harbingers 100 points

Source code:	`harbingers.c, harbingers.cpp, harbingers.pas`
Input files:	`harbingers.in`
Output files:	`harbingers.out`
Time limit:	`1 s`
Memory limit:	`32 MB`

Task

Once upon a time, there were **`N`** medieval towns in the beautiful Moldavian territory, uniquely numbered from **`1`** through **`N`**. The town numbered with **`1`** was the capital city. The towns were connected by **`N-1`** bidirectional roads, each road having a length expressed in kilometers. There was a unique way to travel between any pair of towns without going through a town twice (i.e. the graph of roads was a tree).

When a town was attacked, the situation had to be reported as soon as possible to the capital. The message was carried by harbingers, one of which resided in each town. Each harbinger was characterized by the amount of time required to start the journey and by his constant speed (expressed in minutes per kilometer) after departure.

The message from a town was always carried on the unique shortest path to the capital. Initially, the harbinger from the attacked town carried the message. In each town that he traversed, a harbinger had two options: either go to the next town towards the capital, or leave the message to the harbinger from this town. The new harbinger applied the same algorithm as above. Overall, a message could be carried by any number of harbingers before arriving in the capital.

Your task is to find, for each town, the minimum time required to send a message from that town to the capital.

Description of input

The first line of the input file **`harbingers.in`** contains one integer **`N`**, the number of towns in Moldavia. Each of the following **`N-1`** lines contains three integers **`u v d`**, separated by one space, describing a road of length **`d`** kilometers between towns numbered with **`u`** and **`v`**. Subsequently, **`N-1`** pairs of integers follow, one per line. The **`i`**th pair, S_i V_i, describes the characteristics of the harbinger in the **`(i+1)`**th town: S_i is the number of minutes to prepare for the journey, and V_i is the number of minutes needed to travel one kilometer. There is no harbinger in the capital.

Description of output

The output file **harbingers.out** should consist of exactly one line containing **N-1** integers. The **i**th number represents the minimum time, in minutes, required to send a message from the **(i+1)**th town to the capital.

Constraints

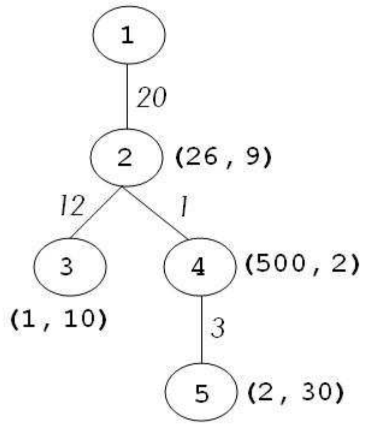
- $3 \le N \le 100\ 000$
- $0 \le S_i \le 10^9$
- $1 \le V_i \le 10^9$
- The length of each road will not exceed **10 000**
- For 20% of the tests, $N \le 2\ 500$
- For 50% of the tests, each town will have at most 2 adjacent roads (i.e., the graph of roads will be a *line*)

Example

harbingers.in	harbingers.out

5
1 2 20
2 3 12
2 4 1
4 5 3
26 9
1 10
500 2
2 30
	206 321 542 328

Explanation



The roads and their lengths are shown in the image on the left. The start-up time and speed of the harbingers are written between brackets.

The minimum time to send a message from town 5 to the capital is achieved as follows. The harbinger from town 5 takes the message and leaves the town after 2 minutes. He walks 4 kilometers in 120 minutes before arriving in town 2. There he leaves the message to the harbinger from that town. The second harbinger requires 26 minutes to start the journey and walks for 180 minutes before arriving to the capital.

The total time is therefore 2 + 120 + 26 + 180 = 328.

photo 100 points

Source code:	**`photo.c, photo.cpp, photo.pas`**
Input files:	**`photo.in`**
Output files:	**`photo.out`**
Time limit:	**`1.0 s`**
Memory limit:	**`16 MB`**

Task

You are given a photo of the skyline of Târgu-Mureş taken during the night. Some rooms still have the light on. You know that all the buildings can be modeled by rectangles of surface area at most **A**. Find the minimum number of buildings that can lead to the picture.

Specifically, you are given an integer **A**, and **N** points at integer coordinates (**x,y**). You must find a minimum number of rectangles that have one side on the *x*-axis and area at most **A,** which cover all points. The rectangles may overlap.

Description of input

The first line of the input file **`photo.in`** will contain two integers **N** and **A**, separated by a single space. The next **N** lines will contain two integers **x** and **y**, representing the coordinates of each point.

Description of output

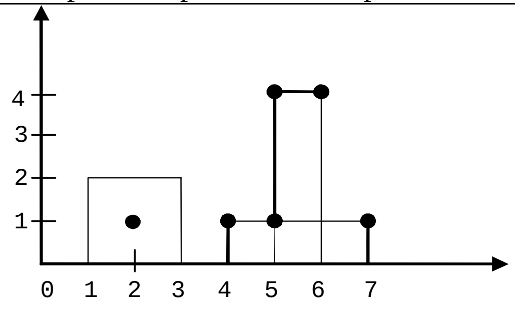
The output file **`photo.out`** should consist of exactly one line containing the minimum number of rectangles.

Constraints

- $1 \leq N \leq 100$
- $1 \leq A \leq 200\ 000$
- Each point has $0 \leq x \leq 3\ 000\ 000$ and $1 \leq y \leq A$
- For **30**% of the test cases, $1 \leq N \leq 18$

Example

photo.in	photo.out	Here is one possible picture that explains the example:

6 4
2 1
4 1
5 1
5 4
7 1
6 4
	3	

logs — 100 points

Source code:	**`logs.c, logs.cpp, logs.pas`**
Input files:	**`logs.in`**
Output files:	**`logs.out`**
Time limit:	**`0.6 s`**
Memory limit:	**`64 MB`**

Task

Given an **`N x M`** binary matrix, find the area of the biggest rectangle consisting entirely of values of **`1`**, knowing that you can permute the **columns** of the matrix.

Constraints

- **$1 \leq N \leq 15000$**
- **$1 \leq M \leq 1500$**
- **30%** of the test cases will have **$N, M \leq 1024$**
- In C/C++, it is recommended that you use **`fgets()`** to read the input. In Pascal, it is recommended to use **`readln()`** on a text file that has a large buffer. The following sample code shows how to do this:

C/C++

```
#define MAXM 1500
…
FILE *f = fopen("logs.in", "r");
char s[MAXM + 3];
fscanf(f,"%d %d\n", &n, &m);
fgets(s, MAXM + 2, f);
```

Pascal

```
var buf:array[1..65536] of char;
    s:array[1..1505] of char;
    f:text;
…
assign(f, 'logs.in');
settextbuf(f, buf, 65536);
reset(f);
readln(f, n, m);
readln(f, s);
```

At the end of these two pieces of code, **`s`** will contain the first line of the matrix.

Input

The first line of the input file **`logs.in`** will contain two integers separated by one space: **`N`** and **`M`**. The following **`N`** lines will contain **`M`** characters of **`0`** or **`1`**, describing the matrix.

Output

The only line of the output file **`logs.out`** will contain the area of the largest rectangle.

Example

logs.in	logs.out	Explanation

10 6
001010
111110
011110
111110
011110
111111
110111
110111
000101
010101
	21	By permuting the columns such that columns 2, 4 and 5 are adjacent you have a rectangle of area 21 (rows 2-8 and columns 2, 4, 5).

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, $1 \le X \le N^2$
- 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.

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

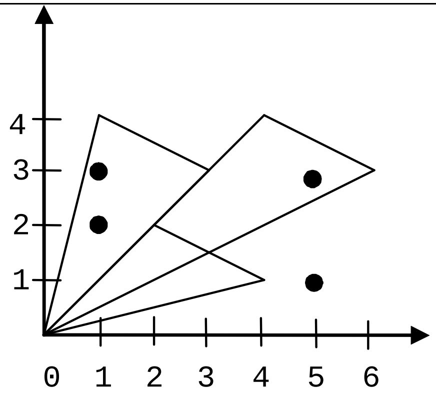
The output file **`tri.out`** should contain exactly **M** lines. The *k*-th line should contain the character **Y** if the *k*-th triangle (in the order of the input file) contains at least one point inside it, or **N** otherwise.

Constraints

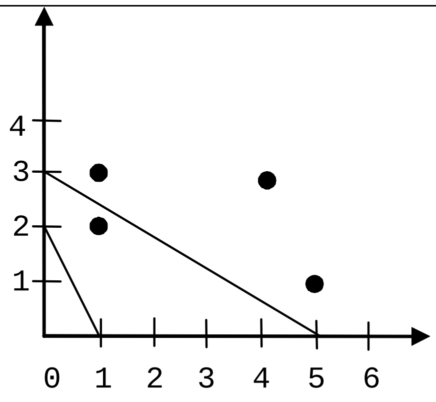
- **$1 \le K,M \le 100\ 000$**
- **$1 \le$ each coordinate of the K points $\le 10^9$**
- **$0 \le$ each coordinate of the triangle vertices $\le 10^9$**
- Triangles are not degenerate (they all have nonzero area).
- In **50%** of the test cases, all triangles have vertices with coordinates **`x1=0`** 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
		 4
		3
		2
		1
		0 1 2 3 4 5 6