SPOJ Problem Set (regionais)

1893. Onde Estao as Bolhas?

Problem code: BOLHAS

Uma das operaçoes mais freqüentes em computaçao é ordenar uma seqüencia de objetos. Por- tanto, nao é surpreendente que essa operaçao seja também uma das mais estudadas.

Um algoritmo bem simples para ordenaçao é chamado *Bubblesort*. Ele consiste de vários turnos. A cada turno o algoritmo simplesmente itera sobre a seqüencia trocando de posiçao dois elementos consecutivos se eles estiverem fora de ordem. O algoritmo termina quando nenhum elemento trocou de posiçao em um turno.

O nome *Bubblesort* (ordenaçao das bolhas) deriva do fato de que elementos menores ("mais leves") movem-se na direçao de suas posiçoes finais na seqüencia ordenada (movem-se na direçao do início da seqüencia) durante os turnos, como bolhas na água. A figura abaixo mostra uma implementaçao do algoritmo em pseudo-código:

```
Para i variando de 1 a N faça
        Para j variando de N - 1 a i faça
                Se seq[j - 1] > seq[j] entao
                        Intercambie os elementos seq[j - 1] e seq[j]
                Fim-Se
        Fim-Para
        Se nenhum elemento trocou de lugar entao
                Final do algoritmo
        Fim-Se
Fim-Para
```

Por exemplo, ao ordenar a seqüencia `[5, 4, 3, 2, 1]` usando o algoritmo acima, quatro turnos sao necessários. No primeiro turno ocorrem quatro intercâmbios: 1 x 2, 1 x 3, 1 x 4 e 1 x 5; no segundo turno ocorrem tres intercâmbios: 2 x 3, 2 x 4 e 2 x 5; no terceiro turno ocorrem dois intercâmbios: 3 x 4 e 3 x 5; no quarto turno ocorre um intercâmbio: 4 x 5; no quinto turno nenhum intercâmbio ocorre e o algoritmo termina.

Embora simples de entender, provar correto e implementar, o algoritmo *bubblesort* é muito ineficiente: o número de comparaçoes entre elementos durante sua execuçao é, em média, diretamente proporcional a N^2, onde N é o número de elementos na seqüencia.

Voce foi requisitado para fazer uma "engenharia reversa" no *bubblesort*, ou seja, dados o comprimento da seqüencia, o número de turnos necessários para a ordenaçao e o número de intercâmbios ocorridos em cada turno, seu programa deve descobrir uma possível seqüencia que, quando ordenada, produza exatamente o mesmo número de intercâmbios nos turnos.

Entrada

A entrada contém vários casos de teste. A primeira linha de um caso de teste contém dois inteiros N e M que indicam respectivamente o número de elementos `(1 <= N <= 100.000)` na seqüencia que está sendo ordenada, e o número de turnos `(0 <= M <= 100.000)` necessários para ordenar a seqüencia usando bubblesort. A segunda linha de um caso de teste contém M inteiros X_i, indicando o

número de intercâmbios em cada turno i ($1 <= X_i <= N - 1$, para $1 <= i <= M$).

O final da entrada é indicado por $N = M = 0$.

A entrada deve ser lida da entrada padrao.

Saída

Para cada caso de teste da entrada seu programa deve produzir uma linha na saída, contendo uma permutaçao dos números {1, 2, . . . , N }, que quando ordenada usando *bubblesort* produz o mesmo número de intercâmbios no mesmo número de turnos especificados na entrada. Ao imprimir a permutaçao, deixe um espaço em branco entre dois elementos consecutivos. Se mais de uma permutaçao existir, imprima a maior na ordem lexicográfica padrao para seqüencias de números (a ordem lexicográfica da permutaçao $a_1, a_2, . . . a_N$ é maior do que a da permutaçao $b_1, b_2, . . . b_N$ se para algum $1 <= i <= N$ temos $a_i > b_i$ e o prefixo $a_1, a_2, . . . a_{i-1}$ é iqual ao prefixo $b_1, b_2, . . . b_{i-1}$).

Em outras palavras, caso exista mais de uma soluçao, imprima aquela onde o primeiro elemento da permutaçao é o maior possível. Caso exista mais de uma soluçao satisfazendo essa restriçao, imprima, dentre estas, aquela onde o segundo elemento é o maior possível. Caso exista mais de uma soluçao satisfazendo as duas restriçoes anteriores, imprima, dentre estas, a soluçao onde o terceiro elemento é o maior possível, e assim sucessivamente.

Para toda entrada haverá pelo menos uma permutaçao soluçao.

A saída deve ser escrita na saída padrao.

Exemplo

Entrada:

```
3 1
1
5 4
4 3 2 1
6 5
2 2 2 2 1
0 0
```

Saída:

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

Added by: Wanderley Guimaraes
Date: 2007-10-11
Time limit: 6s
Source limit:50000B
Languages: All

FLOOD

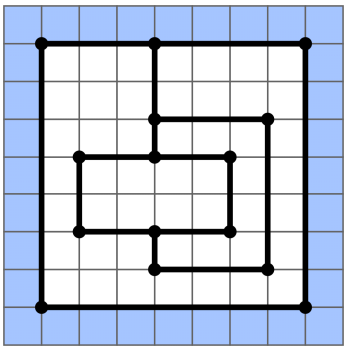
In 1964 a catastrophic flood struck the city of Zagreb. Many buildings were completely destroyed when the water struck their walls. In this task, you are given a simplified model of the city before the flood and you should determine which of the walls are left intact after the flood.

The model consists of N points in the coordinate plane and W walls. **Each wall connects a pair of points and does not go through any other points**. The model has the following additional properties:

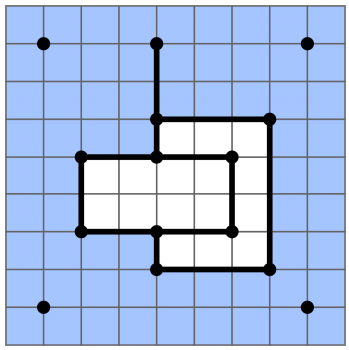
- No two walls intersect or overlap, but they may touch at endpoints;
- Each wall is parallel to either the horizontal or the vertical coordinate axis.

Initially, the entire coordinate plane is dry. At time zero, water instantly floods the exterior (the space not bounded by walls). After exactly one hour, every wall with water on one side and air on the other breaks under the pressure of water. Water then floods the new area not bounded by any standing walls. Now, there may be new walls having water on one side and air on the other. After another hour, these walls also break down and water floods further. This procedure repeats until water has flooded the entire area.

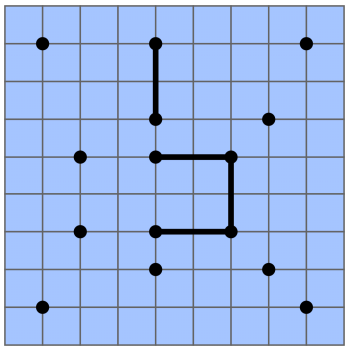
An example of the process is shown in the following figure.



The state at time zero. Shaded cells represent the flooded area, while white cells represent dry area (air).



The state after one hour.



The state after two hours. Water has flooded the entire area and the 4 remaining walls cannot be broken down.

TASK

Write a program that, given the coordinates of the N points, and the descriptions of W walls connecting these points, determines which of the walls are left standing after the flood.

INPUT

The first line of input contains an integer N ($2 \leq N \leq 100\,000$), the number of points in the plane.

Each of the following N lines contains two integers X and Y (both between 0 and 1 000 000, inclusive), the coordinates of one point. The points are numbered 1 to N in the order in which they are given. No two points will be located at the same coordinates.

The following line contains an integer W ($1 \leq W \leq 2N$), the number of walls.

Each of the following W lines contains two different integers A and B ($1 \leq A \leq N$, $1 \leq B \leq N$), meaning that, before the flood, there was a wall connecting points A and B. The walls are numbered 1 to W in the order in which they are given.

OUTPUT

The first line of output should contain a single integer K, the number of walls left standing after the flood.

The following K lines should contain the indices of the walls that are still standing, one wall per line. The indices may be output in any order.

GRADING

In test cases worth a total of 40 points, all coordinates will be at most 500.

In those same cases, and cases worth another 15 points, the number of points will be at most 500.

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 10 submissions for this task to be evaluated (as soon as possible) on part of the official test data. After the evaluation is done, a summary of the results will be available on the contest system.

EXAMPLE

input

```
15
1 1
8 1
4 2
7 2
2 3
4 3
6 3
2 5
4 5
6 5
4 6
7 6
1 8
4 8
8 8
17
1 2
2 15
15 14
14 13
13 1
14 11
11 12
12 4
4 3
3 6
6 5
5 8
8 9
9 11
9 10
10 7
7 6
```

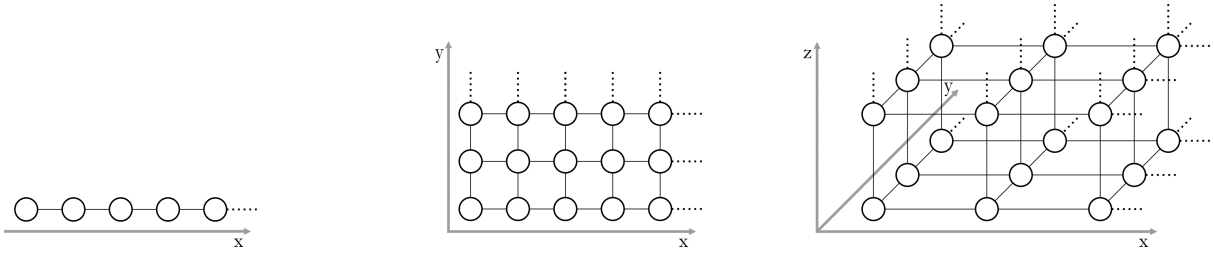
output

```
4
6
15
16
17
```

This example corresponds to the figure on the previous page.

PAIRS

Mirko and Slavko are playing with toy animals. First, they choose one of three boards given in the figure below. Each board consists of cells (shown as circles in the figure) arranged into a one, two or three dimensional grid.



Board 1 Board 2 Board 3

Mirko then places N little **toy animals** into the cells.

The **distance** between two cells is the smallest number of moves that an animal would need in order to reach one cell from the other. In one move, the animal may step into one of the adjacent cells (connected by line segments in the figure).

Two animals can hear each other if the distance between their cells is **at most D**. Slavko's task is to calculate how many **pairs of animals** there are such that one animal can hear the other.

TASK

Write a program that, given the board type, the locations of all animals, and the number D, finds the desired number of pairs.

INPUT

The first line of input contains four integers in this order:

- The board type B ($1 \leq B \leq 3$);
- The number of animals N ($1 \leq N \leq 100\,000$);
- The largest distance D at which two animals can hear each other ($1 \leq D \leq 100\,000\,000$);
- The size of the board M (the largest coordinate allowed to appear in the input):
  - When B=1, M will be at most 75 000 000.
  - When B=2, M will be at most 75 000.
  - When B=3, M will be at most 75.

Each of the following N lines contains B integers separated by single spaces, the coordinates of one toy animal. Each coordinate will be between 1 and M (inclusive).

More than one animal may occupy the same cell.

OUTPUT

Output should consist of a single integer, the number of pairs of animals that can hear each other.

Note: use a 64-bit integer type to calculate and output the result (`long long` in C/C++, `int64` in Pascal).

GRADING

In test cases worth a total of 30 points, the number of animals N will be at most 1 000.

Furthermore, for each of the three board types, a solution that correctly solves all test cases of that type will be awarded at least 30 points.

EXAMPLES

input

```
1 6 5 100
25
50
50
10
20
23
```

output

```
4
```

input

```
2 5 4 10
5 2
7 2
8 4
6 5
4 4
```

output

```
8
```

input

```
3 8 10 20
10 10 10
10 10 20
10 20 10
10 20 20
20 10 10
20 10 20
20 20 10
20 20 20
```

output

```
12
```

Clarification for the leftmost example. Suppose the animals are numbered 1 through 6 in the order in which they are given. The four pairs are:

- 1-5 (distance 5)
- 1-6 (distance 2)
- 2-3 (distance 0)
- 5-6 (distance 3)

Clarification for the middle example. The eight pairs are:

- 1-2 (distance 2)
- 1-4 (distance 4)
- 1-5 (distance 3)
- 2-3 (distance 3)
- 2-4 (distance 4)
- 3-4 (distance 3)
- 3-5 (distance 4)
- 4-5 (distance 3)

2238 - Fixed Partition Memory Management

World Finals - Vancouver - 2000/2001

A technique used in early multiprogramming operating systems involved partitioning the available primary memory into a number of regions with each region having a fixed size, different regions potentially having different sizes. The sum of the sizes of all regions equals the size of the primary memory.

Given a set of programs, it was the task of the operating system to assign the programs to different memory regions, so that they could be executed concurrently. This was made difficult due to the fact that the execution time of a program might depend on the amount of memory available to it. Every program has a minimum space requirement, but if it is assigned to a larger memory region its execution time might increase or decrease.

In this program, you have to determine optimal assignments of programs to memory regions. Your program is given the sizes of the memory regions available for the execution of programs, and for each program a description of how its running time depends on the amount of memory available to it. Your program has to find the execution schedule of the programs that minimizes the average turnaround time for the programs. An execution schedule is an assignment of programs to memory regions and times, such that no two programs use the same memory region at the same time, and no program is assigned to a memory region of size less than its minimum memory requirement. The turnaround time of the program is the difference between the time when the program was submitted for execution (which is time zero for all programs in this problem), and the time that the program completes execution.

Input

The input data will contain multiple test cases. Each test case begins with a line containing a pair of integers *m* and *n*. The number *m* specifies the number of regions into which primary memory has been partitioned ($1 \le m \le 10$), and *n* specifies the number of programs to be executed ($1 \le n \le 50$).

The next line contains *m* positive integers giving the sizes of the *m* memory regions. Following this are *n* lines, describing the time-space tradeoffs for each of the *n* programs. Each line starts with a positive integer *k* ($k \le 10$), followed by *k* pairs of positive integers $s_1,t_1,s_2,t_2,\ \ ,s_k,t_k$, that satisfy $s_i < s_{i+1}$ for $1 \le i < k$. The minimum space requirement of the program is s_1, i.e. it cannot run in a partition of size less than this number. If the program runs in a memory partition of size *s*, where $s_i \le s < s_{i+1}$ for some *i*, then its execution time will be ti. Finally, if the programs runs in a memory partition of size sk or more, then its execution time will be tk.

A pair of zeroes will follow the input for the last test case.

You may assume that each program will execute in exactly the time specified for the given region size, regardless of the number of other programs in the system. No program will have a memory requirement larger than that of the largest memory region.

Output

For each test case, first display the case number (starting with 1 and increasing sequentially). Then print the minimum average turnaround time for the set of programs with two digits to the right of the decimal point. Follow this by the description of an execution schedule that achieves this average turnaround time. Display one line for each program, in the order they were given in the input, that identifies the program number, the region in which it was executed (numbered in the order given in the input), the time when the program started

execution, and the time when the program completed execution. Follow the format shown in the sample output, and print a blank line after each test case.

If there are multiple program orderings or assignments to memory regions that yield the same minimum average turnaround time, give one of the schedules with the minimum average turnaround time.

Sample Input

```
2 4
40 60
1 35 4
1 20 3
1 40 10
1 60 7
3 5
10 20 30
2 10 50 12 30
2 10 100 20 25
1 25 19
1 19 41
2 10 18 30 42
0 0
```

Sample Output

```
Case 1
Average turnaround time = 7.75
Program 1 runs in region 1 from 0 to 4
Program 2 runs in region 2 from 0 to 3
Program 3 runs in region 1 from 4 to 14
Program 4 runs in region 2 from 3 to 10

Case 2
Average turnaround time = 35.40
Program 1 runs in region 2 from 25 to 55
Program 2 runs in region 2 from 0 to 25
Program 3 runs in region 3 from 0 to 19
Program 4 runs in region 3 from 19 to 60
Program 5 runs in region 1 from 0 to 18
```

Vancouver 2000-2001

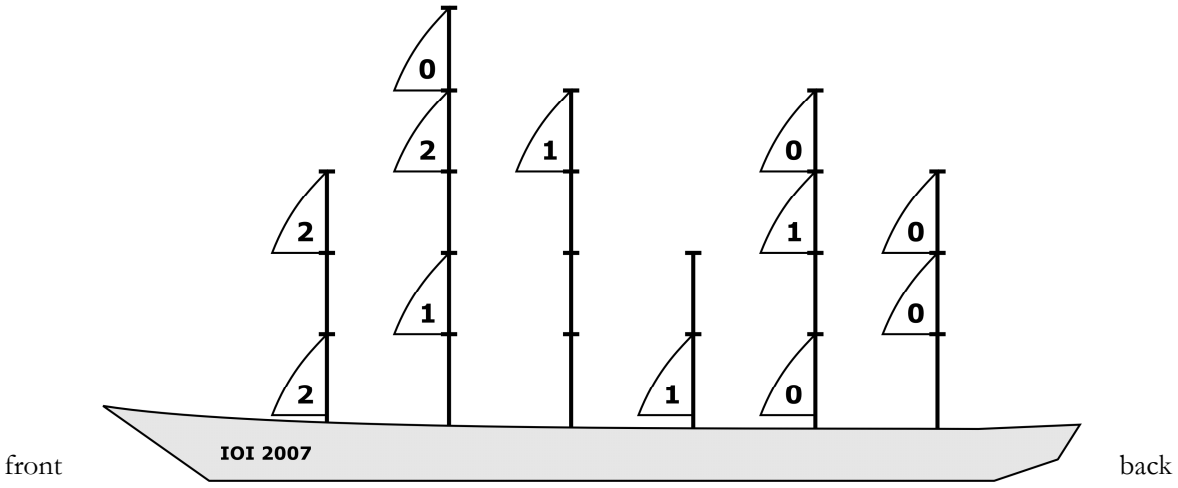
Tests-Setter: Pengqi Cheng, Rujia Liu

SAILS

A new pirate sailing ship is being built. The ship has N masts (poles) divided into unit sized segments – the height of a mast is equal to the number of its segments. Each mast is fitted with a number of sails and each sail exactly fits into one segment. Sails on one mast can be arbitrarily distributed among different segments, but each segment can be fitted with at most one sail.

Different configurations of sails generate different amounts of thrust when exposed to the wind. Sails in front of other sails at the same height get less wind and contribute less thrust. For each sail we define its **inefficiency** as the total number of sails that are **behind** this sail and **at the same height**. Note that "in front of" and "behind" relate to the orientation of the ship: in the figure below, "in front of" means to the left, and "behind" means to the right.

The **total inefficiency** of a configuration is the sum of the inefficiencies of all individual sails.



This ship has 6 masts, of heights 3, 5, 4, 2, 4 and 3 from front (left side of image) to back.
This distribution of sails gives a total inefficiency of 10. The individual inefficiency of each sail is written inside the sail.

TASK

Write a program that, given the height and the number of sails on each of the N masts, determines the **smallest** possible total inefficiency.

INPUT

The first line of input contains an integer N ($2 \leq N \leq 100\,000$), the number of masts on the ship.

Each of the following N lines contains two integers H and K ($1 \leq H \leq 100\,000$, $1 \leq K \leq H$), the height and the number of sails on the corresponding mast. Masts are given in order from the front to the back of the ship.

OUTPUT

Output should consist of a single integer, the smallest possible total inefficiency.

Note: use a 64-bit integer type to calculate and output the result (`long long` in C/C++, `int64` in Pascal).

GRADING

In test cases worth a total of 25 points, the total number of ways to arrange the sails will be at most 1 000 000.

EXAMPLE

input

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

output

```
10
```

This example corresponds to the figure on the previous page.

Central European Olympiad in Informatics
Tîrgu Mureş, România
July 8 – 14, 2009
Contest Day 2

sorting 100 points

Input files: **0-sorting.in, 1-sorting.in, …, 9-sorting.in**
Output files: **0-sorting.out, 1-sorting.out, …, 9-sorting.out**
Time limit: **5 hours**
Source code: **None**

Task

For given **N** and **X**, determine the number of permutations of **{1, 2, …, N}** on which Insertion Sort makes at most **X** times the number of comparisons that Quick Sort makes. Since the number can be pretty big, we ask you to output it modulo **1234567**.

The following is our implementation of Insertion Sort, which also counts the number of comparisons it makes:

```
procedure insertionSort(int N, array A[1..N]) defined as:
    A[0] := -Infinity
    for i := 2 to N do:
        j := i
        Increment(comparison_count)
        while A[j - 1] > A[j] do:
            SWAP(A[j - 1], A[j])
            j := j - 1
            Increment(comparison_count)
         end while
    end for
```

The following is our implementation of Quick Sort. If **L** is the length of the list we are sorting in a recursive step, the partition algorithm makes **L-1** comparisons.

```
procedure quickSort(list A) defined as:
    list less, greater
    if length(A) <= 1  then
        return A

    pivot := A[1]
    for i := 2 to length(A) do:
        Increment(comparison_count)
        if A[i] < pivot then append A[i] to less
                        else append A[i] to greater
        end if
    end for
    return concatenate(quickSort(less), pivot, quickSort(greater))
```

For example, let us consider the permutation **(3, 1, 4, 2)**.
The number of comparisons for Insertion Sort is **6**: **2** comparisons for i=2, **1** for i=3, and **3** for i=4.

The number of comparisons for Quick Sort is **4**. In the first iteration 3 is the pivot. It takes **3** comparisons to partition **(1,4,2)** into **(1,2)** and **(4)**. To further sort **(1,2)** it takes **1** more comparison.

Input

The first line contains 2 integers, separated by a space: **N** and **X**.

Output

The number of permutations for which Insertion Sort is at most **X** times slower than Quick Sort. The number should be printed modulo **1234567**.

Constraints

- In all inputs files, $1 < N < 32$.
- In all inputs files, $1 \leq X \leq N^2$
- The official solution computes the answers to all **10** test cases in less than **6** minutes.

Examples

x-sorting.in	x-sorting.out
3 1	2

For the 6 possible permutations we have **NI** and **NQ**, the number of comparisons for Insertion Sort and Quick Sort:

```
1 2 3 - NI = 2, NQ = 3
1 3 2 - NI = 3, NQ = 3
2 1 3 - NI = 3, NQ = 2
2 3 1 - NI = 4, NQ = 2
3 1 2 - NI = 4, NQ = 3
3 2 1 - NI = 5, NQ = 3
```

x-sorting.in	x-sorting.out
6 2	719

x-sorting.in	x-sorting.out
21 3	660773

Description of input

You are not supposed to submit any program to solve the described task. Instead, in the archive you can download from the grading system, you will find the files **`0-sorting.in`**, **`1-sorting.in`**, …, **`9-sorting.in`**. The files are the input data of each of the 10 test cases. You can use the command "**`unzip sorting.zip`**" to extract files from zip archives.

Each of the input files **`0-sorting.in`**, **`1-sorting.in`**, …, **`9-sorting.in`** describes a single test case. The first and only line contains two integers **`N`** and **`X`**, separated by a single space.

Description of output

For each input file, you should create a corresponding output file **`0-sorting.out`**, **`1-sorting.out`**, …, **`9-sorting.out`**. Write all these files to a directory called **`sorting-out`** and create a zip archive containing this directory. You should submit this archive as your solution.

You can use the command "**`zip –r sorting-out.zip sorting-out`**" to create an archive called **`sorting-out.zip`**.

The output files **`0-sorting.out`**, **`1-sorting.out`**, …, **`9-sorting.out`** should consist of exactly one line containing the requested number.

Submission feedback

When you submit your zip archive, the grading system will test whether your files have correct format, but won't test the correctness of the answers. For instance, if you receive a "score" of **`50`** points, it means your archive had **`5`** files that are named correctly, and each contains exactly one number. It **does not** mean that your answers were correct.

TRAINING

Mirko and Slavko are training hard for the annual tandem cycling marathon taking place in Croatia. They need to choose a route to train on.

There are N cities and M roads in their country. Every road connects two cities and can be traversed in both directions. Exactly N−1 of those roads are **paved**, while the rest of the roads are unpaved trails. Fortunately, the network of roads was designed so that each pair of cities is connected by a path consisting of paved roads. In other words, the N cities and the N−1 **paved roads form a tree structure**.

Additionally, each city is an endpoint for **at most 10 roads total**.

A training route starts in some city, follows some roads and ends in the same city it started in. Mirko and Slavko like to see new places, so they made a rule **never to go through the same city** nor **travel the same road twice**. The training route may start in any city and does not need to visit every city.

Riding in the back seat is easier, since the rider is shielded from the wind by the rider in the front. Because of this, Mirko and Slavko change seats in every city. To ensure that they get the same amount of training, they must choose a route with an **even number of roads**.

Mirko and Slavko's competitors decided to **block** some of the unpaved roads, making it **impossible** for them to find a training route satisfying the above requirements. For each unpaved road there is a **cost** (a **positive** integer) associated with blocking the road. It is impossible to block paved roads.

TASK

Write a program that, given the description of the network of cities and roads, finds the **smallest total cost** needed to block the roads so that **no training route exists** satisfying the above requirements.

INPUT

The first line of input contains two integers N and M ($2 \leq N \leq 1\,000$, $N-1 \leq M \leq 5\,000$), the number of cities and the total number of roads.

Each of the following M lines contains three integers A, B and C ($1 \leq A \leq N$, $1 \leq B \leq N$, $0 \leq C \leq 10\,000$), describing one road. The numbers A and B are different and they represent the cities directly connected by the road. If C=0, the road is paved; otherwise, the road is unpaved and C represents the cost of blocking it.

Each city is an endpoint for at most 10 roads. There will never be more than one road directly connecting a single pair of cities.

OUTPUT

Output should consist of a single integer, the smallest total cost as described in the problem statement.

GRADING

In test cases worth a total of 30 points, the paved roads will form a chain (that is, no city will be an endpoint for three or more paved roads).

DETAILED FEEDBACK WHEN SUBMITTING

During the contest, you may select up to 10 submissions for this task to be evaluated (as soon as possible) on part of the official test data. After the evaluation is done, a summary of the results will be available on the contest system.

EXAMPLES

input

```
5 8
2 1 0
3 2 0
4 3 0
5 4 0
1 3 2
3 5 2
2 4 5
2 5 1
```

output

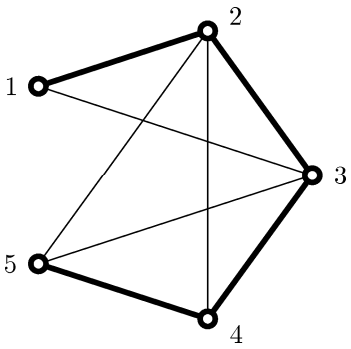
```
5
```

input

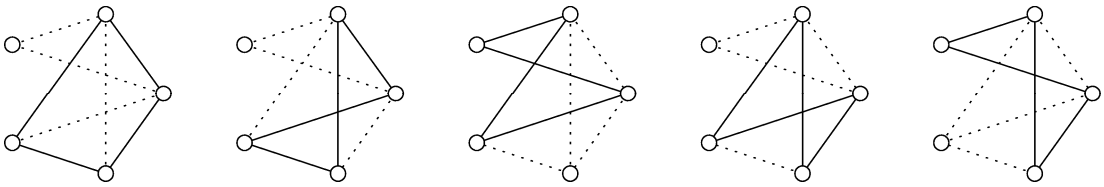
```
9 14
1 2 0
1 3 0
2 3 14
2 6 15
3 4 0
3 5 0
3 6 12
3 7 13
4 6 10
5 6 0
5 7 0
5 8 0
6 9 11
8 9 0
```

output

```
48
```



The layout of the roads and cities in the first example. Paved roads are shown in bold.



There are five possible routes for Mirko and Slavko. If the roads 1-3, 3-5 and 2-5 are blocked, then Mirko and Slavko cannot use any of the five routes. The cost of blocking these three roads is 5.

It is also possible to block just two roads, 2-4 and 2-5, but this would result in a higher cost of 6.

tri 100 points

Source code:	`tri.c, tri.cpp, tri.pas`
Input files:	`tri.in`
Output files:	`tri.out`
Time limit:	`2 s`
Memory limit:	`64 MB`

Task

You are given **K** points with positive integer coordinates. You are also given **M** triangles, each of them having one vertex in the origin and the other **2** vertices with non-negative integer coordinates.
You are asked to determine for each triangle whether it has at least one of the **K** given points inside. (None of the **K** points are on any edge of any triangle.)

Input

The first line of the input file **tri.in** will contain **K** and **M**. The following **K** lines will contain **2** positive integers **x y** separated by one space that represent the coordinates of each point. The next **M** lines have **4** non-negative integers separated by one space, (**x1,y1**) and (**x2, y2**), that represent the other **2** vertices of each triangle, except the origin.

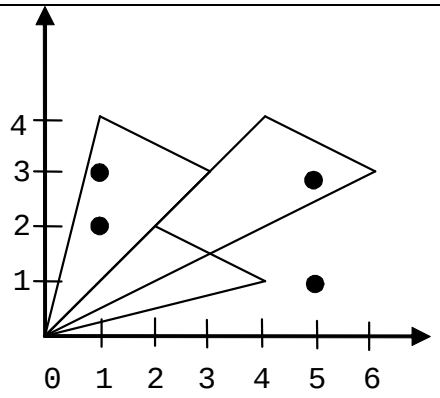
Output

The output file **tri.out** should contain exactly **M** lines. The *k*-th line should contain the character **Y** if the *k*-th triangle (in the order of the input file) contains at least one point inside it, or **N** otherwise.

Constraints

- $1 \le K, M \le 100\,000$
- $1 \le$ each coordinate of the K points $\le 10^9$
- $0 \le$ each coordinate of the triangle vertices $\le 10^9$
- Triangles are not degenerate (they all have nonzero area).
- In **50%** of the test cases, all triangles have vertices with coordinates **x1=0** and **y2=0**. That is, one edge of the triangle is on the *x*-axis, and another is on the *y*-axis.

Example

tri.in	tri.out	Explanation
4 3 1 2 1 3 5 1 5 3 1 4 3 3 2 2 4 1 4 4 6 3	Y N Y	

tri.in	tri.out	Explanation
4 2 1 2 1 3 5 1 4 3 0 2 1 0 0 3 5 0	N Y