промежутке $[0; p)$.

Примеры

Входные данные	**Требуемый результат**
7 11	8
Входные данные	**Требуемый результат**
1 127	1

Пояснения к примерам

В первом примере ответом является число 8, поскольку $7 \cdot 8 = 56 = 1 \mod 11$. Во втором примере обратным к единице является сама же единица.

Решение

Решение основывается на одном из классических результатов в математике — **малой теореме Ферма**. Она гласит, что для всякого простого p и ненулевого a справедливо:

$$a^{p-1} = 1 \mod p.$$

Доказательство этой теоремы выходит за рамки данной книги; его знание также не является необходимым для успешного применения этой теоремы на практике.

Используя эту теорему, можно найти формулу для вычисления обратного элемента. Для этого достаточно обе части этого равенства домножить на a^{-1}, в результате чего мы получаем:

$$a^{p-1} \cdot a^{-1} = a^{-1} \mod p,$$

что, если упростить это выражение, и дает требуемую формулу:

$$a^{-1} = a^{p-2} \mod p.$$

Для быстрого вычисления ответа по этой формуле следует применить алгоритм двоичного возведения в степень.

Обобщение решения на случай составного модуля

Хотя малая теорема Ферма верна только для простых модулей, существует ее обобщение, работающее для произвольных модулей. Это теорема Эйлера, гласящая, что $a^{\phi(p)} = 1 \mod p$, где $\phi(p)$ — функция Эйлера, которая определяется как число взаимно простых с p чисел. Этот математически красивый результат, однако, сравнительно редко применим в спортивном программировании, поскольку эффективное вычисление функции Эйлера требует факторизации p: $\phi(p) = p \cdot (1 - 1 / q_1) \cdot \ldots \cdot (1 - 1 / q_k)$, где q_i — простые делители p. Более практичный алгоритм определения обратного элемента по произвольному модулю основан на модификации алгоритма Евклида и будет рассмотрен нами в отдельной главе.

Реализация

Мы опустим реализацию функции двоичного возведения в степень, поскольку она ничем не отличается от приведенной ранее в этой главе.

Python

```
# ... bin_pow() ...
a, p = map(int, input().split())
print(bin_pow(a, p - 2, p))
```

C++

```
#include <cstdint>
#include <iostream>
using namespace std;
// ... BinPow() ...
int main() {
  int64_t a, p;
  cin >> a >> p;
  cout << BinPow(a, p - 2, p)
       << endl;
}
```

22.8. Пример решения задачи. Быстрое вычисление чисел Фибоначчи. Двоичное возведение матриц в степень

Задача. Дано число n и модуль p. Требуется найти n-е число Фибоначчи, взятое по модулю p.

Напомним, что последовательность F_k чисел Фибоначчи определяется следующим образом: $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-2} + F_{k-1}$.

Входные данные состоят из целых чисел n ($0 \le n \le 10^9$) и p ($2 \le p \le 10^9$). Вывести требуется единственное число — величину $F_n \bmod p$.

Примеры

Входные данные	**Требуемый результат**
`8 17`	`4`
Входные данные	**Требуемый результат**
`12 127`	`17`

Пояснения к примерам

В первом примере $F_8 = 21$, что при делении на 17 дает остаток 4. Во втором примере $F_{12} = 144$, что по модулю 127 равно 17.

Решение

Несмотря на простой вид рекуррентного соотношения, удобной формулы для вычисления n-го числа Фибоначчи не существует. Многие известные из математики формулы неэффективны для практического применения в плане производительности либо точности (некоторые формулы используют дробные числа, такие, как золотое сечение).

Однако существует способ применить метод двоичного возведения в степень:

$$\begin{pmatrix} F_2 & F_3 \end{pmatrix} = \begin{pmatrix} F_0 & F_1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2,$$

$$\begin{pmatrix} F_3 & F_4 \end{pmatrix} = \begin{pmatrix} F_0 & F_1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^3.$$

Теперь мы можем записать общее выражение для вычисления n-го элемента ряда Фибоначчи:

$$\begin{pmatrix} F_n & F_{n+1} \end{pmatrix} = \begin{pmatrix} F_0 & F_1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n.$$

Задача свелась к быстрому возведению матрицы в степень. Следующий шаг — понять, что алгоритм **двоичного возведения** в степень применим и к матрицам снова благодаря ассоциативности операции матричного произведения. В самом деле, из ассоциативности следует, что для любой квадратной матрицы A в случае четной степени b верно:

$$A^b = \left(A^2\right)^{b/2},$$

а в случае нечетной степени b:

$$A^b = A \cdot A^{b-1}.$$

Последний шаг — заметим, что требование взятия по модулю ничего не меняет в приведенных выше рассуждениях. Таким образом, элементы матриц и все вычисления можно производить по заданном модулю p.

Итоговая асимптотика решения — такая же, как у операции двоичного возведения в степень, т. е. $O(\log n)$.

Реализация

Нам потребуется реализовать операцию перемножения двух матриц размером 2×2 с взятием всех операций по модулю p. Алгоритм двоичного возведения в степень останется практически таким же, как и раньше, только вместо операций с числами он будет оперировать матрицами. Изначально результат двоичного возведения в степень инициализируется единичной матрицей:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Наконец, ответ на задачу — искомое число F_n — будет содержаться в первой ячейке второй строки получившейся матрицы, как можно увидеть в приведенной выше формуле.

Python

```
n, p = map(int, input().split())

def matr_mul_2x2(a, b):
  res = [[0] * 2 for _ in range(2)]
  for i in range(2):
    for j in range(2):
      for k in range(2):
        res[i][j] += \
          a[i][k] * b[k][j]
        res[i][j] %= p
  return res

def bin_pow_matr_2x2(a, b):
  res = [[1, 0], [0, 1]]
  while b:
    if b % 2:
      res = matr_mul_2x2(res, a)
    a = matr_mul_2x2(a, a)
    b //= 2
  return res

matr = [[0, 1], [1, 1]]
matr = bin_pow_matr_2x2(matr, n)
print(matr[1][0])
```

C++

```
#include <cstdint>
#include <cstring>
#include <iostream>
using namespace std;

int64_t n, p;

void MatrMul2x2(int64_t a[2][2],
                int64_t b[2][2]) {
  int64_t res[2][2] = {};
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
      for (int k = 0; k < 2; ++k) {
        res[i][j] +=
            a[i][k] * b[k][j];
        res[i][j] %= p;
      }
    }
  }
  memcpy(a, res, sizeof(res));
}

void BinPowMatr2x2(
    int64_t a[2][2],
    int64_t b) {
```

```
  int64_t res[2][2] = {
      {1, 0}, {0, 1}};
  while (b) {
    if (b % 2)
      MatrMul2x2(res, a);
    MatrMul2x2(a, a);
    b /= 2;
  }
  memcpy(a, res, sizeof(res));
}

int main() {
  cin >> n >> p;
  int64_t matr[2][2] = {
      {0, 1}, {1, 1}};
  BinPowMatr2x2(matr, n);
  cout << matr[1][0] << endl;
}
```

Примечание по использованию массивов в решении на C++

Приведенная реализация использует для хранения матриц массивы, а не vector, поскольку вложенные массивы имеют гораздо более компактную запись, чем vector. Кроме того, хотя в данной задаче это не принципиально важно, операции с массивами будут происходить заметно быстрее, чем в случае vector, который при таком небольшом числе элементов привносит существенные накладные расходы.

22.9. Пример решения задачи. Физический движок

Задача. Перед командой, разрабатывающей движок двумерной физической симуляции, встала следующая задача. Есть некоторый физический процесс, который можно смоделировать в виде известной последовательности трансформаций координатной плоскости. Каждая трансформация представляет из себя либо сдвиг координатной плоскости на заданный вектор, либо растяжение координатной плоскости с заданным коэффициентом, либо вращение всей плоскости вокруг заданной точки на 90 градусов в указанном направлении. Цель — вычислить итоговое положение заданных точек после осуществления всей последовательности трансформаций, повторенной k раз.

Первая строка входных данных содержит число n точек, число m трансформаций и количество k повторов. В последующих n строках даны координаты x_i и y_i точек. Следующие m строк содержат описания трансформаций, каждое из которых имеет одну из следующих форм:

- «*1 A B*», что обозначает трансформацию сдвига, т. е. прибавление к абсциссам всех точек A и к ординатам всех точек B;

- «*2 C*», что обозначает трансформацию растяжения, т. е. умножение координат всех точек на *C*;
- «3 *P Q R*», где *R* равно 90 или −90, что обозначает трансформацию поворота на заданный угол (относительно направления по часовой стрелке) вокруг точки с координатами (*P, Q)*.

Все числа во входном наборе целые. Ограничения: $1 \le n, m \le 10^3$, $1 \le k \le 10^9$, $-100 \le x_i, y_i, A, B, P, Q \le 100$, $0 < C \le 10$.

Вывести требуется координаты всех *n* точек после применения описанных трансформаций, в виде *n* строк, каждая из которых содержит по два целых числа — новые координаты *i*-й точки входного набора. Гарантируется, что входные данные таковы, что никакая точка, изначально находившаяся в квадрате $|x_i|, |y_i| \le 100$, ни на каком шаге моделирования не будет иметь координаты, превосходящие по модулю 10^6.

Примеры

Входные данные	**Требуемый результат**
2 3 2 1 1 1 2 3 1 1 90 1 -1 -1 2 2	-2 2 -2 -2
Входные данные	**Требуемый результат**
1 1 10 10 1 2 2	10240 1024

Решение

Тривиальное решение заключалось бы в применении всех $m \cdot k$ трансформаций к каждой из *n* точек. Такое решение имело бы асимптотику $O(nmk)$, что при указанных ограничениях недостаточно быстро.

Каким образом можно компактно представить результат применения трансформаций, чтобы применить его к каждой из входных точек?

Выпишем сначала формулы того, как каждая трансформация преобразует координаты точек. Пусть (u, v) — координаты некоторой (произвольной) точки, а (u', v') — координаты этой же точки после применения трансформации. Тогда в зависимости от типа трансформации мы получаем:

- в случае трансформации сдвига:

$$u' = u + A,$$
$$v' = v + B;$$

- в случае трансформации растяжения:

$$u' = Cu,$$
$$v' = Cv;$$

- в случае трансформации поворота с параметрами $P = Q = 0$ и $R = 90$, как можно легко убедиться на паре примеров, координаты обмениваются местами, дополнительно меняя знак у ординаты:

$$u' = v,$$
$$v' = -u;$$

- в случае трансформации поворота с параметрами $P = Q = 0$ и $R = -90$, зависимость получается обратная:

$$u' = -v,$$
$$v' = u;$$

- в случае трансформации поворота с ненулевыми P и/или Q ее можно представить как комбинацию сдвига на $(-P, -Q)$, поворота вокруг начала координат (в соответствии с тем, что было рассмотрено выше) и последующего сдвига обратно на (P, Q).

Таким образом, каждая трансформация представляет из себя линейное преобразование: новая координата точки выражается через сумму ее изначальных координат с некоторыми постоянными коэффициентами. Следовательно, можно, как и в задаче про числа Фибоначчи, воспользоваться матричным представлением. В этом представлении, трансформация координат точки будет выражаться в виде произведения на некоторую матрицу. Такое представление позволит предварительно подсчитать результат всех трансформаций — этот результат будет равен произведению соответствующих матриц, возведенному в степень k.

Единственная сложность на этом пути — вопрос о том, каким образом представить в матричном виде трансформацию сдвига. Если бы мы ограничились только матрицей размера 2×2 — для абсцисс и для ординат — то мы смогли бы представить только линейные зависимости вида $u' = \alpha u + \beta v$, в то время как нам требуется вид $u' = \alpha u + \beta v + \gamma$.

Решением является ввод дополнительной, третьей размерности, помимо абсцисс и ординат. Если мы положим значение в этой третьей размерности всегда равным единице, то в матричном представлении можно будет использовать эту размерность для прибавления требуемой константы. Таким образом, общий вид матричного представления для одной трансформации будет следующим:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} = \begin{pmatrix} u & v & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

Заметим, что в этой матрице $a_{13} = a_{23} = 0$, поскольку только так можно получить единицу в третьей ячейке получившейся матрицы 1×3.

Осталось только выписать матричные представления для каждой из трансформаций, пользуясь приведенными выше формулами и обычным правилом для умножения матриц:

- трансформация сдвига:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ A & B & 1 \end{pmatrix};$$

- трансформация растяжения:

$$\begin{pmatrix} C & 0 & 0 \\ 0 & C & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

- трансформация поворота вокруг начала координат на, соответственно, 90 и −90 градусов:

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Подводя итог, решение представляет преобразование прочитанных трансформаций в матрицы, перемножение этих матриц, возведение в степень k с помощью алгоритма двоичного возведения в степень и затем вычисление ответа для каждой из n точек путем умножения ее на получившуюся матрицу. Асимптотика решения составит $O(n + m + \log k)$.

Обобщения этого решения

Аналогичное решение можно применять и при наличии других, более сложных трансформаций — например, поворота на произвольный угол. Единственное требование — каждая трансформация должна выражаться в виде линейной функции от координат. Из прочих возможных обобщений алгоритма — циклический повтор не всех трансформаций, а только определенного их подотрезка (решается возведением в степень соответствующего произведения матриц), и переход к бóльшим размерностям (решается переходом к матрицам размера $(D + 1) \times (D + 1)$, где D — размерность пространства).

Реализация

Python

```
def identity_matr():
  return [[1, 0, 0],
          [0, 1, 0],
          [0, 0, 1]]
def shift_matr(dx, dy):
  return [[1, 0, 0],
```

C++

```
#include <cstdint>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
```

```
          [0, 1, 0],
          [dx, dy, 1]]
def stretch_matr(c):
  return [[c, 0, 0],
          [0, c, 0],
          [0, 0, 1]]
def rotate_matr(angle):
  sgn = 1 if angle == 90 else -1
  return [[0, -sgn, 0],
          [sgn, 0, 0],
          [0, 0, 1]]

def matr_mul(a, b):
  res = [[0] * 3 for _ in range(3)]
  for i in range(3):
    for j in range(3):
      for k in range(3):
        res[i][j] += \
          a[i][k] * b[k][j]
  return res

def bin_pow(a, b):
  res = identity_matr()
  while b:
    if b % 2:
      res = matr_mul(res, a)
    a = matr_mul(a, a)
    b //= 2
  return res

n, m, k = map(int, input().split())
points = [
  list(map(int, input().split()))
  for _ in range(n)]
matr = identity_matr()
for _ in range(m):
  descr = list(map(
      int, input().split()))
  if descr[0] == 1:
    dx, dy = descr[1], descr[2]
    matr = matr_mul(
        matr, shift_matr(dx, dy))
  elif descr[0] == 2:
    c = descr[1]
```

```
void MakeIdentity(int matr[3][3]) {
  matr[0][0] = 1;
  matr[1][1] = 1;
  matr[2][2] = 1;
}
void MakeShift(int dx, int dy,
               int matr[3][3]) {
  matr[0][0] = 1;
  matr[1][1] = 1;
  matr[2][0] = dx;
  matr[2][1] = dy;
  matr[2][2] = 1;
}
void MakeStretch(
    int c, int matr[3][3]) {
  matr[0][0] = c;
  matr[1][1] = c;
  matr[2][2] = 1;
}
void MakeRotate(
    int angle, int matr[3][3]) {
  int sgn = angle == 90 ? 1 : -1;
  matr[0][1] = -sgn;
  matr[1][0] = sgn;
  matr[2][2] = 1;
}
void MatrMul(int a[3][3],
             int b[3][3]) {
  int res[3][3] = {};
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      for (int k = 0; k < 3; ++k) {
        res[i][j] +=
            a[i][k] * b[k][j];
      }
    }
  }
  memcpy(a, res, sizeof(res));
}
void BinPow(
    int a[3][3],
    int b) {
  int res[3][3] = {};
  MakeIdentity(res);
```

```
    matr = matr_mul(
        matr, stretch_matr(c))
  else:
    p, q, r = \
      descr[1], descr[2], descr[3]
    matr = matr_mul(
        matr, shift_matr(-p, -q))
    matr = matr_mul(
        matr, rotate_matr(r))
    matr = matr_mul(
        matr, shift_matr(p, q))
matr = bin_pow(matr, k)
for x, y in points:
  nx = (x * matr[0][0] +
        y * matr[1][0] +
        matr[2][0])
  ny = (x * matr[0][1] +
        y * matr[1][1] +
        matr[2][1])
  print(nx, ny)
```

```
  while (b) {
    if (b % 2)
      MatrMul(res, a);
    MatrMul(a, a);
    b /= 2;
  }
  memcpy(a, res, sizeof(res));
}

int main() {
  int n, m, k;
  cin >> n >> m >> k;
  vector<int> x(n), y(n);
  for (int i = 0; i < n; ++i) {
    cin >> x[i] >> y[i];
  }
  int matr[3][3] = {};
  MakeIdentity(matr);
  for (int i = 0; i < m; ++i) {
    int tp;
    cin >> tp;
    int cur[3][3] = {};
    if (tp == 1) {
      int dx, dy;
      cin >> dx >> dy;
      MakeShift(dx, dy, cur);
      MatrMul(matr, cur);
    } else if (tp == 2) {
      int c;
      cin >> c;
      MakeStretch(c, cur);
      MatrMul(matr, cur);
    } else if (tp == 3) {
      int p, q, r;
      cin >> p >> q >> r;
      MakeShift(-p, -q, cur);
      MatrMul(matr, cur);
      memset(cur, 0, sizeof(cur));
      MakeRotate(r, cur);
      MatrMul(matr, cur);
      memset(cur, 0, sizeof(cur));
      MakeShift(p, q, cur);
      MatrMul(matr, cur);
    }
```

```
  }
  BinPow(matr, k);
  for (int i = 0; i < n; ++i) {
    int nx = x[i] * matr[0][0] +
             y[i] * matr[1][0] +
             matr[2][0];
    int ny = x[i] * matr[0][1] +
             y[i] * matr[1][1] +
             matr[2][1];
    cout << nx << ' ' << ny
         << endl;
  }
}
```

22.10. Пример решения задачи. Подсчет путей фиксированной длины

Задача. Дан неориентированный граф и число k. Требуется посчитать для каждой пары вершин (i, j) количество различных путей, имеющих длину, равную k (т. е. содержащих k ребер). Путь может содержать какие-либо ребра более одного раза; при подсчете длины учитывается каждое вхождение каждого ребра. Пути считаются различными, если они отличаются набором использованных ребер или их порядком. Поскольку ответ может быть большим, количества надо искать по модулю заданного числа.

Входные данные содержат в первой строке число вершин n, число ребер m, число k и модуль p и затем описания ребер в последующих m строках. Каждое ребро описывается в виде двух чисел от 1 до n, задающих номера вершин-концов ребра. Гарантируется, что все ребра различны и нет ребер-петель. Ограничения: $1 \le n \le 100$, $0 \le k \le 10^9$, $2 \le p \le 10^9$.

Вывести требуется n строк по n чисел в каждой: j-е число в i-й строке должно быть равно числу путей из вершины i в вершину j по модулю p.

Примеры

Входные данные	**Требуемый результат**
3 3 2 100 1 2 2 3 3 1	2 1 1 1 2 1 1 1 2
Входные данные	**Требуемый результат**
2 1 3 2 2 1	0 1 1 0

Пояснения к примерам

В первом примере из вершины №1 есть два пути длины 2 в себя же: 1-2-1 и 1-3-1 и по одному пути длины 2 до остальных вершин: 1-2-3, 1-3-2. Ввиду симметричности графа количества путей, начинающихся в вершинах №2 и №3, такие же.

Решение

Количество путей — быстрорастущая величина. Для многих графов эта величина растет с экспоненциальной скоростью; простым примером является полный граф, в котором путь может проходить по любым вершинам в произвольном порядке. Поэтому тривиальный алгоритм поиска всех путей с помощью рекурсивного перебора можно сразу отбросить ввиду его неэффективности.

Попробуем двигаться, начиная с простых случаев $k = 0$ и $k = 1$ и постепенно переходя к бóльшим значениям k. При $k = 0$ можно дойти только из вершины в себя же, и единственным путем. При $k = 1$ ответом будет **матрица смежности** графа, поскольку она содержит единицы для тех пар вершин, между которыми есть ребро, а ребро — это путь длины 1.

Предположим, что мы знаем ответ для $k - 1$ — обозначим эту таблицу ответов через d_{k-1} — и попробуем найти на его основе ответ для k — обозначим искомую таблицу через d_k.

Этот подход можно назвать одним из видов динамического программирования. А с точки зрения математики данный подход имеет схожесть с методом математической индукции.

Для каждой пары вершин (i, j) мы можем перебрать ту вершину, через которую пойдет путь после покидания вершины i, и тем самым ответ можно выразить таким образом:

$$d_k[i][j] = \sum_{v \in N(i)} d_{k-1}[v][j],$$

где $N(i)$ — множество всех соседей вершины i.

Эта формула для динамического программирования могла бы быть решением задачи: она позволяет, стартовав с $k = 0$, последовательно пересчитывать таблицу ответов для каждого следующего значения k вплоть до требуемого. Однако следование этой формуле даст лишь решение за $O(n^3k)$, что слишком медленно при больших значениях k.

Для ускорения этой формулы заметим, что она есть не что иное, как **матричное произведение**: умножение матрицы смежности графа на матрицу d_{k-1}:

$$d_k[i][j] = \sum_{v=1}^{n} g[i][v] \cdot d_{k-1}[v][j].$$

Таким образом, ответ на задачу — это матрица смежности графа, возведенная в k-ю степень (каждый элемент матрицы при этом берется по модулю p). Для эффективного вычисления результата воспользуемся методом двоичного возведения в сте-

пень. Нам потребуется совершить $O(\log k)$ операций матричного произведения, каждая из которых (при тривиальной реализации) работает за время $O(n^3)$. Таким образом, итоговая асимптотика решения составит $O(n^3 \log k)$, что уже достаточно эффективно при заданных ограничениях.

Реализация

Python

```
def matr_mul(a, b):
  res = [[0] * n for _ in range(n)]
  for i in range(n):
    for j in range(n):
      for k in range(n):
        res[i][j] += \
          a[i][k] * b[k][j]
        res[i][j] %= p
  return res

def bin_pow_matr(a, b):
  res = [[0] * n for _ in range(n)]
  for i in range(n):
    res[i][i] = 1
  while b:
    if b % 2:
      res = matr_mul(res, a)
    a = matr_mul(a, a)
    b //= 2
  return res

n, m, k, p = map(
    int, input().split())
g = [[0] * n for _ in range(n)]
for _ in range(m):
  v1, v2 = map(
      int, input().split())
  v1 -= 1
  v2 -= 1
  g[v1][v2] = g[v2][v1] = 1
ans = bin_pow_matr(g, k)
for row in ans:
  print(*row)
```

C++

```
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;

using matr =
  vector<vector<int64_t>>;
int n, m, k;
int64_t p;

matr MatrMul(
    const matr& a, const matr& b) {
  matr res(n, vector<int64_t>(n));
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        res[i][j] +=
            a[i][k] * b[k][j];
        res[i][j] %= p;
      }
    }
  }
  return res;
}

matr BinPowMatr(
    matr a, int64_t b) {
  matr res(n, vector<int64_t>(n));
  for (int i = 0; i < n; ++i)
    res[i][i] = 1;
  while (b) {
    if (b % 2)
      res = MatrMul(res, a);
    a = MatrMul(a, a);
    b /= 2;
  }
  return res;
}
```

```
int main() {
  cin >> n >> m >> k >> p;
  matr g(n, vector<int64_t>(n));
  for (int i = 0; i < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1][v2] = g[v2][v1] = 1;
  }
  auto ans = BinPowMatr(g, k);
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      cout << ans[i][j] << ' ';
    cout << endl;
  }
}
```

Глава 23. Структуры данных. Дерево отрезков

Дерево отрезков — это универсальная структура данных, которая позволяет эффективно отвечать на большое количество запросов, относящихся к заданному набору объектов.

Основная мощь этой структуры данных заключается в ее большой гибкости, позволяющей применять ее при решении самых разнообразных задач.

23.1. Базовый вариант. Дерево для минимумов

Начнем с самого простого случая — дерева отрезков для минимумов.

> Задача поиска минимума на отрезке — одна из классических и наиболее изученных в информатике задач. По-английски ее называют range minimum query, или, сокращенно, RMQ. Ее важность обусловлена еще и тем, что она возникает как подзадача при решении многих проблем, например, при поиске наименьшего общего предка в графе.

23.1.1. Постановка задачи

Пусть дан массив целых чисел a_i ($i = 0...n - 1$). Поступает k запросов вида (p_j, q_j), и ответом на j-й запрос должно быть минимальное значение на соответствующем отрезке, т. е.:

$$\min_{i=p_j \ldots q_j} a_i.$$

23.1.2. Тривиальное решение

Тривиальное решение заключалось бы в просмотре всех элементов с p_j по q_j и сравнении их друг с другом; такое решение обрабатывало бы один запрос за время $O(n)$, что в сумме дало бы квадратичную асимптотику.

Дерево отрезков же позволит отвечать на каждый запрос за время $O(\log n)$, что в сумме по всем запросам даст $O(k \log n)$ — позволив тем самым решать эту задачу для сотен тысяч элементов и запросов.

23.1.3. Структура дерева отрезков

На концептуальном уровне дерево отрезков представляет собой многоуровневую структуру:

- первый — самый верхний — уровень содержит одну ячейку (вершину), в которой записано значение минимума для всего массива a_i;
- второй уровень содержит две ячейки; в первой ячейке записан минимум для левой половины массива a_i, во второй — для правой половины;
- третий уровень содержит четыре ячейки — по одной для каждой четверти массива;
- и т. д. — вплоть до последнего уровня, на котором n ячеек; в i-й записан минимум на соответствующем отрезке длины один, т. е. просто a_i.

Все вершины-ячейки организованы в виде одной **иерархии**: у каждой вершины (кроме вершин нижнего уровня) есть две дочерних вершины, которые разбивают подконтрольный ей отрезок на две половинки. Вершины нижнего уровня будем называть листьями.

Например, в случае $n = 8$ дерево отрезков можно изобразить следующим образом (рис. 23.1).

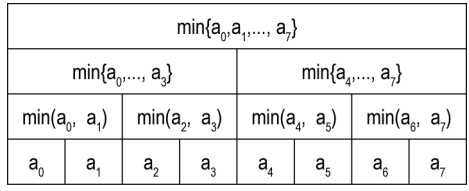

Рис. 23.1. Построение дерева отрезков для 8 элементов

Сразу оговоримся, что такая стройная организация наблюдается только в том случае, когда n является степенью двойки. В остальных случаях некоторые уровни дерева будут недозаполнены.

23.1.4. Размер дерева отрезков

Число уровней в дереве отрезков является величиной $O(\log n)$. Это так, поскольку верхний уровень относится ко всему массиву длины n, а каждый следующий уровень относится к вдвое меньшим, чем предыдущий, подотрезкам массива.

Число вершин в дереве отрезков является величиной $O(n)$, а именно, в точности равно $2n - 1$. Этот факт легче всего доказать в случае, когда n является степенью двойки, поскольку тогда суммарное число вершин по всем уровням равно:

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1,$$

где каждый элемент — это очередная степень двойки, а потому сумма дает $2n - 1$. То же самое верно и для случая произвольного n, что можно было бы доказать, например, методом математической индукции.

23.1.5. Хранение дерева отрезков в памяти

Древовидная структура наглядна и помогает развить интуитивное понимание этой структура данных, однако в программной реализации хранить дерево в виде настоящего графа было бы слишком непрактично.

Вместо этого будем использовать специальную **нумерацию** вершин:

1. Корень дерева является вершиной №1.
2. Если вершина дерева имеет номер x и не является листом, то ее левый сын имеет номер $2x$, а правый — номер $2x + 1$.

Можно убедиться, что при такой нумерации никакие две различных вершины не получат одинакового номера.

Эта нумерация позволяет компактно уложить все вершины дерева в один массив, а также избежать необходимости явно хранить ребра в памяти.

Например, для случая $n = 8$ нумерация примет вид (рис. 23.2).

<table>
<tr><td colspan="8">V = 1</td></tr>
<tr><td colspan="4">V = 2</td><td colspan="4">V = 3</td></tr>
<tr><td colspan="2">V = 4</td><td colspan="2">V = 5</td><td colspan="2">V = 6</td><td colspan="2">V = 7</td></tr>
<tr><td>V = 8</td><td>V = 9</td><td>V = 10</td><td>V = 11</td><td>V = 12</td><td>V = 13</td><td>V = 14</td><td>V = 15</td></tr>
</table>

Рис. 23.2. Нумерация вершин дерева отрезков для 8 элементов

Отметим, что, когда n не является степенью двойки, в нумерации могут возникать «пропуски», а номера вершин могут достигать или превосходить $2n$. Это не создает особых трудностей, надо лишь учитывать, что под хранение дерева отрезков потребуется **массив большего размера** — например, простой рекомендацией будет всегда выделять $4n$ ячеек памяти.

Например, в случае $n = 6$ в дереве отсутствуют вершины №10 и 11 (рис. 23.3).

<table>
<tr><td colspan="6">V = 1</td></tr>
<tr><td colspan="3">V = 2</td><td colspan="3">V = 3</td></tr>
<tr><td colspan="2">V = 4</td><td>V = 5</td><td colspan="2">V = 6</td><td>V = 7</td></tr>
<tr><td>V = 8</td><td>V = 9</td><td></td><td>V = 12</td><td>V = 13</td><td></td></tr>
</table>

Рис. 23.3. Нумерация вершин дерева отрезков для 6 элементов

23.1.6. Построение дерева отрезков

Дерево отрезков для минимумов можно построить **за линейное время**, двигаясь по его уровням снизу вверх:

- для вершин-листьев записывается значение соответствующего элемента массива;
- для всех остальных вершин значение равно минимуму из значений левого сына и правого сына.

23.1.7. Реализация построения дерева отрезков

Реализовать построение дерева отрезков проще всего в форме **рекурсивной функции**, которой передается номер вершины и границы (l, r) соответствующего ей отрезка. Изначально эту функцию надо запустить для корня дерева, т. е. для вершины №1 с границами $l = 0$, $r = n - 1$.

Сама функция будет содержать два случая:

- если $l = r$, то текущая вершина — лист, и поэтому в вершину дерева надо просто записать значение $a[l] = a[r]$;
- если же $l < r$, то следует произвести рекурсивные вызовы и затем объединить их результаты.
  - Один рекурсивный вызов следует сделать для левого сына, имеющего номер $2v$ и границы $[l; m]$, где $m = \lfloor (l + r) / 2 \rfloor$.
  - Другой рекурсивный вызов — для правого сына, имеющего номер $2v + 1$ и границы $[m + 1; r]$.
  - После этого останется только вычислить минимум из значений в дочерних вершинах и записать их в текущую вершину.

Для простоты реализации мы полагаем, что и массив a, и дерево t хранятся в глобальных переменных.

Python

```
def build(v, l, r):
  if l == r:
    t[v] = a[l]
    return
  mid = (l + r) // 2
  build(v * 2, l, mid)
  build(v * 2 + 1, mid + 1, r)
  t[v] = min(t[v * 2],
             t[v * 2 + 1])


# ...
t = [None] * (n * 4)
build(1, 0, n - 1)
```

C++

```
vector<int> a, t;

void Build(int v, int l, int r) {
  if (l == r) {
    t[v] = a[l];
    return;
  }
  int mid = (l + r) / 2;
  Build(v * 2, l, mid);
  Build(v * 2 + 1, mid + 1, r);
  t[v] = min(t[v * 2],
             t[v * 2 + 1]);
}
```

```
int main() {
  // ...
  t.resize(n * 4);
  Build(1, 0, n - 1);
}
```

23.1.8. Нахождение минимума на отрезке

Теперь мы подошли к самой интересной части: каким образом с помощью этой структуры данных можно быстро ответить на запрос минимума на произвольном отрезке $[p; q]$? Понятно, что целью является использовать информацию из как можно более «длинных» вершин дерева, а точнее, разбить отрезок запроса на наименьшее число отрезков, ответы для которых уже содержатся в дереве.

Это процедуру проще всего произвести опять же **рекурсивным** алгоритмом: мы начинаем обработку запроса $[p; q]$ вызовом от вершины $v = 1$, и функция производит следующие операции:

1. Если p и q совпадают с границами l и r отрезка, соответствующего вершине v, то в качестве результата возвращаем значение $t[v]$.
2. В противном случае, пересекаем отрезок $[p; q]$ с отрезком, соответствующим левому сыну, т. е. с $[l; m]$, где $m = \lfloor (l + r) / 2 \rfloor$. Если пересечение дало непустой отрезок, рекурсивно вызываем себя от левого сына, т. е. от вершины $2v$, с получившимся пересечением.
3. Аналогично, пересекаем отрезок $[p; q]$ с отрезком правого сына, т. е. $[m + 1; r]$, и в случае непустого пересечения рекурсивно вызываем себя от вершины $2v + 1$ правого сына.
4. В качестве результата возвращаем минимум из величин, возвращенных рекурсивными вызовами.

Проиллюстрируем этот алгоритм на примере обработки запроса $p = 1$, $q = 4$ (рис. 23.4).

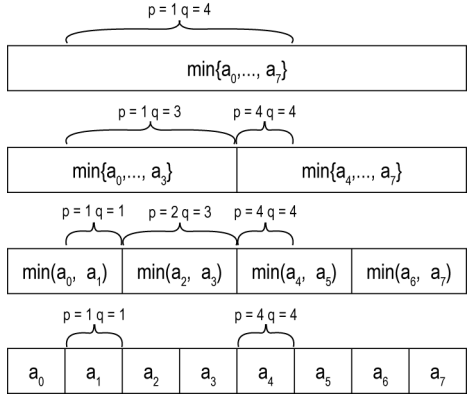

Рис. 23.4. Иллюстрация обработки запроса $p = 1$, $q = 4$ в дереве отрезков для 8 элементов. Исходный запрос при этом сводится к комбинации нескольких запросов, ответы для которых уже хранятся в вершинах дерева

23.1.9. Асимптотика поиска минимума на отрезке

Первое наблюдение: нас интересуют в первую очередь случаи, когда вызов функции порождает не один, а два рекурсивных вызова. Поскольку число уровней в дереве отрезков есть величина $O(\log n)$, то рекурсивные цепочки вызовов, в которых каждый раз совершается по одному вызову, отработают за время $O(\log n)$. Случаи же с двумя рекурсивными вызовами более важны для анализа, поскольку если они слишком часты, асимптотика ухудшится (например, если каждый вызов порождал бы два новых вызова, обработка одного запроса привела бы к посещению всего дерева отрезков и потому линейному времени работы).

Второе наблюдение: если левая граница запроса совпадает с левым концом текущей вершины, т. е. $p = l$, то во всем дереве рекурсивных вызовов на каждый уровень дерева отрезков будет приходиться не более двух вызовов. В самом деле, если $q \le m$, то будет совершен только один рекурсивный вызов, внутри которого левая граница запроса будет снова совпадать с левой границей вершины. Если же $q > m$, то будет произведено два рекурсивных вызова, для $[l; m]$ в левом сыне и для $[m + 1; q]$ в правом сыне. Первый из этих вызовов сразу же завершится, поскольку он совпадает с отрезком левого сына. Во втором же вызове снова будет соблюдаться, что левая граница запроса совпадает с левым концом вершины. Таким образом, мы вернулись к тому же самому состоянию, и поскольку рано или поздно мы дойдем до листа, то по принципу математической индукции мы доказали истинность утверждения. Для наглядности проиллюстрируем оба этих случая (рис. 23.5).

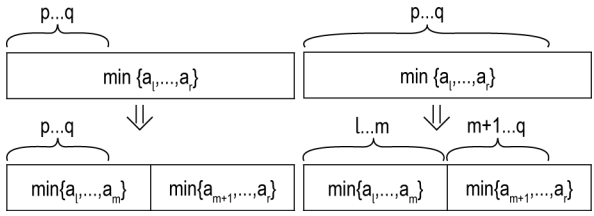

Рис. 23.5. Два возможных варианта в случае, когда левая граница запроса совпадает с левой границей вершины. Слева показан вариант, при котором мы переходим в левого сына с тем же запросом. Справа изображен вариант, при котором мы извлекаем ответ из левого сына и переходим в правого сына. В обоих вариантах возникает не более двух рекурсивных вызовов, и аналогичные рассуждения верны для последующих уровней рекурсии

Третье наблюдение: то же самое верно и для правых границ: при условии если $q = r$, на каждом уровне дерева отрезков будет не более двух рекурсивных вызовов.

Четвертое наблюдение: если в какой-то вершине совершается два рекурсивных вызова, то их границы будут совпадать с правым и левом концом, соответственно, левого и правого сыновей. Графическое изображение этого факта показано на рис. 23.6.

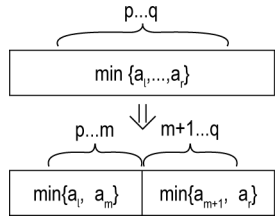

Рис. 23.6. Обработка запроса, затрагивающего левого и правого сыновей, приводит к двум рекурсивным вызовам, причем запрос в левом сыне будет оканчиваться на его правом конце, а запрос в правом сыне будет оканчиваться на его левом конце

Итог: если рассмотреть дерево рекурсивных вызовов, порождаемое обработкой запроса, то на каждый уровень дерева отрезков будет приходиться не более четырех вызовов. Поскольку высота дерева отрезков — порядка логарифма, то и асимптотика обработки одного запроса составляет $O(\log n)$.

23.1.10. Реализация поиска минимума на отрезке

Небольшие соображения по реализации:

- поскольку концы l и r каждой вершины не хранятся явным образом, будем передавать их функции с помощью аргументов;
- для нахождения ответа на запрос требуется вызвать функцию в корневой вершине, т. е. с аргументами $v = 1$, $l = 0$, $r = n - 1$;
- при реализации удобно разрешить вызывать функцию с пустым отрезком, т. е. с аргументами $p > q$, поскольку это позволяет избежать лишних проверок при обработке каждого из сыновей. Для пустого отрезка мы возвращаем значение «бесконечность», которое должно быть константой, заведомо превосходящей всевозможные значения элементов a_i. В языке Python можно использовать вещественное «бесконечно большое» значение; в языке C++ мы используем фиксированную константу 10^9 (разумеется, в зависимости от задачи при необходимости ее нужно будет увеличить)

Python

```
INF = float('inf')

def query(v, l, r, p, q):
  if p > q:
    return INF
  if l == p and r == q:
    return t[v]
  mid = (l + r) // 2
  ans_l = query(
      v * 2, l, mid,
      p, min(mid, q))
```

C++

```
const int INF = (int)1E9;

int Query(int v, int l, int r,
          int p, int q) {
  if (p > q)
    return INF;
  if (l == p && r == q)
    return t[v];
  int mid = (l + r) / 2;
  int ans_l = Query(
      v * 2, l, mid,
```

```
  ans_r = query(
      v * 2 + 1, mid + 1, r,
      max(mid + 1, p), q)
  return min(ans_l, ans_r)

# ...
print(query(1, 0, n - 1, p, q))
```

```
      p, min(mid, q));
  int ans_r = Query(
      v * 2 + 1, mid + 1, r,
      max(mid + 1, p), q);
  return min(ans_l, ans_r);
}

int main() {
  // ...
  cout << Query(1, 0, n - 1, p, q)
       << endl;
}
```

Нерекурсивные реализации дерева отрезков

Существуют также и нерекурсивные реализации дерева отрезков. Как правило, они основаны на несколько иной нумерации вершин, которая позволяет простой формулой получить индекс вершины-листа, а затем подниматься от него вверх по дереву. Такие реализации часто содержат меньше кода и работают с меньшей скрытой константой. Однако рекурсивные реализации проще в понимании и лучше поддаются обобщениям при решении более сложных задач, поэтому в книге мы будем использовать только их.

23.2. Дерево отрезков для максимумов

Описанный выше вариант дерева отрезков позволяет искать минимум на отрезке. Однако эту же структуру данных можно использовать и для многих других видов запросов.

Очевидно, операцию взятия минимума можно заменить на максимум, и корректность и асимптотика алгоритма от этого не поменяются.

С точки зрения реализации изменится всего пара строчек.

Python

```
def build(v, l, r):
  # ...
  t[v] = max(t[v * 2],
             t[v * 2 + 1])

def query(v, l, r, p, q):
  if p > q:
    return -INF
  # ...
  return max(ans_l, ans_r)
```

C++

```
void Build(int v, int l, int r) {
  // ...
  t[v] = max(t[v * 2],
             t[v * 2 + 1]);
}

int Query(int v, int l, int r,
          int p, int q) {
  if (p > q)
    return -INF;
  // ...
  return max(ans_l, ans_r);
}
```

23.3. Дерево отрезков с запросами модификации

До сих пор мы рассматривали ситуации, когда исходный массив a_i статический. Однако дерево отрезков легко обобщить и на случай задач, в которых помимо «информационных» запросов есть также и запросы вида «присвоить элементу a_u значение w».

Для обработки запросов такого вида заметим, что изменившийся элемент влияет только на $O(\log n)$ вершин дерева: по одной вершине на каждом уровне. Следовательно, операцию модификации можно реализовать за время $O(\log n)$.

Реализация снова будет основана на рекурсивной функции. Функция будет определять, какому сыну (левому или правому) принадлежит изменяемый элемент, и вызывать себя рекурсивно от него; после возврата рекурсивного вызова функция будет пересчитывать значение в текущей ячейке согласно значениям в дочерних вершинах (для которых рекурсия уже выставила новые значения). При достижении вершины-листа функция будет просто записывать в вершину новое значение.

Для определенности, приведем пример реализации в случае дерева отрезков для минимумов.

Python

```
def update(v, l, r, pos, new_val):
  if l == r:
    t[v] = new_val
    return
  mid = (l + r) // 2
  if pos <= mid:
    update(v * 2, l, mid,
           pos, new_val)
  else:
    update(v * 2 + 1, mid + 1, r,
           pos, new_val)
  t[v] = min(t[v * 2],
             t[v * 2 + 1])

# ...
update(1, 0, n - 1, pos, new_val)
```

C++

```
void Update(int v, int l, int r,
            int pos, int new_val) {
  if (l == r) {
    t[v] = new_val;
    return;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    Update(v * 2, l, mid,
           pos, new_val);
  } else {
    Update(v * 2 + 1, mid + 1, r,
           pos, new_val);
  }
  t[v] = min(t[v * 2],
             t[v * 2 + 1]);
}

int main() {
  // ...
  Update(1, 0, n - 1, pos,
         new_val);
}
```

23.4. Дерево отрезков для сумм

Дерево отрезков несложно адаптировать для ответа на запросы о суммах на отрезке: для этого достаточно хранить в каждой вершине дерева сумму на отрезке, соответствующем этой вершине.

Запросы модификации элементов можно обрабатывать тем же самым методом, что и вышеописанный алгоритм для деревьев минимумов.

> Напомним, что если запросы модификации отсутствуют, то эту задачу можно решить и проще, предварительно подсчитав частичные суммы для массива.

Асимптотика будет по-прежнему $O(n)$ для построения структуры данных и $O(\log n)$ для ответа на запрос или на модификацию одного элемента.

Реализация совпадает с реализацией дерева для минимумов за исключением нескольких строк.

Python

```
def build(v, l, r):
  # ...
  t[v] = t[v * 2] + t[v * 2 + 1]

def query(v, l, r, p, q):
  if p > q:
    return 0
  # ...
  return ans_l + ans_r;

def update(v, l, r, pos, new_val):
  # ...
  t[v] = t[v * 2] + t[v * 2 + 1]
```

C++

```
void Build(int v, int l, int r) {
  // ...
  t[v] = t[v * 2] + t[v * 2 + 1];
}

int Query(int v, int l, int r,
          int p, int q) {
  if (p > q)
    return 0;
  // ...
  return ans_l + ans_r;
}

void Update(int v, int l, int r,
            int pos, int new_val) {
  // ...
  t[v] = t[v * 2] + t[v * 2 + 1];
}
```

23.5. Прочие виды операций в дереве отрезков

Можно заметить, что приведенные выше реализации дерева отрезков для минимумов, максимумов, сумм не использовали никаких особых свойств этих операций. На самом деле, единственное требование, накладываемое деревом отрезков на операцию — это ее **ассоциативность**, т. е. независимость значения операции от расставленных скобок:

$$(x * y) * z = x * (y * z).$$

Если операция удовлетворяет этому свойству, то дерево отрезков позволяет вычислить ее результат на любом подотрезке за время $O(\log n)$, поддерживая также запросы модификации за время $O(\log n)$.

Это открывает простор для решения самых разнообразных задач; приведем несколько **примеров** запросов на отрезке.

- Подсчет вхождений некоторого фиксированного элемента (например, нуля).
- Подсчет четных или нечетных элементов.
- Подсчет вхождений минимума или максимума.
  - Для этого можно хранить в каждой вершине дерева минимум/максимум и число его вхождений.
- Произведение элементов (по модулю).
- Наибольший общий делитель или наименьшее общее кратное.
- Поиск подотрезка с максимальной суммой.
  - Для этого в каждой вершине можно хранить четыре величины: сумму, наибольший префикс, наибольший суффикс, наибольший подотрезок.
- Поиск наидлиннейшей правильной скобочной последовательности (в строке, содержащей только скобки одного типа).
  - Для этого в каждой вершине будет хранить ответ на ее подотрезке, число непарных открывающих скобок и число непарных закрывающих скобок.
- Матричное произведение (для случая, когда массив состоит из матриц, а не чисел).

… и т. п.

23.6. Запросы обновления на отрезке

До сих пор мы рассматривали только задачи, в которых запросы модификации либо отсутствуют, либо точечные (меняющие один элемент). Однако дерево отрезков позволяет эффективно реализовать и такие виды модификаций, как присвоение на отрезке или прибавление на отрезке.

23.6.1. Прибавление на отрезке

Дан массив чисел $a[i]$ $(i = 0...n - 1)$, и требуется выполнять запросы двух типов:

- запрос прибавления на отрезке: по заданным (p, q, u) увеличить все элементы $a[p...q]$ на u;
- запрос чтения: по заданному w вернуть значение $a[w]$.

Чтобы эффективно обрабатывать запросы прибавления на отрезке, будем хранить в каждой вершине дерева, **сколько следует прибавить** ко всем соответствующим этой вершине элементам.

Например, если $t[1] = 5$, а $t[3] = 10$, то это значит, что к первой половине массива надо прибавить 5, а ко второй — 15.

Изначально во всех вершинах дерева записаны нули. При обработке очередного запроса прибавления будем спускаться по дереву, как обычно, начиная с вершины №1. Если в какой-то момент границы текущего запроса совпали с концами текущей вершины v, то увеличим $t[v]$ на u. Асимптотика времени работы будет такая же, как и у рассмотренных ранее версий дерева отрезков: $O(\log n)$ на запрос.

На запрос чтения также будем отвечать рекурсивным обходом дерева. Мы спускаемся, начиная с вершины №1 и заканчивая листом, соответствующим позиции w. По пути мы должны суммировать значения $t[v]$ всех пройденных вершин дерева, и ответом на запрос будет сумма их и исходного $a[w]$. Асимптотика здесь будет также $O(\log n)$ на каждый запрос.

Реализация

Python

```
def update(v, l, r, p, q, add):
  if p > q:
    return
  if l == p and r == q:
    t[v] += add
    return
  mid = (l + r) // 2
  update(v * 2, l, mid,
         p, min(mid, q), add)
  update(v * 2 + 1, mid + 1, r,
         max(p, mid + 1), q, add)

def query(v, l, r, pos):
  if l == r:
    return t[v]
  mid = (l + r) // 2
  if pos <= mid:
    return query(v * 2, l, mid,
                 pos) + t[v]
  return query(v * 2 + 1, mid + 1,
               r, pos) + t[v]

t = [0] * (n * 4)
# ...
update(1, 0, n - 1, p, q, u)
# ...
print(a[w] + query(1, 0, n - 1, w))
```

C++

```
#include <vector>
using namespace std;

vector<int> t;

void Update(
    int v, int l, int r, int p,
    int q, int add) {
  if (p > q)
    return;
  if (l == p && r == q) {
    t[v] += add;
    return;
  }
  int mid = (l + r) / 2;
  Update(v * 2, l, mid, p,
         min(mid, q), add);
  Update(v * 2 + 1, mid + 1, r,
         max(p, mid + 1), q, add);
}

int Query(int v, int l, int r,
          int pos) {
  if (l == r)
    return t[v];
  int mid = (l + r) / 2;
  if (pos <= mid) {
    return Query(v * 2, l, mid,
                 pos) + t[v];
  }
```

```
  return Query(v * 2 + 1, mid + 1,
               r, pos) + t[v];
}

int main() {
  // ...
  t.resize(n * 4);
  // ...
  Update(1, 0, n - 1, p, q, u);
  // ...
  cout << Query(1, 0, n - 1, w) +
          a[w] << endl;
}
```

23.6.2. Покраска на отрезке. Отложенные операции

Задача заключается в том, чтобы по изначально нулевому массиву $a[i]$ $(i = 0...n - 1)$ научиться эффективно обрабатывать запросы двух видов:

- запрос покраски: по заданным (p, q, u) присвоить всем элементам $a[p...q]$ значение $u > 0$;
- запрос чтения: по заданному w вернуть значение $a[w]$.

Для решения этой задачи будем использовать ту же самую иерархическую структуру дерева отрезков, только теперь в каждой вершине будем хранить цвет, в который она покрашена. Цвет вершины означает, что все элементы соответствующего отрезка покрашены в этот цвет. Изначально все вершины дерева не окрашены, т. е. имеют цвет, равный нулю.

Начнем с простого — с **запроса чтения** элемента $a[w]$. Для ответа на этот запрос будем спускаться по дереву из вершины №1 и искать первую вершину, которая имеет ненулевой цвет.

Обработку **запроса покраски** на отрезке $[p; q]$ будем производить, как обычно, рекурсивным вызовом, спускающимся по дереву из вершины №1. У нас есть два случая:

- если запрос требует покрасить всю вершину, т. е. границы запроса совпадают с границами вершины, то просто запишем новый цвет в вершину дерева $t[v]$;
- если же запрос требует покрасить лишь какой-то подотрезок текущей вершины, то нам потребуется совершить рекурсивные вызовы из дочерних вершин. Однако:
  - если текущая вершина покрашена в какой-то цвет, то было бы ошибкой оставить его без изменений: после обработки текущего запроса соответствующий текущей вершине подотрезок, вообще говоря, перестанет быть одноцветным;
  - для решения этой проблемы снимем с текущей вершины цвет, если он там был, и запишем его в левую и в правую дочерние вершины. С концептуальной точки зрения это является **«отложенным» обновлением**;
  - только после этого мы произведем обычные для дерева отрезков рекурсивные вызовы из левой и правой дочерних вершин.

Подведем **итог**. Мы смогли интегрировать обработку запросов покраски и чтения в стандартные алгоритмы дерева отрезков — следовательно, мы можем утверждать, что это решение имеет асимптотику $O(\log n)$ на один запрос. Ключом к решению стали отложенные обновления, т. е. сохранение в вершинах дерева еще не оконченных операций покраски и постепенное «проталкивание» их вниз по дереву во время обработки последующих запросов.

Реализация

Python

```
def paint(v, l, r, p, q, color):
  if p > q:
    return
  if l == p and r == q:
    t[v] = color
    return
  if t[v] != 0:
    t[v * 2] = t[v]
    t[v * 2 + 1] = t[v]
    t[v] = 0
  mid = (l + r) // 2
  paint(v * 2, l, mid,
        p, min(mid, q), color)
  paint(v * 2 + 1, mid + 1, r,
        max(p, mid + 1), q, color)

def query(v, l, r, pos):
  if l == r or t[v] != 0:
    return t[v]
  mid = (l + r) // 2
  if pos <= mid:
    return query(v * 2, l, mid,
                 pos)
  return query(v * 2 + 1, mid + 1,
               r, pos)

# ...
t = [0] * (n * 4)
# ...
paint(1, 0, n - 1, p, q, u)
# ...
print(query(1, 0, n - 1, w))
```

C++

```
#include <vector>
using namespace std;

vector<int> t;

void Paint(
    int v, int l, int r, int p,
    int q, int color) {
  if (p > q)
    return;
  if (l == p && r == q) {
    t[v] = color;
    return;
  }
  if (t[v] != 0) {
    t[v * 2] = t[v];
    t[v * 2 + 1] = t[v];
    t[v] = 0;
    return;
  }
  int mid = (l + r) / 2;
  Paint(v * 2, l, mid, p,
        min(mid, q), color);
  Paint(v * 2 + 1, mid + 1, r,
        max(p, mid + 1), q, color);
}

int Query(int v, int l, int r,
          int pos) {
  if (l == r || t[v] != 0)
    return t[v];
  int mid = (l + r) / 2;
  if (pos <= mid) {
    return Query(v * 2, l, mid,
                 pos);
  }
```

```
  return Query(v * 2 + 1, mid + 1,
               r, pos);
}

int main() {
  // ...
  t.resize(n * 4);
  // ...
  Paint(1, 0, n - 1, p, q, u);
  // ...
  cout << Query(1, 0, n - 1, w)
       << endl;
}
```

23.6.3. Прочие виды модификаций на отрезке

Комбинируя различные виды модификаций на отрезке (покраска, прибавление и т. п.) с разными видами операций (сумма, минимум, максимум и т. д.), можно получить множество вариантов дерева отрезков. До тех пор пока эти операции работают за время $O(1)$ и способ обхода дерева отрезков остается стандартным, все эти варианты будут иметь одну и ту же асимптотику $O(\log n)$ на запрос.

Как правило, сложности с модификациями на отрезке возникают чисто реализационные: правильно запрограммировать отложенное обновление, не забыть вызвать его при спуске по дереву, правильно учесть метки отложенных обновлений при подъеме по дереву.

23.7. Дальнейшие обобщения дерева отрезков

Дерево отрезков допускает и такие интересные обобщения, при которых каждая вершина дерева хранит в себе не одно значение, а **вложенную структуру данных**. Подробное описание таких сложных вариаций выходит за рамки данной книги, поэтому мы лишь кратко наметим пути этих обобщений.

- Можно хранить в каждой вершине отсортированный список всех чисел, встречающихся на соответствующем отрезке.
  - Суммарный объем этих списков будет $O(n \log n)$.
  - С помощью такого дерева можно будет, например, отвечать за время $O(\log^2 n)$ на запрос вида «найти наименьший элемент, больший u, среди элементов $a[p...q]$».
- Если входные данные представляют собой матрицу, то по ней можно построить двумерное дерево отрезков — т. е. дерево, в каждой вершине которого содержится другое дерево.
  - Суммарный объем памяти будет линеен относительно размера матрицы.

- Если понимать внешнее дерево отрезков как разбивающее матрицу по одной размерности (строки), то каждое из внутренних деревьев отрезков будет содержать информацию о ячейках в «полосе», образованной подотрезком строк и всеми столбцами.
- С помощью такого двумерного дерева можно, например, отвечать на запросы суммы на подпрямоугольниках и обрабатывать запросы модификации элементов.

Другое направление обобщений — это добавление в структуру данных **персистентности**.

> Персистентная структура данных — такая структура, которая компактно хранит всю историю своих модификаций и позволяет эффективно отвечать на запросы в любой указанной версии.

23.8. Пример решения задачи. Наидлиннейшая возрастающая подпоследовательность (быстрый вариант)

Примечание. Это та же самая задача, что была рассмотрена ранее в *разд. 16.2*, только с бóльшими ограничениями.

Задача. Дана последовательность a_i целых чисел длины n. Требуется найти ее наидлиннейшую возрастающую подпоследовательность.

Входные данные содержат число n в первой строке и n чисел a_i во второй. Ограничения: $1 \le n \le 5 \cdot 10^5$, $|a_i| < 10^9$.

Вывести требуется одно число — длину искомой наидлиннейшей возрастающей последовательности.

Пример

Входные данные	Требуемый результат
6 1 4 2 3 6 5	4

Решение

Напомним описанное ранее решение на основе динамического программирования:

$$d_n = \max_{k<n,\ a_k<a_n} d_k + 1.$$

Это решение работает за время $O(n^2)$, поскольку в динамике есть n состояний и каждое вычисляется просмотром всех предыдущих значений и поиском максимума среди подходящих. Однако теперь мы можем ускорить этот шаг с помощью **дерева отрезков**.

Сведем вычисление динамики к запросам максимума на отрезке, поскольку стандартное дерево отрезков не умеет отвечать на запрос «максимум среди всех значений, индекс которых меньше заданного, и значение в массиве a_i меньше заданного».

Во-первых, заметим, что критерий «$k < n$» на самом деле не требуется проверять явно, если мы будем вычислять значения динамики по очереди в порядке возрастания.

Во-вторых, мы можем строить дерево отрезков таким образом, что индексами в нем будут числа a_k, а значения — величины d_k. Иными словами, если мы определим массив b_j как

$$b_{a_j} = d_j,$$

то вычисление очередного значения d_n динамики сведется к запросу максимума в массиве b на отрезке

$$[-\infty; a_n - 1].$$

Это пока еще не дает окончательного решения задачи, поскольку дерево отрезков не умеет работать с такими большими индексами, как в нашей задаче — от -10^9 до 10^9. Решением этой проблемы является техника «**сжатия**» входного массива: мы можем заменить каждое число a_i его индексом в отсортированном массиве всех чисел. Ответ для массива «сжатых» чисел будет таким, и поверх «сжатых» значений уже можно строить стандартное дерево отрезков для максимума.

Асимптотика решения — $O(n \log n)$. Она складывается из времени «сжатия» входного массива, которое легко реализовать за время $O(n \log n)$ с помощью сортировки и двоичного поиска, и суммарного времени вычисления всех значений d, которое составляет n раз по $O(\log n)$.

23.9. Пример решения задачи. Наименьший общий предок

Определение. Наименьшим общим предком вершин u и v в заданном подвешенном (корневом) дереве является вершина, которая является и предком u, и предком v, и при этом наиболее удалена от корня.

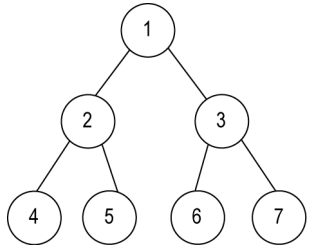

Рис. 23.7. Пример дерева с 7 вершинами

Например, в изображенном дереве наименьшим общим предком вершин №4 и 5 является вершина №2, а наименьшим общим предком вершин №3 и 4 — вершина №1 (рис. 23.7).

> Задача поиска наименьшего общего предка — классическая задача для теории графов и структур данных. По-английски она называется lowest common ancestor или least common ancestor; часто можно встретить ее сокращенное обозначение — LCA.

Задача. Дано дерево из n вершин. Требуется ответить на m запросов вида «найти наименьшего общего предка вершин a_i и b_i».

Входные данные содержат числа n и m в первой строке ($1 \le n, m \le 5 \cdot 10^5$). Вторая строка содержит описание дерева в виде $n - 1$ чисел p_i ($i = 2...n$), обозначающих, что для вершины с номером i родителем является вершина с номером p_i ($1 \le p_i \le n$). Следующие m строк содержат запросы: по паре чисел a_i и b_i в каждой.

Вывести требуется m строк по одной для каждого запроса: номер вершины искомого наименьшего общего предка.

Пример

Входные данные	Требуемый результат
7 4 1 1 2 2 3 3 4 5 3 4 3 6 1 1	2 1 3 1

Решение

Понятно, что тривиальное решение — подниматься от обеих вершин запроса вверх и искать общий элемент — работало бы за время $O(n)$ на один запрос в худшем случае, поскольку дерево может иметь большую глубину. Такое решение работало бы слишком долго при заданных ограничениях.

Есть несколько различных подходов, которые можно использовать для решения этой задачи, однако здесь мы сведем эту задачу к той, которую мы уже умеем решать с помощью дерева отрезков: а именно, к задаче **минимума на отрезке**.

Для этого нам понадобится предварительно подсчитать **глубину** каждой вершины и **порядок** посещений вершин. Нам нужно, чтобы в получившемся порядке посещений каждая вершина сначала упоминалась тогда, когда обход в глубину заходит в нее, а также после каждого возврата из рекурсивного вызова.

Например, для использованного в примере из условия дерева порядок посещений будет выглядеть следующим (рис. 23.8) образом (при условии, что мы посещаем дочерние вершины в порядке возрастания их номеров).

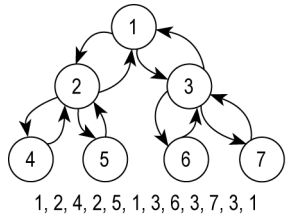

Рис. 23.8. Пример обхода дерева и получающийся при этом порядок посещений вершин

И глубины вершин, и порядок посещений можно построить с помощью обхода в глубину. Порядок посещений будет содержать лишь $O(n)$ элементов (если быть точным, то $2n - 1$, поскольку число дополнительных упоминаний всех вершин равно в сумме числу ребер в дереве, которое, как известно, равно $n - 1$). Предварительно подсчитаем также позиции первого вхождения каждой вершины в список посещений — обозначим это через q_i.

Тогда утверждается, что для ответа на запрос LCA вершин a и b нужно получить позиции q_a и q_b, рассмотреть подотрезок списка посещений между этими позициями и **найти в этом подотрезке вершину с наименьшей глубиной**.

Например, для изображенного выше графа для поиска LCA вершин №4 и 5 соответствующий подотрезок списка посещений выглядит как «4, 2, 5», и наименьшую глубину в этом подотрезке имеет вершина №2. Для LCA вершин №3 и 4 соответствующий подотрезок — «4, 2, 5, 2, 1, 3», и наименьшую глубину в нем имеет вершина №1.

Для доказательства этого утверждения заметим, что относительно своего LCA вершины a и b являются либо ее потомками, либо совпадают с ней, откуда следует, что сама LCA всегда будет присутствовать в списке посещений между вхождениями a и b, и при этом никакого из предков LCA там не будет.

Из этого утверждения следует **алгоритм** решения задачи.

- Препроцессинг.
  - Совершить обход в глубину, который сохранит высоту h_i каждой вершины и построит список посещений t.
  - Для каждой вершины сохранить в q_i позицию ее первого вхождения в t.
  - Построить дерево отрезков над массивом t. Дерево отрезков должно поддерживать запросы поиска минимума и его позиции, причем сравнение элементов производится по массиву высот h.
- Обработка запроса.
  - Получить для вершин запроса a и b соответствующие им индексы q_a и q_b.
  - Сделать запрос к дереву отрезков для поиска минимума на отрезке $[\min(q_a, q_b); \max(q_a, q_b)]$. Вернуть результат в качестве ответа.

Это решение имеет асимптотику $O(\log n)$ для ответа на один запрос с препроцессингом за время $O(n)$.

Глава 24.
Задачи для самостоятельного решения

Для закрепления пройденного материала настоятельно рекомендуем читателю самостоятельно прорешать задачи этой главы.

24.1. Примеры задач

Подсказки к приведенным ниже задачам даны в *приложении*.

24.1.1. Задача. Поиск подпрямоугольника с наибольшей суммой

Дана матрица. Требуется найти ее подпрямоугольник с наибольшей суммой.

Входные данные состоят из размеров матрицы n и m в первой строке и самой матрицы в виде n строк по m чисел в каждой. Ограничения: $1 \le n, m \le 500$, элементы матрицы не превосходят по модулю 10^9.

Вывести требуется искомую сумму, а также границы соответствующего прямоугольника: номер первой строки, номер первого столбца, номер последней строки и номер последнего столбца. Строки и столбцы нумеруются, начиная с единицы.

Примеры

Входные данные	**Требуемый результат**
2 3 1 2 -1 4 -10 8	7 1 3 2 3
Входные данные	**Требуемый результат**
1 1 -1	-1 1 1 1 1

24.1.2. Задача. Пересадки в метро

Дана карта метро: k линий, где i-я линия соединяет станции $a_{i,1}$, $a_{i,2}$, …, a_{i,b_i} в этой последовательности. Все линии двунаправленные. Если через какую-либо станцию проходят несколько линий метро, то можно совершить пересадку. Требуется найти маршрут от станции x до станции y с наименьшим числом пересадок.

Входные данные содержат числа k, x, y в первой строке и описания линий метро в последующих k строках. Каждая строка содержит число b_i, за которым следуют числа $a_{i,1}$, $a_{i,2}$, ..., a_{i,b_i}. Ограничения: $1 \le k \le 10^3$, номера станций являются натуральными числами, не превосходящими 10^5, и сумма всех b_i не превосходит 10^5.

Вывести требуется наименьшее число пересадок на маршруте из x в y, либо «−1», если такого маршрута нет.

Примеры

Входные данные	**Требуемый результат**
3 1 5 3 1 2 3 3 4 2 5 4 5 4 3 2	1
Входные данные	**Требуемый результат**
1 10 20 4 1 2 3 4	-1

24.1.3. Задача. Основание столицы

На определенном этапе развития протогосударства X одно из его поселений становится столицей. Конечно, из столицы должны существовать пути до всех остальных поселений протогосударства. Некоторые поселения уже связаны маршрутами; какое наименьшее число маршрутов требуется добавить, чтобы из столицы можно было добраться до всех поселений?

В первой строке входных данных содержатся число поселений n и число имеющихся маршрутов m. Последующие m строк содержат по паре чисел a_i и b_i, обозначающих наличие **однонаправленного** маршрута из поселения a_i в поселение b_i (поселения нумеруются, начиная с единицы). Столица имеет номер «1». Ограничения: $1 \le n \le 10^5$, $0 \le m \le 10^5$.

Вывести требуется одно число — минимальное количество дополнительных маршрутов.

Примеры

Входные данные	**Требуемый результат**
4 3 2 3 3 4 4 2	1
Входные данные	**Требуемый результат**
3 0	2

Пояснения к примерам

В первом примере достаточно добавить маршрут из столицы (поселения №1) в любое другое поселение, например №2. Во втором примере требуется добавить маршруты из столицы в каждое из остальных поселений.

24.1.4. Задача. Поиск наибольшего нулевого подпрямоугольника

Дана матрица, состоящая из нулей и единиц. Требуется найти наибольший ее подпрямоугольник, состоящий только из нулей.

В первой строке входных данных записаны размеры матрицы n и m ($1 \le n, m \le 1000$), последующие n строк содержат описание матрицы: строки длины m каждая, состоящие только из нулей и единиц. Гарантируется, что матрица содержит хотя бы один нуль.

Вывести требуется границы наибольшего по площади нулевого подпрямоугольника: номер первой строки, номер первого столбца, номер последней строки, номер последнего столбца. Строки и столбцы нумеруются, начиная с единицы. Если решений несколько, разрешается вывести любое.

Примеры

Входные данные	**Требуемый результат**
3 4 0101 1000 1001	2 2 3 3
Входные данные	**Требуемый результат**
1 1 0	1 1 1 1

24.1.5. Задача. Перемножение по большому модулю

Даны два целых числа a и b и модуль p. Требуется вывести результат произведения a и b по модулю p. Все три числа неотрицательны и не превосходят 10^{18}; модуль p не меньше 2.

Примеры

Входные данные	**Требуемый результат**
2 3 5	4
Входные данные	**Требуемый результат**
123 0 6	0

24.1.6. Задача. Подсчет числа путей с петлями и кратными ребрами

По заданному неориентированному графу посчитать для каждой пары вершин количество различных путей заданной длины k между этими двумя вершинами, взятое по указанному модулю p. В графе могут присутствовать петли и кратные ребра.

Первая строка входных данных содержит число вершин, число ребер, длину k и модуль p. Последующие строки содержат описания ребер, где каждая строка содержит номера вершин-концов ребра (в 1-индексации). Гарантируется, что число вершин не превосходит 100, число ребер — 10^4, число k находится в отрезке $[0; 10^9]$, модуль p — в отрезке $[2; 10^9]$.

Вывести требуется матрицу из $n \times n$ чисел, содержащую искомые количества.

Примеры

Входные данные	**Требуемый результат**
2 2 2 10 1 1 1 2	2 1 1 1
Входные данные	**Требуемый результат**
2 2 2 100 2 2 2 2	0 0 0 4

24.1.7. Задача. Подсчет числа путей с длиной, не превосходящей заданную

По заданному неориентированному графу посчитать для каждой пары вершин количество различных путей между этими двумя вершинами, имеющих длину **не более чем** k. Количества требуется найти по заданному модулю p. В графе могут присутствовать петли и кратные ребра.

Первая строка входных данных содержит число вершин, число ребер, длину k и модуль p. Последующие строки содержат описания ребер, где каждая строка содержит номера вершин-концов ребра (в 1-индексации). Гарантируется, что число вершин не превосходит 100, число ребер — 10^4, число k находится в отрезке $[0; 10^9]$, а модуль p — в отрезке $[2; 10^9]$.

Вывести требуется матрицу из $n \times n$ чисел, содержащую искомые количества.

Примеры

Входные данные	**Требуемый результат**
3 2 3 10 1 2 2 3	2 3 1 3 3 3 1 3 2

Входные данные	Требуемый результат
2 2 2 100 2 2 2 2	1 0 0 7

24.1.8. Задача. Поиск самого дешевого пути с заданным числом ребер

Дан неориентированный граф, каждому ребру которого приписана стоимость, и число k. Требуется для каждой пары вершин (i, j) найти самый дешевый путь, ведущий из вершины i в вершину j и содержащий ровно k ребер. Путь может проходить по ребрам более одного раза; в этом случае учитывается каждое вхождение ребра.

Первая строка входных данных содержит число вершин, число ребер, длину k. Последующие строки содержат описания ребер, где каждая строка содержит номера вершин-концов ребра (в 1-индексации) и число — стоимость прохождения по этому ребру. Гарантируется, что число вершин не превосходит 100, число ребер — 10^4, стоимости ребер и число k находятся в отрезке $[0; 10^9]$.

Вывести требуется матрицу из $n \times n$ чисел, содержащую искомые длины. В случае когда между парой вершиной отсутствует требуемый путь, выводить в соответствующей ячейке следует число «−1».

Примеры

Входные данные	Требуемый результат
4 4 2 1 2 4 2 3 3 3 4 2 4 1 1	2 -1 3 -1 -1 6 -1 5 3 -1 4 -1 -1 5 -1 2
Входные данные	**Требуемый результат**
2 2 2 100 2 2 2 2	1 0 0 7

24.2. Задачи в онлайн-системах

- UVA, задача 1230: «MODEX».
- Codeforces, раунд 511, задача A: «Увеличить НОД».

- SPOJ, задача LASTDIG: «The last digit».
- Codeforces, раунд 590, задача D: «Запросы на различные символы».
- SPOJ, задача SUPPER: «Supernumbers in a permutation».
- SPOJ, задача LOCKER: «Magic of the locker».
- Codeforces, раунд 10, задача D: «НОВП».
- UVA, задача 11029: «Leading and Trailing».
- Codeforces, раунд 250, задача D: «Ребенок и последовательность».
- Codeforces, раунд 380, задача C: «Сережа и скобочки».
- Codeforces, раунд 197, задача D: «Ксюша и битовые операции».
- UVA, задача 11402: «Ahoy, Pirates!».
- Codeforces, раунд 745, задача A: «CQXYM считает перестановки».
- SPOJ, задача KQUERY: «K-query».
- SPOJ, задача KQUERYO: «K-Query Online».
- Codeforces, раунд 675, задача F: «Скучные запросы».
- Codeforces, раунд 813, задача E: «Создание армии».
- SPOJ, задача MKTHNUM: «K-th Number».

Ступень IV.
Разносторонняя подготовка

Глава 25. Производительность ввода-вывода

К сожалению, стандартные средства ввода-вывода во многих языках программирования и стандартных библиотеках недостаточно оптимизированы по скорости.

В контексте спортивного программирования это может приводить к тому, что решение правильным алгоритмом оказывается незачтенным по времени (time limit exceeded) из-за того, что чтение входных данных или вывод результата заняли слишком много времени.

> Несмотря на то что авторы задач обычно стараются избежать использования чрезмерно объемных входных и выходных данных, задачи, например, требующие считать или вывести сотни тысяч чисел, — не редкость. У этого есть объективная причина: если у задачи есть тривиальный алгоритм решения с асимптотикой $O(n^2)$, а предполагаемое автором задачи решение имеет асимптотику $O(n)$ или $O(n \log n)$, то граница, когда типичное квадратичное решение начинает работать кардинально медленнее (как минимум, несколько секунд) — находится как раз в области $10^5...10^6$ элементов.

В этой главе мы приведем эмпирические данные о скорости ввода-вывода, некоторые типичные проблемы ввода-вывода и приемы их решения.

25.1. Производительность ввода-вывода в Python

Сразу оговоримся: язык Python, особенно при использовании только его стандартной библиотеки, является не самым подходящим инструментом для работы с объемными входными или выходными данными. Полезно, однако, иметь представление о том, какой именно является производительность стандартных средств ввода-вывода в нем.

25.1.1. Эмпирические показатели ввода-вывода в Python

Разумеется, приведенные ниже числа — лишь начальные ориентиры, поскольку они зависят от производительности компьютера и версии интерпретатора.

Операция	**Размер входных данных**	**Время работы**
Чтение массива 9-значных целых чисел, записанных в одну строку (реализация с помощью вызовов `input()`, `split()`, `map()`, `int()`)	$n = 10^5$ (1 МиБ)	0,1 с
	$n = 10^6$ (10 МиБ)	0,3 с

(окончание)

Операция	Размер входных данных	Время работы
Чтение массива 9-значных целых чисел, записанных в отдельных строках (реализация с помощью вызовов `input()`, `int()`)	$n = 10^5$ (1 МиБ)	0,2 с
	$n = 10^6$ (10 МиБ)	1,5 с
Вывод массива 9-значных целых чисел в одну строку (реализация с помощью вызова `print()`)	$n = 10^5$ (1 МиБ)	0,1 с
	$n = 10^6$ (10 МиБ)	0,2 с
Вывод массива 9-значных целых чисел по одному в строке (реализация с помощью вызова `print()` в цикле)	$n = 10^5$ (1 МиБ)	0,1 с
	$n = 10^6$ (10 МиБ)	0,3 с

Эти результаты показывают, что, уже начиная со ста тысяч элементов, ввод составляет заметное время. Также можно заметить, что чтение данных с большим количеством строк медленнее, чем чтение единственной строки.

Вывод данных, как видно из результатов, имеет меньшие проблемы с производительностью, хотя на размерах порядка миллиона элементов он также начинает занимать заметное время.

25.1.2. Ускорение ввода в Python с помощью *BytesIO*

Класс `BytesIO` из пакета io позволяет ускорить операции чтения. Для этого потребуется заранее считать все входные данные в виде последовательности байтов и затем создать объект `BytesIO` из этих байтов. После этого вызовами метода `readline()` можно извлекать по одной строке входных данных, получив тем самым эквивалент метода `input()`.

Практика показывает, что такой подход позволяет ускорить чтение входных данных, особенно для данных с большим числом строк (до 7 и более раз!).

Приведем пример реализации этого метода для чтения массива целых чисел.

Стандартная реализация

```
n = int(input())
a = [int(input())
     for _ in range(n)]
```

Оптимизированный вариант

```
from io import *
from os import *
instream = BytesIO(
  read(0, fstat(0).st_size))
n = int(instream.readline())
a = [int(instream.readline())
     for _ in range(n)]
```

Отметим, что такой оптимизированный способ работает только тогда, когда ввод программы перенаправлен на чтение из файла. Как правило, большинство тестирующих систем так и поступают, однако при локальном тестировании следует не забывать, что этот метод не будет работать с вводом с клавиатуры. Также этот прием не будет работать для так называемых интерактивных задач, т. е. задач, в которых тестирующая система подает входные данные не целиком, а параллельно с работой самого решения.

> Можно пойти дальше и добавить в начало программы строку `input = instream.readline`, получив тем самым возможность вызывать функцию `input()` привычным образом, используя при этом оптимизированный вариант. Такие приемы типичны в спортивном программировании, хотя их применение — дело предпочтений.

25.1.3. Ускорение вывода массивов в Python

При печати больших объемов данных в Python, как показывает практика, некоторого ускорения можно добиться путем преобразования их в строку вручную. Например, для печати массива можно преобразовать каждый его элемент в строку, затем склеить все эти строки друг с другом и вывести результат.

Стандартная реализация	Оптимизированный вариант
`for item in array:` `  print(item)`	`print("\n".join(map(str, array)))`

Выигрыш на практике зависит от деталей конкретной реализации Python, производительности компьютера и размеров массивов, однако на момент написания книги эта альтернативная реализация позволяет достичь приблизительно двукратного ускорения. Выигрыш достигается как при печати элементов в одну строку, так и при печати по одному элементу в строке.

25.2. Производительность ввода-вывода в C++

Несмотря на то что язык C++ довольно быстр сам по себе, а компиляторы C++ применяют к генерируемому коду множество оптимизаций, стандартные библиотечные средства ввода-вывода в C++ также не «блещут» скоростью.

Ниже мы рассмотрим эмпирические данные и дадим несколько простых советов, которые позволяют в некоторых случаях достичь многократного ускорения ввода-вывода.

> Мы не будем рассматривать «тяжеловесные» методы ускорения, такие, как использование специальных низкоуровневых функций быстрого чтения входных байтов или написание кода конвертации строк в числа вручную. Хотя такие методы позволяют добиться дополнительного многократного ускорения, необходимость прибегать к таким приемам, как правило, означает в контексте спортивного программирования, что выбранный алгоритм решения задачи или его реализация недостаточно быстры.

25.2.1. Эмпирические показатели ввода-вывода в C++

Приведем показатели, полученные автором книги путем измерений на типовом компьютере. Как всегда в таких случаях, важны не конкретные числа (которые варьируются в зависимости от условий тестирования), а приблизительный их порядок и относительные сравнения.

Операция	Размер входных данных	Время работы
Чтение массива 9-значных целых чисел (реализация с помощью `std::cin`)	$n = 10^5$ (1 МиБ)	0,04 с
	$n = 10^6$ (10 МиБ)	0,4 с
Вывод массива 9-значных целых чисел в одну строку (реализация с помощью `std::cout`)	$n = 10^5$ (1 МиБ)	0,01 с
	$n = 10^6$ (10 МиБ)	0,1 с
Вывод массива 9-значных целых чисел по одному в строке (реализация с помощью `std::cout` и `std::endl`)	$n = 10^5$ (1 МиБ)	0,06 с
	$n = 10^6$ (10 МиБ)	0,6 с

Как видно из этих результатов, время, затрачиваемое на чтение, становится ощутимым, только начиная с нескольких сотен тысяч или миллионов элементов. Печать выходных данных медленна только в случае, когда совершается много переводов строки.

Ниже мы рассмотрим несколько простых приемов, позволяющих сильно улучшить эти показатели без усложнения кода.

25.2.2. Настройка *sync_with_stdio* для ускорения ввода-вывода в C++

Одна из причин низкой скорости работы потоков `std::cin` и `std::cout` — в том, что по умолчанию они настроены на «сосуществование» с методами ввода-вывода из языка C: `scanf`, `printf` и т. п. Хотя эти стандартные настройки более безопасны и имеют смысл в программах, одновременно использующих функции ввода-вывода C и C++, или многопоточных программах, они приводят к значительным накладным расходам.

К счастью, существует простой способ избавиться от этих накладных расходов: вызов функции `std::ios::sync_with_stdio` с передачей параметра «false» позволяет выключить синхронизацию потоков. На практике это позволяет многократно — в 4 и более раз — ускорить чтение входных данных через `std::cin`.

Таким образом, имеет смысл включать следующую строку в начало решений, особенно в случае задач с большим объемом входных или выходных данных.

Отключение синхронизации

```
int main() {
  ios::sync_with_stdio(false);
  // далее решение задачи
```

Возможные побочные эффекты

Следует понимать, что при использовании этой настройки нельзя комбинировать методы ввода-вывода из C++ и из C в одной программе. Например, если программа будет выводить одно число с помощью `std::cout`, а другое — с помощью `printf`, то порядок, в котором они окажутся в итоговых выходных данных, не определен. Наша рекомендация, которой мы также следуем во всех примерах, данных в книге, — полностью отказаться от методов ввода-вывода C, таких, как, `scanf` и `printf`, даже несмотря на то что в отдельных случаях они могут быть чуть более удобными в использовании.

25.2.3. Настройка *tie* для ускорения ввода-вывода в C++

У потока `std::cin` есть также и другая настройка, значение которой по умолчанию приводит к излишним накладным расходам. Это так называемое «связывание» потока ввода `std::cin` с потоком вывода `std::cout`.

Его предназначение в языке C++ в том, чтобы упростить написание программ, взаимодействующих с пользователем через консоль, т. е. поочередно выводящих и считывающих какие-либо данные. Без «связывания» программист должен самостоятельно заботиться о том, чтобы предназначенные для пользователя сообщения действительно печатались до того, как программа начинает ожидать пользовательского ввода. Для этого требовалось бы в каждой такой ситуации вызывать метод `flush` (или любой другой метод, который вызывает `flush` неявно). «Связывание» же производит это действие автоматически, т. е. перед каждой операцией чтения из `std::cin` происходит вызов метода `flush` для `std::cout`.

В абсолютном большинстве задач спортивного программирования вызовы метода `flush`, а следовательно, и «связывание», не требуются, поскольку на вход программе подается заранее подготовленный текстовый файл. Исключение составляют лишь сравнительно редкие «интерактивные» задачи, в которых тестирующая система подает данные на вход программы порциями, ожидая каждый раз определенного вывода от программы.

Поскольку «связывание» приводит к дополнительным накладным расходам, особенно при чередующихся операциях с `std::cin` и `std::cout`, имеет смысл его отключать — с помощью вызова метода `tie` с передачей ему нулевого указателя. На практике, в зависимости от различных факторов, ускорение будет варьироваться от пренебрежимо малого до двукратного и более.

В сочетании с оптимизацией из предыдущего раздела начало программы будет выглядеть следующим образом.

Отключение синхронизации и связывания:

```
int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  // далее решение задачи
```

Возможные побочные эффекты

Повторимся, что в случае «интерактивных» задач эту оптимизацию применять нецелесообразно, поскольку при решении такой задачи удобно положиться на «связывание» для автоматической печати ответа на предыдущий запрос перед чтением следующего запроса.

25.2.4. Отказ от *endl* для ускорения вывода в C++

Напомним, что приведенные выше эмпирические данные показали существенное замедление в случае печати данных с большим количеством переводов строк с помощью `std::endl`.

Причиной здесь является то, что `std::endl`, вопреки своему названию, не только выводит перевод строки, но и дополнительно производит вызов метода `flush`. В то время как это поведение полезно в некоторых случаях — например, при написании программ, печатающих для пользователя сообщения о статусе какой-либо операции, — в контексте спортивного программирования в абсолютном большинстве задач это не требуется. Исключением являются лишь «интерактивные» задачи, в которых решение должно вызывать `flush` после вывода ответа на каждый входной запрос.

Поскольку `flush` — относительно «дорогая» операция (она отправляет содержимое буфера вывода операционной системе, а та, в свою очередь, — файловой системе), то при решении типичных задач спортивного программирования имеет смысл отказаться от использования `flush` и `endl`. Приведем пример.

Стандартная реализация

```
for (int i = 0; i < n; ++i)
  cout << a[i] << endl;
```

Оптимизированный вариант

```
for (int i = 0; i < n; ++i)
  cout << a[i] << '\n';
```

Такое небольшое изменение позволяет многократно — в наших измерениях до 6 раз — ускорить вывод данных с большим числом переводов строк.

Глава 26.
Графы. Алгоритм Дейкстры

Алгоритм Дейкстры позволяет эффективно находить кратчайшие пути из заданной стартовой вершины графа до всех остальных его вершин.

В отличие от алгоритма обхода в ширину, алгоритм Дейкстры позволяет работать со **взвешенными** графами, т. е. такими графами, в которых каждому ребру приписан определенный вес.

> ***Историческая справка***
>
> Эдсгер Вибе Дейкстра (Edsger Wybe Dijkstra) — нидерландский ученый, который в 1959 г. опубликовал статью с описанием алгоритма поиска кратчайших путей в графах. По некоторым данным, алгоритм был придуман им еще в 1956 г. Впоследствии лауреат премии Тьюринга и других престижных наград Эдсгер Дейкстра внес большой вклад в развитие различных областей информатики и программирования.

26.1. Постановка задачи поиска кратчайших путей

Пусть дан некоторый граф с n вершинами и m ребрами. Граф может быть как ориентированным, так и неориентированным. Для каждого ребра графа указан **вес** (некоторое число). Также указана стартовая вершина s.

Задача — найти длину кратчайшего пути из s в каждую из остальных вершин графа. Под «кратчайшим» подразумевается путь, сумма весов ребер которого принимает минимально возможное значение.

Расширенная постановка задачи — не только определить длины кратчайших путей, но и вывести весь кратчайший путь до указанной вершины t.

26.2. Пример

Классический и самый наглядный вариант этой задачи — поиск оптимального маршрута из одного города в другой. Пусть нам известна карта дорог, связывающих между собой n городов; для каждой дороги известно, какие города она соединяет и сколько времени необходимо для ее преодоления. Требуется найти самый быстрый маршрут от одного заданного города до другого.

Рассмотрим в качестве примера следующий неориентированный граф (рис. 26.1).

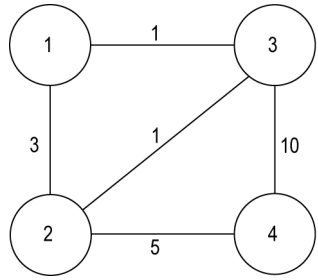

Рис. 26.1. Пример неориентированного графа
с 4 вершинами и 5 ребрами

Если стоит задача найти кратчайший путь из города №1 в город №4, то оптимальным решением будет путь 1-3-2-4 с суммарной стоимостью 7.

26.3. Алгоритм Дейкстры

Алгоритм Дейкстры можно представить как итеративный алгоритм. Начиная с неизвестных расстояний до всех остальных вершин, итерации этого алгоритма постепенно уточняют и улучшают эти расстояния, так что к моменту завершения алгоритма он приходит к искомому ответу.

Пошаговое описание алгоритма:

1. Заведем массив текущих расстояний *dist* и массив отметок посещенных вершин *visited*. Изначально массив *dist* заполнен значениями «бесконечность», кроме стартового элемента *dist[s]*, равного нулю. Массив отметок *visited* заполнен значениями «ложь».
2. Изо всех не посещенных пока вершин выбирается вершина v с наименьшим значением *dist*:

$$v := \underset{\substack{i=1\ldots n,\\ visited[i]=false}}{\arg\min} dist[i].$$

3. Выбранная вершина v отмечается посещенной:

$$visited[v] := true.$$

4. Просматриваются все ребра, исходящие из вершины v, и вдоль каждого ребра производится **релаксация**, т. е. попытка улучшить текущее расстояние до новой вершины. Если обозначить конец текущего ребра через t, а его вес через c, то под релаксацией подразумевается операция:

$$dist[t] := \min\left(dist[t], dist[v] + c\right).$$

5. Если больше непосещенных вершин с конечным расстоянием не осталось, то алгоритм завершается. В противном случае, возвращаемся к шагу №2.

26.4. Ограничения алгоритма Дейкстры

Алгоритм Дейкстры применим только в том случае, когда **веса всех ребер неотрицательны**.

26.5. Пример работы алгоритма Дейкстры

Рассмотрим пошаговую работу алгоритма Дейкстра на приведенном выше графе со стартовой вершиной $s = 1$ (рис. 26.2).

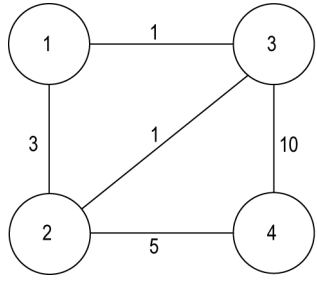

Рис. 26.2. Пример входного графа

Изначально ни одна вершина не посещена, а текущие расстояния до всех вершин, кроме s, равны бесконечности (рис. 26.3).

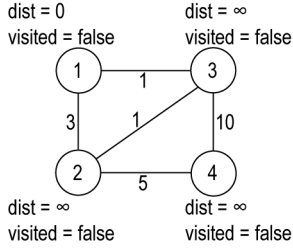

Рис. 26.3. Начальное состояние алгоритма Дейкстры на указанном графе

Алгоритм выбирает непосещенную вершину с минимальным расстоянием — это вершина $v = 1$. Эта вершина отмечается как посещенная, и производятся релаксации по исходящим из нее ребрам в вершины №2 и 3 (обе релаксации улучшают текущие расстояния до этих вершин) (рис. 26.4).

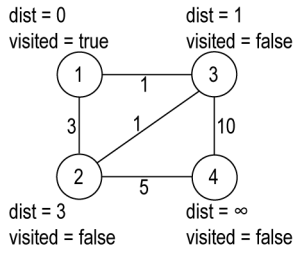

Рис. 26.4. Состояние алгоритма
после проведения релаксаций из вершины №1

Алгоритм снова выбирает непосещенную вершину с минимальным расстоянием — теперь это вершина $v = 3$. Она помечается посещенной, и по ребрам из нее производятся релаксации. Ребро в вершину №1 не приводит к изменениям, но ребра в вершины №2 и 4 улучшают текущие расстояния до этих вершин (рис. 26.5).

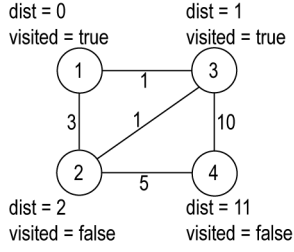

Рис. 26.5. Состояние алгоритма Дейкстры
после проведения релаксаций из вершин №1, 3

Алгоритм снова выбирает вершину — теперь это $v = 2$. Она помечается посещенной, и производятся релаксации; при этом улучшается текущее расстояние до вершины №4 (рис. 26.6).

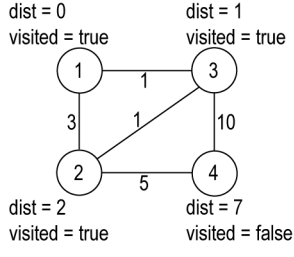

Рис. 26.6. Состояние алгоритма Дейкстры
после проведения релаксаций из вершин №1, 2, 3

Алгоритм снова выбирает вершину — это $v = 4$. Она помечается посещенной; ни одна релаксация из этой вершины больше не улучшает ответ (рис. 26.7).

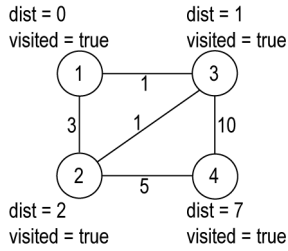

Рис. 26.7. Состояние алгоритма Дейкстры
после проведения релаксаций из всех вершин

Больше непосещенных вершин не осталось. Алгоритм завершает свою работу.

26.6. Восстановление кратчайшего пути

Алгоритм Дейкстры в том виде, как он описан выше, находит только длины кратчайших путей, но не возвращает самого пути. Однако алгоритм несложно улучшить, чтобы по результатам его работы можно было восстановить кратчайший путь до любой заданной вершины.

Для этого введем дополнительный массив *parent*. Для каждой вершины i значение *parent*[i] будет равно номеру вершины, из которой кратчайший путь попадает в i. Для построения этого массива достаточно добавить одну операцию в процедуру релаксации: если при просмотре некоторого ребра (v, t) происходит смена значения *dist*[t], то дополнительно будем присваивать *parent*[t] := v.

Неформальное, но часто употребляемое название массива *parent* — это «массив предков».

Имея такой массив *parent*, кратчайший путь до любой вершины i можно восстановить, переходя от i к *parent*[i], затем к *parent*[*parent*[i]] и т. д., до тех пор пока не будет достигнута вершина s. С точки зрения реализации удобно полагать *parent*[s] = −1.

Продемонстрируем, чему будут равны значения *parent* на нашем примере (рис. 26.8).

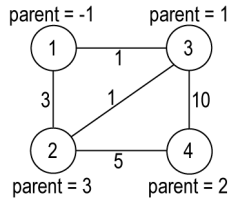

Рис. 26.8. Отметки *parent* для графа,
рассмотренного в предыдущем разделе

Восстановление пути до вершины №4 в этом примере будет происходить следующим образом:

- добавляем в путь вершину №4;
- переходим в *parent*[*4*], т. е. вершину №2. Добавляем в начало пути вершину №2;
- переходим в *parent*[*2*], т. е. вершину №3. Добавляем в начало пути вершину №3;
- переходим в *parent*[*3*], т. е. вершину №1. Добавляем в начало пути вершину №1;
- завершаем восстановление пути, поскольку мы дошли до стартовой вершины $s = 1$. Итоговый кратчайший путь: 1-3-2-4.

26.7. Доказательство алгоритма Дейкстры

Обозначим через $ans[v]$ истинную длину кратчайшего пути от вершины s до вершины v.

Заметим, что алгоритм Дейкстры по своей структуре не может найти путей более коротких, чем истинные кратчайшие пути, т. е. в любой момент работы алгоритма для каждой вершины v выполняется:

$$dist[v] \geq ans[v].$$

Это утверждение можно доказать, заметив, что для всякой вершины v и текущего значения $dist[v]$ можно восстановить цепочку ребер, образующих путь из s в v и имеющих сумму, равную $dist[v]$; если бы такая цепочка имела сумму, меньшую $ans[v]$, то это противоречило бы определению кратчайшего пути.

Тогда **ключевое свойство**, которое требуется доказать: для каждой вершины v верно, что в момент, когда она становится посещенной ($visited[v]=true$), текущее значение $dist[v]$ равно истинному расстоянию $ans[v]$.

Доказывать это утверждение будем по индукции. На самой первой итерации истинность этого утверждения следует из того, что выбрана будет вершина s, которой в начале алгоритма было присвоено значение $dist[s] = 0$ — что равно $ans[s]$.

Предположим теперь, что утверждение верно для нескольких первых итераций алгоритма, и докажем, что для выбранной на очередном шаге вершины v выполняется $dist[v] = ans[v]$. Для этого докажем, что $dist[v] \leq ans[v]$, ведь в сочетании с доказанным ранее неравенством это докажет, что правая и левая части равны.

- Рассмотрим некоторый кратчайший путь из s в v. Найдем в этом пути первую непосещенную вершину — обозначим ее через w (заметим, что она может совпадать с v), а предшествующую ей вершину — через u.
- Из неотрицательности весов ребер следует, что $ans[w] \leq ans[v]$.
- Далее, заметим, что для вершины w справедливо, что $dist[w] = ans[w]$. Это следует из того, что вершина u — уже посещенная, и по предположению индукции верно $dist[u] = ans[u]$, а на итерации обработки вершины u была произведена релаксация вдоль ребра (u, w), входящего в кратчайший путь.

- Последнее наблюдение: $dist[v] \le dist[w]$. Это вытекает из того факта, что вершина v была выбрана алгоритмом Дейкстры на текущей итерации как непосещенная вершина с минимальным текущим расстоянием, притом что как v, так и w — непосещенные вершины.
- Объединяя три предыдущих факта, получаем доказательство требуемого неравенства:

$$dist[v] \le dist[w] = ans[w] \le ans[v].$$

Таким образом, $dist[v] = ans[v]$, что и требовалось доказать.

26.8. Квадратичная реализация алгоритма Дейкстры

Приведем сначала реализацию, непосредственно следующую описанию алгоритма, без использования каких-либо специальных структур данных.

26.8.1. Реализация

Для определенности напишем программу, которая считывает заданный в виде списка ребер ориентированный граф, строит списки смежности графа, запускает на нем алгоритм Дейкстры со стартовой вершиной $s = 1$ и выводит длину кратчайшего пути до вершины n и сам путь (либо «−1», если пути не существует).

Python

```
vertex_count, edge_count = map(
    int, input().split())
adj_list = [
    [] for _ in range(
        vertex_count)]
for _ in range(edge_count):
  vertex_from, vertex_to, price = \
      map(int, input().split())
  vertex_from -= 1
  vertex_to -= 1
  adj_list[vertex_from].append(
      (vertex_to, price))

start = 0
dist = [None] * vertex_count
visited = [False] * vertex_count
parent = [None] * vertex_count
dist[start] = 0
while True:
  v = None
  for i in range(vertex_count):
```

C++

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;
using Edge = pair<int, int>;
int main() {
  int vertex_count, edge_count;
  cin >> vertex_count
      >> edge_count;
  vector<vector<Edge>> adj_list(
      vertex_count);
  for (int i = 0; i < edge_count;
       ++i) {
    int vertex_from, vertex_to,
        price;
    cin >> vertex_from
        >> vertex_to >> price;
    --vertex_from;
    --vertex_to;
    adj_list[vertex_from]
```

```
    if (not visited[i] and
        dist[i] is not None and
        (v is None or
         dist[i] < dist[v])):
      v = i
  if v is None:
    break
  visited[v] = True
  for to, price in adj_list[v]:
    new_d = dist[v] + price
    if (dist[to] is None or
        new_d < dist[to]):
      dist[to] = new_d
      parent[to] = v

finish = vertex_count - 1
if dist[finish] is None:
  print(-1)
else:
  print(dist[finish])
  path = []
  v = finish
  while v is not None:
    path.append(v + 1)
    v = parent[v]
  print(*reversed(path))
```

```
      .push_back(
        Edge(vertex_to, price));
  }

  int start = 0;
  vector<int> dist(
      vertex_count, -1);
  dist[start] = 0;
  vector<char> visited(
      vertex_count);
  vector<int> parent(
      vertex_count, -1);
  for (;;) {
    int v = -1;
    for (int i = 0;
         i < vertex_count;
         ++i) {
      if (!visited[i] &&
          dist[i] != -1 && (
            v == -1 ||
            dist[i] < dist[v])) {
        v = i;
      }
    }
    if (v == -1)
      break;
    visited[v] = true;
    for (Edge e : adj_list[v]) {
      int to = e.first;
      int price = e.second;
      int new_d = dist[v] + price;
      if (dist[to] == -1 ||
          new_d < dist[to]) {
        dist[to] = new_d;
        parent[to] = v;
      }
    }
  }

  int finish = vertex_count - 1;
  cout << dist[finish] << endl;
  if (dist[finish] == -1)
    return 0;
  vector<int> path;
  for (int v = finish; v != -1;
```

```
        v = parent[v]) {
    path.push_back(v);
  }
  reverse(path.begin(),
          path.end());
  for (auto v : path)
    cout << v + 1 << ' ';
}
```

26.8.2. Разбор реализации

Остановимся чуть подробнее на отдельных частях этой реализации.

- Вначале идет чтение входных данных и построение графа в виде списков смежности. Поскольку граф взвешенный, каждый элемент списка смежности — это пара чисел: номер вершины, в которую ведет ребро, и вес ребра.
- Далее происходит инициализация переменных для алгоритма Дейкстры.

Python

```
dist = [None] * vertex_count
visited = [False] * vertex_count
parent = [None] * vertex_count
dist[start] = 0
```

C++

```
vector<int> dist(
    vertex_count, -1);
dist[start] = 0;
vector<char> visited(
    vertex_count);
vector<int> parent(
    vertex_count, -1);
```

Хранение бесконечных расстояний

Отметим, что для обозначения бесконечных расстояний мы используем специальное значение («None» в Python и «−1» в C++); код можно немного упростить, если в конкретной задаче известна некоторая числовая константа, которая заведомо больше всевозможных рассматриваемых длин (такую константу часто обозначают «INF»). Однако, поскольку здесь мы рассматриваем общий вид алгоритма Дейкстры без привязки к конкретной задаче, мы не используем этот прием.

Использование vector<char> в реализации на C++

Несмотря на то что, по смыслу, массив `visited` содержит булевы значения, мы не используем vector<bool>, поскольку vector<bool> использует технику битового сжатия, что приведет лишь к ненужным накладным расходам без сколько-нибудь заметного в данном случае выигрыша в потреблении памяти.

- Далее начинается собственно алгоритм Дейкстры. На каждой итерации алгоритма мы выбираем вершину v как вершину с минимальным расстоянием среди всех еще не посещенных вершин.

Python

```
v = None
for i in range(vertex_count):
  if (not visited[i] and
      dist[i] is not None and
      (v is None or
       dist[i] < dist[v])):
    v = i
```

C++

```
int v = -1;
for (int i = 0;
     i < vertex_count;
     ++i) {
  if (!visited[i] &&
      dist[i] != -1 && (
        v == -1 ||
        dist[i] < dist[v])) {
    v = i;
  }
}
```

- Если подходящей вершины нет, то алгоритм завершается. Иначе — выбранная вершина помечается посещенной.
- Далее происходит обход всех ребер, исходящих из *v*, и релаксации вдоль каждого из них, т. е. попытки улучшить расстояние до вершины-конца ребра.

Python

```
for to, price in adj_list[v]:
  new_d = dist[v] + price
  if (dist[to] is None or
      new_d < dist[to]):
    dist[to] = new_d
    parent[to] = v
```

C++

```
for (Edge e : adj_list[v]) {
  int to = e.first;
  int price = e.second;
  int new_d = dist[v] + price;
  if (dist[to] == -1 ||
      new_d < dist[to]) {
    dist[to] = new_d;
    parent[to] = v;
  }
}
```

- При выводе результата мы восстанавливаем кратчайший путь с помощью движения по ссылкам *parent*.

Python

```
v = finish
while v is not None:
  path.append(v + 1)
  v = parent[v]
```

C++

```
vector<int> path;
for (int v = finish; v != -1;
     v = parent[v]) {
  path.push_back(v);
}
```

- Поскольку восстановленный путь получается в обратном порядке, при печати результата порядок элементов в нем меняется на противоположный.

26.8.3. Время работы

Асимптотическая сложность этой реализации составляет $O(n^2)$, где n — число вершин, поскольку в худшем случае алгоритм совершает n итераций, на каждой из которых затрачивается n операций для выбора вершины v. Помимо этого, на каждой итерации совершаются релаксации, однако это не влияет на асимптотику: общее число релаксаций по всем итерациям равно числу ребер m, и число ребер не превосходит n^2.

> Приведенная оценка асимптотики предполагает, что в графе нет кратных ребер или их немного. Если же кратных ребер много и мы не избавляемся от них до запуска алгоритма Дейкстры, то итоговая асимптотика составит $O(n^2 + m)$.

Отметим, что эта реализация является асимптотически оптимальной для плотных графов — т. е. графов, у которых число ребер m близко к n^2.

На практике можно ожидать, что эта реализация будет работать достаточно быстро (в пределах секунды на типичном компьютере) при числе вершин, не превосходящем нескольких тысяч или десятков тысяч.

26.9. Алгоритм Дейкстры для разреженных графов

В случае разреженных графов, т. е. графов, у которых число ребер m значительно меньше, чем n^2, реализация из предыдущего раздела будет чрезмерно расточительной. Узким местом в ней становится шаг выбора вершины v на каждой итерации.

Для ускорения процедуры выбора вершины нам нужна структура данных, которая эффективно поддерживает операции: **извлечение минимального элемента**, добавление элемента, удаление элемента. В стандартной библиотеке C++ есть две подходящих для этого структуры данных: std::set (обычно основанный на красно-черных деревьях) и std::priority_queue (основанный на двоичной куче). В стандартной библиотеке Python — модуль heapq (основанный на двоичной куче).

Все эти структуры данных поддерживают нужные нам операции с асимптотикой $O(\log n)$, что означает, что алгоритм Дейкстры на их основе будет иметь асимптотику $O(m \log n)$. Узким местом при этом станет выполнение релаксаций: в худшем случае будет произведено m успешных релаксаций, каждая из которых потребует обновления структуры данных за время $O(\log n)$.

26.9.1. Реализация с использованием std::set

Рассмотрим в этом разделе, как можно эффективно реализовать алгоритм Дейкстры с использованием структуры данных std::set из стандартной библиотеки C++.

> В стандартной библиотеке Python нет класса с аналогичными возможностями — сбалансированного дерева, которое поддерживало бы эффективное извлечение минимума.

Самый простой — и, вероятно, самый популярный в спортивном программировании — вариант реализации алгоритма Дейкстры основан на хранении в std::set пар следующего вида:

$$(dist[i], i),$$

где i пробегает по номерам всех вершин, для которых $visited[i] = false$. Опишем подробно операции над этим множеством.

- Изначально множество содержит только одну пару: (0, s).
- Для выбора вершины v на каждой итерации алгоритма Дейкстры требуется извлечь из множества пару с минимальным значением *dist*, для чего достаточно просто обратиться к методу `begin()`.
- При проведении каждой релаксации надо сначала удалить из множества пару со старым значением *dist*, а затем добавить обратно пару с новым значением.

Отметим, что в такой реализации хранение массива *visited* не требуется — его роль теперь выполняет std::set: все уже посещенные вершины отсутствуют в этом множестве, что автоматически означает, что такие вершины не будут выбраны повторно.

Реализация (без ввода-вывода, которые остаются без изменения).

C++

```
set<pair<int, int>> order;
vector<int> dist(vertex_count, -1);
dist[start] = 0;
order.insert({0, start});
vector<int> parent(vertex_count, -1);
while (!order.empty()) {
  int v = order.begin()->second;
  order.erase(order.begin());
  for (Edge e : adj_list[v]) {
    int to = e.first;
    int price = e.second;
    int new_d = dist[v] + price;
    if (dist[to] == -1 || new_d < dist[to]) {
      order.erase({dist[to], to});
      dist[to] = new_d;
      parent[to] = v;
      order.insert({dist[to], to});
    }
  }
}
```

Эта реализация имеет асимптотику $O(m \log n)$.

Возможные дальнейшие оптимизации

Отметим, что теоретически возможна следующая оптимизация этого решения: вместо хранения в std::set пар элементов мы можем хранить одни лишь индексы вершин. Для корректного сравнения вершин по расстояниям при этом можно использовать специальный компаратор (см. аргумент compare конструктора std::set). Впрочем, практика показывает, что такая оптимизация не дает реального увеличения скорости, по крайней мере, на протестированных автором компиляторах.

26.9.2. Реализация с использованием двоичной кучи

Для эффективной реализации алгоритма Дейкстры с использованием двоичной кучи будем, как и в предыдущем разделе, хранить в ней пары вида:

$$(dist[i], i),$$

где i пробегает по номерам всех вершин, для которых $visited[i] = false$.

Напомним, что двоичная куча — это структура данных, которая поддерживает операции добавления элемента и извлечения минимума. При этом куча в стандартной своей реализации **не** поддерживает операцию изменения произвольного элемента, поэтому при проведении релаксаций мы не можем удалить пару со старым значением *dist*. Впрочем, это не составляет большой проблемы: асимптотику эти лишние элементы не ухудшают, а с точки зрения реализации игнорировать эти элементы при извлечении минимума несложно.

Итак, операции над двоичной кучей в алгоритме Дейкстры будут следующими:

- изначально куча содержит только одну пару: $(0, s)$;
- для выбора вершины v на каждой итерации алгоритма Дейкстры требуется извлекать из кучи минимальные элементы до тех пор, пока не будет встречен элемент, в котором сохраненное расстояние равно значению *dist* для этой же вершины;
- при выполнении успешной релаксации, т. е. уменьшении $dist[i]$ для некоторого i, будем добавлять в кучу пару $(dist[i], i)$.

Массив *visited* в этой реализации, как и в реализации на основе std::set, не требуется.

Реализация

Python

```
dist = [None] * vertex_count
order = []
parent = [None] * vertex_count
dist[start] = 0
heappush(order, (0, start))
while order:
  saved_dist, v = heappop(order)
  if saved_dist != dist[v]:
    continue
```

C++

```
priority_queue<pair<int, int>>
    order;
vector<int> dist(vertex_count, -1);
dist[start] = 0;
order.push({0, start});
vector<int> parent(
    vertex_count, -1);
while (!order.empty()) {
  int saved_dist =
```

```
for to, price in adj_list[v]:
  new_d = dist[v] + price
  if (dist[to] is None or
      new_d < dist[to]):
    dist[to] = new_d
    parent[to] = v
    heappush(order,
             (dist[to], to))
```

```
      -order.top().first;
  int v = order.top().second;
  order.pop();
  if (dist[v] != saved_dist)
    continue;
  for (Edge e : adj_list[v]) {
    int to = e.first;
    int price = e.second;
    int new_d = dist[v] + price;
    if (dist[to] == -1 ||
        new_d < dist[to]) {
      dist[to] = new_d;
      parent[to] = v;
      order.push({-dist[to], to});
    }
  }
}
```

Смена знака у элементов priority_queue в решении на C++

Интерфейс priority_queue имеет такую особенность, что по умолчанию эта структура данных находит не минимум, а максимум. Хотя это поведение можно изменить с помощью передачи специального компаратора (std::greater), использованный нами подход со сменой знака (т. е. хранением −dist[i] вместо dist[i]) более лаконичен и, на наш взгляд, вполне оправдан в контексте спортивного программирования.

Эта реализация имеет асимптотику $O(m \log n)$.

Если сравнивать между собой реализации на языке C++, то на практике метод на основе двоичной кучи оказывается до 50% быстрее метода на основе std::set.

26.9.3. Обзор прочих оптимизаций

За прошедшие с момента его публикации десятилетия алгоритм Дейкстры был тщательно изучен и были предложены многочисленные варианты с использованием различных структур данных и прочих оптимизаций. Приведем здесь для справки некоторые из них.

- Структура данных под названием **фибоначчиева куча** позволяет реализовать алгоритм Дейкстры с асимптотикой $O(m + n \log n)$, что в общем случае является **асимптотически оптимальным** алгоритмом. Однако на практике этот алгоритм практически не применяется из-за слишком большой скрытой константы. Этот алгоритм был предложен Фредманом и Тарьяном в 1984 г.
- В случае когда веса ребер ограничены некоторой константой C, хороших результатов можно добиться с использованием структура данных **деревья ван Эмде Боаса**: алгоритм Дейкстры будет иметь асимптотику $O(m \log \log C)$ и хорошие эмпирические показатели. Этот алгоритм был предложен Ахуйей (Ahuja) и пр. в 1990 г.

- Существует сравнительно простая и в то же время неплохо работающая на практике оптимизация: **двунаправленный** алгоритм Дейкстры. Эта оптимизация применима к случаю, когда и стартовая, и конечная вершины искомого пути зафиксированы. Идея заключается в том, что можно одновременно и параллельно вести два обхода алгоритмом Дейкстры из стартовой и из конечной вершин и останавливать их оба в момент, когда они оба посетят какую-либо вершину.
- Наконец, в спортивном программировании можно использовать прием под названием **корневая эвристика**, который позволяет достичь асимптотики $O\left(n\sqrt{n}+m\right)$ при довольно простой реализации.

26.10. Пример решения задачи. Оптимальный путь четной длины

Задача. Дан взвешенный неориентированный граф, содержащий n вершин и m ребер. Требуется найти путь из вершины №1 в вершину №n, состоящий из четного числа ребер, и при этом сумма весов входящих в него ребер была бы минимально возможной.

Входные данные содержат n и m в первой строке и описания ребер в виде троек a_i, b_i, c_i в последующих строках. Гарантируется, что $2 \le n \le 10^5$, $0 \le m \le 10^5$, $1 \le a_i, b_i \le n$, $0 \le c_i \le 1000$. В графе могут быть кратные ребра и петли.

Вывести требуется суммарную стоимость искомого пути в первой строке и вершины пути во второй. Если ответов несколько, разрешается вывести любой. Если искомого пути не существует, следует вывести «−1».

Примеры

Входные данные	**Требуемый результат**
3 3 1 2 10 1 3 3 2 3 10	20 1 2 3
Входные данные	**Требуемый результат**
3 2 1 2 1 1 3 1	-1

Решение

Стандартный алгоритм Дейкстры не дает никаких гарантий относительно четности длин найденных путей. Однако можно модифицировать входной граф: «раздвоить» каждую вершину на две: одну для прибытия в эту вершину по пути четной длины и другую — для прибытия по пути нечетной длины. Каждому ребру исходного графа

будет соответствовать два ребра в модифицированном графе, каждое — со сменой четности.

Опишем этот прием чуть подробнее. Удобно говорить о модифицированном графе как о графе «состояний», где состояние в данном случае — это пара (v, k), т. е. номер v исходной вершины и четность k пути до нее. Для каждого ребра между вершинами i и j исходного графа в графе состояний будут два следующих ребра:

- между (i, 0) и (j, 1);
- между (i, 1) и (j, 0).

Тогда исходная задача превращается в задачу поиска кратчайшего пути из вершины (1, 0) в вершину (n, 0) в графе состояний. К этой стандартной задаче уже можно применить обычный алгоритм Дейкстры (для разреженных графов, поскольку $m << n^2$ при максимальных ограничениях).

Асимптотическая сложность решения — $O(m \log n)$.

Реализация

При реализации удобно, как обычно, использовать сквозную нумерацию всех вершин — в данном случае для графа состояний это будет нумерация от 0 до $2n - 1$. При этом возможны разные подходы к нумерации; использованный нами вариант — это формула $2v + k$ (т. е. младший бит номера вершины содержит четность, а все остальные биты — номер исходной вершины). При восстановлении пути для печати ответа нам надо будет, наоборот, делить номера вершин на 2 нацело.

Отметим, что здесь мы приведем лаконичные реализации алгоритма Дейкстра с краткими именами переменных — такие, какими они чаще всего и бывают в спортивном программировании. В реализации на C++ будем использовать метод на основе std::set, в реализации на Python — heapq.

Python

```
from heapq import *

n, m = map(int, input().split())
g = [[] for _ in range(n * 2)]
for i in range(m):
  v1, v2, price = map(
      int, input().split())
  v1 -= 1
  v2 -= 1
  g[v1 * 2].append(
      (v2 * 2 + 1, price))
  g[v1 * 2 + 1].append(
      (v2 * 2, price))
  g[v2 * 2].append(
      (v1 * 2 + 1, price))
  g[v2 * 2 + 1].append(
      (v1 * 2, price))
```

C++

```
#include <algorithm>
#include <iostream>
#include <set>
#include <utility>
#include <vector>
using namespace std;
using Edge = pair<int, int>;
int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<Edge>> g(n * 2);
  for (int i = 0; i < m; ++i) {
    int v1, v2, price;
    cin >> v1 >> v2 >> price;
    --v1;
    --v2;
    g[v1 * 2].push_back(
```

```
d = [None] * (n * 2)
p = [None] * (n * 2)
q = []
d[0] = 0
heappush(q, (0, 0))
while q:
  saved_d, v = heappop(q)
  if saved_d != d[v]:
    continue
  for to, price in g[v]:
    nd = d[v] + price
    if (d[to] is None or
        nd < d[to]):
      d[to] = nd
      p[to] = v
      heappush(q, (d[to], to))

v = (n - 1) * 2
if d[v] is None:
  print(-1)
else:
  print(d[v])
  path = []
  while v is not None:
    path.append(v // 2 + 1)
    v = p[v]
  print(*reversed(path))
```

```
        make_pair(v2 * 2 + 1,
                  price));
    g[v1 * 2 + 1].push_back(
        make_pair(v2 * 2,
                  price));
    g[v2 * 2].push_back(
        make_pair(v1 * 2 + 1,
                  price));
    g[v2 * 2 + 1].push_back(
        make_pair(v1 * 2,
                  price));
  }

  set<pair<int, int>> q;
  vector<int> d(n * 2, -1);
  d[0] = 0;
  q.insert({0, 0});
  vector<int> p(n * 2, -1);
  while (!q.empty()) {
    int v = q.begin()->second;
    q.erase(q.begin());
    for (Edge e : g[v]) {
      int to = e.first;
      int price = e.second;
      int nd = d[v] + price;
      if (d[to] == -1 ||
          nd < d[to]) {
        q.erase({d[to], to});
        d[to] = nd;
        p[to] = v;
        q.insert({d[to], to});
      }
    }
  }

  int v = (n - 1) * 2;
  cout << d[v] << endl;
  if (d[v] == -1)
    return 0;
  vector<int> path;
  for (; v != -1; v = p[v])
    path.push_back(v);
  reverse(path.begin(),
          path.end());
  for (int idx : path)
    cout << idx / 2 + 1 << ' ';
}
```

26.11. Пример решения задачи. Ребра кратчайших путей

Задача. Органы автомобильной инспекции установили, что наибольшее число превышений скорости допускается водителями города X, и в особенности во время их поездок в город Y и обратно (точные причины этого не установлены, но, возможно, причина кроется в насыщенной ночной жизни Y и активности жителей X). Инспекторы хотят установить радары на некоторые из дорог, чтобы штрафовать нарушителей. С одной стороны, поскольку установка и обслуживание радара требует больших затрат, нужно обойтись как можно меньшим числом радаров. С другой стороны, радары должны стоять на каждом участке всех дорог, используемых «лихачами». Инспекция установила, что «лихачи» всегда ездят вдоль кратчайшего пути из X в Y (что логично из соображений экономии времени), однако если кратчайших путей несколько, разные водители могут использовать разные пути.

Ваша задача — по данной карте дорог найти те дороги, на которые требуется установить радары. Формат карты дорог следующий: число перекрестков и число дорог в первой строке и описания дорог в последующих строках в виде троек целых чисел «A B C», где *A* и *B* — номера перекрестков, *C* — время проезда по дороге. Перекрестки нумеруются, начиная с единицы, все дороги двунаправленные, между каждой парой перекрестков есть не более одной дороги, и никакая дорога не связывает перекресток с самим собой. Выезд из города X соответствует перекрестку №1, въезд в город Y — перекрестку с наибольшим номером; гарантируется, что существует хотя бы один маршрут, связывающий X и Y. Ограничения: число перекрестков лежит в диапазоне $[2; 10^5]$, число дорог — в диапазоне $[1; 5 \cdot 10^5]$, время проезда по каждой дороге — в диапазоне [1; 1000].

Вывести нужно число требуемых радаров в первой строке и номера дорог, на которые их нужно установить, в виде возрастающей последовательности чисел во второй строке. Дороги нумеруются в том порядке, в котором они идут во входных данных, начиная с единицы.

Примеры

Входные данные	**Требуемый результат**
3 3 1 2 1 1 3 3 2 3 1	2 1 3
Входные данные	**Требуемый результат**
3 2 1 2 100 2 3 100	2 1 2

Решение

Если переформулировать условие задачи формально, то нам требуется найти все ребра, лежащие хотя бы на одном кратчайшем пути между вершинами №1 и №n в заданном взвешенном неориентированном графе.

Алгоритм Дейкстры сам по себе ищет лишь один из кратчайших путей, но не все такие пути. Однако можно заметить, что любое ребро (i, j), лежащее на кратчайшем пути между какой-либо парой вершин s и t, удовлетворяет критерию:

$$dist[s][t] = dist[s][i] + g[i][j] + dist[j][t],$$

где $g[u][v]$ обозначает вес ребра (u, v), а $dist[u][v]$ — длину кратчайшего пути из u в v. В самом деле, если взять любое ребро кратчайшего пути, то весь префикс до этого ребра — тоже кратчайший путь, равно как и весь суффикс после этого ребра.

Таким образом, для решения задачи достаточно вычислить длины кратчайших путей из вершины №1 и длины кратчайших путей из вершины №n. Для этого достаточно дважды запустить алгоритм Дейкстры — в данном случае в его версии для разреженных графов (поскольку при максимальных ограничениях в нашей задаче $m << n^2$). После этого останется только перебрать все ребра и проверить каждое по указанному выше критерию.

Итоговая асимптотика решения — $O(m \log n)$.

Реализация

Python

```
from heapq import *

def dijkstra(start):
  d = [None] * n
  q = []
  d[start] = 0
  heappush(q, (0, start))
  while q:
    saved_d, v = heappop(q)
    if saved_d != d[v]:
      continue
    for to, price in g[v]:
      nd = d[v] + price
      if (d[to] is None or
          nd < d[to]):
        d[to] = nd
        heappush(q, (d[to], to))
  return d

n, m = map(int, input().split())
g = [[] for _ in range(n)]
e = [None] * m
```

C++

```
#include <iostream>
#include <set>
#include <utility>
#include <vector>
using namespace std;

using Edge = pair<int, int>;

vector<int> Dijkstra(
    int n,
    const vector<vector<Edge>>& g,
    int start) {
  set<pair<int, int>> q;
  vector<int> d(n, -1);
  d[start] = 0;
  q.insert({0, start});
  while (!q.empty()) {
    int v = q.begin()->second;
    q.erase(q.begin());
    for (Edge e : g[v]) {
      int to = e.first;
      int price = e.second;
```

```
for i in range(m):
  v1, v2, price = map(
      int, input().split())
  v1 -= 1
  v2 -= 1
  e[i] = v1, v2, price
  g[v1].append((v2, price))
  g[v2].append((v1, price))

d1 = dijkstra(0)
d2 = dijkstra(n - 1)
ans = []
for idx, (v1, v2, price) in \
    enumerate(e):
  if (d1[v1] + price + d2[v2] ==
      d1[n - 1]):
    ans.append(idx + 1)
print(len(ans))
print(*ans)
```

```
      int nd = d[v] + price;
      if (d[to] == -1 ||
          nd < d[to]) {
        q.erase({d[to], to});
        d[to] = nd;
        q.insert({d[to], to});
      }
    }
  }
  return d;
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<Edge>> g(n);
  vector<int> ev1(m), ev2(m),
              eprice(m);
  for (int i = 0; i < m; ++i) {
    cin >> ev1[i] >> ev2[i]
        >> eprice[i];
    --ev1[i];
    --ev2[i];
    g[ev1[i]].push_back(
        make_pair(
            ev2[i], eprice[i]));
    g[ev2[i]].push_back(
        make_pair(
            ev1[i], eprice[i]));
  }

  vector<int> d1 = Dijkstra(
      n, g, 0);
  vector<int> d2 = Dijkstra(
      n, g, n - 1);
  vector<int> ans;
  for (int i = 0; i < m; ++i) {
    if (d1[ev1[i]] + eprice[i] +
        d2[ev2[i]] == d1[n - 1]) {
      ans.push_back(i);
    }
  }
  cout << ans.size() << endl;
  for (int idx : ans)
    cout << idx + 1 << ' ';
}
```

Глава 27.
Графы. Компоненты сильной связности

Компоненты сильной связности — это важный инструмент анализа **ориентированных** графов.

Алгоритм поиска этих компонент и связанные с этим базовые задачи являются классическими как в теории графов, так и в спортивном программировании.

27.1. Определения

Вначале введем необходимые определения.

27.1.1. Сильно связный граф

Ориентированный граф называется сильно связным, если из любой его вершины существуют пути до всех остальных вершин.

Если обозначить символом $\rightsquigarrow$ наличие пути от одной вершины до другой, то в сильно связном графе для любой пары вершин (i, j) верно:

$$i \rightsquigarrow j, \quad j \rightsquigarrow i.$$

Например, изображенный слева граф является сильно связным, а изображенный справа — нет (рис. 27.1).

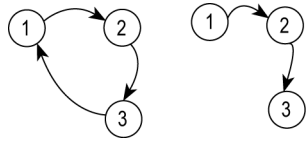

Рис. 27.1. Примеры сильно связного графа (слева) и не сильно связного графа (справа)

27.1.2. Компонента сильной связности

Формальное определение: компонентой сильной связности называется наибольший по включению сильно связный подграф.

Смысл этого определения заключается в том, что мы выбираем одну или несколько вершин таким образом, что между всеми ними попарно существуют пути, и при этом больше ни одной вершины мы добавить таким же образом к ним не можем.

Например, в изображенном ниже графе есть две компоненты сильной связности: одна компонента из вершин №1, 2 и 3 и вторая компонента из вершины №4 (рис. 27.2).

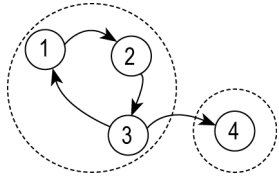

Рис. 27.2. Пример выделения компонент сильной связности в графе

Важное свойство компонент сильной связности: для заданного графа все компоненты выделяются **однозначно** и никакие две компоненты **не пересекаются** друг с другом.

> В математике есть специальный термин для подобного рода отношений: «отношение эквивалентности».

27.1.3. Конденсация графа

Если сжать каждую компоненту сильной связности графа в одну вершину, то получившийся граф называется конденсацией.

Например, для изображенного слева графа конденсацией является изображенный справа граф (рис. 27.3).

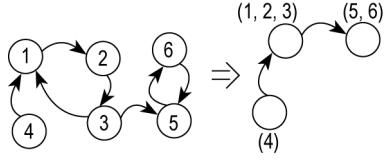

Рис. 27.3. Пример конденсации графа.
В данном случае после конденсации осталось лишь 3 вершины

Ключевое свойство: конденсация — всегда **ациклический** граф. Это легко доказать от противного: если две вершины конденсации принадлежали бы одному циклу, то то же самое было бы верно и для всех входящих в них вершин исходного графа, а, значит, между всеми этими вершинами попарно существовали бы пути — что противоречит определению конденсации.

27.2. Алгоритм поиска компонент сильной связности

Опишем алгоритм выделения компонент сильной связности, основанный на двух обходах в глубину.

> Этот изящный алгоритм был предложен в 1979 г. Косарайю (Kosaraju) и, независимо от него, Шариром (Sharir). Отметим, что это не единственный эффективный алгоритм решения этой задачи: в 1972 г. Тарьяном (Tarjan) был предложен другой алгоритм, несколько более сложный, но зато требующий только одного обхода в глубину.

1. Произведем серию обходов в глубину заданного графа. Сохраним порядок, в котором обход в глубину **выходит** из каждой из вершин.
2. Построим **обратный граф**, т. е. граф, в котором каждое ребро переориентировано в обратную сторону.
3. Произведем серию обходов в глубину обратного графа, причем запускать обходы будем в порядке, **обратном** порядку, полученному на шаге №1. Утверждается, что каждый запуск обхода в глубину на шаге №3 будет обходить очередную компоненту сильной связности и только ее.

27.3. Доказательство алгоритма

> Мы настоятельно рекомендуем читателю вникнуть в это доказательство, поскольку оно помогает лучше разобраться со свойствами сильной связности и конденсации, а кроме того, дает более глубокое понимание алгоритма обхода в глубину.

27.3.1. Времена входа и выхода обхода в глубину

Для удобства введем вспомогательные определения. Предположим, что мы произвели серию обходов в глубину заданного графа.

Выпишем вершины в той последовательности, в которой обход в глубину **впервые заходил** в них. Тогда **временем входа** для каждой вершины v называется индекс этой вершины в указанной последовательности. Мы будем обозначать его через $t_{in}[v]$.

Аналогично, выпишем порядок, в котором обход в глубину **покидал** каждую из вершин, и назовем **временем выхода** $t_{out}[v]$ индекс вершины v в этой последовательности.

Рассмотрим пример. Пусть дан указанный граф, и на нем была запущена серия обходов в глубину: сначала обход из вершины №1 и затем обход из вершины №5. Мы считаем для определенности, что обход в глубину просматривает ребра в порядке увеличения номеров вершин. Тогда времена входа и выхода для каждой вершины будут следующими (рис. 27.4).

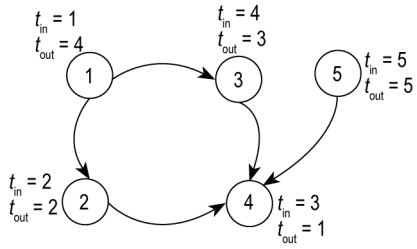

Рис. 27.4. Пример времен входа и выхода для указанного графа

27.3.2. Теорема о связи конденсации с временами выхода

Обозначим через $t_{out}[C]$ максимум из $t_{out}[v]$ по всем вершинам v исходного графа, входящим в компоненту сильной связности C. Тогда утверждается, что для каждого ребра (C_1, C_2) графа-конденсации справедливо:

$$t_{out}[C_1] > t_{out}[C_2].$$

Доказательство

Для доказательства рассмотрим момент, когда обход в глубину впервые достиг одну из вершин C_1 или C_2 — обозначим эту вершину через v. Есть два возможных варианта:

4. Первой была достигнута вершина компоненты C_1, т. е. $v \in C_1$. Поскольку, по определению компоненты связности, из v есть пути до всех остальных вершин, принадлежащих C_1, то запуск обхода в глубину из v обойдет всю компоненту C_1, и, следовательно, $t_{out}[C_1] = t_{out}[v]$. С другой стороны, поскольку из C_1 есть ребро в C_2, то тот же самый запуск обхода в глубину из v обойдет и всю компоненту C_2. Следовательно, $t_{out}[C_1] > t_{out}[C_2]$ (рис. 27.5).

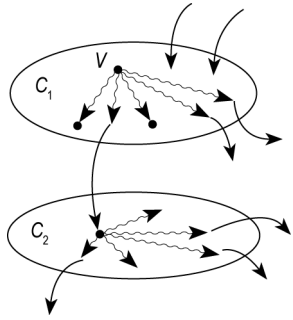

Рис. 27.5. В случае когда первой достигнута C_1,
то в процессе ее обхода будет обнаружено ребро, ведущее в C_2, откуда следует,
что и вся эта компонента будет посещена до выхода из компоненты C_1

5. Другой возможный вариант — $v \in C_2$. С одной стороны, этот запуск обхода в глубину обойдет всю компоненту C_2, и, следовательно, $t_{out}[C_2] = t_{out}[v]$. С другой

стороны, из наличия ребра (C_1, C_2) и ацикличности конденсации следует, что ни одной вершины C_1 при этом достигнуто не будет. Следовательно, вершины C_1 будут посещены после завершения обхода C_2, откуда снова следует $t_{out}[C_1] > t_{out}[C_2]$ (рис. 27.6).

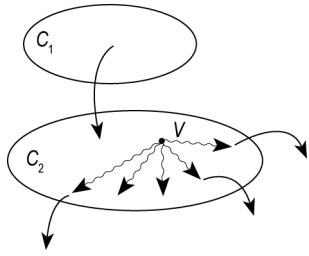

Рис. 27.6. В случае когда первой достигнута C_2,
в процессе ее обхода никакая из вершин C_1 достигнута быть не может

Применение

Из только что доказанной теоремы вытекает важное свойство: если выбрать в качестве v вершину с наибольшим значением $t_{out}[v]$, то в конденсации в соответствующую v компоненту сильной связности не будет входить ни одного ребра. Это свойство мы будем использовать на второй стадии алгоритма, для того чтобы выделить одну из компонент сильной связности.

> Отметим, что симметричное свойство в отношении времен входа t_{in} **не** наблюдается. Как наименьшая, так и наибольшая по времени входа вершины могут оказаться как в «пограничных» вершинах, так и во «внутренних» вершинах конденсации: в каких-то случаях могут быть входящие в них ребра, в других — исходящие.

27.3.3. Обратный граф и его конденсация

Напомним, что обратный граф равен исходному графу, в котором направления всех ребер заменены на противоположные. С точки зрения конденсации есть важная связь между исходным и обратным графами:

1. Обратный граф содержит **те же самые компоненты сильной связности**, что и исходный граф. Это следует из того, что хотя обращение и меняет направление всех путей на противоположное, на истинность утверждений вида «между вершинами i и j есть пути / нет путей в обе стороны» это не влияет.
2. Конденсация обратного графа равна **обратному графу конденсации** исходного графа. Иными словами, для каждого ребра (C_1, C_2) исходной конденсации есть ребро (C_2, C_1) в конденсации обратного графа, и наоборот.

Проиллюстрируем эти свойства примером (рис. 27.7).

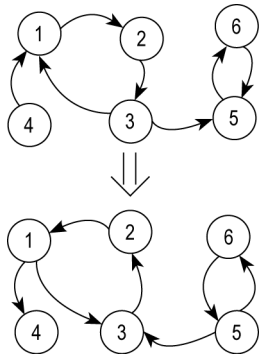

Рис. 27.7. Пример графа и его конденсации. Их конденсации состоят из одних и тех же вершин: (1, 2, 3), (4), (5, 6). Ребра в конденсациях тоже одни и те же, отличаясь только направлением

Применение этих свойств в алгоритме позволяет, запустив обычный обход в глубину на обратном графе из вершины с наибольшим t_{out}, обойти все вершины этой компоненты и только их. В самом деле, если в исходной конденсации в такую компоненту не входило ни одного ребра (что верно на основе теоремы из предыдущего раздела), то в конденсации обратного графа из этой компоненты не выходит ни одного ребра.

27.3.4. Итоговое обоснование алгоритма

Объединяя только что доказанные факты, мы получаем, что обход в глубину на обратном графе, запущенный из вершины с наибольшим t_{out}, выделит одну из компонент сильной связности.

Аналогичное верно и для запуска обхода в глубину из следующей по убыванию не посещенной пока вершины. Корректность этого шага базируется на тех же самых утверждениях, после того как мы мысленно удаляем первую компоненту сильной связности из рассмотрения.

Повторяя те же самые рассуждения, мы приходим к тому, что алгоритм Косарайю-Шарира действительно правильно обнаружит все компоненты сильной связности.

27.4. Демонстрация работы алгоритма

Теперь, когда мы разобрались с тем, **почему** алгоритм выполняет именно такие, на первый взгляд неожиданные, операции, можно проследить его работу на конкретном примере.

Пусть дан следующий граф (рис. 27.8).

В первой серии обходов в глубину сначала произойдет запуск из вершины №1, посещающий вершины №1, 2, 3, 5, 6, и затем запуск из вершины №4, посещающий вершину №4 (рис. 27.9).

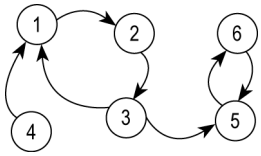

Рис. 27.8. Пример графа

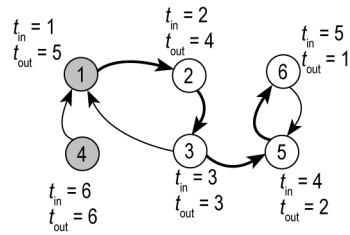

Рис. 27.9. Времена входа и выхода после серии обходов в глубину в указанном графе

Если упорядочить вершины по убыванию t_{out}, то получится следующая последовательность: 4, 1, 2, 3, 5, 6.

После этого мы строим обратный граф и запускаем вторую серию обходов в глубину в порядке убывания t_{out}. При запуске из вершины №4 будет посещена только сама эта вершина; при запуске из вершины №1 будут посещены вершины №1, 2, 3; при запуске из вершины №5 будут посещены вершины №5, 6 (рис. 27.10).

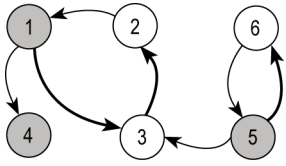

Рис. 27.10. Работа обхода в глубину в обратном графе, когда стартовые вершины рассматриваются в найденном ранее порядке

Таким образом, алгоритм правильно выделил все компоненты сильной связности: (4), (1, 2, 3) и (5, 6).

27.5. Временная сложность алгоритма

Алгоритм представляет собой две серии обходов в глубину. Поскольку каждая серия обходов посещает все вершины и просматривает все ребра по одному разу, итоговая асимптотика составляет $O(n + m)$, где n — число вершин, m — число ребер.

Иными словами, описанный алгоритм работает **за линейное время**.

27.6. Реализация алгоритма

Для определенности мы приведем программу, которая считывает заданный в виде списка ребер граф, выделяет в нем компоненты сильной связности и выводит для каждой вершины исходного графа номер компоненты сильной связности, в которую она попала.

Реализация использует представление графов в виде списков смежности.

Python

```
def dfs1(vertex):
  visited[vertex] = True
  for to in adj_list[vertex]:
    if not visited[to]:
      dfs1(to)
  order.append(vertex)

def dfs2(vertex):
  component[vertex] = \
      component_count
  for to in inv_adj_list[vertex]:
    if component[to] is None:
      dfs2(to)

vertex_count, edge_count = map(
    int, input().split())
adj_list = [
  [] for _ in range(vertex_count)]
inv_adj_list = [
  [] for _ in range(vertex_count)]
for _ in range(edge_count):
  vertex_from, vertex_to = map(
      int, input().split())
  vertex_from -= 1
  vertex_to -= 1
  adj_list[vertex_from].append(
      vertex_to)
  inv_adj_list[vertex_to].append(
      vertex_from)

visited = [False] * vertex_count
order = []
for vertex in range(vertex_count):
  if not visited[vertex]:
    dfs1(vertex)
component = [None] * vertex_count
component_count = 0
for vertex in reversed(order):
  if component[vertex] is None:
    dfs2(vertex)
    component_count += 1

print(*[i + 1 for i in component])
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>>
    adj_list, inv_adj_list;
vector<char> visited;
vector<int> order, component;
int component_count;

void Dfs1(int vertex) {
  visited[vertex] = true;
  for (int to : adj_list[vertex]) {
    if (!visited[to])
      Dfs1(to);
  }
  order.push_back(vertex);
}

void Dfs2(int vertex) {
  component[vertex] =
      component_count;
  for (int to :
       inv_adj_list[vertex]) {
    if (component[to] == -1)
      Dfs2(to);
  }
}

int main() {
  int vertex_count, edge_count;
  cin >> vertex_count
      >> edge_count;
  adj_list.resize(vertex_count);
  inv_adj_list.resize(
      vertex_count);
  for (int i = 0; i < edge_count;
       ++i) {
    int vertex_from, vertex_to;
    cin >> vertex_from
        >> vertex_to;
    --vertex_from;
    --vertex_to;
```

```
    adj_list[vertex_from]
        .push_back(vertex_to);
    inv_adj_list[vertex_to]
        .push_back(vertex_from);
  }

  visited.assign(
      vertex_count, false);
  order.clear();
  for (int v = 0; v < vertex_count;
       ++v) {
    if (!visited[v])
      Dfs1(v);
  }
  reverse(order.begin(),
          order.end());
  component.assign(
      vertex_count, -1);
  component_count = 0;
  for (int v : order) {
    if (component[v] == -1) {
      Dfs2(v);
      ++component_count;
    }
  }

  for (int v = 0; v < vertex_count;
       ++v) {
    cout << component[v] + 1
         << ' ';
  }
}
```

Примечание по использованию глобальных переменных

Самая лаконичная реализация алгоритма поиска компонент сильной связности получается при использовании глобальных переменных. Разумеется, ни в промышленном программировании, ни при решении сложных задач, работающих сразу с несколькими графами, такой подход годиться не будет. Однако во многих задачах спортивного программирования входной граф только один, да и все остальные данные, используемые алгоритмом обхода в глубину, удобнее не передавать явно в виде параметров функций.

Отсутствие массива visited во втором обходе в глубину

Во втором обходе в глубину нет нужды использовать массив `visited`, поскольку его роль выполняет отсутствие или наличие значения в массиве номеров компонент связности `component`.

27.7. Дополнительные свойства алгоритма

Отметим, что алгоритм будет находить компоненты сильной связности **в топологическом порядке**.

Это следует из того, что второй обход в глубину запускается из вершины с наибольшим временем выхода, и при этом он не покидает вершин этой компоненты.

27.8. Пример решения задачи. Железнодорожный вокзал

Задача. В городе N строится железнодорожный вокзал. Из-за нехватки места архитектор пытается определить, какое наименьшее число железнодорожных путей требуется на вокзале. Всего на вокзал будут прибывать k поездов, однако не все из них должны находиться на вокзале одновременно. Известен набор ограничений вида (a_i, b_i), которые обозначают, что поезд с номером a_i должен прибыть на вокзал не позднее поезда b_i (в частности, это делается для того, чтобы пассажиры одного поезда могли пересесть в другой). Архитектор понял, что, несмотря на все его попытки разнести прибытие поездов по времени, из-за этих ограничений возможны ситуации, когда сразу несколько поездов должны одновременно находиться на вокзале. Помогите архитектору определить минимально необходимое число железнодорожных путей на вокзале.

Входные данные состоят из числа поездов k и числа ограничений m в первой строке ($1 \le k, m \le 10^5$) и пар чисел «a_i b_i» в последующих m строках ($1 \le a_i, b_i \le n$, $a_i \ne b_i$). Вывести требуется единственное число — искомое минимальное число путей.

Примеры

Входные данные	**Требуемый результат**
4 4 1 2 2 3 3 4 4 2	3
Входные данные	**Требуемый результат**
4 4 1 2 1 3 2 4 3 4	1

Пояснения к примерам

В первом примере поезда №2, 3 и 4 должны находиться на вокзале одновременно, поэтому требуется минимум три пути; поезд №1 может прибыть и уехать заранее. Во втором примере все поезда могут прибывать на вокзал по одному.

Решение

Заметим, что если граф ограничений является сильно связным, то все поезда должны находиться на вокзале одновременно.

Аналогично, если в графе есть несколько компонент сильной связности, то для каждой компоненты все соответствующие ей поезда должны находиться на вокзале одновременно. При этом поезда разных компонент можно разнести друг относительно друга по времени.

Таким образом, ответ на задачу равен числу вершин в самой большой компоненте сильной связности. Асимптотика решения — $O(k + m)$.

27.9. Пример решения задачи. Задача умозаключенного

Задача. Умозаключенный Вася разрабатывает математическую теорию для изобретенной им области математики. В силу своей умозаключенности он пытается свести эти теорию к минимальному набору теорем, из которых простыми, по его мнению, умозаключениями можно вывести все остальные теоремы его теории.

Входные данные состоят из пары чисел n и m в первой строке и пар чисел a_i и b_i в последующих m строках: n — число теорем, m — простых умозаключений, из теоремы a_i можно вывести простым умозаключением теорему b_i. Ограничения: $1 \le n \le 10^5$, $0 \le m \le 10^5$, $1 \le a_i, b_i \le n$, $a_i \ne b_i$.

Вывести требуется количество выбранных теорем в первой строке и номера самих теорем в порядке возрастания во второй строке. Если решений несколько, разрешается выбрать любое.

Примеры

Входные данные	**Требуемый результат**
4 3 1 2 2 3 3 1	2 1 4
Входные данные	**Требуемый результат**
3 2 1 3 2 3	2 1 2

Пояснения к примерам

В первом примере из теоремы №1 «простыми умозаключениями» выводятся теоремы №2 и 3, а теорема №4 не выводится ни из какой другой. Во втором примере теорема №3 выводится как из теоремы №1, так и теоремы №2, но в ответ требуется взять их обе.

Решение

В терминах теории графов, задача — выбрать наименьшее число вершин орграфа так, чтобы все остальные вершины были достижимы из выбранных.

Первой мыслью может быть рассмотреть вершины, в которые **не входит ни одного ребра** (т. е. вершины, имеющие степень входа, равную нулю). Очевидно, что все такие вершины следует включить в ответ, потому что в них нельзя прийти по ребрам графа ни из каких других вершин.

Однако это решение было бы недостаточным: уже в первом примере из условия наблюдается цикл, когда в каждую из вершин цикла входит по одному ребру, и тем не менее мы обязаны выбрать одну из вершин цикла. Однако если мы мысленно сожмем весь цикл в одну вершину, то рассуждения из предыдущего параграфа станут применимы и здесь: в цикл не входит ни одного ребра извне, значит, мы обязаны взять в ответ одну из вершин цикла.

В этот момент становится ясно, что решение наверняка связано с **компонентами сильной связности и конденсацией** графа. В самом деле, с одной стороны, внутри одной компоненты сильной связности не имеет смысла выбирать более одной вершины, поскольку из каждой вершины компоненты достижимы все ее остальные вершины. С другой стороны, в каждой компоненте сильной связности, имеющей нулевую степень входа, мы обязаны выбрать по одной вершине.

Можно понять, что выбранных такой стратегией вершин достаточно, для того чтобы все остальные вершины были достижимы. Таким образом, алгоритм решения задачи имеет вид:

1. Выделить компоненты сильной связности и построить конденсацию.
2. Ответ равен числу вершин в конденсации, не имеющих входящих ребер.

Асимптотика решения — $O(n + m)$.

27.10. Пример решения задачи. Сбор дани

Задача. Команда историков занимается исследованиями экономических отношений между княжеской администрацией и подконтрольными ей городами. Удалось восстановить карту возможных маршрутов, а также размер дани, которую должен был выплатить каждый город. Известно, что княжеская дружина отправилась из города S в город T, однако конкретный путь их следования неизвестен. Историки предполагают, что дружина следовала маршрутом, который максимизирует суммарный сбор дани — при этом, однако, если дружина посещала какой-либо город повторно,

собирать дань второй раз они там уже не могли. Помогите историкам найти, каким мог быть максимально возможный сбор.

Первая строка входных данных содержит число городов N ($2 \le N \le 10^5$), число маршрутов M ($1 \le M \le 10^5$), номера S и T ($1 \le S, T \le N$, $S \ne T$). Во второй строке записаны N целых чисел W_i — размеры дани для каждого из городов ($1 \le W_i \le 1000$). Последующие M строк описывают дороги: каждая строка содержит по два числа A_i и B_i ($1 \le A_i, B_i \le N$, $A_i \ne B_i$), обозначающих, что из города A_i можно добраться в город B_i. Отметим, что маршруты являются однонаправленными (например, потому что это могли быть водные маршруты по быстрым рекам).

Вывести требуется единственное число — искомый максимально возможный сбор дани вдоль оптимального пути. Гарантируется, что из S в T можно добраться по указанным маршрутам.

Примеры

Входные данные	**Требуемый результат**
4 4 1 4 1 1 1 1 1 2 2 3 3 1 1 4	4
Входные данные	**Требуемый результат**
4 4 1 3 1 2 3 4 1 2 1 4 2 3 4 3	8

Пояснения к примерам

В первом примере оптимальным путем является путь 1-2-3-1-4; дань при этом собирается со всех городов. Во втором примере оптимально двигаться вдоль пути 1-4-3, что позволяет собрать дань со всех городов, кроме города №2.

Решение

Если переформулировать условие в терминах теории графов, требуется найти такой путь между двумя заданными вершинами орграфа, который бы максимизировал сумму весов входящих в него вершин. Путь при этом не обязан быть простым, а посещенные вершины учитываются только один раз.

Схожесть с NP-полными задачами?

На первый взгляд кажется, что поставленная задача очень похожа на классические графовые задачи, эффективные решения которых науке неизвестны: например, на задачу о длиннейшем пути. Однако рассматриваемая здесь задача имеет существенные отличия: путь не обязан быть простым, а веса вершин учитываются только единожды. Это позволяет найти эффективный алгоритм для ее решения.

Одной из «наивных» попыток решения этой задачи может быть следующее: просуммировать веса всех вершин, которые достижимы из S и из которых достижима T. Однако уже на втором примере из условия это решение выдаст неверный ответ: это тест-«ромб», в котором мы должны выбрать сторону, по которой мы спускаемся от одной вершины до противоположной ей (мы можем пройти по каждой из сторон, но не по обеим из них одновременно).

Начнем решение с наблюдения: если граф является **сильно связным**, то ответом на задачу будет сумма весов всех вершин. Это следует из того, что в сильно связном графе из любой вершины можно дойти до любой другой, и из того, что по условию нам не требуется минимизировать длину пути.

Аналогичное утверждение верно в случае, когда S и T лежат **в одной компоненте сильной связности**: в таком случае ответ будет равен сумме весов вершин этой компоненты. Ответ не может быть больше, поскольку, по свойству графа-конденсации, если бы мы вышли из компоненты сильной связности по какому-либо ребру, вернуться назад в эту же компоненту мы уже не смогли бы.

Последний оставшийся вариант — когда S и T лежат **в разных компонентах сильной связности**. В таком случае рассмотрим граф-конденсацию. Если мы обозначим через C_S и C_T компоненты, которым принадлежат вершины S и T, то наша задача — найти самый выгодный путь из C_S в C_T. Выгодность пути при этом определяется посещенными компонентами сильной связности, поскольку, попав в какую-либо компоненту сильной связности, мы можем обойти все ее вершины.

Кажется, что мы пришли к такой же проблеме, что и исходная задача, однако есть существенное отличие: теперь мы имеем дело с **ациклическим** графом (ведь граф-конденсация не может содержать циклов). Это означает, что решать задачу поиска самого дорогого пути можно с помощью **динамического программирования**: пусть $d[C]$ — это стоимость самого выгодного пути от компоненты C до компоненты C_T. Это значение можно выразить через значения этой же динамики по всех исходящим из C ребрам:

$$d[C] = w[C] + \max_{(C,U)\in G} d[U],$$

где $w[C]$ — сумма весов вершин, входящих в компоненту C, G — это граф-конденсация. Базой динамики является:

$$d[C_T] = w[C_T].$$

Эти формулы позволяют вычислить значение динамики для любой компоненты. Ответом на задачу будет $d[C_S]$.

Отметим, что это решение на основе динамического программирования верно и в том случае, когда весь граф сильно связен и/или S и T лежат в одной компоненте сильной связности. Наше описание решения следовало обычному мыслительному процессу: сначала стоит найти интересные частные случаи и попытаться решить их, затем перейти к решению оставшихся сложных случаев, и в конце попытаться обобщить найденный алгоритм на все случаи.

Решение

Для удобства реализации мы немного изменим решение, развернув динамическое программирование наоборот: будем полагать $d[C]$ равным стоимости самого выгодного пути от компоненты C_S до компоненты C. Подсчет такого динамического программирования будет вестись по обратному графу. Это позволяет получить чуть более компактную реализацию, поскольку компоненты сильной связности обнаруживаются в порядке топологической сортировки, а, значит, мы можем вычислить значение динамики для текущей компоненты сразу после ее обнаружения.

Кроме того, мы не перенумеровываем компоненты сильной связности, а используем для их идентификации номер вершины, из которой мы начинали ее обход вторым обходом в глубину. В реализации это обозначается массивом `root`.

Python

```
def dfs1(v):
  u[v] = True
  for to in g[v]:
    if not u[to]:
      dfs1(to)
  order.append(v)

def dfs2(v, cur_root):
  root[v] = cur_root
  comp.append(v)
  for to in ginv[v]:
    if root[to] is None:
      dfs2(to, cur_root)

def best_d(comp):
  best = None
  for v in comp:
    for to in ginv[v]:
      if root[to] == root[v]:
        continue
      nd = d[root[to]]
      if nd is None:
        continue
      if best is None or nd > best:
```

C++

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>> g, ginv;
vector<char> u;
vector<int> order, root, comp, d;

void Dfs1(int v) {
  u[v] = true;
  for (int to : g[v]) {
    if (!u[to])
      Dfs1(to);
  }
  order.push_back(v);
}

void Dfs2(int v, int cur_root) {
  root[v] = cur_root;
  comp.push_back(v);
  for (int to : ginv[v]) {
    if (root[to] == -1)
      Dfs2(to, cur_root);
```

```
        best = nd
  return best

n, m, s, t = map(
    int, input().split())
s -= 1
t -= 1
price = list(map(
    int, input().split()))
g = [[] for _ in range(n)]
ginv = [[] for _ in range(n)]
for _ in range(m):
  v1, v2 = map(int, input().split())
  v1 -= 1
  v2 -= 1
  g[v1].append(v2)
  ginv[v2].append(v1)

u = [False] * n
order = []
for v in range(n):
  if not u[v]:
    dfs1(v)
root = [None] * n
d = [None] * n
for v in reversed(order):
  if root[v] is not None:
    continue
  comp = []
  dfs2(v, v)
  d[v] = best_d(comp)
  if d[v] is None and s in comp:
    d[v] = 0
  if d[v] is None:
    continue
  for i in comp:
    d[v] += price[i]
print(d[root[t]])
```

```
  }
}

int BestD() {
  int best = -1;
  for (int v : comp) {
    for (int to : ginv[v]) {
      if (root[to] == root[v])
        continue;
      int nd = d[root[to]];
      if (nd == -1)
        continue;
      if (best == -1 || nd > best)
        best = nd;
    }
  }
  return best;
}

int main() {
  int n, m, s, t;
  cin >> n >> m >> s >> t;
  --s;
  --t;
  vector<int> price(n);
  for (int& i : price)
    cin >> i;
  g.resize(n);
  ginv.resize(n);
  for (int i = 0; i < m; ++i) {
    int v1, v2;
    cin >> v1 >> v2;
    --v1;
    --v2;
    g[v1].push_back(v2);
    ginv[v2].push_back(v1);
  }

  u.resize(n);
  for (int v = 0; v < n; ++v) {
    if (!u[v])
      Dfs1(v);
  }
  reverse(order.begin(),
          order.end());
```

```
  root.assign(n, -1);
  d.assign(n, -1);
  for (int v : order) {
    if (root[v] != -1)
      continue;
    comp.clear();
    Dfs2(v, v);
    d[v] = BestD();
    if (d[v] == -1 && root[s] == v)
      d[v] = 0;
    if (d[v] == -1)
      continue;
    for (int i : comp)
      d[v] += price[i];
  }
  cout << d[root[t]] << endl;
}
```

Глава 28.
Работа с вещественными числами

Современные процессоры в своем большинстве используют так называемые числа с плавающей запятой (*англ.* floating-point numbers) для представления вещественных чисел. Это же представление используется и во многих языках программирования, включая Python и C++.

При всех своих преимуществах представление в виде чисел с плавающей запятой имеет много неожиданных и контринтуитивных свойств. Не претендуя на полноту изложения, в этой главе мы дадим краткий обзор этого формата, типичных проблем, связанных с ним, а также стандартных приемов для преодоления этих сложностей.

28.1. Формат чисел с плавающей запятой

Хотя программисту не требуется вручную переводить числа в формат с плавающей запятой и не требуется работать с битами этого представления, тем не менее полезно в общих чертах представлять себе его устройство — без этого понять его проблемы будет невозможно.

28.1.1. Описание формата

Основная идея формата довольно проста. Вместо того чтобы хранить целую и дробную части вещественного числа, будем использовать следующее представление:

$$m \cdot 2^n,$$

где число n называется **экспонентой** и является целым, а вещественное число m называется **мантиссой** и либо равно нулю, либо удовлетворяет условию:

$$1 \le |m| < 2.$$

И мантисса, и экспонента хранятся в фиксированном объеме памяти; это означает, что для мантиссы сохраняется лишь какое-то количество цифр после запятой в двоичном представлении, а все остальные цифры мантиссы **отбрасываются**.

Альтернативные форматы

В целях простоты изложения мы описываем самый распространенный в реализациях языков программирования формат; этот формат описан в стандарте IEEE-754, наряду с

несколькими другими форматами. Этот формат является частным случаем **экспоненциальной записи** чисел, которая может использовать и другие параметры, например, десятичное основание (т. е. возводя не два, а десять в степень экспоненты) или другой диапазон для параметра m.

Примеры представления чисел в описанном формате:

- $0{,}5 = 1 \cdot 2^{-1}$;
- $0{,}1 = 1{,}6 \cdot 2^{-4}$;
- $0{,}01 = 1{,}28 \cdot 2^{-7}$;
- $12 = 1{,}5 \cdot 2^{3}$.

Название формата «плавающей запятой»

Название этого формата происходит от противопоставления его форматам с фиксированной запятой, например, таким, которые хранят вещественные числа всегда с двумя знаками после запятой (т. е., фактически, в виде целого числа, получающегося домножением исходного на 100). «Плавающая» же запятая всегда оказывается в мантиссе после первой значащей цифры, независимо от позиции запятой в исходном числе. Кстати, иногда можно также встретить название «числа с плавающей точкой», которое является не вполне корректной калькой с английского языка: в русском языке целая и дробная части чисел разделяются запятой.

28.1.2. Преимущества формата с плавающей запятой

- **Эффективность** — главное преимущество этого формата. Оно заключается в том, что процессорные операции с такими числами выполняются достаточно быстро, и за время $O(1)$. В отличие от некоторых других форматов, время исполнения одной операции ограничено сверху константой, независимо от того насколько большим было исходное число и сколько в нем было цифр после запятой.
- Гибкость этого формата в том, что как очень большие, так и очень маленькие вещественные числа представляются в нем в достаточно компактной форме. Например, число $1\,099\,511\,627\,776 = 2^{40}$ хранится в виде $m = 1$ $n = 40$, а число $2^{-40} \approx 0{,}0000000000009095$ хранится в виде $m = 1$ $n = -40$.
- Сохранение определенного числа значащих цифр является следствием предыдущего свойства. Например, если мы имеем дело с числом 0,125, то если бы мы использовали формат с фиксированным хранением двух знаков после запятой, то тысячные доли были бы отброшены — приводя к ошибке округления, несмотря на то что в числе всего две значащих цифры; аналогичная ситуация и со слишком большими числами. Формат же с плавающей запятой, напротив, абстрагируется с помощью экспоненты от несущественных лидирующих нулей и делает акцент на сохранении значащих цифр.

28.1.3. Недостаток: ошибки округления

Большинство «неприятностей», связанных с этим форматом, так или иначе вызваны погрешностями и ошибками округления.

Напомним, что в формате с плавающей запятой значащие цифры исходного числа попадают в мантиссу m, и эта мантисса хранится в виде вещественного числа фиксированной длины. Если исходное число имело больше значащих цифр, чем их помещается в мантиссу, то все остальные цифры отбрасываются, тем самым приводя к **погрешности**.

Более того, погрешности имеют свойство накапливаться: если программа выполняет какую-то арифметическую операцию с числами, которые уже содержат некоторые погрешности, то результат тоже будет иметь погрешность, и в некоторых случаях значительно бóльшую, чем погрешности аргументов!

28.2. Проблемы чисел с плавающей запятой

Рассмотрим некоторые типичные проблемы, вызванные погрешностями округления.

28.2.1. Округление точных чисел

Простейший пример, в котором возникает погрешность, — это число 0,1. В экспоненциальной форме это число будет выглядеть $1{,}6 \cdot 2^{-4}$, т. е. $m = 1{,}6$, $n = -4$. Переведем оба параметра в двоичную систему счисления, которая, напомним, используется в типичных реализациях представления с плавающей запятой. Если с переводом целого числа n никаких проблем нет, то, как можно убедиться самостоятельно, вещественное число 1,6 в двоичной системе представимо лишь в виде периодической дроби:

$$m = 1{,}6_{10} = 1{,}1001100110011001\ldots_2$$

Таким образом, как это ни удивительно на первый взгляд, уже для хранения числа 0,1 возникает погрешность — это число не представимо точным образом в нашем формате!

Другой класс примеров — это числа, которые и в десятичной записи имеют **бесконечно длинную** запись. Это иррациональные числа — такие, как π, e, $\sqrt{2}$, и, кроме того, числа, которые ни в двоичной, ни в десятичной системе счисления не имеют конечной записи — такие, как 1 / 3.

Независимо от того сколько байтов памяти будет выделено под хранение числа с плавающей запятой, ни одно из таких чисел точно представлено быть не может. Впрочем, ошибка округления будет по меньшей мере убывать с увеличением размерности типа данных.

28.2.2. Отличие результата арифметической операции от точного ответа

Если арифметическая операция производится над числами с плавающей запятой, то результат ее может отличаться от истинного ответа, записанного в формате с плавающей запятой.

Самой простой **иллюстрацией** этой проблемы является следующая программа.

Python

```
if 0.1 * 3 == 0.3:
  print("OK")
else:
  print("BAD")
```

C++

```
#include <iostream>
using namespace std;
int main() {
  if (0.1 * 3 == 0.3)
    cout << "OK";
  else
    cout << "BAD";
}
```

Эти программы выведут «BAD» (по крайней мере, при использовании популярных интерпретаторов и компиляторов). Чем можно объяснить этот неожиданный результат?

Как было описано ранее, для числа 0,1 не существует точного представления в стандартном формате с плавающей запятой; аналогичная ситуация и с числом 0,3. При умножении числа 0,1 на 3 его погрешность, условно говоря, утраивается. Результат выражения `0.1 * 3` при использовании стандартных чисел с плавающей запятой будет равен:

0.30000000000000004440892098500 6…

Заметим, что эта проблема **не** связана конкретно с операцией умножения — то же самое значение будет иметь и выражение «0,1 + 0,2». Основная причина здесь в округлениях, происходящих при переводе из десятичного в двоичный формат, и в неспособности последнего сохранить точное значение бесконечной дроби.

«Проблема 0,3»

Указанный пример с числом 0,3 уже стал притчей во языцех. Благодаря своей наглядности он упоминается во всевозможных обсуждениях, связанных с числами с плавающей запятой. Энтузиасты даже создали веб-сайт **https://0.30000000000000004.com/**, который посвящен этой проблеме и приводит примеры из множества языков программирования.

28.2.3. Погрешности при выполнении арифметических операций

Даже в случае когда операнды точно представимы в формате с плавающей запятой, результат арифметической операции над ними может иметь ошибки округления.

Это является естественным следствием того, что мантисса числа сохраняет лишь не более определенного числа значащих цифр числа.

Такого рода округления весьма часто происходят на практике. Приведем для наглядности несколько произвольно выбранных примеров:

- значение выражения «15 / 11 * 11» при использовании стандартного формата с плавающей запятой равно 14.99999999999999822364…;
- «sqrt(2) * sqrt(2)» равно 2.00000000000000044409…

28.2.4. Накопление погрешностей

В предыдущих разделах мы видели, как небольшие ошибки округления возникают в процессе перевода чисел в формат с плавающей запятой и при выполнении какой-либо арифметической операции. Однако если программа содержит целую цепочку вычислений, в которой результат предыдущей операции в формате с плавающей запятой подается на вход следующей операции, то погрешности начинают накапливаться и суммироваться.

Проще всего **проиллюстрировать** этот эффект можно на примере программы, которая последовательно прибавляет число 0,1, т. е. вычисляет следующую последовательность:

$$s_0 = 0,$$
$$s_{i+1} = s_i + 0{,}1.$$

Если запустить такую программу и сравнить вычисленные ею (в числах с плавающей запятой) величины s_n с истинным ответом, равным n / 10, то погрешность будет изменяться следующим образом:

Величина *n*	10	10^2	10^3	10^4	10^5	10^6
Погрешность	$\approx 10^{-16}$	$\approx 10^{-14}$	$\approx 10^{-12}$	$\approx 10^{-10}$	$\approx 10^{-8}$	$\approx 10^{-6}$

По этим результатам видно, как пренебрежимо малая вначале погрешность постепенно накапливается и достигает заметных величин уже после сотен тысяч итераций.

> Внимательный читатель может задаться вопросом: почему погрешность в этом примере нарастает с вдвое большей скоростью, чем число итераций? Могло бы показаться, что поскольку на каждой итерации прибавляется одно и то же число 0,1 с его фиксированной ошибкой, то и погрешность результата должна была бы увеличиваться линейно. Однако у накапливающейся погрешности есть и другая причина, которую мы рассмотрим в следующем разделе.

28.2.5. Ухудшение погрешности для чисел разного порядка

При сложении сравнительно большого и сравнительно малого (по абсолютным величинам) чисел погрешность результата резко возрастает.

Причину этого эффекта легко понять, если вспомнить, что формат с плавающей запятой предполагает хранение лишь ограниченного количества значащих цифр мантиссы. Чем больше разница по величине между операндами, тем больше значащих цифр будет в истинном результате и тем больше этих цифр будет потеряно при записи мантиссы результата (мысленно это можно представить себе как «прикладывание» мантиссы малого числа к хвосту мантиссы большого числа).

Аналогичная проблема возникает с вычитанием и с другими арифметическими операциями.

Наглядным примером этого явления будет сложение чисел 2^{100} и 2^{-100}. Хотя оба этих числа точно представимы в стандартном формате с плавающей запятой ($m = 1$, $n = 100$ для первого числа и $m = 1$, $n = -100$ для второго), для точного хранения их суммы потребовалась бы мантисса следующего вида:

$$m = 1,000...0001,$$

где число нулей было бы равно 199. На практике мантиссы такой длины стандартные реализации вместить не могут, и поэтому произойдет округление — и итоговым результатом будет 2^{100}.

Это был патологический пример: одно из слагаемых было целиком отброшено как погрешность. На практике, конечно, операнды редко отличаются настолько сильно, однако аналогичный эффект потери младших разрядов будет наблюдаться и в таких случаях.

Неассоциативность операций с плавающей запятой

Эти примеры показывают, что вычисления с плавающей запятой не являются ассоциативными, т. е. порядок вычисления операций влияет на результат. Например, хотя результат выражения «$1 + 2^{100} - 2^{100}$» при вычислениях с плавающей запятой равен нулю, результат выражения «$1 + (2^{100} - 2^{100})$» будет равен единице. Хотя с математической точки зрения результатом должно было быть одно и то же число (1), на практике результат будет зависеть от порядка вычислений.

28.2.6. Погрешность при вычитании близких чисел

Проблемы будут возникать и при вычитании близких друг к другу по значению чисел.

В таком случае причиной потери точности будет тот факт, что при вычитании близких чисел будут взаимно уничтожаться нескольких первых значащих цифр их мантисс, и результат будет содержать меньше значащих цифр, чем их может вместить тип с плавающей запятой.

28.2.7. Значения NaN

Формат с плавающей точкой поддерживает несколько особых, нечисловых, значений:

- результатом деления на ноль будет значение «бесконечность» или «минус бесконечность»;
- результатом вычисления квадратного корня из отрицательного числа будет значение «NaN».

Эти значения также особые в том смысле, что арифметические операции с их участием также возвращают нечисловое значение, а результат операции сравнения не вполне определен (хотя типичные реализации считают значение «NaN» не равным ничему, даже самому себе).

28.2.8. Положительный и отрицательный нули

Одна из неожиданностей стандартного формата чисел с плавающей запятой — это наличие у нуля знака. Иными словами, существуют два отдельных значения: «+0» и «−0», и они кодируются разными битовыми представлениями.

С одной стороны, для операций сравнения эти два значения считаются равными друг другу: «+0 = −0». С другой стороны, при побитовом сравнении памяти значения «+0» и «−0» окажутся разными.

28.2.9. Повторно вычисленное значение может отличаться

Это сравнительно редкая проблема, отчасти вызванная абстракциями и оптимизациями, вводимыми компиляторами для языков высокого уровня, таких, как C++.

Ее можно условно проиллюстрировать кодом следующего вида, где `f()` — некоторая функция, вычисляющая и возвращающая число с плавающей запятой.

C++

```
double x = ...;
double y = f(x);
// ...
if (f(x) == x)
  cout << 1;
```

Этот код не обязательно выведет «1», как это ни покажется удивительным. И дело здесь не только в NaN: этот парадоксальный эффект может наблюдаться даже тогда, когда функция возвращает обычное вещественное число.

Дело здесь заключается в том, что для вычисления функции `f(x)` компилятор может, теоретически, использовать тип данных с бóльшей точностью, чем тип double, который округляется до размера double лишь в момент записи результата в оперативную память. В большинстве случаев это поведение не вызывает никаких проблем: наоборот, чем больше точности компилятор может «выжать» из конкретного процессора, тем лучше. Однако в сравнениях вида `f(x) == x` может возникнуть неожиданный побочный эффект: левая часть здесь может иметь более высокую точность, чем правая. Например, если компилятор произвел инлайнинг функции `f(x)`, то результат ее работы не будет записан в память и, как следствие, не округлен до размера double — что, в конечном итоге, приведет к парадоксальному, на первый взгляд, эффекту неравенства.

Нам неизвестно, наблюдается ли аналогичная проблема в языке Python, учитывая, что с точки зрения интерфейса языка в нем есть только один тип для работы с вещественными значениями с плавающей запятой и возможности по оптимизации сильно ограничены структурой языка. Потенциально проблема может появиться в оптимизирующих реализациях языка, таких, как PyPy, однако нам не удалось воспроизвести этой проблемы.

28.3. Приемы работы с числами с плавающей запятой

Из описанного выше видно, что работа с числами с плавающей запятой сопряжена с большим количеством трудностей.

Большинство проблем решаются несложными приемами, описанными ниже, однако иногда может потребоваться более детальный анализ программы и даже переработка решения (например, реорганизация формул).

28.3.1. Использовать вычисления в целых числах

Лучший способ избежать проблем чисел с плавающей запятой — это не использовать их.

Разумеется, не в каждой задаче это возможно. Однако, например, если в задаче идет речь о денежных суммах, имеющих не более чем фиксированное число k знаков после запятой, в программе лучше работать с суммами, домноженными на 10^k: это позволит гарантировать отсутствие погрешностей и прочих проблем.

Аналогичное верно в отношении геометрических задач: многие из этих задач можно решить в целых числах, хоть это не всегда и очевидно на первый взгляд.

28.3.2. Использование 64-битных типов с плавающей запятой

- В типичной реализации 64-битное число с плавающей запятой — это число с 53 битами, выделенными под хранение мантиссы.
  - 53 бита мантиссы обозначают, что она может вместить 2^{53} различных значений, что немногим меньше 10^{16}.
  - Следовательно, такой формат способен точно сохранить 15 значащих цифр.
- В языке C++ есть выбор между несколькими типами с плавающей запятой: float, double, long double.
  - Хотя Стандарт языка не гарантирует фиксированных размеров этих типов, на практике на типичных архитектурах со стандартными настройками компиляторов эти типы имеют размеры: float — 4 байта, double — 8 байтов, long double — 16 или 8 байтов.
  - Наша рекомендация: **по умолчанию использовать тип double**.

- В редких случаях, когда требуется повышенная точность, пусть даже ценой падения производительности, и среда предоставляет 16-байтный тип long double, может быть оправдано использование long double.
- Использовать тип float практически никогда смысла не имеет.

♦ В языке Python есть только один тип с плавающей запятой, поэтому при его использовании такой вопрос не стоит.

- Стандартом языка не дается гарантий о размере и точности типа float, однако на практике, как правило, это 64-битный тип.
- Отметим, что в Python есть и альтернативный вещественнозначный тип — decimal, который предоставляет повышенную и настраиваемую точность, а также содержит встроенное решение некоторых проблем чисел с плавающей запятой. Однако в контексте спортивного программирования этот тип применим сравнительно редко из-за низкой производительности операций над ним.

28.3.3. Сравнения с допусками

Из-за наличия погрешностей доверять результатам обычных операторов сравнения нельзя. Значения, истинные величины которых равны друг другу, могут оказаться на практике неравными друг другу, и наоборот.

Типичное решение этой проблемы — сравнение с допусками. Как правило, вводится фиксированная константа EPS с небольшим значением, которое должно превосходить любую возможную в данной программе погрешность.

Тогда операции сравнения с допуском, равным популярному выбору «10^{-9}», можно реализовать следующим образом.

Python

```
EPS = 1E-9

def smaller(a, b):
  return a < b - EPS
def equals(a, b):
  return abs(a - b) < EPS
def greater(a, b):
  return a > b + EPS
```

C++

```
const double EPS = 1E-9;

bool Smaller(double a, double b) {
  return a < b - EPS;
}
bool Equals(double a, double b) {
  return abs(a - b) < EPS;
}
bool Greater(double a, double b) {
  return a > b + EPS;
}
```

Можно сказать, что, по меньшей мере, в спортивном программировании почти всегда, когда сравниваются числа с плавающей запятой, должны использоваться сравнения с допусками.

У сравнения с допусками есть и одна неочевидная проблема: это нарушение транзитивности. Иными словами, при использовании сравнений с допусками не гарантируется, что из равенств «a=b» и «b=c» следует, что «a=c». В реалиях спортивного программирования эту потенциальную проблему можно не учитывать, поскольку по сути она возникает только тогда, когда допуск выбран неверно или задача вообще не имеет надежного решения в числах с плавающей запятой. В промышленном или научном ПО можно использовать специальные методы для гарантии транзитивности, например — хранение всех вещественных значений, отличающихся друг от друга более чем на величину допуска, в отдельной структуре данных.

Как выбрать значение допуска?

- Популярный выбор (в спортивном программировании и за его пределами) — это значение 10^{-9}.
- Произвести осмысленный выбор допуска для конкретной задачи — это, как следует из описанного ранее в этой главе, **очень** сложная задача, практически нереальная в реалиях спортивного программирования.
- Если есть подозрения на то, что выбранное значение допуска приводит в конкретной задаче к проблемам, можно эмпирически попробовать подобрать другие значения (например, 10^{-8}, 10^{-6} или 10^{-12}). Дать какие-то более конкретные рекомендации очень сложно в общем случае.
- В некоторых случаях может помочь переход на сравнение чисел по критерию «относительная или абсолютная разница не должна превосходить EPS»; однако такой критерий работает медленнее и имеет свои особенности, поэтому в общем случае применять его стоит с осторожностью.

28.3.4. Когда сравнения с допусками не нужны

Однако есть и исключения — случаи, в которых сравнения с допусками не нужны или даже могут приводить к потере точности и неверному ответу.

Например:

- сравнения с допусками, как правило, не нужны **в двоичном поиске** по вещественнозначной функции. При наличии погрешностей двоичный поиск автоматически подберет ближайшее пороговое значение; использование дополнительного допуска при сравнениях только сместит этот порог дальше от истинного значения;
- сравнения с допусками не нужны при проверках допустимости значений **при вызове стандартных функций**, таких, как `sqrt()`. Вычисление квадратного корня от отрицательного числа приводит к ошибке, даже если с точки зрения нашей программы это отрицательное число равно нулю (с допуском). Поэтому в таких случаях допуск использовать не следует.

28.3.5. Не следует сравнивать число побитово

Как упоминалось выше в перечне потенциальных проблем, результат сравнения чисел с плавающей запятой может отличаться от их побитового сравнения: например, из-за существования положительного нуля и отрицательного нуля.

На практике это означает, что, даже если нам требуется реализовать сравнение без допусков, практически нет ситуаций, в которых сравнивать числа побитово было бы корректно. Это относится как к сравнениям блоков в памяти (например, с помощью функции `memcmp()` в языке C++), так и к вычислениям хеш-функции на основе содержимого байтов памяти.

Глава 29.
Геометрия на плоскости. Основы

Несмотря на то что для реализации базовых примитивов вычислительной геометрии достаточно минимальных знаний из области математики, написание надежного, компактного и быстрого кода — не самая простая задача. Многие сложности проистекают от наличия всевозможных особых случаев (например, совпадающие или лежащие на одной прямой точки) и от стремления производить операции в целых числах (для избегания, по возможности, ошибок округления).

> Отметим, что приведенные ниже реализации ориентированы на спортивное программирование и на объяснение принципа работы алгоритмов. Повторное использование кода сведено к минимуму, и отсутствуют абстракции вида «отрезок» или «многоугольник», которые были бы в духе объектно-ориентированного или структурного программирования. Цель этой главы, как и книги в целом — **не** создать переиспользуемую библиотеку стандартных алгоритмов (что имеет мало смысла в реальном мире, где уже существует целый ряд тщательно спроектированных и отлаженных библиотек). Вместо этого акцент в нашей книге — на том, как устроены эти стандартные алгоритмы, из каких предпосылок они выводятся и каким образом можно быстро и компактно самостоятельно реализовать их для конкретной задачи.

29.1. Расстояние между точками

Для двух точек A и B с заданными координатами расстояние между ними можно найти по известной формуле: квадратный корень из суммы квадратов разностей абсцисс и разностей ординат.

Реализация

Python

```
from math import *
def distance(ax, ay, bx, by):
  return sqrt((ax - bx) ** 2 +
              (ay - by) ** 2)
```

C++

```
#include <cmath>
using namespace std;
double Distance(
    double ax, double ay,
    double bx, double by) {
  return sqrt(
    (ax - bx) * (ax - bx) +
    (ay - by) * (ay - by));
}
```

Библиотечная функция hypot()

Стандартные библиотеки C++ и Python также предоставляют функцию `hypot()`, которая вычисляет квадратный корень из суммы квадратов аргументов. Таким образом, можно было бы использовать эту функцию вместо возведения значений в квадрат и вызова `sqrt()`. Однако типичные реализации `hypot()` — несколько медленнее: они ориентированы на устранение погрешностей в «патологических» случаях, например когда величины аргументов сильно отличаются друг от друга. В контексте спортивного программирования проблемы погрешностей часто решаются другими способами (например, с помощью вычислений в целых числах — см. ниже), поэтому необходимость использования более медленной `hypot()` возникает редко.

В случае когда входные координаты — целочисленные, а решение может оперировать квадратом расстояния (например, для сравнения расстояний между разными парами точек), предпочтительнее перейти к вычислениям в целых числах и убрать извлечение квадратного корня. Тем самым мы избавимся от возможных погрешностей, а также заметно ускорим вычисление расстояний.

Реализация в целых числах

Python

```
def distance2(ax, ay, bx, by):
  return ((ax - bx) ** 2 +
          (ay - by) ** 2)
```

C++

```
int64_t Distance2(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by) {
  return (ax - bx) * (ax - bx) +
         (ay - by) * (ay - by);
}
```

29.2. Косое произведение векторов

Косое произведение векторов — один из основополагающих примитивов в вычислительной геометрии. Эта операция обозначается знаком $\wedge$ и определяется как произведение длин векторов и синуса угла между ними:

$$\vec{p} \wedge \vec{q} = |\vec{p}||\vec{q}| \sin \angle(\vec{p}, \vec{q}),$$

причем угол между векторами берется ориентированным, т. е. это угол вращения от $\vec{p}$ к $\vec{q}$ против часовой стрелки (рис. 29.1).

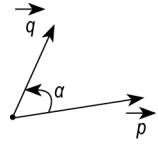

Рис. 29.1. Два вектора p и q и угол между ними в направлении против часовой стрелки

Терминология

Косое произведение также называют псевдоскалярным произведением. Иногда также употребляется термин «векторное произведение» — что не совсем точно, поскольку результатат векторного произведения есть также вектор; но подразумеваемая при этом аналогия основана на том, что векторное произведение равно вектору нормали, домноженному на косое произведение. В английском языке ситуация с терминологией также несколько запутана, поскольку термин cross product может обозначать как векторное, так и косое произведение — в зависимости от того, идет ли речь о трехмерном или двумерном пространстве.

Для косого произведения существует и другая, не менее важная, формула, которая использует знание координат векторов:

$$\vec{p} \wedge \vec{q} = \begin{vmatrix} p_x & p_y \\ q_x & q_y \end{vmatrix} = p_x q_y - p_y q_x.$$

В то время как последняя формула удобна в компьютерных вычислениях, приведенная ранее формула позволяет интерпретировать результат с геометрической точки зрения.

Реализация в целых числах
(для случая, когда входные координаты также целые)

Python

```
def cross_product(px, py, qx, qy):
  return px * qy - py * qx
```

C++

```
int64_t CrossProduct(
    int64_t px, int64_t py,
    int64_t qx, int64_t qy) {
  return px * qy - py * qx;
}
```

29.3. Скалярное произведение векторов

Скалярное произведение (*англ.* dot product) — это второй по важности примитив в вычислительной геометрии. Эта операция обозначается точкой (так же как умножение чисел), и она определена как произведение длин векторов и косинуса угла между ними:

$$\vec{p} \cdot \vec{q} = |\vec{p}||\vec{q}| \cos \angle(\vec{p}, \vec{q}),$$

причем угол здесь, в отличие от косого произведения, не обязательно брать в определенном направлении (это следует и из свойств косинуса от отрицательных углов).

Для скалярного произведения существует также простая формула на основе координат векторов:

$$\vec{p} \cdot \vec{q} = p_x q_x + p_y q_y.$$

Реализация в целых числах
(для случая, когда входные координаты также целые)

Python

```
def dot_product(px, py, qx, qy):
  return px * qx + py * qy
```

C++

```
int64_t DotProduct(
    int64_t px, int64_t py,
    int64_t qx, int64_t qy) {
  return px * qx + py * qy;
}
```

29.4. Площадь треугольника

Пусть требуется посчитать площадь треугольника с заданными координатами вершин A, B, C.

Самый простой способ решить эту задачу — воспользоваться известным фактом, что площадь треугольника равна половине произведения длин двух сторон на синус угла между ними:

$$S = \frac{1}{2} ab \sin \alpha.$$

Как легко заметить, эта формула практически совпадает с определением **косого произведения**, за исключением деления пополам и взятия неориентированного угла. Поскольку направление угла влияет лишь на знак, но не на величину синуса, мы получаем формулу:

$$S = \frac{1}{2} \left| \overrightarrow{AB} \wedge \overrightarrow{AC} \right|.$$

(В этой формуле пара векторов AB и AC была выбрана произвольно: результат будет тем же самым, если выбрать любую другую пару векторов, соответствующих сторонам треугольника.)

Реализация в целых числах

Напишем функцию, вычисляющую удвоенную площадь треугольника. Возврат удвоенной площади позволяет получить результат в целых числах и тем самым исключить возможные ошибки округления в дальнейших вычислениях; при необходимости (что требуется не в каждой задаче) результат затем можно разделить пополам.

Python

```
def triangle_area2(
    ax, ay, bx, by, cx, cy):
  return abs(cross_product(
```

C++

```
int64_t TriangleArea2(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by,
```

```
      bx - ax, by - ay,
      cx - ax, cy - ay))
```

```
    int64_t cx, int64_t cy) {
  return abs(CrossProduct(
      bx - ax, by - ay,
      cx - ax, cy - ay));
}
```

Для наглядности приведем и реализацию, не использующую функцию вычисления косого произведения и сразу выполняющую все вычисления в теле одной функции.

Python

```
def triangle_area2(
    ax, ay, bx, by, cx, cy):
  return abs(
      (bx - ax) * (cy - ay) -
      (cx - ax) * (by - ay))
```

C++

```
int64_t TriangleArea2(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by,
    int64_t cx, int64_t cy) {
  return abs(
      (bx - ax) * (cy - ay) -
      (cx - ax) * (by - ay));
}
```

Целочисленные переполнения в решении на C++

Приведенная реализация на языке C++ использует 64-битный целочисленный тип для уменьшения риска переполнений. В случаях когда известно, что входные координаты небольшие, в целях ускорения функцию можно изменить для использования 32-битных типов. Для оценки возможных значений координат заметим, что если координаты ограничены по модулю некоторой константой C, то максимально возможную удвоенную площадь треугольника и промежуточные величины можно оценить сверху величиной $4C^2$. Следовательно, можно оценить, что в случае 32-битного типа безопасным является диапазон $C \le 20\,000$, а в случае int64_t можно будет работать в диапазоне $C \le 1{,}5 \cdot 10^9$.

Формула Герона

Из математики известен и другой способ вычисления площади треугольника, основанный на знании длин сторон треугольника — так называемая формула Герона: $\sqrt{p(p-a)(p-b)(p-c)}$, где $p = (a+b+c)/2$. Однако применять эту формулу, вообще говоря, **не рекомендуется**, поскольку она включает существенно более сложные арифметические операции с использованием вещественных чисел — извлечение квадратного корня сначала для вычисления длин сторон и затем для итогового результата — и потому значительно более подвержена ошибкам округления. Применение формулы Герона оправдано лишь в том редком случае, когда известны длины сторон треугольника, но не координаты его вершин.

29.5. Направление поворота. Ориентированная площадь треугольника

Пусть дана тройка точек A, B, C и требуется определить, направлена она по часовой стрелке или против (или же точки лежат на одной прямой).

Альтернативная формулировка той же задачи — посчитать ориентированную площадь треугольника, которая по модулю равна обычной площади треугольника, а знак положителен или отрицателен в зависимости от порядка обхода, в котором даны вершины: положительный в случае обхода против часовой стрелки и отрицательный в случае обхода по часовой стрелке.

Для решения задачи воспользуемся, как и в предыдущем разделе, **косым произведением** векторов. Поскольку косое произведение равно произведению длин на синус ориентированного угла (взятого в направлении против часовой стрелки) и поскольку синус положителен для углов от 0 до 180 градусов и отрицателен для углов от 180 до 360 градусов, то по знаку косого произведения можно определить искомое направление. Также заметим, что косое произведение равно нулю тогда и только тогда, когда векторы лежат на одной прямой (включая случай, когда один из них имеет нулевую длину).

Реализация в целых числах
(предполагая, что координаты точек — также целые)

Python

```
def counterclockwise(
    ax, ay, bx, by, cx, cy):
  return cross_product(
      bx - ax, by - ay,
      cx - ax, cy - ay) > 0
```

C++

```
bool Counterclockwise(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by,
    int64_t cx, int64_t cy) {
  return CrossProduct(
      bx - ax, by - ay,
      cx - ax, cy - ay) > 0;
}
```

Приведем для наглядности и реализацию для случая, когда косое произведение в программе ни для чего другого не требуется, и потому можно использовать компактную реализацию в одну функцию.

Python

```
def counterclockwise(
    ax, ay, bx, by, cx, cy):
  return (
      (bx - ax) * (cy - ay) -
      (cx - ax) * (by - ay)) > 0
```

C++

```
bool Counterclockwise(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by,
    int64_t cx, int64_t cy) {
  return (bx - ax) * (cy - ay) -
         (cx - ax) * (by - ay) > 0;
}
```

29.6. Площадь многоугольника

Пусть дан некоторый многоугольник с вершинами в заданных точках; требуется определить его площадь. Предполагается, что многоугольник не имеет самопересечений.

Расширенный вариант этой задачи: требуется определить ориентированную площадь, т. е. площадь, домноженную на минус один, если вершины многоугольника даны в направлении обхода по часовой стрелке.

Для решения этой задачи переберем все стороны многоугольника и посчитаем для каждой стороны площадь трапеции, ограниченной этой стороной. Если быть точным, то выбирается координатная ось (можно выбрать любую — допустим, ось абсцисс), из каждой вершины многоугольника опускается перпендикуляр к этой оси, и, таким образом, можно образовать по одной трапеции из каждой стороны, двух перпендикуляров из ее вершин-концов и координатной оси (рис. 29.2).

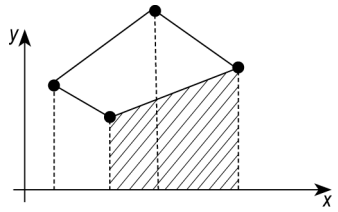

Рис. 29.2. Опустив перпендикуляры из каждой вершины многоугольника, мы образуем для каждой стороны многоугольника по одной трапеции. Заштрихована такая трапеция для одной из сторон

Утверждается, что искомая ориентированная площадь многоугольника — это сумма площадей этих трапеций, которые взяты со знаком «плюс», если порядок обхода на соответствующей стороне был справа налево, и со знаком «минус» в противном случае. В самом деле, при таком подходе «лишняя» площадь — площадь вне многоугольника — взаимно компенсируется, и останется только площадь самого многоугольника с нужным знаком.

Этот алгоритм проще всего понять по наглядной визуализации (рис. 29.3).

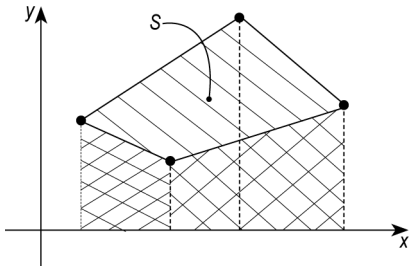

Рис. 29.3. При суммировании площадей трапеций, взятых с нужным знаком в зависимости от направления, области под многоугольником взаимно уничтожаются, а в итоге получается искомая площадь многоугольника

Осталось лишь вывести формулу для вычисления знаковой площади каждой трапеции:

$$S_i = (x_i - x_{i+1})(y_i + y_{i+1}) / 2,$$

тогда окончательный ответ будет равен:

$$S = \sum_{i=1}^{n} S_i,$$

где n — число вершин в многоугольнике. Отметим, что суммирование ведется до n включительно, так что последний элемент суммы будет образован вершинами номер n и номер 1.

Реализация в целых числах

Приведем реализацию вычисления ориентированной площади многоугольника. Площадь возвращается удвоенной, что позволяет избежать проблем, связанных с вычислениями с плавающей запятой.

Python

```
def polygon_area2(n, x, y):
  area = 0
  for i in range(n):
    next = (i + 1) % n
    area += ((x[i] - x[next]) *
             (y[i] + y[next]))
  return area
```

C++

```
int64_t PolygonArea2(
    int n,
    const vector<int64_t>& x,
    const vector<int64_t>& y) {
  int64_t area = 0;
  for (int i = 0; i < n; ++i) {
    int next = (i + 1) % n;
    area += (x[i] - x[next]) *
            (y[i] + y[next]);
  }
  return area;
}
```

29.7. Проверка точки на принадлежность прямой

Пусть дана точка P и прямая, заданная двумя различными точками A и B. Требуется определить, лежит ли точка P на прямой, проходящей через точки A и B.

Для реализации этого примитива достаточно свести его к рассмотренной ранее задаче: точка лежит на прямой тогда и только тогда, когда косое произведение соответствующих векторов равно нулю. Таким образом, достаточно вычислить косое произведение векторов AB и AP и сравнить результат с нулем.

Реализация косого произведения была приведена в *разд. 1.2*.

29.8. Проверка точки на принадлежность отрезку

Пусть дана точка P и отрезок AB. Требуется определить, лежит ли точка P на отрезке AB.

Для решения этой задачи заметим, что если точка лежит на отрезке, то она лежит и на прямой, проходящей через этот отрезок. Для невырожденных случаев — когда

точки A и B не совпадают — проверку точки на принадлежность прямой мы уже научились решать ранее.

Осталось научиться отсекать случаи, когда точка лежит на прямой, но при этом вне отрезка, а также обрабатывать вырожденный случай совпадающих точек. Можно понять, что для обоих этих случаев достаточно проверить, что абсцисса точки P лежит между абсциссами точек A и B и аналогично для ординаты.

Проверка на bounding box

Описанную выше проверку — то, что и абсцисса, и ордината точки лежат внутри промежутков абсцисс и ординат отрезка — часто сокращенно называют проверкой на bounding box (или проверкой через ограничивающий прямоугольник). Вообще говоря, bounding box для какой-либо геометрической фигуры — это минимальный прямоугольник со сторонами, параллельными осям координат, который вмещает в себя всю эту фигуру. В случае отрезка получаем, что его bounding box образован абсциссами и ординатами точек-концов отрезка; соответственно, «проверка на bounding box» обозначает здесь проверку, что точка лежит внутри этого прямоугольника.

Проиллюстрируем эти случаи (рис. 29.4).

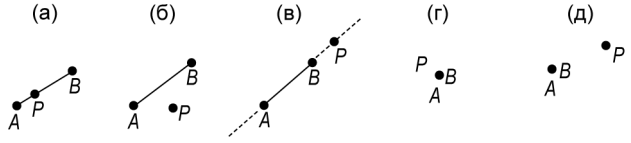

Рис. 29.4. Иллюстрация основных возможных случаев:
(а) точка, лежащая на отрезке; *(б)* точка, не лежащая на отрезке;
(в) точка, лежащая на прямой, образованной отрезком; *(г)* отрезок, вырожденный в точку, совпадающую с данной; *(д)* отрезок, вырожденный в точку, отличающуюся от данной

Таким образом, алгоритм состоит из комбинации двух критериев:

1. Проверка, что точка лежит на прямой, т. е. что косое произведение равно нулю.
2. Проверка, что точка лежит внутри bounding box отрезка.

Реализация в целых числах

Python

```
def point_on_segment(
    ax, ay, bx, by, px, py):
  return (
      within(ax, bx, px) and
      within(ay, by, py) and
      point_on_line(ax, ay, bx,
                    by, px, py))
def within(end1, end2, value):
  return (min(end1, end2) <= value
```

C++

```
bool PointOnLine(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by,
    int64_t px, int64_t py) {
  return
      (ax - px) * (by - py) -
      (bx - px) * (ay - py) == 0;
}
bool Within(
```

```
          <= max(end1, end2))
def point_on_line(
    ax, ay, bx, by, px, py):
  return (
      (ax - px) * (by - py) -
      (bx - px) * (ay - py) == 0)
```

```
    int64_t end1, int64_t end2,
    int64_t value) {
  if (end1 > end2)
    swap(end1, end2);
  return end1 <= value &&
         value <= end2;
}
bool PointOnSegment(
    int64_t ax, int64_t ay,
    int64_t bx, int64_t by,
    int64_t px, int64_t py) {
  return
      Within(ax, bx, px) &&
      Within(ay, by, py) &&
      PointOnLine(ax, ay, bx, by,
                  px, py);
}
```

Порядок выполнения проверок

Проверки критериев №1 и №2 можно было бы выполнять в любом порядке. Однако поскольку критерий №1 (проверка на принадлежность прямой) требует более сложных арифметических операций, при реализации имеет смысл проверять его в самом конце, что и сделано в наших реализациях.

29.9. Проверка двух отрезков на пересечение

Даны два отрезка AB и PQ и требуется определить, пересекаются ли они (т. е. имеют ли они хотя бы одну общую точку).

Попытаемся найти решение в целых числах, чтобы избежать возможных проблем с округлениями и точностью. Начнем решение задачи с рассмотрения базовых случаев, не заботясь пока о вырожденных примерах (когда какие-то точки совпадают) и прочих исключениях. Изобразим эти два **базовых** случая — первый для пересекающихся отрезков, второй для непересекающихся (рис. 29.5).

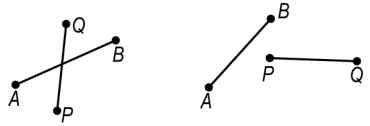

Рис. 29.5. Примеры пересекающихся (слева) и непересекающихся (справа) отрезков

Заметим, что в изображенном слева примере (это пример без пересечений) отрезок PQ **целиком лежит по одну сторону** от прямой AB. В изображенном же справа

примере, напротив, как точки P и Q лежат по разные друг от друга стороны от прямой AB, так и точки A и B лежат по разные стороны от прямой PQ (рис. 29.6).

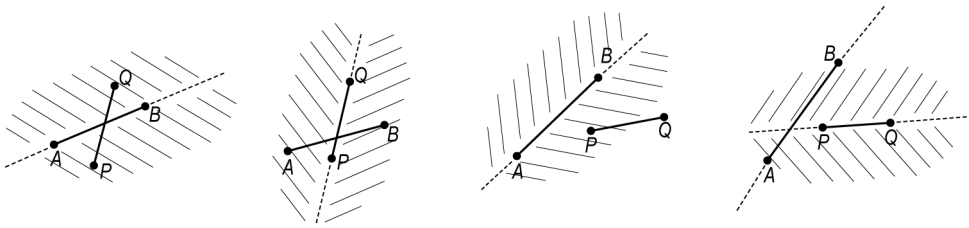

Рис. 29.6. Если провести через отрезок прямую, то в одних случаях другой отрезок будет пересекать эту прямую, а в других — целиком лежать по одну сторону от нее

Можно понять, что этот критерий работает, как минимум, в одну сторону: если A и B лежат по разные стороны от PQ и, одновременно, P и Q лежат по разные стороны от AB, то у отрезков есть пересечение. Реализовать этот критерий легко на основе примитива «направление поворота», рассмотренного нами в предыдущих разделах: достаточно лишь проверить, что косые произведения имеют разные знаки.

Осталось лишь рассмотреть **пограничные** случаи, когда какие-то три точки лежат на одной прямой. Для корректной обработки этих случаев нужны дополнительные критерии, поскольку ответ может быть как «есть пересечение», так и «нет пересечений» (рис. 29.7).

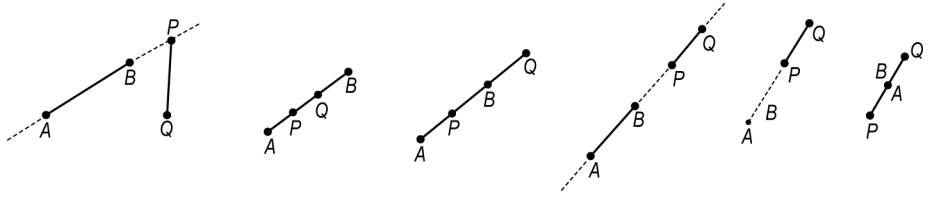

Рис. 29.7. Примеры случаев, которые могут возникнуть при наличии нескольких точек на одной прямой

Разнообразие вариантов этих пограничных случаев может поначалу пугать, однако можно заметить, что все эти случаи корректно решаются после следующих небольших уточнений:

- при проверке направлений поворотов будем засчитывать случай «точка лежит на прямой» эквивалентно случаю «точка лежит по обратную сторону»;
- дополнительно будем проверять, что ограничивающие прямоугольники (bounding box) первого отрезка и второго отрезка пересекаются. Иными словами, будем проверять, что промежуток абсцисс первого отрезка пересекается с промежутком абсцисс второго и то же самое для ординат.

Упражнение

Мы рекомендуем читателю самостоятельно разобрать каждый из проиллюстрированных выше примеров и убедиться, что указанная комбинация критериев всегда приводит к правильному ответу. На наш взгляд, после выполнения этого упражнения читатель не только детальнее поймет работу алгоритма, но и улучшит навыки интуитивного поиска «каверзных» случаев и простых способов их разрешения (эти умения особенно важны при решении геометрических задач).

Итак, **итоговый алгоритм** решения задачи выглядит следующим образом:

1. Проверить направления поворота с помощью косых произведений:

$$\left(\overrightarrow{AB} \wedge \overrightarrow{AP}\right) \cdot \left(\overrightarrow{AB} \wedge \overrightarrow{AQ}\right) \le 0,$$
$$\left(\overrightarrow{PQ} \wedge \overrightarrow{PA}\right) \cdot \left(\overrightarrow{PQ} \wedge \overrightarrow{PB}\right) \le 0.$$

2. Проверить, что bounding box двух отрезков пересекаются друг с другом.

Реализация в целых числах

Python

```
from collections import *

Point = namedtuple(
    'Point', ['x', 'y'])

def segments_intersect(a, b, p, q):
  return (
      intersects_1d(
          a.x, b.x, p.x, q.x) and
      intersects_1d(
          a.y, b.y, p.y, q.y) and
      direction(a, b, p) *
        direction(a, b, q) <= 0 and
      direction(p, q, a) *
        direction(p, q, b) <= 0)
def intersects_1d(a, b, p, q):
  left1 = min(a, b)
  right1 = max(a, b)
  left2 = min(p, q)
  right2 = max(p, q)
  return (max(left1, left2) <=
          min(right1, right2))
def direction(a, b, p):
  product = (
      (a.x - p.x) * (b.y - p.y) -
      (b.x - p.x) * (a.y - p.y))
  if product == 0:
```

C++

```
struct Point {
  int64_t x, y;
};
int Direction(
    Point a, Point b, Point p) {
  int64_t product =
      (a.x - p.x) * (b.y - p.y) -
      (b.x - p.x) * (a.y - p.y);
  if (product == 0)
    return 0;
  return product > 0 ? 1 : -1;
}
bool Intersects1D(
    int64_t a, int64_t b,
    int64_t p, int64_t q) {
  if (a > b)
    swap(a, b);
  if (p > q)
    swap(p, q);
  return max(a, p) <= min(b, q);
}
bool SegmentsIntersect(
    Point a, Point b,
    Point p, Point q) {
  return
      Intersects1D(
          a.x, b.x, p.x, q.x) &&
```

```
    return 0
return 1 if product > 0 else -1
```

```
        Intersects1D(
            a.y, b.y, p.y, q.y) &&
        Direction(a, b, p) *
          Direction(a, b, q) <= 0 &&
        Direction(p, q, a) *
          Direction(p, q, b) <= 0;
}
```

Тип данных Point

Реализации данного алгоритма — первые во всей главе, которые используют вспомогательный тип данных для хранения координат точки. Как уже упоминалось в начале главы, мы **не** пытаемся следовать традициям структурного или объектно-ориентированного программирования. Однако в данном случае при передаче каждой координаты по отдельности у функций стало бы слишком много аргументов, что увеличило бы шанс совершить опечатку.

Знаки косых произведений

Обращаем внимание, что в приведенных реализациях мы перемножаем не значения косых произведений, а лишь их знаки — этого достаточно для сравнения результата с нулем. Благодаря этому приему мы избегаем целочисленного переполнения в случае больших координат в решении на C++ и улучшаем производительность в решении на Python.

29.10. Расстояние от точки до прямой

Дана прямая, заданная двумя различными точками A и B, через которые она проходит, и точка P. Требуется определить расстояние от точки до прямой.

Для решения этой задачи рассмотрим треугольник ABP. Высота, опущенная из вершины P, равна искомому расстоянию. С другой стороны, высота связана с площадью треугольника известным соотношением:

$$S = \frac{1}{2}kh,$$

где k — длина стороны, на которую опущена высота; в данном случае это длина отрезка AB. Площадь треугольника мы уже умеем считать с помощью косого произведения.

Таким образом, для нахождения ответа достаточно лишь разделить модуль косого произведения на длину отрезка AB.

Альтернативный метод через уравнение прямой

Из математики известно, что если использовать уравнение прямой $ax + by + c = 0$, то расстояние от точки P до прямой можно найти с помощью формулы $|aP_x + bP_y + c| / \sqrt{a^2 + b^2}$. Этот альтернативный способ решения задачи, который условно можно назвать алгебраическим, также дает хорошие результаты. Однако геометри-

ческий подход в целом интересен тем, что он приводит к наглядному и интуитивному пониманию принципа работы алгоритма, производящихся в нем операций и возникающих при этом погрешностей.

Реализация приводится с использованием вещественных чисел, поскольку из-за наличия операции деления результат (и даже его квадрат) не обязательно целый. Поскольку многие реализации вспомогательных примитивов в предыдущих разделах использовали вычисления в целых числах, мы приводим их здесь снова, на этот раз в форме с использованием вещественных чисел.

Python

```
def point_line_distance(
    ax, ay, bx, by, px, py):
  return abs(
      cross_product(
          ax - px, ay - py,
          bx - px, by - py) /
      distance(ax, ay, bx, by))
def distance(ax, ay, bx, by):
  return sqrt((ax - bx) ** 2 +
              (ay - by) ** 2)
def cross_product(px, py, qx, qy):
  return px * qy - py * qx
```

C++

```
double Distance(
    double ax, double ay,
    double bx, double by) {
  return sqrt(
    (ax - bx) * (ax - bx) +
    (ay - by) * (ay - by));
}
double CrossProduct(
    double ax, double ay,
    double bx, double by) {
  return ax * by - ay * bx;
}
double PointLineDistance(
    double ax, double ay,
    double bx, double by,
    double px, double py) {
  return abs(
      CrossProduct(
          ax - px, ay - py,
          bx - px, by - py) /
      Distance(ax, ay, bx, by));
}
```

Знаковое расстояние

Отметим, что если убрать вычисление модуля от результата (т. е. вызовы функции `abs()` в коде), то получится величина, которую условно можно назвать знаковым расстоянием. Знак этой величины будет соответствовать тому, в какой полуплоскости относительно прямой лежит точка. В некоторых задачах такое расстояние со знаком более удобно, чем обычное расстояние.

29.11. Расстояние от точки до отрезка

Дана точка P и отрезок AB. Требуется определить расстояние от точки до отрезка.

Для решения заметим, что во многих случаях задача сводится к задаче о вычислении **расстояния от точки до прямой**, которую мы уже умеем решать. Однако есть

особые случаи, в которых ответ на задачу с отрезком будет отличаться, а также вырожденные случаи, когда A и B совпадают и решение с прямой будет неприменимо. Как всегда в геометрических задачах, изобразим наглядно все основные случаи.

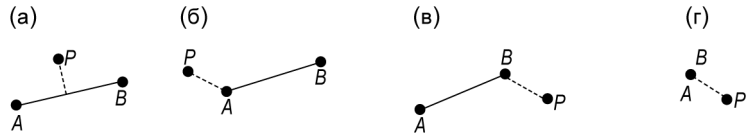

Рис. 29.8. Основные случаи, возникающие при определении расстояния от точки до отрезка: *(а)* кратчайший путь от точки до отрезка оканчивается внутри отрезка; *(б)* кратчайший путь оканчивается в точке A; *(в)* кратчайший путь оканчивается в точке B; *(г)* отрезок вырожден в точку

Можно заметить, что в то время как в случае «а» ответом является расстояние от P до прямой AB, во всех случаях «б», «в», «г» ответом является **минимум из расстояний** PA и PB (рис. 29.8). Таким образом, для полного решения задачи нам осталось научиться отличать случай «а» от всех остальных случаев.

Заметим, что главная характеристика случая «а» — это то, что проекция точки P находится внутри отрезка AB. Поскольку находить проекцию мы пока не умеем и ее поиск потребует работы с вещественными числами, то попробуем рассмотреть углы $\angle PAB$ и $\angle PBA$ (рис. 29.9).

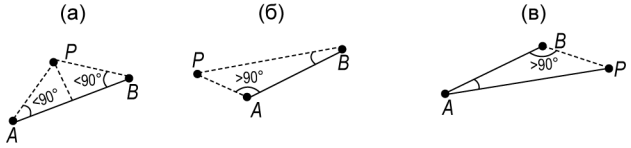

Рис. 29.9. Величины углов в случаях *(а)*, *(б)* и *(в)*

В случае «а» оба этих угла острые, в то время как в случаях «б» и «в» хотя бы один из углов — тупой. При наличии прямого угла не важно, к какому из случаев относить пример — ответ будет получаться одинаковым.

Каким образом проще всего проверять, острый ли угол или тупой? Вспомним, что скалярное произведение основано на косинусе угла между векторами и что косинус положителен для острых углов и отрицателен для тупых. Таким образом, для определения того, острые ли углы или тупые, достаточно проверить знаки двух скалярных произведений:

$$\overrightarrow{AB} \cdot \overrightarrow{AP},$$

$$\overrightarrow{BA} \cdot \overrightarrow{BP}.$$

Осталось лишь научиться корректно разбирать вырожденный случай «г». Заметим, что скалярное произведение в случае «г» всегда будет равно нулю. Верно ли будет

утверждение, что при нулевом скалярном произведении следует решать задачу методом для случаев «б», «в», «г»? Да, это верно — для этого достаточно разобрать все случаи, когда скалярное произведение может быть равным нулю:

- если точки A и B совпадают, то оба скалярных произведения равны нулю. В этом случае ответ на задачу равен и PA, и PB;
- если точка P совпадает с точкой A или B, то соответствующее скалярное произведение будет равно нулю. В этом случае ответ на задачу будет равен нулю, и либо PA, либо PB будет равно нулю;
- если $\angle PAB$ или $\angle PBA$ прямой, то соответствующее скалярное произведение будет равно нулю. Этот случай, как уже упоминалось выше, можно решать любым из двух методов.

Подводя **итог**, алгоритм будет выглядеть следующим образом:

3. Проверить, верно ли, что $\overrightarrow{AB} \cdot \overrightarrow{AP} > 0$ и $\overrightarrow{BA} \cdot \overrightarrow{BP} > 0$.
4. Если ответ — «да», то вычислить и вернуть расстояние от P до прямой AB.
5. Если же ответ — «нет», то вычислить и вернуть минимум из расстояний PA и PB.

Реализация будет частично в целых числах и частично в вещественных, поскольку проверки на шаге №1 можно произвести в целых числах (при условии того, что входные координаты целочисленные), в то время как вычисление расстояний уже потребует вычислений с плавающей запятой.

Мы снова приводим здесь реализацию всех примитивов, на этот раз с использованием структуры «точка» с определенной операцией вычитания — что позволяет компактно записать многочисленные векторные операции и уменьшить риск опечаток в громоздких выражениях.

Python

```
from collections import *
from math import *

class Point(namedtuple(
    'Point', ['x', 'y'])):
  def __sub__(self, other):
    return Point(self.x - other.x,
                 self.y - other.y)

def point_segment_distance(
    a, b, p):
  if (dot_product(
          p - a, b - a) > 0 and
      dot_product(
          p - b, a - b) > 0):
    return point_line_distance(
```

C++

```
#include <algorithm>
#include <cmath>
#include <cstdint>
using namespace std;

struct Point {
  int64_t x, y;
  Point operator-(Point p) const {
    return {x - p.x, y - p.y};
  }
};

double Distance(Point a, Point b) {
  return sqrt(
    (a.x - b.x) * (a.x - b.x) +
    (a.y - b.y) * (a.y - b.y));
```

```
        a, b, p)
  return min(distance(p, a),
             distance(p, b))
def point_line_distance(a, b, p):
  return abs(
      cross_product(a - p, b - p) /
      distance(a, b))
def distance(a, b):
  return sqrt((a.x - b.x) ** 2 +
              (a.y - b.y) ** 2)
def dot_product(p, q):
  return p.x * q.x + p.y * q.y
def cross_product(p, q):
  return p.x * q.y - p.y * q.x
```

```
}
int64_t CrossProduct(
    Point p, Point q) {
  return p.x * q.y - p.y * q.x;
}
int64_t DotProduct(
    Point p, Point q) {
  return p.x * q.x + p.y * q.y;
}
double PointLineDistance(
    Point a, Point b, Point p) {
  return abs(
      CrossProduct(a - p, b - p) /
      Distance(a, b));
}
double PointSegmentDistance(
    Point a, Point b, Point p) {
  if (DotProduct(
          p - a, b - a) > 0 &&
      DotProduct(
          p - b, a - b) > 0) {
    return PointLineDistance(
        a, b, p);
  }
  return min(Distance(p, a),
             Distance(p, b));
}
```

Использование перегрузки операторов

Приведенные выше реализации перегружают только оператор вычитания двух точек. Можно было бы пойти дальше и оформить операции скалярного и косого произведений также в виде перегруженных операторов — например, оператора умножения для скалярного произведения и оператора «исключающего ИЛИ» для косого произведения. Хотя это позволило бы достичь еще большей краткости реализации, читаемость кода с точки зрения неподготовленного человека ухудшилась бы — поэтому примеры в книге не используют такой прием. Однако в условиях спортивного программирования, где участник пишет код так, как ему это удобнее и быстрее всего, применение такого рода сокращений может быть оправдано.

29.12. Точка пересечения двух прямых

Даны две прямых: первая задана точками A и B, вторая — P и Q (точки A и B не совпадают, и то же самое верно в отношении точек P и Q). Требуется найти точку пересечения этих прямых, если прямые пересекаются и не совпадают.

В зависимости от требований конкретной задачи, могут быть вариации в постановке задания: например, иногда может потребоваться отдельно определять случай совпа-

дающих прямых. Сделать это несложно: если мы знаем, что прямые параллельны, то достаточно проверить, лежит ли произвольная точка одной прямой на другой прямой — например, сравнив косое произведение $\overrightarrow{AB}$ и $\overrightarrow{AP}$ с нулем.

Для решения этой задачи удобно применять **векторное** представление прямых. В этом представлении множество всех точек прямой задается в виде:

$$\vec{u} + t \cdot \vec{v},$$

где t — вещественное число, принимающее всевозможные значения, $\vec{u}$ — вектор до точки, лежащей на прямой, $\vec{v}$ — вектор, направленный вдоль прямой. Это представление удобно тем, что оно наглядным образом связано со входными параметрами и не требует для его построения сложных арифметических операций.

Итак, входные прямые можно представить в векторной форме следующим образом:

$$\vec{A} + t_1 \cdot \overrightarrow{AB},$$

$$\vec{P} + t_2 \cdot \overrightarrow{PQ}.$$

Искомая точка пересечения двух прямых — это общая точка этих двух множеств. Ее можно вычислить, если найти t_1 или t_2 как решение соответствующего уравнения:

$$\vec{A} + t_1 \cdot \overrightarrow{AB} = \vec{P} + t_2 \cdot \overrightarrow{PQ}.$$

Это уравнение можно решать разными способами. Быстрый способ прийти к ответу — это избавиться от одной переменной путем домножения обеих частей косым произведением на вектор $\overrightarrow{PQ}$ (и пользуясь тем фактом, что косое произведение вектора на себя же равно нулю). Тогда мы можем сразу определить искомое t_1:

$$\left(\vec{A} + t_1 \cdot \overrightarrow{AB}\right) \wedge \overrightarrow{PQ} = \left(\vec{P} + t_2 \cdot \overrightarrow{PQ}\right) \wedge \overrightarrow{PQ},$$

$$t_1 = \frac{\left(\vec{P} - \vec{A}\right) \wedge \overrightarrow{PQ}}{\overrightarrow{AB} \wedge \overrightarrow{PQ}}.$$

Альтернативные способы решения

Описанный выше способ решения уравнения может показаться нестандартным и нестрогим. Можно использовать и классический подход, когда векторное уравнение заменяется двумя уравнениями по каждой из координат, и эта система уравнений решается с помощью формул Крамера. Как можно убедиться, результат будет тем же самым.

Знаменатель этой дроби может быть равен нулю, но это возможно только в случае параллельных и/или совпадающих прямых (поскольку косое произведение основано на синусе угла между векторами), и в этом случае решения по условию не существует. Во всех остальных случаях эта формула дает искомое значение t_1, которое можно подставить в векторное представление первой прямой и вычислить искомую точку пересечения.

Реализация использует целые числа для определения того, параллельны ли прямые или нет. Вычисления точки пересечения используют операции с плавающей запятой.

Python

```
from collections import *

class Point(namedtuple(
    'Point', ['x', 'y'])):
  def __add__(self, other):
    return Point(self.x + other.x,
                 self.y + other.y)
  def __sub__(self, other):
    return Point(self.x - other.x,
                 self.y - other.y)
  def __mul__(self, coeff):
    return Point(self.x * coeff,
                 self.y * coeff)

def lines_intersection(
    a, b, p, q):
  ab = b - a
  pq = q - p
  denominator = cross_product(
      ab, pq)
  if denominator == 0:
    return None
  t = (cross_product(p - a, pq) /
       denominator)
  return a + ab * t
def cross_product(p, q):
  return p.x * q.y - p.y * q.x
```

C++

```
#include <cstdint>
#include <iostream>
using namespace std;

struct Point {
  double x, y;
  Point operator+(Point p) const {
    return {x + p.x, y + p.y};
  }
  Point operator*(
      double coeff) const {
    return {x * coeff, y * coeff};
  }
};
struct IntPoint {
  int64_t x, y;
  IntPoint operator-(
      IntPoint p) const {
    return {x - p.x, y - p.y};
  }
  operator Point() const {
    return {double(x), double(y)};
  }
};

int64_t CrossProduct(
    IntPoint p, IntPoint q) {
  return p.x * q.y - p.y * q.x;
}
bool LinesIntersection(
    IntPoint a, IntPoint b,
    IntPoint p, IntPoint q,
    Point& res) {
  int64_t denominator =
      CrossProduct(b - a, q - p);
  if (denominator == 0)
    return false;
  double t = double(CrossProduct(
      p - a, q - p)) / denominator;
  res = Point(a) +
        Point(b - a) * t;
  return true;
}
```

Типы Point и IntPoint в решении на C++

Чтобы производить часть вычислений в целых числах, а часть — в вещественных, нам приходится определять два различных типа данных для хранения координат точки. Благодаря этому мы избегаем проблем с точностью при проверке прямых на параллельность и одновременно с этим пользуемся компактной формой записи операций над векторами. Следует отметить, что на практике абсолютная точность проверок в таких задачах требуется редко, и реализация полностью на вещественных числах будет проще за счет использования единственного типа данных.

29.13. Точка пересечения двух отрезков

Даны два отрезка AB и PQ. Требуется найти точку их пересечения; если их несколько — разрешается выбрать любую.

Решение этой задачи начнем со сведения ее к задаче **пересечения двух прямых**, решение которой мы рассмотрели в предыдущем разделе. Это сведение можно произвести при условии, если оба отрезка не вырождены, т. е. точки A и B различны, и аналогично для точек P и Q.

При таком подходе требуется научиться отсекать случаи, когда точка пересечения прямых лежит за пределами отрезков. Однако если сравнивать координаты точки пересечения с координатами концов отрезков, то при сравнениях вещественных чисел могут возникнуть ошибки, вызванные погрешностями. Можно было бы применить алгоритм проверки отрезков на пересечение, рассмотренный нами в одном из предыдущих разделов, однако это привело бы к слишком громоздкому решению.

Вспомним, что у решения задачи о пересечении прямых есть геометрический смысл: найденный нами коэффициент t_1 является, в некотором смысле, «коэффициентом пропорции» между точками A и B; то же самое верно в отношении t_2, P и Q:

$$\vec{A} + t_1 \cdot \overrightarrow{AB} = \vec{P} + t_2 \cdot \overrightarrow{PQ}.$$

Можно заметить, что, для того чтобы эта точка пересечения лежала на обоих отрезках, должно выполняться:

$$0 \le t_1, t_2 \le 1.$$

Графически это можно представить себе следующим образом (рис. 29.10).

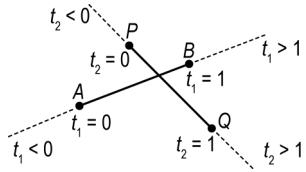

Рис. 29.10. Движение вдоль отрезка AB соответствует постепенному изменению коэффициента t_1 от 0 до 1; аналогично, движение вдоль отрезка PQ — это изменение t_2 от 0 до 1. Таким образом, в точке пересечения отрезков, если она есть, оба этих коэффициента должны быть между 0 и 1

Формулу для нахождения t_1 мы получили в предыдущем разделе, а формулу для t_2 можно вывести аналогичным образом:

$$t_1 = \frac{\left(\vec{P} - \vec{A}\right) \wedge \overrightarrow{PQ}}{\overrightarrow{AB} \wedge \overrightarrow{PQ}}, \quad t_2 = \frac{\left(\vec{P} - \vec{A}\right) \wedge \overrightarrow{AB}}{\overrightarrow{AB} \wedge \overrightarrow{PQ}}.$$

Если числители и знаменатели этих дробей целые — что верно в случае целочисленных входных координат — то эти дроби можно сравнивать с нулем и единицей с помощью одних только целочисленных операций, домножив части неравенств на знаменатель. Единственная тонкость — если знаменатель отрицателен, то при домножении на него знак неравенства изменился бы; в математической записи этого можно избежать, домножая на модуль знаменателя.

Итоговый критерий того, что точка пересечения принадлежит обоим отрезкам, будет выглядеть следующим образом (отметим, что в программной реализации та же самая идея будет выглядеть чуть проще):

$$\begin{cases} 0 \le \left(\left(\vec{P} - \vec{A}\right) \wedge \overrightarrow{PQ}\right) \cdot \operatorname{sgn}\left(\overrightarrow{AB} \wedge \overrightarrow{PQ}\right) \le \left|\overrightarrow{AB} \wedge \overrightarrow{PQ}\right|, \\ 0 \le \left(\left(\vec{P} - \vec{A}\right) \wedge \overrightarrow{AB}\right) \cdot \operatorname{sgn}\left(\overrightarrow{AB} \wedge \overrightarrow{PQ}\right) \le \left|\overrightarrow{AB} \wedge \overrightarrow{PQ}\right|. \end{cases}$$

В завершение разберем случаи, когда один или оба отрезка вырождаются в точки. В таком случае, как и в алгоритме проверки отрезков на пересечение, проверим с помощью косого произведения, лежат ли точки на одной прямой; если нет, то ответа не существует. Если же отрезки лежат на одной прямой, то, как и в одномерном случае, ответом будет любая точка между максимумом из левых концов и минимумом из правых (в качестве критерия сравнения можно выбрать, например, сравнение сначала по абсциссе и при равенстве — по ординате).

> Требуемый ответ в вырожденном случае может варьироваться в зависимости от задачи. Например, в некоторых задачах в случае накладывающихся отрезков может требоваться выдать весь отрезок-пересечение в качестве результата. Алгоритм решения таких вариаций будет таким же, а изменится лишь последний шаг — заполнение ответа на основе вычисленных минимумов и максимумов.

Реализация использует целые числа для определения того, пересекаются отрезки или нет. Вычисления точки пересечения используют операции с плавающей запятой.

Python

```
from collections import *

class Point(namedtuple(
    'Point', ['x', 'y'])):
  def __add__(self, other):
    return Point(self.x + other.x,
                 self.y + other.y)
  def __sub__(self, other):
    return Point(self.x - other.x,
```

C++

```
#include <cstdint>
#include <tuple>
using namespace std;

struct Point {
  double x, y;
  Point operator+(Point p) const {
    return {x + p.x, y + p.y};
  }
```

```
                  self.y - other.y)
  def __mul__(self, coeff):
    return Point(self.x * coeff,
                  self.y * coeff)

def segments_intersection(
    a, b, p, q):
  nominator1 = cross_product(
      p - a, q - p)
  nominator2 = cross_product(
      p - a, b - a)
  denominator = cross_product(
      b - a, q - p)
  if denominator == 0:
    if nominator1 != 0:
      return None
    begin1 = min(a, b)
    begin2 = min(p, q)
    begin = max(begin1, begin2)
    end1 = max(a, b)
    end2 = max(p, q)
    end = min(end1, end2)
    if end < begin:
      return None
    return begin
  if denominator < 0:
    denominator *= -1
    nominator1 *= -1
    nominator2 *= -1
  if not (0 <= nominator1 <=
          denominator):
    return None
  if not (0 <= nominator2 <=
          denominator):
    return None
  t = nominator1 / denominator
  return a + (b - a) * t
def cross_product(p, q):
  return p.x * q.y - p.y * q.x
```

```
  Point operator*(
      double coeff) const {
    return {x * coeff, y * coeff};
  }
};
struct IntPoint {
  int64_t x, y;
  bool operator<(
      IntPoint p) const {
    return tie(x, y) <
           tie(p.x, p.y);
  }
  IntPoint operator-(
      IntPoint p) const {
    return {x - p.x, y - p.y};
  }
  operator Point() const {
    return {double(x), double(y)};
  }
};

int64_t CrossProduct(
    IntPoint p, IntPoint q) {
  return p.x * q.y - p.y * q.x;
}
bool SegmentsIntersection(
    IntPoint a, IntPoint b,
    IntPoint p, IntPoint q,
    Point& res) {
  int64_t nominator1 =
      CrossProduct(p - a, q - p);
  int64_t nominator2 =
      CrossProduct(p - a, b - a);
  int64_t denominator =
      CrossProduct(b - a, q - p);
  if (denominator == 0) {
    if (nominator1 != 0)
      return false;
    auto begin1 = min(a, b);
    auto begin2 = min(p, q);
    auto begin = max(
        begin1, begin2);
    auto end1 = max(a, b);
    auto end2 = max(p, q);
    auto end = min(end1, end2);
    if (end < begin)
```

```
      return false;
    res = begin;
    return true;
  }
  if (denominator < 0) {
    denominator *= -1;
    nominator1 *= -1;
    nominator2 *= -1;
  }
  if (nominator1 < 0 ||
      nominator1 > denominator ||
      nominator2 < 0 ||
      nominator2 > denominator) {
    return false;
  }
  double t = double(nominator1) /
      denominator;
  res = Point(a) +
        Point(b - a) * t;
  return true;
}
```

29.14. Матрица поворота

Дана точка P и угол α. Требуется найти координаты этой точки после поворота против часовой стрелки на угол α вокруг начала координат (рис. 29.11).

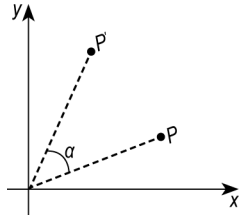

Рис. 29.11. Полярные углы точек P и P' должны отличаться ровно на α

Для решения этой задачи мы сошлемся на известный из математики факт, что координаты точки до и после поворота связаны следующим соотношением:

$$x' = x \cdot \cos\alpha - y \cdot \sin\alpha,$$
$$y' = x \cdot \sin\alpha + y \cdot \cos\alpha.$$

Это соотношение удобно понимать как матричное равенство:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix},$$

где стоящая в правой части матрица 2×2 называется **матрицей поворота**.

Запоминание формул поворота

Поскольку в формулах поворота легко перепутать синусы с косинусами и знаки перед ними, можно воспользоваться двумя простыми самопроверками: при $\alpha = 0$ и $\alpha = 90^\circ$. В первом случае координаты меняться не должны — значит, перед соответствующей координатой должен стоять косинус со знаком «плюс». Во втором случае, как несложно представить в голове или нарисовать на листочке, координаты обмениваются местами со сменой знака у ординаты — значит, в первом равенстве перед y должен стоять синус со знаком «минус», а во втором равенстве перед x — синус со знаком «плюс».

Дополнительные применения матрицы поворота

Существуют обобщения матрицы поворота на трехмерные и многомерные пространства — основанные на том, что вращение вокруг одной координатной оси (или, в многомерном пространстве, вращение в какой-либо плоскости) эквивалентно двумерному вращению по соответствующим координатам с сохранением всех остальных координат. Другая сфера применений — это возможность перемножать матрицу поворота с матрицами других преобразований, возводить их в степень и т. п., получая тем самым способ эффективно применять большое число однотипных или повторяющихся преобразований.

Реализация (предполагается, что угол задается в радианах; например, π соответствует повороту на развернутый угол, т. е. 180 градусам).

Python

```
from math import *

def rotate(x, y, angle):
  s = sin(angle)
  c = cos(angle)
  return (x * c - y * s,
          x * s + y * c)
```

C++

```
#include <cmath>
using namespace std;

void RotatePoint(
    double& x, double& y,
    double angle) {
  double oldx = x, oldy = y;
  double s = sin(angle);
  double c = cos(angle);
  x = oldx * c - oldy * s;
  y = oldx * s + oldy * c;
}
```

29.15. Пример решения задачи. Проверка окружностей на пересечение

Задача. Две окружности заданы координатами своих центров и радиусами. Требуется найти число точек их пересечения.

Входные данные состоят из двух строк, каждая из которых описывает одну окружность. Каждая строка содержит три разделенных пробелами целых числа:

абсциссу, ординату и радиус; координаты и радиус не превосходят 10^9, и радиус неотрицателен.

Вывести требуется одно число — ответ на задачу. Если число точек больше трех, следует выводить «3».

Примеры

Входные данные	Требуемый результат
0 0 2 3 0 2	2
Входные данные	**Требуемый результат**
6 0 5 0 8 5	1

Решение

Легко заметить, что ключ к решению задачи — это расстояние между центрами окружностей. Например, если расстояние больше суммы радиусов, то ответ равен нулю, а если расстояние в точности равно сумме радиусов, то ответ — единица.

Однако в этой задаче есть несколько каверзных случаев:

1. Одна окружность может лежать внутри другой — в таком случае при достаточно малом расстоянии между центрами (меньшем, чем разность радиусов) ни одной точки пересечения не будет.
2. Окружности могут совпадать. Тогда, если их радиусы ненулевые, они будут иметь бесконечное число точек пересечения (по условию в таком случае надо будет вывести «3»).
3. Окружности могут иметь нулевой радиус. Это создает особые случаи, такие, как в случае совпадающих окружностей, в котором ответ либо бесконечность, либо единица.

Кроме того, из-за большой величины входных координат желательно производить все операции в целых числах, чтобы избежать ошибок округления. Этого легко добиться, если вместо рассмотрения расстояния рассматривать квадрат расстояния — все неравенства останутся теми же.

Реализация

Python

```
x1, y1, r1 = map(
    int, input().split())
x2, y2, r2 = map(
    int, input().split())
d = (x1 - x2) ** 2 + (y1 - y2) ** 2
ans = 0
```

C++

```
#include <cstdint>
#include <iostream>
using namespace std;
int main() {
  int64_t x1, y1, r1, x2, y2, r2;
  cin >> x1 >> y1 >> r1
```

```
if d == 0 and r1 == r2 == 0:
  ans = 1
elif d == 0 and r1 == r2:
  ans = 3
elif d == (r1 + r2) ** 2:
  ans = 1
elif d == (r1 - r2) ** 2:
  ans = 1
elif ((r1 - r2) ** 2 < d <
      (r1 + r2) ** 2):
  ans = 2
print(ans)
```

```
      >> x2 >> y2 >> r2;
  int64_t d =
      (x1 - x2) * (x1 - x2) +
      (y1 - y2) * (y1 - y2);
  int64_t sum =
      (r1 + r2) * (r1 + r2);
  int64_t diff =
      (r1 - r2) * (r1 - r2);
  int ans = 0;
  if (!d && r1 == r2 && !r1)
    ans = 1;
  else if (!d && r1 == r2)
    ans = 3;
  else if (d == sum)
    ans = 1;
  else if (d == diff)
    ans = 1;
  else if (diff < d && d < sum)
    ans = 2;
  cout << ans << endl;
}
```

29.16. Пример решения задачи. Пересечение окружности и прямой

Задача. Дана окружность с центром в заданной точке C и радиусом R, а также прямая, проходящая через две заданные точки A и B. Требуется найти координаты всех точек пересечения прямой и окружности.

Входные данные состоят из описания окружности в первой строке и описания прямой во второй. Первая строка содержит три разделенных пробелами целых числа: абсциссу и ординату точки C и радиус R; вторая строка содержит четыре целых числа: координаты точки A, за которыми следуют координаты точки B. Координаты и радиус не превосходят 10^7; радиус больше нуля.

Вывести требуется количество точек пересечения и координаты каждой точки пересечения в отдельных строках. Координаты следует выводить с такой точностью, чтобы относительная или абсолютная погрешность не превосходила 10^{-9}. Точки разрешается выводить в любом порядке.

Примеры

Входные данные	Требуемый результат
0 0 2 2 0 2 1	1 2 0

Входные данные	Требуемый результат
1 2 2 0 2 1 0	2 -0.6000000000 3.2000000000 1.0000000000 0.0000000000

Решение

Хотя эту задачу можно было бы решать и чисто алгебраически, записав систему уравнений относительно координат искомой точки, вывод решения потребует немалого времени и большой внимательности. Вместо этого мы будем решать задачу, исходя из геометрических соображений.

Во-первых, для упрощения выкладок перенесем центр окружности в начало координат, для чего просто отнимем координаты точки C от всех точек входного набора (впоследствии, при вычислении ответа, просто прибавим вычтенные координаты обратно к точкам ответа).

Во-вторых, число точек пересечения зависит от расстояния между центром окружности и прямой. Задачу вычисления расстояния до прямой мы уже научились решать; в нашем случае, когда точка находится в начале координат, формула для вычисления расстояния d будет иметь вид:

$$d = \frac{\left|\vec{A} \wedge \vec{B}\right|}{AB}.$$

В зависимости от того, больше ли d, чем R, равно ему или меньше него, ответ будет состоять из нуля, одной или двух точек соответственно.

В-третьих, если мы найдем ближайшую к началу координат точку прямой — обозначим ее через P, — то искомые точки пересечения Q_1 и Q_2 будут располагаться симметрично относительно P и на равном расстоянии k. Проиллюстрируем это наглядно (рис. 29.12).

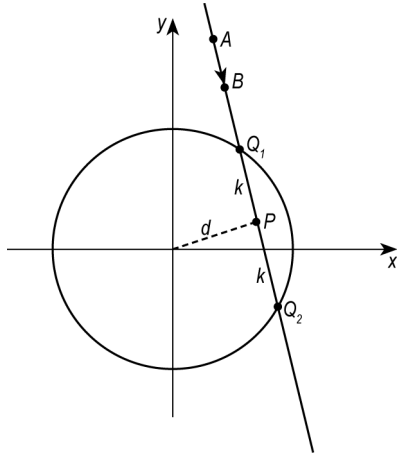

Рис. 29.12. Пересечение окружности и прямой, проведенной через данные точки A и B

Осталось лишь найти координаты точки P и расстояние k, потому что тогда координаты искомых точек можно найти, откладывая от P в обе стороны вектор $\overrightarrow{AB}$ с нужным коэффициентом:

$$\overrightarrow{Q_1} = \vec{P} - \overrightarrow{AB} \cdot \frac{k}{AB},$$

$$\overrightarrow{Q_2} = \vec{P} + \overrightarrow{AB} \cdot \frac{k}{AB}.$$

Расстояние k легко вычислить по известным d и R:

$$k = \sqrt{R^2 - d^2}.$$

Для нахождения P можно заметить, что это проекция начала координат на нашу прямую. Скалярное произведение любых двух векторов обладает тем свойством, что оно равно длине одного вектора, домноженной на проекцию другого вектора на него (это свойство можно вывести из того факта, что скалярное произведение равно произведению длин векторов и косинуса угла между ними). Используя это свойство, мы можем спроецировать вектор $-\vec{A}$ на вектор $\overrightarrow{AB}$, найти коэффициент, на который надо домножить $\overrightarrow{AB}$ для перехода из $\vec{A}$ в $\vec{P}$, и тем самым вычислить положение точки P:

$$\vec{P} = \vec{A} - \overrightarrow{AB} \cdot \frac{\vec{A} \cdot \overrightarrow{AB}}{AB^2}.$$

Реализация

Python

```
from collections import *
from math import *

class Point(namedtuple(
    'Point', ['x', 'y'])):
  def __add__(self, other):
    return Point(self.x + other.x,
                 self.y + other.y)
  def __sub__(self, other):
    return Point(self.x - other.x,
                 self.y - other.y)
  def __mul__(self, coeff):
    return Point(self.x * coeff,
                 self.y * coeff)

def distance2(a, b):
  return ((a.x - b.x) ** 2 +
          (a.y - b.y) ** 2)
```

C++

```
#include <cmath>
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;

const double EPS = 1E-9;

struct Point {
  double x, y;
  Point operator+(Point p) const {
    return {x + p.x, y + p.y};
  }
  Point operator-(Point p) const {
    return {x - p.x, y - p.y};
  }
  Point operator*(
      double coeff) const {
```

```
def cross_product(a, b):
  return a.x * b.y - a.y * b.x
def dot_product(a, b):
  return a.x * b.x + a.y * b.y
def intersect_circle_line(r, a, b):
  a_b = b - a
  a_b_dist2 = distance2(a, b)
  a_b_cross = cross_product(a, b)
  p = (a - a_b * (
         dot_product(a, a_b) /
         a_b_dist2))
  if (a_b_cross ** 2 >
      r ** 2 * a_b_dist2):
    return []
  if (a_b_cross ** 2 ==
      r ** 2 * a_b_dist2):
    return [p]
  d_2 = a_b_cross ** 2 / a_b_dist2
  coeff = sqrt(
      (r ** 2 - d_2) / a_b_dist2)
  return [p - a_b * coeff,
          p + a_b * coeff]

cx, cy, r = map(
    int, input().split())
ax, ay, bx, by = map(
    int, input().split())
a = Point(ax - cx, ay - cy)
b = Point(bx - cx, by - cy)
ans = intersect_circle_line(
    r, a, b)
print(len(ans))
for p in ans:
  print('%.20f %.20f' % (
      p.x + cx, p.y + cy))
```

```
    return {x * coeff, y * coeff};
  }
};

double Distance2(Point p, Point q) {
  return
      (p.x - q.x) * (p.x - q.x) +
      (p.y - q.y) * (p.y - q.y);
}
double CrossProduct(
    Point p, Point q) {
  return p.x * q.y - p.y * q.x;
}
double DotProduct(
    Point p, Point q) {
  return p.x * q.x + p.y * q.y;
}
vector<Point> IntersectCircleLine(
    double r, Point a, Point b) {
  Point a_b = b - a;
  double a_b_dist2 =
      Distance2(a, b);
  double a_b_cross =
      CrossProduct(a, b);
  Point p = (a - a_b * (
        DotProduct(a, a_b) /
        a_b_dist2));
  double d = abs(a_b_cross) /
             sqrt(a_b_dist2);
  if (d > r + EPS)
    return {};
  if (abs(d - r) < EPS)
    return {p};
  double coeff = sqrt(
      (r * r - d * d) / a_b_dist2);
  return {p - a_b * coeff,
          p + a_b * coeff};
}

int main() {
  double r;
  Point c, a, b;
  cin >> c.x >> c.y >> r
      >> a.x >> a.y
      >> b.x >> b.y;
  vector<Point> res =
```

```
        IntersectCircleLine(
            r, a - c, b - c);
    cout.setf(ios::fixed);
    cout.precision(20);
    cout << res.size() << endl;
    for (auto p : res) {
      cout << p.x + c.x << ' '
           << p.y + c.y << endl;
    }
}
```

Вычисления в целых числах

Отметим, что только в реализации на Python мы выполнили часть вычислений в целых числах; в особенности это касается части, которая проверяет, сколько имеется точек пересечения. В реализации на C++ это сделать было бы не так легко, поскольку при заданных ограничениях даже типа int64_t не хватило бы для хранения промежуточных результатов; поэтому реализация на C++ производит все вычисления с плавающей запятой.

29.17. Пример решения задачи. Сортировка точек по углу

Задача. Даны n точек. Требуется упорядочить их в порядке возрастания полярного угла; при равенстве угла точки должны следовать в порядке увеличения расстояния от начала координат.

Полярный угол определяется как угол, отсчитываемый против часовой стрелки между положительным направлением оси абсцисс и направлением на точку. Таким образом, полярный угол (измеренный в радианах) может принимать значения от нуля включительно до 2π не включительно.

Первая строка входного файла содержит число n, задающее количество точек ($1 \le n \le 10^5$). Последующие n строк содержат по два целых числа каждая — координаты i-й точки (координаты не превосходят по модулю 10^9). Все точки попарно различны, и ни одна точка не совпадает с началом координат.

Вывести требуется перестановку из n чисел, т. е. последовательность из чисел от 1 до n, соответствующих номерам точек в искомом порядке.

Примеры

Входные данные	Требуемый результат
4 1 0 0 1 0 -1 -1 0	1 2 4 3

Входные данные	Требуемый результат
5 1 1 2 2 3 3 2 3 3 2	5 1 2 3 4

Решение

Эта задача — очередной пример того, как проблема погрешностей выходит на первый план. Дело здесь в том, что полярные углы могут принимать достаточно близкие значения, и вычисленные в числах с плавающей запятой значения не только нельзя сравнивать стандартным методом (через сравнение с помощью константы 10^{-9} или подобной ей), но и, более того, математически разные углы могут оказаться округленными к одному и тому же значению с плавающей запятой. Таким образом, от методов решения этой задачи, основанных на тригонометрических функциях (atan2 и т. п.), следует отказаться.

> ***Примеры «плохих» случаев***
>
> Самый трудный случай для вычисления полярных углов — это точки, лежащие рядом с диагональю системы координат. Например, полярные углы для точек (99; 100) и (98; 99) различаются всего на $5 \cdot 10^{-5}$. Чем больше значения координат, тем ближе окажутся полярные углы в примерах такого рода, причем близость углов будет нарастать с квадратичной скоростью. Уже на координатах порядка 10^8 полярные углы будут отличаться менее, чем на 10^{-16}, и при использовании стандартных 64-битных типов с плавающей запятой точности перестанет хватать: вычисленный угол будет ошибочно совпадать для разных направлений.

Для решения этой задачи в целых числах воспользуемся примитивом «направление поворота». Напомним, что этот примитив позволяет путем вычисления косого произведения определить, направлена ли указанная тройка точек по часовой стрелке или против (либо же лежит на одной прямой). Мы можем применить этот примитив к тройке OPQ, где O — начало координат, а P и Q — две точки, которые мы хотим сравнить по полярному углу, и полученное направление поворота даст нам информацию об относительном порядке точек P и Q (рис. 29.13).

Однако это еще не дает нам полного решения, поскольку при разнице полярных углов в 180 градусов и более этот критерий будет давать неверные результаты (рис. 29.14).

Кроме того, возникает проблема с началом отсчета: по определению полярного угла, сначала должны следовать точки первого квадранта, потом второго и т. д., однако критерий на основе угла поворота не имеет такой упорядоченности (по сути, критерий на основе направления поворота «зацикливается»).

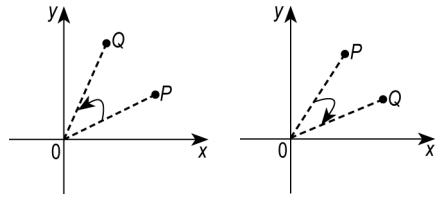

Рис. 29.13. Определение относительного порядка точек по направлению угла поворота. На изображении слева точка *P* следует до точки *Q* и угол поворота ориентирован против часовой стрелки; на изображении справа порядок точек обратный и угол поворота ориентирован по часовой стрелке

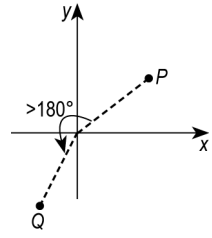

Рис. 29.14. Случай, когда полярные углы отличаются более чем на 180 градусов, требует особой обработки: в изображенном примере точка *P* следует до точки *Q*, однако угол поворота направлен по часовой стрелке

У обеих этих проблем есть простое решение: перед тем как сравнивать точки по направлению поворота, проверим, **в какой полуплоскости** они находятся: верхней или нижней. Если сравниваемые точки находятся в разных полуплоскостях, то их порядок легко определить; и только если точки находятся в одной полуплоскости, мы будем сравнивать их с помощью направления поворота. Важно, чтобы положительное направление оси абсцисс было отнесено к верхней полуплоскости, а отрицательное — к нижней (рис. 29.15).

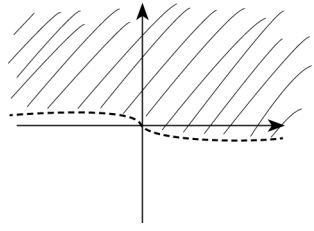

Рис. 29.15. При реализации удобно разбивать задачу на две половины: в верхней полуплоскости и в нижней. Ось абсцисс тоже разбивается пополам между этими двумя полуплоскостями

Реализация

Python

```
from collections import *

class Point(namedtuple(
    'Point', ['x', 'y', 'id'])):
  def __lt__(self, other):
    self_top = top_halfplane(self)
    other_top = top_halfplane(
        other)
    if self_top != other_top:
      return self_top
    prod = cross_product(
        self, other)
    if prod:
      return prod > 0
    return (distance0(self) <
            distance0(other))

def top_halfplane(p):
  return (p.y > 0 or
          p.y == 0 and p.x > 0)
def cross_product(p, q):
  return p.x * q.y - p.y * q.x
def distance0(p):
  return p.x ** 2 + p.y ** 2

n = int(input())
ps = [Point(
    *map(int, input().split()), id)
    for id in range(n)]
ps.sort()
print(*[p.id + 1 for p in ps])
```

C++

```
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;

struct Point {
  int64_t x, y;
  int id;
};
bool TopHalfplane(Point p) {
  return p.y > 0 ||
         (p.y == 0 && p.x > 0);
}
int64_t CrossProduct(Point p,
                     Point q) {
  return p.x * q.y - p.y * q.x;
}
int64_t Distance0(Point p) {
  return p.x * p.x + p.y * p.y;
}
bool CmpAng(Point p, Point q) {
  bool p_top = TopHalfplane(p);
  bool q_top = TopHalfplane(q);
  if (p_top != q_top)
    return p_top;
  int64_t prod =
      CrossProduct(p, q);
  if (prod)
    return prod > 0;
  return Distance0(p) <
         Distance0(q);
}

int main() {
  int n;
  cin >> n;
  vector<Point> ps(n);
  for (int i = 0; i < n; ++i) {
    cin >> ps[i].x >> ps[i].y;
    ps[i].id = i;
```

```
  }
  sort(ps.begin(), ps.end(),
       CmpAng);
  for (auto p : ps)
    cout << p.id + 1 << ' ';
}
```

Примечание к реализации компаратора

Обе представленные реализации можно было бы немного сократить, если реализовать операцию сравнения как сравнение по тройкам значений: номер полуплоскости, значение косого произведения и расстояние. Однако это привело бы к росту накладных расходов, поскольку при таком подходе все три значения вычислялись бы для каждого сравнения, в то время как представленная реализация вычисляет следующее значение только тогда, когда предыдущее совпало.

Глава 30.
Расширенный алгоритм Евклида

В отличие от стандартного алгоритма Евклида, находящего наибольший общий делитель двух чисел, расширенный алгоритм Евклида также находит коэффициенты, выражающие НОД через исходные числа. Иными словами, расширенный алгоритм Евклида находит по заданным неотрицательным целым a и b такие целые x и y, что:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

Отметим, что, вообще говоря, это уравнение имеет бесконечно много решений, и расширенный алгоритм Евклида находит только одно из них.

30.1. Алгоритм

Проще всего описать алгоритм в рекурсивной форме:

- если $b = 0$, то следует положить $x = 1$, $y = 0$;
- если $b > 0$, то вызовем алгоритм рекурсивно для $a' = b$, $b' = a \bmod b$; рекурсивный вызов вернет соответствующие x' и y'. Тогда коэффициенты для a и b следует положить равными:

$$x = y', \quad y = x' - \left\lfloor \frac{a}{b} \right\rfloor \cdot y'.$$

Асимптотика алгоритма составляет $O(\log(\min(a, b))$ — т. е. это точно та же асимптотика, что и у стандартного алгоритма Евклида. Это объясняется тем, что расширенный алгоритм Евклида совершает те же самые преобразования аргументов и рекурсивные вызовы — единственное отличие в добавлении нескольких дополнительных арифметических операций на каждом шаге.

30.2. Доказательство

Во-первых, заметим, что использование рекурсии здесь корректно, поскольку, как было доказано в главе об алгоритме Евклида *(см. гл. 10)*, второй аргумент рекурсивного вызова всегда строго убывает.

Во-вторых, в случае $b = 0$ верно, что $\gcd(a, b) = a$, и, следовательно, справедливо и тождество $a \cdot 1 + b \cdot 0 = \gcd(a, b)$.

Отметим, что здесь мы полагаемся на то, что a и b неотрицательны. Для отрицательных чисел нет единой точки зрения на то, как следует определять НОД; но если полагать НОД неотрицательным, то в расширенном алгоритме Евклида при $b = 0$ и $a < 0$ потребуется возвращать $x = -1$.

В-третьих, корректность обработки случая $b > 0$ можно доказывать с помощью метода индукции. Мы предполагаем, что рекурсивный вызов нашел *x'* и *y'*, удовлетворяющие равенству:

$$a' \cdot x' + b' \cdot y' = \gcd(a', b'),$$

которое можно переписать в виде:

$$b \cdot x' + (a \bmod b) \cdot y' = \gcd(a, b).$$

Далее, поскольку при делении нацело остаток и целая часть связаны равенством:

$$a = \left\lfloor \frac{a}{b} \right\rfloor \cdot b + (a \bmod b),$$

то предыдущее тождество можно переписать в виде:

$$b \cdot x' + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b \right) \cdot y' = \gcd(a, b),$$

и, перегруппировывая слагаемые, получаем окончательно:

$$a \cdot y' + b \cdot \left(x' - \left\lfloor \frac{a}{b} \right\rfloor \cdot y' \right) = \gcd(a, b).$$

Коэффициенты при a и b в этом выражении совпадают с приведенными в алгоритме выражениями, что завершает доказательство.

30.3. Реализация расширенного алгоритма Евклида

Приведем, для определенности, программу, которая по двум заданным целым числам выводит их НОД и найденные коэффициенты.

Python

```
def gcd_ex(a, b):
  if b == 0:
    return a, 1, 0
  g, x1, y1 = gcd_ex(b, a % b)
  return g, y1, x1 - (a // b) * y1

a, b = map(int, input().split())
print(gcd_ex(a, b))
```

C++

```
#include <iostream>
using namespace std;

int GcdEx(int a, int b,
          int& x, int& y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }
```

```
  int x1, y1, g;
  g = GcdEx(b, a % b, x1, y1);
  x = y1;
  y = x1 - (a / b) * y1;
  return g;
}

int main() {
  int a, b;
  cin >> a >> b;
  int x, y, g;
  g = GcdEx(a, b, x, y);
  cout << g << ' ' << x << ' ' << y
       << endl;
}
```

К ВОПРОСУ О ЦЕЛОЧИСЛЕННЫХ ПЕРЕПОЛНЕНИЯХ

При реализации расширенного алгоритма Евклида даже на языке с целочисленными типами фиксированного размера, как C++, беспокоиться о целочисленных переполнениях **не стоит**. Коэффициенты, находимые расширенным алгоритмом Евклида, равно как и значения промежуточных выражений, не превосходят по модулю максимума из исходных чисел.

30.4. Пример решения задачи. Прыжки вперед и назад

Задача. Листая страницы различных книг рекордов, Петя обратил внимание на то, что человеку гораздо труднее прыгать спиной вперед: рекорды по длине прыжка с места отличаются более чем в полтора раза (рекорд прыжка спиной вперед, конечно, хуже). Размышляя над этим, Петя задумался: а насколько близко можно оказаться к стартовой точке, совершив некоторое число прыжков вперед и некоторое — назад? Петя исходит из предположений, что в одном направлении спортсмен прыгает всегда на одно и то же расстояние. Кроме того, запрещено возвращаться прямо в стартовую точку или оказываться до нее: во всякий момент времени после первого прыжка координата спортсмена должна оставаться строго положительной.

Входной файл содержит два целых числа F и B — длина одного прыжка вперед и длина одного прыжка назад ($1 \le B < F \le 10^9$).

Вывести требуется три целых числа — искомое минимально возможное расстояние, количество совершенных для этого прыжков вперед и количество совершенных прыжков назад. Изо всех вариантов ответа следует выбрать вариант с наименьшим числом прыжков вперед.

Примеры

Входные данные	**Требуемый результат**
`30 20`	`10 1 1`
Входные данные	**Требуемый результат**
`99 5`	`1 4 79`

Пояснения к примерам

В первом примере можно оказаться на расстоянии 10 от стартовой точки, совершив один прыжок вперед и один назад. Во втором примере можно оказаться на расстоянии 1 от стартовой точки, совершив четыре прыжка вперед и семьдесят девять прыжков назад: $99 \cdot 4 - 5 \cdot 79 = 1$.

Решение

Задача заключается в нахождении таких целых x и y, что следующее выражение принимает положительное и притом минимально возможное значение:

$$F \cdot x - B \cdot y,$$

причем при наличии нескольких оптимальных вариантов следует выбрать тот, который минимизирует x.

На самом деле, минимально возможное положительное значение этого выражения ищется легко — это просто **НОД** чисел F и B. Это можно доказать, заметив, что поскольку оба числа делятся на их НОД, в выражении $(F \cdot x - B \cdot y)$ этот НОД можно вынести за скобку, а, значит, все выражение делится на НОД. Таким образом:

$$\min_{\substack{x,y:\\ F\cdot x - B\cdot y > 0}} \left(F \cdot x - B \cdot y\right) = \gcd(F, B).$$

Теперь нам нужно найти соответствующие значения x и y:

$$F \cdot x - B \cdot y = \gcd(F, B),$$

причем $x > 0$ и должно принимать минимально возможное значение.

Если выполнить **расширенный алгоритм Евклида** и поменять знак у выданного им значения y, то мы найдем **одно из решений** этого уравнения, однако не обязательно с положительным или минимальным x.

Заметим следующий факт: если есть некоторое решение (x_0, y_0) этого уравнения, то также решением будет и **любая пара вида**:

$$\left(x_0 + k \cdot \frac{B}{\gcd(F, B)},\ y_0 - k \cdot \frac{F}{\gcd(F, B)}\right),$$

где k — произвольное целое число. В этом легко убедиться, подставив эти значения в уравнение: все прибавки взаимно уничтожаются и остается только НОД.

Более сильное утверждение заключается в том, что **только пары такого вида** являются решениями нашего уравнения. Это можно доказать, рассмотрев два произвольных решения: если мы подставим их в уравнение, вычтем одно тождество из

другого, разделим их на НОД и возьмем остаток от деления на $F / \gcd(F, B)$ или на $B / \gcd(F, B)$, то окажется, что разница между компонентами двух решений должна делиться на соответствующую величину.

Таким образом, задача свелась к следующей: имея некоторое решение (x_0, y_0), полученное из расширенного алгоритма Евклида, требуется **путем прибавления или вычитания** $B / \gcd(F, B)$ превратить x_0 в минимально возможное положительное значение. Эта задача уже чисто реализационная, и ее можно решить за $O(1)$, рассмотрев целую часть от деления x_0 на $B / \gcd(F, B)$ и аккуратно разобрав граничные случаи.

Итоговая асимптотика решения — $O(\log(\min(F, B)))$.

Реализация

В приведенных фрагментах кода мы опустим реализации расширенного алгоритма Евклида, поскольку они ничем не отличаются от приведенных ранее в этой главе.

Python

```
f, b = map(int, input().split())
g, x, y = gcd_ex(f, b)
fnorm = f // g
bnorm = b // g
k = x // bnorm
while x - k * bnorm <= 0:
  k -= 1
x -= k * bnorm
y += k * fnorm
print(g, x, -y)
```

C++

```
#include <iostream>
using namespace std;
int main() {
  int f, b;
  cin >> f >> b;
  int x, y, g;
  g = GcdEx(f, b, x, y);
  int fnorm = f / g;
  int bnorm = b / g;
  int k = x / bnorm;
  while (x - k * bnorm <= 0)
    --k;
  x -= k * bnorm;
  y += k * fnorm;
  cout << g << ' ' << x << ' '
       << -y << endl;
}
```

30.5. Пример решения задачи. Линейное диофантово уравнение с двумя переменными

Определение. Диофантовым называется уравнение, которое можно представить в виде поиска корней некоторого многочлена, причем в качестве корней рассматриваются только целые числа.

Линейным диофантовым уравнением называется такое диофантово уравнение, степень многочлена которого равна единице. Например, $5x + 2y - 3z = 1$, где x, y и z — целые, является линейным диофантовым уравнением.

Задача. Для заданных целых a, b и c решить линейное диофантово уравнение:

$$ax + by = c.$$

Входные данные состоят из трех разделенных пробелами целых чисел a, b, c, не превосходящих по модулю 10^9.

Вывести требуется два разделенных пробелами целых числа — искомое решение, либо «No solution», если решения не существует. Если решений несколько, можно вывести любое решение, элементы которого не превосходят по модулю 10^{18}.

Примеры

Входные данные	**Требуемый результат**
`3 4 2`	`2 -1`
Входные данные	**Требуемый результат**
`2 4 1`	`No solution`

Решение

Идеи из решения предыдущей задачи применимы и здесь: вне зависимости от значений x и y сумма $a \cdot x + b \cdot y$ будет делиться на НОД чисел a и b (за исключением особого случая $a = b = 0$ — мы рассмотрим его ниже отдельно). Следовательно, решения не существует, если c не делится на НОД a и b; в противном случае мы можем перейти к решению следующего уравнения:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

Решив это уравнение, ответ на исходную задачу мы сможем получить простым домножением найденных коэффициентов на c / gcd(a, b).

Это уравнение совпадает с уравнением, решаемым расширенным алгоритмом Евклида, за исключением того, что параметры a и b теперь могут быть отрицательными. Однако это не является проблемой — либо надо применить алгоритм Евклида к модулям чисел и затем поменять знаки у найденных коэффициентов, либо реализовать алгоритм Евклида так, чтобы он работал и для отрицательных чисел.

Наконец, вернемся к особому случаю $a = b = 0$, который решать расширенным алгоритмом Евклида невозможно (например, из-за деления на нуль при проверке c). Однако этот случай легко решается простым разбором вариантов: при $c = 0$ любая пара (x, y) является решением, а при прочих c решений не существует.

Асимптотика решения — $O(\log(\min(a, b)))$.

Реализация

Заметим, что приведенная ранее реализация расширенного алгоритма Евклида работает и для отрицательных чисел. Хотя в качестве НОД эта реализация может выдать как отрицательное, так и положительное число, главное свойство — что коэффициенты x и y должны соответствовать возвращаемому значению — будет выпол-

няться. Поэтому для решения текущей задачи разбирать вручную случаи отрицательных a или b не требуется.

Python

```
a, b, c = map(int, input().split())
g, x, y = gcd_ex(a, b)
if a == b == c == 0:
  print(0, 0)
elif a == b == 0 or c % g != 0:
  print("No solution")
else:
  x *= c // g
  y *= c // g
  print(x, y)
```

C++

```
#include <cstdint>
#include <iostream>
using namespace std;

int main() {
  int a, b, c;
  cin >> a >> b >> c;
  int x1, y1, g;
  g = GcdEx(a, b, x1, y1);
  if (a == 0 && b == 0 && c == 0) {
    cout << "0 0" << endl;
  } else if ((a == 0 && b == 0) ||
             (c % g != 0)) {
    cout << "No solution" << endl;
  } else {
    int64_t coeff = c / g;
    int64_t x = x1 * coeff;
    int64_t y = y1 * coeff;
    cout << x << ' ' << y << endl;
  }
}
```

Целочисленные переполнения в решении на C++

Отметим, что при вычислении итоговых коэффициентов требуется использовать 64-битный целочисленный тип, поскольку при домножении результатов расширенного алгоритма Евклида на «c / g» результат может достичь величин порядка 10^{18}.

30.6. Пример решения задачи. Обратное по составному модулю

Задача. Дано натуральное число a и модуль p. Требуется найти число, обратное к a по модулю p, т. е. решение следующего уравнения:

$$a \cdot x = 1 \mod p.$$

Входные данные состоят из натуральных чисел a и p, разделенных пробелом. Гарантируется, что $a < p < 10^9$. Вывести требуется одно число — искомый обратный элемент в промежутке $[0; p)$. Если решения не существует, нужно вывести «−1».

Примеры

Входные данные	Требуемый результат
7 11	8
Входные данные	**Требуемый результат**
3 20	7

Решение

Напомним, что, в случае когда модуль p простой, в *разд. 22.7* мы описывали несложное решение этой задачи, основанное на малой теореме Ферма и алгоритме двоичного возведения в степень. Однако в текущей задаче, где модуль p не обязательно является простым, этот метод не является практичным, поскольку требует факторизации модуля p.

В том же разделе приводилась ссылка на математический факт о том, что обратное к данному числу существует тогда и только тогда, когда это число и модуль **взаимно просты** (этот факт несложно вывести и самому из того утверждения, что сумма двух чисел с любыми коэффициентами всегда делится на их НОД). Таким образом, нам нужно вычислить НОД чисел a и p, и если он больше единицы, то в качестве ответа сразу можно вывести «−1».

Случай нулевого элемента

По условию нашей задачи случая $a = 0$ не бывает. Если же, однако, это было бы возможно, то надо было бы не забыть правильно обработать этот случай: обратного к нулю элемента не существует.

Рассмотрим теперь вспомогательное уравнение:

$$a \cdot x + p \cdot y = 1.$$

Если мы возьмем обе части этого уравнения по модулю p, то мы получим нашу исходную задачу. Следовательно, решив это уравнение, мы найдем тем самым и искомый обратный элемент. С другой стороны, это уравнение мы уже умеем решать с помощью расширенного алгоритма Евклида.

Асимптотика решения составит $O(\log p)$ или, если точнее, $O(\log(\min(a, p)))$.

Реализация

Единственный нетривиальный момент в реализации — это то, что возвращаемый расширенным алгоритмом Евклида коэффициент x может быть отрицательным, в то время как по условию нам требуется оставаться в интервале $[0; p)$. Это можно учесть дополнительной проверкой.

Python

```
a, p = map(int, input().split())
g, x, y = gcd_ex(a, p)
if g > 1:
  print(-1)
else:
  if x < 0:
    x += p
  print(x)
```

C++

```
#include <iostream>
using namespace std;
int main() {
  int a, p;
  cin >> a >> p;
  int x, y, g;
  g = GcdEx(a, p, x, y);
  if (g > 1) {
    cout << -1 << endl;
  } else {
    if (x < 0)
      x += p;
    cout << x << endl;
  }
}
```

Глава 31.
Задачи для самостоятельного решения

Для закрепления пройденного материала настоятельно рекомендуем читателю самостоятельно прорешать задачи этой главы.

31.1. Примеры задач

Подсказки к приведенным ниже задачам даны в *приложении*.

31.1.1. Задача. Цикл минимального веса

Дан взвешенный неориентированный граф. Требуется найти в этом графе простой цикл, сумма весов ребер которого минимальна. Простым циклом является такая последовательность вершин, в которой все элементы различны, кроме последнего элемента, который должен совпадать с первым элементом.

Входные данные содержат описание графа. В первой строке указано число вершин и число ребер. Каждая из последующих строк описывает очередное ребро графа в виде тройки чисел: номеров вершин-концов ребра (вершины нумеруются, начиная с единицы) и веса ребра (все веса лежат в интервале от 1 до 100). Число вершин не превосходит 500. Все ребра различны, и петли отсутствуют.

Вывести требуется «−1», если цикла нет, либо последовательность номеров вершин, входящих в искомый цикл. Если циклов с минимальным весов несколько, можно вывести любой.

Примеры

Входные данные	**Требуемый результат**
4 6 1 2 2 1 4 2 2 3 1 2 4 1 3 4 1	2 3 4 2

Входные данные	Требуемый результат
3 2 1 2 1 2 3 1	-1

31.1.2. Задача. Картельный сговор

Антимонопольная служба поручила вам реализовать эвристический алгоритм, обнаруживающий потенциально нелегальные отношения между компаниями. В наличии имеется база данных контрактов, представляющая собой пары вида (a_i, b_i), обозначающие, что компания под номером a_i оказала услугу компании b_i. Пара компаний (i, j) считается подозрительной, если обе этих компании оказывали друг другу услуги (возможно, через посредников).

Первая строка входных данных содержит число компаний n ($2 \le n \le 10^5$), число контрактов m ($0 \le m \le 10^5$), количество запросов k ($1 \le k \le 10^5$). Следующие m строк содержат описания контрактов в виде двух чисел a_i и b_i ($1 \le a_i, b_i \le n$). Последние k строк содержат запросы: пары x_i и y_i ($1 \le x_i, y_i \le n$).

Вывести требуется k строк — ответы на запросы, т. е. «YES», если пара (x_i, y_i) является подозрительной, и «NO» в противном случае.

Примеры

Входные данные	Требуемый результат
4 3 3 1 2 2 3 3 1 1 2 1 4 2 3	YES NO YES
Входные данные	**Требуемый результат**
4 0 1 1 2	NO

31.1.3. Задача. Превращение графа в сильно связный

Дан ориентированный граф. Требуется определить наименьшее число ребер, которые можно добавить, чтобы получить сильно связный граф.

Входные данные содержат описание графа. В первой строке указано число вершин и число ребер. Каждая из последующих строк описывает очередное ребро графа в

виде пары чисел: номер вершины-начала и номер вершины-конца. Число вершин и ребер не превосходит 10^5.

Примеры

Входные данные	Требуемый результат
4 4 1 2 1 3 2 4 3 4	1
Входные данные	**Требуемый результат**
5 2 1 2 3 4	3

Пояснения к примерам

В первом примере достаточно добавить ребро (4, 1). Во втором примере один из возможных вариантов — ребра (2, 3), (4, 5), (5, 1).

31.1.4. Задача. Отражение точки относительно прямой

Дана прямая, заданная двумя различными точками A и B, а также точка P. Требуется найти отражение точки P относительно прямой.

Входные данные состоят из трех строк, содержащих координаты точек A, B и P соответственно. Координаты — целые числа, по модулю не превосходящие 10^6.

Вывести требуется координаты искомой точки. Координаты следует выводить с такой точностью, чтобы относительная или абсолютная погрешность не превосходила 10^{-9}.

Примеры

Входные данные	Требуемый результат
0 0 1 0 3 2	3 -2
Входные данные	**Требуемый результат**
1 1 0 -1 2 -1	0 3

31.1.5. Задача. Пересечение окружностей

Две окружности заданы координатами своих центров и радиусами. Требуется найти число точек их пересечения.

Входные данные состоят из двух строк, каждая из которых описывает одну окружность. Каждая строка содержит три разделенных пробелами целых числа: абсциссу, ординату и радиус; координаты и радиус не превосходят 10^6, и радиус положителен.

Если точек пересечения конечное число, то следует вывести количество точек в первой строке и координаты самих точек в последующих строках; точки можно выводить в произвольном порядке. Координаты следует выводить с такой точностью, чтобы относительная или абсолютная погрешность не превосходила 10^{-9}. Если число точек пересечения бесконечно, следует вывести «−1».

Примеры

Входные данные	**Требуемый результат**
0 0 3 5 0 4	2 1.8000000000 -2.4000000000 1.8000000000 2.4000000000
Входные данные	**Требуемый результат**
2 2 1 -2 -2 1	0

31.1.6. Задача. Определение направления обхода многоугольника

Многоугольник задан координатами своих вершин, заданных в порядке некоторого обхода (по часовой стрелке или против нее). Требуется определить направление обхода, использованное во входных данных.

Первая строка входных данных содержит число n вершин ($1 \le n \le 10^5$), и последующие n строк содержат координаты вершин многоугольника. Координаты — целые числа, по модулю не превосходящие 10^9. Многоугольник не обязательно выпуклый. Гарантируется, что площадь многоугольника отлична от нуля и самопересечения отсутствуют.

Вывести требуется строку «cw» или «ccw» в зависимости от направления обхода точек, указанных во входных данных.

Примеры

Входные данные	**Требуемый результат**
3 0 0 1 0 1 1	ccw

Входные данные	Требуемый результат
4 -1 -1 -1 2 1 1 1 -2	cw

31.1.7. Задача. Проверка многоугольника на выпуклость

Определение. Многоугольник называется выпуклым, если он лежит по одну сторону от любой прямой, проходящей через одну из его сторон.

Отметим, что, в отличие от так называемой строгой выпуклости, это определение допускает вырожденные многоугольники (имеющие нулевую площадь) и многоугольники, имеющие стороны, лежащие на одной прямой.

Задача. Многоугольник задан координатами своих вершин, заданных в порядке некоторого обхода (по часовой стрелке или против нее). Требуется определить, является ли он выпуклым.

Первая строка входных данных содержит число n вершин ($1 \le n \le 10^5$), и последующие n строк содержат координаты вершин многоугольника. Координаты — целые числа, по модулю не превосходящие 10^9. Многоугольник может быть вырожденным и может иметь самопересечения.

Вывести требуется строку «yes» или «no» в зависимости от того, выпуклый многоугольник или нет.

Примеры

Входные данные	Требуемый результат
5 0 0 1 0 1 0 1 1 1 2	yes
Входные данные	**Требуемый результат**
4 0 0 2 2 2 0 0 2	no

Заключение

Методики решения задач

Примеры, разобранные на протяжении книги, дают некоторое представление о том, как можно «подступиться» к новой задаче. Подведем здесь в качестве итога некоторые из ключевых идей.

- Переформулировать условие задачи в абстрактных, формальных терминах.
  - Абстрактное описание — например, в виде графа, — позволяет отбросить несущественные детали, сфокусироваться на самом важном и, возможно, быстрее заметить связь с уже известными алгоритмами.
- Разобрать вручную примеры входных и выходных данных из условия.
  - Это помогает лучше «почувствовать» задачу. Заодно это самопроверка: правильно ли было понято условие?
- Придумать и вручную проанализировать пару самостоятельно придуманных входных наборов.
- Оценить на основе входных ограничений, какая асимптотика требуется от решения.
  - Иногда уже знание асимптотики позволяет отсечь множество неподходящих идей и подходов.
- Перебрать мысленно и попробовать применить каждый из стандартных подходов.
  - Например: можно ли представить задачу как поиск кратчайшего пути в графах? Или верен ли будет жадный алгоритм? Можно ли применить метод динамического программирования?
- Попытаться решить более простую задачу или же исходную задачу, но для какого-то частного случая.
  - Можем ли мы решить ту же задачу, но для тысячи элементов вместо миллионов из условия? Если задача связана с произвольными графами, то можем ли мы решить ее для случая дерева? Решив упрощенную задачу, можно пытаться обобщить алгоритм для решения исходной задачи.

Благодарности

Я выражаю особую благодарность Федоровой Антонине Гавриловне, руководителю Центра олимпиадной подготовки Саратовского ГУ, и Мирзаянову Михаилу Расиховичу, многолетнему тренеру и лектору Центра. Центр олимпиадной подготовки стал для меня, как и для множества других студентов и школьников, не только местом изучения олимпиадных алгоритмов, но и средой общения, взаимного обмена идеями и стимулирующей личное развитие взаимной конкуренции.

Без школы, пройденной в Центре олимпиадной подготовки, эта книга была бы невозможна.

Я признателен моей семье за поддержку и помощь: написание этой книги, а за несколько лет до этого — занятия олимпиадами и работа над статьями об алгоритмах — заняли несчетное количество часов свободного времени.

Также я благодарен издательству «БХВ-Петербург» за предоставленную возможность опубликовать эту книгу.

Послесловие

«В определенном смысле я никогда не умру, потому что частицы моего Я, мои статьи и книги, разлетелись осколками на века, попав на плодородную почву молодых пытливых умов, изменив ход их бытия, и теперь уже непросто провести границу, где они, а где Я. Да и Я — это на самом деле никакой не я, а продукт брожения мозга, читавший определенные книги, слушающий срывающую крышу музыку и так далее по списку. Я же не вещь в себе. Во мне живут осколки тех, кто вспыхнул до меня. И так по эстафете. А потому мы приходим к тому, что вначале было слово. Именно слово делает людей бессмертными. Пока кто-то грезит о возможности скопировать свое сознание в компьютер будущего, другие копируют свое сознание посредством письменности».

Крис Касперски

В отличие от статей Криса Касперски (настоящее имя Николай Лихачев), бывшего талантливым специалистом в области компьютерной безопасности и программистом, наша книга не содержит абсолютно новых идей и оригинальных умозаключений. Эта книга — лишь попытка систематизировать и собрать в одном месте эти разрозненные «осколки», помочь читателю сориентироваться в их многообразии и принять «эстафету».

Целый спектр тем — например, потоки в графах, сбалансированные деревья, теория игр — остались за скобками данной книги. Желающие еще более углубленно познакомиться со спортивным программированием могут попробовать изучить эти темы по материалам в Интернете. Множество великолепных статей содержатся на сайте Codeforces, созданном Михаилом Мирзаяновым. Также — позволим себе небольшую саморекламу — автор данной книги опубликовал более сотни статей по этим и другим темам на сайте **e-maxx.ru/algo**.

Мы повторим данный уже много раз в этой книге совет: лучший способ закрепить новую тему — через решение соответствующих задач и, затем, чтение разборов решений. Очень важно посвящать бóльшую часть времени на решение задач на скорость, т. е. в формате соревнований, поскольку в таком режиме мозг тренируется решать задачи **быстро**.

Приложение. Решения задач

Задачи из главы 8

8.1.1. Задача. Фибоначчилистник

Перефразируя условие, получаем, что число новых ветвей, появляющихся в день i, равно числу почек в день $i - 1$. А число новых почек, появляющихся в день i, равно числу новых ветвей, появляющихся в день $i - 1$. Исходя из этого, можно заметить, что число ветвей подчиняется тому же закону, что и числа Фибоначчи.

Поскольку, по условию, каждая ветвь дает ежедневно по новому листу, начиная со второго дня своего существования, получаем, что ответ на задачу равен сумме количеств ветвей среди всех дней с первого до $(n + 1)$-го:

$$\sum_{i=1}^{n-1} F_i.$$

Примечание

Хотя речь в задаче идет о вымышленном растении, числа Фибоначчи действительно регулярно встречаются в природе. Согласно исследованиям, ячейки на ананасах, как правило, образуют несколько цепочек с количествами 8–13–21 или 5–8–13. Лепестки в соцветиях артишока обычно формируют кольца с длинами, образующими последовательность Фибоначчи. Есть и несколько других примеров; впрочем, скептики утверждают, что это лишь единичные совпадения на фоне огромного разнообразия видов и форм живой природы.

Для решения задачи достаточно запрограммировать подсчет чисел Фибоначчи и их суммирование. Впрочем, можно немного упростить решение, если воспользоваться известной формулой для суммирования чисел Фибоначчи, которая дает следующий ответ:

$$F_{n+1} - 1.$$

Эту формулу легко доказать по индукции (т. е. проверив ее при $n = 1$ и затем, предполагая ее истинность для $k - 1$, показывая ее истинность и для k).

Приведем и альтернативное доказательство, которое, в отличие от метода математической индукции, конструктивно и не предполагает знания ответа наперед. Для этого развернем формулу чисел Фибоначчи следующим образом: $F_i = F_{i+2} - F_{i+1}$; тогда при суммировании этих выражений все члены суммы взаимно уничтожат друг друга, за исключением F_{n+1} и единицы, что и дает искомое тождество.

8.1.2. Задача. Картина на кафеле

Задачу можно решать двумя способами: либо перебирая все плитки стены и вычисляя пересечение с картиной, либо посчитав путем деления нацело номера пограничных плиток. Второй способ решения — более быстрый, однако требует аккуратного разбора случаев. Первый способ решения проще в реализации и достаточно быстр при заданных ограничениях.

8.1.3. Задача. Лексикографически минимальное исправление перестановки

Основной принцип решения остается тем же, что и в разобранной в книге задаче «Исправление перестановки» *(см. разд. 6.4)*. Отличие заключается в том, что мы должны, двигаясь по последовательности слева направо и исправляя ее, всякий раз применять по возможности наименьшую замену. Для этого необходимо учесть два момента.

Во-первых, следует хранить список текущих «недостающих» чисел в отсортированном порядке и брать из него всякий раз наименьший элемент.

Во-вторых — и это более каверзный момент — при наличии дубликатов в исходной последовательности мы должны аккуратно выбирать, какие дубликаты следует заменить, а какие — оставить. Проиллюстрируем это на двух примерах: «1 1 1» и «3 3 3»; правильным ответом для обоих будет «1 2 3», однако в первом примере для его достижения нужно заменить второй и третий элемент, а во втором примере — первый и второй. Правильной стратегией является следующая: всегда пытаться заменять каждый дубликат, за исключением случая, когда число-замена больше по величине. Однако эта стратегия должна быть проигнорирована, если замена является неизбежной (т. е. если один из предыдущих дубликатов уже был оставлен без изменений) или обязана не произойти (т. е. если это последний дубликат, и все предыдущие уже были заменены).

8.1.4. Задача. XOR нечетных чисел от 1 до *n*

Заметим, что если мы отбросим самый младший бит в каждом числе XOR-суммы, то мы придем к задаче нахождения XOR-суммы всех чисел вплоть до данного числа. Решение этой задачи было разобрано в *разд. 6.6*.

Осталось определить младший бит ответа. Для этого достаточно лишь определить четность количества чисел в сумме; говоря формально, этот младший бит равен $\lceil n / 2 \rceil \bmod 2$.

8.1.5. Задача. Манхэттенский центр

В этой задаче при вычислении расстояний используется манхэттенская метрика: $|x_1 - x_2| + |y_1 - y_2|$. Если внимательно посмотреть на эту формулу, то видно, что она распадается на два слагаемых, одно из которых зависит только от абсцисс, дру-

гое — только от ординат. Следовательно, задачу можно решать по каждой из осей координат независимо.

Решение одномерной задачи уже было рассмотрено нами в книге — напомним, оно заключается в поиске медианы среди значений координат. Таким образом, мы должны найти медиану абсцисс входных точек, а также медиану ординат входных точек — и эти две медианы дадут координаты искомой оптимальной точки.

8.1.6. Задача. Превращение в одинаковые числа

Очевидно, всегда выгодно изменять элементы по кратчайшему пути, т. е. либо только увеличивать, либо только уменьшать их, до тех пор пока общее значение не будет достигнуто. Запишем задачу формально — требуется найти следующую величину:

$$\min_x \sum_{i=1}^{n} |a_i - x|.$$

Как можно заметить, эта задача — в точности задача поиска геометрической медианы в одномерном случае. Эту задачу мы уже умеем решать: оптимальный x равен медиане последовательности a_i (или, если n четно, может принимать любое значение между двух ее медиан). После нахождения x единственное, что осталось сделать, — это посчитать сумму модулей разностей, что можно сделать простым проходом по последовательности.

Асимптотика решения — $O(n \log n)$, если медиана ищется сортировкой. (Как было упомянуто в тексте главы, медиану можно искать и быстрее, однако в данном случае этого не требуется.)

Задачи из главы 10

10.11.1. Квадратный кафель

Ответом является $\gcd(l, w)$, поскольку по условиям замощения размер плитки должен быть делителем и длины, и ширины, и в задаче требуется найти наибольший из всех таких делителей.

10.11.2. НОД диапазона натуральных чисел

Перебирать все числа в указанном отрезке и считать НОД от всех них было бы слишком медленным решением.

Вместо этого заметим следующий факт: если $l = r$, то ответ равен этому же числу; если же $l < r$, то ответ всегда равен единице.

В справедливости этого утверждения можно убедиться, например, проследив шаги алгоритма Евклида при вызове для пары $(l, l + 1)$.

10.11.3. Подмножество с заданным НОД

Решение — жадный алгоритм: следует оставить все числа, которые делятся на k. После этого надо посчитать их НОД, и если он не равен k, то решения не существует; в противном случае можно вывести все эти числа в качестве ответа.

Для доказательства корректности достаточно заметить, что мы ничего не теряем, если мы добавляем в набор любое число, делящееся на k: НОД текущего набора либо не изменится, либо улучшится за счет своего уменьшения.

10.11.4. НОК дробей

Решением является следующая дробь:

$$\frac{\mathrm{lcm}(a_1, ..., a_n)}{\gcd(b_1, ..., b_n)}.$$

10.11.5. Целые точки на отрезке

Ответ равен:

$$1 + \gcd\left(|x_1 - x_2|, |y_1 - y_2|\right).$$

10.11.6. НОД многочленов

Для решения необходимо реализовать аналог алгоритма Евклида для многочленов, на каждом шаге отнимая от одного многочлена другой, домноженный на такой многочлен, чтобы в итоге старший коэффициент обнулился.

Для замедления роста коэффициентов в многочленах имеет смысл сокращать после каждого шага их на наибольший общий делитель.

Задачи из главы 17

17.1.1. Задача. Пары фиксированной суммы, с повторами

Отсортируем последовательность и будем идти по ней двумя указателями, т. е. перебирая $i = 1...n$ и двигая каждый раз j до первого элемента, имеющего нужное значение. После того как нужное j найдено, продвинем оба указателя дальше, до тех пор пока не найдем концы блоков повторяющихся элементов. Тем самым мы найдем количества дублей a_i и a_j и увеличим ответ на их произведение. Единственный каверзный случай — когда $a_i = a_j$, поскольку в этом случае ответ надо увеличить на $c(c - 1) / 2$ — столько различных пар можно образовать из блока длиной c.

17.1.2. Задача. Подстроки с заданным числом единиц

Один из способов эффективного решения задачи — два указателя, один из которых перебирается по всевозможным позициям начала подстроки, а второй указывает на

самую правую из возможных позиций окончания такой строки (т. е. на такую позицию, что в подстроке получается ровно k единиц, и увеличение позиции невозможно). Понадобится также вспомогательный третий указатель, который хранит самую левую из возможных позиций окончания строки; тогда ответ надо будет всякий раз увеличивать на разность второго и третьего указателей.

Асимптотика решения — $O(n)$, так как каждый из трех указателей пробегает по входной строке только единожды.

17.1.3. Задача. Пары с суммой в диапазоне

Отсортируем числа входной последовательности. Дальше задачу можно решить либо за $O(n)$ методом двух указателей, либо за $O(n \log n)$ с помощью двоичного поиска.

При решении методом двух указателей будем перебирать всевозможные значения индекса p в порядке увеличения и поддерживать указатель на наименьшее q такое, что $a_p + a_q \geq l$. Для учета границы r будем поддерживать третий указатель, который будет указывать на наибольшее q такое, что $a_p + a_q \leq r$. Для каждого p ответ нужно увеличить на количество подходящих q, которое равно увеличенной на единицу разности между двумя указателями. С увеличением первого указателя другие два указателя могут только уменьшаться, откуда и следует линейное время работы.

Альтернативное решение — на основе двоичного поиска. Будем перебирать всевозможные значения индекса p и искать двумя двоичными поисками левую и правую границы на возможные значения q (например, в C++ это можно сделать библиотечными функциями `lower_bound` и `upper_bound`). Как и в первом решении, ответ надо будет увеличивать на разность между найденными границами (увеличенную на один, если границы берутся включительно).

17.1.4. Задача. Подотрезок с наибольшим минимумом

Будем перебирать всевозможные стартовые позиции подотрезка. Для быстрого определения минимума на текущем подотрезке заведем структуру данных, поддерживающую добавление, удаление элементов и поиск минимального элемента (например, с асимптотикой операций $O(\log n)$ в C++ можно использовать set, а в Python — heapq). Тогда изначально наполним эту структуру данных первыми k элементами, а затем при сдвигании текущего подотрезка вправо на один будем удалять самый левый элемент старого подотрезка и добавлять самый правый элемент нового подотрезка.

Условно это решение можно классифицировать как основанное на методе двух указателей. Асимптотика решения составит $O(n \log n)$.

17.1.5. Задача. Произведения-кубы

Аналогично разобранной в книге задаче о произведениях-кубах рассмотрим факторизации чисел. У точного куба степени всех простых — кратны трем. Таким обра-

зом, мы сразу можем удалить полные кубы из входных чисел. В факторизациях оставшихся чисел каждое простое входит в степени 1 или 2; можно понять, что для получения точного куба нужно домножать на число, имеющее, наоборот, степени 2 и 1 у соответствующих простых.

Это дает нам решение за $O(n\sqrt{m} + n\log n)$, где $m = \max_{i=1\ldots n}\{a_i\}$. Обработка каждого числа будет заключаться в факторизации, удалении точного куба и вычислении значения нужного множителя, запросе количества по этому значению в «словаре», увеличении количества в «словаре».

17.1.6. Задача. Мытье окон

Тривиальное решение — перебрать стартовую позицию и подсчитать число единиц среди последующих k позиций — слишком медленно: оно имело бы асимптотику $O(nk)$, т. е. в худшем случае порядка 10^{12} операций.

Ускорим это решение. Нам нужно научиться быстро вычислять количество единиц в заданном отрезке $[x; x + k - 1]$. Для этого предварительно подсчитаем массив частичных сумм, т. е. величины p_i, определенные следующим образом:

$$p_i = \sum_{j=1}^{i} a_j.$$

Тогда подсчет числа единиц в нужном отрезке можно сделать за $O(1)$: надо просто взять разность $p_{x+k-1} - p_{x-1}$.

Итого решение получается следующим: сначала одним циклом подсчитать величины p_i, затем в другом цикле перебрать стартовую позицию x, для каждой позиции вычислить число единиц путем вычитания двух предпосчитанных значений и положить ответ равным наименьшему из полученных чисел.

Асимптотика этого решения — $O(n)$.

Альтернативное решение

Эту задачу можно решить и по-другому, с использованием метода двух указателей: сначала посчитаем сумму первых k элементов, получив тем самым число единиц для позиции 1; затем пересчитаем эту сумму для позиции 2, вычтя значение a_1 и прибавив значение a_{k+1} и т. д. Это решение будет иметь ту же асимптотику $O(n)$ по времени, однако оно не будет требовать дополнительной памяти, помимо $O(1)$ места для хранения счетчиков. Если линейное потребление памяти не является проблемой, то выбор решения является вопросом предпочтений.

17.1.7. Задача. Санскритская поэтика

Для решения этой задачи можно воспользоваться методом динамического программирования. Обозначим входное число через n. Рассмотрим первый слог метра: это «лагу» или «гуру». Если первый слог «лагу», то число метров с этим первым слогом равно ответу для $n - 1$. Если же первый слог «гуру», то число метров равно

ответу для $n - 2$. Таким образом, мы можем выразить ответ $d[n]$ через ответы для меньших значений:

$$d[n] = d[n-1] + d[n-2].$$

Это дает нам решение задачи в форме динамического программирования: будем пошагово вычислять ответ — сначала для длины, равной единице, затем двойке, и т. д. вплоть до n; всякий раз очередной ответ будет равняться сумме двух предыдущих ответов. Базой будут являться случаи $d[0] = 0$ и $d[1] = 1$.

Асимптотика этого решения составит $O(n)$.

> ***Числа Фибоначчи***
>
> Внимательный читатель мог заметить, что полученная выше формула — не что иное, как формула для вычисления очередного члена ряда Фибоначчи. К этому результату можно прийти и другим путем; динамическое программирование является лишь одним из возможных инструментов для достижения этого решения. В каком-то смысле все упражнение является шуткой, потому что это задание — формулировка той самой задачи, которую во II в. до н. э. решал древнеиндийский математик Пингала, и об этом было кратко упомянуто в главе о числах Фибоначчи.

17.1.8. Число способов набрать сумму

Для решения задачи будем считать динамику d, в которой d_j равно числу способов набрать сумму j.

Будем добавлять монеты в рассмотрение по одной, пересчитывая при этом d. Изначально $d_0 = 1$, а все остальные элементы равны нулю. При добавлении в рассмотрение очередной монеты a_i мы совершаем переходы из каждого состояния j в состояние $j + a_i$, т. е. d_{j+a_i} увеличивается на значение d_j. При этом индексы j надо перебирать в порядке убывания, чтобы использовать каждую монету не более одного раза.

17.1.9. Задача о неограниченном рюкзаке

В этой задаче есть два отличия по сравнению с изначальной задачей о рюкзаке: неограниченный запас предметов каждого вида и необходимость вывода выбранных предметов.

Как и в изначальной задаче о рюкзаке, для решения мы можем использовать динамику вида d_j, равную наибольшей стоимости, которую можно достичь для веса j. При добавлении в рассмотрение очередного предмета i мы должны совершать переходы из каждого состояния j в состояние $j + w_i$, обновляя значение d_{j+w_i} значением $d_j + v_i$.

Теперь остановимся подробно на изменениях, которые нам нужно внести в решение:

1. Для поддержки возможности использовать предмет несколько раз достаточно лишь одного изменения в алгоритме: перебирать индексы j в порядке возрастания, а не в порядке убывания. Тем самым мы можем неоднократно применить один и тот же предмет для получения определенной суммы.

2. Для вывода выбранных предметов будем запоминать массив «предков», т. е. для каждого состояния j будем хранить индекс p_j предмета, использованного для достижения текущего оптимального d_j. Тогда для восстановления ответа надо лишь «раскрутить» эту цепочку: сначала взять p_m, затем перейти к состоянию $m - w_{p_m}$ и вывести его значение p и т. д.

Асимптотика решения сохранится: $O(n \cdot m)$.

17.1.10. Задача об ограниченном рюкзаке

Аналогично другим задачам о рюкзаке будем добавлять предметы по одному и совершать переходы из каждого состояния динамики j, однако теперь мы будем переходить в состояния $j + w_i$, $j + 2w_i$ и т. д. до $j + c_i w_i$. Чтобы не превосходить количество c_i, состояния j будем перебирать при этом в порядке убывания.

Задачи из главы 24

24.1.1. Поиск подпрямоугольника с наибольшей суммой

Эту задачу можно решить за $O(n^2 \cdot m)$, перебрав номера первой и последней строки прямоугольника, построив массив сумм по каждому столбцу в получившейся «полосе» матрицы и применив алгоритм решения одномерной задачи о наибольшей сумме.

Отметим: чтобы быстро строить массив сумм при фиксированных номерах строк, можно либо поддерживать эти суммы в процессе перебора строк, либо пользоваться предварительно подсчитанными частичными суммами в каждом из столбцов.

24.1.2. Пересадки в метро

Если представить карту всего метро в виде графа, то задача поиска кратчайшего пути будет нестандартной: каждому ребру приписана метка, и требуется найти путь, вдоль которого происходит меньше всего смен меток. Изученные нами алгоритмы не позволяют решать такую задачу.

Однако можно свести задачу к стандартной, построив альтернативный граф. Создадим граф с $n + k$ вершинами: по одной вершине для каждой станции и по одной вершине — для каждой линии метро. Для каждой линии соединим ее вершину со всеми соответствующими вершинами-станциями (друг с другом соединять станции не будем).

Утверждается, что кратчайший путь в таком графе будет соответствовать оптимальному с точки зрения числа пересадок пути. Например, если вершины x и y лежат на одной линии метро, кратчайший путь будет иметь длину 2: одно ребро из станции x в вершину-линию и второе ребро из вершины-линии в станцию y. Если же, например, из x до y можно добраться с одной пересадкой, то кратчайший путь

будет иметь длину 4: станция x, затем линия №1, затем станция пересадки, затем линия №2, затем станция y.

Таким образом, можно запустить стандартный обход в ширину для поиска кратчайшего пути в этом альтернативном графе, и ответом на задачу будет кратчайший путь, деленный пополам и уменьшенный на единицу. Особый случай, который надо будет учесть отдельно, — случай, когда стартовая и конечная вершины совпадают.

24.1.3. Основание столицы

Задача заключается в добавлении в заданный орграф наименьшего количества ребер так, чтобы все вершины были достижимы из вершины №1.

Во-первых, заметим, что для каждой вершины $v > 1$, имеющей нулевую степень входа (т. е. в которую не входит ни одно ребро), потребуется добавить по одному ребру. В самом деле, без добавления такого ребра не будет никакой возможности добраться до этой вершины из вершины №1.

Во-вторых, при наличии выбора мы ничего не теряем, если будем добавлять ребра именно в такие вершины с нулевой степенью входа. Здесь имеется в виду то, что добавленное ребро, оканчивающееся на вершине с нулевой степенью входа, автоматически «покрывает» все достижимые из нее вершины.

Однако этих наблюдений недостаточно для полного решения задачи, поскольку в графе могут быть циклы. Если в цикл не входит ни одного ребра извне, то мы обязаны добавить хотя бы одно такое ребро — и это несмотря на то что все вершины цикла имеют положительную степень входа. Таким образом, концептуально нам требуется «сжать» все циклы в вершины и найти все вершины/циклы с нулевой степенью входа.

Хотя этой цели можно достичь с помощью более сложного алгоритма, рассматриваемого в последующих главах книги (алгоритм поиска компонент сильной связности и конденсации), у данной задачи есть и более простое жадное решение.

Упорядочим вершины в порядке, обратном к порядку выхода обхода в глубину из них. Утверждается, что первая вершина в этом списке будет либо иметь нулевую степень входа, либо принадлежать циклу с нулевой степенью входа. Поэтому увеличим ответ на единицу и мысленно удалим из графа все достижимые из нее вершины. Повторим то же самое для следующей не удаленной еще вершины списка и т. д., до тех пор пока не будут обработаны все вершины.

С точки зрения реализации этот проход по всем вершинам и удаление достижимых частей графа проще всего реализовать с помощью обхода в глубину. «Удаление» — это просто неочищение массива `visited`. По сути получается, что ответ на задачу — это количество запусков этого второго обхода в глубину (минус один, поскольку запуск, посетивший вершину №1, учитывать не надо).

Асимптотика решения — $O(n + m)$, поскольку решение представляет собой две серии обходов в глубину.

24.1.4. Поиск наибольшего нулевого подпрямоугольника

Сведем эту задачу к набору одномерных задач.

Для этого переберем одну из границ искомого подпрямоугольника — например, его последнюю строку. Рассмотрим для каждой ячейки этой строки ближайшую к ней сверху единицу. Тогда задача сводится к тому, чтобы найти наибольший по площади подпрямоугольник, вписанный в эти ограничения. Решение этой задачи мы уже разобрали в тексте главы, и решается она за линейное время. Ближайшие сверху единицы также можно поддерживать за $O(1)$ для каждой ячейки, если хранить эту информацию в процессе перебора строк.

Таким образом, вся задача решается за линейное время путем запуска n раз алгоритма вписывания наибольшего подпрямоугольника.

24.1.5. Перемножение по большому модулю

Единственная сложность этой задачи — в величине входных чисел, поскольку произведение таких чисел не будет помещаться в 64-битный тип данных.

В зависимости от языка программирования и доступных средств эту задачу можно решать по-разному. Например, в языке Python эта задача не составит никакой сложности, поскольку встроенный числовой тип данных поддерживает числа произвольной длины. Для языка C++ некоторые компиляторы предоставляют тип данных __int128, который также позволил бы выполнить требуемые операции. Однако предложим простое решение для ситуации, когда по той или иной причине эти библиотечные решения недоступны.

Разумеется, есть различные способы быстрого вычисления произведения по модулю: например, можно разбить оба множителя на две 32-битных половинки и выразить искомый ответ через произведения этих половинок.

Однако опишем и альтернативный подход, основанный на методе двоичного возведения в степень (что в очередной раз демонстрирует гибкость этого алгоритма!). В самом деле, если b — четно, то можно вдвое уменьшить его путем удвоения a:

$$a \cdot b = (2a) \cdot \frac{b}{2},$$

а если b нечетно, то верно:

$$a \cdot b = a + a \cdot (b - 1).$$

Таким образом, за $O(\log b)$ шагов мы можем посчитать искомое произведение, совершая только операции сложения и удвоения — что уже можно совершать по модулю, не выходя за пределы 64-битных типов данных.

24.1.6. Подсчет числа путей с петлями и кратными ребрами

Эта задача решается аналогично рассмотренной в тексте задачи для простых графов, т. е. возведением матрицы смежности в степень.

Единственное отличие — в том, что теперь матрица смежности должна будет содержать информацию о петлях и кратных ребрах, т. е. ее диагональ теперь не обязательно будет нулевой (поскольку ребру-петле соответствует ячейка на диагонали), и ячейки матрицы могут содержать значения, большие единицы (поскольку нескольким кратным ребрам соответствует одна и та же ячейка матрицы).

С точки зрения реализации требуемая матрица смежности строится следующим образом: для каждого ребра (i, j) мы увеличиваем на единицу значение в $g[i][j]$ и $g[j][i]$, если $i \neq j$, и увеличиваем на единицу значение в $g[i][i]$, если $i = j$.

Итоговая асимптотика решения остается такой же, как и у исходной задачи, — $O(n^3 \log k)$.

24.1.7. Подсчет числа путей с длиной, не превосходящей заданную

Из решения предыдущих задач мы знаем, как подсчитывать пути фиксированной длины. Однако запускать такое решение в цикле, суммируя ответы по всевозможным k было бы слишком медленно, учитывая, что по условию k может достигать значений порядка 10^9.

Для эффективного решения задачи модифицируем граф таким образом, чтобы к нему можно было применить тот же алгоритм без необходимости перебора всех возможных длин. Для этого раздвоим каждую вершину, так что для каждой вершины v исходного графа есть дополнительная вершина v'. Эти дополнительные вершины свяжем ребрами с вершинами исходного графа:

$$v \to v',$$

а также добавим ребра-петли для каждой из дополнительных вершин:

$$v' \to v'.$$

Преимущество этого модифицированного графа в том, что он позволяет «удлинять» любой путь до сколь угодно большей длины. В самом деле, если мы рассмотрим любой путь из i в j в исходном графе, то в модифицированном графе также существует путь из i в j', причем к концу этого пути можно приписать сколь угодно большое число проходов по петле $j \to j'$.

Таким образом, для получения ответа на задачу требуется найти число путей длины $k + 1$ в модифицированном графе.

Итоговая асимптотика решения остается такой же, как и у исходной задачи, — $O(n^3 \log k)$. Следует учитывать, однако, что на практике это решение будет работать приблизительно в восемь раз дольше, чем исходный алгоритм, из-за увеличения числа вершин вдвое при модификации графа.

24.1.8. Поиск самого дешевого пути с заданным числом ребер

Для решения этой задачи воспользуемся подходом, аналогичным предыдущим задачам. Итак, сначала попытаемся решить задачу с помощью динамического программирования, постепенно подсчитывая ответ для все больших значений k.

При $k = 0$ ответ определяется тривиальным образом: из любой вершины можно дойти в себя же с помощью пути нулевой стоимости и нельзя дойти ни до каких других вершин.

Предположим, что мы знаем ответ для $k - 1$, т. е. известен массив $d_{k-1}[i][j]$, содержащий стоимости самых дешевых путей с $k - 1$ ребрами для каждой пары вершин (i, j). Для удобства в случае отсутствия пути положим значение ячейки в этой таблице равным бесконечности.

Теперь нам требуется научиться вычислять значения $d_k[i][j]$. Как и в предыдущих задачах, переберем все возможные варианты, по которым путь может выйти из вершины i, и тем самым выразим ответ через уже известные значения:

$$d_k[i][j] = \min_{v \in N(i)} \left(g[i][v] + d_{kk-1}[v][j] \right),$$

где $N(i)$ — множество всех вершин-соседей i, а $g[i][v]$ — стоимость ребра (i, v). Тем самым мы получили возможное решение этой задачи с помощью метода динамического программирования, работающее с асимптотикой $O(n^3 \cdot k)$.

Каким образом можно ускорить это решение? Формула динамического программирования не сводится к матричному произведению: вместо суммы произведений в формуле присутствует минимум из сумм. Тем не менее алгоритм **двоичного возведения в степень применим** и к такой альтернативной операции перемножения матриц.

В самом деле, единственным требованием для применимости алгоритма двоичного возведения в степень является ассоциативность операции, т. е. возможность изменить порядок вычисления операций следующим образом:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C).$$

Мы не будем приводить здесь строгого доказательства ассоциативности нашего альтернативного матричного произведения, однако укажем неформальные соображения в пользу этого утверждения. Если мы вернемся к интерпретации этого альтернативного произведения как к способу найти оптимальные пути заданной длины в графе, то получим, что, зная кратчайшие пути между всеми парами вершин для длин a, b и c, мы можем найти кратчайшие пути для длины $a + b + c$ любым из двух способов: либо применяя формулу динамического программирования сначала для вычисления ответа для $a + b$ и затем для объединения его с c, либо вычисляя сначала ответ для $b + c$ и затем объединяя его с ответом для a. Это (математически нестрогое) обоснование показывает корректность применения алгоритма двоичного возведения в степень.

Подводя итог, решение будет заключаться в построении матрицы смежности для заданного взвешенного графа. Каждая ячейка матрицы смежности должна содержать стоимость ребра между соответствующей парой вершин или бесконечно большую величину при отсутствии такого ребра (при реализации в качестве бесконечности можно либо взять константу, заведомо большую стоимости любого пути — в данной задаче это 10^{18} — либо использовать специальное значение, такое,

как «−1», и обрабатывать его при арифметических операциях так, как если бы оно было бесконечно большим). После построения таким образом матрицы смежности решение должно возвести ее в степень k с помощью алгоритма двоичного возведения в степень — полученная матрица и будет содержать искомые стоимости путей.

Итоговая асимптотика решения составит $O(n^3 \log k)$.

Задачи из главы 31

31.1.1. Цикл минимального веса

Для решения задачи будем **перебирать стартовую вершину**, лежащую на искомом минимальном цикле — обозначим текущую вершину-кандидата через s. Найдем алгоритмом Дейкстры длины кратчайших путей из s до всех остальных вершин — обозначим их через $dist[s][i]$, а массив «предков» через $parent[s][i]$.

Тогда утверждается, что искомый минимальный цикл имеет стоимость, равную:

$$\min_{\substack{s,\ i \neq j,\\ i \neq parent[s][j],\\ j \neq parent[s][i]}} \left(dist[s][i] + dist[s][j] + g[i][j]\right),$$

где $g[i][j]$ — вес ребра между вершинами i и j (или бесконечность, если такого ребра нет). Иными словами, для поиска решения мы помимо стартовой вершины **перебираем ребро, принадлежащее циклу**, и конструируем цикл как кратчайшие пути из s до обоих концов этого ребра.

На первый взгляд кажется, что такое решение не удовлетворяет требованию поиска **простого** цикла, ведь при поиске кратчайших путей алгоритмом Дейкстры нет никакой гарантии, что пути в разные вершины не пересекаются друг с другом.

Однако рассмотрим такой случай, когда для каких-то s, i и j кратчайшие пути $s \to i$ и $s \to j$ пересекаются друг с другом; обозначим через w последнюю общую вершину этих путей. Если мы рассмотрим фрагменты этих путей $w \to i$ и $w \to j$, то они уже не будут иметь пересечений, а суммарная стоимость этих фрагментов будет строго меньше (благодаря тому что по условию в графе нет ребер нулевого веса). Следовательно, при условии что ребро *(i, j)* не принадлежит ни одному из этих путей, цикл $w \to i \to j \to w$ будет простым, и он будет обнаружен в момент, когда w будет являться стартовой вершиной.

Следовательно, в решении нам важно убедиться, что мы игнорируем такие случаи, когда ребро *(i, j)* принадлежит кратчайшим путям $s \to i$ или $s \to j$. Поскольку мы рассматриваем только кратчайшие пути и в графе нет ребер нулевого веса, ребро *(i, j)* могло быть лишь на конце, но не в середине путей $s \to i$ и $s \to j$. Последнее ребро кратчайшего пути, найденного алгоритмом Дейкстры, хранится в массиве *parent* — следовательно, проверок $i \neq parent[s][j]$ и $j \neq parent[s][i]$ достаточно.

Последнее утверждение, которое нам осталось доказать, — это то, что со всеми этими дополнительными критериями искомый цикл действительно будет найден.

Это в самом деле так, поскольку на минимальном цикле можно найти «среднее» относительно стартовой вершины s ребро — такое ребро, которое делит цикл на примерно равные по стоимости половины. Поскольку, опять же, веса ребер в графе положительны, кратчайшие пути из s до обоих концов этого ребра будут непересекающимися — следовательно, при переборе всевозможных пар (i, j) это «среднее» ребро будет найдено.

Это завершает доказательство корректности описанного выше решения.

Асимптотика решения составит $O(n^3)$: мы перебираем n кандидатов на стартовую вершину, и для каждого кандидата мы запускаем алгоритм Дейкстры (для которого имеет смысл использовать его простую квадратичную реализацию) и затем перебираем всевозможные ребра.

31.1.2. Картельный сговор

В терминах теории графов для ответа на каждый запрос (x_i, y_i) нам требуется проверить, существует ли как путь из x_i в y_i, так и путь из y_i в x_i. Это в точности совпадает с определением компоненты сильной связности.

Следовательно, для решения задачи требуется заранее найти все компоненты сильной связности и для ответа на запрос проверять, лежат ли указанные вершины в одной компоненте или нет.

Итоговая асимптотика решения — $O(n + m + k)$.

31.1.3. Превращение графа в сильно связный

Если входной граф уже является сильно связным, то ответ равен нулю.

Построим конденсацию входного графа, поскольку легко понять, что добавлять несколько входящих или исходящих ребер в одну компоненту сильной связности смысла нет, и нет разницы, какая вершина в компоненте сильной связности является началом/концом добавленного ребра.

Заметим, что для каждой вершины графа-конденсации, имеющей нулевую степень входа, должно быть добавлено входящее в нее ребро — иначе попасть в эту вершину из других вершин было бы невозможно. Аналогично для каждой вершины, имеющей нулевую степень выхода, должно быть добавлено исходящее из нее ребро.

Таким образом, ответ для не сильно связного графа равен как минимум:

$$\max(S, T),$$

где S — число вершин графа-конденсации с нулевой степенью входа, T — число вершин с нулевой степенью выхода.

На самом деле, эта нижняя граница — оптимальна, т. е. ответ в точности равен этой величине. Доказательство этого факта непросто и выходит за рамки этой главы книги.

Конструктивное доказательство

Для доказательства этого факта можно построить алгоритм, ищущий искомые ребра. Этот алгоритм нетривиален, однако в общих чертах он основан на поиске наибольшего паросочетания между вершинами с нулевой степенью входа и вершинами с нулевой степенью выхода и на построении искомых ребер согласно определенной конструкции.

31.1.4. Отражение точки относительно прямой

Отраженную точку можно получить, например, сначала найдя проекцию Q точки P на прямую, а затем дважды отложив вектор $\overrightarrow{PQ}$ от точки P.

Алгоритм поиска точки проекции обсуждался нами в решении задачи «Пересечение окружности и прямой» *(см. разд. 29.16)*.

31.1.5. Пересечение окружностей

Задачу пересечения двух окружностей можно свести к задаче пересечения окружности и прямой. Это можно сделать из геометрических соображений. В качестве другой альтернативы это можно сделать, записав два уравнения окружностей и вычтя одно из другого: после упрощения останется лишь линейное уравнение, которое можно интерпретировать как уравнение прямой.

Таким образом, мы свели задачу к уже известной: алгоритм пересечения окружности и прямой был разобран в тексте главы.

Останется только разобрать особый случай совпадающих окружностей, в котором ответ равен бесконечности.

31.1.6. Определение направления обхода многоугольника

Если бы многоугольник был выпуклым, не имел повторяющихся точек и лежащих на одной прямой сторон, то для нахождения ответа было бы достаточно проверить направление поворота любой тройки последовательных вершин. Однако, по условию, входной многоугольник может быть произвольным.

Самый простой способ решения задачи для произвольного многоугольника без самопересечений — это проверка знака его знаковой площади. Напомним, что знаковая площадь основана на суммировании площадей трапеций, образованных сторонами и их проекциями на координатную ось, и площади трапеций берутся с разными знаками в зависимости от направления движения. В зависимости от заданного направления обхода вершин площадь получится либо отрицательной, либо положительной.

Это решение будет иметь асимптотику $O(n)$.

31.1.7. Проверка многоугольника на выпуклость

Неверно решать эту задачу одними лишь проверками того, что все последовательные тройки точек имеют одно и то же направление поворота. Хотя этот критерий отсечет большинство невыпуклых многоугольников, в случае наличия самопересечений он может привести к неправильному ответу: многоугольник может иметь вид нескольких «витков», совершенных в одну и ту же сторону.

Проверки на самопересечения реализовывать непросто, особенно при заданных ограничениях, и, кроме того, с ними тоже могут возникнуть каверзные случаи, когда все самопересечения происходят только в вершинах многоугольника.

Поэтому вместо проверки на самопересечения можно просуммировать все углы поворота последовательных троек (например, с помощью функции atan2). Для многоугольника без самопересечений должно получиться значение 2π. При этом не возникает проблем с точностью, поскольку сумма углов поворота всегда кратна 2π, и потому есть большой запас для разделения значений по сравнению с возникающими погрешностями. Единственная тонкость — это то, что последовательные точки-дубликаты должны игнорироваться, иначе вместо истинных углов поворота мы получим в таких случаях нулевые углы.

Предметный указатель