Март 18, 2022

Задача A. Веселая функция
Задача B. Простое сложение
Задача C. Строки Фибоначчи
Задача D. Муу
Задача E. Деление по модулю
Задача F. Числа Каталана
Задача G. Парад скобок
Задача H. Мост
Задача I. Городской горизонт
Задача J. Теодор Рузвельт

8304. Веселая функция

Вычислить значение функции

$$f(x,y)=\begin{cases}0, x\le 0 \text{ or } y\le 0\\ f(x-1,y-2)+f(x-2,y-1)+2, x\le y\\ f(x-2,y-2)+1, x>y\end{cases}$$

Вход. Два целых числа x, y ($0 \le x, y \le 50$).

Выход. Вывести значение функции f(x, y).

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

Пример входа 2	**Пример выхода 2**
34 12	6

Реализуем рекурсивную функцию с запоминанием результатов.

Реализация алгоритма

Объявим двумерный массив для запоминания значений функции: dp[x][y] = f(x, y).

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

Реализация рекурсивной функции с запоминанием.

```
long long f(int x, int y)
{
  if (x <= 0 || y <= 0) return 0;
  if (dp[x][y] != -1) return dp[x][y];
  if (x <= y) return dp[x][y] = f(x-1,y-2) + f(x-2,y-1) + 2;
  return dp[x][y] = f(x-2,y-2) + 1;
}
```

Основная часть программы. Читаем входные данные. Инициализируем ячейки массива dp значением -1. Вызываем функцию f(x, y) и выводим ее значение.

```
scanf("%d %d",&x,&y);
memset(dp,-1,sizeof(dp));
printf("%lld\n",f(x,y));
```

1517. Простое сложение

Определим следующую рекурсивную функцию f(n):

$$f(n) = \begin{cases} n\%10, \textit{если } n\%10 > 0 \\ 0, \textit{если } n = 0 \\ f(n/10) \textit{ иначе} \end{cases}$$

Определим функцию S(p, q) следующим образом:

$$S(p, q) = \sum_{i=p}^{q} f(i)$$

По заданным p и q необходимо вычислить S(p, q).

Вход. Состоит из нескольких тестов. Каждая строка содержит два неотрицательных целых числа p и q ($p \le q$), разделенных пробелом. p и q являются 32 битовыми знаковыми целыми. Последняя строка содержит два отрицательных целых числа и не обрабатывается.

Выход. Для каждой пары p и q в отдельной строке вывести значение S(p, q).

Пример входа	**Пример выхода**
1 10 10 20 30 40 -1 -1	46 48 52

Приведенная в условии функция f(n) находит последнюю ненулевую цифру числа n. Например, f(1234) = 4, f(3900) = f(390) = f(39) = 9.

Обозначим через

$$g(p) = \sum_{i=1}^{p} f(i)$$

Тогда $S(p, q) = g(q) – q(p – 1)$.

Для вычисления функции $g(p)$, суммы последних значащих цифр для чисел от 1 до p, разобьем числа от 1 до p на три множества (операция деления ‘/’ является целочисленной):

1. Числа от $(p / 10) * 10 + 1$ до p;
2. Числа от 1 до $(p / 10) * 10$, не оканчивающиеся нулем;
3. Числа от 1 до $(p / 10) * 10$, оканчивающиеся нулем;

Сумма последних значащих цифр в первом множестве равна $1 + 2 + … + p \bmod 10 = t * (t + 1) / 2$, где $t = p \bmod 10$. Во втором множестве искомая сумма равна $p / 10 * 45$, так как сумма всех цифр от 1 до 9 равна 45, а число полных десятков равно $p / 10$. Требуемую сумму для третьего множества найдем рекурсивно: она равна $g(p / 10)$. Получим рекуррентность:

$$g(p) = \frac{t \cdot (t+1)}{2} + 45 \cdot \left\lfloor \frac{p}{10} \right\rfloor + g\left(\left\lfloor \frac{p}{10} \right\rfloor \right), \; t = p \bmod 10$$

$$g(0) = 0$$

Пример. При $p = 32$ к первому множеству отнесутся числа 31, 32, ко второму 1, …, 9, 11, …, 19, 21, …, 29, к третьему 10, 20, 30. Значение $t = 32 \bmod 10 = 2$.

I

31	32

$S_1 = 1 + 2 = 3$

II

1	2	3	...	9
11	12	13	...	19
21	22	23	...	29

$S_2 = 45 * 3 = 135$

III

10	20	30
1	**2**	**3**

$S_3 = f(1) + f(2) + f(3) = g(3)$

$$g(32) = \frac{2 \cdot 3}{2} + 45 * 5 + g(3) = 3 + 135 + (1 + 2 + 3) = 144$$

Пусть $p = 1234$.

I

1231	1232	1233	1234

$S_1 = 1 + 2 + 3 + 4 = 10$

II

1	2	3	...	9
11	12	13	...	19
. . .				
1221	1222	1223	...	1229

$S_2 = 45 * 123 = 5535$

III

10	20	30	...	1230
1	**2**	**3**	**...**	**123**

$S_3 = f(1) + f(2) + … + f(123) = g(123)$

$$g(1234) = \frac{4 \cdot 5}{2} + 45 * 123 + g(123) = 10 + 5535 + g(123) = 5545 + g(123)$$

Вычислив значение g(123) = 595, получим:

$$g(1234) = 5545 + g(123) = 5545 + 595 = 6140$$

Реализация алгоритма

Поскольку выполняется обработка 32-битовых знаковых чисел, то для избежания переполнения при вычислениях используем тип long long.

```
long long p, q;
```

Функция g(*p*) вычисляет сумму значений функции f(*n*) для значений аргумента *n* от 1 до *p*.

```
long long g(long long p)
{
  long long t = p % 10;
  if (p == 0) return 0;
  return t * (1 + t) / 2 + p / 10 * 45 + g(p / 10);
}
```

Значение функции S(*p*, *q*) считаем как g(*q*) – q(*p* – 1).

```
long long s(long long p, long long q)
{
  return g(q) - g(p - 1);
}
```

Основной цикл программы. Для каждой пары чисел *p* и *q* выводим значение s(*p*, *q*).

```
while(scanf("%lld %lld",&p,&q), p + q >= 0)
  printf("%lld\n",s(p,q));
```

2523. Строки Фибоначчи

Последовательность строк Фибоначчи определяется следующим образом: $s_1 = b$, $s_2 = a$, $s_k = s_{k-1} + s_{k-2}$ для $k > 2$. Например, $s_3 = ab$, $s_4 = aba$, $s_5 = abaab$ и т.д.

Даны натуральные числа *n*, *m*, *l*. Вывести подстроку строки s_n, начинающуюся с позиции *m* и имеющую длину *l*.

Вход. Содержит одну строку, в которой находятся три разделённых пробелом натуральных числа *n*, *m* и *l*, где $1 \le n \le 40$; $1 \le m \le$ длина(S_n); $1 \le l \le 1000$.

Выход. Вывести подстроку строки s_n, начинающуюся с позиции *m* и имеющую длину *l* (длина выведенной подстроки может оказаться меньше, если длина оставшейся части строки s_n, начинающейся с позиции *m*, меньше *l*).

Пример входа	**Пример выхода**
5 3 10	aab

Пусть fib(i) – i-ое число Фибоначчи. Используя тот факт, что $s_n = s_{n-1} + s_{n-2}$, будем искать подстроку стоящую на позициях [m … $m + l$] либо в s_{n-1}, либо в s_{n-2}, либо на их пресечении – то есть искомая подстрока является конкатенацией суффикса s_{n-1} и префикса s_{n-2}.

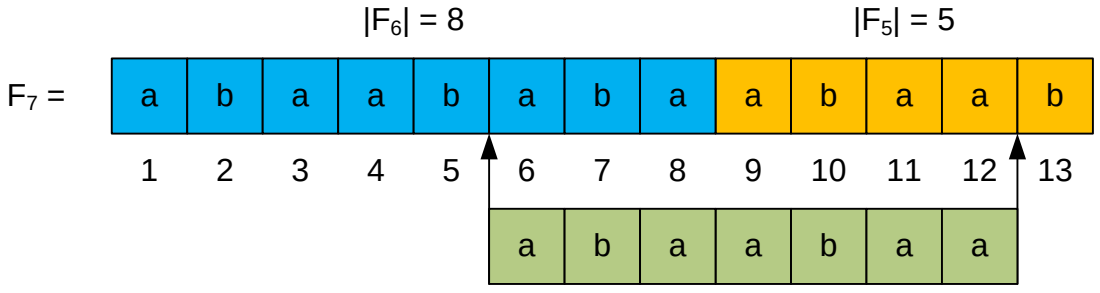
Реализуем функцию solve(n, *Left*, *Right*), которая возвращает подстроку Фибоначчи s_n, лежащую в позициях от *Left* до *Right* включительно.

Если $n = 1$ или $n = 2$, то возвращаем соответствующую строку, состоящую из одного символа ($s_1 = b$, $s_2 = a$).

Поскольку s_n является конкатенацией строк s_{n-1} и s_{n-2}, то попробуем установить, в какой из этих частей (s_{n-1} или s_{n-2}) лежит искомая подстрока. При этом длины s_{n-1} и s_{n-2} нам известны (именно для этого мы посчитаем числа Фибоначчи в массиве fib).

- Если *Right* ≤ fib(n – 1), то искомая подстрока целиком лежит в s_{n-1}. Возвращаем solve(n – 1, *Left*, *Right*).
- Если *Left* > fib(n – 1), то искомая подстрока полностью лежит в s_{n-2}. При этом следует пересчитать индексы искомой подстроки, а именно вычесть из *Left* и *Right* длину s_{n-1}. Возвращаем solve(n – 2, *Left* – fib(n – 1), *Right* – fib(n – 1)).
- Иначе искомая подстрока начинается в s_{n-1} и заканчивается в s_{n-2}. Следует вернуть подстроку из s_{n-1} на позициях [*Left* … fib(n – 1)] и конкатенировать с подстрокой из s_{n-2} на позициях [1 … *Right* – fib(n – 1)]. То есть возвращаем solve(n – 1, *Left*, fib(n – 1)) + solve(n – 2, 1, *Right* – fib(n – 1)).

Пример. Пусть $n = 7$, $m = 6$, $l = 7$.



solve(7, 6, 12) = solve(7 – 1, 6, fib(6)) + solve(7 – 2, 1, 12 – fib(6)) =
= solve(6, 6, 8) + solve(5, 1, 4) = “*aba*” + “*abaa*” = “*abaabaa*”.

Реализация алгоритма

В массив fib занесем числа Фибоначчи: fib[i] содержит длину s_i.

```
#define MAX 44
int fib[MAX] = {0, 1};
```

Реализация функции solve.

```
string solve(int n, int Left, int Right)
{
  if (n == 1) return "b";
  if (n == 2) return "a";
  if (Right <= fib[n-1]) return solve(n-1, Left, Right);
  if (Left > fib[n-1])
    return solve(n-2, Left - fib[n-1], Right - fib[n-1]);
  return solve(n-1, Left, fib[n-1]) + solve(n-2, 1, Right - fib[n-1]);
}
```

Основная часть программы. Вычисляем первые MAX чисел Фибоначчи.

```
for(int i = 2; i < MAX; i++)
  fib[i] = fib[i-1] + fib[i-2];
```

Читаем входные данные. Вычисляем и выводим ответ.

```
scanf("%d %d %d",&n,&m,&l);
string res = solve(n,m,m+l-1);
puts(res.c_str());
```

5491. Муу

Коровы подсели на новую игру в слова, называемую "Муу". В нее играют несколько коров, стоящих в линию. Каждая корова должна назвать одну определенную букву как можно быстрее. Корова, которая ошибется, выбывает из игры.

Последовательность букв в игре Муу бесконечна. Начинается она так:

m o o m o o o m o o m o o o o m o o m o o o m o o m o o o o o

Наилучшим образом последовательность задается рекурсивно: пусть S(0) – слово из 3-х букв "m o o". Последовательность S(k) получается из копии последовательности S(k – 1), слова "m o ... o" с k + 2 буквами o, за которым следует еще одна копия последовательности S(k – 1). Например:

S(0) = "m o o"
S(1) = "m o o m o o o m o o"
S(2) = "m o o m o o o m o o m o o o o m o o m o o o m o o"

Можно заметить, что таким образом строится бесконечно длинная строка, и именно она используется в игре Муу. Бесси, мудрая корова, хочет узнать n – ую букву этой последовательности: какой она будет – "m" или "o"? Помогите ей узнать это!

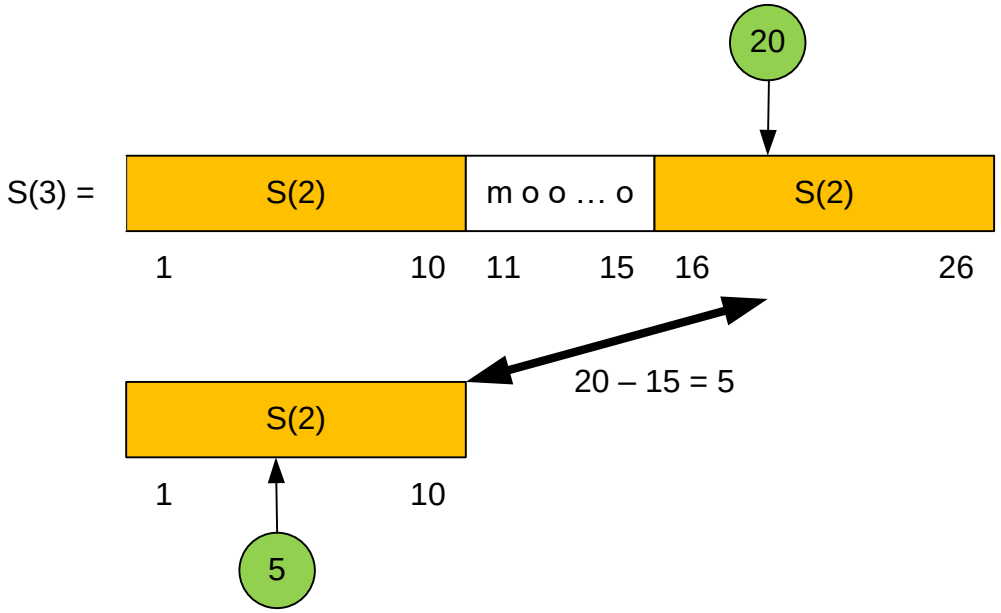
Вход. Одно целое число n ($1 \le n \le 10^9$).

Выход. Одна буква – m или o.

Пример входа	**Пример выхода**
11	m

Ищем наименьшее k, для которого S(k) содержит n-ую букву. Далее определяем, в какой из трех частей S(k) лежит n-ая буква. Она лежит либо во второй части (посередине), либо в третьей (в этом случае положим m равным сумме длин первой и второй частей и запустим поиск (n – m)-ой буквы в S(k – 1), так как третья часть S(k) полностью совпадает с S(k – 1)).

Пример. Пусть необходимо определить n = 20 -ую букву. Поскольку длина S(2) равна 10, а длина S(3) равна 25, то 20-ая буква лежит в S(3).



20-ая буква лежит в третьей части S(3), равной S(2). Следовательно 20-ая буква будет такой же, как и 20 – 15 = 5-ая буква S(2).

Реализация алгоритма

Функция len(k) возвращает длину последовательности S(k).

```
int len(int k)
{
  if (k == -1) return 0;
  int x = len(k - 1);
  return x + k + 3 + x;
}
```

Поиск n-ой буквы в последовательности S(k).

```
char Solve(int n, int k)
{
```

Если длина S(k) меньше чем n, то следует искать n-ую букву в S(k + 1).

```
  if (n > len(k)) return Solve(n,k+1);
```

Если n не больше длины S(k – 1), то следует искать n-ую букву в S(k – 1).

```
  if (n <= len(k-1)) return Solve(n,k-1);
```

Поскольку n > len(k – 1), то n-ую букву следует искать или посередине, или во второй части S(k).

```
  n = n - len(k-1);
```

Проверим, лежит ли n-ая буква в средней части последовательности S(k).

```
  if (n <= k + 3)
    return (n == 1) ? 'm' : 'o';
```

n-ая буква находится в последней части S(k). Это все равно что искать ее в последовательности S(k – 1).

```
  n = n - (k + 3);
  return Solve(n,k-1);
}
```

Основная часть программы. Начинаем искать n-ую букву в S(0).

```
scanf("%d",&n);
printf("%c\n",Solve(n,0));
```

9606. Деление по модулю

Заданы три натуральных числа a, b и n. Вычислите a / b mod n. То есть найдите такое значение x что b * x = a mod n.

Вход. Три натуральных числа a, b, n ($n \le 2 * 10^9$, $1 \le a, b < n$). Известно, что n простое.

Выход. Выведите значение a / b mod n.

Пример входа 1
```
3 4 7
```

Пример выхода 1
```
6
```

Пример входа 2
```
4 8 13
```

Пример выхода 2
```
7
```

Поскольку число n простое, то по теореме Ферма $b^{n-1} \bmod n = 1$ для всякого $1 \le b < n$. Равенство можно переписать в виде $(b * b^{n-2}) \bmod n = 1$, откуда обратным числом к b является $y = b^{n-2} \bmod n$.

Следовательно $a / b \bmod n = a * b^{-1} \bmod n = a * y \bmod n$.

Обратное число можно найти используя *расширенный алгоритм Евклида*. Пусть следует решить сравнение: $ax = 1 \pmod n$. Рассмотрим уравнение

$$ax + ny = 1$$

и найдем его частичное решение (x_0, y_0) при помощи расширенного алгоритма Евклида. Возьмем уравнение $ax_0 + ny_0 = 1$ по модулю n, получим $ax_0 = 1 \pmod n$. В случае отрицательного значения x_0, к нему следует прибавить n. Таким образом $x_0 = a^{-1} \pmod n$ является обратным к a.

Пример. Рассмотрим второй пример. Вычислим $4 / 8 \bmod 13$. Для этого следует решить уравнение $8 * x = 4 \bmod 13$, откуда $x = (4 * 8^{-1}) \bmod 13$.

Число 13 простое, из теоремы Ферма следует что $8^{12} \bmod 13 = 1$ или $(8 * 8^{11}) \bmod 13 = 1$. Следовательно $8^{-1} \bmod 13 = 8^{11} \bmod 13 = 5$.

Вычисляем ответ: $x = (4 * 8^{-1}) \bmod 13 = (4 * 5) \bmod 13 = 20 \bmod 13 = 7$.

Реализация алгоритма

Функция ***powmod*** вычисляет значение $x^n \bmod m$.

```
long long powmod(long long x, long long n, long long m)
{
  if (n == 0) return 1;
  if (n % 2 == 0) return powmod((x * x) % m, n / 2, m);
  return (x * powmod(x, n - 1, m)) % m;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%lld %lld %lld", &a, &b, &n);
```

Вычисляем $y = b^{n-2} \bmod n$, $x = a * y \bmod n$.

```
y = powmod(b, n - 2, n);
x = (a * y) % n;
```

Выводим ответ.

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

9643. Числа Каталана

Числа Каталана c_n задаются рекуррентным соотношением:

$$c_0 = 1,$$
$$c_n = \sum_{k=0}^{n-1} c_k c_{n-k-1}\ ,\ n > 0$$

Вычислите n-ое число Каталана по модулю m.

Вход. Два целых числа n ($0 \le n \le 10^4$) и m ($0 < m \le 10^9$).

Выход. Выведите значение c_n mod m.

Пример входа	**Пример выхода**
5 100	42

Рекуррентное соотношение перепишем в виде:

$$c_0 = 1,$$
$$c_n = c_0c_{n-1} + c_1c_{n-2} + c_2c_{n-3} + \ldots + c_{n-1}c_0, \text{ при } n > 0$$

Например,

- $c_0 = 1$
- $c_1 = c_0c_0 = 1$,
- $c_2 = c_0c_1 + c_1c_0 = 1 + 1 = 2$,
- $c_3 = c_0c_2 + c_1c_1 + c_2c_0 = 2 + 1 + 2 = 5$,
- $c_4 = c_0c_3 + c_1c_2 + c_2c_1 + c_3c_0 = 5 + 2 + 2 + 5 = 14$,
- $c_5 = c_0c_4 + c_1c_3 + c_2c_2 + c_3c_1 + c_4c_0 = 14 + 5 + 4 + 5 + 14 = 42$

Поскольку значение c_n пересчитывается через все предыдущие значения c_0, c_1, c_2, ..., c_{n-1}, то значения чисел Каталана будем запоминать в линейном массиве.

Реализация алгоритма

Объявим массив cat, в котором будем запоминать числа Каталана: cat[i] = c_i.

```
long long cat[10001];
```

Читаем входные данные.

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

Вычисляем числа Каталана по рекуррентной формуле.

```
cat[0] = 1;
for (i = 1; i <= n; i++)
{
  for (j = 0; j < i; j++)
    cat[i] = (cat[i] + cat[j] * cat[i - j - 1]) % m;
}
```

Выводим *n*-ое число Каталана.

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

230. Парад скобок

Посчитайте количество различных правильных скобочных последовательностей, состоящих из k_1 пар скобок первого типа, k_2 пар скобок второго типа, …, k_m пар скобок *m*-го типа. Последовательность скобок считается правильной в следующих случаях:

- Пустая последовательность – правильная;
- Если A и B – правильны, то AB – тоже правильная
- Если A – правильная, то (iA)i – правильная, где (i и)i – открывающая и закрывающая скобка одного типа.

Вход. Первая строка содержит количество тестов $0 < n \leq 1000$. Каждая из следующих *n* строк описывает тест. Каждая строка начинается с числа $0 < m \leq 100$ – количества различных типов скобок. Затем *m* положительных чисел $k_1, k_2, \ldots, k_m$ следуют одно за другим через пробел. Число k_i – это количество пар скобок *i*-го типа. Общее количество пар скобок – не больше 1000.

Выход. Для каждого теста выведите строку, содержащую одно целое число – ответ задачи по модулю 1000000007.

Пример входа	**Пример выхода**
3 1 4 2 2 2 3 1 2 3	14 84 7920

Будем говорить, что скобка имеет *i*-ый тип, если она покрашена краской номер *i*. Пусть изначально все скобки не покрашены. Тогда изначально имеется $s = \sum_{i=1}^{m} k_i$ пар скобок, из которых можно составить Cat(*s*) правильных скобочных последовательностей. Здесь через Cat(*s*) обозначено *s*-ое число Каталана.

Покрасить первой краской мы должны k_1 пару скобок, которые можно выбрать $C_s^{k_1}$ способами. k_2 пары скобок, которые следует покрасить второй краской, можно выбрать $C_{s-k_1}^{k_2}$ способами. И так далее, для покраски *i*-ой краской можно выбрать пары скобок $C_{s-k_1-\ldots-k_{i-1}}^{k_i}$ способами.

Количество искомых различных правильных скобочных последовательностей равно произведению

$$\text{Cat}(s) * C_s^{k_1} * C_{s-k_1}^{k_2} * \ldots * C_{s-k_1-\ldots-k_{i-1}}^{k_i} * \ldots * C_{s-k_1-\ldots-k_{m-1}}^{k_m},$$

взятому по модулю1000000007 . Заметим, что последний множитель в произведении равен 1, так как

$$C_{s-k_1-\ldots-k_{m-1}}^{k_m} = C_{k_m}^{k_m} = 1$$

Пример.

Для второго теста $s = 4$ и ответ равен Cat(4) * C_4^2 * C_2^2 = 14 * 6 * 1 = 84.

Для третьего теста $s = 6$ и ответ равен Cat(6) * C_6^1 * C_5^2 * C_3^3 = 132 * 6 * 10 * 1 = 7920.

Реализация алгоритма

Объявим макросы и необходимые массивы. В ячейках массива cat будем вычислять числа Каталана, в ячейках cnk – биномиальные коэффициенты. Входные значения k_i запоминаем в массиве k.

```
#define MD 1000000007
#define MAX 1001
long long k[MAX], cat[MAX];
long long cnk[MAX][MAX];
```

Функция c вычисляет значение биномиального коэффициента C_n^k и запоминает его в ячейке cnk[n][k].

```
long long c(int n, int k)
{
```

Если cnk[n][k] > 0, то значение C_n^k было вычислено ранее. Следует его просто вернуть.

```
  if (cnk[n][k] > 0) return cnk[n][k];
```

Поскольку $C_n^k = C_n^{n-k}$, то если при $n - k < k$ вместо C_n^k находить C_n^{n-k}, то время вычисления биномиального коэффициента уменьшится.

```
  if (n - k < k) return c(n,n-k);
```

Если $k = 0$, то значение C_n^0 равно 1.

```
  if (!k) return cnk[n][k] = 1;
```

Для вычисления биномиального коэффициента используем рекуррентное соотношение

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$$

```
  return cnk[n][k] = (c(n-1,k) + c(n-1,k-1)) % MD;
}
```

Основная часть программы. В массиве cat вычисляем числа Каталана исходя из рекуррентной формулы $cat_i = \sum_{j=0}^{i-1} c_j c_{i-j-1}$. Вычисления проводим по модулю MD = 1000000007.

```
cat[0] = cat[1] = 1;
for(i = 2; i < 1001; i++)
{
  for(j = 0; j < i; j++)
    cat[i] = (cat[i] + cat[j] * cat[i-j-1]) % MD;
}
```

Читаем входные данные. В переменной *s* вычисляем сумму $\sum_{i=1}^{m} k_i$.

```
scanf("%d",&n);
while(n--)
{
  scanf("%d",&m);
  for(s = 0, i = 1; i <= m; i++)
    scanf("%d",&k[i]), s += k[i];
```

Положим сначала результат *res* равным Cat(*s*).

```
  res = cat[s];
```

Умножаем значение *res* на $C_{s-k_1-\ldots-k_{i-1}}^{k_i}$ для всех *i* от 1 до *m*.

```
  for(i = 1; i <= m; i++)
    res = (res * c(s,k[i])) % MD, s -= k[i];
```

Выводим результат.

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

1592. Мост

n человек подошли ночью к реке. Имеется узкий мост, который одновременно может вместить только двух человек. У них есть один фонарик, и, поскольку сейчас ночь, при переходе через мост его необходимо использовать. Движение по мосту без фонаря запрещено.

Для каждого человека известно время, за которое он пересекает мост. Если по мосту двигаются двое, то время их передвижения равно времени более медленного. Необходимо найти минимальное время, за которое все *n* людей перейдут на другой берег реки, а также последовательность переходов, как указано в примере.

Вход. Состоит из нескольких тестов. Первая строка каждого теста содержит количество людей n ($n \le 1000$), а вторая строка содержит n чисел – время перехода людей через мост. Время перехода каждого человека через мост не более 100 секунд.

Выход. Для каждого теста вывести следующую информацию. Первая строка содержит минимальное время, за которое могут пересечь мост n людей. Далее следует стратегия пересечения моста людьми. Каждая строка стратегии содержит одно или два числа, характеризующие людей, которые с фонарем пересекают мост. При существовании нескольких оптимальных стратегий пересечения реки вывести любую.

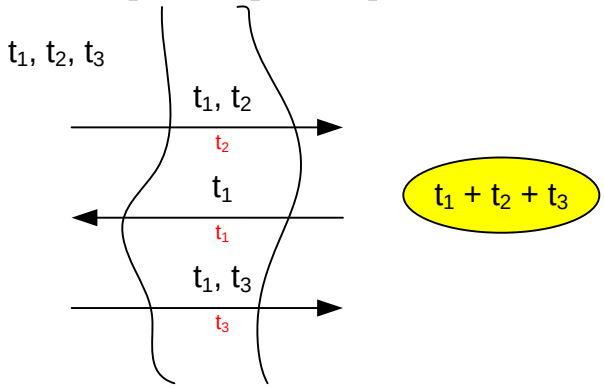
Пример входа

```
4
1 2 5 10
3
1 2 3
```

Пример выхода

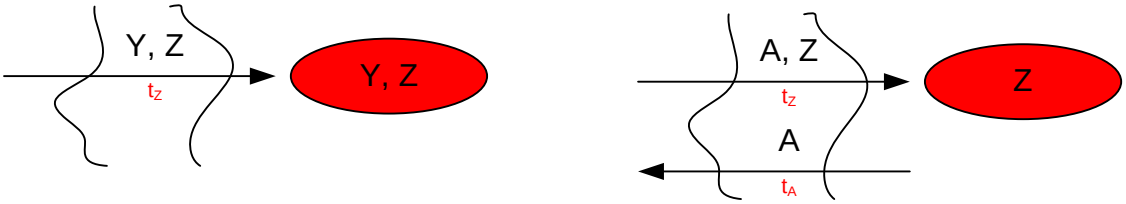
```
17
1 2
1
5 10
2
1 2
6
1 2
1
1 3
```

Отсортируем время, за которое люди пересекают реку, по возрастанию. Обозначим через t_i время пересечения реки i – ым человеком ($t_1 \le t_2 \le \ldots \le t_n$). Рассмотрим, как следует пересекать мост одному, двум или трем людям. При $n = 1$ и $n = 2$ оптимальная скорость пересечения реки соответственно равна t_1 и $t_2 = \max(t_1, t_2)$ (скорость передвижения двух людей равна скорости медленного). При наличии трех людей ($n = 3$) первый и второй перебираются на другую сторону, самый быстрый (первый) возвращается с фонарем назад и переводит третьего. Таким образом, оптимальное время перехода равно $t_1 + t_2 + t_3$.



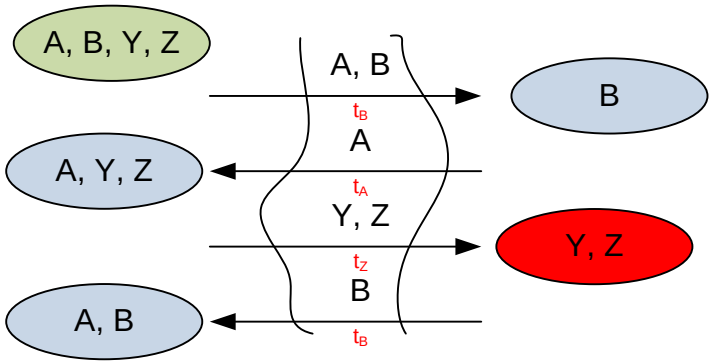
Рассмотрим случай, когда $n > 3$. Пусть A ≤ B ≤ … ≤ Y ≤ Z – люди, отсортированные по возрастанию времени пересечения моста (A – самый быстрый, Z – самый медленный). Пусть J – это человек, с которым перемещается

Z. Если J остается на другом берегу и никогда больше не пойдет по мосту, то оптимально выбрать его равным Y. Если J будет возвращаться, то его оптимально выбрать самым быстрым, то есть A. Таким образом Z может пересекать мост или с Y, или с A.

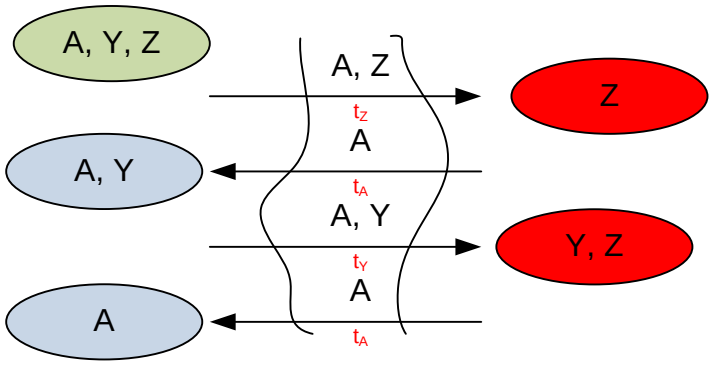


Вычислим время перевода двух самых медленных людей (Y и Z) согласно этим двум стратегиям.

1. Z идет с Y. Но тогда до этого там должен быть кто-то, кто вернет фонарь, например K. Но этого K также на другой берег должен был кто-то провести чтобы, вернув фонарь, отдать его Y и Z. Пусть им будет L. Таким образом, K и L должны возвращаться. Для минимизации времени в качестве K и L следует выбрать двух самых быстрых, то есть A и B. Время перевода Y и Z равно $t_A + 2t_B + t_Z$.



2. Z пересекает мост с A, A возвращается. Далее A переводит Y и также возвращается. Время, за которое Y и Z перейдут на другой берег, равно $2t_A + t_Y + t_Z$.



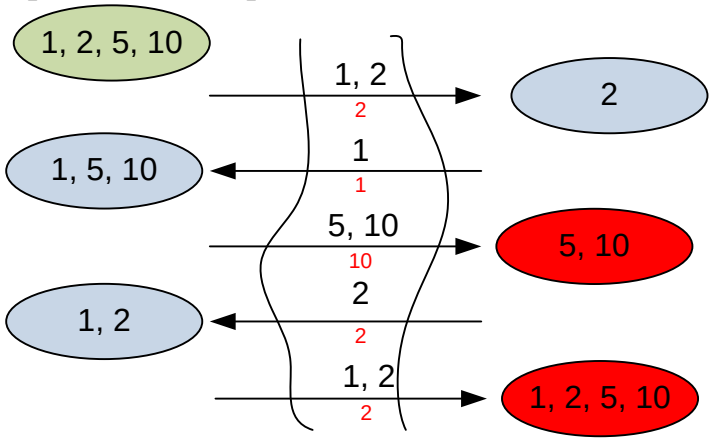
В обоих случаях через мост переводятся только двое самых медленных людей. Стратегия (первая или вторая) выбирается в зависимости от того, какое из значений ($t_A + 2t_B + t_Z$ или $2t_A + t_Y + t_Z$) меньше. Если изначально следовало перевести n людей, то далее рекурсивно следует перевести $n - 2$ людей.

Пример. Отсортируем время пересечения моста людьми: 1, 2, 5, 10. Здесь $t_A = 1$, $t_B = 2$, $t_Y = 5$, $t_Z = 10$. Время перевода двух самых медленных людей согласно первой и второй стратегиям соответственно равны

- $t_A + 2t_B + t_Z = 1 + 2 * 2 + 10 = 15$;
- $2t_A + t_Y + t_Z = 2 * 1 + 5 + 10 = 17$;

Поскольку первое значение меньше, то следует Z вести с Y. Z с Y пересекут мост за время 15, после чего остается перевести на другой берег A и B. Это делается за время $\max\{t_A, t_B\} = 2$.

Общее время перехода моста равно 15 + 2 = 17.



Реализация алгоритма

В массиве m хранится время перехода людей через реку.

```
int m[1001];
```

Функция go(*n*, *visible*) возвращает оптимальное время, за которое могут пересечь реку *n* людей. Переменная *visible* = 1, если следует выводить на экран саму стратегию перехода, и *visible* = 0 иначе.

```
int go(int n,int visible)
{
  int First, Second, Best;
```

Случай пересечения реки одним человеком.

```
  if (n == 1)
  {
    if (visible) printf("%d\n",m[0]);
    return m[0];
  } else
```

Случай пересечения реки двумя людьми

```
  if (n == 2)
  {
    if (visible) printf("%d %d\n",m[0],m[1]);
    return m[1];
  } else
```

Случай пересечения реки тремя людьми

```
  if (n == 3)
  {
    if (visible)
    {
      printf("%d %d\n",m[0],m[1]);
      printf("%d\n",m[0]);
      printf("%d %d\n",m[0],m[2]);
    }
    return m[0] + m[1] + m[2];
  };
```

Вычисляем оптимальное время *First* и *Second*, которое получается при использовании двух выше описанных стратегий.

```
  First = m[0] + 2 * m[1] + m[n-1];
  Second = 2 * m[0] + m[n-2] + m[n-1];
  Best = (First < Second) ? First : Second;
  if (visible)
  {
    if (Best == First)
    {
      printf("%d %d\n",m[0],m[1]);
      printf("%d\n",m[0]);
      printf("%d %d\n",m[n-2],m[n-1]);
      printf("%d\n",m[1]);
    } else
    {
      printf("%d %d\n",m[0],m[n-2]);
      printf("%d\n",m[0]);
      printf("%d %d\n",m[0],m[n-1]);
      printf("%d\n",m[0]);
    }
  }
```

Рекурсивно вычисляем оптимальную стратегию для оставшихся $n - 2$ людей.

```
  return Best + go(n-2,visible);
}
```

Основная часть программы. Читаем количество тестов, вводим время, за которое люди пересекают мост в массив m.

```
while(scanf("%d",&n) == 1)
{
  for(i = 0; i < n; i++) scanf("%d",&m[i]);
```

Сортируем время пересечения реки людьми по возрастанию.

```
  sort(m,m+n);
```

Запускаем функцию ***go*** с параметром *visible* = 0, которая возвращает оптимальное время пересечения реки. Выводим его, после чего снова запускаем

функцию ***go*** с параметром *visible* = 1, которая печатает последовательность переходов.

```
  res = go(n,0);
  printf("%d\n",res);
  res = go(n,1);
}
```

7747. Городской горизонт

Фермер Джон приехал со своими коровами в город! Во время захода солнца коровы смотрят на городской горизонт и наблюдают красивые силуэты, образованные прямоугольными зданиями.

Весь горизонт представлен числовой прямой с n ($1 \le n \le 40000$) зданиями. Силуэт i-го здания распростерся вдоль горизонта между точками a_i и b_i ($1 \le a_i < b_i \le 10^9$) и имеет высоту h_i ($1 \le h_i \le 10^9$). Определите площадь в квадратных единицах общего силуэта, образованного всеми n зданиями.

Вход. Первая строка содержит целое число n. Каждая из следующих n строк описывает здание i тремя целыми числами: a_i, b_i и h_i.

Выход. Выведите общую площадь городского силуэта в квадратных единицах, образованного всеми n зданиями.

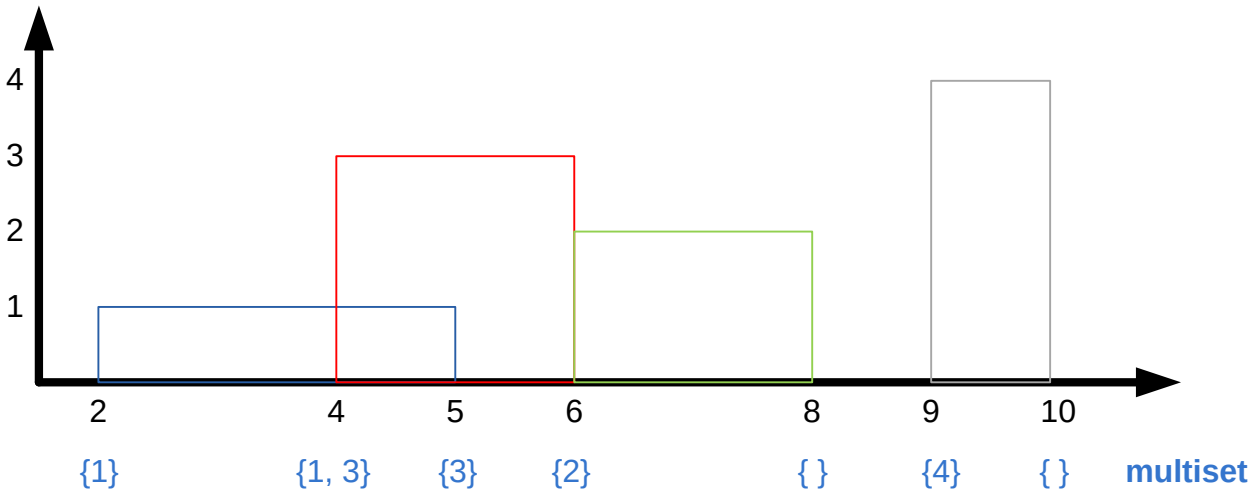
Пример входа	**Пример выхода**
4 2 5 1 9 10 4 6 8 2 4 6 3	16

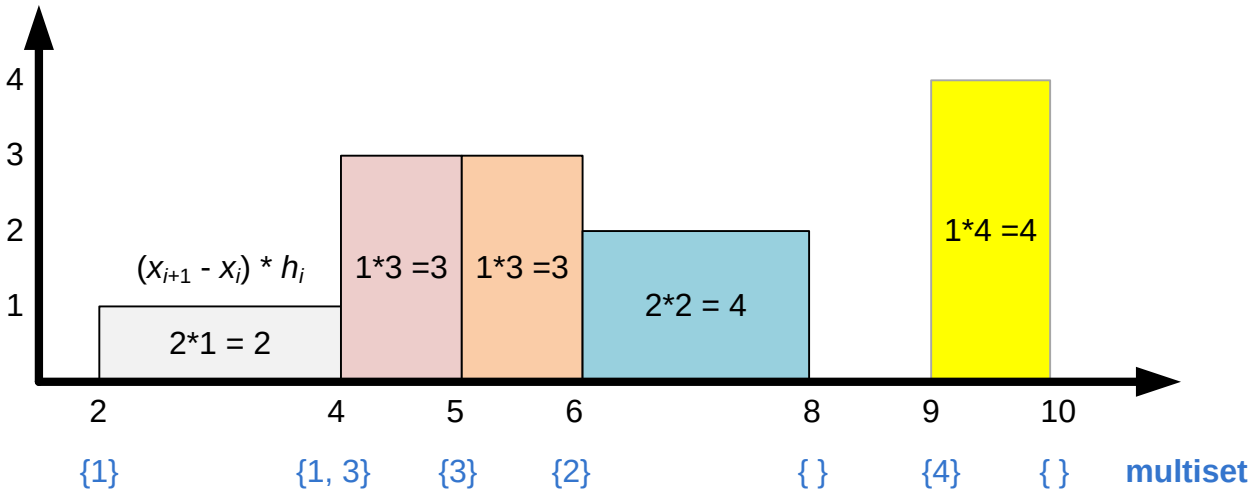
Отсортируем точки – абсциссы начал и концов зданий по возрастанию. С каждой точкой запомним, является ли она началом (*type* = 0) или концом (*type* = 1) здания, а также высоту самого здания. Будем последовательно обрабатывать точки слева направо.

В мультимножестве s храним высоты зданий, которые на данный момент уже начались, но еще не закончились. Мультимножество будет поддерживать высоты по убыванию.

Пусть на данный момент мы находимся в точке x_i. Пусть h – наибольшее число в мультимножестве. Это значит, что текущее (на интервале $[x_i, x_{i+1}]$) самое высокое здание имеет высоту h (которое еще не закончилось). Добавим к площади горизонта значение $(x_{i+1} - x_i) * h$. Если точка x_i – начало здания, то соответствующую ей высоту добавим в мультимножество. Если точка x_i конец здания, то удалим из мультимножества его высоту. Порядок обработки начал и концов зданий с одинаковыми абсциссами не имеет значения.

Пример





Реализация алгоритма

Для точки объявим структуру ***node***, которая будет хранить ее абсциссу *x*, информацию является ли она началом (*type* = 0) или концом (*type* = 1) отрезка, а также высоту *h* соответствующего здания.

```
struct node
{
  int x, type, h;
  node(int x, int type, int h) : x(x), type(type), h(h) {};
};
```

Определим компаратор ***f*** для точек – сортировать их будем по абсциссам.

```
int f(node a, node b)
{
  return a.x < b.x;
}
```

Объявим массив точек Event, а также мультимножество s для хранения высот зданий, пересекающихся с текущей абсциссой сканирующей прямой.

```
vector<node> Event;
multiset<int, greater<int> > s;
```

Читаем входные данные. Строим массив точек.

```
scanf("%d",&n);
for(i = 0; i < n; i++)
{
  scanf("%d %d %d",&left,&right,&height);
  Event.push_back(node(left,0,height));
  Event.push_back(node(right,1,height));
}
```

Сортируем массив точек по неубыванию абсцисс.

```
sort(Event.begin(),Event.end(),f);
```

В переменной *res* подсчитываем общую площадь городского силуэта.

```
res = 0;
```

Двигаемся слева направо по точкам массива Event. Для каждого значения *i* ***for*** цикла рассматриваем инервал (Event[*i*].*x*, Event[*i* + 1].*x*).

```
for (i = 0; i < Event.size() - 1; i++)
{
```

Если точка x_i – начало здания, то соответствующую ей высоту добавим в мультимножество.

```
  int h = Event[i].h;
  if (Event[i].type == 0) s.insert(h);
```

Если точка x_i – конец здания, то удалим из мультимножества соответствующую ей высоту.

```
  else s.erase(s.find(h));
```

Если стек не пустой, то между точками Event[*i*].*x* и Event[*i* + 1].*x* имеется линия горизонта, высота которой равна наибольшему значению в мультимножестве s, а именно значению *s.begin().

```
  if (!s.empty())
    res += 1LL * *s.begin() * (Event[i+1].x - Event[i].x);
}
```

Выводим искомую площадь.

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

2299. Теодор Рузвельт

"Теодор Рузвельт" – флагман военно-морского флота Кукуляндии. Заклятые враги кукуляндцев, флатландцы, решили уничтожить его. Они узнали, что "Теодор Рузвельт" представляет собой выпуклый многоугольник из n вершин и узнали его координаты. Затем они выпустили m баллистических ракет и определили координаты точек, где эти ракеты взорвались. По расчетам штаба флатландцев, "Теодор Рузвельт" будет уничтожен, если в него попадёт хотя бы k ракет. Вычислите, удалось ли флатландцам уничтожить корабль.

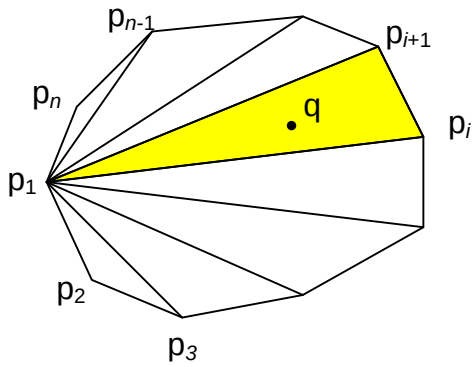
Вход. В первой строке записаны целые числа n, m, k ($3 \le n \le 10^5$, $0 \le k \le m \le 10^5$). В следующих n строках записаны координаты вершин многоугольника в порядке обхода против часовой стрелки. В следующих m строках записаны координаты точек. Гарантируется, что все координаты – целые числа, не превосходящие по модулю 10^9.

Выход. Выведите YES, если в многоугольнике лежит по крайней мере k точек, и NO в противном случае.

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

Рассмотрим, как решить задачу принадлежности точки выпуклому многоугольнику.

Координаты вершин многоугольника заданы в порядке обхода против часовой стрелки: p_1, p_2, …, p_n. Рассмотрим первую точку многоугольника p_1. Относительно нее все остальные точки будут отсортированы по углу. Проведём от точки p_1 все лучи, содержащие диагонали. Бинарным поиском ищем такое ребро p_ip_{i+1}, что повороты p_1p_iq и $p_1p_{i+1}q$ различны. Таким образом мы найдем область, в которой лежит точка q.



Когда найден угол $p_ip_1p_{i+1}$, в котором находится точка q, за константное время можно проверить, с одной ли стороны от прямой p_ip_{i+1} лежат точки p_1 и q. Для этого поворот $p_ip_{i+1}q$ должен быть левым (поворот $p_ip_{i+1}p_1$ является левым, так как исходные точки многоугольника заданы в порядке обхода против часовой стрелки).

В начале алгоритма следует проверить, не является ли поворот p_1p_nq левым, или поворот p_1p_2q правым. Если выполняется хотя бы одно из этих условий, то q лежит вне многоугольника.

Направление поворота. Пусть совершается движение от точки A до B, затем от B до C. При движении имеет место левый поворот (движение происходит против часовой стрелки), если AB × BC > 0 (псевдоскалярное произведение векторов положительно) и правый, если AB × BC < 0.

Реализация алгоритма

Объявим структуру точка и массив точек.

```
#define MAX 100001
struct Point
{
  long long x, y;
  Point(long long x = 0, long long y = 0) : x(x), y(y) {}
};
Point p[MAX];
```

Функция angle определяет левый или правый поворот происходит при движении по точкам A , B, C. Вычислим вектора:

$$AB = (b.x - a.x,\ b.y - a.y)$$
$$BC = (c.x - b.x,\ c.y - b.y)$$

Псевдоскалярное произведение AB x BC равно

$$\begin{vmatrix} b.x - a.x & b.y - a.y \\ c.x - b.x & c.y - b.y \end{vmatrix} = (b.x - a.x) * (c.y - b.y) - (b.y - a.y) * (c.x - b.x)$$

и его знак определяет направление поворота:

- положительный – левый поворот;
- отрицательный – правый поворот;

```
int angle(Point a, Point b, Point c)
{
  long long q = (b.x - a.x) * (c.y - b.y) - (b.y - a.y) * (c.x - b.x);
  return (q > 0) - (q < 0); // sgn(q)
}
```

Функция inside возвращает *true*, если точка q лежит внутри многоугольника.

```
bool inside(Point q)
{
```

Первую точку выберем базовой.

```
  Point a = p[1];
```

Если поворот p_1p_nq левый, или поворот p_1p_2q правый, то точка q лежит вне угла $p_np_1p_2$ и соответственно вне многоугольника.

```
  if (angle(a, p[2], q) < 0 || angle(a, p[n], q) > 0) return false;
```

Стартуем бинарный поиск на отрезке $[l; r] = [2; n]$. Ищем такие точки p_l и p_r ($r = l + 1$) что точка q лежит внутри угла $p_lp_1p_r$.

```
  int l = 2, r = n, m;
  while (l + 1 < r)
  {
    m = (l + r) / 2;
    if (angle(a, p[m], q) < 0)
      r = m;
    else
      l = m;
  }
  return angle(p[l], p[r], q) >= 0;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d %d", &n, &m, &k);
for (int i = 1; i <= n; i++)
{
  scanf("%lld %lld", &x, &y);
  p[i] = Point(x, y);
}
```

Вычисляем количество точек, лежащих внутри многоугольника.

```
res = 0;
for (int i = 1; i <= m; ++i)
{
  scanf("%lld %lld", &x, &y);
  if (inside(Point(x, y))) res++;
}
```

Выводим ответ.

```
if (res < k) puts("NO"); else puts("YES");
```