Министерство науки и высшего образования Российской Федерации
Тольяттинский государственный университет
Институт математики, физики и информационных технологий

А.В. Очеповский, О.М. Гущина

РЕШЕНИЕ ЗАДАЧ ОЛИМПИАДНОГО ПРОГРАММИРОВАНИЯ

Электронное учебно-методическое пособие

© ФГБОУ ВО «Тольяттинский государственный университет», 2021

ISBN 978-5-8259-1585-2

УДК 004.9
ББК 3811

Рецензенты:
канд. техн. наук, доцент, доцент кафедры «Управление качеством и инновационные технологии» Поволжского государственного университета сервиса *Д.И. Панюков*;
д-р. физ.-мат. наук, профессор кафедры «Прикладная математика и информатика» Тольяттинского государственного университета *А.И. Сафронов*.

Очеповский, А.В. Решение задач олимпиадного программирования : электронное учебно-методическое пособие / А.В. Очеповский, О.М. Гущина. – Тольятти : Изд-во ТГУ, 2021. – 1 оптический диск. – ISBN 978-5-8259-1585-2.

Учебно-методическое пособие содержит информацию о большинстве стандартных тем и примеров реализации алгоритмов, которые помогают участникам олимпиады по программированию: структура данных, динамическое программирование, графовые алгоритмы и алгоритмы на деревьях, запросы по диапазону, работа со строками.

Предназначено для тех, кто хочет освоить алгоритмы, навыки писать код без багов, думать о крайних случаях, о производительности.

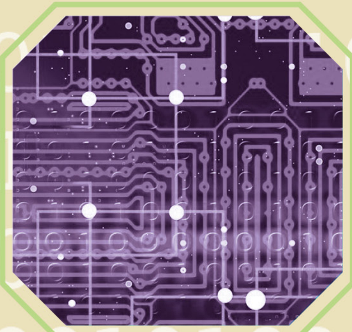
Пособие может быть полезно студентам, профессорско-преподавательскому составу высших учебных заведений, а также любому желающему получить знания в области алгоритмов в качестве практического руководства при подготовке к олимпиадному программированию.

Текстовое электронное издание.

Рекомендовано к изданию научно-методическим советом Тольяттинского государственного университета.

Минимальные системные требования: IBM PC-совместимый компьютер: Windows XP/Vista/7/8; PIII 500 МГц или эквивалент; 128 Мб ОЗУ; SVGA; CD-ROM; Adobe Acrobat Reader.

© ФГБОУ ВО «Тольяттинский государственный университет», 2021



Редактор *Т.М. Воропанова*
Технический редактор *Н.П. Крюкова*
Компьютерная верстка: *Л.В. Сызганцева*
Художественное оформление,
компьютерное проектирование: *Г.В. Карасева*

Дата подписания к использованию 13.10.2021.
Объем издания 5 Мб.
Комплектация издания: компакт-диск, первичная упаковка.
Заказ № 1-36-20.

Издательство Тольяттинского государственного университета
445020, г. Тольятти, ул. Белорусская, 14,
тел. 8 (8482) 53-91-47, www.tltsu.ru

Содержание

ВВЕДЕНИЕ 6

1. БАЗОВЫЕ АЛГОРИТМЫ 7
   1.1. Алгоритм бинарного возведения в степень 7
   1.2. Алгоритм «Решето Эратосфена» 9
   1.3. Различные оптимизации решета Эратосфена 10
   1.4. Числа Фибоначчи 11
   1.5. Факториал по модулю p за $O(p \log(n))$ 16
   1.6. Линейные диофантовы уравнения с двумя переменными 22

2. АЛГОРИТМЫ НА СТРОКАХ 28
   2.1. Префикс-функция 28
   2.2. Алгоритмы хэширования в задачах на строки 30
   2.3. Задачи на строках 32

3. ГЕОМЕТРИЯ 34
   3.1. Линейные операции 34
   3.2. Пересечение линии 42
   3.3. Пересечение плоскостей 43

4. АЛГОРИТМЫ НА ГРАФАХ 46
   4.1. Поиск в ширину (BFS) 46
   4.2. Примеры решения задач с использованием алгоритма DFS 49
   4.3. 0-1 Поиск в ширину (0-1 BFS) 52
   4.4. Примеры решения задач с использованием алгоритма BFS 54
   4.5. Поиск в глубину (DFS) 59
   4.6. Классификация рёбер графа 60
   4.7. Примеры решения задач с использованием алгоритма DFS 62
   4.8. Алгоритм Дейкстры 67
   4.9. Примеры решения задач с использованием алгоритма Дейкстры 72
   4.10. Алгоритм поиска компонент связности в графе 76
   4.11. Алгоритм топологической сортировки вершин графа 77

5. ВСПОМОГАТЕЛЬНЫЕ СТРУКТУРЫ ДАННЫХ 83
5.1. Дерево отрезков 83
5.2. Задача, использующая в решении дерево отрезков 85
5.3. Декартово дерево 87

6. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ 90
6.1. «Разделяй и властвуй» 90
6.2. Динамика по профилю. Задача «Паркет» 92
6.3. Нахождение самой большой нулевой подматрицы 97
6.4. Игры на произвольных графах 101

7. ПОТОКИ 108
7.1. Алгоритм Диница для нахождения
максимального потока 108
7.2. Поиск блокирующего потока 110
7.3. Единичные сети 110
7.4. Масштабирование потока 111
7.5. Реализация алгоритма Диница 111
7.6. Пример решения задачи 113

8. ПРОДВИНУТЫЕ АЛГОРИТМЫ 115
8.1. Быстрое преобразование Фурье 115
8.2. Дискретное преобразование Фурье 115
8.3. Применение ДПФ: быстрое умножение многочленов 116
8.4. Быстрое преобразование Фурье 117
8.5. Обратное БПФ 118
8.6. Теоретико-числовое преобразование 121
8.7. Вычисление многочленов с большими
коэффициентами с помощью нескольких NTT 124
8.8. Примеры решения задач 124

БИБЛИОГРАФИЧЕСКИЙ СПИСОК 128

ВВЕДЕНИЕ

Спортивное программирование является замечательным способом совершенствования навыков мышления. Конкурентная среда, в которую попадет студент, дает мотивацию для изучения различных разделов математики и совершенствования своих навыков программирования. Спортивное программирование показывает, насколько могут отличаться подходы к решению задачи и насколько сильно это влияет на конечный результат. Может казаться, что в современной разработке уже забыли о том, сколько вычислительных ресурсов требует программа, но по-прежнему присутствует требование писать эффективный код.

В этом методическом пособии перечислено несколько основных типов задач, которые встречаются на соревнованиях по олимпиадному программированию. Пособие поделено на большие разделы, каждый из которых содержит несколько основных алгоритмов с примерами решения типовых задач. Все примеры кода приведены на языке программирования C++. Данное пособие рекомендуется для студентов, которые уже имеют базовые навыки программирования и знают элементарные алгоритмы и структуры данных.

1. БАЗОВЫЕ АЛГОРИТМЫ

1.1. Алгоритм бинарного возведения в степень

Бинарное (двоичное) возведение в степень – это приём, позволяющий возводить любое число в n-ю степень за $O(\log n)$ умножений (вместо n умножений при обычном подходе).

Более того, описываемый здесь приём применим к любой ассоциативной операции, а не только к умножению чисел. Напомним, операция называется ассоциативной, если для любых a, b, c выполняется

$$(a \cdot b) \cdot c = a \cdot (b \cdot c).$$

Наиболее очевидное обобщение – на остатки по некоторому модулю (очевидно, ассоциативность сохраняется). Следующим по «популярности» является обобщение на произведение матриц (его ассоциативность общеизвестна).

Заметим, что для любого числа a и чётного числа n выполнимо очевидное тождество (следующее из ассоциативности операции умножения):

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}.$$

Оно и является основным в методе бинарного возведения в степень. Действительно, для чётного n мы показали, как, потратив всего одну операцию умножения, можно свести задачу к вдвое меньшей степени.

Осталось понять, что делать, если степень n нечётна. Здесь мы поступаем очень просто: перейдём к степени $n - 1$, которая будет уже чётной:

$$a^n = a^{n-1} \cdot a.$$

Итак, мы фактически нашли рекуррентную формулу: от степени n мы переходим, если она чётна, к $n/2$, а иначе – к $n - 1$. Понятно, что всего будет не более $2 \log n$ переходов, прежде чем мы придём к $n = 0$ (базе рекуррентной формулы). Таким образом, мы получили алгоритм, работающий за $O(\log(n))$ умножений.

Листинг 1. Простейшая рекурсивная реализация алгоритма бинарного возведения в степень

```
int binpow (int a, int n) {
    if (n == 0)
          return 1;
    if (n % 2 == 1)
          return binpow (a, n-1) * a;
    else {
          int b = binpow (a, n/2);
          return b * b;
    }
}
```

Листинг 2. Нерекурсивная реализация алгоритма бинарного возведения в степень

```
int binpow (int a, int n) {
    int res = 1;
    while (n)
          if (n & 1) {
                res *= a;
                --n;
          }
          else {
                a *= a;
                n >>= 1;
          }
    return res;
}
```

Эту реализацию можно ещё несколько упростить, заметив, что возведение a в квадрат осуществляется всегда, независимо от того, сработало условие нечётности n или нет:

Листинг 3. Оптимизированная нерекурсивная реализация алгоритма бинарного возведения в степень

```
int binpow (int a, int n) {
    int res = 1;
    while (n) {
          if (n & 1)
                res *= a;
          a *= a;
```

```
            n >>= 1;
        }
        return res;
    }
```

Наконец, стоит отметить, что бинарное возведение в степень уже реализовано в языке Java, но только для класса длинной арифметики BigInteger (функция pow этого класса работает именно по алгоритму бинарного возведения).

1.2. Алгоритм «Решето Эратосфена»

Решето Эратосфена – это алгоритм, позволяющий найти все простые числа в отрезке $[1; n]$ за $O(n \log(\log(n)))$ операций.

Идея проста: запишем ряд чисел 1 ... n, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем делящиеся на 3, кроме самого числа 3, затем на 5, затем на 7, 11 и все остальные простые до n.

Листинг. Реализация алгоритма «Решето Эратосфена»

```
int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
    if (prime[i])
          if (i * 1ll * i <= n)
                for (int j=i*i; j<=n; j+=i)
                      prime[j] = false;
```

Этот код сначала помечает все числа, кроме нуля и единицы, как простые, а затем начинает процесс отсеивания составных чисел. Для этого мы перебираем в цикле все числа от 2 до n, и если текущее число i простое, то помечаем все числа, кратные ему, как составные.

При этом мы начинаем идти от i^2, поскольку все меньшие числа, кратные i, обязательно имеют простой делитель меньше i, а значит, все они уже были отсеяны раньше. (Но поскольку i^2 легко может переполнить тип int, в коде перед вторым вложенным циклом делается дополнительная проверка с использованием типа long long.)

При такой реализации алгоритм потребляет $O(n)$ памяти (что очевидно) и выполняет $O(n \log(\log(n)))$ действий (это доказывается в следующем разделе).

1.3. Различные оптимизации решета Эратосфена

Самый большой недостаток алгоритма — то, что он «гуляет» по памяти, постоянно выходя за пределы кэш-памяти, из-за чего константа, скрытая в $O(n \log(\log(n)))$, сравнительно велика.

Кроме того, для достаточно больших n узким местом становится объём потребляемой памяти.

Ниже рассмотрены методы, позволяющие как уменьшить число выполняемых операций, так и значительно сократить потребление памяти.

1.3.1. Просеивание простыми числами до корня

Самый очевидный момент — для того, чтобы найти все простые до n, достаточно выполнить просеивание только простыми, не превосходящими корня из n.

Таким образом, изменится внешний цикл алгоритма:

```
for (int i=2; i*i<=n; ++i)
```

На асимптотику такая оптимизация не влияет (действительно, повторив приведённое выше доказательство, мы получим $n \ln \ln \sqrt{n} + o(n)$, что, по свойствам логарифма, асимптотически есть то же самое), хотя число операций заметно уменьшится.

1.3.2. Решето только по нечётным числам

Поскольку все чётные числа, кроме 2, — составные, то можно вообще не обрабатывать никак чётные числа, а оперировать только нечётными числами.

Во-первых, это позволит вдвое сократить объём требуемой памяти. Во-вторых, это уменьшит число делаемых алгоритмом операций примерно вдвое.

1.3.3. Уменьшение объёма потребляемой памяти

Заметим, что алгоритм решета Эратосфена фактически оперирует с n битами памяти. Следовательно, можно существенно сэкономить потребление памяти, храня не n байт — переменных булевского типа, а n бит, т. е. $n/8$ байт памяти.

Однако такой подход — «битовое сжатие» — существенно усложнит оперирование этими битами. Любое чтение или запись бита будут представлять собой несколько арифметических операций, что в итоге приведёт к замедлению алгоритма.

Таким образом, этот подход оправдан, только если n настолько большое, что n байт памяти выделить уже нельзя. Сэкономив память (в 8 раз), мы заплатим за это существенным замедлением алгоритма.

В завершение стоит отметить, что в языке C++ уже реализованы контейнеры, автоматически осуществляющие битовое сжатие: vector<bool> и bitset<>. Впрочем, если скорость работы очень важна, то лучше реализовать битовое сжатие вручную, с помощью битовых операций — на сегодняшний день компиляторы всё же не в состоянии генерировать достаточно быстрый код.

1.4. Числа Фибоначчи

Последовательность Фибоначчи определяется следующим образом:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

Первые элементы последовательности (OEIS A000045):

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Числа Фибоначчи обладают множеством интересных свойств. Вот несколько из них.

- Соотношение Кассини

$$F_{n-1} F_{n+1} - F_n^2 = (-1)^n.$$

- Правило «сложения»:

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$$

- Из предыдущего равенства, при $k = n$, мы получим:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

• Отсюда можно по индукции доказать, что для любого натурального числа K F_{nk} кратно F_n .

• Обратное также верно: если F_m кратно F_n, тогда m кратно n.

• НОД-равенство:

$$CCD(F_m, F_n) = F_{CCD(m,\, n)}.$$

• Числа Фибоначчи являются наихудшими входными данными для евклидова алгоритма.

1.4.1. Кодирование Фибоначчи

Мы можем использовать последовательность для кодирования натуральных чисел в двоичные кодовые слова. Согласно теореме Цекендорфа, любое натуральное число n может быть однозначно представлено в виде суммы чисел Фибоначчи

$$N = F_{k1} + F_{k2} + \ldots + F_{kr}$$

такое, что $k_1 \geq k_2 + 2$, $k_2 \geq k_3 + 2$, ..., $k_т \geq 2$ (то есть представление не может использовать два последовательных числа Фибоначчи).

Отсюда следует, что любое число может быть уникально закодировано в кодировании Фибоначчи. И мы можем описать это представление с помощью двоичных кодов $d_0 d_1 d_2 \ldots d_s 1$, где d_i является 1, если F_{i+2} + 2 используется в представлении. Код будет добавляться через единицу, означая конец кодового слова. Обратите внимание, что это единственный случай, когда появляются два последовательных 1-битных символа (рис. 1).

$$\begin{aligned}
1 &= 1 && = F_2 && = (11)_F \\
2 &= 2 && = F_3 && = (011)_F \\
6 &= 5 + 1 && = F_5 + F_2 && = (10011)_F \\
8 &= 8 && = F_6 && = (000011)_F \\
9 &= 8 + 1 && = F_6 + F_2 && = (100011)_F \\
19 &= 13 + 5 + 1 && = F_7 + F_5 + F_2 && = (1001011)_F
\end{aligned}$$

Рис. 1. Пример кодирования числами Фибоначчи

Кодировку целого числа N можно сделать с помощью простого жадного алгоритма:

1. Перебирайте числа Фибоначчи от самого большого до самого маленького, пока не найдете одно меньше или равное N.
2. Предположим, это число было F_i, тогда вычитать F_i от N и передавать единицу в i вторую позицию кодового слова (индексация от 0 до крайнего левого до правого бита).
3. Повторяйте, пока не появится остаток.
4. Добавить последнюю 1 к кодовому слову, чтобы указать его конец.

Чтобы декодировать кодовое слово, сначала удалите последнюю 1, тогда, если i-й бит задан (индексирование от 0 до крайнего левого до крайнего правого бита), суммируйте F_{i+2} с его номером.

1.4.2. Формулы для *n*-го числа Фибоначчи

N-е число Фибоначчи можно легко найти за $O(n)$, вычисляя числа последовательно, начиная с 1 до N. Однако, как мы видим, есть и более быстрые способы.

Существует формула, известная как формула Бине, хотя она уже была известна Моивре:

$$F_n = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}.$$

Эту формулу легко доказать по индукции, но ее можно вывести с помощью понятия образующих функций или решения функционального уравнения.

Вы можете сразу заметить, что значение второго слагаемого всегда меньше 1, а также уменьшается оно очень быстро (экспоненциально). Следовательно, значение первого слагаемого даёт «почти» значение F_n. Это можно записать в строгом виде:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right],$$

где квадратные скобки обозначают округление до ближайшего целого числа. Поскольку эти две формулы требуют очень высо-

кой точности при работе с дробными числами, они мало полезны в практических вычислениях.

1.4.3. Матричная форма

Нетрудно доказать следующее соотношение:

$$(F_{n-1}\ F_n) = (F_{n-2}\ F_{n-1}) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Обозначив $P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, мы имеем:

$$F_n\ F_{n+1} = (F_0\ F_1) \cdot P^n.$$

Таким образом, чтобы найти F_n, мы должны возвести матрицу P в степень N. Это можно сделать за $O(\log n)$ (возведением в степень), получается, что n-е число Фибоначчи можно легко вычислить за $O(\log(n))$ с использованием только целочисленной арифметики.

1.4.4. Метод быстрого дублирования

Используя вышеуказанный метод, мы можем получить эти уравнения:

$$F_{2k} = F_k(2F_{k+1} - F_k).$$

$$F_{2k+1} = F_{k+1}^2 + F_k^2.$$

Таким образом, используя два приведенных выше уравнения, числа Фибоначчи можно легко вычислить с помощью следующего кода:

Листинг. Реализация метода быстрого дублирования

```
pair<int, int> fib (int n) {
    if (n == 0)
        return {0, 1};
    auto p = fib(n >> 1);
    int c = p.first * (2 * p.second - p.first);
    int d = p.first * p.first + p.second * p.second;
    if (n & 1)
        return {d, c + d};
    else
        return {c, d};
}
```

Код выше возвращает F_n, а также F_{n+1} как пару.

1.4.5. Периодичность по модулю p

Рассмотрим последовательность Фибоначчи по модулю p. Мы докажем, что последовательность является периодической, и период начинается с $F_1 = 1$ (то есть предварительный период содержит только F_0).

Давайте докажем это от противного. Рассмотрим $P^2 + 1$ пары чисел Фибоначчи, взятые по модулю p:

$$(F_1, F_2), (F_2, F_3), ..., (F_{p2+1}, F_{p2+2}).$$

По модулю p может быть только p^2 различных пар, поэтому среди них есть как минимум две одинаковые пары. Таким образом, последовательность является периодической.

Теперь мы выберем две пары одинаковых остатков с наименьшими индексами в последовательности. Пусть пары будут (F_a, F_{a+1}), а также (F_b, F_{b+1}). Мы докажем, что $a = 1$. Если предположить, что это не так, то две предыдущие пары (F_{a-1}, F_a), а также (F_{b-1}, F_b) по свойству чисел Фибоначчи также будут равны. Однако это противоречит тому, что мы выбрали пары с наименьшими индексами, что и требовалось доказать.

Напишите функцию *int fib*(*int n*), которая возвращает F_n. Например, если $n = 0$, то *fib*() должна возвращать 0. Если $n = 1$, то она должна возвращать 1. Для $n > 1$ она должна возвращать .

Листинг. Использование матрицы

```
#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);
void power(int F[2][2], int n);

int fib(int n)
{
 int F[2][2] = {{1,1},{1,0}};
 if (n == 0)
 return 0;
 power(F, n-1);

 return F[0][0];
}
```

```
void multiply(int F[2][2], int M[2][2])
{
 int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
 int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
 int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
 int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

 F[0][0] = x;
 F[0][1] = y;
 F[1][0] = z;
 F[1][1] = w;
}

void power(int F[2][2], int n)
{
 int i;
 int M[2][2] = {{1,1},{1,0}};

 for (i = 2; i <= n; i++)
 multiply(F, M);
}

int main()
{
 int n = 9;
 printf("%d", fib(n));
 getchar();
 return 0;
}
```

1.5. Факториал по модулю *p* за *O*(*p* log(*n*))

В некоторых случаях необходимо учитывать сложные формулы по модулю p, содержащие факториалы в числителе и знаменателе. Рассмотрим случай, когда p относительно небольшой. Эта проблема имеет смысл только тогда, когда факториалы включены в числитель и знаменатель дробей. В противном случае $p!$ и последующие слагаемые будут уменьшаться до нуля, но в дробях все множители, содержащие p, могут быть уменьшены, и полученное выражение будет ненулевым по модулю p.

Таким образом, формально задача такова: вы хотите рассчитать $n!\ mod\ p$ без учета всех многочисленных множителей p, которые появляются в факториале. Представим, что вы записали первичную факторизацию $n!$, удалили все делители p и вычислили произведение по модулю p. Обозначим этот модифицированный факториал через $n!_{\%p}$.

Научившись эффективно вычислять такой модифицированный факториал, мы будем быстро вычислять значение различных комбинаторных формул (например, биномиальных коэффициентов).

Алгоритм

Давайте напишем этот модифицированный факториал.

$$\begin{aligned} n!_{\%p} &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1) \cdot \underbrace{2}_{2p} \\ &\quad \cdot (2p+1) \cdot \ldots \cdot (p^2-1) \cdot \underbrace{1}_{p^2} \cdot (p^2+1) \cdot \ldots \cdot n \pmod{p} \\ &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot 2 \cdot \ldots \cdot (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2 \\ &\quad \cdot \ldots \cdot (p-1) \cdot \underbrace{1}_{p^2} \cdot 1 \cdot 2 \cdot \ldots \cdot (n \bmod p) \pmod{p} \end{aligned}$$

Хорошо видно, что факториал делится на несколько блоков одинаковой длины, кроме последнего.

$$\begin{aligned} n!_{\%p} = & \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{1\text{st}} \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 2}_{2\text{nd}} \cdot \ldots \\ & \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{p\text{th}} \cdot \ldots \cdot \underbrace{1 \cdot 2 \cdot \cdot \ldots \cdot (n \bmod p)}_{\text{tail}} \pmod{p}. \end{aligned}$$

Общую часть блоков легко посчитать — это просто $(p-1)!\ mod\ p$, что вы можете рассчитать программой или с помощью теоремы Вильсона, в соответствии с которой $(p-1)!\ mod\ p = p-1$. Чтобы умножить эти общие части всех блоков, мы можем поднять значение до более высокой мощности по модулю p, что можно сделать за $O(\log(n))$ операции с использованием бинарного возведения в степень. Тем не менее вы можете заметить, что результат всегда будет либо 1, либо $p-1$, в зависимости от чётности показателя. Значение последнего частичного блока можно рассчитать отдельно за $O(p)$. Оставляя только последние элементы блоков, мы можем рассмотреть их внимательнее:

$$n!_{\%p} = \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 2} \cdot \ldots \cdot \underbrace{\ldots \cdot (p-1)} \cdot \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 2} \cdots$$

И снова, удаляя блоки, которые мы уже вычислили, мы получаем «модифицированный» факториал, но с меньшей размерностью (блоков было $[n/p]$). Таким образом, при расчете «модифицированного» факториала $n!_{\%p}$ мы сделали $O(p)$ операций к вычислению уже $(n/p)!_{\%p}$. Выявив эту рекурсивную зависимость, получаем, что глубина рекурсии $O(\log_p n)$. Таким образом, общее асимптотическое поведение алгоритма $O(p \log_p n)$.

Нам не нужна рекурсия, потому что это случай хвостовой рекурсии, и поэтому ее можно легко реализовать с помощью итерации.

Листинг 1. Нерекурсивный поиск факториала по заданному модулю

```
int factmod(int n, int p) {
    int res = 1;
    while (n > 1) {
        res = (res * ((n/p) % 2 ? p-1 : 1)) % p;
        for (int i = 2; i <= n%p; ++i)
            res = (res * i) % p;
        n /= p;
 }
 return res % p;
}
```

Эта реализация работает за $O(p \log_p n)$. Учитывая большое число n и простое число p, как эффективно вычислить $n!_{\%p}$?

Несколько примеров проверки корректности работы алгоритма:

1. Ввод: $n = 5$, $p = 13$

Выход: 3

5! = 120 и 120 % 13 = 3

2. Ввод: $n = 6$, $p = 11$

Выход: 5

6! = 720 и 720 % 11 = 5

Листинг 2. Метод 1 (простой)

```
#include <bits/stdc++.h>
using namespace std;

int modFact(int n, int p)
{
    if (n >= p)
        return 0;

    int result = 1;
    for (int i = 1; i <= n; i++)
        result = (result * i) % p;

    return result;
}

int main()
{
    int n = 25, p = 29;
    cout << modFact(n, p);
    return 0;
}
```

Листинг 3. Метод 2 (решето)

```
#include <bits/stdc++.h>
using namespace std;

int largestPower(int n, int p)
{
    int x = 0;

    while (n) {
    n /= p;
    x += n;
  }
  return x;
}

 int power(int x, int y, int p)
 {
  int res = 1; // Initialize result
```

```
  x = x % p; // Update x if it is more than or
  while (y > 0) {
              if (y & 1)
        res = (res * x) % p;

    y = y >> 1; // y = y/2
    x = (x * x) % p;
  }
  return res;
}

int modFact(int n, int p)
{
   if (n >= p)
       return 0;

   int res = 1;

   bool isPrime[n + 1];
   memset(isPrime, 1, sizeof(isPrime));
   for (int i = 2; i * i <= n; i++) {
       if (isPrime[i]) {
           for (int j = 2 * i; j <= n; j += i)
               isPrime[j] = 0;
    }
 }

 for (int i = 2; i <= n; i++) {
     if (isPrime[i]) {
        int k = largestPower(n, i);

                   res = (res * power(i, k, p)) % p;
    }
 }
    return res;
}

int main()
{
    int n = 25, p = 29;
    cout << modFact(n, p);
    return 0;
}
```

Листинг 4. Метод 3 (с использованием теоремы Вильсона)

```
#include <bits/stdc++.h>
using namespace std;

int power(int x, unsigned int y, int p)
{
    int res = 1;
    x = x % p;
    while (y > 0) {
        if (y & 1)
            res = (res * x) % p;

        y = y >> 1; // y = y/2
        x = (x * x) % p;
   }
   return res;
}

int modInverse(int a, int p)
{
    return power(a, p – 2, p);
}

int modFact(int n, int p)
{
    if (p <= n)
        return 0;

    int res = (p – 1);

    for (int i = n + 1; i < p; i++)
        res = (res * modInverse(i, p)) % p;
    return res;
}

int main()
{
    int n = 25, p = 29;
    cout << modFact(n, p);
    return 0;
}
```

1.6. Линейные диофантовы уравнения с двумя переменными

Линейное диофантово уравнение (в двух переменных) – это уравнение общего вида:

$$ax + by = c,$$

где a, b, c – заданные целые числа; x, y – неизвестные целые числа.

Мы рассмотрим несколько классических задач по этим уравнениям:

- поиск одного решения;
- поиск всех решений;
- поиск количества решений и самих решений в заданном интервале.

Вырожденный случай, о котором нужно знать, – это когда $a = b = 0$. Легко увидеть, что у нас либо нет решений, либо бесконечно много решений, в зависимости от того, является ли $c = 0$ или нет. Мы будем игнорировать этот случай.

Чтобы найти одно решение диофантова уравнения с двумя неизвестными, можно использовать расширенный алгоритм Евклида. Во-первых, предположим, что a и b неотрицательны. Когда мы применяем расширенный евклидов алгоритм для a и b, мы можем найти их наибольший общий делитель q и два числа x_g и y_g такие, что:

$$ax_g + by_g = g.$$

Если c делится на $g = gcd(a, b)$, то данное диофантово уравнение имеет решение, иначе уравнение решений не имеет. Доказательство простое: линейная комбинация двух чисел делится на их общий делитель.

Теперь предположим, что c делится на g, тогда мы имеем:

$$a \cdot x_g \cdot \frac{c}{g} + b \cdot y_g \cdot \frac{c}{g} = c.$$

Поэтому одним из решений диофантова уравнения является:

$$x_0 = x_g \cdot \frac{c}{g};$$

$$y_0 = y_g \cdot \frac{c}{g}.$$

Вышеприведенная идея все еще работает, когда a или b или оба они отрицательны. Нам нужно только изменить знак x_0 и y_0, когда это необходимо.

Мы можем реализовать эту идею следующим образом (этот код не рассматривает случай $a = b = 0$).

Листинг. Реализация решения диофантова уравнения

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
          x = 1;
          y = 0;
          return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 – y1 * (a / b);
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0,
int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
          return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

Из одного решения (x_0, y_0) можно получить все решения данного уравнения.

Пусть $g = gcd(a, b)$ и пусть x_0, y_0 – целые числа, удовлетворяющие условию:

$$a \cdot x_0 + b \cdot y_0 = c.$$

Теперь мы видим, что прибавление $\frac{b}{g}$ к x_0 и в то же время вычитание $\frac{a}{g}$ из y_0 не нарушит равенства

$$a \cdot \left(x_0 + \frac{b}{g}\right) + b \cdot \left(y_0 - \frac{a}{g}\right) = a \cdot x_0 + b \cdot y_0 + a \cdot \frac{b}{g} - b \cdot \frac{a}{g} = c.$$

Очевидно, что этот процесс можно повторить, поэтому все числа вида

$$x = x_0 + k \cdot \frac{b}{g};$$

$$y = y_0 - k \cdot \frac{a}{g}$$

являются решениями данного диофантова уравнения.

Более того, это множество всех возможных решений данного диофантова уравнения.

1.6.1. Нахождение количества решений и самих решений в заданном отрезке

Из предыдущего раздела должно быть ясно, что если мы не будем накладывать никаких ограничений на решения, то их будет бесконечное количество. Поэтому мы добавим некоторые ограничения на интервал x и y, а также попытаемся подсчитать и перечислить все решения.

Пусть существует два интервала: [min x; max x] и [min y; max y], и предположим, что мы хотим найти решения только в этих двух интервалах.

Обратите внимание, что если a или b равно 0, то задача имеет только одно решение. Мы не рассматриваем здесь этот случай.

Сначала мы можем найти решение, которое имеет минимальное значение x, такое, что $x \geq \min x$. Для этого мы сначала находим любое решение диофантова уравнения. Затем мы сдвигаем это решение, чтобы получить $x \geq \min x$ (используя то, что мы знаем о множестве всех решений из предыдущего раздела). Это можно сделать в $O(1)$. Обозначим это минимальное значение x через l_{x1}.

Аналогично мы можем найти максимальное значение x, которое удовлетворяет $x \leq \max x$. Обозначим это максимальное значение x через r_{x1}.

По тому же принципу мы можем найти минимальное значение y ($y \geq \min y$) и максимальное значение y ($y \leq \max y$). Обозначим соответствующие значения x по l_{x2} и r_{x2}.

Конечное решение – это все решения с x в пересечении $[l_{x1}, r_{x1}]$ и $[l_{x2}, r_{x2}]$. Обозначим это пересечение через $[l_x, r_x]$.

Ниже приведен код, реализующий эту идею. Обратите внимание, что мы делим a и b вначале на g. Поскольку уравнение $ax + by = c$ эквивалентно уравнению $\frac{a}{g} \cdot x + \frac{b}{g} \cdot y = \frac{c}{g}$, мы можем использовать это вместо него и получить $gcd\left(\frac{a}{g}, \frac{b}{g}\right) = 1$, что упрощает формулы.

Листинг 1. Реализация решения в заданном отрезке

```
void shift_solution(int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}

int find_all_solutions(int a, int b, int c, int minx, int maxx, int miny, int maxy) {
    int x, y, g;
    if (!find_any_solution(a, b, c, x, y, g))
        return 0;
    a /= g;
    b /= g;

    int sign_a = a > 0 ? +1 : -1;
    int sign_b = b > 0 ? +1 : -1;

    shift_solution(x, y, a, b, (minx - x) / b);
    if (x < minx)
        shift_solution(x, y, a, b, sign_b);
    if (x > maxx)
        return 0;
    int lx1 = x;

    shift_solution(x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution(x, y, a, b, -sign_b);
    int rx1 = x;
```

```
    shift_solution(x, y, a, b, -(miny - y) / a);
    if (y < miny)
        shift_solution(x, y, a, b, -sign_a);
    if (y > maxy)
        return 0;
    int lx2 = x;

    shift_solution(x, y, a, b, -(maxy - y) / a);
    if (y > maxy)
        shift_solution(x, y, a, b, sign_a);
    int rx2 = x;

    if (lx2 > rx2)
        swap(lx2, rx2);
    int lx = max(lx1, lx2);
    int rx = min(rx1, rx2);

    if (lx > rx)
        return 0;
    return (rx - lx) / abs(b) + 1;
}
```

Как только у нас есть l_x и r_x, мы можем просто перечислить все решения. Просто нужно перебрать $x = lx + k \cdot \frac{b}{g}$ для всех $k \geq 0$ до $x = r_x$ и найти соответствующие значения y, используя уравнение $ax + by = c$.

Пример с сайта codeforcesы.com: http://codeforces.com/contest/633/problem/A

Разбор

Задача состоит в том, чтобы найти, существует ли решение уравнения $ax + by = c$, где x и y – оба положительные целые числа. Пределы достаточно малы, чтобы перебрать все значения x и соответственно перебрать, существует ли такой y. Этот вопрос также может быть решен более эффективно, используя тот факт, что интегральное решение этой проблемы существует, если $gcd(a, b)|c$. Нам просто нужно сделать еще одну проверку, чтобы убедиться в правильности интегрального решения.

Сложность: $(\log(\min(a, b))$.

Листинг 2. Решение задачи

```
main()
{
    int a, b, c;
    cin >> a >> b >> c;
    for (int i = 0; i <= c; i++)
           if (c – a * i >= 0 && (c – a * i) % b == 0)
{ cout << “Yes” << endl; return 0; }
    cout << «No» << endl;
}
```

2. АЛГОРИТМЫ НА СТРОКАХ

2.1. Префикс-функция

Задана строка s длины n. Префикс-функция для этой строки определяется как массив π длины n, где $\pi[i]$ – длина самого длинного собственного префикса подстроки $s[0...i]$, который также является суффиксом этой подстроки. Собственный префикс строки – это префикс, который не равен самой строке. По определению $\pi[0] = 0$.

Рассмотрим тривиальный алгоритм построения префикс-функции строки (рис. 2).

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 0; i < n; i++)
        for (int k = 0; k <= i; k++)
            if (s.substr(0, k) == s.substr(i-k+1, k))
                pi[i] = k;
    return pi;
}
```

Рис. 2. Функция вычисления префикс-функции строки

Легко видеть, что данный алгоритм работает за $O(n$^3). Рассмотрим некоторые оптимизации, чтобы улучшить временную сложность алгоритма.

Первое важное наблюдение состоит в том, что значения префиксной функции могут увеличиваться не более чем на один.

Если $\pi[i + 1] > \pi[i] + 1$, то мы можем взять этот суффикс, заканчивающийся в позиции $i + 1$ длиной $\pi[i + 1]$, и удалить из него последний символ. В итоге получаем суффикс, заканчивающийся в позиции i длиной $\pi[i + 1] - 1$, что больше, чем $\pi[i]$, т. е. получаем противоречие.

Таким образом, при переходе в следующую позицию значение префиксной функции может увеличиваться на единицу, оставаться

неизменным или уменьшаться на некоторую величину. Этот факт уже позволяет нам свести сложность алгоритма к $O(n^2)$, потому что за один шаг префиксная функция может расти не более чем на один. В общем случае функция может расти не более n шагов и, следовательно, также может уменьшиться только на n. Это означает, что нам нужно только выполнить сравнения строк за $O(n)$ и достичь сложности $O(n^2)$.

Пойдем дальше: мы хотим избавиться от сравнения строк. Для этого мы должны использовать всю информацию, вычисленную на предыдущих шагах.

Итак, давайте вычислим значение префиксной функции π для $i + 1$. Если $s[i + 1] = s[\pi[i]]$, то можно с уверенностью сказать, что $\pi[i + 1] = \pi[i] + 1$, так как мы уже знаем, что суффикс в позиции i длины $\pi[i]$ равен префиксу длины $\pi[i]$.

Если это не так, $s[i + 1] \neq s[\pi[i]]$, тогда нам нужно попробовать более короткую строку. Чтобы ускорить процесс, мы хотели бы сразу перейти к самой длинной длине $j < \pi[i]$, чтобы свойство префикса в позиции i выполнялось, т. е. $s[0...j - 1] = s[i - j + 1...i]$.

Действительно, если мы найдем такую длину j, то нам снова нужно будет сравнить только символы $s[i + 1]$ и $s[j]$. Если они равны, то $\pi[i + 1] = j + 1$. В противном случае нам нужно будет найти наибольшее значение, меньшее j, для которого выполнено свойство префикса, и так далее. Может случиться так, что это будет продолжаться до $j = 0$. Если тогда $s[i + 1] = s[0]$, то $\pi[i + 1] = 1$, иначе $\pi[i + 1] = 0$.

Итак, у нас уже есть общая схема алгоритма. Остается только вопрос, как эффективно найти длины для j. Для текущей длины j в позиции i, для которой выполнено свойство префикса, т. е. $s[0...j - 1] = s[i - j + 1...i]$, мы хотим найти наибольшее $k < j$, для которого выполнено свойство префикса. Это будет значение $\pi[j - 1]$, которое мы уже рассчитали ранее.

Таким образом, мы, наконец, можем построить алгоритм, который не выполняет никаких строковых сравнений и выполняет только $O(n)$ операций.

2.2. Алгоритмы хэширования в задачах на строки

Алгоритмы хэширования полезны при решении множества задач. Задача, которую мы хотим решить, – это сравнение двух строк. Наивным способом мы можем запустить цикл и сравнивать две строки «побуквенно», таким образом сложность алгоритма $O(\min(n1, n2))$, если $n1$ и $n2$ – размеры двух строк. Мы хотим сделать лучше. Идея состоит в следующем: мы преобразуем каждую строку в целое число и будем сравнивать их вместо строк. Таким образом, получим алгоритм сравнения двух строк за $O(1)$.

Для преобразования нам нужна так называемая хэш-функция. Ее целью является преобразование строки в целое число, так называемые хэш-строки. Следующее условие должно выполняться: если две строки s и t равны, то и их хэши должны быть равны ($hash(s) = hash(t)$). В противном случае мы не можем сравнивать строки.

Обратим внимание, что противоположное условие не обязательно должно соблюдаться. Если хэши равны ($hash(s) = hash(t)$), то строки необязательно должны быть равны. Причина, по которой противоположное условие не требуется, в том, что существует экспоненциальное множество строк. Если мы хотим, чтобы хэш-функция различала все строки, состоящие из символов нижнего регистра длиной меньше 15, то уже хэш не вписывался бы в 64-битное целое число. И, конечно, мы не будем сравнивать произвольные длинные целые числа, потому что это также будет иметь сложность $O(n)$.

Поэтому обычно стремимся, чтобы хэш-функция отображала строки на числа фиксированного диапазона $[0, m)$, тогда сравнение строк – это просто сравнение двух целых чисел с фиксированной длиной. И мы считаем: чтобы $hash(s) \neq hash(t)$ был очень вероятным, то $s \neq t$.

Это важная часть, которую нужно иметь в виду. Использование хэширования не будет на 100 % детерминистически правильным, потому что две разные строки могут иметь один и тот же хэш. Однако в широком большинстве задач это можно смело игнорировать, так как вероятность совпадения хэшей двух разных строк все еще очень мала.

Хорошим и широко используемым способом определения хэша строки является полиномиальное хэширование (рис. 3).

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

Рис. 3. Полиномиальная хэш-функция

Разумно сделать p простым числом, примерно равным числу символов во входном алфавите. Например, если ввод состоит только из строчных букв английского алфавита, то $p = 31$ является хорошим выбором.

Очевидно, что m должно быть большим числом, так как вероятность столкновения двух случайных строк составляет около $1/p$. Иногда выбирается m = 2^64, так как тогда целочисленные переполнения 64-битных целых чисел работают точно так же, как и операция взятия по модулю. Однако существует метод, который генерирует сталкивающиеся строки (которые работают независимо от выбора p). Поэтому на практике m = 2^64 не рекомендуется. Хорошим выбором для m является некоторое большое простое число, например, m = 10^9 + 9.

Приведем пример вычисления хэш-функции строки s на языке C++ (рис. 4).

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Рис. 4. Вычисление хэш-функции строки

Заметим, что если предподсчитать степени p, можно увеличить производительность.

2.3. Задачи на строках

Подсчёт числа вхождений одной строки в другую

Рассмотрим следующую задачу. Даны строки s и t. Требуется посчитать количество вхождений строки t в строку s, а также вывести позиции этих вхождений. Тривиальным алгоритмом можно решить задачу за время $O(|s| \cdot |t|)$, перебирая каждый символ строки s и сравнивая его с первым символом строки t; в случае если они равны, то проверять последующие символы. Однако такой алгоритм решения задачи неэффективен. Решим задачу за время $O(|s| + |t|)$. Используя алгоритм Кнута – Морриса – Пратта, построим префикс-функцию для строки $r = t + c + s$, где c – любой символ, не встречающийся в обеих строках. Если префикс-функция примет значение, равное длине строки t, то это будет означать, что в данном месте строка t входит в строку s. Рассмотрим код программы, реализующий данный алгоритм (рис. 5).

```
5   vector<int> prefix_function(string s){
6       int n = s.length();
7       vector<int> result(n, 0);
8       for(int i = 1; i < n; i++){
9           int j = result[i - 1];
10          while(j > 0 && s[i] != s[j]) j = result[j - 1];
11          if(s[j] == s[i]) j++;
12          result[i] = j;
13      }
14      return result;
15  }
16
17  int main(){
18      string s, t;
19      cin >> s >> t;
20      string r = t + "#" + s;
21      vector<int> answer;
22      vector<int> vec = prefix_function(r);
23      for(int i = 0; i < vec.size(); i++){
24          if(vec[i] == t.length()){
25              answer.push_back(i - 2 * t.length() + 1);
26          }
27      }
28      cout << answer.size() << endl;
29      for(auto u: answer) cout << u << ' ';
30      return 0;
31  }
```

Рис. 5. Программный код решения задачи поиска числа вхождений одной строки в другую

На вход программе подается две строки s и t. Строится новая строка $r = t$ + "#" + s. Затем вычисляется префикс-функция от этой строки. Далее проверяются значения префикс-функции, и если оно равно длине строки t, то вычисляем позицию начала вхождения и добавляем ее к ответу. В конце выводим количество вхождений и их позиции.

3. ГЕОМЕТРИЯ

В этом пункте мы рассмотрим основные операции над точками в евклидовом пространстве, которые поддерживают основу всей аналитической геометрии. Мы рассмотрим для каждой точки r вектор $\vec{r}$, направленный от 0 в r. Позже мы не будем различать r и $\vec{r}$, и использовать термин «точка» в качестве синонима вектора.

3.1. Линейные операции

Как 2D, так и 3D-точки поддерживают линейное пространство; это означает, что для них определены сумма точек и умножение точки на некоторое число.

Листинг 1. Основные двухмерные операции

```
struct point2d {
    ftype x, y;
    point2d() {}
    point2d(ftype x, ftype y): x(x), y(y) {}
    point2d& operator+=(const point2d &t) {
        x += t.x;
        y += t.y;
        return *this;
    }
    point2d& operator-=(const point2d &t) {
        x -= t.x;
        y -= t.y;
        return *this;
    }
    point2d& operator*=(ftype t) {
        x *= t;
        y *= t;
       return *this;
    }
    point2d& operator/=(ftype t) {
       x /= t;
       y /= t;
       return *this;
    }
    point2d operator+(const point2d &t) const {
       return point2d(*this) += t;
```

```
    }
    point2d operator-(const point2d &t) const {
      return point2d(*this) -= t;
    }
    point2d operator*(ftype t) const {
       return point2d(*this) *= t;
    }
    point2d operator/(ftype t) const {
       return point2d(*this) /= t;
    }
};
point2d operator*(ftype a, point2d b) {
       return b * a;
}
```

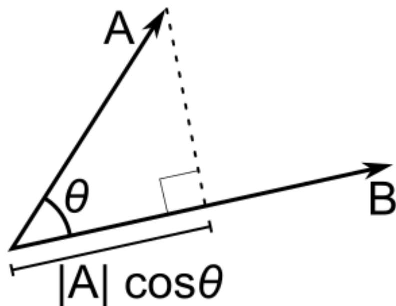
Листинг 2. Основные трехмерные операции

```
struct point3d {
    ftype x, y, z;
    point3d() {}
    point3d(ftype x, ftype y, ftype z): x(x), y(y), z(z)
{}
    point3d& operator+=(const point3d &t) {
       x += t.x;
       y += t.y;
       z += t.z;
       return *this;
    }
    point3d& operator-=(const point3d &t) {
       x -= t.x;
       y -= t.y;
       z -= t.z;
       return *this;
    }
    point3d& operator*=(ftype t) {
      x *= t;
      y *= t;
      z *= t;
      return *this;
    }
    point3d& operator/=(ftype t) {
      x /= t;
      y /= t;
```

```
      z /= t;
      return *this;
   }
   point3d operator+(const point3d &t) const {
   return point3d(*this) += t;
   }
   point3d operator-(const point3d &t) const {
   return point3d(*this) -= t;
   }
   point3d operator*(ftype t) const {
   return point3d(*this) *= t;
   }
   point3d operator/(ftype t) const {
   return point3d(*this) /= t;
   }
};
point3d operator*(ftype a, point3d b) {
    return b * a;
}
```

Это ftype – некоторый тип, используемый для координат, обычно int, double или long long.

Точечное (или скалярное) произведение $a \cdot b$ для векторов a, b могут быть определены двумя одинаковыми способами. Геометрически это произведение длины первого вектора на длину проекции второго вектора на первый. Как вы можете видеть из изображения ниже, эта проекция не что иное, как $|a|\cos 0$, где 0 – это угол между a и b, таким образом $a \cdot b = |\mathrm{a}|\cos 0 \cdot |b|$.



Векторное произведение обладает некоторыми заметными свойствами:

1. $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
2. $(\alpha \cdot \mathbf{a}) \cdot \mathbf{b} = \alpha \cdot (\mathbf{a} \cdot \mathbf{b})$
3. $(\mathbf{a} + \mathbf{b}) \cdot \mathbf{c} = \mathbf{a} \cdot \mathbf{c} + \mathbf{b} \cdot \mathbf{c}$

То есть это коммутативная функция, которая линейна относительно обоих аргументов. Обозначим единичные векторы как

$$\mathbf{e}_x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \mathbf{e}_y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{e}_z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

С помощью этой записи мы можем написать: $r = (x; y; z)$ как $r = x \cdot e_x + y \cdot e_y + z \cdot e_z$. И поскольку для единичных векторов

$$\mathbf{e}_x \cdot \mathbf{e}_x = \mathbf{e}_y \cdot \mathbf{e}_y = \mathbf{e}_z \cdot \mathbf{e}_z = 1,$$
$$\mathbf{e}_x \cdot \mathbf{e}_y = \mathbf{e}_y \cdot \mathbf{e}_z = \mathbf{e}_z \cdot \mathbf{e}_x = 0$$

мы можем заметить, что с точки зрения координат для $a = (x_1; y_1; z_1)$ и $b = (x_2; y_2; z_2)$ содержит

$$\mathbf{a} \cdot \mathbf{b} = (x_1 \cdot \mathbf{e}_x + y_1 \cdot \mathbf{e}_y + z_1 \cdot \mathbf{e}_z) \cdot (x_2 \cdot \mathbf{e}_x + y_2 \cdot \mathbf{e}_y + z_2 \cdot \mathbf{e}_z) =$$
$$= x_1 x_2 + y_1 y_2 + z_1 z_2$$

Это также алгебраическое определение скалярного произведения. Из этого мы можем написать функции, которые могут его вычислить.

Листинг 3. Скалярное произведение

```
ftype dot(point2d a, point2d b) {
    return a.x * b.x + a.y * b.y;
}
ftype dot(point3d a, point3d b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

При решении задач следует использовать алгебраическое определение для вычисления точечных произведений, но имейте в виду геометрические определения и свойства для его использования.

Мы можем определить многие геометрические свойства через скалярное произведение. Например:

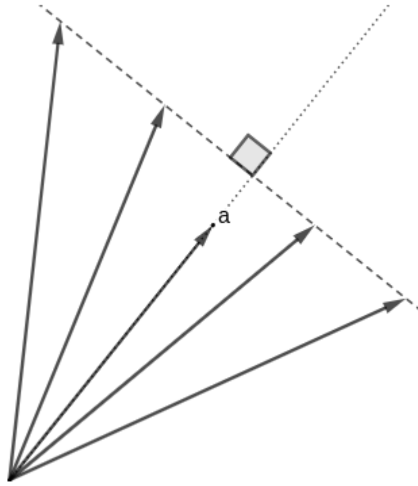
1. Норма a: (квадрат длины): $|a|^2 = a \cdot a$.
2. Длина a: $|a| = \sqrt{a \cdot a}$.
3. Проекция a на b: $\frac{a \cdot b}{|a| \cdot |b|}$.
4. Угол между векторами: $\arccos\left(\frac{a \cdot b}{|a| \cdot |b|}\right)$.
5. Из предыдущего пункта мы можем видеть, что скалярное произведение положительно, если угол между ними острый, и отрицательно, если он тупой, и равен нулю, если они ортогональны, то есть они образуют прямой угол.

Обратите внимание, что все эти функции не зависят от количества измерений, поэтому они будут одинаковыми для случая 2D и 3D.

Листинг 4. Пример вычисления некоторых функций

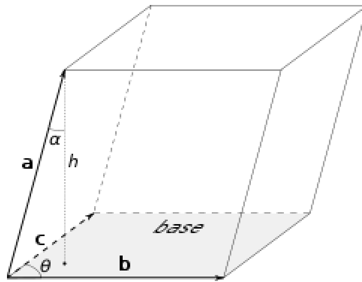
```
ftype norm(point2d a) {
    return dot(a, a);
}
double abs(point2d a) {
    return sqrt(norm(a));
}
double proj(point2d a, point2d b) {
    return dot(a, b) / abs(b);
}
double angle(point2d a, point2d b) {
    return acos(dot(a, b) / abs(a) / abs(b));
}
```

Чтобы заметить следующее важное свойство, мы должны взглянуть на множество точек r, для которых $r \cdot a = C$, при некоторой фиксированной константе C. Вы можете видеть, что этот набор точек является определённым набором точек, для которых проекция на a – это точка $C \cdot \frac{a}{|a|}$ и они образуют гиперплоскость, ортогональную a. Обратите внимание на вектор a наряду с несколькими такими векторами, имеющими одинаковое точечное произведение с ним в 2D на рисунке ниже.



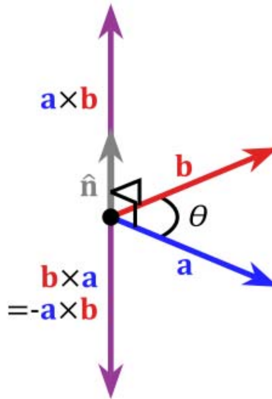
В 2D эти векторы образуют линию, в 3D они образуют плоскость. Заметим, что этот результат позволяет нам определить линию в 2D как $r \cdot n = C$ или $(r - r_0) \cdot n = 0$, где n — вектор, ортогональный к прямой, и r_0 — любой вектор, уже существующий на прямой, и $C = r_0 \cdot n$. Таким же образом плоскость может быть определена в 3D.

Предположим, у вас есть три вектора — a, b и c в трехмерном пространстве, соединенном в параллелепипед, как на картинке ниже:



Как бы вы рассчитали его объем? Из школьного курса известно, что мы должны умножить площадь основания на высоту, которая является проекцией a в направлении, ортогональном к основанию. Это означает, что если мы определим $b \cdot c$ как вектор, который ортогонален обоим b и c, а также длину, равную площади параллелограмма, образованного b и c, тогда $|a \cdot (b \cdot c)|$ будет равен объему паралле-

лепипеда. Для честности скажем, что $b \cdot c$ всегда будет направлено таким образом, чтобы вращение от вектора b к вектору c с точки зрения $b \cdot c$ всегда против часовой стрелки (см. рисунок ниже).



Это определяет перекрестное (или векторное) произведение $b \cdot c$ векторов b и c и тройное произведение $a \cdot (b \cdot c)$ векторов a, b и c.

Некоторые заметные свойства перекрестных и тройных произведений:

1. $a \cdot b = -b \cdot a$.

2. $(\alpha \cdot a) \cdot b = \alpha \cdot (a \cdot b)$.

3. Для любой b и c есть ровно один вектор r такой, что $a \cdot (b \cdot c) = a \cdot r$ для любого вектора a. Действительно, если есть два таких вектора r_1 и r_2, тогда $a \cdot (r_1 - r_2) = a$ для всех векторов, что возможно только тогда, когда $r_1 = r_2$.

4. $a \cdot (b \cdot c) = b \cdot (c \cdot a) = -a \cdot (c \cdot b)$.

5. $(a + b) \cdot c = a \cdot c + b \cdot c$. Действительно, для всех векторов r цепочка уравнений содержит:

$$r \cdot \big((a+b) \cdot c\big) = (a+b) \cdot (c \cdot r) = a \cdot (c \cdot r) + b \cdot (c \cdot r)$$

$$= r \cdot (a \cdot c) + r \cdot (b \cdot c) = r \cdot (a \cdot c + b \cdot c).$$

Что доказывает $(a + b) \cdot c = a \cdot c + b$ ввиду пункта 3.

6. $|a \cdot b| = |a| \cdot |b| \sin 0$, где 0 – угол между a и b, поскольку $|a \cdot b|$ равна площади параллелограмма, образованного a и b.

Учитывая все это, следующая запись справедлива для единичных векторов:

$$\mathbf{e}_x \times \mathbf{e}_x = \mathbf{e}_y \times \mathbf{e}_y = \mathbf{e}_z \times \mathbf{e}_z = \mathbf{0},$$
$$\mathbf{e}_x \times \mathbf{e}_y = \mathbf{e}_z,\ \mathbf{e}_y \times \mathbf{e}_z = \mathbf{e}_x,\ \mathbf{e}_z \times \mathbf{e}_x = \mathbf{e}_y$$

Мы можем рассчитать векторное произведение $a = (x_1;\ y_1;\ z_1)$ и $b = (x_2; y_2; z_2)$ в координатной форме:

$$\mathbf{a} \times \mathbf{b} = (x_1 \cdot \mathbf{e}_x + y_1 \cdot \mathbf{e}_y + z_1 \cdot \mathbf{e}_z) \times (x_2 \cdot \mathbf{e}_x + y_2 \cdot \mathbf{e}_y + z_2 \cdot \mathbf{e}_z) =$$
$$= (y_1 z_2 - z_1 y_2)\mathbf{e}_x + (z_1 x_2 - x_1 z_2)\mathbf{e}_y + (x_1 y_2 - y_1 x_2)$$

которое также можно записать в более элегантной форме:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{e}_x & \mathbf{e}_y & \mathbf{e}_z \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix},\ a \cdot (b \times c) = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

Здесь $|\cdot|$ обозначает определитель матрицы.

Некоторый вид скалярного произведения (а именно псевдоскалярное произведение) также может быть реализован в 2D-случае. Если бы мы хотели рассчитать площадь параллелограмма, образованного векторами a и b, мы бы вычислили $|e_z^*(a \cdot b)| = |x_1 y_2 - y_1 x_2|$. Другой способ получить тот же результат — это умножить $|a|$ (основание параллелограмма) с высотой, которая является проекцией вектора b на вектор a, повернутый на 90°, что в свою очередь $\hat{a} = (-y_1; x_1)$. То есть рассчитать $|\hat{a} \cdot b| = |x_1 y_2 - y_1 x_2|$.

Если мы примем знак во внимание, то площадь будет положительной, если поворот от a в b (то есть с точки зрения e_z) выполняется против часовой стрелки, и отрицательной — в противном случае. Это определяет псевдоскалярное произведение. Обратите внимание, что оно также равно $|a| \cdot |b| \sin 0$, где 0 — это угол от a в b, если считать против часовой стрелки, и отрицательно, если вращение по часовой стрелке.

Листинг 5. Псевдоскалярное произведение

```
point3d cross(point3d a, point3d b) {
    return point3d(a.y * b.z - a.z * b.y,
                   a.z * b.x - a.x * b.z,
                   a.x * b.y - a.y * b.x);
}
ftype triple(point3d a, point3d b, point3d c) {
    return dot(a, cross(b, c));
}
```

```
ftype cross(point2d a, point2d b) {
    return a.x * b.y – a.y * b.x;
}
```

Что касается скалярного произведения, оно равно нулевому вектору, если векторы a и b являются коллинеарными (они образуют общую линию, т. е. они параллельны). То же самое верно для тройного произведения: оно равно нулю, если векторы a, b и c являются компланарными (они образуют общую плоскость).

Отсюда можно получить универсальные уравнения, определяющие линии и плоскости. Прямая может быть определена через вектор направления d и начальной точкой r_0 или двумя точками a и b. Определяется как $(r - r_0) \cdot d = 0$ или как $(r - a) \cdot (b - a) = 0$. Что касается плоскостей, то их можно определить тремя точками a, b и c в виде $(r - a) \cdot ((b - a) \cdot (c - a)) = 0$ или по начальной точке r_0 и двум направляющим векторам, лежащим в этой плоскости d_1 и d_2: $(r - r_0) \cdot (d_1 \cdot d_2) = 0$.

В 2D псевдоскалярное произведение также может использоваться для проверки ориентации между двумя векторами, поскольку оно положительно, если вращение от первого ко второму вектору происходит по часовой стрелке, а в противном случае – отрицательно. И, конечно же, его можно использовать для вычисления площадей многоугольников. Тройное произведение может быть использовано для той же цели в трехмерном пространстве.

3.2. Пересечение линии

Есть много возможных способов определить линию в 2D, и вы не должны стесняться комбинировать их. Например, у нас есть две линии, и мы хотим найти их точки пересечения. Можно сказать, что все точки из первой линии могут быть параметризованы как $r = a_1 + t \cdot d_1$, где a_1 является начальной точкой, d_1 – это направление и t – это некоторый параметр. Что касается второй прямой, то все ее точки должны удовлетворять $(r - a_2) \cdot d_2 = 0$. Отсюда легко найти параметр t:

$$(a_1 + t \cdot d_1 - a_2) \cdot d_2 = 0 \quad \Rightarrow \quad t = \frac{(a_2 - a_1) \cdot d_2}{d_1 \cdot d_2}.$$

Листинг. Пересечение линий

```
point2d intersect(point2d a1, point2d d1, point2d a2,
point2d d2) {
      return a1 + cross(a2 – a1, d2) / cross(d1, d2) * d1;
}
```

3.3. Пересечение плоскостей

Однако иногда бывает сложно использовать некоторые геометрические идеи. Например, даны три плоскости, определенные начальными точками a_i и направлениями d_i, и вы хотите найти их точку пересечения. Вы можете заметить, что просто нужно решить систему уравнений:

$$\begin{cases} r \cdot n_1 = a_1 \cdot n_1, \\ r \cdot n_2 = a_2 \cdot n_2, \\ r \cdot n_3 = a_3 \cdot n_3. \end{cases}$$

Вместо того чтобы думать о геометрическом подходе, можно разработать алгебраический, который можно получить сразу. Например, учитывая, что вы уже реализовали точечный класс, будет легко решить эту систему, используя правило Крамера, потому что тройное произведение — это просто определитель матрицы, полученной из векторов, являющихся ее столбцами.

Листинг 1. Пересечение плоскостей

```
point3d intersect(point3d a1, point3d n1, point3d a2,
point3d n2, point3d a3, point3d n3) {
      point3d x(n1.x, n2.x, n3.x);
      point3d y(n1.y, n2.y, n3.y);
      point3d z(n1.z, n2.z, n3.z);
     point3d d(dot(a1, n1), dot(a2, n2), dot(a3, n3));
     return point3d(triple(d, y, z),
                    triple(x, d, z),
                    triple(x, y, d)) / triple(n1, n2, n3);
}
```

Проверка работоспособности алгоритма поиска пересечения двух отрезков. Для заданных точек A и B, соответствующих прямой AB, и точек P и Q, соответствующих прямой PQ, найдите точку

пересечения этих линий. Точки приведены в 2D плоскости с их координатами *X* и *Y*.

1. Вход: $A = (1, 1)$, $B = (4, 4)$, $C = (1, 8)$, $D = (2, 4)$

Выход: пересечение данных линий

AB и *CD*: (2,4, 2,4)

2. Вход: $A = (0, 1)$, $B = (0, 4)$, $C = (1, 8)$, $D = (1, 4)$

Выход: заданные линии *AB* и *CD* параллельны.

Листинг 2. Листинг программы

```
#include <bits/stdc++.h>
using namespace std;

#define pdd pair<double, double>

void displayPoint(pdd P)
{
     cout << "(" << P.first << ", " << P.second
          << ")" << endl;
}

pdd lineLineIntersection(pdd A, pdd B, pdd C, pdd D)
{

    double a1 = B.second - A.second;
    double b1 = A.first - B.first;
    double c1 = a1*(A.first) + b1*(A.second);

       double a2 = D.second - C.second;
    double b2 = C.first - D.first;
    double c2 = a2*(C.first)+ b2*(C.second);

    double determinant = a1*b2 - a2*b1;

    if (determinant == 0)
    {
       return make_pair(FLT_MAX, FLT_MAX);
    }
    else
    {
       double x = (b2*c1 - b1*c2)/determinant;
       double y = (a1*c2 - a2*c1)/determinant;
       return make_pair(x, y);
```

```
    }
}

// Driver code
int main()
{
    pdd A = make_pair(1, 1);
    pdd B = make_pair(4, 4);
    pdd C = make_pair(1, 8);
    pdd D = make_pair(2, 4);

    pdd intersection = lineLineIntersection(A, B, C, D);

    if (intersection.first == FLT_MAX &&
        intersection.second==FLT_MAX)
    {
        cout << "The given lines AB and CD are
parallel.\n";
 }

 else
 {

    cout << "The intersection of the given lines AB "
            "and CD is: ";
    displayPoint(intersection);
   }

   return 0;
}
```

4. АЛГОРИТМЫ НА ГРАФАХ

4.1. Поиск в ширину (BFS)

Поиск в ширину – это один из базовых алгоритмов поиска в графе. Результат работы алгоритма – кратчайший путь до заданной вершины, т. е. путь, проходящий через минимальное количество рёбер в невзвешенном графе.

Алгоритм выполняется за $O(n + m)$, где n – количество вершин, а m – количество рёбер.

Алгоритм принимает на вход невзвешенный граф и идентификатор исходной вершины s. Является ли входной граф направленным или неориентированным, для алгоритма не имеет значения.

Алгоритм можно понимать как распространение огня по графу: на нулевой итерации горит только источник s. На каждом шаге из уже горящих вершин огонь распространяется на соседние с ними. За одну итерацию алгоритма «огненное кольцо» расширяется в ширину на одну единицу (отсюда и название алгоритма).

Более точно алгоритм может быть сформулирован так: создаётся очередь q, которая будет содержать обрабатываемые вершины и логический массив *used*[ ], указывающий для каждой вершины, была ли она зажжена (посещена) или нет.

Сначала нужно поместить в очередь исходную вершину s и задать для неё *used*[*s*] = *true*, а для всех остальных v вершин *used*[*v*] = *false*. Затем повторять цикл до тех пор, пока очередь не опустеет, на каждой итерации брать вершину из начала очереди. Перебирайте все рёбра, выходящие из этой вершины, и если некоторые из этих рёбер идут к вершинам, которые ещё не зажжены, поджигайте их, помещайте в очередь.

В итоге когда очередь пуста, «огненное кольцо» достигло все вершины, доступные из начальной вершины s, при том каждая вершина достигается кратчайшим способом. Вы так же можете вычислить длины кратчайших путей (для этого необходимо ввести массив «родителей» *p*[ ], который хранит для каждой вершины ту вершину, из которой мы её достигли).

Листинг 1. Алгоритм поиска в ширину

```
vector<vector<int>> adj; // граф представляется в
виде списка смежности
int n; // количество вершин
int s; // заданная вершина

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
           used[u] = true;
           q.push(u);
           d[u] = d[v] + 1;
           p[u] = v;
        }
    }
}
```

В случае если нам нужно восстановить и отобразить кратчайший путь от источника до некоторой вершины *u*, это можно сделать следующим образом.

Листинг 2. Восстановление кратчайшего пути

```
if (!used[u]) {
   cout << “No path!”;
}
else {
    vector<int> path;
    for (int v = u; v != -1; v = p[v])
        path.push_back(v);
    reverse(path.begin(), path.end());
    cout << “Path: “;
    for (int v : path)
       cout << v << « «;
}
```

Применение алгоритма поиска в ширину:

• Поиск кратчайшего пути от заданной вершины к другим вершинам невзвешенного графа.

• Поиск всех компонент связанности графа за $\boldsymbol{O}(n + m)$.

Для этого мы запускаем обход в ширину для каждой вершины, исключая те, которые уже были посещены во время предыдущих запусков. Таким образом, мы выполняем обычный обход в ширину, но каждый раз, когда мы получаем новую компоненту связанности, мы не обнуляем массив *used*[ ], и общее время выполнения всё равно будет $O(n + m)$ (выполнение нескольких поисков в ширину без обнуления массива называется серией обходов в ширину).

• Поиск решения задачи или игры с наименьшим числом ходов, если каждое состояние игры может быть представлено вершиной графа, а переходы из одного состояния в другое – его рёбрами.

• Поиск кратчайшего пути в графе с весами 0 или 1.

Это требует небольшой модификации обычного поиска по ширине: вместо записи массива *used*[ ] мы проверим, является ли расстояние до вершины короче текущего найденного расстояния, а затем, если текущее ребро имеет нулевой вес, мы добавим его в начало очереди или в конец – в противном случае.

• Поиск кратчайшего пути в направленном невзвешенном графе.

Для поиска самого короткого цикла, содержащего исходную вершину, начинаем обход по ширине с каждой вершины до перехода от текущей вершины к исходной. В этот момент мы можем остановить обход и начать новый из следующей вершины. Из всех таких циклов (не более одного от обхода) выбирают кратчайший.

• Поиск всех рёбер, лежащих на любом кратчайшем пути между вершинами (a, b).

Для этого сначала нужно выполнить два обхода в ширину: один из a и один из b. Пусть $d_a[\ ]$ – массив, содержащий кратчайшие расстояния, полученные при первом обходе (из a), а $d_b[\ ]$ – массив, содержащий кратчайшие расстояния, полученные при втором обходе (из b). Затем для каждого ребра (u, v) легко проверить, лежит ли ребро на любом кратчайшем пути между a и b: критерием является условие $d_a[u] + 1 + d_b[v] = d_a[b]$.

• Поиск всех вершин на любом кратчайшем пути между данной парой вершин (a, b).

Для этого сначала нужно выполнить два обхода в ширину: один из a и один из b. Пусть $d_a[\,]$ – массив, содержащий кратчайшие расстояния, полученные при первом обходе (из a), а $d_b[\,]$ – массив, содержащий кратчайшие расстояния, полученные при втором обходе (из b). Затем для каждой вершины легко проверить, лежит ли она на любом кратчайшем пути между a и b: критерием является условие $d_a[v] + 1 + d_b[v] = d_a[b]$.

• Поиск кратчайшего пути чётной длины от начальной вершины s до заданной вершины t в невзвешенном графе.

Для этого необходимо построить вспомогательный граф, вершинами которого являются (v, c), где v – текущий узел, $c = 0$ или $c = 1$ – текущая чётность. Любое ребро (a, b) исходного графа в этом новом графе превратится в два ребра ((u, 0), (v, 1)) и ((u, 1), (v, 0)). После этого мы запускаем обход в ширину, чтобы найти кратчайший путь от начальной вершины (s, 0) до конечной вершины (t, 0).

4.2. Примеры решения задач с использованием алгоритма DFS

Посты полиции

Инзейн наконец-то нашел Зейна, и у них еще много денег! Поэтому они решили создать собственную страну.

Правление страной – непростая задача. Бандиты и террористы постоянно пытаются нарушить покой. Чтобы бороться с этим, Зейн и Инзейн разработали следующее правило: из каждого города должна быть возможность достичь пост полиции, проехав не более d километров по дорогам.

В стране n городов, пронумерованных от 1 до n, соединенных n–1 дорогами. Все дороги имеют длину 1 километр. Возможно добраться от любого города до любого другого, используя эти дороги. Кроме того, есть k постов полиции, расположенных в некоторых городах. Расположение постов полиции удовлетворяет закону, описанному выше. Заметьте, что некоторые посты могут быть расположены в одном и том же городе.

Однако Зейн считает, что n–1 дорога – это слишком много. Страна испытывает финансовый кризис, поэтому необходимо закрыть как можно больше дорог.

Помогите Зейну определить максимальное число дорог, которое можно закрыть, не нарушая закон. Кроме того, найдите эти дороги.

Входные данные

Первая строка содержит три целых числа n, k и d ($2 \le n \le 3 \cdot 105,1 \le k \le 3 \cdot 105,0 \le d \le n–1$) – число городов, число постов полиции, ограничение на расстояние в законе соответственно.

Вторая строка содержит k целых чисел $p_1, p_2, ..., p_k$ ($1 \le p_i \le n$) – города, в которых расположены посты полиции.

i-я из следующих n–1 строк содержит два целых числа u_i и v_i ($1 \le u_i, v_i \le n, u_i \ne v_i$) – города, непосредственно соединенные дорогой i.

Гарантируется, что возможно добраться от любого города до любого другого, используя эти дороги. Кроме того, от любого города можно добраться до поста полиции, проехав не более d километров.

Выходные данные

В первой строке выведите одно целое число s, означающее максимальное число дорог, которое можно закрыть.

Во второй строке выведите s различных чисел – номера дорог, для которых это верно.

Если ответов несколько, выведите любой из них.

Разбор

Жадный алгоритм, закрывающий каждую из d-х дорог или встречающий полицейский участок, на деле оказывается неправильным.

Стоит рассмотреть возможность выполнения поиска в ширину с городами, в которых есть полицейский участок, в качестве начальных вершин, и закрытием дорог, если те ведут к посещённой вершине (городу).

Это приведёт к тому, что каждый «плохой» город будет связан (прямо или косвенно) с одним из ближайших полицейских участков и таким образом не будет нарушать закон.

С помощью этого метода вы можете видеть, что именно k'–1 дорог будут закрыты (где k' – количество городов, в которых есть полицейский участок). Предположим, что это не оптимально,

и могут быть закрыты $k' + c$ ($c \ge 0$) дорог. Тогда дерево будет разбито на $k' + c + 1$ компонентов, в то время как есть только k' городов с полицейским участком, так что это противоречие, так как будет по крайней мере один компонент без какого-либо полицейского участка. Следовательно, закрытие дорог $k'-1$ является оптимальным.

Листинг. Решение задачи

```
#include <stdio.h>
#include <queue>
#include <vector>
using namespace std;

queue<pair<int, int>> q;
vector<pair<int, int>> way[300005];
int v[300005];
int res[300005];

int main() {
    int n, k, d;
    scanf("%d%d%d", &n, &k, &d);
    for (int i = 0; i < k; i++) {
        int p;
        scanf("%d", &p);
        q.push({ p, 0 });
    }
    for (int i = 0; i < n - 1; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        way[u].push_back({ v, i + 1 });
        way[v].push_back({ u, i + 1 });
    }
    while (!q.empty()) {
        int pos = q.front().first;
        int from = q.front().second;
        q.pop();
        if (v[pos]) continue;
        v[pos] = 1;
        for (int i = 0; i < way[pos].size(); i++) if
(way[pos][i].first != from) {
            if (v[way[pos][i].first]) res[way[pos][i].
second] = 1;
```

```
            else q.push({ way[pos][i].first, pos });
        }
    }
    int rescnt = 0;
    for (int i = 1; i <= n - 1; i++) if (res[i])
rescnt++;
    printf("%d\n", rescnt);
    for (int i = 1; i <= n - 1; i++) if (res[i])
printf("%d ", i);
    return 0;
}
```

4.3. 0-1 Поиск в ширину (0-1 BFS)

Хорошо известно, что вы можете найти кратчайший путь между исходной вершиной и всеми другими в $O(|E|)$, используя обход в ширину в невзвешенном графе, т. е. расстояние – это минимальное количество рёбер, которое надо пересечь от исходной до другой вершины. Мы также можем интерпретировать такой граф как взвешенный, где каждое ребро имеет вес 1. Если не все ребра в графе имеют одинаковый вес, то нам нужен более общий алгоритм, например, алгоритм Дейкстры, который работает в $O(|V|^2 + |E|)$ или $O(|E|\log|V|)$.

Однако если весы более ограничены, мы часто можем сделать лучше. В этой статье показано, как можно использовать поиск в ширину для решения SSSP задачи (задачи о кратчайших путях с единственным источником) за $O(|E|)$, если вес каждого ребра равен либо 0, либо 1.

Мы можем разработать алгоритм, внимательно изучив алгоритм Дейкстры и подумав о следствиях, которые подразумевает наш специальный граф. Общая форма алгоритма Дейкстры такова (здесь используется для приоритетной очереди):

Листинг. 0-1 Поиск в ширину

```
d.assign(n, INF);
d[s] = 0;
set<pair<int, int>> q;
q.insert({ 0, s });
```

```
while (!q.empty()) {
    int v = q.begin()->second;
    q.erase(q.begin());

    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;

        if (d[v] + w < d[u]) {
            q.erase({ d[u], u });
            d[u] = d[v] + w;
            q.insert({ d[u], u });
        }
    }
}
```

Мы можем заметить, что разница расстояний между источником *s* и двумя другими вершинами в очереди отличается не более чем на единицу. В частности мы знаем, что $d[v] \le d[u] \le d[v] + 1$ для каждого $u \in Q$. Причина этого заключается в том, что мы добавляем вершины с равным расстоянием или с расстоянием плюс один в очередь каждой итерации. Если предположить, что существует *u* в очереди с $d[u] - d[v] > 1$, то *u* должен быть вставлен в очередь через другую вершину *t* с $d[t] \ge d[u] - 1 > d[v]$. Однако это невозможно, так как алгоритм Дейкстры перебирает вершины в возрастающем порядке.

Это означает, что порядок следования очереди выглядит следующим образом:

$$Q = \underbrace{v}_{d[v]}, \dots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]}, \dots, \underbrace{n}_{d[v]}.$$

Эта структура настолько проста, что нам фактически не нужна приоритетная очередь, т. е. сбалансированное двоичное дерево, – это перебор. Мы можем просто использовать обычную очередь и добавлять новые вершины в начале, если соответствующее ребро имеет вес 0, т. е. если $d[u] = d[v]$, или в конце, если ребро имеет вес 1, т. е. если $d[u] = d[v] + 1$. Таким образом, очередь все еще остается отсортированной в любое время.

```
    vector<int> d(n, INF);
    d[s] = 0;
    deque<int> q;
    q.push_front(s);
    while (!q.empty()) {
        int v = q.front();
        q.pop_front();
        for (auto edge : adj[v]) {
            int u = edge.first;
            int w = edge.second;
            if (d[v] + w < d[u]) {
                d[u] = d[v] + w;
                if (w == 1)
                    q.push_back(u);
                else
                    q.push_front(u);
            }
        }
    }
```

4.4. Примеры решения задач с использованием алгоритма BFS

Три государства

Опасаясь приближения мирового экономического кризиса, государства Бермания, Беранция и Берталия заключили союз и разрешили жителям всех входящих в него государств беспрепятственный проход по территории любого из них. Дополнительно было решено построить дороги между государствами так, чтобы из любой точки любого государства можно было добраться до любой точки любого другого государства.

Так как дороги всегда обходятся недёшево, правительства государств новообразованного союза попросили вас помочь им оценить затраты. Для этого вам была выдана карта, которая представляет собой клетчатый прямоугольник из n строк по m клеток в каждой. Любая клетка карты либо принадлежит одному из трёх государств, либо является областью, на которой можно построить дорогу, либо областью, на которой дорогу построить нельзя. Проходимыми явля-

ются все клетки, принадлежащие государствам, а также все клетки, в которых построены дороги. Из любой проходимой клетки можно перемещаться вверх, вниз, вправо и влево, если соответствующая такому перемещению клетка существует и является проходимой.

Требуется построить дороги в минимальном количестве клеток так, чтобы стало возможным добраться из любой клетки любого государства до любой клетки любого другого государства.

Гарантируется, что изначально из любой клетки любого государства можно добраться до любой клетки этого же государства, перемещаясь только по его клеткам. Также гарантируется, что на карте присутствует хотя бы одна клетка каждого государства.

Входные данные

В первой строке ввода записаны размеры карты n и m ($1 \leq n$, $m \leq 1000$) – количество строк и столбцов соответственно.

Следующие n строк содержат по m символов каждая и описывают строки карты. Цифры от 1 до 3 означают принадлежность соответствующему государству. Символ «.» соответствует клетке, на которой можно построить дорогу, а символ «#» соответствует клетке, на которой дорогу построить нельзя.

Выходные данные

Выведите единственное целое число – минимальное количество клеток, в которых необходимо построить дороги, чтобы соединить все клетки всех государств. Если это сделать невозможно, то выведите –1.

Разбор

Утверждение. Пусть в неориентированном невзвешенном связном графе выделены три различные вершины u, v, w. Одна из минимальных сетей, связывающих выделенные вершины, выглядит как некоторая вершина графа c, возможно совпадающая с одной из выделенных, из которой исходят кратчайшие пути к каждой из выделенных вершин, причём эти пути являются вершинно-непересекающимися.

Доказательство. Одним из оптимальных связывающих подграфов обязательно является дерево. Действительно, в противном случае на любом цикле найдётся ребро, которое можно выкинуть, и это не ухудшит ответ, поскольку он не зависит от количества используе-

мых рёбер. Листьями дерева могу являться только вершины u, v и w, иначе ответ можно было бы улучшить, просто выкинув такой лист. Дерево, у которого не более чем три листа, имеет не более одной вершины степени больше двух, которая и будет вершиной c из утверждения выше. Разумеется, любой путь от c до листа имеет смысл заменить на кратчайший. Отдельно возможен вырожденный случай, что дерево ответа — это бамбук, но в таком случае вершиной c является одна из трёх выделенных вершин (не лист).

Теперь имеем следующий метод для нахождения длины кратчайшей связывающей сети: перебрать все вершины, включая выделенные, и из сумм кратчайших расстояний от данной вершины до выделенных выбрать минимальную. Ясно, что таким образом мы переберём длины различных связывающих сетей, среди которых будет и длина кратчайшей, и поэтому минимум будет ответом.

Для сведения исходной задачи к задаче поиска минимальной связывающей сети можно представить карту в виде графа, где вершинам соответствуют клетки, принадлежащие государствам или допускающие постройку дороги, а ребро между двумя вершинами ставится, если они являются соседними в таблице. Все вершины, соответствующие одному государству, необходимо сжать в одну. Несложно заметить, что исходная задача таким образом свелась к вышеописанной.

Листинг. Решение задачи

```
#include <string>
#include <deque>
#include <cstdio>

#define mp make_pair
#define pb push_back

#ifdef LOCAL
#define eprintf(...) fprintf(stderr,__VA_ARGS__)
#else
#define eprintf(...)
#endif


#define TIMESTAMP(x) eprintf("["#x"] Time : %.3lf s.\n", clock()*1.0/CLOCKS_PER_SEC)
```

```
#define TIMESTAMPf(x,...) eprintf("[" x "] Time : %.3lf s.\n", __VA_ARGS__, clock()*1.0/CLOCKS_PER_SEC)

#if ( ( _WIN32 || __WIN32__ ) && __cplusplus < 201103L)
#define LLD "%I64d"
#else
#define LLD "%lld"
#endif

using namespace std;

#define TASKNAME "C"

#ifdef LOCAL
static struct __timestamper {
    string what;
    __timestamper(const char* what) : what(what) {};
    __timestamper(const string& what) : what(what) {};
    ~__timestamper() {
        TIMESTAMPf("%s", what.data());
    }
} __TIMESTAMPER("end");
#else
struct __timestamper {};
#endif

typedef long ll;
typedef long double ld;

const int MAXN = 1010;
char s[MAXN][MAXN];
int n, m;
int dist[3][MAXN][MAXN];

const int dx[4] = { 0, 1, 0, -1 };
const int dy[4] = { -1, 0, 1, 0 };

int main() {
#ifdef LOCAL
    assert(freopen(TASKNAME".in", "r", stdin));
    assert(freopen(TASKNAME".out", "w", stdout));
#endif

    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i++)
        scanf("%s", s[i]);
```

```
    memset(dist, -1, sizeof(dist));
    for (int c = '1'; c <= '3'; c++) {
        deque<pair<int, int>> q;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                if (s[i][j] == c) {
                    dist[c – '1'][i][j] = 0;
                    q.push_back(make_pair(i, j));
                }
            while (!q.empty()) {
                int x = q.front().first;
                int y = q.front().second;
                q.pop_front();
                for (int i = 0; i < 4; i++) {
                int nx = x + dx[i];
                int ny = y + dy[i];
                if (0 <= nx && nx < n && 0 <= ny && ny
< m && s[nx][ny] != '#') {
                    int nd = dist[c – '1'][x][y] + (s[nx]
[ny] == '.');
                    if (dist[c – '1'][nx][ny] == -1 ||
dist[c – '1'][nx][ny] > nd) {
                        dist[c – '1'][nx][ny] = nd;
                        if (s[nx][ny] == '.') {
                            q.push_back(make_pair(nx,
ny));
                        }
                        else {
                            q.push_front(make_pair(nx,
ny));
                        }
                    }
                }
                }
            }
        }
        int ans = -1;
        for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++) {
            if (dist[0][i][j] != -1 && dist[1][i][j]
!= -1 && dist[2][i][j] != -1) {
```

```
                    int nval = dist[0][i][j] + dist[1][i]
[j] + dist[2][i][j] – 2 * (s[i][j] == '.');
                    if (ans == -1 || ans > nval) {
                        ans = nval;
                    }
                }
            }
        printf("%d\n", ans);
        return 0;
   }
```

4.5. Поиск в глубину (DFS)

Поиск в глубину является одним из основных алгоритмов для графов. Поиск в глубину находит лексикографически первый путь в графе от исходной вершины *u* до каждой вершины. Поиск в глубину так же найдёт самые короткие пути в дереве (потому что существует только один простой путь), но на общих графах это не так.

Алгоритм работает за $O(m + n)$ времени, где n – число вершин, а m – число рёбер.

Описание алгоритма

Идея обхода в глубину заключается в том, чтобы проникнуть как можно глубже в граф и вернуться назад, как только вы окажетесь в вершине без каких-либо непосещённых соседних вершин.

Очень легко описать/реализовать алгоритм рекурсивно: начинаем поиск с одной вершины, после посещения вершины мы так выполняем обход в глубину для каждой соседней вершины, которую мы ещё не посещали. Таким образом мы посещаем все вершины, доступные из начальной.

Применение алгоритма

- Поиск любого пути в графе от изначальной вершины *u* до остальных вершин.
- Поиск лексикографически первого пути в графе от источника *u* до всех вершин.

• Проверка, является ли вершина дерева предком какой-либо другой вершины. В начале и конце каждого поискового вызова мы запоминаем входное и выходное «время» каждой вершины. Теперь мы можем найти ответ для любой пары вершин (I, j) в $O(1)$: вершина i является предком вершины j тогда и только тогда, когда

$$entry[i] < entry[j] \text{ и } exit[i] > exit[j].$$

• Поиск наименьшего общего предка двух вершин.

• Топологическая сортировка.

Сначала выполните серию обходов в глубину, чтобы посетить каждую вершину ровно один раз за $O(n + m)$. Требуемым топологическим упорядочением будут являться вершины, отсортированные в порядке убывания времени выхода.

• Проверка, является ли граф ациклическим, и поиск всех циклов в графе.

• Поиск компонент сильной связанности в направленном графе.

Сначала выполните топологическую сортировку графа. Затем транспонируйте граф и выполните еще одну серию обходов в глубину в порядке, определенном топологической сортировкой. Для каждого вызова обхода созданная им компонента связанности является сильно связанной компонентой.

• Поиск мостов в неориентированном графе.

Сначала преобразуйте имеющийся граф в ориентированный, выполнив серию обходов в глубину и сделав каждое ребро направленным в том векторе движения, в котором проходили через него. Затем найдите сильно связанные компоненты в этом направленном графе. Мосты – это рёбра, концы которых принадлежат различным компонентам сильной связанности.

4.6. Классификация рёбер графа

Мы можем классифицировать рёбра, используя время входа и выхода конечных вершин u и v рёбер (u, v). Эти классификации часто используются для решения таких задач, как поиск мостов и точек сочленения.

Мы выполняем обход в глубину и классифицируем встречающиеся рёбра, используя следующие правила:

Если v не посещена:

• Рёбра дерева – если v посещается после u, то ребро (u, v) называется ребром дерева. Другими словами, если v посещается в первый раз, а u в данный момент посещается, то (u, v) называется ребром дерева. Эти ребра образуют дерево обхода и, следовательно, называются ребрами дерева.

Если v посещена раньше u:

• Задние рёбра – если v является предком u, то ребро (u, v) является задним ребром. v является предком именно тогда, когда мы уже вошли в v, но еще не вышли из него. Задние рёбра завершают цикл, поскольку существует путь от предка v к потомку u (в рекурсии обхода в глубину) и ребро от потомка u к предку v (задний край), таким образом формируется цикл. С помощью задних рёбер можно обнаружить циклы.

• Передние ребра – если v является потомком u, то ребро (u, v) является передним ребром. Другими словами, если мы уже посетили и вышли из v и $entry[u] < entry[v]$, то ребро (u, v) образует переднее ребро.

• Перекрёстные ребра – если v не является ни предком, ни потомком u, то ребро (u, v) является перекрестным ребром. Другими словами, если мы уже посетили и вышли из v и $\text{entry}[u] > \text{entry}[v]$, то (u, v) – это поперечное ребро.

Листинг 1. Реализация алгоритма поиска в глубину

```
vector<vector<int>> adj; // граф представляется в виде
списка смежности
int n; // количество вершин

vector<bool> visited;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
        dfs(u);
    }
}
```

Это самая простая реализация поиска в глубину. Как описано выше, в применениях, может быть полезно также вычислить время входа и выхода и цвет вершины. Мы будем окрашивать все вершины цветом 0, если мы их еще не посетили, цветом 1, если мы их посетили, и цветом 2, если мы уже вышли из вершины.

Листинг 2. Реализация алгоритма поиска в глубину с использованием времени входа и выхода

```
vector<vector<int>> adj; // граф представляется в виде списка смежности
int n; // количество вершин

vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v) {
    time_in[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
   color[v] = 2;
   time_out[v] = dfs_timer++;
}
```

4.7. Примеры решения задач с использованием алгоритма DFS

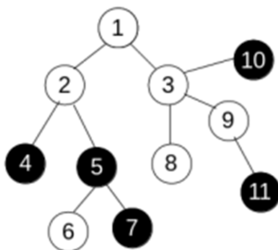
Антон и дерево

Антон посадил в своём саду дерево. Напоминаем, что дерево — это связный неориентированный граф, не содержащий циклов.

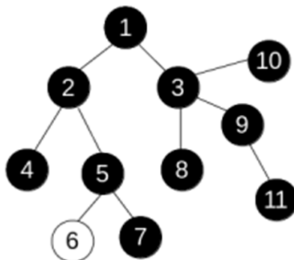
Вскоре дерево выросло. Каждая из его вершин оказалась покрашена либо в чёрный, либо в белый цвет. Однако Антону не нравятся разноцветные деревья, и поэтому он решил покрасить дерево так, чтобы оно было либо полностью белым, либо полностью чёрным.

Для изменения цвета вершин дерева Антону доступна только одна операция, обозначим её как *paint*(v), где v – некоторая вершина дерева.

Эта операция перекрашивает в противоположный цвет все такие вершины u, что на кратчайшем пути от u до v все вершины окрашены в один цвет (при этом сами вершины u и v также учитываются). Например, дерево



после применения операции *paint*(3) станет следующим:



Антону стало интересно, за какое минимальное количество операций покраски ему удастся сделать так, чтобы дерево покрасилось в один цвет. Помогите ему найти это число!

Входные данные

В первой строке входных данных находится одно целое число $n (1 \leq n \leq 200\,000)$ – количество вершин в дереве.

Во второй строке входных данных находятся n целых чисел $color_i$ $(0 \leq color_i \leq 1)$ – цвета вершин. Если *color i* = 0, это значит, что i-я вершина покрашена в белый цвет, если *color i* = 1, это значит, что i-я вершина покрашена в чёрный цвет.

В каждой из следующих $n - 1$ строк входных данных находится пара целых чисел u_i и v_i $(1 \leq u_i, v_i \leq n, u_i \neq v_i)$– рёбра дерева. Гаран-

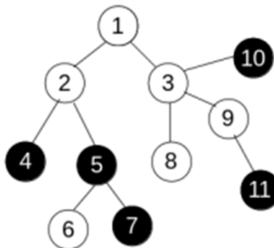
тируется, что все пары (u_i, v_i) различны, то есть граф не содержит кратных рёбер.

Выходные данные

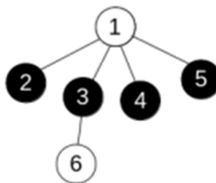
В единственной строке выходных данных выведите единственное число – минимальное количество операций покраски, которое необходимо Антону, чтобы дерево стало либо полностью белым, либо полностью черным.

Разбор

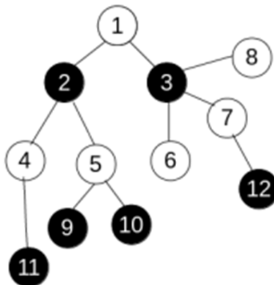
Для начала можно заметить, что если две вершины одинакового цвета, соединенные ребром, объединить в одну, то ответ не изменится. Давайте так и сделаем. Тогда, например, дерево



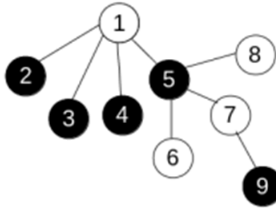
превратится в дерево



Будем так же делать такое «сжатие» дерева после каждой операции покраски. Тогда, например, дерево



после применения операции *paint*(2) и «сжатия» дерева превратится в следующее:



Понятно, что дерево будет покрашено в один цвет тогда и только тогда, когда после таких операций покраски со «сжатием» останется ровно одна вершина.

Давайте назовем диаметром дерева максимально возможную длину кратчайшего расстояния между двумя вершинами в дереве. Нетрудно заметить, что дерево будет покрашено в один цвет тогда и только тогда, когда диаметр дерева станет равен 0, поскольку диаметр равен 0 только в дереве, состоящем из одной вершины.

Теперь можно заметить следующее: диаметр дерева не может уменьшиться более чем на два за одну операцию покраски со «сжатием». Поэтому ответ будет не менее $\frac{d+1}{2}$, где d – диаметр дерева.

Теперь докажем, что всегда можно покрасить дерево за $\frac{d+1}{2}$ операций. Для этого давайте найдем такую вершину, где длина кратчайшего расстояния от нее до любой другой вершины не превышает $\frac{d+1}{2}$. Такая вершина всегда найдется, поскольку иначе диаметр дерева будет не менее $d + 1$, что невозможно. Можно заметить, что, применив $\frac{d+1}{2}$ операций покраски к этой вершине, мы покрасим дерево в один цвет.

Сложность решения – $O(n)$.

Листинг. Решение задачи

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

using namespace std;

int n;
vector <int> color;
```

```
vector < vector <int> > g;
vector <char> used;
vector <int> comp;
int n1;
vector < vector <int> > g1;
vector <int> dp;
int ans = 0;

void dfs1(int v, int col, int cmp)
{
    if (used[v]) return;
    if (color[v] != col) return;
    used[v] = true;
    comp[v] = cmp;
    for (int i = 0; i < g[v].size(); i++)
    {
        int to = g[v][i];
        dfs1(to, col, cmp);
    }
}

void dfs2(int v, int p = -1)
{
    int mx1 = 0, mx2 = 0;
    for (int i = 0; i < g1[v].size(); i++)
    {
        int to = g1[v][i];
        if (to == p) continue;
        dfs2(to, v);
        int val = dp[to] + 1;
        mx2 = max(mx2, val);
        if (mx1 < mx2) swap(mx1, mx2);
    }
    dp[v] = mx1;
    ans = max(ans, mx1 + mx2);
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin >> n;
    color.resize(n);
    g.resize(n);
```

```
    comp.resize(n);
    used.assign(n, false);
    for (int i = 0; i < n; i++) cin >> color[i];
    for (int i = 1; i < n; i++)
    {
        int a, b; cin >> a >> b; a--, b--;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i, color[i], n1++);
    g1.resize(n1);
    dp.resize(n1);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < g[i].size(); j++)
        {
            int to = g[i][j];
            if (comp[i] != comp[to])
               g1[comp[i]].push_back(comp[to]);
        }
    dfs2(0);
    cout << (ans + 1) / 2 << endl;
    return 0;
}
```

4.8. Алгоритм Дейкстры

Вам дается ориентированный или неориентированный взвешенный граф с n вершинами и m ребрами. Веса всех ребер неотрицательны. Вам также дается начальная вершина s. Мы рассмотрим поиск длин кратчайших путей от начальной вершины s до всех других вершин и вывод самих кратчайших путей.

Такую задачу называют «задачей о кратчайших путях с единственным источником» (single-source shortest paths problem).

Алгоритм, описанный голландским ученым Э.В. Дейкстрой в 1959 году.

Создадим массив $d[\ ]$, где для каждой вершины v длина кратчайшего пути от s до v равна $d[v]$. Изначально $d[s] = 0$, а для всех остальных вершин эта длина равна бесконечности. В реализации

в качестве бесконечности выбирается достаточно большое число (которое гарантированно будет больше любой возможной длины пути).

$$\mathrm{d}[v] = \infty, \quad v \neq \mathrm{s}.$$

Кроме того, мы храним логический массив *u*[ ], который хранит для каждой вершины v, помечена ли она. Изначально все вершины не помечены:

$$u[v] = \mathit{false}.$$

Алгоритм Дейкстры работает в течение *n* итераций. На каждой итерации вершина v выбирается как непомеченная вершина, имеющая наименьшее значение $d[v]$.

Очевидно, что на первой итерации будет выбрана начальная вершина *s*.

Выбранная вершина v помечается. Далее из вершины v выполняются релаксации: рассматриваются все ребра вида (v, to), и для каждой вершины алгоритм пытается улучшить значение $d[\mathit{to}]$. Если длина текущего ребра равна *len*, то код для релаксации выглядит так:

$$d[\mathit{to}] = \min(d[\mathit{to}], d[v] + \mathit{len}).$$

После того, как все такие ребра будут рассмотрены, текущая итерация заканчивается. Затем после *n* итераций все вершины будут помечены, и алгоритм завершится. Найденные значения $d[v]$ – это длины кратчайших путей от *s* до всех вершин v.

Заметим, что если некоторые вершины недоступны из начальной вершины *s*, то значения $d[v]$ для них останутся бесконечными. Очевидно, что последние несколько итераций алгоритма будут выбирать эти вершины, но никакой полезной работы для них не будет сделано. Поэтому алгоритм может быть остановлен, как только выбранная вершина получит бесконечное расстояние до нее.

4.8.1. Восстановление кратчайших путей

Обычно нужно знать не только длину кратчайших путей, но и сами кратчайшие пути. Давайте посмотрим, как сохранить достаточную информацию, чтобы восстановить кратчайший путь от *s* до любой вершины. Мы будем хранить массив предшественников *p*[ ], в котором для каждой вершины $v \neq s$ $p[v]$ является предпо-

следней вершиной на кратчайшем пути от s до v. Мы используем тот факт, что если мы возьмем кратчайший путь к некоторой вершине v и удалим v из этого пути, то получим путь, заканчивающийся в вершине $p[v]$, и этот путь будет самым коротким для вершины $p[v]$. Этот массив предшественников можно использовать для восстановления кратчайшего пути к любой вершине: начиная с v, берите предшественника текущей вершины, пока мы не достигнем начальной вершины s, чтобы получить необходимый кратчайший путь с вершинами, перечисленными в обратном порядке. Таким образом, кратчайший путь P к вершине v равен

$$P = (s, \ldots, p[\mathrm{p}[p[v]]],\ p[p[v]],\ p[v], v).$$

Построение этого массива предшественников очень просто: для каждой успешной релаксации, т. е. когда для некоторой выбранной вершины происходит улучшение расстояния до некоторой вершины to, мы записываем, что предком вершины to является вершина v:

$$p[to] = v.$$

Основное утверждение, на котором основана корректность алгоритма Дейкстры, заключается в следующем.

После того как любая вершина v будет помечена, текущее расстояние до нее $d[v]$ будет самым коротким и больше не будет меняться.

Доказательство производится путем индукции. Для первой итерации это утверждение очевидно: единственная отмеченная вершина – s, а расстояние до нее $d[s] = 0$ – это действительно длина кратчайшего пути до s. Теперь предположим, что это утверждение верно для всех предыдущих итераций, т. е. для всех уже помеченных вершин; докажем, что оно не нарушается после завершения текущей итерации. Пусть v – вершина, выбранная в текущей итерации, т. е. v – вершина, которую будет помечать алгоритм. Теперь мы должны доказать, что $d[v]$ действительно равно длине кратчайшего пути к нему $l[v]$.

Рассмотрим кратчайший путь P к вершине v. Этот путь можно разделить на две части: P_1, которая состоит только из отмеченных вершин (по крайней мере, начальная вершина s является частью P_1), и остальную часть пути P_2 (она может включать в себя помеченную

вершину, но всегда начинается с непомеченной вершины). Обозначим первую вершину пути P_2 как p, а последнюю вершину пути P_1 как q.

Сначала мы докажем наше утверждение для вершины p, то есть докажем, что $d[p] = l[p]$. Это почти очевидно: на одной из предыдущих итераций мы выбрали вершину q и выполнили релаксацию из нее. Так как (в силу выбора вершины p) кратчайший путь к p – это кратчайший путь к q плюс ребро (p, q), то релаксация от q задает величину $d[p]$ к длине кратчайшего пути $l[p]$.

Поскольку веса ребер неотрицательны, длина кратчайшего пути $l[p]$ (которая, как мы только что доказали, равна $d[p]$) не превышает длины $l[v]$ кратчайшего пути к вершине v. Учитывая, что $l[v] \le d[v]$ (поскольку алгоритм Дейкстры не мог найти более короткий путь, чем самый короткий из возможных), мы получаем неравенство

$$d[p] = l[p] \le l[v] \le d[v].$$

С другой стороны, поскольку обе вершины p и v не помечены, а в текущей итерации выбрана вершина v, а не p, мы получаем еще одно неравенство:

$$d[p] \ge d[v].$$

Из этих двух неравенств мы заключаем, что $d[p] = d[v]$, а затем из ранее найденных уравнений получаем:

$$d[v] = l[v].$$

Что и требовалось доказать.

Алгоритм Дейкстры выполняет n итераций. И на каждой итерации он выбирает непомеченную вершину v с наименьшим значением $d[v]$, помечает ее и проверяет все рёбра (v, to), пытаясь улучшить значение $d[to]$.

Время выполнения алгоритма состоит из:

- n поисков вершины с наименьшим значением $d[v]$ среди $O(n)$ непомеченных вершин;
- m попыток релаксации.

Для простейшей реализации этих операций на каждой итерации поиска вершин требуется $O(n)$ операций, и каждая релаксация может быть выполнена в $O(1)$. Следовательно, результирующее асимптотическое поведение алгоритма составит $O(n^2 + m)$.

Здесь граф *adj* хранится как список смежности: для каждой вершины v *adj*[v] содержит список ребер, идущих из этой вершины, т. е. список *pair* < *int*, *int* >, где первый элемент в паре – вершина на другом конце ребра, а второй элемент – вес ребра.

Функция принимает начальную вершину *s* и два вектора, которые будут использоваться в качестве возвращаемых значений.

Прежде всего, код инициализирует массивы: расстояния *d*[ ], меток *u*[ ] и предшественников *p*[ ]. Затем он выполняет *n* итераций. На каждой итерации выбирается вершина v, которая имеет наименьшее расстояние *d*[v] среди всех непомеченных вершин. Если расстояние до выбранной вершины v равно бесконечности, то алгоритм останавливается. В противном случае вершина помечается, и все ребра, выходящие из этой вершины, проверяются. Если релаксация по ребру возможна (т. е. расстояние *d*[*to*] может быть улучшено), то расстояние *d*[*to*] и предшественник *p*[*to*] обновляются.

После выполнения всех итераций массив *d*[ ] хранит длины кратчайших путей ко всем вершинам, а массив *p*[ ] хранит предшественников всех вершин (кроме начальной вершины *s*). Путь к любой вершине *t* может быть восстановлен следующим образом.

Листинг. Восстановление пути

```
vector<int> restore_path(int s, int t, vector<int> const& p) {
        vector<int> path;

        for (int v = t; v != s; v = p[v])
            path.push_back(v);
        path.push_back(s);

        reverse(path.begin(), path.end());
        return path;
}
```

4.9. Примеры решения задач с использованием алгоритма Дейкстры

Президент и дороги

В Берляндии n городов, столица находится в городе s, а историческая родина Президента – в городе t ($s \neq t$). Города соединены односторонними дорогами, время проезда по каждой из дорог – целое положительное число.

Раз в год Президент посещает свою историческую родину t, для чего его кортеж проезжает по некоторому пути из s в t (обратно он всегда возвращается на личном самолете). Так как Президент – очень занятой человек, то он всегда выбирает путь из s в t, по которому он проедет быстрее всего.

Министерство дорог и путей сообщения хочет узнать для каждой дороги: обязательно ли проедет по ней Президент во время своего путешествия, и если нет, то возможно ли её починить так, чтобы она в любом случае входила в кратчайший путь из столицы на историческую родину Президента. Очевидно, что дорогу невозможно починить так, чтобы время проезда по ней стало меньше единицы. Министерство Берляндии, как и любое другое, заинтересовано в сохранении бюджета, поэтому оно хочет узнать минимальную стоимость починки дороги. Также оно очень любит точность, поэтому ремонтирует дороги так, что время проезда по ним всегда остаётся целым числом.

Входные данные

В первой строке записаны четыре целых числа n, m, s и t ($2 \leq n \leq 105$; $1 \leq m \leq 105$; $1 \leq s, t \leq n$) – количество городов и дорог в Берляндии, номера столицы и исторической родины Президента ($s \neq t$).

Далее в m строках перечислены дороги. Каждая из дорог задается тройкой целых чисел a_i, b_i, l_i ($1 \leq a_i, b_i \leq n$; $a_i \neq b_i$; $1 \leq l_i \leq 106$) – городами, которые соединяет i-я дорога, и временем проезда по ней. Дорога направлена от города a_i к городу b_i.

Города пронумерованы от 1 до n. Между парой городов может быть несколько дорог. Гарантируется, что существует путь из s в t по дорогам.

Выходные данные

Выведите m строк. В i-й строке должна содержаться информация об i-й дороге (дороги нумеруются с единицы в порядке следования во входных данных).

Если Президент в любом случае проедет по ней во время своего путешествия, строка должна содержать единственное слово «YES» (без кавычек).

Иначе если i-ю дорогу возможно починить так, чтобы время проезда по ней осталось положительным и Президент в любом случае выбрал бы её для своего путешествия, выведите через пробел слово «CAN» (без кавычек) и минимальную стоимость ремонта.

Если же нельзя починить дорогу так, чтобы по ней в обязательном порядке проехал Президент, выведите «NO» (без кавычек).

Разбор

Сначала давайте разберёмся, какие рёбра не будут лежать ни на одном кратчайшем пути из s в t. Если запустить два алгоритма поиска кратчайших путей (из вершины s и из вершины t) и сохранить расстояния в массивах $d1$ и $d2$ соответственно, можно сделать следующий вывод: если у нас есть ребро (u, v), то оно будет лежать хотя бы на одном кратчайшем пути из s в t тогда и только тогда, когда $d1[u] + w(u, v) + d2[v] = d1[t]$ (где $w(u, v)$ – вес ребра (u, v)).

После того, как мы построили граф кратчайших путей из s в t, мы можем определить, какие из рёбер лежат на всех кратчайших путях. Если представить путь из s в t в виде отрезка $[0...D]$, а расстояния между парами вершин, соединённых рёбрами в этом графе, в виде подотрезков данного отрезка (например, если у нас есть ребро (u, v), тогда данный подотрезок будет иметь вид $[d1[u]...d1[v]]$), то можно заметить, что все подотрезки каким-то образом касаются других подотрезков (некоторые также могут пересекаться с другими подотрезками). Ребро (u, v) будет лежать на всех кратчайших путях из s в t, если подотрезок $[d1[u]...d1[v]]$ будет только касаться других подотрезков (внутренние пересечения с другими подотрезками недопустимы). Теперь мы можем однозначно отвечать «YES» для этих рёбер.

Остальная часть задачи более простая. Если ребро (u, v) с весом w не лежит на всех кратчайших путях, можно попробовать уменьшить

его. Данное ребро будет лежать на всех кратчайших путях только в том случае, если его вес станет равен $d1[t] - d1[u] - d2[v] - 1$. Итак, если величина $d1[t] - d1[u] - d2[v] - 1$ (предполагаемый новый вес нашего ребра) строго положительна, тогда мы сможем уменьшить наше ребро так, чтобы оно лежало на всех кратчайших путях, выведем «CAN» и через пробел разность между старым и новым весами ребра. Иначе же выведем «NO».

Листинг. Решение задачи

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

typedef long ll;
typedef pair <ll, ll> llll;

const int Maxn = 100005;
const int mod1 = 1000000007;
const int mod2 = 1000000009;
const ll Inf = 1000000000000000000ll;

int n, m, s, t;
int a[Maxn], b[Maxn], l[Maxn];
vector <llll> neigh[Maxn], rneigh[Maxn];
ll dist1[Maxn], dist2[Maxn];
llll ways1[Maxn], ways2[Maxn];

void Dijkstra(int s, const vector <llll> neigh[], ll
dist[], llll ways[])
{
    fill(dist, dist + n + 1, Inf); dist[s] = 0;
    ways[s] = llll(1, 1);
    priority_queue  <llll>  Q;  Q.push(llll(-dist[s],
s));
    while (!Q.empty()) {
          ll v = Q.top().second, d = -Q.top().first;
Q.pop();
          if (dist[v] != d) continue;
          for (int i = 0; i < neigh[v].size(); i++) {
```

```
                    llll u = neigh[v][i];
                    if (d + u.second < dist[u.first]) {
                          dist[u.first] = d + u.second;
ways[u.first] = llll(0, 0);
                          Q.push(llll(-dist[u.first],
u.first));
                    }
                    if (d + u.second == dist[u.first]) {
                          ways[u.first].first = (ways[u.
first].first + ways[v].first) % mod1;
                          ways[u.first].second = (ways[u.
first].second + ways[v].second) % mod2;
                    }
              }
        }
   }

   int main()
   {
        cin >> n >> m >> s >> t;
        for (int i = 0; i < m; i++) {
              cin >> a[i] >> b[i] >> l[i];
              neigh[a[i]].push_back(llll(b[i], l[i]));
              rneigh[b[i]].push_back(llll(a[i], l[i]));
        }
        Dijkstra(s, neigh, dist1, ways1);
        Dijkstra(t, rneigh, dist2, ways2);
        ll res = dist1[t];
        for (int i = 0; i < m; i++)
              if (dist1[a[i]] + l[i] + dist2[b[i]] == res
&&
                    ll(ways1[a[i]].first) * ways2[b[i]].
first % mod1 == ways1[t].first &&
                    ll(ways1[a[i]].second) * ways2[b[i]].
second % mod2 == ways1[t].second)
                    cout << "YES\n";
              else {
                    ll x = res - dist1[a[i]] - dist2[b[i]]
- 1;
                    if (x > 0)
                          cout << "CAN " << l[i] - x <<
endl;
```

```
                else
                        cout << "NO\n";
        }
    return 0;
}
```

4.10. Алгоритм поиска компонент связности в графе

Дан неориентированный граф G с n вершинами и m рёбрами. Требуется найти в нём все компоненты связности, т. е. разбить вершины графа на несколько групп так, что внутри одной группы можно дойти от одной вершины до любой другой, а между разными группами пути не существует.

Для решения можно воспользоваться как обходом в глубину, так и обходом в ширину.

Фактически мы будем производить серию обходов: сначала запустим обход из первой вершины, и все вершины, которые он при этом обошёл, образуют первую компоненту связности. Затем найдём первую из оставшихся вершин, которые ещё не были посещены, и запустим обход из неё, найдя тем самым вторую компоненту связности. И так далее, пока все вершины не станут помеченными.

Итоговая асимптотика составит $O(n + m)$: в самом деле, такой алгоритм не будет запускаться от одной и той же вершины дважды, а значит, каждое ребро будет просмотрено ровно два раза (с одного конца и с другого конца).

Для реализации чуть более удобным является обход в глубину.

Листинг. Реализация алгоритма поиска компонент связанности

```
int n;
vector<int> g[MAXN];
bool used[MAXN];
vector<int> comp;

void dfs (int v) {
    used[v] = true;
    comp.push_back (v);
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (! used[to])
```

```
                dfs (to);
    }
}

void find_comps() {
    for (int i=0; i<n; ++i)
        used[i] = false;
    for (int i=0; i<n; ++i)
        if (! used[i]) {
            comp.clear();
            dfs (i);

            cout << "Component:";
            for (size_t j=0; j<comp.size(); ++j)
                cout << ' ' << comp[j];
            cout << endl;
        }
}
```

Основная функция для вызова — find_comps(), она находит и выводит компоненты связности графа.

Мы считаем, что граф задан списками смежности, т. е. $g[i]$ содержит список вершин, в которые есть рёбра из вершины i. Константе MAXN следует задать значение, равное максимально возможному количеству вершин в графе.

Вектор comp содержит список вершин в текущей компоненте связности.

4.11. Алгоритм топологической сортировки вершин графа

Дан ориентированный граф с n вершинами и m рёбрами. Требуется перенумеровать его вершины таким образом, чтобы каждое ребро вело из вершины с меньшим номером в вершину с большим.

Иными словами, требуется найти перестановку вершин (топологический порядок), соответствующую порядку, задаваемому всеми рёбрами графа.

Топологическая сортировка может быть не единственной (например, если граф — пустой; или если есть три такие вершины a, b, c, что из a есть пути в b и в c, но ни из b в c, ни из c в b добраться нельзя).

Топологической сортировки может не существовать вовсе — если граф содержит циклы (поскольку при этом возникает противоречие: есть путь и из одной вершины в другую, и наоборот).

Распространённая задача на топологическую сортировку — следующая. Есть n переменных, значения которых нам неизвестны. Известно лишь про некоторые пары переменных, что одна переменная меньше другой. Требуется проверить, не противоречивы ли эти неравенства, и если нет, выдать переменные в порядке их возрастания (если решений несколько — выдать любое). Легко заметить, что это в точности и есть задача о поиске топологической сортировки в графе из n вершин.

Для решения воспользуемся обходом в глубину.

Предположим, что граф ацикличен, т. е. решение существует. Что делает обход в глубину? При запуске из какой-то вершины v он пытается запуститься вдоль всех рёбер, исходящих из v. Вдоль тех рёбер, концы которых уже были посещены ранее, он не проходит, а вдоль всех остальных — проходит и запускает себя из конца проходимых ребер.

Таким образом, к моменту выхода из вызова $dfs(v)$ все вершины, достижимые из v как непосредственно (по одному ребру), так и косвенно (по пути) — все такие вершины уже посещены обходом. Следовательно, если мы будем в момент выхода из $dfs(v)$ добавлять нашу вершину в начало некоего списка, то в конце концов в этом списке получится топологическая сортировка.

Эти объяснения можно представить и в несколько ином свете, с помощью понятия «времени выхода» обхода в глубину. Время выхода для каждой вершины v — это момент времени, в который закончил работать вызов $dfs(v)$ обхода в глубину от неё (времена выхода можно занумеровать от 1 до n). Легко понять, что при обходе в глубину время выхода из какой-либо вершины v всегда больше, чем время выхода из всех вершин, достижимых из неё (так как они были посещены либо до вызова $dfs(v)$, либо во время него). Таким образом, искомая топологическая сортировка — это сортировка в порядке убывания времён выхода.

Приведём реализацию, предполагающую, что граф ацикличен, т. е. искомая топологическая сортировка существует. При необхо-

димости проверку графа на ацикличность легко вставить в обход в глубину, как описано в статье по обходу в глубину.

Листинг 1. Алгоритм топологической сортировки вершин графа

```
int n; // число вершин
vector<int> g[MAXN]; // граф
bool used[MAXN];
vector<int> ans;

void dfs (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (!used[to])
            dfs (to);
    }
    ans.push_back (v);
}

void topological_sort() {
    for (int i=0; i<n; ++i)
        used[i] = false;
    ans.clear();
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs (i);
    reverse (ans.begin(), ans.end());
}
```

Здесь константе MAXN следует задать значение, равное максимально возможному числу вершин в графе.

Основная функция решения – это topological_sort, она инициализирует пометки обхода в глубину, запускает его, и ответ в итоге получается в векторе ans.

Пример решения задачи на топологическую сортировку вершин графа

Система родственных отношений у марсиан достаточно запутана. Собственно говоря, марсиане почкуются когда им угодно и как им угодно, собираясь для этого разными группами, так что у марсианина может быть и один родитель, и несколько десятков, а сотней

детей сложно кого-нибудь удивить. Марсиане привыкли к этому, и такой жизненный уклад кажется им естественным.

А вот в Планетарном Совете запутанная генеалогическая система создает серьёзные неудобства. Там заседают достойнейшие из марсиан, и поэтому, чтобы никого не обидеть, во всех обсуждениях слово принято предоставлять по очереди, так, чтобы сначала высказывались представители старших поколений, потом те, что помладше, и лишь затем уже самые юные и бездетные марсиане. Однако соблюдение такого порядка на деле представляет собой непростую задачу. Не всегда марсианин знает всех своих родителей, что уж тут говорить про бабушек и дедушек! Но когда по ошибке сначала высказывается праправнук, а потом только молодо выглядящий прапрадед – это настоящий скандал.

Задача

Ваша цель – написать программу, которая определила бы раз и навсегда такой порядок выступлений в Планетарном Совете, который гарантировал бы, что каждый член совета получает возможность высказаться раньше любого из своих потомков.

Исходные данные

В первой строке входных данных к этой задаче находится единственное число N, $1 \le N \le 100$ – количество членов Марсианского Планетарного Совета. По многовековой традиции все члены Совета нумеруются целыми числами от 1 до N. Далее следуют ровно N строк, причем i-я строка содержит список детей члена Совета с порядковым номером i. Список детей представляет собой последовательность порядковых номеров детей, разделенных пробелами и следующих в произвольном порядке. Список детей может быть пустым. Список детей (даже если он пуст) оканчивается нулем.

Результат

Выход должен содержать последовательность номеров выступающих, разделенных пробелами. Если несколько последовательностей удовлетворяют условиям задачи, то можно вывести любую из них. Гарантируется, что хотя бы одна такая последовательность существует.

Условие задачи полностью повторяет определение топологической сортировки, где в качестве вершин графа выступают марсиане, а в качестве рёбер – их родственные связи. Для решения воспользуемся приведенным выше алгоритмом.

Листинг 2. Решение задачи

```
#include <bits/stdc++.h>

using namespace std;

vector <int> used(120, 0), ans;
vector < vector <int> > g(120);
int n;

void dfs(int v){
    used[v] = true;
    for(int i = 0; i < g[v].size(); i++){
        int to = g[v][i];
        if(!used[to]){
            dfs(to);
        }
     }
     ans.push_back(v);
}

void top_sort(){
    ans.clear();
    used.assign(n, 0);
    for(int i = 0; i < n; i++){
        if(!used[i]){
            dfs(i);
        }
    }
    reverse(ans.begin(), ans.end());
}

int main()
{
    cin >> n;
    for(int i = 0; i < n; i++){
        int tmp;
        cin >> tmp;
        while(tmp){
```

```
            g[i].push_back(--tmp);
            cin >> tmp;
        }
    }

    top_sort();
    for(auto i: ans)
        cout << i + 1<< « «;
    return 0;
}
```

5. ВСПОМОГАТЕЛЬНЫЕ СТРУКТУРЫ ДАННЫХ

5.1. Дерево отрезков

Рассмотрим следующую задачу, чтобы понять, что такое деревья отрезков. Есть массив $a[0...n-1]$. Мы должны уметь:

- находить сумму элементов массива с индекса l по индекс r;
- изменять значение указанного элемента массива на новое значение x.

Простым решением является запуск цикла от l до r и вычисление суммы элементов в заданном диапазоне. Чтобы обновить значение, просто сделаем $a[i] = x$. Первая операция занимает $O(n)$ времени, а вторая – $O(1)$.

Другим решением является создание другого массива и сохранение суммы от начала до i в i-м индексе в этом массиве. Сумма заданного диапазона теперь может быть рассчитана в течение $O(1)$, но операция обновления теперь занимает $O(n)$. Это хорошо работает, если количество операций запроса велико и очень мало обновлений.

Что, если количество запросов и обновлений одинаково? Можно ли выполнить обе операции в $O(\log n)$? Мы можем использовать дерево отрезков для выполнения обеих операций в $O(\log n)$ времени.

Представление дерева отрезков:

- листья дерева – это элементы входного массива;
- каждая внутренняя вершина дерева представляет собой слияние некоторых листьев дерева. Для различных задач слияние может быть различным. В данной задаче слияние представляет собой сумму значений в листьях, находящихся под данной вершиной.

Для представления дерева отрезков в памяти компьютера используют массивы. Так, элемент с индексом 1 является корнем дерева, а для каждого элемента i левый сын имеет индекс $2 \cdot i$, а правый – $2 \cdot i + 1$.

Для построения дерева отрезков начнем с отрезка $a[0...n-1]$. Каждый раз мы делим текущий отрезок на две половины (если он еще не стал отрезком длины 1), а затем вызываем одну и ту же процеду-

ру на обеих половинах, и для каждого такого отрезка мы сохраняем сумму в соответствующем узле.

Все уровни построенного дерева отрезков будут полностью заполнены, за исключением последнего уровня. Кроме того, дерево будет полным двоичным деревом, потому что мы всегда разделяем отрезки на две половины на каждом уровне. Поскольку построенное дерево всегда является полным двоичным деревом с n листьями, у него будут $n - 1$ внутренних узлов. Таким образом, общее число узлов будет $2 \cdot n - 1$.

Как только дерево построено, как получить сумму, используя построенное дерево отрезков? Для этого будем рекурсивно спускаться по дереву отрезков. Изначально запустим функцию получения суммы из корня дерева. Теперь может возникнуть два случая: отрезок запроса полностью находится в одном из сыновей корня, или одна часть отрезка находится в левом сыне, а другая в правом. В первом случае мы рекурсивно вызовем функцию от левого или правого сына. Во втором случае необходимо посчитать ответ сначала в левом сыне, затем в правом, а после этого сложить полученные результаты.

Подобно построению дерева и операциям запроса, обновление также можно выполнить рекурсивно. Нам предоставляется индекс, который необходимо обновить. Пусть *diff* – значение, которое нужно добавить. Мы начинаем с корня дерева отрезков и добавляем *diff* ко всем узлам, которые содержат данный индекс в своем диапазоне. Если узел не содержит данный индекс в своем диапазоне, мы не вносим никаких изменений в этот узел.

Временная сложность для построения дерева – $O(n)$. Всего в дереве $2n - 1$ узлов, а значение каждого узла вычисляется только один раз.

Временная сложность запроса – $O(\log n)$. Чтобы запросить сумму, мы обрабатываем не более четырех узлов на каждом уровне, а количество уровней – $O(\log n)$.

Сложность обновления также $O(\log n)$. Чтобы обновить значение листа, мы обрабатываем один узел на каждом уровне, а количество уровней – $O(\log n)$.

5.2. Задача, использующая в решении дерево отрезков

Рассмотрим задачу, объединяющую алгоритм хэширования и дерево отрезков. Дана строка *s* (длина строки от 1 до 100 000 символов). Поступают запросы двух видов. Первый – изменить в слове букву на позиции *i*. Второй – проверить, является ли подстрока палиндромом.

Алгоритм решения задачи следующий. Построим два дерева отрезков: одно из хэшей символов строки *s*, а другое из хэшей символов строки *s*, записанной в обратном порядке. Тогда для запроса первого типа в обоих деревьях меняются значения хэша для символа на данной позиции на новые. А для запроса второго типа считаются хэш подстроки левой половины и хэш подстроки правой половины. А после за $O(1)$ проверяется равенство этих двух хэшей. Рассмотрим программный код решения данной задачи.

На рис. 6 изображен фрагмент программного кода, который реализует ввод данных, а именно строку *s* и количество запросов *m*, а также считаются степени числа *p* и строятся деревья отрезков, в виде последовательных вызовов функции *update*.

```
30  int main()
31  {
32      string s;
33      cin >> s >> m;
34      n = s.length();
35      long long p = 1;
36      for(int i = 0; i < n; i++) pows[i] = p, p *= 31;
37      for(int i = 0; i < n; i++){
38          update(0, 1, 0, n - 1, i, (s[i] - 'a' + 1) * pows[i]);
39          update(1, 1, 0, n - 1, i, (s[n - 1 - i] - 'a' + 1) * pows[i]);
40      }
```

Рис. 6. Фрагмент кода ввода данных и построения деревьев

На рис. 7 изображен фрагмент кода обработки запросов. Вначале вводится тип запроса. Если ввели запрос первого типа, то вычисляем хэши этого символа в строке *s* и «развернутой» строке *s* и обновляем эти значения в деревьях. Если ввели запрос второго типа, то вычисляются хэш подстроки левой половины и хэш подстроки правой половины, затем они проверяются на равенство.

```
41      for(int i = 0; i < m; i++){
42          string t;
43          cin >> t;
44          if(t[0] == 'p'){
45              int l, r;
46              cin >> l >> r;
47              if(l == r){
48                  cout << "Yes" << endl;
49                  continue;
50              }
51              l--, r--;
52              int len = (r - l + 1) / 2;
53              int add = 0;
54              if((r - l + 1) % 2) add = 1;
55              long long h1 = get(0, 1, 0, n - 1, l, l + len - 1);
56              long long h2 = get(1, 1, 0, n - 1, n - 1 - r, n - 1 - (l + len + add));
57              if(h1 * pows[n - 1 - r] == h2 * pows[l]) cout << "Yes" << endl;
58              else cout << "No" << endl;
59          }
60          else{
61              int pos;
62              char val;
63              cin >> pos >> val;
64              int delta = val - s[pos - 1];
65              s[pos - 1] = val;
66              update(0, 1, 0, n - 1, pos - 1, delta * pows[pos - 1]);
67              update(1, 1, 0, n - 1, n - 1 - (pos - 1), delta * pows[n - 1 - (pos - 1)]);
68          }
69      }
```

Рис. 7. Фрагмент кода обработки запросов

```
10  void update(int t, int v, int vl, int vr, int pos, long long val){
11      if(vl == vr){
12          tree[t][v] += val;
13          return;
14      }
15      int vm = (vl + vr) >> 1;
16      if(pos <= vm) update(t, 2 * v, vl, vm, pos, val);
17      else update(t, 2 * v + 1, vm + 1, vr, pos, val);
18      tree[t][v] = tree[t][2 * v] + tree[t][2 * v + 1];
19  }
20
21  long long get(int t, int v, int vl, int vr, int l, int r){
22      if(l > vr || r < vl) return 0;
23      if(l <= vl && vr <= r) return tree[t][v];
24      int vm = (vl + vr) >> 1;
25      long long q1 = get(t, 2 * v, vl, vm, l, r);
26      long long q2 = get(t, 2 * v + 1, vm + 1, vr, l, r);
27      return q1 + q2;
28  }
```

Рис. 8. Фрагмент кода метода изменения элемента
и метода запроса суммы

На рис. 8 изображена реализация методов, обеспечивающих работу с деревьями отрезков. В качестве параметров функции *update* передаются номер дерева, узел дерева, границы отрезка, сумма которого хранится в этом узле, позиция, на которой необходи-

мо поменять значение и само значение. Если данный узел является листом, то в нем меняется значение. Иначе вычисляем, в каком поддереве находится данная позиция, и рекурсивно вызываем функцию *update* от этого поддерева.

На рис. 9 представлен результат работы программы. Как видно, программа корректно обрабатывает данные запросы.

```
abcda
5
palindrome? 1 5
No
palindrome? 1 1
Yes
change 4 b
palindrome? 1 5
Yes
palindrome? 2 4
Yes
```

Рис. 9. Результат работы программы «Подпалиндромы»

Таким образом, время работы данного алгоритма $O(m \cdot \log(|s|))$, что гораздо эффективнее тривиального алгоритма за время $O(m \cdot |s|)$. При больших объемах входных данных это является существенной разницей.

5.3. Декартово дерево

В информатике декартово дерево и рандомизированное дерево двоичного поиска являются двумя тесно связанными формами структур данных двоичного дерева поиска, которые поддерживают динамический набор упорядоченных ключей и позволяют бинарный поиск среди ключей. После любой последовательности вставок и удалений ключей форма дерева представляет собой случайную переменную с тем же распределением вероятности, что и случайное двоичное дерево; в частности, с большой вероятностью его высота пропорциональна логарифму количества ключей, так что каждая операция поиска, вставки или удаления требует логарифмического времени для выполнения.

Рассмотрим один из вариантов декартова дерева – дерамиду. Дерамиды впервые описал Раймунд Сейдел и Сесилия Р. Арагон в 1989 году, его название получается из слияния слов *дерево* и *пирамида*. Это декартово дерево, в котором каждому ключу присваивается (случайно выбранный) числовой приоритет. Как и в любом двоичном дереве поиска, порядок обхода узлов совпадает с упорядоченным порядком ключей. Структура дерева определяется требованием, чтобы оно было упорядочено по куче: то есть номер приоритета для любого нелистового узла должен быть больше или равен приоритету его дочерних элементов. Таким образом, как и в случае с декартовыми деревьями в более общем смысле, корневой узел является узлом с максимальным приоритетом, а его левое и правое поддеревья формируются таким же образом из подпоследовательностей влево и вправо этого узла.

Эквивалентным способом описания дерамиды является то, что оно может быть сформировано путем вставки узлов с наивысшим приоритетом в двоичное дерево поиска без каких-либо перебалансировок. Поэтому если приоритетами являются независимые случайные числа, тогда дерамида имеет такое же распределение вероятности, как и случайное двоичное дерево поиска, дерево поиска, образованное путем вставки узлов без перебалансировки в произвольно выбранном порядке вставки. Поскольку, как известно, случайные двоичные деревья поиска имеют логарифмическую высоту с высокой вероятностью, то же самое верно для дерамид.

Дерамиды поддерживают следующие основные операции:

- поиск – чтобы найти заданное значение ключа, применим стандартный алгоритм двоичного поиска в двоичном дереве поиска, игнорируя приоритеты;
- вставка – чтобы вставить новый ключ x в дерамиду, создадим случайный приоритет y для x. Двоичным поиском найдем x в дереве и создадим новый узел в позиции листа, где двоичным поиском нашли позицию, где x должен быть. Затем, пока x не является корнем дерева и имеет больший приоритет, чем его родитель z, выполним поворот дерева, который меняет соотношение родитель – потомок между x и z;

- удаление – если x является листом дерева, просто удалим его. Если x имеет один дочерний узел v, удалим x из дерева и сделаем z дочерним по отношению к родительскому элементу x. Наконец, если x имеет двух детей, замените его положение в дереве с положением его непосредственного преемника z в отсортированном порядке, что приведет к одному из предыдущих случаев. В этом последнем случае своп может нарушать свойство упорядочения кучи для z, поэтому для восстановления этого свойства могут потребоваться дополнительные вращения.

6. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

6.1. «Разделяй и властвуй»

Некоторые задачи динамического программирования имеют схожую структуру $dp(i, j) = \min_{k \leq j}\{dp(i-1, k) + C(k, j)\}$, где $C(k, j)$ – любая фунция затрат.

Возьмем $1 \leq i \leq n$ и $1 \leq j \leq m$, функция C работает за $O(1)$. Прямая оценка вышеуказанного составляет $O(nm^2)$. Существует $n \times m$ состояний и m переходов для каждого состояния.

Это позволяет нам решать все состояния эффективнее. Скажем, мы вычисляем $opt(i, j)$ для некоторых фиксированных i и j. Тогда для любого $j' < j$ мы знаем, что $opt(i, j') \leq opt(i, j)$. Это означает, что при вычислении $opt(i, j')$ нам не нужно учитывать столько точек разделения.

Чтобы минимизировать время выполнения, мы применяем идею «разделяй и властвуй». Во-первых, вычислите $opt(i, n/2)$. Затем вычислите $opt(i, n/4)$, зная, что он меньше или равен $opt(i, n/2)$ и $opt(i, 3n/4)$ зная, что он больше или равен $opt(i, n/2)$. Рекурсивно отслеживая нижнюю и верхнюю границы opt, мы достигаем времени выполнения $O(mn\ loq\ n)$. Каждое возможное значение $opt(i, j)$ появляется в различных узлах $loq(n)$.

Обратите внимание, что не имеет значения, насколько «сбалансирован» $opt(i, j)$. На фиксированном уровне каждое значение k используется не более двух раз, и есть не более $loq(n)$ уровней.

Реализация варьируется в зависимости от проблемы. Это довольно общий шаблон.

Листинг. Общий шаблон применения стратегии «разделяй и влавствуй»

```
int n;
long long C(int i, int j);
vector<long long> dp_before(n), dp_cur(n);

// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr)
{
```

```
        if (l > r)
                return;
        int mid = (l + r) >> 1;
        pair<long long, int> best = { INF, -1 };

        for (int k = optl; k <= min(mid, optr); k++) {
                best = min(best, { dp_before[k] + C(k, mid),
k });
        }

        dp_cur[mid] = best.first;
        int opt = best.second;

        compute(l, mid - 1, optl, opt);
        compute(mid + 1, r, opt, optr);
    }
```

Пример с codeforces:

https://codeforces.com/contest/321/problem/E

Разбор

Решать будем через динамику:

$$dp[i][j] = \max\{k \mid dp[i-1][k] + costdp(k+1...j)\}$$

($dp[i][j]$ означает минимальную стоимость, если мы разделим $1...j$ лис в i группы).

Пусть $opt[i][j]$ = наименьший k такой, что $dp[i][j] = dp[i][k]$ + стоимость $(k + 1..j)$. Тогда интуитивно мы имеем

$$opt[i][1] <= opt[i][2] <= \ldots <= opt[i][n].$$

Пусть $n = 200$, и предположим, что мы уже получаем $dp[i][j]$ для $i <= 3$, и теперь нам нужно вычислить $dp[4][j]$: Если мы сначала вычисляем $dp[4][100]$, то мы так же можем узнать $opt[4][100]$.

И когда мы вычисляем $dp[4][1]... dp[4][99]$, мы знаем, что k должен лежать в $1... opt[4][100]$. Когда мы вычисляем $dp[4][101]... dp[4][200]$, мы знаем, что k должен лежать в $opt[4][100]...n$.

Мы используем $compute(d, L, R, optL, optR)$ для обозначения того, что мы вычисляем $dp[d][L...R]$, и мы знаем, что k должен быть в диапазоне $optL...optR$.

Тогда у нас есть: *compute*(d, L, R, *optL*, *optR*) =

1. Частный случай: $L = R$.
2. Пусть $M = (L + R)/2$, мы решаем *dp*[d][M], а также *opt*[d][M]. Используем операции *about*(*optR* – *optL* + 1).
3. *compute*(d, L, M – 1, *optL, opt*[d][M]).
4. *compute*(d, M + 1, R, *opt*[d][M], *optR*).

Можно показать, что это решение будет работать в $O(NloqN \cdot K)$. Нам не нужен *opt*[d][M] в центре интервала *optL*... *optR*. Мы можем доказать, что на каждой рекурсивной глубине общая стоимость по строке 2 будет не более $2n$, а также есть не более $O(loq(n))$ глубин.

6.2. Динамика по профилю. Задача «Паркет»

Общие задачи, решаемые с помощью динамического программирования на изломанном профиле, включают:

- поиск количества способов полностью заполнить область (например, шахматную доску/сетку) некоторыми фигурами (например, домино);
- поиск способа заполнить область с минимальным количеством фигур;
- поиск способа заполнения с минимальным количеством незаполненного пространства (или ячеек, в случае сетки);
- поиск способа заполнения с минимальным количеством фигур, чтобы в него нельзя было добавить ещё одну фигуру.

Задача «Паркет»

Дана сетка размером $N \times M$. Найдите количество способов заполнения сетки фигурами размером 2×1 (ни одна ячейка не должна оставаться незаполненной, а фигуры не должны перекрывать друг друга).

Построим такую динамику: *dp*[i, *mask*], где $i = 1...N$ и *mask* $= 0...2^M - 1$.

i представляет собой количество строк в текущей сетке, а *mask* – состояние последней строки текущей сетки. Если j-й бит *mask* равен 0, то соответствующая ячейка заполнена, в противном случае она не заполнена.

Очевидно, что ответом на эту задачу будет *dp*[N, 0].

Мы будем строить динамику, перебирая каждое $i = 1...N$ и каждый $mask = 0...2^M - 1$, и для каждой маски мы будем только переходить вперед, то есть будем добавлять фигуры в текущую сетку.

Листинг 1. Подсчет динамики

```
 int n, m;
vector < vector<long long> > dp;

void calc(int x = 0, int y = 0, int mask = 0, int
next_mask = 0)
{
    if (x == n)
          return;
    if (y >= m)
          dp[x + 1][next_mask] += dp[x][mask];
    else
    {
          int my_mask = 1 << y;
          if (mask & my_mask)
                calc(x, y + 1, mask, next_mask);
          else
          {
                calc(x, y + 1, mask, next_mask | my_
mask);
                if (y + 1 < m && !(mask & my_mask) &&
!(mask & (my_mask << 1)))
                      calc(x, y + 2, mask, next_mask);
          }
    }
}


int main()
{
    cin >> n >> m;

    dp.resize(n + 1, vector<long long>(1 << m));
    dp[0][0] = 1;
    for (int x = 0; x < n; ++x)
          for (int mask = 0; mask < (1 << m); ++mask)
                calc(x, 0, mask, 0);

    cout << dp[n][0];

}
```

Пример задачи
с сайта codeforces.com: https://codeforces.com/contest/342/problem/D

Разбор

В этой задаче можно было как считать количество хороших расстановок, так и из общего числа вычесть плохие расстановки. В любом из решений нужно было уметь считать динамику по маскам, состояние (*j*, *mask*), *j* – номер текущего полностью заполненного столбца, *mask* – маска того, что находится в последнем столбце (также этот прием называется динамикой по профилю).

Чтобы получить само решение задачи, можно пойти так. К клетке с кружочком приписывать с четырех сторон доминошку, после этого все три клетки закрашивать в черный цвет и считать общее количество способов. Однако чтобы не учитывать один ответ несколько раз, нужно использовать формулу включения исключения для этих четырех направлений. Это также известный прием, который позволяет правильно считать ответ для многих задач и не учитывать один ответ несколько раз.

Листинг 2. Решение задачи

```
typedef long LL;
const int N = 10005;
const int MOD = 1000000007;
char str[3][N];
int n, sx, sy;
vector<pair<pair<int, int>, pair<int, int> > > v;
int dp[N][1 << 3];
void add(int &a, int b) {
    a = (a + b) % MOD;
    a = (a + MOD) % MOD;
}
int main() {
cin >> n;
int ans = 0;
for (int i = 0; i < 3; i++) {
    scanf("%s", str[i] + 1);
    for (int j = 1; j <= n; j++)
          if (str[i][j] == 'O') {
                sx = i; sy = j;
          }
```

```
    }
    if (sx == 0) {
        if (str[sx + 1][sy] == '.' && str[sx + 2][sy] ==
'.') {
            v.push_back(make_pair(make_pair(sx + 1,
sy), make_pair(sx + 2, sy)));
        }
    }
    if (sx == 2) {
        if (str[sx - 1][sy] == '.' && str[sx - 2][sy] ==
'.') {
            v.push_back(make_pair(make_pair(sx - 1,
sy), make_pair(sx - 2, sy)));
        }
    }
    if (sy > 2) {
        if (str[sx][sy - 1] == '.' && str[sx][sy - 2] ==
'.') {
            v.push_back(make_pair(make_pair(sx, sy -
1), make_pair(sx, sy - 2)));
        }
    }
    if (sy <= n - 2) {
        if (str[sx][sy + 1] == '.' && str[sx][sy + 2] ==
'.') {
            v.push_back(make_pair(make_pair(sx, sy +
1), make_pair(sx, sy + 2)));
        }
    }
    int m = v.size();
    for (int mask = 1; mask < (1 << m); mask++) {
        int cnt = 0;
        for (int j = 0; j < m; j++) {
            if (mask & (1 << j)) {
                cnt++;
                str[v[j].first.first][v[j].first.
second] = 'X';
                str[v[j].second.first][v[j].second.
second] = 'X';
            }
        }
        memset(dp, 0, sizeof(dp));
```

```
	dp[1][0] = 1;
	for (int i = 1; i <= n; i++) {
		for (int j = 0; j < (1 << 3); j++) {
			if (dp[i][j] == 0) continue;
			int remain = 7 - j;
			for (int k = 0; k < 3; k++)
				if ((remain & (1 << k)) &&
str[k][i] != '.')
					remain -= 1 << k;
			if (remain == 0) {
				add(dp[i + 1][0], dp[i][j]);
			}
			for (int k = 0; k < 3; k++) {
				if ((1 << k) == remain) {
					if (str[k][i + 1] ==
'.') {
						add(dp[i + 1][1
<< k], dp[i][j]);
					}
				}
			}
			if (remain == 3) {
				add(dp[i + 1][0], dp[i][j]);
				if (str[0][i + 1] == '.' &&
str[1][i + 1] == '.')
					add(dp[i + 1][3], dp[i]
[j]);
			}
			if (remain == 6) {
				add(dp[i + 1][0], dp[i][j]);
				if (str[2][i + 1] == '.' &&
str[1][i + 1] == '.')
					add(dp[i + 1][6], dp[i]
[j]);
			}
			if (remain == 5) {
				if (str[2][i + 1] == '.' &&
str[0][i + 1] == '.')
					add(dp[i + 1][5], dp[i]
[j]);
			}
			if (remain == 7) {
```

```
                        if (str[2][i + 1] == '.' &&
str[0][i + 1] == '.' && str[1][i + 1] == '.')
                              add(dp[i + 1][7], dp[i]
[j]);
                        if (str[0][i + 1] == '.')
                              add(dp[i + 1][1], dp[i]
[j]);
                        if (str[2][i + 1] == '.')
                              add(dp[i + 1][4], dp[i]
[j]);
                  }
            }
      }
      if (cnt & 1) add(ans, dp[n + 1][0]);
      else add(ans, -dp[n + 1][0]);
      for (int j = 0; j < m; j++) {
            if (mask & (1 << j)) {
                  str[v[j].first.first][v[j].first.
second] = '.';
                  str[v[j].second.first][v[j].second.
second] = '.';
            }
      }
   } cout << ans << endl;
   }
```

6.3. Нахождение самой большой нулевой подматрицы

Вам дается матрица с n строками и m столбцами. Найдите самую большую подматрицу, состоящую только из нулей (подматрица – это прямоугольная область матрицы).

Для удобства элементы в матрице будем нумеровать в 0-индексации, тогда элементами матрицы будут являться $a[i][j]$, где $i = 0...n - 1$, $j = 0...m - 1$.

Шаг 1. Вспомогательная динамика

Во-первых, мы вычисляем следующую вспомогательную динамику: $d[i][j]$, ближайшая строка, имеющая 1 над $a[i][j]$. Формально говоря, $d[i][j]$ – это самый большой номер строки (от 0 до $i - 1$), в котором есть элемент, равный 1 в j-м столбце. При выполнении цикла от верхнего левого угла до нижнего правого, когда мы сто-

им в строке i, мы знаем значения из предыдущей строки, поэтому достаточно обновить только элементы со значением 1. Мы можем сохранить значения в простом массиве $d[i]$, $i = 1...m - 1$, потому что в дальнейшем алгоритме мы будем обрабатывать матрицу по одной строке за раз и нужны только значения текущей строки.

Листинг 1. Вспомогательная динамика

```
vector<int> d(m, -1);
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
            if (a[i][j] == 1) {
                    d[j] = i;
            }
    }
}
```

Шаг 2. Решение задачи

Мы можем решить эту задачу в $O(nm^2)$, перебирая строки, рассматривая все возможные левые и правые столбцы для подматрицы. Нижняя часть прямоугольника будет текущей строкой, и с помощью $d[i][j]$ мы можем найти верхнюю строку. Однако можно пойти дальше и значительно уменьшить сложность решения.

Понятно, что искомая нулевая подматрица ограничена со всех четырех сторон единицами (либо границами поля), которые мешают ей увеличиться в размерах и увеличить окончательный ответ. Поэтому мы найдем ответ, если будем действовать следующим образом: для каждой ячейки j в строке i (нижняя строка потенциальной нулевой подматрицы) мы будем иметь $d[i][j]$ в качестве верхней строки текущей нулевой подматрицы. Теперь осталось определить оптимальную левую и правую границы нулевой подматрицы, т. е. максимально отодвинуть эту подматрицу влево и вправо от j-го столбца.

Что значит сдвинуть максимально влево? Это значит найти индекс $k1$, для которого $d[i][k1] > d[i][j]$, и при этом $k1$ – ближайший слева от индекса j. Понятно, что тогда $k1 + 1$ дает номер левого столбца искомой нулевой подматрицы. Если такого индекса вообще нет, то ставим $k1 = -1$ (это значит, что мы смогли расширить текущую нулевую подматрицу влево до самой границы матрицы a).

Симметрично можно определить индекс $k2$ для правой границы: это ближайший индекс справа от j такой, что $d[i][k2] > d[i][j]$ (или m, если такого индекса нет).

Итак, индексы $k1$ и $k2$, если мы научимся их эффективно искать, дадут нам всю необходимую информацию о текущей нулевой подматрице. В частности, её площадь будет равна

$$(i - d[i][j]) \cdot (k2 - k1 - 1).$$

Как эффективно искать эти индексы $k1$ и $k2$ с фиксированными i и j? В среднем мы можем сделать это в $O(1)$.

Для достижения такой сложности мы можем использовать стек (*stack*) следующим образом. Давайте сначала научимся искать индекс $k1$ и сохранять его значение для каждого индекса j в текущей строке i в матрице $d[i][j]$. Для этого мы будем просматривать все столбцы j слева направо и хранить в стеке только те столбцы, в которых значение динамики $d[\ ][\ ]$ строго больше, чем $d[i][j]$. Понятно, что при переходе от столбца j к следующему столбцу необходимо обновить содержимое стека. Пока в верхней части стека есть неподходящий элемент (т. е. $d[\ ][\ ] <= d[i][j]$), удалять этот элемент. Легко понять, что достаточно удалить из стека только его верхнюю часть (потому что стек будет содержать увеличивающуюся последовательность столбцов d).

Значение $d1[i][j]$ для каждого j будет равно значению, лежащему в этот момент на вершине стека.

Динамика $d2[i][j]$ для нахождения индексов $k2$ считается аналогичной, только нужно просматривать столбцы справа налево.

Понятно, что поскольку в стек на каждой строке добавляется ровно m кусочков, то и больше удалений быть не может, а сумма сложностей будет линейной, поэтому конечная сложность алгоритма равна $O(nm)$.

Следует также отметить, что данный алгоритм потребляет $O(nm)$ памяти (не считая входных данных – матрицы $a[\ ][\ ]$).

Листинг 2. Решение задачи

```
int zero_matrix(vector<vector<int>> a) {
    int n = a.size();
    int m = a[0].size();
```

```
        int ans = 0;
        vector<int> d(m, -1), d1(m), d2(m);
        stack<int> st;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                if (a[i][j] == 1)
                    d[j] = i;
            }

            for (int j = 0; j < m; ++j) {
                while (!st.empty() &&  d[st.top()] <=
d[j])
                    st.pop();
                d1[j] = st.empty() ? -1 : st.top();
                st.push(j);
            }
            while (!st.empty())
                st.pop();

            for (int j = m – 1; j >= 0; --j) {
                while (!st.empty() && d[st.top()] <=
d[j])
                    st.pop();
                d2[j] = st.empty() ? m : st.top();
                st.push(j);
            }
            while (!st.empty())
                st.pop();

            for (int j = 0; j < m; ++j)
                ans = max(ans, (i – d[j]) * (d2[j] –
d1[j] – 1));
        }
        return ans;
    }
```

6.4. Игры на произвольных графах

Пусть в игру играют два игрока на произвольном графе G, т. е. текущее состояние игры – это определенная вершина. Игроки выполняют ходы по очереди и перемещаются из текущей вершины в соседнюю вершину с помощью соединительного ребра. В зависимости от игры человек, который не может двигаться, либо проиграет, либо выиграет.

Рассмотрим наиболее общий случай – случай произвольного направленного графа с циклами. Наша задача состоит в том, чтобы определить, учитывая начальное состояние, кто выиграет, если оба игрока будут играть с оптимальными стратегиями, или определить, что результатом игры будет ничья.

Мы эффективно решим эту задачу. Мы найдем решение для всех возможных начальных вершин графа в линейном времени относительно числа ребер: $O(m)$.

Мы будем называть вершину выигрышной вершиной, если игрок, начинающий в этом состоянии, выиграет игру, если он играет оптимально (независимо от того, какие повороты делает другой игрок). Аналогично мы будем называть вершину проигравшей вершиной, если игрок, начинающий с этой вершины, проиграет игру, если противник играет оптимально.

Для некоторых вершин графа мы уже заранее знаем, что это выигрышные или проигрышные вершины, а именно все вершины, у которых нет исходящих ребер.

Также у нас есть следующие правила:

- если вершина имеет исходящее ребро, которое ведет к проигрышной вершине, то сама вершина является выигрышной вершиной;
- если все исходящие ребра определенной вершины ведут к выигрышным вершинам, то сама вершина является проигрышной вершиной;
- если в какой-то момент есть еще неопределенные вершины, и ни одна из них не будет соответствовать первому или второму правилу, то каждая из этих вершин, когда она используется в качестве начальной вершины, приведет к ничьей, если оба игрока играют оптимально.

Таким образом, мы можем сразу же определить алгоритм, который работает в $O(mn)$ времени. Мы проходим через все вершины и пытаемся применить первое или второе правило, а затем повторяем.

Однако мы можем ускорить эту процедуру и снизить сложность до $O(m)$.

Мы пройдем по всем вершинам, для которых изначально знаем, являются ли они выигрышными или проигрышными состояниями. Для каждого из них мы начинаем сначала поиск в глубину. Этот поиск в глубину будет двигаться назад по обратным рёбрам. Прежде всего, он не будет заходить в вершины, которые уже определены как выигрышные или проигрышные вершины. И далее, если поиск идет от проигравшей вершины к неопределенной вершине, то мы помечаем эту вершину как выигрышную и продолжаем поиск в глубину, используя эту новую вершину. Если мы переходим от выигрышной вершины к неопределенной, то мы должны проверить, все ли ребра из этой вершины ведут к выигрышным вершинам. Мы можем выполнить эту проверку в $O(1)$, сохранив количество ребер, которые приводят к выигрышной вершине для каждой вершины. Так что если мы перейдем от выигрышной вершины к неопределенной, то увеличим счетчик и проверим, равно ли это число числу исходящих ребер. Если это так, то мы можем отметить эту вершину как проигрышную вершину и продолжить поиск из этой вершины. В противном случае мы еще не знаем, является ли эта вершина выигрышной или проигрышной, и поэтому нет смысла продолжать использовать ее в поиске в глубину.

В итоге мы посещаем каждую выигрышную и каждую проигрышную вершину ровно один раз (неопределенные вершины не посещаются), и мы проходим через каждое ребро также не более одного раза. Следовательно, сложность равна $O(m)$.

Вот реализация такого поиска в глубину. Мы предполагаем, что переменная adj_rev хранит список смежности для графа в обратном виде, т. е. вместо хранения ребра (i, j) графа мы храним (j, i). Также для каждой вершины мы предполагаем, что степень исхода уже вычислена.

Листинг 1. Реализация описанного поиска в глубину

```
 vector<vector<int>> adj_rev;
vector<bool> winning;
vector<bool> losing;
vector<bool> visited;
vector<int> degree;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj_rev[v]) {
          if (!visited[u]) {
                if (losing[v])
                      winning[u] = true;
                else if (--degree[u] == 0)
                      losing[u] = true;
                else
                      continue;
                dfs(u);
          }
    }
}
```

Пример. «Полицейский и вор»

Есть доска $m \times n$. В некоторые клетки заходить нельзя. Начальные координаты полицейского и вора известны. Одна из ячеек – это выход. Если полицейский и вор находятся на одной клетке в один момент, то полицейский выигрывает. Если вор находится на клетке с выходом (без полицейского, также находящегося на клетке), то вор выигрывает. Полицейский может ходить во всех 8 направлениях, вор только в 4 (по оси координат). И полицейский, и вор будут двигаться по очереди. Однако они также могут пропустить ход, если захотят. Первый шаг делает полицейский.

Теперь мы построим граф. Для этого мы должны формализовать правила игры. Текущее состояние игры определяется координатами полицейского P, координатами вора T, а также тем, чей это ход, назовем эту переменную P_{turn} (принимает значение истины, когда наступает очередь полицейского). Поэтому вершина графа определяется тройкой P, T, P_{turn}, и тогда граф может быть легко построен, просто следуя правилам игры.

Далее нам нужно определить, какие вершины изначально выигрывают, а какие проигрывают. Здесь есть тонкий момент. Выигрышные/проигрышные вершины зависят, помимо координат, еще и от P_{turn} – чей это ход. Если сейчас ход полицейского, то вершина выигрышная, если координаты полицейского и вора совпадают – вершина проигрышная, если она не выигрышная и вор находится на выходе. Если же сейчас ход вора, то вершина выигрышная, если вор находится на выходе, и проигрышная, если она не выигрышная и координаты полицейского и вора совпадают.

Единственный момент перед реализацией заключается в том, что вам нужно решить, хотите ли вы построить граф явно или просто построить его «на ходу». С одной стороны, построение граф в явном виде будет намного проще и меньше шансов ошибиться. С другой стороны, это увеличит объем кода, и время выполнения будет медленнее, чем если бы вы строили граф «на ходу».

Следующая реализация построит граф явно:

Листинг 2. Решение задачи

```
struct State {
    int P, T;
    bool Pstep;
};

vector<State> adj_rev[100][100][2]; // [P][T][Pstep]
bool winning[100][100][2];
bool losing[100][100][2];
bool visited[100][100][2];
int degree[100][100][2];

void dfs(State v) {
visited[v.P][v.T][v.Pstep] = true;
for (State u : adj_rev[v.P][v.T][v.Pstep]) {
    if (!visited[u.P][u.T][u.Pstep]) {
            if (losing[v.P][v.T][v.Pstep])
                    winning[u.P][u.T][u.Pstep] = true;
            else if (--degree[u.P][u.T][u.Pstep] == 0)
                    losing[u.P][u.T][u.Pstep] = true;
            else
                    continue;
```

```
            dfs(u);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> a(n);
    for (int i = 0; i < n; i++)
            cin >> a[i];

for (int P = 0; P < n*m; P++) {
for (int T = 0; T < n*m; T++) {
for (int Pstep = 0; Pstep <= 1; Pstep++) {
int Px = P / m, Py = P % m, Tx = T / m, Ty = T % m;
if (a[Px][Py] == '*' || a[Tx][Ty] == '*')
    continue;

bool& win = winning[P][T][Pstep];
bool& lose = losing[P][T][Pstep];
if (Pstep) {
    win = Px == Tx && Py == Ty;
    lose = !win && a[Tx][Ty] == 'E';
}
else {
    lose = Px == Tx && Py == Ty;
    win = !lose && a[Tx][Ty] == 'E';
}
if (win || lose)
    continue;

State st = { P,T,!Pstep };
adj_rev[P][T][Pstep].push_back(st);
st.Pstep = Pstep;
degree[P][T][Pstep]++;

const int dx[] = { -1, 0, 1, 0, -1, -1, 1, 1 };
const int dy[] = { 0, 1, 0, -1, -1, 1, -1, 1 };
for (int d = 0; d < (Pstep ? 8 : 4); d++) {
    int PPx = Px, PPy = Py, TTx = Tx, TTy = Ty;
    if (Pstep) {
```

```
                PPx += dx[d];
                PPy += dy[d];
            }
            else {
                TTx += dx[d];
                TTy += dy[d];
            }

            if (PPx >= 0 && PPx < n && PPy >= 0 && PPy < m &&
a[PPx][PPy] != '*' &&
                TTx >= 0 && TTx < n && TTy >= 0 && TTy < m
&& a[TTx][TTy] != '*')
            {
                adj_rev[PPx*m + PPy][TTx*m + TTy][!Pstep].
push_back(st);
                ++degree[P][T][Pstep];
            }
        }
        }
        }
        }

            for (int P = 0; P < n*m; P++) {
                for (int T = 0; T < n*m; T++) {
                    for  (int  Pstep  =  0;  Pstep <= 1;
Pstep++) {
                        if  ((winning[P][T][Pstep]  ||
losing[P][T][Pstep]) && !visited[P][T][Pstep])
                            dfs({ P, T, (bool)Pstep
});
                    }
                }
            }

            int P_st, T_st;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    if (a[i][j] == 'P')
                        P_st = i * m + j;
                    else if (a[i][j] == 'T')
                        T_st = i * m + j;
                }
```

```
}

if (winning[P_st][T_st][true]) {
       cout << “Police catches the thief” << endl;
}
else if (losing[P_st][T_st][true]) {
       cout << “The thief escapes” << endl;
}
else {
       cout << «Draw» << endl;
}
```

7. ПОТОКИ

7.1. Алгоритм Диница для нахождения максимального потока

Алгоритм Диница решает задачу нахождения максимального потока за $O(V^2E)$. Этот алгоритм был открыт Ефимом Диницем в 1970 году.

Сеть $G = (V, E)$ представляет собой ориентированный граф, в котором каждое ребро $(u, v) \in E$ имеет положительную пропускную способность $c(u, v) > 0$. В транспортной сети выделяются две вершины: исток s и сток t.

Потоком f в G является действительная функция f: $V \times V \to R$, удовлетворяющая условиям:

1. $f(u, v) = -f(v, u)$ (антисимметричность).
2. $|f(u, v)| <= c(u, v)$ (ограничение пропускной способности), если ребра нет, $f(u, v) = 0$.
3. $\sum_{u} f(u, v) = 0$ для всех вершин u, кроме s и t (закон сохранения потока).

Величина потока f определяется как $|f| = \sum_{u \in V} f(s, u)$.

Остаточная сеть G^R для сети G – это сеть, которая содержит два ребра для каждого ребра $(v, u) \in G$:

- (v, u) с пропускной способностью $c_{vu}{}^R = c_{vu} - f_{vu}$;
- (u, v) с пропускной способностью $c_{uv}{}^R = f_{vu}$.

Блокирующий поток для сети является таким потоком, что каждый путь из истока в сток содержит по крайней мере одно ребро, которое насыщено этим потоком. Следует понимать, что блокирующий поток не обязательно максимальный.

Слоистая сеть для сети G – это сеть, построенная следующим образом. Во-первых, для каждой вершины v вычисляется $level[v]$ – кратчайший (по количеству ребер) путь из истока до этой вершины, состоящий только из ребер с положительной пропускной способностью. В слоистой сети будут только те ребра (v, u), для которых $level[v] + 1 = level[u]$. Очевидно, что эта сеть ациклична.

Алгоритм состоит из нескольких фаз. На каждой фазе происходит построение остаточной сети G. Затем в слоистой сети находится произвольный блокирующий поток и добавляется к текущему потоку. Если величина блокирующего потока равна 0, значит, пути из истока в сток в остаточной сети нет, поток невозможно увеличить, значит, он является максимальным.

7.1.1. Доказательство корректности

Докажем, что если алгоритм завершается, он находит максимальный поток. Если алгоритм завершился, он не смог найти блокирующий поток в слоистой сети. Это означает, что слоистая сеть не имеет пути из истока в сток, а значит, и остаточная сеть не имеет пути из истока в сток. Следовательно, поток максимален.

Алгоритм заканчивается менее чем за V фаз. Чтобы доказать это, мы должны сначала доказать две леммы.

Лемма 1. Расстояния из истока до каждой вершины не уменьшаются после каждой итерации, т. е. $level_{i+1}[v] \geq level_i[v]$.

Доказательство. Зафиксируем фазу i и вершину v. Рассмотрим любой кратчайший путь P из s в v в G^R_{i+1}. Длина P равна $level_{i+1}[v]$. Обратите внимание, что G^R_{i+1} может содержать только ребра из G^R_i и ребра, обратные ребрам из G^R_i. Если P не содержит обратных ребер из G^R_i, то $level_{i+1}[v] \geq level_i[v]$, потому что P также является путем в G^R_i. Теперь предположим, что P содержит хотя бы одно обратное ребро. Пусть первое такое ребро будет (u, w). Тогда $level_{i+1}[u] \geq level_i[u]$. Ребро (u, w) не принадлежит G^R_i, поэтому на ребро (w, u) влиял блокирующий поток на предыдущей итерации. Это означает, что $level_i[u] = level_i[w] + 1$. Также $level_{i+1}[w] = level_{i+1}[u] + 1$. Из этих двух уравнений и $level_{i+1}[u] \geq level_i[u]$ мы получаем $level_{i+1}[w] \geq level_i[w] + 2$. Теперь можно использовать ту же идею для остальной части пути.

Лемма 2. $level_{i+1}[t] > level_i[t]$.

Доказательство. Из предыдущей леммы $level_{i+1}[t] \geq level_i[t]$. Предположим, что $level_{i+1}[t] = level_i[t]$. Обратите внимание, что G^R_{i+1} может содержать только ребра из G^R_i и ребра, обратные ребрам из G^R_i. Это означает, что в G^R_i есть кратчайший путь, который не был заблокирован блокирующим потоком. Это противоречие.

Из этих двух лемм следует, что выполняется меньше V фаз, поскольку $level[t]$ увеличивается, но он не может превысить $V - 1$.

7.2. Поиск блокирующего потока

Чтобы найти блокирующий поток на каждой итерации, можно просто пытаться пропускать поток с помощью DFS из s в t в слоистой сети, пока его можно пропустить. Асимптотика этого подхода составит $O(E^2)$, так как на поиск увеличивающего пути уходит $O(E)$ и их может быть до $O(E)$, так как каждый путь насыщает одно ребро в худшем случае. Чтобы сделать это быстрее, нужно удалять ребра, через которые больше нельзя проталкивать поток. Для этого нужно хранить указатель в каждой вершине, который указывает на следующее ребро, которое можно использовать. Каждый проход по ребру приведет либо к тому, что через него пройдет увеличивающий путь, либо к передвижению указателя. Длина увеличивающего пути не больше V, а их количество не превышает E, суммарное передвижение указателей во всех вершинах равно E, значит, будет совершено не более $VE + E$ действий, и асимптотика составит $O(VE)$.

Так как у алгоритма меньше V фаз, итоговая асимптотика составляет $O(V^2E)$.

7.3. Единичные сети

Единичная сеть – это сеть, в которой все ребра имеют единичную пропускную способность, и для любой вершины, кроме s и t, в эту вершину либо входит одно ребро, либо из нее выходит лишь одно. Это как раз такая сеть, которую мы строим для решения задачи максимального паросочетания с потоками.

На единичных сетях алгоритм Диница работает за $O(E\sqrt{V})$. Докажем это.

Во-первых, каждая фаза теперь работает за $O(E)$, потому что каждое ребро будет рассмотрено не более одного раза.

Во-вторых, предположим, что уже прошли $\sqrt{V}$ фаз. Тогда все увеличивающие пути с длиной $\leq \sqrt{V}$ были найдены. Пусть f – текущий поток, f' – максимальный поток. Рассмотрим их разницу $f' - f$. Это поток в G^R величиной $|f'| - |f|$ и на каждом ребре это либо 0, либо 1. Он может быть разложен на $|f'| - |f|$ путей из s в t и, возможно, циклов. Поскольку сеть является единичной, у них не может

быть общих вершин, поэтому общее число вершин $\geq (|f'| - |f|)\sqrt{V}$, но оно также $\leq V$, поэтому на следующих $\sqrt{V}$ итерациях максимальный поток обязательно будет найден.

7.4. Масштабирование потока

Алгоритм Диница является достаточно быстрым, и построить сеть, на которой количество итераций приближалось бы к его асимптотической оценке $O(V^2E)$, на практике нереально. Однако существует технология, позволяющая ускорить и без того быстрый алгоритм и добиться асимптотики $O(VE \log U)$, где $U = \max_{(u,v)\in E} c(u,v)$. Идея алгоритма заключается в том, чтобы сначала искать увеличивающие пути с большой пропускной способностью и пускать поток по ним, затем переходить к тем, где пропускная способность меньше. На первой итерации запускается алгоритм Диница, который рассматривает лишь ребра с пропускной способностью не менее $\Delta = 2^{\log_2 U}$, на следующей с пропускной способностью не менее $\Delta/2$, затем $\Delta/4$, и так до $\Delta = 1$. На каждой итерации алгоритм Диница будет отрабатывать за $O(VE)$, ведь для каждой итерации с $\Delta = 2^k$ в сети будет разрез, состоящий только из ребер, по которым нельзя пустить поток величины больше 2^{k+1}, значит, величина добавленного на этой итерации потока не превысит $E \cdot 2^{k+1}$, а так как каждый увеличивающий путь будет добавлять как минимум 2^k, их будет не более $2E$. Количество итераций масштабирования $\log U$, отсюда и оценка в $O(VE \log U)$.

7.5. Реализация алгоритма Диница

Алгоритм состоит из трех функций: главная *flow* будет искать блокирующий поток, пока он есть, на каждой итерации она выполнит построение слоистой сети, вызвав для этого *bfs*, затем будет искать в слоистой сети увеличивающие пути с помощью *dfs*.

Структуры для хранения сети и функция добавления ребра $v \to u$ с пропускной способностью сар в сеть *add_edge:*

Листинг 1. Структура для хранения сети

```
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

const long long flow_inf = 1e18;
const int n=1e3;
vector<FlowEdge> edges;
vector<vector<int>> adj(n);
int m = 0;
int s, t;
vector<int> level(n), ptr(n);
queue<int> q;

void add_edge(int v, int u, long long cap) {
    edges.emplace_back(v, u, cap);
    edges.emplace_back(u, v, 0);
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
}
```

Листинг 2. Основная функция *flow*

```
long long flow() {
    long long f = 0;
    while (true) {
        fill(level.begin(), level.end(), -1);
        level[s] = 0;
        q.push(s);
        if (!bfs())
            break;
        fill(ptr.begin(), ptr.end(), 0);
        while (long long pushed = dfs(s, flow_inf)) {
            f += pushed;
        }
    }
    return f;
}
```

Листинг 3. Функция построения слоистой сети *bfs*

```
bool bfs() {
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int id : adj[v]) {
            if (edges[id].cap - edges[id].flow < 1)
                continue;
            if (level[edges[id].u] != -1)
                continue;
```

```
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }
```

Листинг 4. Функция поиска увеличивающего пути *dfs*

```
long long dfs(int v, long long pushed) {
    if (pushed == 0)
        return 0;
    if (v == t)
        return pushed;
    for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
        int id = adj[v][cid];
        int u = edges[id].u;
        if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
            continue;
        long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
        if (tr == 0)
            continue;
        edges[id].flow += tr;
        edges[id ^ 1].flow -= tr;
        return tr;
    }
    return 0;
}
```

7.6. Пример решения задачи

Условие. Дан двудольный неориентированный граф, который может содержать кратные ребра. Нужно найти минимальное по размеру множество из k ребер графа, такое, что каждой вершине инциндентно не менее k ребер из этого множества, для всех k от 0 до минимальной степени вершины среди всех вершин.

Входные данные. В первой строке даны числа n_1 (кол-во вершин в первой доле), n_2 (кол-во вершин во второй доле) и m (кол-во ребер). $1 <= n_1, n_2 <= 2000$, $0 <= m <= 2000$. В следующих m строках описаны ребра, каждое ребро – числа u (вершина 1-й доли) и v (вершина 2-й доли).

Выходные данные. Для каждого k начиная с 0 в отдельной строке вывести сначала размер множества, а затем номера его ребер.

Пример.

Ввод:	3 3 7 1 2 2 3 1 3 3 2 3 3 2 1 2 1
Вывод:	0 3 3 7 4 6 1 3 6 7 4 5

Решение. Чтобы получить ответ для некоторого k, можно построить следующую сеть: соединить исток с каждой вершиной первой части ребром с пропускной способностью $deg_x - k$ (где deg_x — степень вершины), а затем преобразовать каждое ребро из данного графа в направленное ребро с пропускной способностью 1, а затем связать каждую вершину из второй доли со стоком ребром с емкостью $deg_x - k$. Тогда ребра, насыщенные максимальным потоком, не присутствуют в множестве (а все остальные ребра находятся в нем).

Чтобы быстро решить эту задачу, можно просто перебирать k от ее наибольшего значения до 0 и каждый раз увеличивать поток, который мы нашли на предыдущей итерации. Поскольку максимальный поток в сети не более m и будет выполнено не более m поисков, которые не увеличат поток, это решение работает за $O((n + m)^2)$.

8. ПРОДВИНУТЫЕ АЛГОРИТМЫ

8.1. Быстрое преобразование Фурье

Быстрое преобразование Фурье – это алгоритм, который позволяет нам умножать два многочлена длины n за время $O(n \log n)$, что лучше, чем тривиальное умножение в столбик, которое занимает $O(n$^2) времени. Очевидно, что умножение двух длинных чисел может быть сведено к умножению многочленов, поэтому также можно умножить два длинных числа за время $O(n \log n)$ (где n – количество цифр в числах).

Открытие быстрого преобразования Фурье (БПФ) приписывается Кули и Тьюки, которые опубликовали алгоритм в 1965 году. Но на самом деле БПФ неоднократно обнаруживался ранее, но важность этого не была осознана до изобретения современных компьютеров. Некоторые исследователи связывают открытие БПФ с Рунге и Кенигом в 1924 году. Но на самом деле Гаусс разработал такой метод уже в 1805 году, но так и не опубликовал его.

8.2. Дискретное преобразование Фурье

Возьмем многочлен степени $n - 1$:

$$A(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}.$$

Без ограничения общности мы предполагаем, что n – число коэффициентов – является степенью 2. Если n не является степенью 2, то мы просто добавляем отсутствующие члены $a_i x^i$ и устанавливаем коэффициенты a_i равными 0.

Теория комплексных чисел говорит нам, что уравнение $x^n = 1$ имеет n комплексных решений (называемых n-ми корнями единицы), и решения имеют вид $w_{n,k} = e^{\frac{2k\pi i}{n}}$, где $k = 0 \ldots n - 1$. Также эти комплексные числа имеют некоторые очень интересные свойства: например, n-й корень $w_n = w_{n,1} = e^{\frac{2\pi i}{n}}$ можно использовать для описания всех остальных n-х корней: $w_{n,k} = (w_n)^k$.

$$\begin{aligned}\text{DFT}(a_0, a_1, \ldots, a_{n-1}) &= (y_0, y_1, \ldots, y_{n-1}) \\ &= (A(w_{n,0}), A(w_{n,1}), \ldots, A(w_{n,n-1})) \\ &= (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1}))\end{aligned}$$

Дискретное преобразование Фурье (ДПФ) многочлена $A(x)$ (или вектора с коэффициентами (a_0, a_1, ..., a_{n-1}) определяется как значения многочлена в точках $x = w_{n,k}$, т. е. это вектор.

Аналогичным образом определяется обратное дискретное преобразование Фурье: обратное ДПФ многочлена (y_0, y_1, ..., y_{n-1}) это многочлен (a_0, a_1, ..., a_{n-1}).

$$\text{InverseDFT}(y_0, y_1, \ldots, y_{n-1}) = (a_0, a_1, \ldots, a_{n-1}).$$

Таким образом, если прямое ДПФ вычисляет значения многочлена в точках n-го корня, обратное ДПФ может восстановить коэффициенты многочлена, используя эти значения.

8.3. Применение ДПФ: быстрое умножение многочленов

Пусть нам даны два многочлена – A и B. Вычислим ДПФ для каждого из них: DFT(A) и DFT(B).

Что произойдет, если мы умножим эти многочлены? Очевидно, что в каждой точке значения просто умножаются, т. е.

$$(A \cdot B)(x) = A(x) \cdot B(x).$$

Это означает, что если мы умножим векторы DFT(A) и DFT(B), умножив каждый элемент одного вектора на соответствующий элемент другого вектора, то мы получим не что иное, как DFT полинома DFT($A \cdot B$):

$$\text{DFT}(A \cdot B) = \text{DFT}(A) \cdot \text{DFT}(B).$$

Наконец, применяя обратное ДПФ, получаем:

$$A \cdot B = \text{InverseDFT}(\text{DFT}(A) \cdot \text{DFT}(B)).$$

Справа – произведение двух ДПФ – парное произведение векторных элементов. Оно может быть вычислено за $O(n)$. Если мы можем вычислить ДПФ и обратное ДПФ за $O(n \log n)$, то мы можем вычислить произведение двух многочленов (и, следовательно, также двух длинных чисел) с этой же временной сложностью.

Следует отметить, что два многочлена должны иметь одинаковую степень. В противном случае два вектора результата ДПФ будут иметь разную длину. Мы можем сделать это, добавив коэффициенты со значением 0.

Также, поскольку результатом произведения двух полиномов является многочлен степени $2(n - 1)$, мы должны удвоить степени каждого многочлена (опять же заполнив нулями старшие разряды). Из вектора с n значениями мы не можем восстановить искомый многочлен с $2n - 1$ коэффициентами.

8.4. Быстрое преобразование Фурье

Быстрое преобразование Фурье — это метод, который позволяет вычислять ДПФ за $O(n \log n)$. Основная идея БПФ заключается в применении метода «разделяй и властвуй». Мы делим вектор коэффициентов многочлена на два вектора, рекурсивно вычисляем ДПФ для каждого из них и объединяем результаты, чтобы вычислить ДПФ полного многочлена.

Итак, возьмем многочлен $A(x)$ со степенью $n - 1$, где n — степень 2 и $n > 1$:

$$A(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}.$$

Разобьем его на два меньших многочлена, один из которых содержит только коэффициенты четных позиций, а другой только нечетных:

$$A_0(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-2} x^{n/2-1};$$

$$A_1(x) = a_1 x^0 + a_3 x^1 + \ldots + a_{n-1} x^{n/2-1}.$$

Легко заметить, что $A(x) = A_0(x^2) + xA_1(x^2)$.

Многочлены A_0 и A_1 имеют вдвое меньше коэффициентов, чем многочлен A. Если мы можем вычислить $\mathrm{DFT}(A)$ за линейное время, используя $\mathrm{DFT}(A_0)$ и $\mathrm{DFT}(A_1)$, тогда получится вычислить ДПФ многочлена степени n за $O(n \log n)$ с помощью метода «разделяй и властвуй».

Предположим, что мы вычислили векторы $(y_k^0)_{k=0}^{n/2-1} = \mathrm{DFT}(A_0)$ и $(y_k^1)_{k=0}^{n/2-1} = \mathrm{DFT}(A_1)$. Найдем выражение для $(y_k)_{k=0}^{n-1} = \mathrm{DFT}(A)$.

Для первых $n/2$ значений мы можем просто использовать ранее отмеченное уравнение $A(x) = A_0(x^2) + xA_1(x^2)$:

$$y_k = y_k^0 + w_n^k y_k^1,\ k = 0 \ldots n/2 - 1.$$

Однако для следующих $n/2$ значений нам нужно найти значение немного другого выражения:

$$\begin{aligned} y_{k+n/2} &= A\left(w_n^{k+n/2}\right) \\ &= A_0\left(w_n^{2k+n}\right) + w_n^{k+n/2} A_1\left(w_n^{2k+n}\right) \\ &= A_0\left(w_n^{2k} w_n^n\right) + w_n^k w_n^{n/2} A_1\left(w_n^{2k} w_n^n\right) \\ &= A_0\left(w_n^{2k}\right) - w_n^k A_1\left(w_n^{2k}\right) \\ &= y_k^0 - w_n^k y_k^1 \end{aligned}$$

Здесь мы снова использовали $A(x) = A_0(x^2) + xA_1(x^2)$ и два тождества $w_n^n = 1$ и $w_n^{n/2} = -1$.

Таким образом, получены формулы для вычисления всего вектора (y_k):

$$y_k = y_k^0 + w_n^k y_k^1,\ k = 0 \ldots n/2 - 1.$$

$$y_{k+n/2} = y_k^0 - w_n^k y_k^1,\ k = 0 \ldots n/2 - 1.$$

Эти формулы иногда называют «преобразование бабочки».

Таким образом, ДПФ можно вычислять за $O(n \log n)$.

8.5. Обратное БПФ

Пусть задан вектор $(y_0, y_1, \ldots, y_{n-1})$ – значения многочлена A степени $n - 1$ в точках $x = w_n^k$. Мы хотим восстановить коэффициенты $(a_0, a_1, \ldots, a_{n-1})$ многочлена. Эта известная задача называется интерполяцией, и для ее решения существуют общие алгоритмы. Но в этом особом случае (поскольку мы знаем значения точек в корнях из единицы) мы можем получить гораздо более простой алгоритм (который практически совпадает с прямым БПФ).

Можно записать ДПФ, согласно его определению, в виде матрицы:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Эта матрица называется матрицей Вандермонда.

Таким образом, можно вычислить вектор $(a_0, a_1, \ldots, a_{n-1})$, умножив вектор $(y_0, y_1, \ldots, y_{n-1})$ на матрицу, обратную матрице Вандермонда.

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Проверив, можно убедиться, что обратная матрица выглядит следующим образом:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}$$

Таким образом получаем формулу

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj} .$$

Сравнивая это с формулой для y_k:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{-kj},$$

можно заметить, что эти задачи почти идентичны, поэтому коэффициенты a_k можно найти с помощью того же алгоритма «разделяй и властвуй», что и с помощью прямого БПФ, только вместо w_n^k мы должны использовать w_n^{-k}, а в конце нужно разделить полученные коэффициенты на n.

Таким образом, вычисление обратного ДПФ почти такое же, как вычисление прямого ДПФ, и оно также может быть выполнено за $O(n \log n)$.

Реализация

Здесь представлена простая рекурсивная реализация БПФ и обратного БПФ в одной функции, так как разница между прямым и обратным БПФ небольшая. Для хранения комплексных чисел используется комплексный тип в С ++ STL.

Листинг. Реализация рекурсивного БПФ

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
    int n = (int) a.size();
    if (n == 1)  return;

    vector<base> a0 (n/2),  a1 (n/2);
    for (int i=0, j=0; i<n; i+=2, ++j) {
        a0[j] = a[i];
        a1[j] = a[i+1];
    }
    fft (a0, invert);
    fft (a1, invert);

    double ang = 2*PI/n * (invert ? -1 : 1);
    base w (1),  wn (cos(ang), sin(ang));
    for (int i=0; i<n/2; ++i) {
        a[i] = a0[i] + w * a1[i];
        a[i+n/2] = a0[i] - w * a1[i];
        if (invert)
            a[i] /= 2,  a[i+n/2] /= 2;
        w *= wn;
    }
}
```

Функция получает вектор коэффициентов и вычисляет ДПФ или обратное ДПФ, сохраняя результат в этом векторе. Аргумент invert показывает, следует ли вычислять прямое или обратное ДПФ. Внутри функции сначала проверяется, равна ли длина вектора единице, если это так, то ничего не нужно делать. В противном случае вектор a делится на два вектора – a_0 и a_1 и ДПФ для них вычисляется рекурсивно. Затем инициализируется значение w_n и переменная w, которая будет содержать текущую степень w_n. Затем значениярезультирующего ДПФ вычисляются с использованием приведенных выше формул.

Если установлен флаг invert, то мы заменяем w_n на w_n^{-1}, и каждое из значений результата делится на 2 (так как это будет сделано на каждом уровне рекурсии, это приведет к делению конечных значений на n).

Следует помнить, что, используя complex <double>, невозможно добиться 100%-й точности вычислений, и когда коэффициенты многочлена начнут превышать 10^{10}–10^{11}, начнут появляться ошибки в связи с округлением.

8.6. Теоретико-числовое преобразование

Теперь немного изменим задачу. Мы все еще хотим умножить два многочлена за $O(n \log n)$, но на этот раз необходимо вычислить коэффициенты по модулю некоторого простого числа p. Конечно, для этой задачи можно использовать обычное ДПФ и взять коэффициенты результата по этому модулю. Однако это может привести к ошибкам округления, особенно при работе с большими числами. Теоретико-числовое преобразование (NTT – Number Theoretic Transform) имеет преимущество в том, что оно работает только с целыми числами, и, следовательно, результат гарантированно будет правильным.

Дискретное преобразование Фурье основано на комплексных числах и n-х корнях из единицы. Чтобы эффективно его вычислить, мы используем свойства корней (например, то, что существует один корень, который генерирует все остальные корни путем возведения в степень).

Но те же свойства имеют место для n-х корней единицы в модульной арифметике. N-й корень единицы по модулю p — это такое число w_n, которое удовлетворяет

$$(w_n)^n = 1 \pmod{p},$$
$$(w_n)^k \neq 1 \pmod{p}, \quad 1 \leq k < n.$$

Другие $n - 1$ корня могут быть получены как степени корня w_n.

Чтобы применить это в алгоритме быстрого преобразования Фурье, нужен корень для некоторого n, которое является степенью 2, а также для всех меньших степеней. Можно заметить следующее интересное свойство:

$$(w_n^2)^m = w_n^n = 1 \pmod{p}, \quad \text{with } m = \frac{n}{2}$$
$$(w_n^2)^k = w_n^{2k} \neq 1 \pmod{p}, \quad 1 \leq k < m.$$

Таким образом, если w_n n-й корень из единицы, то w_n^2 — это $n/2$-й корень из единицы. И, следовательно, для всех меньших степеней двойки существуют корни требуемой степени и их можно вычислить с помощью w_n.

Для вычисления обратного ДПФ нам понадобится вычислить обратное w_n значение w_n^{-1}. Для простого модуля обратное всегда существует.

Таким образом, все свойства, которые необходимы от комплексных корней, также доступны в модульной арифметике, при условии, что есть достаточно большой модуль p, для которого существует n-й корень из единицы.

Например, можно принять следующие значения: модуль $p = 7340033$, $w_{2^{20}} = 5$. Если этого модуля недостаточно, можно найти другую пару. Мы можем использовать тот факт, что для модулей вида $p = c2^k + 1$ (p простое) всегда существует $2k$-й корень из единицы. Можно показать, что g^c такой $2k$-й корень из единицы, где g является первообразным корнем p.

Приведем нерекурсивную реализацию теоретико-числового преобразования, которая в несколько раз превосходит свой рекурсивный аналог по производительности.

Листинг. Нерекурсивная реализация теоретико-технического преобразования

```
void ntt(vector <int> &A, int w, int mod)
{
    int n=A.size();
    stw[0]=1;
    for(int a=1; a<n; a++) stw[a]=(1ll*w*stw[a-1])%mod;
    for(int a=0, b=0; a<n; a++)
    {
        if(a<b) swap(A[a], A[b]);
        int i=(n>>1);
        while(b&i)
        {
            b-=i, i>>=1;
        }
        b+=i;
    }
    int cs=n;
    for(int i=1; i<n; i<<=1)
    {
        cs>>=1;
        for(int a=0; a<n; a+=(i<<1))
        {
            int st=0;
            for(int b=a; b<a+i; b++)
            {
                int x=A[b], y=(1ll*stw[st]*A[b+i])%mod;
                A[b]=x+y;
                if(A[b]>=mod) A[b]-=mod;
                A[b+i]=x-y;
                if(A[b+i]<0) A[b+i]+=mod;
                st+=cs;
            }
        }
    }
}
```

Данная функция производит теоретико-числовое преобразование вектора *A*, размер которого должен быть степенью двойки по простому модулю *mod*, а *w* должен быть примитивным корнем степени, равной размеру вектора *A* по модулю *mod*. Эту функцию также можно использовать для вычисления обратного NTT, только вместо *w* нужно передать w^{-1}, что в модульной арифметике равно w^{mod-2}, при большом модуле это можно вычислить с помощью бинарного возведения в степень за $O(\log(mod))$. Также нужно не забыть разделить все коэффициенты на *n* (размер вектора), что в модульной арифметике эквивалентно умножению на n^{mod-2} (тоже можно посчитать за $O(\log(mod))$ в случае большого модуля).

8.7. Вычисление многочленов с большими коэффициентами с помощью нескольких NTT

Если при умножении многочленов нельзя вычислять значения коэффициентов по модулю и в то же время их значения превышают порог, при котором БПФ с комплексными числами становится неточным (если коэффициенты больше 10^{10}–10^{11}). В этом случае можно вычислить NTT по нескольким различным модулям, чтобы произведение всех модулей превосходило коэффициенты итогового многочлена. Вычислив значения по всем этим модулям, можно будет узнать значение по модулю, равному их произведению, используя китайскую теорему об остатках (так как все модули являются простыми, а следовательно, взаимопростыми). Этот модуль больше самих значений коэффициентов, а значит, значения по нему равны реальным.

Теоретико-числовое преобразование по модулю предоставляет больше возможностей, чем быстрое преобразование Фурье с использованием комплексных чисел, однако следует учитывать, что NTT в несколько раз медленнее БПФ из-за большого числа тяжелых операций взятия по модулю, поэтому предпочтительно использовать БПФ, когда коэффициенты небольшие.

8.8. Примеры решения задач

Условие

Дан массив a из n целых чисел. Необходимо найти, сколько в этом массиве отрезков, в которых ровно k чисел меньше числа x, для всех k от 0 до n.

Входные данные

В первой строке через пробел 2 числа n и x ($1 <= n <= 2 \cdot 10^5$, $-10^9 <= x <= 10^9$).

Во второй строке массив a из n чисел ($-10^9 <= a_i <= 10^9$).

Выходные данные

$n + 1$ – число, где i-е по порядку число – количество отрезков при $k = i$.

Пример.

Ввод:	2 6 –5 9
Вывод:	1 2 0

– 1 отрезок, на котором 0 чисел меньше 6: [9].

– 2 отрезка, на которых 1 число меньше 6: [–5 9], [–5].

– отрезков, на которых 2 числа меньше 6, нет.

Решение

Сначала найдем количество чисел меньше x на каждом префиксе (включая пустой префикс). Получим массив s с этими значениями. Можно заметить, что для каждого i, j, где $i < j$, что $s[i] \le s[j]$, а если $s[i] < s[j]$, то $i < j$.

Давайте посчитаем массив r таким образом, что $r[i]$ – это число вхождений i в s. Тогда ответ для каждого d от 1 до k $ans[d]$ равен $\sum_{i,j,i-j=d} r[i] \cdot r[j]$.

Создадим массив v, $v[i] = r[n - i]$.

$$p[n+d] = \sum_{i,h,i+h=n+d} r[i] \cdot v[h] =$$

$$\sum_{i,j,i+n-j=n+d} r[i] \cdot r[j] =$$

$$\sum_{i,j,i-j=d} r[i] \cdot r[j] = ans[d]$$

Если $p = r \cdot v$, то умножение r и v может быть выполнено с помощью БПФ. Рекомендуется использовать расширенные типы с плавающей запятой, потому что значения в p могут достигать 10^{10}. Также можно использовать NTT по двум модулям и восстановить ответ, используя китайскую теорему об остатках.

Случай $k = 0$ следует обработать отдельно, нужно разбить массив на отрезки, удалив все числа меньше x, для каждого отрезка посчитать количество его подотрезков и добавить к $ans[0]$. Асимптотика этого решения составит $O(n \log n)$.

Условие

Необходимо выбрать натуральное число $m > 0$ и m натуральных чисел $b_1, b_2, ..., b_m$ так, чтобы $\prod_{i=1}^{m} b_i \ge n$ и $\sum_{i=1}^{m} b_i$ была минимальной.

Входные данные

В единственной строке находится число n ($1 <= n <= 1,5 \cdot 10^6$).

Выходные данные

Минимально возможное значение $\sum_{i=1}^{m} b_i$.

Пример.

Ввод:	36
Вывод:	10

$m = 4$, $b = [3, 3, 2, 2]$, $\prod_{i=1}^{m} b_i = 36 \geq 36$, $\sum_{i=1}^{m} b_i = 10$

Ввод:	37
Вывод:	11

$m = 4$, $b = [3, 3, 3, 2]$, $\prod_{i=1}^{m} b_i = 54 \geq 37$, $\sum_{i=1}^{m} b_i = 11$

Решение

Предположим, что в оптимальном решении есть $x \geq 4$ среди b_i. Всегда будет лучше разделить его на 2 и $(x - 2)$: сумма остается неизменной, а произведение увеличивается (или остается прежним). Таким образом, мы будем использовать только 2 и 3 в качестве b_i. Если среди b_i есть хотя бы три 2, мы можем заменить их двумя 3: сумма остается неизменной, произведение увеличивается.

Итак, оптимальное решение выглядит так: ноль, одна или две 2 и несколько 3.

Пока давайте допустим, что мы попробуем все три варианта количества двоек. Теперь задача выглядит как «найдите log3 n».

Мы можем очень точно оценить ответ. Допустим, длина n в десятичной записи равна L. Тогда log3 n очень близок к $L = \frac{\log 10}{\log 3}$, разница не превысит 3. Поэтому легко вычислить число p такое, что $3^p < n/4 < n < 3^{p+6}$.

Если мы вычислим 3^p, придется немного откорректировать, умножив на 3 несколько раз. Умножение на 3 может быть выполнено за линейное время, сравнение двух чисел также за линейное время. Давайте теперь вспомним, что мы перепробовали все возможности

для количества чисел 2. Сделаем это не заранее, а сейчас, потому что теперь каждый вариант можно проверять за линейное время.

Для вычисления 3^p будем использовать двоичное возведение в степень с БПФ. Если длина результата равна L, то время выполнения будет $O(L \log L + (L/2)\log L + (L/4)\log L + \ldots) = O(L \log L)$.

Чтобы сократить время выполнения, можно хранить числа по основанию 1000, а не 10. Это уменьшит длину числа в 3 раза, и числа, которые мы получим в БПФ, будут не более $5 \cdot 10^{11}$, что достаточно для избежания проблем с точностью.

В итоге время выполнения примерно эквивалентно 4 вызовам БПФ размером 2^{19}, что не так уж и много. Общая сложность решения – $O(L \log L)$.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Лааксонен, А. Олимпиадное программирование / Антти Лааксонен ; пер. с англ. А.А. Слинкин. – Москва : ДМК Пресс, 2018. – 300 с.
2. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – Изд. 3-е. – Вильямс, 2019. – 1328 с.
3. Хайнеман, Д. Алгоритмы. Справочник с примерами на C, C++, Java и Python / Джордж Хайнеман, Гари Поллис, Стэнли Селков. – Вильямс, 2017. – 432 с.
4. Бабенко, М.А. Введение в теорию алгоритмов и структур данных / М.А. Бабенко, М.В. Левин. – Москва : Изд-во МЦНМО, 2012. – 144 с.