Март 28

Задача A. Безопасное расстояние
Задача B. Произведение матриц
Задача C. Частые значения - 2
Задача D. Фенечка
Задача E. Мутация
Задача F. Башня карликов
Задача G. Лексикография
Задача H. Ожидаемая минимальная степень
Задача I. Лемурьи вечеринки – базовая
Задача J. Быстрый побег

10724. Безопасное расстояние

Прошлый год был трудным, вирус распространился среди населения. К счастью, Алиса знает, что один из ключей к здоровью – держаться на безопасном расстоянии от других людей.

Алиса в настоящее время находится в закрытой комнате, представленной в 2D-плоскости, шириной x и высотой y. В комнате есть n человек, и нам известны их координаты (x_i, y_i).

Будем рассматривать Алису и этих n людей как точки в 2D плоскости. Начальное положение Алисы (0, 0), и она хочет перейти к выходу в позиции (x, y). Она может свободно перемещаться в любом направлении внутри комнаты, но не может выходить за пределы комнаты.

Найдите максимальное расстояние, на котором Алиса может держаться от других людей при перемещении от (0, 0) до (x, y).

Вход. Вход начинается с одной строки, содержащей два целых числа x и y ($1 \le x, y \le 10^6$), где x – ширина, y – высота комнаты. Вторая строка содержит количество людей n ($1 \le n \le 1000$) в комнате. Каждая из следующих n строк состоит из двух действительных чисел x_i и y_i ($0 \le x_i \le x$, $0 \le y_i \le y$) – координат i-го человека в комнате.

Выход. Выведите одно действительное число d – максимальное безопасное расстояние. Допускается аддитивная или мультипликативная ошибка 10^{-5}.

Пример входа

```
8 6
3
3 1
3 5.5
6.5 1.5
```

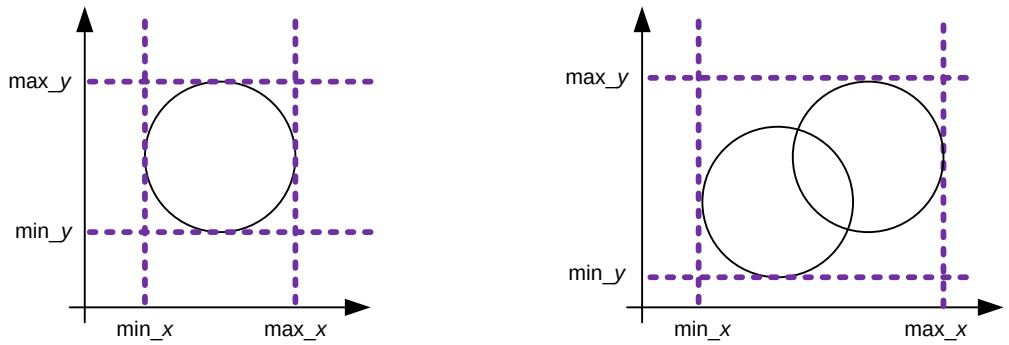
Пример выхода

```
2.250000
```

Найдем ответ на вопрос: существует ли путь у Алисы если безопасное расстояние равно d? Если мы ответим на этот вопрос, то наибольшее возможное d будем искать бинарным поиском.

Представим человека в виде круга радиуса d. Нам следует найти путь от (0, 0) до (x, y), не проходя по кругам. Для каждого круга будем хранить границы его абсцисс и ординат min_x, max_x, min_y, max_y. Прямоугольник с противоположными вершинами (min_x, max_x) – (min_y, max_y) назовем ***окружающим***.

Если круги пересекаются, то будем их объединять используя ***структуру непересекающихся множеств***. При объединении кругов будем пересчитывать границы абсцисс и ординат.



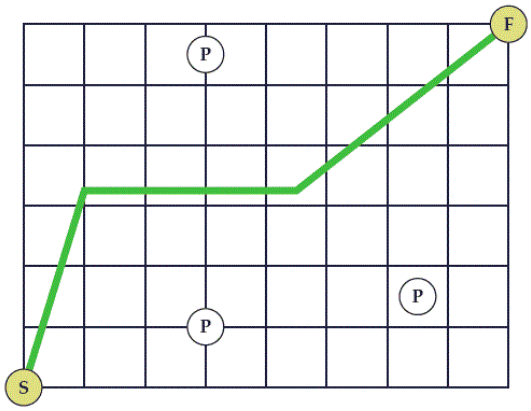
Рассмотрим всех людей. Объединим все пересекающиеся круги в компоненты связности. Теперь следует определить, существует ли путь из (0, 0) в (x, y). Для этого необходимо выполнение следующих условий:

- Никакой окружающий прямоугольник не содержит точку (0, 0);
- Никакой окружающий прямоугольник не содержит точку (x, y);
- Не существует окружающего прямоугольника, для которого min_x < 0 и max_x > x;
- Не существует окружающего прямоугольника, для которого min_y < 0 и max_y > y;

Достаточно рассматривать только те окружающие прямоугольники, которые соответствуют кругам - представителям множеств.

Пример

Алиса может держаться на расстоянии 2.25 от любого другого человека, и это лучшее, на что она способна. На картинке ниже показан возможный путь.



Реализация алгоритма

Объявим константы.

```
const int MAX_N = 5000;
const double EPS = 1e-8;
```

Объявим структуру **Point** – точка (человек). Определим функцию ***Distance*** – евклидово расстояние между людьми. Объявим массив точек people.

```
struct Point
{
  double x, y;
  double Distance(Point& other)
  {
    return sqrt((x - other.x) * (x - other.x) +
                (y - other.y) * (y - other.y));
  }
} people[MAX_N + 5];
```

Объявим структуру кругов **Circlesets**. В массиве *p* будем поддерживать систему непересекающихся множеств. Массив *rank* используется для ранговой эвристики на основе глубины деревьев.

```
struct CircleSets
{
  double min_x[MAX_N], max_x[MAX_N], min_y[MAX_N], max_y[MAX_N];
  int p[MAX_N], rank[MAX_N];
```

Функция ***Init*** инициализирует данные *n* кругов радиуса *radius*. Для каждого круга вычисляем координаты противоположных вершин окружающего прямоугольника (min_*x*, max_*x*) – (min_*y*, max_*y*).

```
  void Init(double radius)
  {
    for (int i = 0; i < n; ++i)
    {
      min_x[i] = people[i].x - radius;
      max_x[i] = people[i].x + radius;
      min_y[i] = people[i].y - radius;
      max_y[i] = people[i].y + radius;
```

i-ое множество состоит из одной *i*-ой вершины.

```
      p[i] = i;
```

Глубина дерева с одной вершиной считается равной нулю.

```
      rank[i] = 0;
    }
  }
```

Функция ***Repr*** возвращает представителя множества, в котором находится элемент *n*. Поддерживаем эвристику сжатия пути.

```
int Repr(int n)
{
  if (n == p[n]) return n;
  return p[n] = Repr(p[n]);
}
```

Функция ***Join*** объединяет элементы *i* и *j*.

```
void Join(int i, int j)
{
  i = Repr(i);
  j = Repr(j);
  if (i == j) return;
```

Поддерживаем ранговую эвристику. К большему множеству присоединяем меньшее.

```
  if (rank[i] > rank[j]) swap(i, j);
  p[i] = j;
  if (rank[i] == rank[j]) rank[j]++;
```

При объединении кругов *i* и *j* пересчитываем границы *j*-го окружающего прямоугольника, так как теперь он становится представителем множества.

```
  min_x[j] = min(min_x[j], min_x[i]);
  max_x[j] = max(max_x[j], max_x[i]);
  min_y[j] = min(min_y[j], min_y[i]);
  max_y[j] = max(max_y[j], max_y[i]);
}
```

Функция ***HasPath*** проверяет, существует ли путь из (0, 0) в (*x*, *y*).

```
bool HasPath()
{
  for (int i = 0; i < n; i++)
  {
```

Рассматриваем только те круги, которые являются представителями в своих множествах.

```
      if (p[i] != i) continue;
```

Проверяем четыре условия. Если хотя бы одно из них выполняется, то путь из (0, 0) в (*x*, *y*) будет заблокирован.

```
      if (min_x[i] < 0 && max_x[i] > x)
        return false; // Horizontal
      if (min_y[i] < 0 && max_y[i] > y)
        return false; // Vertical
      if (min_x[i] < 0 && min_y[i] < 0)
        return false; // no path from (0, 0)
      if (max_x[i] > x && max_y[i] > y)
        return false; // no path from (x, y)
      }
      return true;
    }
  } circle_sets;
```

Функция ***IsSafe*** проверяет, является ли безопасным путь для расстояния *radius*.

```
  bool IsSafe(double radius)
  {
```

Инициализируем множество кругов.

```
    circle_sets.Init(radius);
```

Перебираем все пары кругов.

```
    for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
```

Если круги *i* и *j* пересекаются (расстояние между людьми *i* и *j* меньше 2 * *radius*), то объединяем эти круги в одну компоненту связности.

```
      if (people[i].Distance(people[j]) < 2 * radius)
        circle_sets.Join(i, j);
    return circle_sets.HasPath();
  }
```

Функция ***MaxSafeDistance*** ищет наибольшее безопасное расстояние между людьми, для которого существует путь у Алисы.

```
  double MaxSafeDistance()
  {
```

Ответ ищем бинарным поиском на отрезке [*left*; *right*] = [0; min(*x*, *y*)].

```
    double left = 0.0, right = min(x, y);
```

```
  while (right - left > EPS)
  {
    double mid = (left + right) / 2;
    if (IsSafe(mid)) left = mid;
    else right = mid;
  }
  return left;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d %d", &x, &y, &n);
for (i = 0; i < n; i++)
  scanf("%lf %lf", &people[i].x, &people[i].y);
```

Вычисляем и выводим ответ.

```
printf("%lf\n", MaxSafeDistance());
```

1322. Произведение матриц

Даны три квадратные матрицы A, B, C, каждая из которых имеет размер $n \times n$. Необходимо проверить равенство: A × B = C.

Вход. Каждый тест начинается значением n ($n \le 500$). Далее следуют три матрицы A, B, C, каждая из которых представляется n строками, содержащих в точности n чисел. Элементы матриц A и B по модулю не превышают 1000. Последний тест содержит $n = 0$ и не обрабатывается. Например, в первом тесте следует проверить равенство

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 5 & 9 \\ 11 & 21 \end{pmatrix}$$

Выход. Для каждого теста в отдельной строке вывести “YES” или “NO” в зависимости от того, имеет ли место равенство A × B = C или нет.

Пример входа	**Пример выхода**
2 1 2 3 4 1 3 2 3 5 9 11 21 2 1 2 3 4 1 3	YES NO

```
2 3
5 9
10 21
0
```

Тесты были подобраны так, что простое умножение матриц с временной оценкой $O(n^3)$ дает ***Time Limit***. Задачу следует решать вероятностным методом.

Сгенерируем вектор r длины n из 0 и 1. Вычислим вектора ABr = A(Br) и Cr за $O(n^2)$. Если они равны, то с вероятностью 50% можно сказать, что A × B = C. Проведем такое тестирование, например, 10 раз. Тогда с вероятностью $1 - 1/2^{10}$ получим правильный ответ.

ABr вычисляется так (умножение матриц и векторов ассоциативно): сначала умножаем матрицу B на вектор r за $O(n^2)$, получаем вектор. Потом множим матрицу A на этот вектор опять за время $O(n^2)$.

Реализация алгоритма

Объявим рабочие массивы.

```
#define MAX 501
int a[MAX][MAX], b[MAX][MAX], c[MAX][MAX], r[MAX], r1[MAX], r2[MAX];
```

Читаем входные данные для каждого теста. Последний тест содержит $n = 0$ и не обрабатывается.

```
while (scanf("%d", &n), n)
{
  for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) scanf("%d", &a[i][j]);

  for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) scanf("%d", &b[i][j]);

  for (i = 0; i < n; i++) for (j = 0; j < n; j++)
    scanf("%d", &c[i][j]);
```

Для каждого теста проверку равенства ABr = Cr производим 10 раз.

```
  for (cnt = 0; cnt < 10; cnt++)
  {
```

Заполняем линейный массив r нулями и единицами произвольным образом.

```
    for (flag = i = 0; i < n; i++) r[i] = rand() % 2;
```

Выполняем умножение r_1 = Br.

```
    for (i = 0; i < n; i++)
    for (r1[i] = k = 0; k < n; k++) r1[i] += b[i][k] * r[k];
```

Выполняем умножение $r_2 = Ar_1 = ABr$.

```
for (i = 0; i < n; i++)
for (r2[i] = k = 0; k < n; k++) r2[i] += a[i][k] * r1[k];
```

Выполняем умножение $r_1 = Cr$.

```
for (i = 0; i < n; i++)
for (r1[i] = k = 0; k < n; k++) r1[i] += c[i][k] * r[k];
```

Сравниваем массивы r_2 и r_1. То есть проверяем равенство $ABr = Cr$. Если $ABr \neq Cr$, то устанавливаем $flag = 1$.

```
    for (i = 0; i < n; i++)
      if (r1[i] != r2[i])
      {
        flag = 1; break;
      }
    if (flag) break;
  }
```

Для текущего теста выводим ответ.

```
  if (flag) printf("NO\n"); else printf("YES\n");
}
```

4081. Частые значения - 2

Задана последовательность n целых чисел a_1 , a_2 , ... , a_n в неубывающем порядке. Вам также заданы несколько запросов, состоящих из индексов i и j ($1 \le i \le j \le n$). Для каждого запроса определите число, которое чаще всего встречается среди a_i , ... , a_j.

Вход. Состоит из нескольких тестов. Каждый тест начинается со строки, содержащей два целых числа n и q ($1 \le n, q \le 500000$). Следующая строка содержит n целых чисел a_1 , ... , a_n ($-500000 \le a_i \le 500000$, для каждого $i \in \{1, ..., n\}$), разделенных пробелом. Считайте, что для каждого $i \in \{1, ..., n - 1\}$: $a_i \le a_{i+1}$. Каждая из следующих q строк содержит один запрос, состоящий из двух целых значений i и j ($1 \le i \le j \le n$) – границы индексов запроса.

За последним тестом следует строка, содержащая один 0.

Выход. Для каждого запроса выведите одно целое число: количество вхождений чаще всего встречаемого числа в заданном интервале.

Пример входа	**Пример выхода**
10 3 -1 -1 1 1 1 1 3 10 10 10 2 3 1 10 5 10 0	1 4 3

Данная задача отличается от 3838 (частые значения) увеличением ограничений по n и q, а также набором тестов, на котором решение с использованием дерева отрезков даст Time Limit.

Поскольку входная последовательность задается сразу и никогда не меняется (отсутствуют операции update), то задачу можно решить при помощи структуры Range Maximum Query. Ее преимущество состоит в том, что запрос на отрезке выполняется за константное время, а не за логарифмическое как на дереве отрезков. Итого:

- время работы решения при помощи дерева отрезков: $O(n + q \log_2 n)$.
- время работы решения при помощи RMQ: $O(n \log_2 n + q)$.

Отметим также, что используемая память составит:

- $O(n)$ для дерева отрезков
- $O(n \log_2 n)$ для RMQ

Чтобы свести указанную задачу к RMQ, создадим по входной последовательности новый массив b: $b[i]$ содержит количество повторов числа a_i вплоть до индекса i. Например, если a = (3, 3, 5, 9, 9, 9, 9, 10, 10), то b = (1, 2, 1, 1, 2, 3, 4, 1, 2).

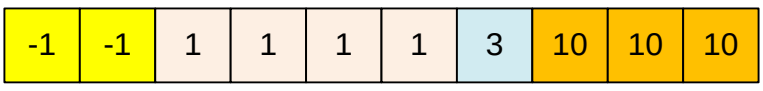
Ответ на требуемый запрос $q(i, j)$ совершим следующим образом:

- если $a_i = a_j$, то ответом будет значение $j - i + 1$;
- иначе ищем самый правый индекс k, для которого $a_i = a_k$. После чего находим максимальное значение на отрезке $b[k + 1, \ldots, j]$, то есть $RMQ_{k+1,j}$ (для удобства вычислений Range Maximum Query будет возвращать не индекс с максимальным элементом, а сам максимальный элемент). Тогда ответом на запрос будет

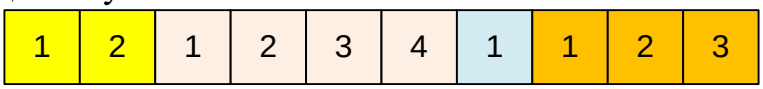
$$\max (k - i + 1, RMQ_{k+1,j}).$$

Пример

Входной массив a:



Соответствующий ему массив b:



Рассмотрим запрос $q(4, 9)$. Тогда $k = 6$ ($a_4 = a_6$, $a_4 \neq a_7$). Ответ равен

$$\max (k - i + 1, RMQ_{k+1,j}) = \max (6 - 4 + 1, RMQ_{7,9}) = \max (3, 2) = 3$$

Рассмотрим запрос $q(2, 5)$. Тогда $k = 2$ ($a_2 = a_2$, $a_2 \neq a_3$). Ответ равен

$$\max(k - i + 1, RMQ_{k+1,j}) = \max(2 - 2 + 1, RMQ_{3,5}) = \max(1, 3) = 3$$

Реализация алгоритма

Объявим необходимые константы. Входная последовательность хранится в массиве a. Объявим вспомогательный массив b. Массив mas будет использоваться под структуру RMQ: mas[i][j] содержит максимальный элемент на отрезке [i, …, $i + 2^j$]. Массив mas хранит не индексы последовательности, а сами максимальные значения на отрезках. Таким образом мы избавимся от большого числа операций индексации.

```
#define MAX 500010
#define LOGMAX 19
int a[MAX], b[MAX];
int mas[MAX][LOGMAX];
```

По массиву b строим массив mas, для которого mas[i][j] = max(b_i, b_{i+1}, …, $b_{i+2\wedge j}$).

```
void Build_RMQ_Array(int *b)
{
  int i, j;
  for (i = 1; i <= n; i++) mas[i][0] = b[i];
  for (j = 1; 1 << j <= n; j++)
    for (i = 1; i + (1 << j) - 1 <= n; i++)
      mas[i][j] = max(mas[i][j - 1], mas[i + (1 << (j - 1))][j - 1]);
}
```

Функция RMQ возвращает максимум на отрезке (b_i, b_{i+1}, …, b_j) за O(1).

```
int RMQ(int i, int j)
{
  int k = 0;
  while ((1 << (k + 1)) <= j - i + 1) k++;
  return max(mas[i][k],mas[j - (1<<k) + 1][k]);
}
```

Основная часть программы. Читаем входной массив a. Одновременно строим по нему массив b.

```
while(scanf("%d %d",&n,&q), n)
{
  scanf("%d",&a[1]); b[1] = 1;
  for(i = 2; i <= n; i++)
  {
    scanf("%d",&a[i]);
    if (a[i] == a[i-1]) b[i] = b[i-1] + 1; else b[i] = 1;
  }
```

Строим структуру данных RMQ.

```
  Build_RMQ_Array(b);
```

Последовательно обрабатываем запросы.

```
  for(i = 0; i < q; i++)
  {
    scanf("%d %d",&Left,&Right);
    if (a[Left] == a[Right]) printf("%d\n",Right - Left + 1);
    else
    {
```

LeftEnd равно самому правому индексу, для которого $a_i = a_{LeftEnd}$.

```
      int LeftEnd =
        (int)(upper_bound(a+Left,a+Right,a[Left]) - a) - 1;
      res = max(LeftEnd - Left + 1, RMQ(LeftEnd + 1, Right));
      printf("%d\n",res);
    }
  }
}
```

5274. Фенечка

Саша находится в процессе творческого поиска. Она хочет сплести ещё одну фенечку, но испытывает сложности при выборе цветов. Сейчас все n ниток, которые она планирует использовать для плетения, выложены в ряд. В процессе размышления Саша время от времени заменяет нитку одного цвета ниткой другого, а также для проверки того, что узор получается тем, который подразумевается, проверяет, что некоторые последовательности цветов ниток равны.

Напишите программу, которая автоматизирует эти проверки.

Вход. В первой строке записаны два целых числа n и k – количество ниток в фенечке и запросов к программе, соответственно ($1 \le n, k \le 100000$). Во второй строке записана строка из n символов – цвета ниток в начальном состоянии. Каждый цвет обозначается строчной или прописной буквой латинского алфавита или цифрой. В следующих k строках заданы запросы двух видов:

"* *i c*" – заменить нитку с номером *i* на нитку цвета *c*,

"? *i j len*" – проверить, равны ли последовательности цветов ниток, начинающиеся в позициях *i* и *j* и имеющие длину *len*.

Выход. Для каждого запроса второго вида выведите "+", если последовательности равны, или "-" в противном случае.

Пример входа	**Пример выхода**
7 4 abacaba ? 1 5 3 * 6 c	+-+

```
? 2 6 2
? 3 5 3
```

Пусть в строке S = $s_1 s_2 \ldots s_n$ находится текущее состояние фенечки. Построим по ней полиномиальный хеш, равный H(S) = $s_1 * p + s_2 * p^2 + \ldots + s_n * p^n$, где p – некоторое простое число, большее количества букв. Возьмем, например $p = 37$.

Пусть ячейки массива m содержат слагаемые хеша так что $m[i] = s_i * p^i$. Массив m будем моделировать при помощи дерева Фенвика. Изначально обнулим массив m, после чего совершим прибавление Inc(i, $s_i * p^i$). Для удобства буквы закодируем начиная с 1: букве ‘*a*’ поставим в сооветствие код 1, букве ‘*b*’ код 2, и так далее.

Пусть функция Sum(i) возвращает сумму $s_1 * p + s_2 * p^2 + \ldots + s_i * p^i$. Тогда последовательность цветов ниток, начинающихся в позиции i длины *len*, имеет хеш $s_i * p^i + s_{i+1} * p^{i+1} + \ldots + s_{i+len-1} * p^{i+len-1}$, равный Sum($i + len - 1$) – Sum($i - 1$). А последовательность цветов ниток, начинающихся в позиции j длины *len*, имеет хеш $s_j * p^j + s_{j+1} * p^{j+1} + \ldots + s_{j+len-1} * p^{j+len-1}$, равный Sum($j + len - 1$) – Sum($j - 1$). Подстроки $s_i s_{i+1} \ldots s_{i+len-1}$ и $s_j s_{j+1} \ldots s_{j+len-1}$ равны, когда их хеши различаются в p^{j-i} раз (пусть $i < j$ для определенности). Действительно:

$$(\text{Sum}(i + len - 1) - \text{Sum}(i - 1)) * p^{j-i} = s_i * p^j + s_{i+1} * p^{j+1} + \ldots + s_{i+len-1} * p^{j+len-1} =$$

(учитывая что $s_i = s_j$, $s_{i+1} = s_{j+1}$, …, $s_{i+len-1} = s_{j+len-1}$) $= s_j * p^j + s_{j+1} * p^{j+1} + \ldots +$

$$s_{j+len-1} * p^{j+len-1} =$$

$$= \text{Sum}(j + len - 1) - \text{Sum}(j - 1)$$

Все вычисления проводим в типе данных long long, не обращая внимание на переполнение.

Реализация алгоритма

Определим рабочие массивы.

```
#define MAX 100010
long long p = 37;
long long Pow[MAX], Fenwick[MAX];
char s[MAX];
```

Прибавление m_i += *delta*.

```
void Inc(int i, long long delta)
{
  for (; i <= n; i = (i | (i+1)))
    Fenwick[i] += delta;
}
```

Вычисление суммы $m_0 + m_1 + ... + m_i$.

```
long long Sum(int i)
{
  long long result = 0;
  for (; i >= 0; i = (i & (i + 1)) - 1)
    result += Fenwick[i];
  return result;
}
```

Основная часть программы. Читаем входные данные. Строку s читаем с позиции 1, так как индексы в запросах начинаются с 1.

```
scanf("%d %d\n",&n,&q);
gets(s+1);
```

Вычисляем степени числа p = 37: Pow[i] = p^i. На переполнение не обращаем внимание.

```
Pow[0] = 1;
for(i = 1; i <= n; i++)
  Pow[i] = p * Pow[i - 1];
```

Совершаем присваивание m[i] = s_i * p^i, массив m моделируем деревом Фенвика.

```
for(i = 1; i <= n; i++)
  Inc(i, Pow[i] * (s[i] - 'a' + 1));
```

Обрабатываем q запросов.

```
while(q--)
{
  scanf("%c",&c);
  if(c == '?')
  {
    scanf("%d %d %d\n",&l,&r,&len);
```

Пересчитаем позиции запросов так чтобы $l < r$.

```
    if(l > r) swap(l, r);
```

Выводим ответ на запрос о равенстве подстрок.

```
    printf((Sum(l + len - 1) - Sum(l - 1)) * Pow[r - l] ==
           Sum(r + len - 1) - Sum(r - 1) ? "+" : "-");
  }
  else
  {
    scanf("%d %c\n",&l,&c);
```

На данный момент m[l] = s_l * p^l. Следует произвести присваивание s[l] = c или m[l] = c * p^l. На дереве Фенвика промоделируем прибавление m[l] = m[l] + c * p^l – s_l * p^l.

```
    Inc(l, (c - s[l]) * Pow[l]);
    s[l] = c;
  }
}
printf("\n");
```

4973. Мутация

Ученые-генетики планеты Олимпия опять проводят эксперименты с ДНК примитивных организмов. Геном организма – это последовательность генов, каждый из которых можно закодировать одним натуральным числом. Гены, которые кодируются одними и теми же числами, считаются одинаковыми, и наоборот, гены, которые кодируются разными числами, считаются разными.

Ученые уже вывели некоторый примитивный организм и хотят модифицировать ему геном таким образом, чтобы получить идеальный организм. Они считают, что в дальнейшем это поможет найти лекарства от многих болезней.

Организм считается идеальным, если любые два одинаковых гена либо стоят на соседних позициях в геноме, либо между ними есть хотя бы один такой же, как они, ген.

За одну операцию ученые могут выбрать и удалить один или несколько одинаковых генов из генома организма, после чего вставить их обратно в геном, но, возможно, на другие позиции. Поскольку каждая такая операция ослабляет организм, ученые хотят достичь своей цели, выполнив при этом как можно меньше операций.

Напишите программу, которая по заданному представлению генома определит наименьшее количество операций, необходимое для получения идеального организма.

Вход. Первая строка содержит количество генов n ($1 \le n \le 10^5$) в геноме примитивного организма. В следующей строке записано n натуральных чисел, каждое из которых не превышает n – последовательность генов в геноме.

Выход. Выведите наименьшее количество операций, за которое ученые смогут получить идеальный организм.

Пример входа	**Пример выхода**
9 1 2 1 3 1 3 2 4 5	2

Перестановку генов следует произвести таким образом, чтобы одинаковые гены стояли рядом. При этом следует минимизировать количество произведенных операций.

Каждому натуральному числу a_i поставим в соответствие отрезок $[s_i; e_i]$, где s_i – позиция, в которой a_i встречается впервые, а e_i – позиция, в которой a_i встречается в последний раз.

Пусть *res* – наибольшее возможное количество непересекающихся отрезков. Это значение находим с помощью алгоритма “выбора заявок”. Это означает, что каждый из остальных отрезков пересекается хотя бы с одним из выбранных. И над генами, им отвечающим, необходимо провести описанную процедуру.

Если k – число всех построенных отрезков, то ответом на задачу будет значение $k - res$.

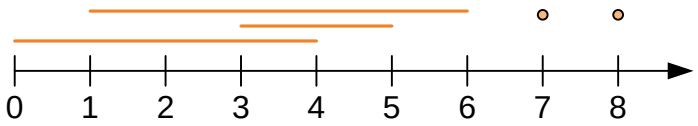
Пример

Для приведенного примера отрезки будут иметь вид (нумеровать позиции будем с нуля):

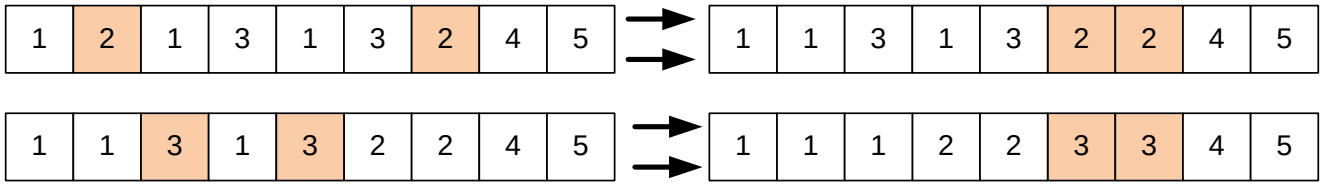
- для 1: [0; 4];
- для 2: [1; 6];
- для 3: [3; 5];
- для 4: [7; 7];
- для 5: [8; 8];

Отсортируем отрезки по координате конца:

[0; 4], [3; 5], [1; 6], [7; 7], [8; 8]



Максимальное количество непересекающихся отрезков равно 3. Например, можно выбрать отрезки [0; 4], [7; 7], [8; 8]. Над генами, отвечающим остальным отрезкам, следует проводить описанную в условии задачи процедуру. Таковыми генами являются 2 и 3. Две перестановки можно, например, произвести следующим образом:



Реализация алгоритма

Пусть MAX – максимальное возможное количество геномов во входной последовательности.

```
#define MAX 100001
```

Вектор пар v хранит отрезки [s_i; e_i].

```
vector<pair<int, int> > v;
int s[MAX], e[MAX];
```

Читаем входные данные. Инициализируем s[i] = -1, e[i] = -1.

```
scanf("%d", &n);
memset(s, -1, sizeof(s));
memset(e, -1, sizeof(e));

for (i = 0; i < n; i++)
{
  scanf("%d", &x);
```

Если число *x* встречается впервые, то его позицию во входной последовательности заносим в s[*x*].

```
  if (s[x] == -1) s[x] = i;
```

Если число *x* встречается не первый раз, то его позицию заносим в e[*x*], считая что *x* встречается в последний раз.

```
  if (i > e[x]) e[x] = i;
}
```

Все построенные отрезки заносим в вектор v.

```
for (i = 1; i <= n; i++) // numbers 1..n
  if (s[i] != -1) v.push_back(make_pair(e[i], s[i]));
```

Отрезки заносили в вектор задом наперед (в виде [e_i; s_i]), чтобы по умолчанию сортировка отрезков происходила в возрастающем порядке их концов.

```
sort(v.begin(), v.end());
```

Находим максимальное количество *res* непересекающихся отрезков.

```
i = res = 0;
while (i < v.size())
{
```

i-ый отрезок добавляем во множество непересекающихся отрезков. Переменная *temp* содержит конец этого отрезка (назовем его текущим отрезком).

```
  res++; temp = v[i++].first; // end
```

Пока начало следующего отрезка меньше конца текущего, такой отрезок пропускаем (он пересекается с текущим).

```
  while (i < v.size() && v[i].second < temp) i++;
}
```

Выводим ответ.

```
printf("%d\n", v.size() - res);
```

6263. Башня карликов

Маленький Вася играет в игру “Башня карликов”. В этой игре имеются *n* различных предметов, которые Вы можете дать персонажу – карлику. Предметы пронумерованы от 1 до *n*. Вася хочет получить предмет номер 1.

Имеются два варианта получения предметов:

- Вы можете купить предмет. *i*-ый предмет стоит c_i денег.

- Вы можете сконструировать предмет. В игре имеется возможность сделать только m типов предметов. Для получения нового предмета Вам следует отдать два различных предмета.

Помогите Васе потратить наименьшее количество денег для приобретения предмета номер 1.

Вход. Первая строка содержит два числа n и m ($1 \le n \le 10000$, $0 \le m \le 100000$) – количество различных предметов и число конструирований.

Вторая строка содержит n целых чисел c_i ($0 \le c_i \le 10^9$) – стоимости предметов.

Следующие m строк описывают преобразования предметов, каждая строка содержит три различных целых числа a_i, x_i, y_i – a_i предмет, который можно сконструировать из x_i и y_i ($1 \le a_i, x_i, y_i \le n$, $a_i \ne x_i$, $x_i \ne y_i$, $y_i \ne a_i$).

Выход. Выведите одно число – наименьшее количество потраченных денег.

Пример входа 1

```
5 3
5 0 1 2 5
5 2 3
4 2 3
1 4 5
```

Пример выхода 1

```
2
```

Пример входа 2

```
3 1
2 2 1
1 2 3
```

Пример выхода 2

```
2
```

Построим граф – список смежности g, где g[i] содержит пары чисел (j, a) такие что из предметов i и j можно сконструировать предмет a: $(i, j) \to a$.

Пусть cost – массив, изначально содержащий стоимость покупки предметов (cost[i] = c_i). В конце алгоритма cost[i] будет содержать наименьшее количество денег, за которое можно получить предмет i.

Изначально все предметы являются необработанными (used[i] = 0).

Пока существует хотя бы один необрабротанный предмет

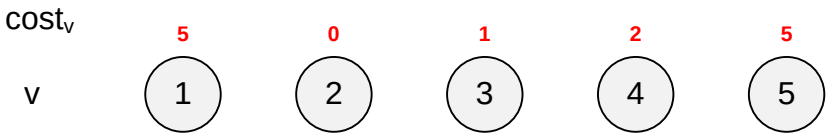
- Находим предмет a с наименьшей стоимостью;
- Применяем все правила, перечисленные в g[a];
- Отмечаем предмет a обработанным: used[a] = 0;

Для каждого правила $(a, b) \to to$ из g[a] пересчитываем

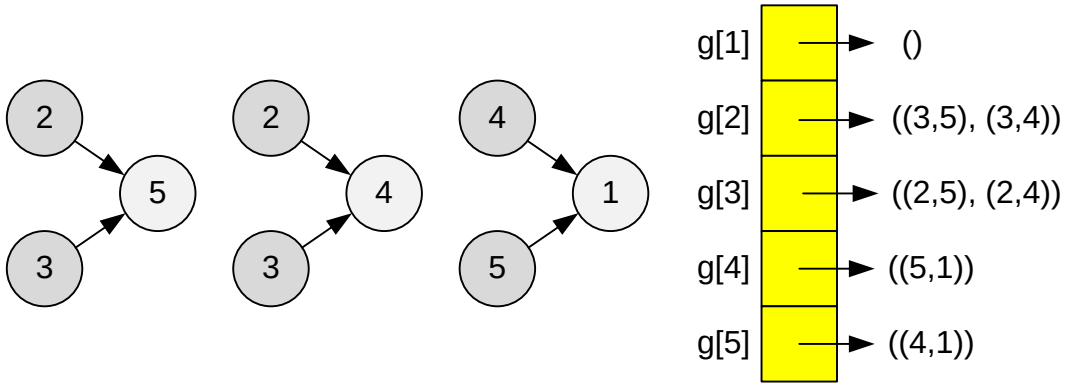
cost[to] = min(cost[to], cost[a] + cost[b])

Пример

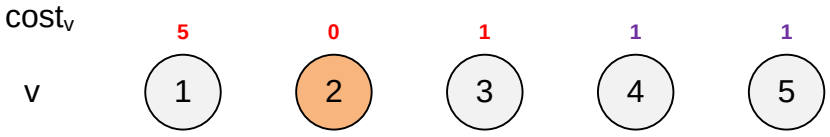
Рассмотрим первый тест. Имеются 5 предметов со следующими стоимостями их покупки:



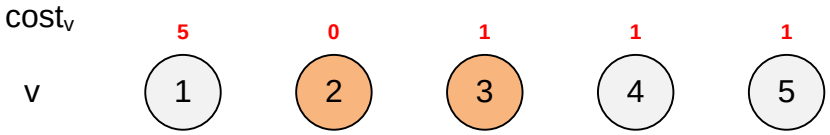
Имеются три правила конструирования предметов. Построим из них список смежности.



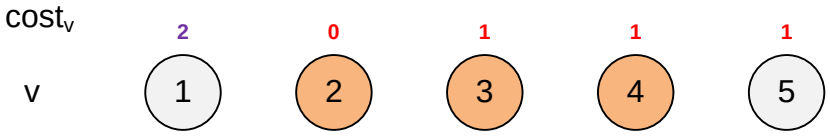
Предмет 2 имеет наименьшую стоимость cost[2] = 0. Применяем правила, в которых задействован предмет 2, а именно правила, перечисленные в g[2]. Отмечаем предмет 2 обработанным.



Следующим необработанным предметом с наименьшей стоимостью будет 3. Применяем правила, перечисленные в g[3]. Никакая из стоимостей не изменится.



Следующим необработанным предметом с наименьшей стоимостью будет 4. Применяем правило в g[4].



В следующих операциях стоимости предметов не изменятся. Таким образом стоимость получения предмета 1 равна 2.

Реализация алгоритма

Объявим рабочие массивы. Объявим список смежности g, содержащий правила конструирования предметов.

```
vector<int> cost, used;
vector<vector<pair<int, int>>> g;
```

Читаем входные данные. Инициализируем массивы.

```
scanf("%d %d", &n, &m);
g.resize(n + 1);
cost.resize(n + 1);
used.resize(n + 1);
```

Читаем стоимости покупки предметов.

```
for (i = 1; i <= n; i++)
  scanf("%d", &cost[i]);
```

Читаем правила конструирования предметов. Строим из них список смежности.

```
for (i = 0; i < m; i++)
{
  scanf("%d %d %d", &a, &x, &y);
  g[x].push_back(make_pair(y, a));
  g[y].push_back(make_pair(x, a));
}
```

Изначально все предметы считаются необработанными.

```
for (i = 1; i <= n; i++)
  used[i] = 0;

for (k = 0; k < n; k++)
{
```

Среди необработанных предметов ищем тот, который имеет наименьшую стоимость. Таким будет предмет номер *a*.

```
  mn = 2e9; a = -1;
  for (i = 1; i <= n; i++)
    if (!used[i] && cost[i] < mn)
    {
      mn = cost[i];
      a = i;
    }
```

Отмечаем предмет *a* обработанным.

```
  used[a] = 1;
```

Перебираем правила, в которых задействован предмет *a*.

```
  for (i = 0; i < g[a].size(); i++)
  {
    b = g[a][i].first;
    to = g[a][i].second; // (a, b) -> to
    if (cost[a] + cost[b] < cost[to]) cost[to] = cost[a] + cost[b];
  }
}
```

Выводим наименьшее количество потраченных денег для покупки первого предмета.

```
printf("%d\n", cost[1]);
```

9591. Лексикография

Люси любит буквы. Она изучала определение лексикографического порядка в школе и играет с ним.

Сначала она попыталась составить лексикографически наименьшее слово из заданных букв. Это было так просто! Затем она попыталась составить несколько слов и свести к минимуму одно из них. Это было намного сложнее!

Формально Люси хочет составить n слов длины l каждое из заданных $n * l$ букв, чтобы k-ое из них в лексикографическом порядке было лексикографически наименьшим.

Вход. Первая строка содержит три целых числа n, l и k ($1 \le k \le n \le 1000$, $1 \le l \le 1000$) – общее количество слов, длина каждого слова и индекс слова, которое Люси хочет минимизировать.

Далее следует строка из $n * l$ строчных букв английского алфавита.

Выход. Выведите n слов по l букв каждое, по одному слову в строке, используя буквы из входных данных. Слова должны быть отсортированы в лексикографическом порядке, а k - ое из них должно быть лексикографически как можно меньше. Если существует несколько ответов с наименьшим k-ым словом, то выведите любой из них.

Пример входа 1

```
3 2 2
abcdef
```

Пример выхода 1

```
ad
bc
ef
```

Пример входа 2

```
2 3 1
abcabc
```

Пример выхода 2

```
aab
bcc
```

Отсортируем заданные буквы. Пусть v – результирующий двумерный символьный массив. Будем заполнять его по столбцам: сначала заполняем буквами первый столбец, потом второй и так далее. Сначала столбцы заполняем до k-ой строки.

После того как розданы буквы в первом столбце (от 1 до k-ой строки), нам следует найти минимальный номер строки pt, совпадающей с k-ой. Раздаем буквы во втором столбце от строки pt до k-ой. Снова ищем минимальный номер строки

pt, совпадающей с *k*-ой. Раздаем буквы в третьем столбце от строки *pt* до *k*-ой. И так далее пока вся *k*-ая строка не будет заполнена.

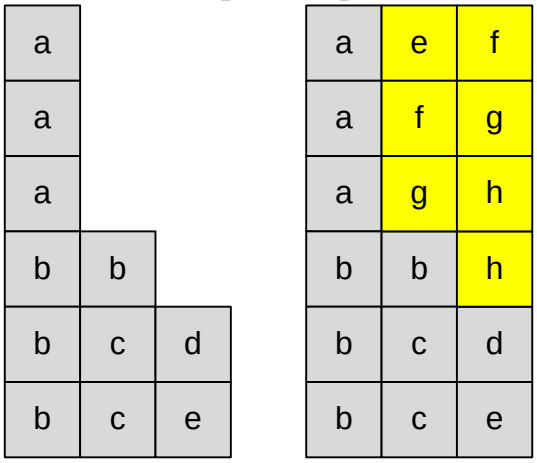
Оставшиеся буквы раздаем произвольным образом, *k*-ое слово в любом случае будет лексикографически наименьшим. Например, можно раздать оставшиеся буквы в лексикографическом порядке по строкам сверху вниз и слева направо.

Пример

Рассмотрим следующий тест

```
6 3 6
fgahhfgaaebdcbbebc
```

Отсортируем строку: `aaabbbbccdeeffgghh`. Нам следуем заполнить матрицу из 6 слов по 3 буквы, минимизировав при этом 6-е слово.



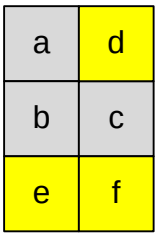
Заполняем первый столбец буквами сверху вниз (от 1-го до 6-го слова). Второй столбец следует заполнять буквами, начиная с самой ранней строки, у которой префикс совпадает с 6-ой. Положим *pt* = 4, и заполняем буквами второй столбец от строки номер *pt* до 6-ой. Наименьшей строкой, префикс которой совпадает с 6-ой, будет строка *pt* = 5. Начинаем с этой строки раздавать третью букву вплоть до 6-ой строки. Строка номер 6 при такой раздаче букв оказалась минимальной.

Далее раздаем недостающие буквы в лексикографическом порядке. Например, это можно сделать по строкам сверху вниз и слева направо.

Рассмотрим первый тест

```
3 2 2
abcdef
```

Отсортируем строку: `abcdef`. Нам следуем заполнить матрицу из 3 слов по 2 буквы, минимизировав при этом 2-е слово.



Заполняем первый столбец буквами сверху вниз (от 1-го до 2-го слова). Второй столбец следует заполнять буквами, начиная с самой ранней строки, у которой префикс совпадает с 2-ой. Положим *pt* = 2, и заполняем буквами второй столбец от строки номер *pt* до 2-ой. Строка номер 2 при такой раздаче букв оказалась минимальной.

Далее раздаем недостающие буквы в лексикографическом порядке по строкам сверху вниз и слева направо.

Реализация алгоритма

Читаем входные данные.

```
cin >> n >> l >> k;
cin >> s;
```

Отсортируем буквы во входной строке. Текущей обрабатываемой буквой будет s[*pos*].

```
pos = 0;
sort(s.begin(), s.end());
```

Объявим выводимую матрицу букв v (*n* слов по *l* букв).

```
vector<string> v(n, string(l, ' '));
```

Раздаем буквы по столбцу номер *j* по строкам от *pt* до *k* – 1 (нумерация строк идет с нуля).

```
int pt = 0;
for (j = 0; j < l; j++)
{
  for (i = pt; i < k; i++)
    v[i][j] = s[pos++];
```

Двигаем *pt* вперед (*pt*++) пока *j*-ый символ (*k* – 1)-ой строки не будет совпадать с *j*-ым символом строки *pt*. *pt* указывает на строку с наименьшим номером, которая на данный момент совпадает с (*k* – 1)-ой.

```
  while (v[pt][j] != v[k - 1][j]) pt++;
}
```

Заполняем остальные буквы матрицы. Двигаемся по строкам сверху вниз и слева направо.

```
for (i = 0; i < n; i++)
```

```
for (j = 0; j < l; j++)
  if (v[i][j] == ' ') v[i][j] = s[pos++];
```

Выводим ответ.

```
for (i = 0; i < n; i++)
  cout << v[i] << endl;
```

11262. Ожидаемая минимальная степень

Вам даны два положительных целых числа n и x.

Вы хотите выбрать x различных целых чисел, каждое от 1 до n включительно. Выбор будет сделан равномерно случайным образом. То есть каждое из возможных x-элементных подмножеств целых чисел от 1 до n будет выбрано с равной вероятностью.

Пусть S будет наименьшим целым числом среди x выбранных. Вычислите ожидаемое значение 2^S. Другими словами, определите среднее значение 2 в степени S, где среднее значение берется по всем возможным выборам x различных целых чисел.

Вход. Два натуральных числа: n ($1 \le n \le 50$) и x ($1 \le x \le n$).

Выход. Выведите среднее значение 2 в степени S с 4 десятичными цифрами.

Пример входа 1	**Пример выхода 1**
4 4	2.000000
Пример входа 2	**Пример выхода 2**
3 2	2.666667

Выбрать x различных целых чисел из n можно C_n^x способами.

Рассмотрим, сколько существует подмножеств из x чисел, минимальным элементом в которых будет число i. В такое множество следует выбрать число i и еще $x - 1$ число, каждое из которых больше i. Чисел, больших i, имеется $n - i$. Выбрать из них $x - 1$ число можно C_{n-i}^{x-1} способами. Отметим также, что должно выполняться неравенство $i + x - 1 \le n$, чтобы в подмножестве из x элементов наименьшим было i. Откуда $i \le n - x + 1$.

Для вычисления ожидаемого значения 2^S следует вычислить выражение

$$\frac{\sum_{i=1}^{n} 2^i \cdot C_{n-i}^{x-1}}{C_n^x}$$

При $x - 1 > n - i$ считаем $C_{n-i}^{x-1} = 0$.

Пример

В первом тесте единственная возможная ситуация состоит в том, чтобы выбрать (1, 2, 3, 4). Минимальным является число 1, ожидаемое значение равно $2^1 = 2$.

Во втором тесте имеется три равновероятных сценария: выбрать можно или {1, 2} или {1, 3} или {2, 3}. Соответствующие значения S равны 1, 1 и 2 соответственно. Средним значением 2^S будет $(2^1 + 2^1 + 2^2) / 3 = 8 / 3 = 2.6666666$.

Вычислим указанный ответ по формуле:

$$\frac{\sum_{i=1}^{n} 2^i \cdot C_{n-i}^{x-1}}{C_n^x} = \frac{2^1 \cdot C_2^1 + 2^2 \cdot C_1^1}{C_3^2} = \frac{2 \cdot 2 + 4 \cdot 1}{3} = \frac{8}{3}$$

Реализация алгоритма

Функция ***Cnk*** вычисляет значение C_n^k. Значения биномиального коэффициента будем заносить в ячейки массива dp.

```
long long dp[51][51];

long long Cnk(int n, int k)
{
   if (n == k) return 1;
   if (k == 0) return 1;
   if (dp[n][k] != -1) return dp[n][k];
   return dp[n][k] = Cnk(n - 1, k - 1) + Cnk(n - 1, k);
}
```

Основная часть программы.

```
scanf("%d %d", &n, &x);
memset(dp, -1, sizeof(dp));
```

Вычисляем ответ: $a = \frac{\sum_{i=1}^{n} 2^i \cdot C_{n-i}^{x-1}}{C_n^x}$.

```
a = 0;
for (i = 1; i <= n - x + 1; i++) // x - 1 <= n - i
   a = a + Cnk(n - i, x - 1) * (1LL << i);
res = 1.0 * a / Cnk(n, x);
```

Выводим ответ.

```
printf("%lf\n", res);
```

11269. Лемурьи вечеринки – базовая

В подчинении у короля лемуров Джулиана есть ровно 2 * k лемуров по 2 лемура каждого из k видов. Джулиан обожает вечеринки, поэтому каждый вечер он устраивает тусовку, однако в VIP-зоне, к сожалению, хватает мест только для него и еще n других лемуров.

Поскольку Джулиан не любит устраивать “одинаковые” вечеринки, то ему каждый день приходится выбирать кого звать в VIP-зону, чтобы наборы лемуров из VIP-зоны никогда не повторялись. Два лемура одного вида считаются неразличимыми. Наборы считаются одинаковыми, если они совпадают как мультимножества видов лемуров.

Помогите Джулиану определить, сколько дней он сможет проводить различные вечеринки. Так как ответ может быть большим, выведите его по модулю 1000000007.

Вход. В одной строке даны два целых числа k и n ($1 \le k \le 500\,000$, $0 \le n \le 2 * k$) – количество видов лемуров и количество мест в VIP-зоне.

Выход. Выведите одно число – ответ на задачу по модулю 1000000007.

Пример входа 1	**Пример выхода 1**
3 3	7

Пример входа 2	**Пример выхода 2**
4 3	16

На n мест некоторые виды будут представлены двумя лемурами, а некоторые одним. Пусть выбранными окажутся i пар лемуров (i видов представлены двумя лемурами), этот выбор можно совершить C_k^i способами. После этого в VIP зоне останется $n - 2i$ мест. Такое количество лемуров можно выбрать из $k - i$ оставшихся видов (по одному лемуру из каждого из $k - i$ оставшихся видов). Такой выбор можно сделать C_{k-i}^{n-2i} способами. Поскольку i может принимать любое значение от 0 до k, получаем ответ

$$\sum_{i=0}^{k} C_k^i \cdot C_{k-i}^{n-2i}$$

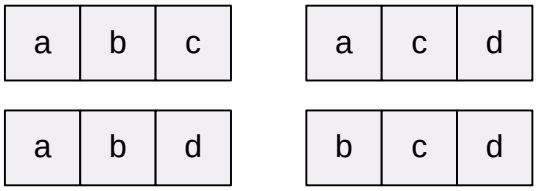
Пример

Рассмотрим второй тест, где имеется $k = 4$ пары лемуров и $n = 3$ места в VIP-зоне. По формуле получим:

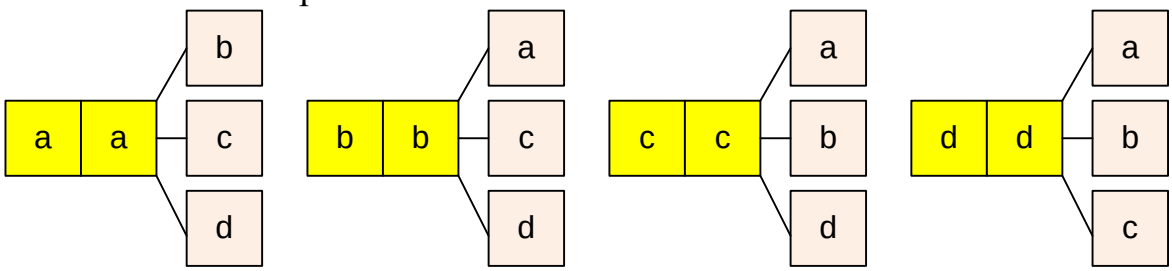
$$\sum_{i=0}^{4} C_4^i \cdot C_{4-i}^{3-2i} = C_4^0 \cdot C_4^3 + C_4^1 \cdot C_3^1 = 4 + 12 = 16$$

Выпишем только ненулевые слагаемые. C_n^k считаем равным нулю, если выполняется одно из трех неравенств: $k < 0$, $n < 0$ или $k > n$.

Обозначим лемуров буквами {*a*, *a*, *b*, *b*, *c*, *c*, *d*, *d*}. Одинаковым буквам соответствуют лемуры одного типа. Слагаемому $C_4^0 \cdot C_4^3 = 4$ соответствует факт, что никакие два лемура одного вида не являются выбранными, каждый из трех выбранных лемуров принадлежит разным видам. Возможными 4 выборками являются:



Рассмотрим слагаемое $C_4^1 \cdot C_3^1 = 12$. Из одного вида выбраны два лемура (4 варианта). На оставшееся одно место выбирается один лемур из трех оставшихся видов. Возможны 12 вариантов:



Очевидно, что два лемура из двух видов на 3 места взять невозможно. Поэтому остальные слагаемые суммы равны нулю.

Реализация алгоритма

Объявим константы.

```
#define MAX 1000001
#define MOD 1000000007
```

Объявим массивы: fact содержит факториалы чисел по модулю MOD, factinv содержит числа, обратные факториалам чисел по модулю MOD:

$$fact[n] = n! \bmod 1000000007$$

$$factinv[n] = n!^{-1} \bmod 1000000007$$

```
typedef long long ll;
ll fact[MAX], factinv[MAX];
```

Функция ***pow*** вычисляет степень числа по модулю: $x^n \bmod p$.

```
ll pow(ll x, ll n, ll p)
{
  if (n == 0) return 1;
  if (n % 2 == 0) return pow((x * x) % p, n / 2, p);
  return (x * pow(x, n - 1, p)) % p;
}
```

Функция ***inverse*** находит число, обратное x по простому модулю p. Поскольку число p простое, то по теореме Ферма $x^{p-1} \bmod p = 1$ для всякого $1 \le x < p$. Равенство можно переписать в виде $(x * x^{p-2}) \bmod p = 1$, откуда обратным к x является число $x^{p-2} \bmod p$.

```
ll inverse(ll x, ll p)
{
  return pow(x, p - 2, p);
}
```

Функция ***Cnk*** вычисляет биномиальный коэффициент по формуле:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

```
ll Cnk(int n, int k)
{
  return ((fact[n] * factinv[k]) % MOD * factinv[n - k]) % MOD;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d", &k, &n);
```

Заполняем массивы факториалов fact и factinv.

```
fact[0] = 1;
for (i = 1; i < MAX; i++)
  fact[i] = (fact[i - 1] * i) % MOD;

factinv[0] = 1;
for (i = 1; i < MAX; i++)
   factinv[i] = inverse(fact[i], MOD);
```

Вычисляем ответ по формуле $\sum_{i=0}^{k} C_k^i \cdot C_{k-i}^{n-2i}$. Значение биномиального коэффициента C_n^k считаем равным нулю, если выполняется одно из трех неравенств: $k < 0$, $n < 0$ или $k > n$.

```
res = 0;
for (i = 0; i <= k; i++)
{
  if (n - 2 * i < 0 || k - i < 0 || n - 2 * i > k - i) continue;
  res = (res + Cnk(k, i) * Cnk(k - i, n - 2 * i)) % MOD;
}
```

Выводим ответ.

```
printf("%lld\n", res);
```

546. Быстрый побег

Братья Ньютон планируют ограбить банк в городе Алвизо и хотят найти способ избежать всего лишь одной полицейской машины города. Они знают, что их машина быстрее полицейской. Поэтому если им удастся добраться до одной из магистралей, выходящих из города, то они без особых проблем смогут оторваться от полиции.

Максимальная скорость полицейской машины составляет 160 км/ч. К счастью, братья знают, откуда полицейская машина начнет движение (она припаркована в полицейском участке). Им также известно, что полицейская машина начнет движение, как только они покинут банк и начнут движение на своей машине (момент времени, когда сработает сигнал тревоги).

Братья хотят найти такой маршрут, который будет им гарантировать возможность покинуть город, независимо от того, какой маршрут выберет полицейская машина, и с какой скоростью она будет передвигаться. Поскольку братья чувствуют себя не очень уверенными водителями, то не хотят ехать быстрее, чем это необходимо. К счастью, они недавно инвестировали в новую хай-тек систему отрыва от полицейского автомобиля, которую Вы только что создали. Эта система сообщит Вам минимально необходимую максимальную скорость, достаточную для избегания встречи с полицейской машиной (а также и другие полезные вещи, как например, сам маршрут).

Давайте немного повернем время вспять, когда Вы были заняты построением системы эвакуации из города и сосредоточены на поиске минимальной требуемой скорости. Можете ли Вы найти ее правильно?

Все дороги можно считать бесконечно узкими, а обе машины как точечные объекты. Если браться окажутся в одной точке (на любой дороге или перекрестке) вместе с полицейской машиной, то они будут пойманы. И по закону Мерфи произойдет то что должно произойти. Обе машины начинают движение одновременно и могут ускориться / замедлиться мгновенно в любой момент времени до любой скорости, не большей максимально возможной. Они могут изменять дороги на перекрестках или направление движения в любом месте дороги мгновенно независимо от текущей скорости.

Вход. Первая строка содержит три целых числа n ($2 \le n \le 100$), m ($1 \le m \le 5000$) и e ($1 \le e \le n$), где n – количество перекрестков, m – количество дорог в городе, e – количество существующих магистралей. Далее следует m строк, каждая из которых содержит три целых числа a, b, l ($1 \le a < b \le n$, $1 \le l \le 100$), описывающих дорогу длины l (в сотнях метров) между перекрестками a и b. Далее следует строка из e целых чисел, каждое из промежутка 1...n, описывающих какой из перекрестков соединен с существующей магистралью. Последняя строка содержит два числа b и p ($1 \le b, p \le n$ и $b \ne p$), задающих перекрестки, с которых соответственно стартуют братья и полицейская машина.

Всегда есть возможность добраться от одного перекрестка к любому другому. Дороги соединяются только перекрестками (хотя могут пересекаться в других

местах используя мосты или тоннели). Дороги двусторонние, между двумя перекрестками может быть не более одной дороги.

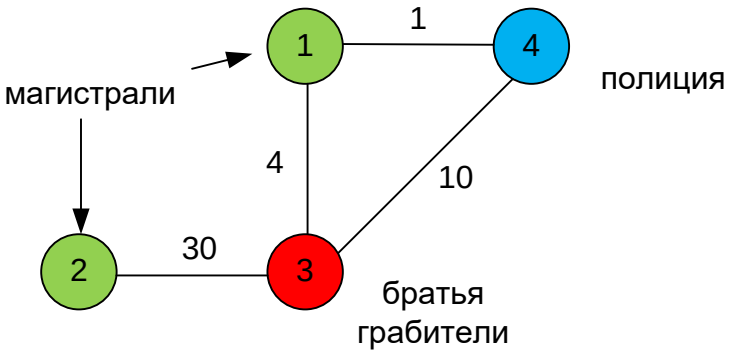
Выход. Вывести наименьшую скорость в км/ч, достаточную для того чтобы убежать из города, или слово IMPOSSIBLE, если это невозможно. В первом случае вывести ответ с точностью 10^{-9}.

Пример входа	**Пример выхода**
4 4 2 1 4 1 1 3 4 3 4 10 2 3 30 1 2 3 4	137.142857143

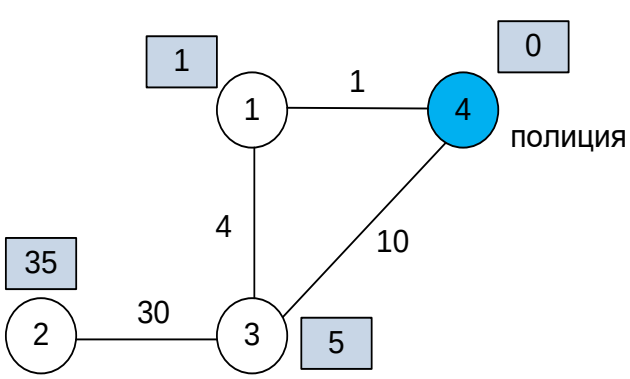
При помощи алгоритма Дейкстры ищем кратчайшие расстояния до всех вершин для полиции.

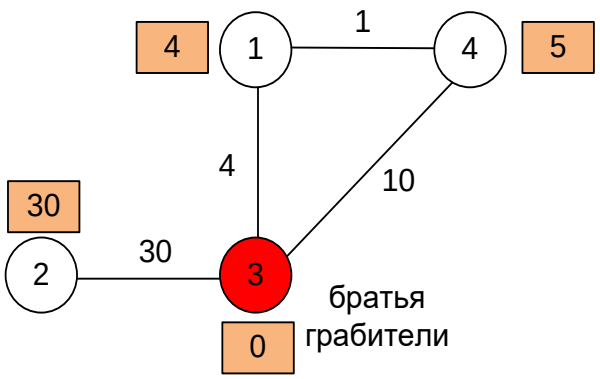
Искомую скорость братьев ищем бинарным поиском. Пусть она равна v_0. Запускаем алгоритм Дейкстры поиска кратчайших путей для братьев, причем движение в вершину возможно, если они там окажутся раньше полиции. Если братья смогут добраться хотя бы до одной магистрали, то они убегут от полиции, иначе нет. В зависимости от этого, используя бинарный поиск, изменяем скорость братьев.

Пример



При помощи алгоритма Дейкстры ищем кратчайшие расстояния до вершин для полиции (слева) и для братьев (справа).





Реализация алгоритма

Объявляем константы и глобальные переменные.

```
#define INF 0x3F3F3F3F
#define MAX 110
#define EPS 1e-9

int d[MAX][MAX], isHighWay[MAX];
int distPol[MAX], distBr[MAX], used[MAX];
```

Релаксация ребра (*v*, *to*) используя массив текущих кратчайших путей dist.

```
void Relax(int *dist, int v, int to)
{
  if (dist[to] > dist[v] + d[v][to])
    dist[to] = dist[v] + d[v][to];
}
```

Находим индекс минимального элемента в массиве dist среди еще не просмотренных (для которых used[*i*] = 0) вершин.

```
int Find_Min(int *dist)
{
  int i, v = -1, min = 0x3F3F3F3F;
  for(i = 1; i <= n; i++)
    if (!used[i] && (dist[i] < min)) min = dist[i], v = i;
  return v;
}
```

Алгоритм нахождения кратчайших путей из вершины source.

```
void Dijkstra(int *dist, int source)
{
  memset(used,0,sizeof(used));
  dist[source] = 0;

  for(int i = 1; i < n; i++)
  {
    int v = Find_Min(dist);
    for(int to = 1; to <= n; to++)
      if (!used[to]) Relax(dist,v,to);
    used[v] = 1;
```

```
  }
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d %d",&n,&m,&e);
memset(d,0x3F,sizeof(d));
memset(distPol,0x3F,sizeof(distPol));
memset(isHighWay,0,sizeof(isHighWay));
for(i = 0; i < m; i++)
{
  scanf("%d %d %d",&a,&b,&l);
  d[a][b] = d[b][a] = l;
}
```

Если вершина *i* – магистраль, то установим isHighWay[*i*] = 1.

```
for(i = 0; i < e; i++)
{
  int HW;
  scanf("%d",&HW);
  isHighWay[HW] = 1;
}

scanf("%d %d",&start,&police);
```

Вычисляем кратчайшие расстояния до всех вершин для полиции.

```
Dijkstra(distPol, police);
```

Ищем минимальную скорость *vel* братьев бинарным поиском на промежутке [*MinSpeed*; *MaxSpeed*].

```
MinSpeed = 0; MaxSpeed = INF;
while(MaxSpeed - MinSpeed > EPS)
{
    vel = (MaxSpeed + MinSpeed) / 2;
    memset(used,0,sizeof(used));
    memset(distBr,0x3F,sizeof(distBr));
    distBr[start] = 0;
```

Дейкстрой ищем минимальный путь до вершин, до которых могут добраться братья.

```
    int flag;
    for(int i = 1; i < n; i++)
    {
      int v = Find_Min(distBr);
      if (v == -1) break;
      for(int to = 1; to <= n; to++)
        if (!used[to] && (d[v][to] < INF))
        {
```

Братья, двигаясь со скоростью *vel*, могут попасть в *to* только если они там окажутся раньше полиции. Братья доберутся в *to* за вермя (distBr[*v*] + d[*v*][*to*]) / *vel*, а полиция за distPol[*to*] / 160.0.

```
            if ((distBr[to] > distBr[v] + d[v][to]) &&
               (distPol[to] / 160.0 > (1.0*distBr[v] + d[v][to]) / vel))
               distBr[to] = distBr[v] + d[v][to];
         }
       used[v] = 1;
     }
```

Проверяем, смогут ли братья добраться хотя бы до одной магистрали. Если смогут, устанавливаем *flag* = 1.

```
  flag = 0;
  for(int i = 1; i <= n; i++)
    if((distBr[i] < INF) && isHighWay[i]) flag = 1;
```

В зависимости от значения *flag* продолжаем бинарный поиск.

```
  if (flag) MaxSpeed = vel; else MinSpeed = vel;
}
```

Выводим найденную скорость.

```
vel = (MaxSpeed + MinSpeed) / 2;
if (vel > INF - 1) printf("IMPOSSIBLE\n");
else printf ("%.9lf\n", vel);
```