Февраль 7, 2022

Задача A. Дипломы
Задача B. Количество чисел вида 2^x * 3^y
Задача C. Защита королевства
Задача D. Самый длинный путь в дереве
Задача E. Замена
Задача F. Жадный Азиз
Задача G. SpaceX
Задача H. Очередь
Задача I. Автогонки
Задача J. Поиск в глубину – расстановка меток

3969. Дипломы

Когда Петя учился в школе, он часто участвовал в олимпиадах по информатике, математике и физике. Так как он был достаточно способным мальчиком и усердно учился, то на многих из этих олимпиад он получал дипломы. К окончанию школы у него накопилось n дипломов, причём, как оказалось, все они имели одинаковые размеры: w в ширину и h в высоту.

Сейчас Петя учится в одном из лучших университетов и живёт в общежитии со своими одногруппниками. Он решил украсить свою комнату, повесив на одну из стен свои дипломы за школьные олимпиады. Так как к бетонной стене прикрепить дипломы достаточно трудно, он решил купить специальную доску из пробкового дерева, чтобы прикрепить её к стене, а к ней – дипломы. Для того, чтобы эта конструкция выглядела более красиво, Петя хочет, чтобы была квадратной и занимала как можно меньше места на стене. Каждый диплом должен быть размещён строго в прямоугольнике w на h. Дипломы запрещается поворачивать на 90 градусов. Прямоугольники, соответствующие различным дипломам, не должны иметь общих внутренних точек.

Напишите программу, которая вычислит минимальный размер стороны доски, которая потребуется Пете для размещения всех своих дипломов.

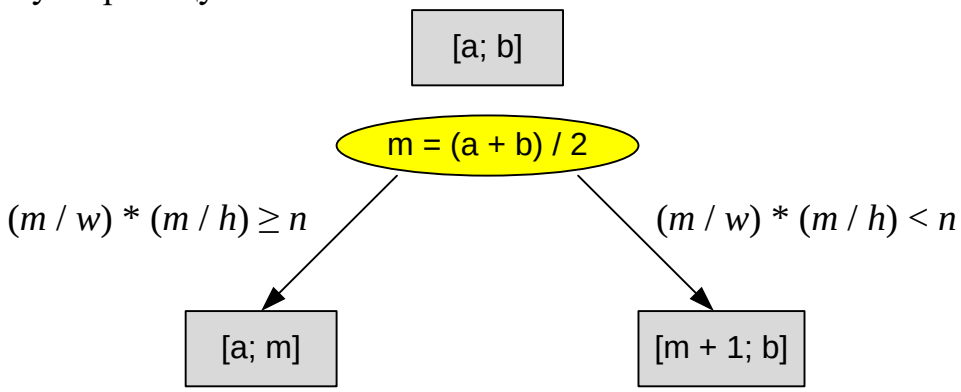
Вход. Три целых числа w, h, n ($1 \le w, h, n \le 10^9$).

Выход. Выведите искомый минимальный размер стороны доски.

Пример входа	Пример выхода
2 3 10	9

Пусть x – искомая длина стороны квадратной доски. Будем искать x бинарным поиском. Очевидно, что $0 < x \le \min(w, h) * n$. Поиск начнем на отрезке $[a; b] = [0; \min(w, h) * n]$.

Пусть $m = (a + b) / 2$ – текущая длина стороны доски. Поскольку диплом имеет размер $w * h$, то на ней можно разместить $(m / w) * (m / h)$ дипломов. Если это число меньше n, то размер доски маловат, устанавливаем $a = m + 1$. Иначе двигаем правую границу поиска: $b = m$.



Реализация алгоритма

Читаем входные данные.

```
scanf("%lld %lld %lld", &w, &h, &n);
```

Устанавливаем границы бинарного поиска $[a; b] = [0; \min(w, h) * n]$.

```
a = 0;
b = min(w, h) * n;
```

Запускаем бинарный поиск.

```
while (a < b)
{
  m = (a + b) / 2;
  if ((m / h) * (m / w) < n) a = m + 1;
  else b = m;
}
```

Выводим ответ.

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

9021. Количество чисел вида 2^x * 3^y

Найдите количество целых чисел на промежутке $[a, b]$, представимых в виде $2^x * 3^y$ ($x \ge 0, y \ge 0$).

Вход. Содержит не более 10^6 строк. Каждая строка содержит два целых числа a и b ($0 \le a \le b \le 10^{18}$), задающих один запрос.

Выход. Для каждого запроса выведите в отдельной строке количество целых чисел на интервале [a, b] включительно, которые имеют вид $2^x * 3^y$.

Пример входа	**Пример выхода**
1 10 100 200	7 5

Количество чисел вида $2^x * 3^y$, не больших 10^{18}, не так уж много. Сгенерируем их в массиве v.

Количество искомых чисел f(a, b) на отрезке [a, b] равно f(0, b) – f(0, a – 1). Запрос f(0, q) означает количество чисел вида $2^x * 3^y$, не больших q. Ответ на него ищем при помощи бинарного поиска на массиве v.

Пример

Сгенерируем числа вида $2^x * 3^y$, не больших 200.

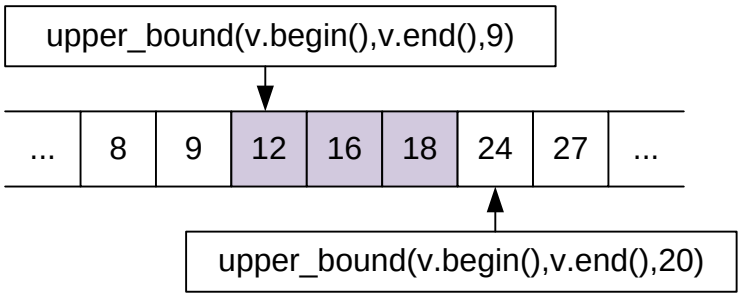
y	0	1	2	3	4
x = 0	1	3	9	27	81
x = 1	2	6	18	54	162
x = 2	4	12	36	108	
x = 3	8	24	72		
x = 4	16	48	144		
x = 5	32	96			
x = 6	64	192			
x = 7	128				

$2^x * 3^y$

Отрезку [1; 10] принадлежит 7 чисел (выделены синим).
Отрезку [100; 200] принадлежит 5 чисел (выделены зеленым).

Найдем решение для отрезка [10; 20]:

upper_bound(v.***begin***(), v.***end***(), 20) – ***upper_bound***(v.***begin***(), v.***end***(), 9) = 3



Действительно, отрезку [10; 20] принадлежит 3 числа искомого вида:
12, 16, 18

Реализация алгоритма

Объявим константу MAX = 10^{18}. Объявим рабочий массив v.

```
#define MAX 1000000000000000000LL
vector<long long> v;
```

Функция ***preprocess*** генерирует все числа вида $2^x * 3^y$, не больших 10^{18}, в массиве *v*. Для каждого значения $x = 1, 2, 4, 8, \ldots$ переберем $y = 1, 3, 9, 27, \ldots$ до тех пор пока $x * y < \text{MAX}$.

```
void preprocess()
{
  long long x = 1, y = 1;
  while (x < MAX)
  {
    while (x * y < MAX)
    {
      v.push_back(x * y);
      y *= 3;
    }
    x *= 2;
    y = 1;
  }
```

Отсортируем числа.

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

Основная часть программы. Генерируем массив чисел вида $2^x * 3^y$.

```
preprocess();
```

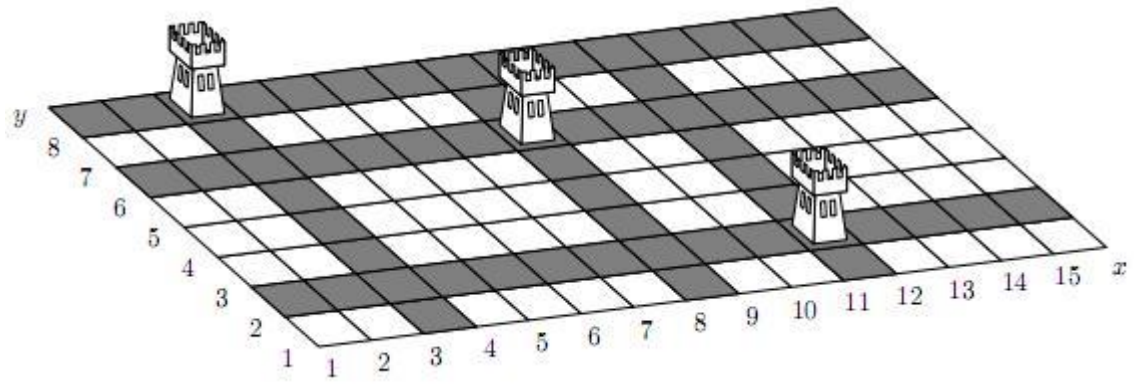
Читаем запрос – отрезок $[a, b]$. Количество искомых чисел $f(a, b)$ на отрезке $[a, b]$ равно $f(0, b) – f(0, a – 1)$.

```
while (scanf("%lld %lld", &a, &b) == 2)
{
  res = upper_bound(v.begin(), v.end(), b) - upper_bound(v.begin(),
                    v.end(), a - 1);
  printf("%lld\n", res);
}
```

2261. Защита королевства

Теодор реализует новую стратегию игры "Оборона Царства". На каждом уровне игрок защищает королевство, которое представлено прямоугольной сеткой ячеек. В некоторых клетках игрок строит арбалетные башни. Башня защищает все клетки в той же строке и том же столбце. Никакие две башни не находятся на одной строке или столбце.

Штрафом положения является количество клеток в крупнейшем незащищенном прямоугольнике. Например, положение, показанное на рисунке имеет штраф 12.



Помогите Теодору написать программу, вычисляющую штраф в заданной позиции.

Вход. Первая строка содержит три целых числа: w – ширина сетки, h – высота сетки и n – количество арбалетных башен ($1 \le w, h \le 40000$; $0 \le n \le \min(w, h)$).

Каждая из следующих n строк содержит два целых числа x_i и y_i – координаты клетки с башней ($1 \le x_i \le w$; $1 \le y_i \le h$).

Выход. Вывести одно число – количество клеток в наибольшем прямоугольнике, не защищенном башнями.

Пример входа	**Пример выхода**
15 8 3 3 8 11 2 8 6	12

Добавим две граничные башни с координатами (0, 0) и (w + 1, h + 1). Отсортируем абсциссы x_i башен и их ординаты y_i. Найдем максимальное расстояние между соседними парами абсцисс dx и ординат dy. Произведение dx * dy равно количеству клеток в крупнейшем незащищенном прямоугольнике.

Реализация алгоритма

Объявим массивы x и y в которых будем хранить координаты башен.

```
#define MAX 40010
int x[MAX], y[MAX];
```

Читаем входные данные.

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

Добавим две башни с координатами (0, 0) и (w + 1, h + 1), увеличим количество башен n на 2.

```
x[n] = y[n] = 0;
x[n+1] = w + 1; y[n+1] = h + 1;
n += 2;
```

Сортируем массив абсцисс и ординат.

```
sort(x,x+n);
sort(y,y+n);
```

Ищем максимальное расстояние dx между соседними абсциссами.

```
dx = 0;
for(i = 0; i < n - 1; i++)
  if (x[i+1] - x[i] > dx) dx = x[i+1] - x[i];
```

Ищем максимальное расстояние dy между соседними ординатами.

```
dy = 0;
for(i = 0; i < n - 1; i++)
  if (y[i+1] - y[i] > dy) dy = y[i+1] - y[i];
```

Выводим размер максимального незащищенного прямоугольника.

```
printf("%lld\n",1LL * (dx - 1) * (dy - 1));
```

10050. Самый длинный путь в дереве

Задано неориентированное взвешенное дерево. Найдите в нем самый длинный путь. То есть найдите такие две вершины, расстояние между которыми максимально.

Вход. Первая строка содержит количество вершин в дереве n ($2 \le n \le 10^5$). Следующие $n - 1$ строка описывают ребра. Каждая строка содержит три целых числа: номера вершин, соединенных ребром (вершины пронумерованы числами от 1 до n), и вес ребра w ($1 \le w \le 10^5$).

Выход. Выведите длину самого длинного пути.

Пример входа

```
6
1 2 3
2 3 4
2 6 2
6 4 6
6 5 5
```

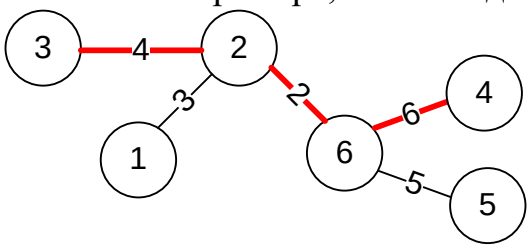
Пример выхода

```
12
```

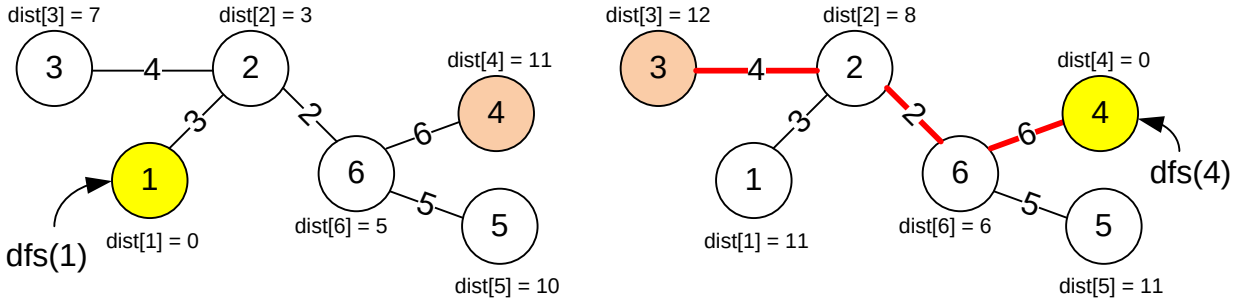
Кратчайший путь на взвешенном дереве можно найти поиском в ширину (не обязательно алгоритмом Дейкстры).

Запустим поиск в ширину из первой вершины. Найдем вершину *v*, до которой путь наибольший. Далее запустим поиск в ширину из вершины *v* и найдем вершину *u*, до которой путь наибольший. Найденный путь из *v* в *u* является самым длинным в дереве (диаметр дерева).

Пример. Граф, представленный в примере, имеет вид:



Запустим поиск в ширину из вершины 1 (граф слева). Самый длинный путь будет до вершины 4. Теперь запустим поиск в ширину из вершины 4 (граф справа). Самый длинный путь будет до вершины 3, он равен 12.



Реализация алгоритма

Определим структуру ребро, которое будем использовать в списке смежности графа:

- *v* – вершина, в которую направлено ребро;
- *dist* – длина ребра;

```
struct road
{
  int v;
  long long dist;
  road(int v, long long dist) : v(v), dist(dist) {}
};
```

Список смежности графа храним в *m*. Объявим очередь *q* и массив кратчайших расстояний *d*.

```
vector<vector<road> > m;
deque<long long> q;
vector<long long> d;
```

Функция ***bfs*** реализует поиск в ширину из вершины *v*.

```
int bfs(int v)
{
  q.clear();
  q.push_back(v);
  while (!q.empty())
  {
    int v = q.front(); q.pop_front();
    for (int i = 0; i < m[v].size(); i++)
    {
      int to = m[v][i].v;
      if (d[to] == -1)
      {
        d[to] = d[v] + m[v][i].dist;
        q.push_back(to);
      }
    }
  }
```

Возвращаем номер вершины, расстояние до которой от *v* наибольшее.

```
  return max_element(d.begin() + 1, d.begin() + n + 1) - d.begin();
}
```

Основная часть программы. Читаем входные данные. Строим граф.

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

for (i = 0; i < n - 1; i++)
{
  scanf("%d %d %lld", &b, &e, &dist);
  m[b].push_back(road(e, dist));
  m[e].push_back(road(b, dist));
}
```

Инициализация массива кратчайших расстояний.

```
d.resize(n + 1);
for (i = 0; i < d.size(); i++) d[i] = -1;
d[1] = 0;
```

Запускаем *первый* поиск в ширину из вершины 1. Вершина *v* – самая дальняя от 1.

```
int v = bfs(1);
```

Инициализируем массив кратчайших расстояний и запускаем *второй* поиск в ширину из вершины *v*.

```
for (i = 0; i < d.size(); i++) d[i] = -1;
d[v] = 0;
v = bfs(v);
```

Выводим ответ – длину наибольшего пути.

```
printf("%lld\n", d[v]);
```

4892. Замена

Дана последовательность натуральных чисел из n элементов. Требуется каждый элемент заменить на ближайший следующий за ним (то есть с большим индексом) элемент, который строго больше его по значению. Если большего элемента после данного нет, следует заменить данный элемент на ноль.

Вход. Первая строка содержит число элементов n ($1 \le n \le 10^5$). Вторая строка содержит n натуральных чисел a_i ($a_i \le 10^9$) – значения элементов последовательности.

Выход. Выведите искомую последовательность, разделяя соседние элементы одним пробелом.

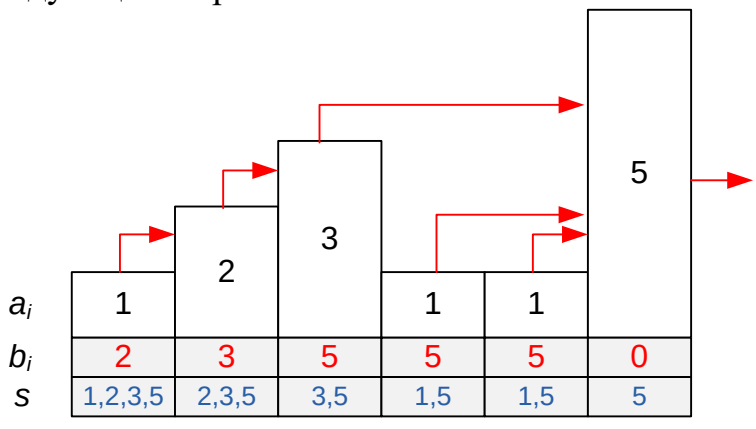
Пример входа 1
```
6
1 2 3 1 1 5
```

Пример выхода 1
```
2 3 5 5 5 0
```

Пример входа 2
```
5
1 2 3 4 5
```

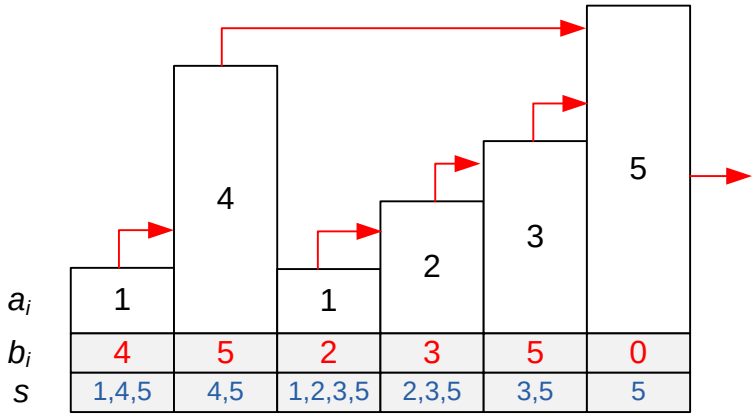
Пример выхода 2
```
2 3 4 5 0
```

Пусть числа входной последовательности a_1, a_2, …, a_n соответствуют высотам домов, расположенных рядом друг с другом. Например, первый тест можно изобразить следующим образом:



Пусть b_1, b_2, …, b_n – результирующий массив. Тогда b_i равно такому a_j, что если посмотреть в торец i-го дома слева, то дом a_j будет ближайшим видимым. Очевидно, что $b_n = 0$.

Будем обрабатывать дома справа налево. Пусть a_i – текущий рассматриваемый дом. Будем поддерживать множество s высот домов, видимых с левого торца i-го дома (включая a_i). Тогда b_i равно второму элементу множества s.



Рассмотрим, как поддерживать множество s при добавлении нового дома. Двигаемся справа налево, пусть уже был рассмотрен третий дом с высотой $a_3 = 1$. Текущее множество равно s = {1, 2, 3, 5}. Обрабатываем второй дом с высотой $a_2 = 4$. Очевидно, что он перекроет вид слева на все дома, не больше его высоты. Следует добавить значение $a_2 = 4$ во множество s, после чего удалить из s все числа, меньшие $a_2 = 4$ (можно воспользоваться методом ***erase*** для промежутка). После описанных операций множество примет вид s = {4, 5}.

Реализация алгоритма

Объявим множество s и итератор *it* на него. Входную и выходную последовательности храним в массивах in и out.

```
#define MAX 100001
set<int> s;
set<int>::iterator it;
int in[MAX], out[MAX];
```

Читаем входной массив.

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

Обрабатываем дома справа налево.

```
for(i = n - 1; i >= 0; i--)
{
```

Вставляем во множество высоту текущего дома.

```
  s.insert(in[i]);
```

Ищем позицию высоты in_i вставленного дома. Удаляем из множества s все высоты, меньшие in_i.

```
  it = s.find(in[i]);
  s.erase(s.begin(),it);
```

Выводим второй элемент множества. Если его не существует, выводим 0.

```
  it++;
  if(it == s.end())
    out[i] = 0;
  else
    out[i] = *it;
}
```

Выводим результирующий массив.

```
for(i = 0; i < n; i++)
  printf("%d ",out[i]);
printf("\n");
```

9033. Жадный Азиз

Знаете ли Вы, что Азиз очень любит шоколадные конфеты? Однако отец не позволяет ему есть много шоколада, так как он вредит зубам.

Отец дал ему массив, в котором много одинаковых чисел. В день Азиз может съесть столько конфет, сколько раз повторяется максимально встречаемое число в массиве.

Азиз хочет смошенничать чтобы съесть больше конфет. Он может изменить некоторые числа в массиве, и отец этого не заметит. Он может увеличить или уменьшить любое число в массиве на 1 единицу и только 1 раз.

Азиз хочет съесть максимальное количество конфет. Помогите ему в этом деле.

Найдите максимальное количество конфет, которое Азиз может получить.

Вход. Первая строка содержит количество элементов n ($1 \le n \le 10^5$) в массиве. Вторая строка содержит n элементов a_i ($0 \le a_i \le 10^9$) массива.

Выход. Выведите максимальное количество конфет, которое Азиз может получить.

Пример входа 1

```
2
1 2
```

Пример выхода 1

```
2
```

Пример входа 2

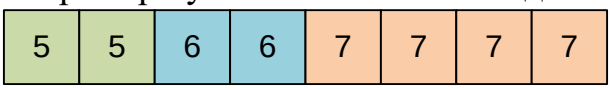
```
4
1 3 3 5
```

Пример выхода 2

```
3
```

Подсчитаем количество раз, которое каждое число встречается в массиве. Для этого воспользуемя структурой данных ***map***: m[x] будет содержать количество раз, которое число x встречается в массиве.

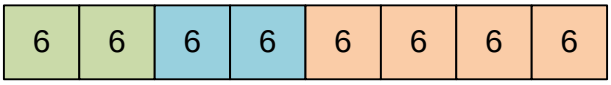
Для каждого числа a[i] в массиве предположим что оно является максимально встречаемым после мошенничества Азиза. Для этого Азиз должен число (a[i] – 1) увеличить на 1, а число (a[i] + 1) уменьшить на 1 (если такие числа в массиве существуют). Например пусть массив имеет вид



Структура map содержит следующие данные (две пятерки, две шестерки и четыре семерки):

$$m[5] = 2, m[6] = 2, m[7] = 4$$

Для того чтобы число 6 было максимально встречаемым после мошенничества, необходимо ко всем числам 5 прибавить 1, а изо всех чисел 7 вычесть 1:

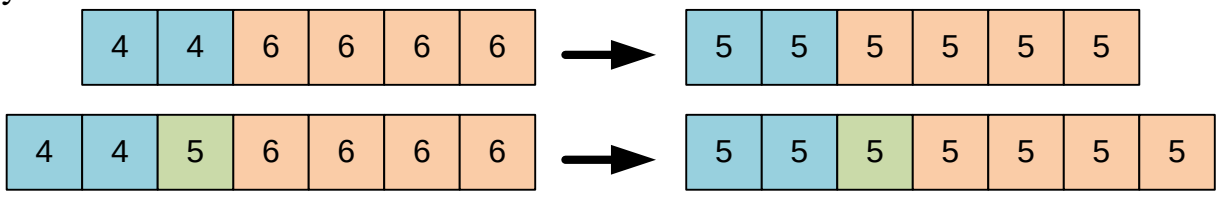


Пусть изначально a[i] = 6. Тогда массив изначально содержит m[a[i]] шестерок, m[a[i] – 1] пятерок и m[a[i] + 1] семерок. После мошенничества число шестерок станет равным

$$m[a[i] - 1] + m[a[i]] + m[a[i] + 1],$$

что и будет равно количеству полученных Азизом конфет.

Рассмотрим ситуацию когда массив содержит два числа a[i] = 4 и a[i] + 2 = 6 (разность между которыми равна 2), но при этом число a[i] + 1 например может отсутствовать.



В таком случае оптимально будет все числа a[i] увеличить на 1, а все числа a[i] + 2 уменьшить на 1. После мошенничества количество чисел a[i] + 1 станет равным

$$m[a[i]] + m[a[i] + 1] + m[a[i] + 2]$$

Реализация алгоритма

Объявим входной массив a. Объявим переменную m типа ***map***.

```
map<int, int> m;
int a[100000];
```

Читаем входной массив. Подсчитаем количество раз, которое каждое число a[i] встречается в массиве a.

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

В переменной *res* вычисляем ответ.

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

Если максимально встречаемым числом после мошеничества будет a[i].

```
  res = max(res, m[a[i]] + m[a[i] - 1] + m[a[i] + 1]);
```

Если максимально встречаемым числом после мошеничества будет a[i] + 1.

```
  res = max(res, m[a[i]] + m[a[i] + 1] + m[a[i] + 2]);
}
```

Выводим ответ.

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

10030. SpaceX

Илон Маск планирует отправить свои космические корабли на k различных планет. Для этого у него есть n космических кораблей. Изначально известно, куда будет отправлен каждый корабль. Планеты пронумерованы от 1 до 10^9. Как главному космоинженеру компании SpaceX, вам предоставлено право менять пункт назначения любого корабля. Вам, за минимальное количество изменений, нужно сделать так, чтобы все корабли были отправлены на k различных планет.

Вход. В первой строке даны два числа n ($1 \le n \le 10^5$) и k ($1 \le k \le n$). Во второй строке расположены n целых чисел p_i ($1 \le p_i \le 10^5$) – изначальные пункты назначения кораблей.

Выход. Выведите минимальное количество изменений.

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

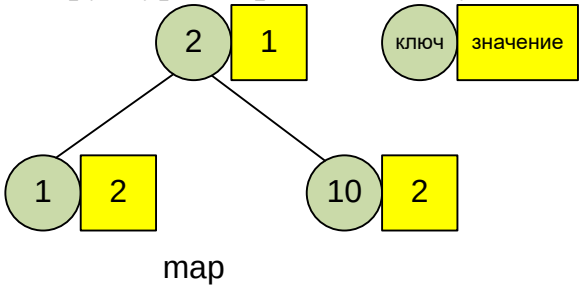
Пример входа 2

```
5 4
10 1 2 1 10
```

Пример выхода 2

```
1
```

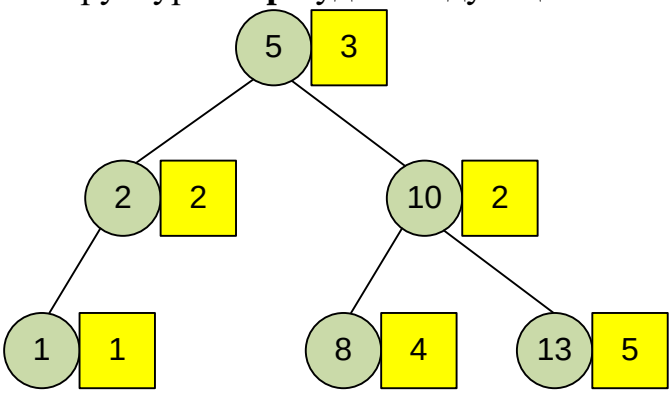
Для каждого пункта назначения *p* в отображении *m* подсчитаем количество кораблей m[*p*], отправленных туда. Например, во втором тесте две ракеты отправлены на планету 1, одна ракета отправлена на планету 2, и две ракеты отправлены на планету 10. Структура **map** имеет следующий вид:



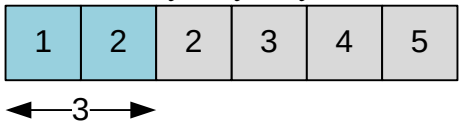
map

Размер отображения *m*.size() = 3, все корабли отправены на 3 планеты. Корабли следует отправить в точности на *k* = 4 различных планет. Если *m*.size() < *k*, то *k* – *m*.size() кораблям следует сменить курс.

Рассмотрим случай, когда космические корабли отправлены на более чем *k* планет. Пусть *k* = 4, но структура **map** будет следующей:



17 кораблей отправлены на 6 планет. Из двух планет следует перенаправить корабли на любые из 4 оставшихся. Поскольку требуется минимизировать количество изменений, то следует найти две планеты, на которые отправлены наименьшее количество кораблей. Отсортируем числа – количество кораблей, отправленные на планеты. И найдем сумму двух наименьших.



Реализация алгоритма

Объявим структуры данных.

```
map<long long, long long> m;
vector<long long> v;
```

Читаем входные данные.

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

В m[*x*] подсчитываем количество кораблей, отправленных на планету *x*.

```
  m[x]++;
}
```

Если все космические корабли отправлены менее чем на *k* планет, то минимальное количество изменений равно *k* – *m*.size().

```
if (m.size() < k)
{
  printf("%d\n", k - m.size());
  return 0;
}
```

Космические корабли отправлены более чем на *k* планет. Строим вектор *v*, содержащий количество кораблей, отправленных на разные планеты.

```
for (iter = m.begin(); iter != m.end(); iter++)
  v.push_back((*iter).second);
```

Сортируем вектор *v*.

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

Ищем сумму *m*.size() – *k* наименьших чисел (именно с такого количества планет следует изменить курс ракет).

```
sum = 0;
for (i = 0; i < m.size() - k; i++)
  sum += v[i];
```

Выводим ответ.

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

3004. Очередь

В цивилизованных странах на железнодорожном вокзале работают *k* касс, однако очередь в них всего одна. Обслуживание происходит следующим образом. Изначально, когда все кассы свободны, первые *k* человек из очереди подходят к кассам. Остальные ждут своей очереди. Как только кто-нибудь будет обслужен и соответствующая касса освободится, следующий человек из очереди подходит к этой кассе. Так продолжается до тех пор, пока не будут обслужены все клиенты.

Определите время, за которое будут обслужены все клиенты.

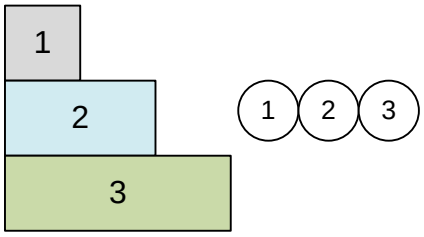
Вход. В первой строке находится два целых числа: размер очереди n и количество касс k ($1 \le n \le 10^5$, $1 \le k \le 10^4$). Во второй строке задаются n натуральных чисел. i-ое число определяет время t_i ($1 \le t_i \le 10^5$), которое требуется для того, чтобы обслужить i-го клиента из очереди.

Выход. Выведите одно число – время, за которое будет обслужена заданная очередь.

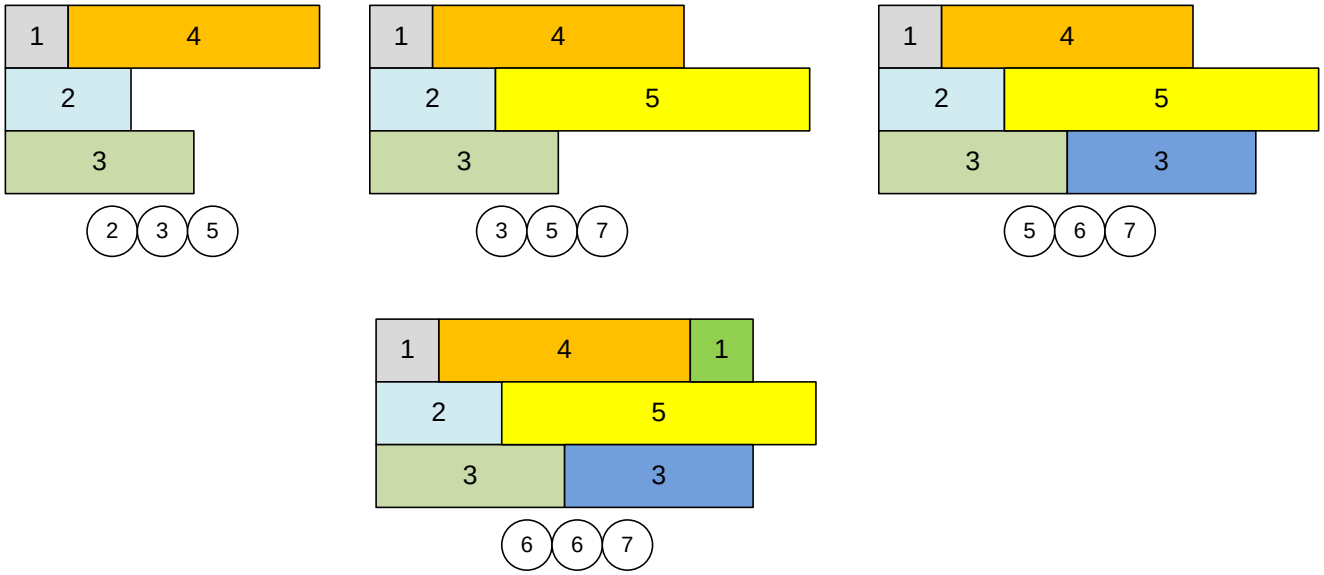
Пример входа	**Пример выхода**
7 3 1 2 3 4 5 3 1	7

Будем моделировать процесс продажи билетов при помощи мультимножества s. Первые k людей подводим к свободным кассам – заносим время их обслуживания в мультимножество. При дальнейшей обработке мультимножество будет содержать k элементов. Каждый их них отображает момент времени, в которое соответствующая касса станет свободной и к ней сможет подойти следующий человек. Очевидно, что каждый раз новый человек должен подходить к той кассе, для которой это время минимально. Время последнего обслуженного клиента и будет искомым.

Пример. Рассмотрим приведенный пример. Кладем в кучу k = 3 первых элементов.



Далее на каждой итерации берем следующий элемент (следующего человека из очереди) и кладем вместо наименьшего (человека ставим в ту очередь которая раньше закончится). В очередь кладем время, в которое этот новый человек отойдет от кассы. Возле каждого рисунка приведены числа, находящиеся в куче.



Реализация алгоритма

Информацию о моментах времени, в которые происходит продажа билетов в кассах, храним в мультимножестве *s*.

```
multiset<long long> s;
```

Читаем входные данные. Время обслуживания первых *k* людей заносим в мультимножество *s*.

```
scanf("%d %d",&n,&k);
for(i = 0; i < n; i++)
{
  scanf("%d",&ti);
  if (s.size() != k) s.insert(ti);
  else
  {
```

Находим человека, который раньше всего отойдет от касс, и ставим за ним следующего человека из очереди.

```
    long long temp = *s.begin();
    s.erase(s.begin());
    s.insert(temp + ti);
  }
}
```

Наибольшее число в мультимножестве *s* равно моменту времени, в который обслужится последний клиент. Оно и будет искомым.

```
while(s.size() > 1) s.erase(s.begin());
printf("%lld\n",*s.begin());
```

Реализация через очередь с приоритетами

Заведем кучу, в корне которой находится наименьший элемент.

```
priority_queue<long long, vector<long long>, greater<long long> > pq;
```

Читаем входные данные. Время обслуживания первых *k* людей заносим в очередь *pq*.

```
scanf("%d %d",&n,&k);
for(i = 0; i < n; i++)
{
  scanf("%d",&ti);
  if (pq.size() != k) pq.push(ti);
  else
  {
```

Находим человека, который раньше всего отойдет от касс, и ставим за ним следующего человека из очереди.

```
    long long temp = pq.top(); pq.pop();
    pq.push(temp + ti);
  }
}
```

Наибольшее число в очереди *pq* равно моменту времени, в который обслужится последний клиент. Оно и будет искомым.

```
while(pq.size() > 1) pq.pop();
printf("%lld\n",pq.top());
```

1390. Автогонки

В городе N в ближайшее время состоится этап чемпионата мира по автогонкам среди автомобилей класса Формула-0. Поскольку специальный автодром для этих соревнований организаторы построить не успели, было решено организовать трассу на улицах города.

В городе N есть *n* перекрёстков, некоторые пары которых соединены дорогами, движение по которым возможно в обоих направлениях. При этом любые два перекрёстка соединены не более чем одной дорогой, и есть возможность доехать по дорогам от любого перекрёстка до любого другого.

Трасса, на которой будут проводится соревнования, должна быть круговой (т.е. должна начинаться и заканчиваться на одном и том же перекрёстке), при этом в процессе движения по ней никакой перекрёсток не должен встречаться более одного раза.

На предварительном этапе подготовки оргкомитетом был создан список всех дорог города. Теперь настало время его использовать. Первый вопрос, который необходимо решить – это вопрос о существовании в городе требуемой круговой трассы (разумеется, если ответ будет отрицательным, организаторам придётся в срочном порядке построить ещё несколько дорог). Единственная проблема заключается в том, что у организаторов есть подозрение, что, поскольку список составлялся не очень внимательно, в нём некоторые дороги указаны более одного раза.

Напишите программу, которая по заданному списку дорог города определит, возможна ли организация в городе требуемой круговой трассы.

Вход. Первая строка содержит два целых числа: количество перекрёстков n ($1 \le n \le 1000$) в городе N и количество дорог m ($0 \le m \le 10^5$) в составленном списке.

Последующие m строк описывают дороги. Каждая дорога описывается двумя числами: u и v ($1 \le u, v \le n, u \ne v$) – номера перекрёстков, которые она соединяет. Так как дороги двусторонние, то пара чисел (u, v) и пара чисел (v, u) описывают одну и ту же дорогу.

Выход. Вывести YES, если в городе возможно организовать круговую трассу для соревнований, и слово NO в противном случае.

Пример входа 1

```
3 4
1 2
2 3
3 1
3 2
```

Пример выхода 1

```
YES
```

Пример входа 2

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

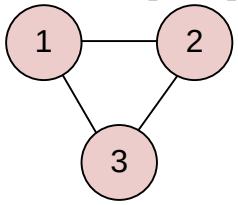
Пример выхода 2

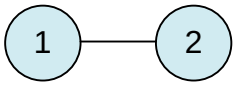
```
NO
```

Имеется неориентированный не обязательно связный граф. Необходимо проверить, содержит ли он цикл (который можно превратить в трассу Формулы-0).

Неориентированный граф содержит цикл, если существует обратное ребро. То есть ребро, ведущее в уже пройденную вершину.

Пример. Приведенные в примере графы имеют вид:





Первый граф содержит цикл, второй нет.

Реализация алгоритма

Граф храним в матрице смежности g. Объявим массив *used* для поиска в глубину.

```
#define MAX 1010
int g[MAX][MAX], used[MAX];
```

Поиск в глубину из вершины *v*. Если имеется ребро из *v* в *i*, причем *i* уже пройдена (used[*i*] = 1), то в графе имеется цикл. Устанавливаем *flag* = 1.

Необходимо только отсечь случай, когда из *v* мы направляемся в предка: предок уже пройден, но цикла нет. Для этого в функции dfs введем второй параметр *prev* – предка вершины *v*.

```
void dfs(int v, int prev = -1)
{
  used[v] = 1;
  for(int i = 1; i <= n; i++)
    if ((i != prev) && g[v][i])
      if (used[i]) flag = 1; else dfs(i,v);
}
```

Основная часть программы. Читаем входной неориентированный граф. Граф храним в матрице смежности, таким образом повторяющиеся дороги будут учитываться только один раз.

```
scanf("%d %d",&n,&m);
memset(g,0,sizeof(g));
memset(used,0,sizeof(used));
for(i = 0; i < m; i++)
{
  scanf("%d %d",&u,&v);
  g[u][v] = g[v][u] = 1;
}
```

Запускаем поиск в глубину из каждой вершины (граф может быть несвязным).

```
flag = 0;
for(i = 1; i <= n; i++)
  if (!used[i]) dfs(i);
```

В зависимости от значения переменной *flag* выводим ответ.

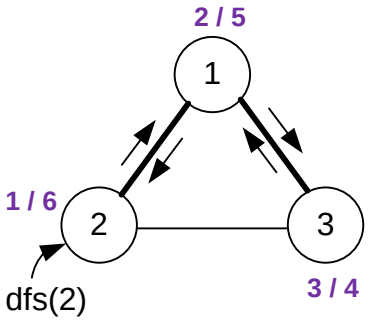
```
if (flag) printf("YES\n");
else printf("NO\n");
```

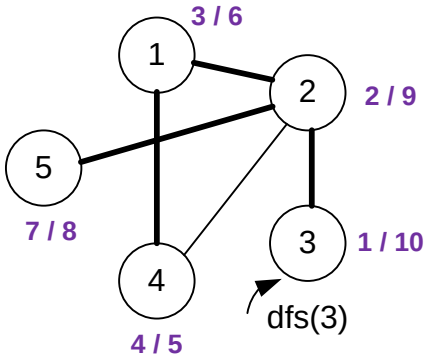
8761. Поиск в глубину – расстановка меток

Задан неориентированный граф. Запустите поиск в глубину из заданной вершины *v*. Выведите метки d[*v*] и f[*v*] для каждой вершины *v* в порядке возрастания вершин.

Вход. Первая строка содержит количество вершин n ($n \le 100$) и ребер m неориентированного графа. Вершины нумеруются начиная с 1. Каждая из следующих m строк содержит две вершины a и b – неориентированное ребро графа. Последняя строка содержит вершину v.

Выход. Запустите ***dfs***(v). Выведите метки d[v] и f[v] для каждой вершины v (v = 1, 2, ..., n). Метки для каждой вершины следует выводить в отдельной строке.





Пример входа 1

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

Пример выхода 1

```
2 5
1 6
3 4
```

Пример входа 2

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

Пример выхода 2

```
3 6
2 9
1 10
4 5
7 8
```

Построим матрицу смежности по списку ребер. Запустим поиск в глубину из заданной вершины v. Значения меток заносим в массивы d и f.

Реализация алгоритма

Объявим матрицу смежности g, массив лампочек used, а также массивы d и f. Объявим счетчик времени t.

```
#define MAX 101
int g[MAX][MAX], used[MAX];
int d[MAX], f[MAX];
int t;
```

Функция ***dfs*** совершает поиск в глубину из вершины *v*.

```
void dfs(int v)
{
```

Заносим метку входа.

```
  d[v] = t++;
```

Отмечаем вершину *v* как пройденную.

```
  used[v] = 1;
```

Перебираем вершины *i*, в которые можно пойти из *v*. В вершину *i* можно пойти из *v*, если между ними существует ребро (g[*v*][*i*] = 1) и вершина *i* еще не пройдена (used[*i*] = 0).

```
  for (int i = 1; i <= n; i++)
    if ((g[v][i] == 1) && (used[i] == 0)) dfs(i);
```

Заносим метку выхода.

```
  f[v] = t++;
}
```

Основная часть программы. Читаем количество вершин *n* и ребер *m*.

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

Обнуляем рабочие массивы.

```
memset(g, 0, sizeof(g));
memset(used, 0, sizeof(used));
```

По списку ребер строим матрицу смежности.

```
for (i = 0; i < m; i++)
{
  scanf("%d %d", &a, &b);
  g[a][b] = g[b][a] = 1;
}
```

Читаем вершину *v*, устанавливаем время *t* = 1 и запускаем из *v* поиск в глубину.

```
scanf("%d", &v);
t = 1;
dfs(v);
```

Выводим метки d[*i*] и f[*i*] для каждой вершины *i*.

```
for (i = 1; i <= n; i++)
  printf("%d %d\n", d[i], f[i]);
```