
bubble pop!

Content:

bubble cup 1

bubble cup 2

bubble cup 3

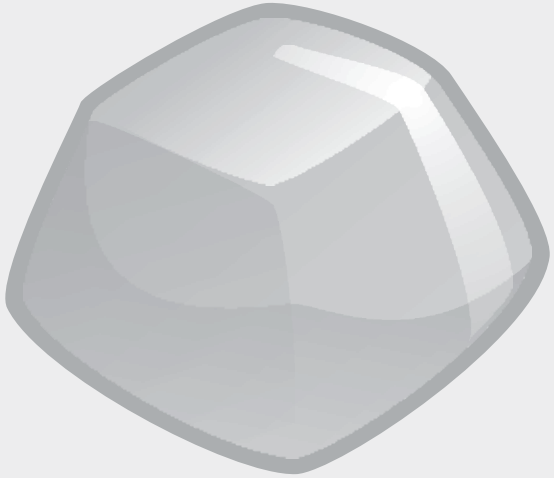
bubble cup 4

bubble cup 5

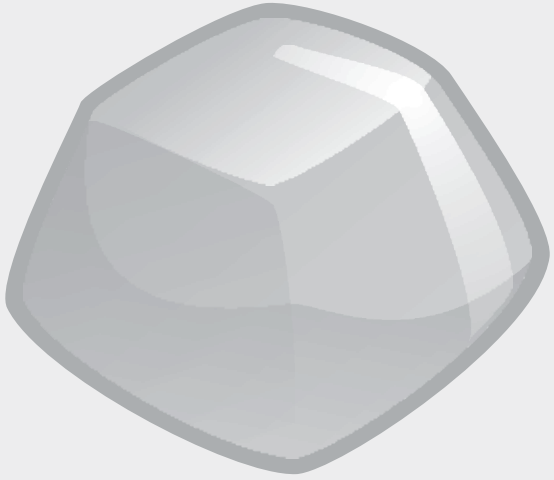
Welcome

This book contains all problems from the first five finals in the 10+ year long history of Bubble Cup. It is intended for high school and university students, and anyone else wanting to learn more about programming and algorithms. Solving the problems in this book will require skills far greater than those that are taught in high schools and universities.

Don't be discouraged if you can't immediately solve the problems. They are intended to challenge the best programming teams in the world, and as such are very difficult. We hope you expand your programming knowledge by learning new interesting algorithmic tricks while reading this book.



bubble cup 1



Problem A: Lock

Statement:

In a dark basement, there is a wooden case with printed solutions to all tasks in this contest. However, the basement has thick walls and a door, and a lock on the door. On the lock, there are n horizontal iron bars, and on each of the bars there is a word with letters of equal width. Each bar can be moved independently to the left or right for one or more widths of a letter. There is, at least, one letter that is common to all words. Therefore, bars can be lined up so that there is a vertical line of n identical letters above each of them (each letter on one bar). To unlock the door, bars should be positioned in such a way that there is a maximal number of such consecutive vertical lines.
You are naturally interested in writing a program that solves this problem.

Input:

The first line contains a number n, the number of bars. In each of next n lines, there is a word corresponding to one of the bars. Each word contains only capital letters and is at least 1 and at most 100 characters long.

Output:

A string of maximal length, that appears in every word as a sequence of consecutive letters. If there is more than one solution, you should print the leftmost one.

Constraints:

- *1 ≤ n ≤ 1,000*
- *Each word is between 1 and 100 characters long*

Example input:

```
3
THATBALLOONRISEDTOTHETOP
FOOTBALLWIZARDWASSLEEPY
SOHELOSTBALANCEANDDROPPEDBALLON
```

Example output:

```
TBAL
```

> Time and memory limit: 4s / 16MB

Solution and analysis:

There exists a well-known solution to a simpler version of this problem, where the goal is instead to find the longest common substring between only two strings. The solution uses dynamic programming and a two-dimensional matrix, of form $dp[i][j]$, to find the longest common substring between all ending indexes in both strings; where the max value in dp is taken as the final answer.
(The exact approach to this simpler problem will not be explained here but can be found in many dynamic programming tutorials).
Now, we can apply this idea of finding all longest common substrings between two strings, to solving the original problem. Since we are finding the longest common substring between all strings, this implies that there exists a pair of indexes, a_i and b_i for each string where the substring $string_i[a_i..b_i]$ is equal to the final answer.
Since the answer is to be found in all given strings, we can set a random string to be our 'reference' string. Using the algorithm for the simpler problem with two strings, we compare all other strings to the reference string.
In a separate array of length n, at each index i we can store the length of the longest substring found in all other strings that is in our reference string and ends at index i. After comparing the reference string and another string, the longest length substring in both strings that ends at index i in the first string is equal to $max(dp[i][j])$ for all j. After each comparison, this separate array is updated. And the final answer is equal to the maximum value in this array, after all comparisons are done.

Complexity:

- *Time: $O(n^3)$*
- *$O(n^2)$ for each comparison between two strings, and there are n comparisons.*
- *Memory: $O(n^2)$*
- *Each comparison can be done in the same memory.*

Problem B: Triangles

Statement:

You are given two triangles $A_1B_1C_1$ and $A_2B_2C_2$ in a plane. Is there an isometric transformation of the plane that maps one triangle onto another, preserving the order of vertices? If so, what kind of transformation is that?

Input:

12 integer numbers in one line separated by a space, which are the coordinates of vertices in the following order: $A_{1,x}, A_{1,y}, B_{1.x}, B_{1,y}, C_{1,x}, C_{1,y}, A_{2,x}, A_{2,y}, B_{2,x}, B_{2,y}, C_{2,x}, C_{2,y}$. The vertices of one triangle are not collinear, i.e. triangles do exist.

Output:

One of the following letters: N, I, T, R, S, O.
Here is the meaning for each letter:

- N: there is no isometric transformation mapping one triangle onto another;
- I: the transformation asked for is the identity transformation;
- T: the transformation asked for is a translation;
- R: the transformation asked for is a rotation;
- S: the transformation asked for is an axial symmetry;
- O: the transformation asked for is some other isometric transformation.

Constraints:

- *All coordinates are between -10,000 and +10,000, inclusive*
- *The vertices of one triangle are not collinear*

Example input:

```
1 1 2 2 1 2 5 7 6 8 5 8
```

Example output:

```
T
```

> Time and memory limit: 0.1s / 16MB

Solution and analysis:

This problem boils down to checking step-by-step whether the two given triangles satisfy the set of criterions for each isometric transformation. The following list contains the criteria for each transformation. Note that the ordering is important, and the first transformation that is satisfied in the list is the final answer.

N (No isometric transformation):

$Distance(A_1,B_1) \neq Distance(A_2,B_2)$ *or* $Distance(A_1,C_1) \neq Distance(A_2,C_2)$ *or* $Distance(B_1,C_1) \neq Distance(B_2,C_2)$, *where* $Distance(x,y)$ *is a function that calculates the distance between two points.*

I (Identity transformation):

The identity transformation does not change anything about the original triangle when transforming it.
Criteria: $A_1 = A_2$, *and* $B_1 = B_2$ *and* $C_1 = C_2$

T (Translation transformation):

The translation transformation translates each vertex of the original triangle by a certain vector.
Criteria: $A_2 - A_1 = B_2 - B_1 = C_2 - C_1$

R (Rotation transformation):

Since we are, at this point, sure that we are dealing with an isometric transformation(not a translation), it is enough to check whether the vertices of both triangle, $A_1B_1C_1$ *and* $A_2B_2C_2$, *occur in the same order; clockwise or counter-clockwise. Remember, this direction can be computed through the sign of the cross product of vectors* $\overrightarrow{AB}$ *and* $\overrightarrow{AC}$ *in triangle* ABC.

S (Axial symmetry transformation):

If a transformation is an axial symmetry then vectors $\overrightarrow{A_1A_2}$, $\overrightarrow{B_1B_2}$, $\overrightarrow{C_1C_2}$ *need to be perpendicular to the same axis and the axis needs to split those lines in half. Using the dot product, we can easily check if the lines are perpendicular, since in that case their dot product will be equal to* 0.

(Other isometric transformation):

This transformation is satisfied if none of the criteria for the transformations above have been satisfied.

Problem C: Ring

Statement:

A sequence of letters is written on a miraculous ring. When the sequence is read aloud starting from any letter, magic happens. To get the greatest magic, you need the most powerful word: you want to find a place to start reading from, so that the word you get is the greatest possible word in the alphabetical order.

Input:

One string, consisting of English capital letters A - Z. The number of letters is at least 1 and at most 100000.

Output:

Two numbers in one line, separated by a space. The first number is the number of places from which the most powerful word can be achieved. The second number is the smallest index of the starting letter, (counting from one), that gives the most powerful word.

Constraints:

- *The string consists of English capital letters A-Z*
- *The number of letters is at least 1 and at most 100,000*

Example input:

```
ABRACADABRACAD
```

Example output:

```
2 3
```

```
> Time and memory limit:  0.1s / 16MB
```

Solution and analysis:

There are several feasible solutions to this problem.

The following algorithms is called the Booth's algorithm for minimal string rotation.

The first approach uses a slightly modified version of the preprocessing function from the Knuth-Morris-Pratt algorithm. The failure function is calculated as normal, but the string is being rotated during the computation, so some indices must be calculated more than once, as they wrap around. When all indices have been successfully computed without the string rotating again, the solution is found. We have to notice that we need the maximal rotation, so we must reverse the comparisons.
After finding the maximal string rotation, we now have to find the number of times this string appears as a rotation. This is equal to the minimal period of the calculated string, and the minimal period is just the smallest string that when concatenated several times gives the starting string. The minimal period can be computed also via the KMP algorithm. This yields total time complexity $O(n)$.
An alternative solution to finding the maximal string rotation would be to create a suffix array. First, we concatenate our string to itself and then sort the suffixes, our solution is the largest suffix which starts in the first half. Then we do a binary search on the sorted suffixes to find how many times the maximal rotation appears.

Reference:
https://en.wikipedia.org/wiki/Lexicographically_minimal_string_rotation#Booth.27s_Algorithm

Problem D: Secretary

Statement:

The big boss has a new secretary who is rather clumsy and computer illiterate. Once, the boss asked her secretary to make n copies of some text, which contained a very important alphanumeric code of length m. The poor secretary has retyped the whole text n times. A bit later, in each copy she found one mistyped character (not necessarily a different one). Unfortunately, each time it was one of the characters in the code. To make things worse, the secretary has lost the original text, so the correction cannot be done easily. Please help the secretary to keep her job and reconstruct the correct code if reconstruction is possible.

Input:

The first line contains two positive integer numbers n and m, separated by a space. In each of the next n lines, there is a sequence of m characters, which are either capital English letters or digits.

Output:

If there are several sequences that differ from each of the input sequences in exactly one character, the output should contain the word "AMBIGUOUS". If there is no such a sequence (meaning that the secretary did not even count the mistakes correctly), the output should contain the word "IMPOSSIBLE". If there is only one such a sequence, it is considered correct and the output should contain that sequence.

Constraints:

- $1 \leq n \leq 100$
- $1 \leq m \leq 40$

Characters are capital English letters or digits

Example input:

```
3 9
12305A7Q9
12375A7Q9
12375A7Q9
```

Example output:

```
AMBIGUOUS
```

```
> Time and memory limit:  0.1s / 16MB
```

Solution and analysis:

Due to the small problem limits, a brute force approach is possible. Here are the constraints:

- *The number of text copies ($n \leq 100$)*
- *The length of each copy ($m \leq 40$)*
- *The alphabet size ($|A| \leq 40$)*

First, a few definitions to help us during the editorial:

- *Column I: The set of characters at the i^{th} index of the n strings.*
- *Corrupted Column: A column which has two or more distinct characters.*

Now let's examine the m columns one-by-one and label the column we are examining at any one moment as the column i. If the column i is not corrupted, we will assume that the letter contained at that index is correct (we will get back to this assumption when forming the final answers). Otherwise, the column i is corrupted and any character of the alphabet could be the correct character for the index i. Here is an example showing that:

Given strings:

```
aac
abc
acc
```

Possible answer:

```
azc
```

For each corrupted column, we will try to place every letter of the alphabet at that index and see if it is a potential solution. We do this by taking a random string, placing the candidate letter in that position and claiming it as the correct code. Since we assume to have the correct code, we compare all other codes against this code and make sure they all differ by only one character. If this condition holds, we have found one potential solution.
There is a special case when there are no corrupted columns (all strings equal), and then there are many correct original codes.
The final answer is ambiguous if we have found more than one potential solution, or if all the given codes are equal. If we have found only one solution, then we print that solution out as the final answer. Otherwise, no correct code exists.

Problem E: Proper Half

Statement:

You are given n integer numbers, where $n + 1$ is a power of 2. You should select any $m = \frac{n+1}{2}$ of those n numbers, so that their sum is divisible by m.

Input:

First line contains an integer n, the number of given integers. In the next line there are n integers a_k separated by a space.

Output:

A sequence of n characters, each being '+' or '-' . If the k^{th} character is '+', that means the k^{th} number in the input sequence is selected into the sum, and '-' means that it is not.

Constraints:

- $1 \leq n \leq 50{,}000$
- $k = 1,2,\ldots n$
- $|a_k| \leq 10^8$

Example input:

```
7
1 -4 3 7 6 -2 5
```

Example output:

```
---++++
```

```
> Time and memory limit:  0.1s / 16MB
```

Solution and analysis:

This task falls into the class of constructive algorithms. One should notice that out of N numbers, we can select $M-1$ pairs of numbers (such that each number can appear in one pair at most) of the same pairity, "throwing" one excess number away. By summing up each pair, we get a new array P_i with $M-1$ elements, where each element is obviously even. It means that the sum of all elements of array P is even:

$$P_1 + P_2 + P_3 + \cdots + P_{m-1} = S$$

and we can divide the equation with 2 getting:

$$\frac{P_1}{2} + \frac{P_2}{2} + \ldots + \frac{P_{m-1}}{2} = \frac{S}{2}$$

Now, the recursive pattern should be noticed. We have a new array with $M-1$ elements and should select $\frac{M}{2}$ of those so that their sum is even. The final step is with one element, for which statment from the task obviously holds.
The reconstruction of elements is simple. In the final recursion step we end up with one number B. From the previous step we should select a pair of numbers that B was constructed from. One step back, and we should select four numbers (two pairs) that these two were made from. We continue this process until we get to the starting array.
At each step (excluding the final one) we "gain" the factor of 2 and since we have $floor(logN)$ of these steps we know that the sum of the numbers that we select from the starting array will be divisible by $2^{floor(logN)}$ which is equal to M.

The complexity is $O(NlogN)$.

Problem F: Romeo and Juliet

Statement:

Romeo and Juliet are in love. However, their families forbid them to meet each other and Juliet wants to see Romeo so much. Luckily, they both live in Bubble Town, and Juliet is allowed to go to sing in a choir on Sunday afternoons.
She leaves her home at exactly four o'clock and she must drive using one of the shortest paths to the church. Similarly, Romeo plays football for the Bubble Team at a nearby football stadium every Sunday. He also leaves his home at four o'clock and must follow one of the shortest paths to the stadium.
All streets in Bubble Town are one-way streets. Every Sunday Juliet hopes that on the way to church she will suddenly see Romeo hurrying to football. However, so far it has not happened, and Juliet does not know whether it is possible at all.

Input:

The first line contains two positive integer numbers n and m, where n is the number of junctions and m is the number of streets in the town. Junctions are numbered from 1 to n. Every street connects exactly two junctions in one direction.
On the second line there are four numbers: JS, JG, RS, RG. Juliet lives at the junction JS, the church is at the junction JG, Romeo's home is at the junction RS and the football stadium is at the junction RG. Each of the next m lines contain three integers, a, b and d describing one street, where a and b are the numbers of the starting and ending junctions of this street and d is the time in minutes necessary to drive from a to b using this street. It is possible to go from any junction to any other junction using streets. For a pair of junctions, there are at most two streets connecting them (at most one in each direction).

Output:

Your task is to determine whether the shortest path for Romeo and the shortest path for Juliet exist, such that they will see each other at some junction. Both Romeo and Juliet start at the same time. They meet at a junction only if they arrive there at the same time. If they cannot meet, output -1. If they can meet, output the time in minutes from 4 o'clock to the first possible meeting of Romeo and Juliet.

Constraints:

- *1 ≤ n ≤ 20,000*
- *1 ≤ m ≤ 40,000*
- *1 ≤ d ≤ 30*

Junctions are numbered from 1 to n

Example input:

```
5 7
1 2 4 5
1 3 5
3 2 7
4 3 5
3 5 8
4 5 13
5 1 9
2 4 6
```

Example output:

```
5
```

> Time and memory limit: 0.5s / 16 MB

Solution and analysis:

It is possible for Romeo and Juliet to meet at any junction, therefore we will check each one. Let the current candidate junction be the junction x. We must check the following conditions for it to be possible for Romeo and Juliet to meet here:
Whether they can arrive at that junction at the same time (following their shortest paths to that junction).
Junction x is on one of the shortest paths for both Romeo and Juliet.

Let's denote the shortest path between two junctions x and y as $D(x, y)$.
These two conditions can be formally checked with the following criteria:

$D(JS, x) = D(RS, x)$ *–the shortest paths to* x *for both Romeo and Juliet are equal.*
$D(JS, x) + D(x, JG) = D(JS, JG)$ *and* $D(RS, x) + D(x, RG) = D(RS, RG)$.

Because we are only doing queries of the type:

$D(JS, x)$
$D(RS, x)$
$D(x, JG)$
$D(x, RG)$

It is only needed to do the Dijkstra shortest path algorithm four times with the respectable parameters to precompute these queries:

- *Original graph, starting from JS*
- *Original graph, starting from RS*
- *Inverted graph, starting from JS*
- *Inverted graph, starting from RS*

For the final answer, we only need to memorize the valid junction x with the minimum value for the meeting time between Romeo and Juliet.

Problem G: Security

Statement:

A space of size n x n x n is to be guarded by a complex security system, consisting of n sensors. The location of each sensor must have three integer coordinates between 1 and n, inclusive. Furthermore, each sensor should secure (and be located in) a particular part of the space, given by minimal and maximal values along axes. Finally, in order to be able to catch most of the movements, no two sensors can share any of the coordinates. For example, two sensors cannot have the same x coordinate, regardless of y and z coordinates. It is not always possible to meet all these conditions, but when it is, any arrangement of sensors will do the job. You are asked to help arrange the sensors.

Input:

The first line contains the number n. In each of the next n lines, there are six integers x_1, y_1, z_1, x_2, y_2, z_2, describing the required subspace for k^{th} sensor.

Output:

In each of n lines, there should be three integers separated by a space, coordinates of k^{th} sensor. If there are no solutions, each of the n lines should contain three zeroes.

Constraints:

- $1 \le n \le 10{,}000$
- $1 \le x_1 \le x_2 \le n$
- $1 \le y_1 \le y_2 \le n$
- $1 \le z_1 \le z_2 \le n$
- $k = 1,2,\ldots n$

Example input:

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

Example output:

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

> Time and memory limit: 3s / 16MB

Solution and analysis:

There are no enforced relations between the x, y, z axes. Therefore, these 3 axes can be analyzed independently.
The problem boils down to assigning each integer coordinate (for the x, y, and z axis) to a sensor and making sure it is between the sensor's minimal and maximal values.
This is a popular problem of assigning points to a set of intervals, and there exists a greedy solution. The points are considered, left to right, and the intervals are removed from the available set of intervals when it is assigned to a point. When considering a point with coordinate value x, we will assign to it an interval which it fits into, but with the smallest value for the right side. The reasoning behind this greedy approach is that we want to keep/save the other intervals for points which will be considered later, and we don't want to waste that extra space.
This matching of points to intervals can be done in a brute force approach because of the large enough time limit, in $O(n^2)$. Or we can sort the intervals and do a two-pointer solution, in $O(n \log n)$ because of the sort.

Problem H: Posters

Statement:

Many artists have announced their concerts for the upcoming holidays. Currently, there is an advertising war going on. Several agencies have employed workers to put up posters on streets. In Bubble City there is only one (although very long) wall along the main street where posters are allowed. In the advertising union they have an agreement not to put posters over other, recently put up posters. In order to make sure that the agreement is upheld, they introduced a rule stating that an agency can send workers only if they are given a specific section of the wall, which they can cover fully with the posters. The section is specified by the start and end positions.
Since there is little time left till the beginning of the show, agencies have started to play rough: they still obey the rule about specifying the sections in advance, but they do not care anymore about what was on the wall before. There are n sections specified and each of them is covered by a different poster. Workers come out one after another.
After the concerts, agencies will want to analyze the effects of the campaign and they need to know, for each different poster, how much of the poster is still visible on the wall. They will certainly employ some programmers. Are you interested?

Input:

The First line contains a positive integer n, the number of poster types. Each of the next n lines have two real numbers a_k and b_k, determining the section where poster of type k has been put. For each poster type, there is exactly one section, and poster types are given in the order of execution.

Output:

The output consists of n real numbers in one line, separated by a space, printed with 3 decimal places per number. Each number represents the length of all parts of the wall where the corresponding poster is visible.

Constraints:

- $1 \leq n \leq 100{,}000$
- a_k and b_k are real numbers

Example input:

```
3
2.0 5.0
3.0 7.0
4.0 6.0
```

Example output:

```
1.000 2.000 2.000
```

> Time and memory limit: 1s / 16MB

Solution and analysis:

Let's first summarize the problem. Posters can be considered as intervals that cover an infinite set of points. Each point in the set must be assigned to an interval that contains it, and if there are multiple intervals, the point is assigned to the interval that comes last in the input file. But instead of considering each point, we will consider 'important' points where the answer changes. Going left to right, changes in the solution happen at the ends of the intervals. Using a line sweep approach (a popular programming approach), we will scan the ends of the intervals left to right and accumulate the answer during this scan.

Let's classify these ends as 'markers'. Each marker will contain some additional info: an x-coordinate, a flag that shows whether it opens or closes the interval, the index of the interval it belongs to in the input file. Each interval end will be converted to a marker, and the markers will be sorted based on their x-coordinate.

During this scan we will maintain an additional structure ActiveIndexes, a set that contains indexes of all the intervals we are currently 'inside of'. So, once we pass the opening end of an interval, its index is added to this structure. And once we pass the closing end of an interval, its index is removed from this structure.

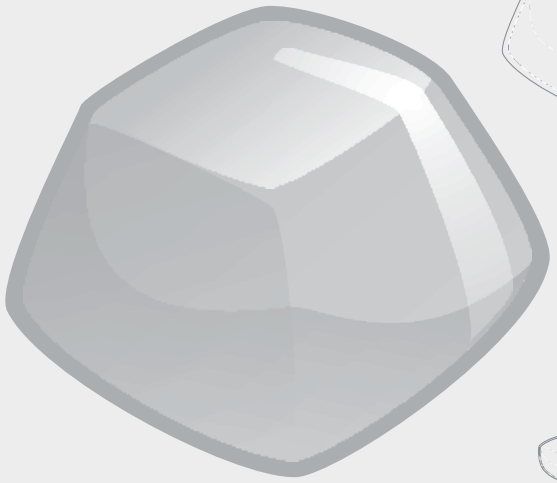
Now, we scan the markers from left to right. At each marker we will perform two steps:

Between this marker and the previous one, there might have be a poster that is considered to be the answer for that interval of points. Therefore, if ActiveIndexes is not empty, we take the maximal poster index from that set and add the distance between these two markers to that poster's answer.

If the marker opens its interval, add its index to ActiveIndexes. If it closes the interval, remove its index from ActiveIndexes.

After this scan, the final answer for each poster has been accumulated. So, all that is left is to output them.

bubble cup 2



Problem A: Decorations

Statement:

For the next year's Bubble Cup, the organizers thought of the possible decorations that could be arranged. One idea was to create a string of balloons that could go all around the competition arena. The problem is that the director of the competition likes only certain sequences of balloon colors to be used for the string. For example, if there are four different colors: A, B, C and D, the director might say that only the sequences ABB, BCA, BCD, CAB, CDD and DDA can be used for decoration.
If the length of the string was 5, then the only possible arrangements could be $BCABB$ and $BCDDA$ (strings such as $ABBCA$ could not be used because BBC is not an approved sequence). Since the director likes variety, it is important to know the total number of arrangements possible, given the list of approved sequences.

Input:

Input consists of two lines. The first line will contain three positive integers n, l and m, where n indicates the number of different colors, l is the length of the arrangements we want to create, and m indicates the number of sequences which the director likes. The next line will contain m sequences. Each sequence will be of the same length and will be separated by a single space. All sequences will consist only of uppercase letters of the alphabet.

Output:

Output should be a single line containing the number of possible arrangements. All answers will be within the range of a 32-bit integer.

Constraints:

- *1 ≤ n ≤ 26*
- *1 ≤ m ≤ 600*
- *1 ≤ l ≤ 100*

Each sequence is of the same length, between 1 and 10.

Example input 1:

```
4 5 6
ABB BCA BCD CAB CDD DDA
```

Example input 2:

```
5 4 5
X Y Z W Q
```

Example input 3:

```
4 8 3
AA BB CC
```

Example output 1:

```
2
```

Example output 2:

```
625
```

Example output 3:

```
3
```

> Time and memory limit: 0.5s / 16MB

Solution and analysis:

Simply said we have the following problem: How many different strings of length l are there with the restriction that every substring must belong to the given list of allowed strings.
Let us denote the length of every string in the list as Len.
In the first place, we will create a graph so that the strings in the given list are the nodes. We will create a directed edge between two nodes if the two strings can be concatenated, so that the resulting string has length $Len + 1$. For this to happen, we need the following condition to be true: the last $Len - 1$ characters of the first string must be equal to the first $Len - 1$ characters of the second string. This basically means that we pick a starting string and if we follow some edge we extend our string by 1.
So, what we have here is a graph theory problem. The question now is how many paths of length $l - Len$ are there in our graph. This is because we start from one string that has the length l and every time we follow some edge we extend the string by 1.
Because our graph is a DAG we can solve this problem with Dynamic Programming. Our state will be: $f(i, j) \Rightarrow$ how many paths of length j end in the node i. To calculate $f(i, j)$ for some node i and some length j we just need to add all $f(k, j - 1)$ where k is a parent node of i.
Our result is the sum of all $f(i, l - Len)$ for every node i in our graph.

Problem B: Mosquitoes

Statement:

It is well known that organizers of the BubbleCup work hard to come up with interesting problems, provide delicious food and drinks, design cool t-shirts and throw an unforgivable party to make the event as pleasant as possible for the contestants. A little less is known about details like balloon color selection, choosing the brand of coffee or, for example, making sure there are no mosquitoes to bite the contestants.

Mosquitoes live in groups near Ada in Belgrade and organizers have successfully located them. The plan is to rent an airplane with anti-mosquito pesticide, fly over Ada and release the pesticide. For simplicity, we will consider mosquito groups as points. Also, since the airplane would be flying in a straight line, the area covered with pesticide would be a piece of land between two parallel lines. The higher the airplane flies, the wider the area it covers. However, the lower the airplane flies, pesticide is more concentrated when it hits the ground and therefore more effective.

To save on budget, the organizers would like to make only one flight over Ada in a straight line and use as little pesticide as possible. To achieve this, the airplane should fly as low as possible and thus cover the narrowest area possible. Help organizers find how they should fly the airplane.

Input:

The first line contains one positive integer n, where n is the number of mosquito groups. Next n lines contain real numbers x_i and y_i, the coordinates of the group i. Please note that the same mosquito group can be listed more than once.

Output:

One number rounded to 6 decimal digits after the decimal point – the minimal width of the area (between two parallel lines) that contains all mosquito groups.

Constraints:

- $1 \leq n \leq 500{,}000$
- $-500{,}000 \leq x_i, y_i \leq 500{,}000$

Example input:

```
4
0.0 0.0
0.0 100.0
100.0 0.0
10.0 70.0
```

Example output:

```
70.710678
```

> Time and memory limit: 2.5s / 16MB

Solution and analysis:

Notice that if each point lies between two parallel lines, then the convex hull of those points also lies between those two lines. Actually, these two conditions are equivalent.
Let's see the properties of two parallel lines that are the solution, i.e. whole convex hull lies between them and there are no two closer lines with the same property. It is obvious that each of them passes through some point of the convex hull. If that were not the case, then we could bring them closer, and find a better solution. Actually, one of the lines will pass through two adjacent points of the hull. Why does that hold true? Let p and q be the lines, and P and Q points they contain. If we rotate both p and q by the same angle $\measuredangle\alpha$ around P and Q as fixed points, they will remain parallel but become closer. We can continue rotating them until one hits the next point on the convex hull. At that moment, it will contain two adjacent points of the hull.
We can find the minimum distance by positioning one line through each of the edges on the convex hull and for each of them determining the point where the second line should pass through. That would be the furthest point from that edge. For the first edge we can find it in linear time. When we move to the next edge, we don't need to test all of the points, but only move to the next point which is further away from the previous one. This way, time complexity will remain linear in the number of points of the convex hull. However, we need to find the convex hull first, which requires $O(n \cdot log(n))$ time complexity.

This method is known as the rotating calipers.

References:
https://en.wikipedia.org/wiki/Rotating_calipers

Problem C: Teleport

Statement:

It is the year 2109 and 100^{th} Bubble Cup is scheduled (it is indeed 100^{th}, two were skipped in mid '70s during the Great Recession II). Perica takes part in all important programming competitions and of course he is a member of a team that is invited to Bubble Cup. But Perica has a bad habit, he really likes to sleep late in the mornings. He wants to be on time for the contest, but he wants to leave his home as late as possible to get the most sleep.

Perica always travels by train and uses Serbian Fast Railways Inc. He possesses a railway map and a train timetable, so for every two connected stations he knows the distance between stations and the departure and arrival time. Between any two stations there exists at least one direct or indirect path. Between two directly connected stations there exists only one train that goes back and forth in an infinite loop. If stations A and B are connected and the train needs time t to get from A to B, and we know from the timetable that the train departs from the station A at the moment t_0, then the train is at the station A at the following moments in time: ... $t_0 - 4t,\ t_0 - 2t,\ t_0 + 2t,\ t_0 + 4t$... and also the train is at the station B at: ... $t_0 - 3t,\ t_0 - t,\ t_0 + t,\ t_0 + 3t$... We assume that the train travels at constant velocity and spends no time at stations. However, different trains may have different velocities.

Luckily for Perica, Serbian Fast Railways Inc. has just installed brand new teleport machines in every train and at every station. So even if he misses a train he still has a chance to teleport himself from the station into the train, but only if the train is close enough to the station. If two stations are closer to each other than the distance supported by teleport machines, Perica can teleport directly from one station to another without using the train. On the other hand, teleporting forms a train to a station, from a train to a train and between non-connected stations is not possible. Help Perica find the fastest way from home to the contest.

Input:

First line of input contains three positive integer numbers: n –the number of train stations, m – the number of train lines (connections) and d – the maximal distance supported by all teleport machines.

Each of the next m lines contains five integers: $st_1\ st_2\ t_0\ t\ s$, where st_1 and st_2 are ordinal numbers of connected stations (enumeration starts with one), t_0 is the moment in time when the train is at the station st_1, t is the time that the train needs to travel between these two stations and s is the distance between them.

Note 1: The starting and final station (home and contest locations) are those with ordinal numbers 1 and n, respectively.

Note 2: All t_0's are relative to some moment in time and can take negative values

Output:

Print two integers separated by one space:

The shortest time that Perica needs to get from home to the contest if he is at the starting station at the moment 0.

The maximum amount of time that Perica can arrive late at the starting station so that he still arrives at the final station at the same time as if he arrived at the starting station at the moment 0.

Constraints:

- $1 \le n \le 10,000$
- $1 \le m \le 1,000,000$
- $1 \le d \le 10,000$
- $1 \le st_1, st_2 \le n$
- $-10,000 \le t_0 \le 10,000$
- $0 < t, s \le 10,000$

Example input:

```
4 4 1
1 2 0 5 2
1 3 0 7 2
2 4 4 5 2
3 4 6 2 2
```

Example output:

```
8 3
```

```
> Time and memory limit:  5s / 128MB
```

Solution and analysis:

The problem is naturally split into two parts (the two numbers we need to output) and we will use the Dijkstra's algorithm to solve them.
In the first part, we calculate the earliest arrival time for each station as the distance in the algorithm. We start the algorithm from the home station with its time set to 0*. We update the time of the neighboring stations depending on their distance and train departure time.*

After that, we move to the next unvisited station with the smallest earliest arrival time and repeat the process until we visit the contest station. There are three cases of updating the neighbor station B *from the station* A*:*

- *If we can teleport directly from* A *to* B *(the teleport distance is greater than or equal to the distance between the stations) the earliest arrival time of* B *is updated with that of* A*.*
- *If 1) does not hold we calculate the last departure time of the train from* A *to* B *before the earliest arrival time of* A*. If we can teleport to it, then the earliest arrival time of* B *is updated with the arrival time of this train.*
- *If 1) and 2) do not hold, then the earliest arrival time of the station* B *is updated with the arrival time of the first train we can catch after we arrive at* A*.*

The earliest arrival time of the contest station is the first number we need to print.

The second part of the problem is solved similarly. This time we calculate the latest departure time of each station so that we could still arrive at the final station on time if we were to start from that station. We start the algorithm from the final station and set its time to the earliest arrival time from the first part. We update its neighbors and move to the unvisited one with the largest latest departure time until we arrive at the home station.
Since we cannot teleport from a train to a station, this time there are only two cases of updating the neighbor station A *from station* B*:*
If we can teleport directly from A *to* B*, the latest departure time of the station* A *is updated with that of* B*.*
Otherwise the latest departure time of the station A *is updated with the departure time of the last train that arrives at* B *before the latest departure time of* B*.*

The latest departure time of the home station is the solution to the second part of the problem.
The complexity of the Dijkstra's algorithm in each part is $O(M \cdot log(n))$*.*

Problem D: Knight

Statement:

There is a white knight on an endless chess board. Find the minimum number of moves needed by the knight in order to move from one given field to another.

Input:

First (and the only) line contains four integer numbers K_x, K_y, T_x, T_y, separated by a space. First two numbers represent the starting position of the knight, and the other two represent the position of the target field.

Output:

The output consists of one integer number, the minimal number of moves.

Constraints:

- $-10^{10} \leq K_x, K_y, T_x, T_y \leq 10^{10}$

Example input:

```
1 2 3 4
```

Example output:

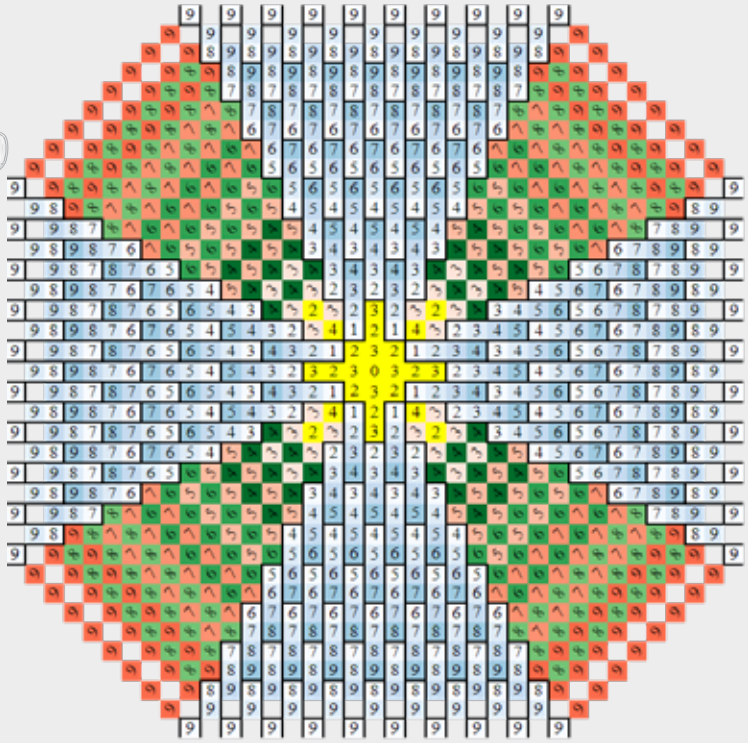
```
4
```

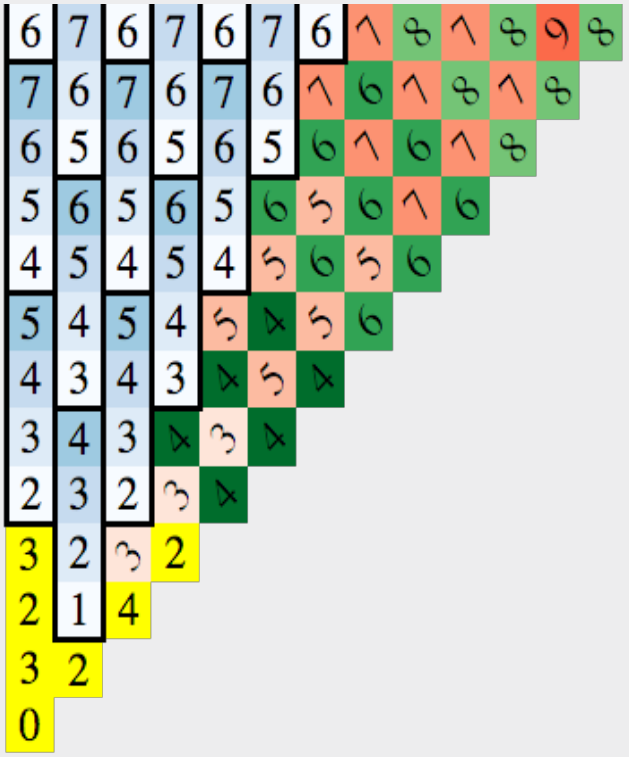
```
> Time and memory limit:  0.1s / 16MB
```

Solution and analysis:

First, we can translate both fields, and the solution won't change. So, we can assume that the knight is starting from the point $(0,0)$. *Also, we can change target coordinates, so the target is in the first quadrant. It's obvious also that* $f(T_x,\ T_y) = f(T_y,\ T_x)$.

So, every input (K_x, K_y, T_x, T_y) *could be transformed to input* $(0, 0, T'_x, T'_y)$, *where* $T'_y \geq T'_x \geq 0$, *which will have the same result.*





When looking for the results for the nearest fields, there are two patterns to notice:

- *The incrementing blue vertical groups of* 4
- *The red/green diagonals*
- *Yellow fields are special cases.*

After analyzing, we can come up with $O(1)$ solution for non-yellow fields:

```
var delta = x-y;
if (y > delta) {
    return delta - 2*Math.floor((delta-y)/3);
} else {
    return delta - 2*Math.floor((delta-y)/4);
}
```

About "magic" constants 3 and 4: We can notice that $y = 2x$ is splitting our solution into two cases. When the target field is on the left side of this line ("near" $y = 0$), we can move towards it and decrease x by 2 in each step by staying near $y = 0$, and when it's on the right side of this line, ("near" $y = x$) we can decrease $y + x$ by 3 for each move.

References:
https://stackoverflow.com/questions/2339101/knights-shortest-path-chess-question

Problem E: Billiard

Statement:

Fifteen billiard balls are placed randomly in the triangle rack. In one step you can switch any two balls. In order to start the game, you are required to place the balls correctly in as few steps as possible.

Here is the description of the correct ball placements: Balls are labeled with numbers from 1 to 15. The ball number 8 is black and must be placed in the position of the black circle (see images a and b). Balls $1-7$ are called solid, and balls $9-15$ are called striped. Position of all balls of either group must match all places denoted by an X on any of two images bellow, giving four types of correct placements in total.

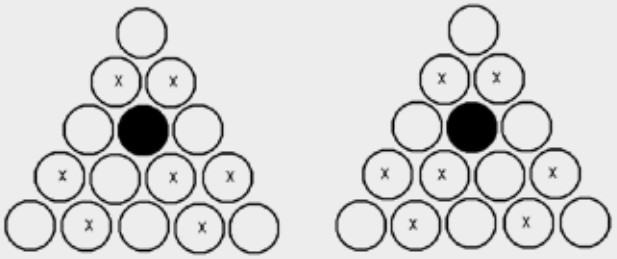


Image A and B

Input:

The first (and only) line contains a permutation of numbers 1 – 15, separated by a space. These numbers are ball labels read row by row in top down order, each row from left to right.

Output:

The output consists of one integer number, the minimal number of moves.

Constraints:

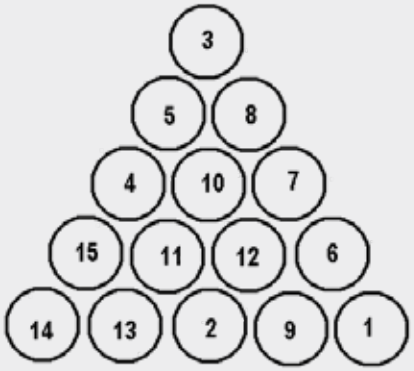
The input contains the permutation of numbers in range 1 to 15

Example input (image c):

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

Example output:

```
3
```



Image

```
> Time and memory limit:  0.1s / 16MB
```

Solution and analysis:

First, notice that if the black 8 *ball is not already in the center we will need to find the* 8 *ball and swap it with the current ball in the center. We will need to do this regardless of the chosen configuration.*

Now, the only thing that is left is to count the minimal number of swaps for any of the four configurations. If in the given configuration a stripped ball is not in its correct place that means, there is also a solid ball that isn't in the correct place. By swapping these two balls we will improve our order. Because of that, for every configuration we just need to count the number of striped balls that are not in their correct place.

Problem F: Bugs Bunny & Elmer Fudd

Statement:

Elmer Fudd's farm, way north of/from here, is well known for having the best carrots in the whole area. Elmer spends most of his time working on his farm and he would not be pleased even if someone just stepped into it. One day, Bugs Bunny and his bunny friends wanted to have a snack. Actually, they were not as hungry as much as they wanted to mess with Elmer. They went to his farm to eat as many carrots as they could. What they did not know was that Elmer had set a trap! The ground around the bunnies suddenly collapsed and each of them was left standing on a fragile looking pillar while the fire began to rage below, and you could see Elmer running with his gun… Help Bugs Bunny and his friends! Get as many bunnies as possible out of the farm and report the number of casualties.

The pillars on the farm are aligned in a grid with each pillar one unit away from the pillars to its east, west, north and south. Pillars at the edge of the farm grid are one unit away from safety, because as long as the bunnies are outside the farm, they can hide so Elmer cannot shoot them anymore. Not all pillars necessarily have a bunny. A bunny is able to jump onto any unoccupied pillar that is within d units of his current one (Euclidian distance between those two pillars has to be less than or equal to d). A bunny standing on a pillar within jumping distance of the edge of the farm may always jump to safety, but there is a catch…

Each pillar becomes weakened after each jump and will soon collapse and no longer be usable by other bunnies. Jumping onto a pillar does not cause it to weaken, only jumping off of it causes it to weaken and eventually collapse. Only one bunny can be on one pillar at any given moment.

Input:

Input begins with a line containing a single positive integer n representing the number of rows in the map of the farm, followed by a single non-negative integer d representing the maximum jumping distance for the bunnies. Two maps follow, each as a map of characters with one row per line.

The first map will contain a digit $[0-3]$ in each position representing the number of jumps the pillar in that position will sustain before collapsing (0 means there is no pillar there).

The second map will follow, with a 'B' for every position where a bunny is on the field and a '.' for every empty field. There will never be a bunny on a position where there is no pillar. Each input map is guaranteed to be a rectangle of size $n \cdot m$. The jumping distance is guaranteed to be in the range of d .

Output:

You should output a single line, containing the number of bunnies that could not escape.

Constraints:

- $1 \leq n \leq 20$
- $1 \leq m \leq 20$
- $1 \leq d \leq 4$

Example input 1:

```
3 1
1111
1111
1111
BBBB
BBBB
BBBB
```

Example input 2:

```
3 2
00000
01110
00000
.....
.BBB.
.....
```

Example input 3:

```
5 2
00000000
02000000
00321100
02000000
00000000
........
........
..BBBB..
........
........
```

Example output 1:

```
2
```

Example output 2:

```
0
```

Example output 3:

```
1
```

> Time and memory limit: 3s / 16MB

Solution and analysis:

We will convert this problem to a maximum-flow problem by representing the grid of pillars as a directed graph where the maximum flow is the number of the bunnies that escape.
Each two pillars within the jumping distance are connected with edges going both ways. Bunnies are represented with a source node that is connected to the corresponding pillars by directed edges with the capacity of one. The pillars that can reach safety are connected to the sink.
The restriction of the number of jumps from a pillar corresponds to a vertex capacity. In order to convert vertex capacities to edge capacities, each pillar will be represented with two nodes: an input node that accepts all edges ending at this pillar and an output one that contains edges starting from it. There will be one edge directed from the input to the output node, with its capacity set to the number of jumps that the pillar can survive.
The total number of bunnies subtracted by the maximum-flow of this network is the solution of the problem.
Each augmenting path can be found by breath-first search in $O(n \cdot m \cdot d^2)$. Since there are $O(n \cdot m)$ bunnies, there are $O(n \cdot m)$ iterations of the algorithm. Therefore, the total time complexity is $O(n^2 \cdot m^2 \cdot d^2)$.

Problem G: Robots

Statement:

Somewhere in space and time, the never-ending battle between the Good and the Evil is taking place. So far the Evil is winning on the eastern and northern front. The Good is retreating under heavy ionic fire and black hole grenades. Luckily, the Good stumbled upon an abandoned robot factory, the Good's last and only hope... Unfortunately, most of the robots the Good has found are rusty and broken. They are all out of function, sitting on a long straight assembly line. All robots have lasers that could be used either as a weapon or for communication with other robots.
After a quick inspection, it turns out that every robot can be repaired. Once a robot is repaired, it will repair and communicate with all robots within its laser scope. The repaired robots will in turn do the same. After two robots establish communication, the robot with a smaller laser scope will increase its scope to match the laser scope of the other robot. A robot's laser scope can be increased more than once.
The goal is to repair enough robots for the Good to defeat the Evil (although this may not always be possible). Since the repair is lengthy and energy consuming, the Good has time to manually repair only one initial robot. Also, some robots are more damaged than the others, meaning their repair is more energy consuming. To save energy, the Good needs to choose the least damaged robot which is capable of repairing enough robots to defeat the Evil.

Input:

The first line contains two positive integer numbers n and k, where n is the number of robots in the factory and k is the smallest number of repaired robots the Good needs in order to defeat the Evil. Next n lines contain information about robots: each line contains three real numbers p_i, d_i, e_i, where p_i is the position, d_i is the laser scope and e_i is the energy required to repair the robot i manually.

Output:

The output consists of one real number, printed with six decimal places – the minimal energy needed for initial repair, such that at least k robots end up repaired. If the Good cannot win, the output should be -1.

Constraints:

- $0 \leq k \leq n \leq 1{,}000{,}000$
- $0 \leq p_i, d_i, e_i \leq 1{,}000{,}000$

Example input:

```
3 2
1.2 10 5.3
5 1.54 1
13 10 3.14
```

Example output:

```
3.140000
```

> Time and memory limit: 5s / 64MB

Solution and analysis:

The problem is solved using dynamic programming.
For each starting robot we will find the set of robots that will get repaired. Let's call this the repair set of each robot. If we sort robots by their position. each of these sets will always be a range of robots. So, for each starting robot we will calculate the index of the furthest robot on the left and right we can repair.

First, we need to find the largest range (in an array) in each repair set, such that it does not contain robots that are larger (have larger laser range) than the starting robot but does contain the starting robot. Let's call this the cover set of each robot. The cover set of a robot can be obtained if we repeatedly repair its neighbor robots until the next one is larger or out of laser range. Notice that that the range of all repaired robots in the cover set is going to equal the starting robot. However, obtaining the cover set for each robot as described gives a quadratic complexity in the worst case.
In order to achieve linear complexity, we will determine the cover set for robots in order of increasing range. The smallest robot will always be the only one in his cover set. The second smallest will have the smallest one in his set if they are next to each other and the second one can reach him. The algorithm for finding the cover set of i^{th} robot is the following:
Initialize the cover set to contain only the i^{th} robot.
If the first robot outside the set on either side is smaller and in laser range, expand the cover set to also contain his cover set.
Repeat step 2 until no smaller robots can be reached.
The addition of the new set in step 2 is safe to perform since we know that all the robots it contains are smaller and in range. That cover set will always be computed since its starting robot is smaller than the current one.
This way of computing the cover set is faster since we are adding multiple robots at a time. To prove that it will have linear complexity, we have to notice that the current cover set is expanded only by adding other cover sets to it. Also, a single cover set can be added to the other ones at most two times. Once when it is reached by a larger robot from the left and once when it's reached from the right.

We can now use cover sets to compute the repair sets. Notice that the largest robot always has an identical cover and repair set. The cover set for the second largest robot is his repair set unless he can reach the largest one, in which case he can also reach the exact same set of robots the largest robot can. The algorithm for determining the repair set of the i^{th} robot is fairly simple. If it can reach a larger robot then he inherits the repair set from him. Otherwise, his repair set is exactly the same as his cover set. That is why the computing of repair sets is done for larger robots first.
Now that we have the repair set for each robot, we can easily determine the number of robots in each set as the difference of the index of the rightmost and leftmost robot increased by one. Then we find the cheapest robot that can repair at least k robots or print -1 if none are found.
The complexity of finding the cover and repair sets are both $O(n)$. However, sorting the robots takes linearithmic time which gives us the total complexity of $O(n \cdot log(n))$.

Problem H: String

Statement:

You are given a very, very long string. We will give you instructions what needs to be done with the string and ask you to find out what the string will look like after all these manipulations.

Input:

The first line contains a string of uppercase English alphabet letters of length n. Each other line contains a single command that affects the current string:
I, N, S - Insert the string S of length s starting from the index N, where $0 \leq N \leq [current\ string\ length] - 1$;
D, N_1, N_2 - Delete the substring starting from the index N_1, ending with the index N_2, where $N_1 \leq N_2$;
F, N_1, N_2 - Flip (reverse) the substring starting from the index N_1, ending with the index N_2, where $N_1 \leq N_2$;
All string indices are zero-based and always represent a valid position within the current string. The number of commands is k.

Output:

A single line containing the current string, after all commands are sequentially executed on the input string.

Constraints:

- $1 \leq n \leq 1{,}000{,}000$
- $0 \leq k \leq 10{,}000$
- $1 \leq s \leq 1{,}000$

Example input:

```
GOWAYUP
I,5,ELB
I,5,BC
I,0,BU
D,2,6
F,3,6
```

Example output:

```
BUBBLECUP
```

> Time and memory limit: 2s / 64MB

Solution and analysis:

As hard as it may seem, this task is all about managing intervals. We achieve that by having one big string S that we will concatenate all the input strings to, and a list of segments of string S that, when concatenated, give what the current string looks like.
The segments are in the form (i, j). If $i \leq j$ then the segment should be read from i to j, and $i > j$ means that the segment is reversed, and it should be read from i to j backwards.
One operation that we define is segment splitting. We use this operation if we want to split the segment that the i^{th} character belongs to into two pieces, such that the first segment contains all characters up to i^{th} and the second one all the remaining characters from that segment. Firstly, we find the segment by summing the lengths of all segments until the sum exceeds the given index i. Then we know that we found the right segment. We move all the remaining segments one position to the right and add one new segment next to the segment which the i^{th} character falls into and adjust the indices of those two.

Now, the three manipulations from the task are simple:
Inserting a new string at the index A: First split the segment that A belongs to and add a new segment next to it with indices from the string S that correspond to the newly concatenated string.
Delete the substring from the index A to the index B: Split the segments that A and B are in. Remove all the segments from the right segment generated from splitting around A to the left segment generated by splitting around B.
Reverse the substring from the index A to the index B: Split the segments that A and B are in. For all the segments from the right segment generated from splitting around A to the left segment generated by splitting around B swap beginning and end of the segment.
For each manipulation, the number of generated or deleted segments is $O(1)$ which means that after K manipulations we end up with $O(K)$ segments. Since we may pass through all segments at each manipulation, total complexity is $O(k^2)$.

Problem I: Tractor Disruptor

Statement:

In a galaxy far, far away, it is a period of civil war. Rebel spaceships, striking from a hidden base, have won their first victory against the evil Galactic Empire. However, the Empire struck back, and the main Rebel spaceship has been trapped by a positronic tractor beam. Therefore, the Rebels need an ionic tractor disruptor to set the main spaceship free. Trouble is, a regular ionic tractor disruptor, which they have plenty of, will not do. No, no, the Rebels actually need a negative ionic tractor disruptor and there is only one in the whole galaxy. The problem is that the spaceship that carries the negative ionic tractor disruptor has a broken odometer and Rebels are not sure if they have enough fuel to get to the destination. So, they want to calculate the correct number of light-years they have traveled.

The odometer is a device made of gears that rotate along the same axis and every gear has digits attached to it (a consecutive array of digits from 0 to max_digit). When all current digits on every gear are looked at as one number, the odometer displays the number of light-years traveled as an integer. However, the odometer has a defect – one (same) digit is broken on all gears. Usually, the gear proceeds from a digit d to a digit $d + 1$ or from a digit d to a digit 0 if d is the maximum digit on a certain gear. But for a broken digit, the odometer skips that digit and proceeds to the following digit on the gear. This defect shows up in all positions. For example, if the broken digit is 3, the odometer displays 45229 and the spaceship travels one light-year, the odometer reading changes to 45240 (instead of 45230).

The additional problem is that the maximum digit doesn't have to be the same on all gears. You have the information about what the maximum digit is for every gear. If the maximum digit on the gear 0 is a, on the gear 1 is b, on the gear 2 is c, on the gear 3 is d, then the maximum 4-digit number on the odometer is $dcba$. Then, after the spaceship travels another light-year, the odometer reading changes to 10000.

Your task is to calculate what would the odometer reading be if none of the digits were broken. You are provided with the incorrect odometer reading, the broken digit and the odometer description (maximal digit per every gear).

Note that the broken digit can be larger than some maximum digits. In these positions, the odometer works just fine. Also, if the broken digit is 0, the odometer does not make mistakes on leading zeros.

Input:

The first line of input contains two numbers: a positive integer n which represents the odometer reading, and a broken digit b. You may assume that the odometer reading will not contain the broken digit.

The second line contains 19 maximum digits from (least significant) position 0 to (the most significant) position 18. Note that the first maximal digit matches the right-most gear and so on.

Output:

Print the number of light-years that would be present on the odometer if none of the digits were broken.

Constraints:

- $1 \le max\_digit_i \le 9$
- $1 \le n \le 2^{63}-1$
- $0 \le b \le 9$

Example input 1:

```
45 3
8 7 6 5 4 3 2 1 9 8 7 6 5 4 3 2 2 2 2
```

Example output 1:

```
31
```

Example input 2:

```
45 0
8 7 6 5 4 3 2 1 9 8 7 6 5 4 3 2 2 2 2
```

Example output 2:

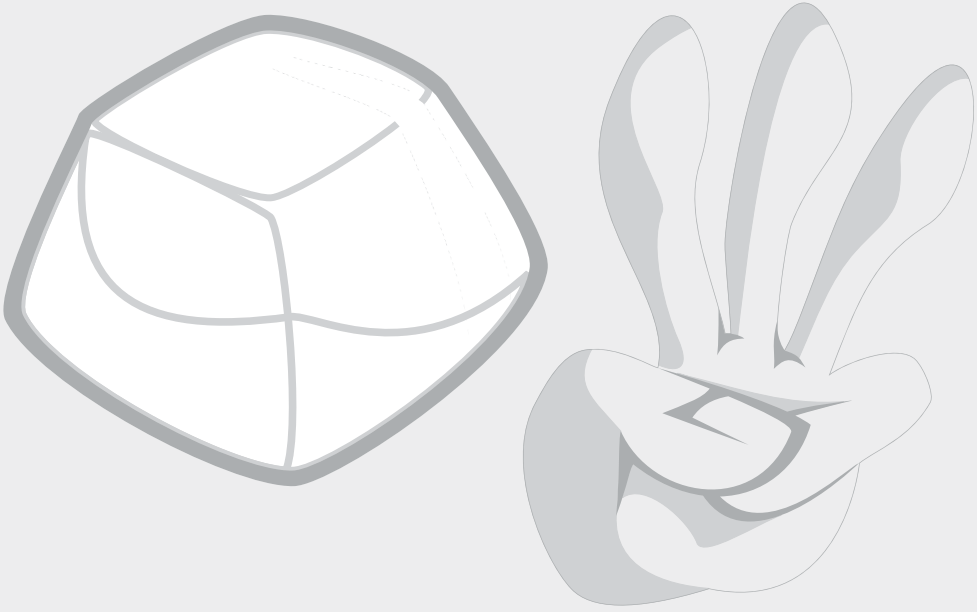
```
41
```

> Time and memory limit: 1s / 64MB

Solution and analysis:

Probably the simplest and the most logical solution is to convert the read number from the "broken odometer number system" to the decade number system and then convert the obtained number to the "correct odometer number system".
The first step: Let the number of different digits on i-th wheel on broken odometer is m_i (not counting the broken digit b) and the read number is r_i. Let k_i is number of possible values on i^{th} wheel greater than zero and lower or equal to r_i. So k_i equals r_i if r_i is lower than b, otherwise it is $r_i - 1$. Then, the read number converted to a decade number system is $dec = \sum_{i=1}^{19} k_i \prod_j^i m_j$.
The second step: We need to calculate the $weight$ of one increment on each wheel. It is equal to the multiplication of the number of different digits on all less significant wheels. Now, going from the most significant wheels to the less significant, it is easy to calculate the digit on each wheel so that the odometer value is equal dec.
The time complexity of both steps depends on the number of digits in the read number ($n = 19$), and it is $O(log(n))$.

bubble cup 3



Problem A: Brackets

***Statement*:**
You are given an array of n strings, and each string contains only open and closed brackets.
Find out if those strings can be sorted in such a way that after the concatenation of all strings, a valid arrangement of brackets is achieved (like as in a math expression after removing all other characters).

***Input*:**
The first line contains the positive integer n, the number of strings. Each of the next n lines contains a sequence of '(' and ')' characters, up to the end of the line.

***Output*:**
The output consists of one word:
"yes" (without quotes) if the required arrangement of strings exists
"no" if it doesn't exist

Constraints:

- *$1 \le n \le 100{,}000$*
- *Total number of all characters in all strings does not exceed* $10{,}000{,}000$ *(ten million).*

Example input:

```
3
(()
(
))
```

Example output:

```
yes
```

```
> Time and memory limit: 3s / 64MB
```

Solution and analysis:

In this problem each string is equivalent to a string starting with zero or more closed brackets, followed by zero or more open brackets. For example, underlined brackets are matching and can be removed from the string: " ())(()(()(", reducing it to: ")((". So, each string is completely characterized with two integer attributes: the number of unmatched closed brackets at the beginning of a string and the number of unmatched open brackets at the end of a string.

Let's introduce the following notation:

- $UO[i]$ – *Number of unmatched open brackets at the end of i-th string;*
- $UC[i]$ – *Number of unmatched open brackets at the beginning of i-th string;*
- $B[i] = UO[i] - UC[i]$ – *Bracket balance of i-th string, which can also be negative.*

In the previous example $UC = 1,\ UO = 2,\ B = 1$.

If $\sum_{i=1}^{n} B[i] \neq 0$, it is clearly impossible to arrange the strings as required.

Otherwise (if $\sum_{i=1}^{n} B[i] = 0$), we can first sort the strings according to the following criteria:

- *First we put all strings with positive (i.e. non-negative) balance, and then all strings with negative balance.*
- *UC should be increasing among strings with positive balance, and UO should be decreasing among strings with negative balance.*
- *If two strings with positive balance have the same UC (or two strings with negative balance have the same UO), we first put the one with higher B.*

For the global string (obtained by concatenation of given strings), we want to check that at each point the number of closed brackets does not exceed the number of open brackets, i.e. that balance at each position is non-negative.
It is not difficult to prove that for any two consecutive strings the suggested order maximizes the lowest balance over all positions in the global string. Consequently, if a solution exists, it can be obtained by sorting as described. Let's prove this.

We can look at the string as an ordered triple $(UC[i], UO[i], B[i])$. Let's assume that these strings are arranged in correct form. In other words:

$$UC[1] = 0$$
$$UC[i] \leq B[1] + B[2] + \cdots + B[i-1] \ , \ \texttt{for}\ \ i \in [2, n]$$

First, let us prove that a nonnegative balanced string can be moved in front of negative ones. The necessary and sufficient condition for this is to prove that, if we have two successive strings with indexes i and $i+1$, where $B[i] < 0$ and $B[i+1] \geq 0$, we can swap them. Denote $B[1] + \cdots B[i]$as $SUM[i]$. Now from $UC[i] \leq SUM[i-1]$ and $UC[i+1] \leq SUM[i-1] + B[i]$ we have that $UC[i+1] \leq SUM[i-1]$ and $UC[i] \leq SUM[i-1] + B[i+1]$ because $B[i]$ is negative and $B[i+1]$ is nonnegative. Of course, because this is a successive string, described transformation does not affect the rest of strings. With this operation we only increase the required differences.

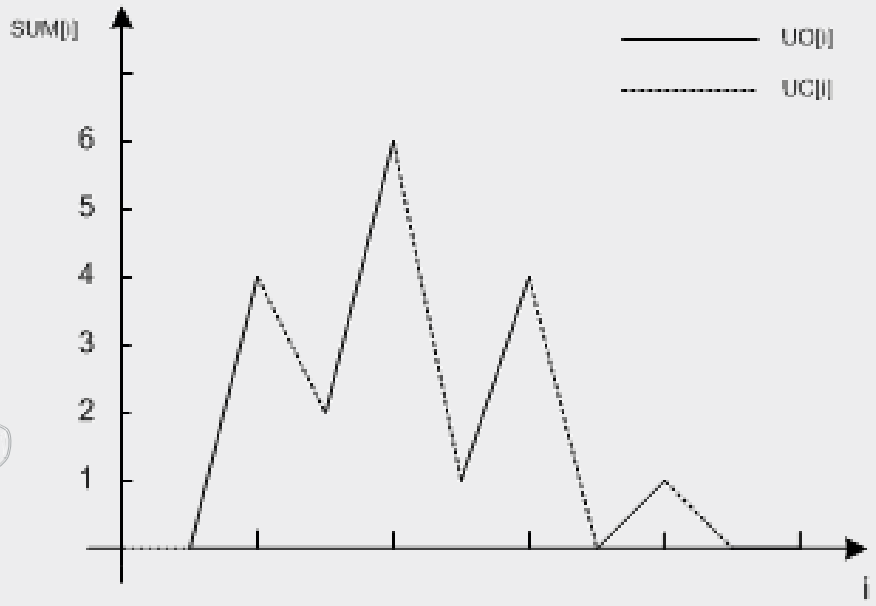


*Figure 1. We can look at the arrangement as **Dyck lattice path** with $UC[i]$ and $UO[i]$ as jups.*

Now we can look at the strings with positive balance and the strings with negative balance as two sub-problems for sorting. For strings with positive balances, if we swap successive strings so that one with smaller UC goes first, again we have only strengthened the inequalities. On the other hand, for negative ones this is not so obvious. But if we put this 'on paper' we get this (again for successive strings):

$$\begin{matrix} SUM[i-1] \geq UC[i] \\ SUM[i] \geq UC[i+1] \\ UO[i+1] \geq UO[i] \end{matrix} \quad \Rightarrow \quad SUM[i-1] \geq UC[i+1], \quad \text{because } SUM[i-1] > SUM[i]$$

$$SUM[i-1] + UO[i+1] - UC[i+1] \geq UO[i+1] + UC[i] - UO[i] \geq UC[i]$$

Using the above transformation over successive strings, starting from a proper arrangement we can generate a new proper arrangement – which is also the output of our sort with the above criteria.
Therefore, it is enough to check strings in described order. If for this order balance stays non-negative (and is zero at the end), the answer is "yes", but otherwise "no".

Implementation:

Read the strings and sort them as described,
Check whether balance is non-negative at all points in concatenated string and zero at the end.

Complexity:

Time complexity is obviously $O(n\,log(n) + N)$, where N is the total number of characters in all given strings. Memory complexity is $O(N + n)$.

Problem B: Cutting

Statement:

Given an integer m and an integer sequence a of length n, you have to split the given sequence into consecutive subsequences. The sum of elements in any subsequence must be less than or equal to m. Let M be the sum of maximal elements of the subsequences. Your task is to find the split that minimizes M.

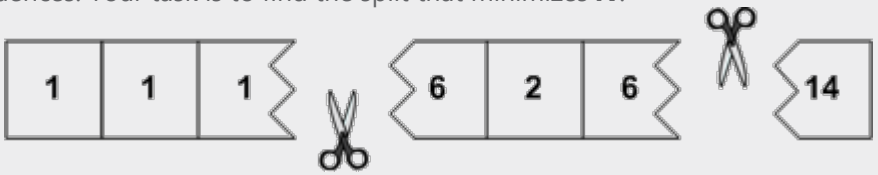


Figure 1. One possible cutting for the given example bellow

Input:

The first line contains two positive integers n and m, where n is the number of elements in the given sequence and m is the maximal allowed sum of elements in a subsequence. The following line contains n integers – elements of the sequence.

Output:

The output consists of one integer:
"-1" (without quotes) if a solution does not exist
otherwise, the minimal sum of maximal elements for any split(M)

Constraints:

- $1 \le n \le 100{,}000$
- $1 \le m \le 10^9$
- *All elements are in the range* $[0, 10^6]$.

Example input:

```
7 14
1 1 1 6 2 6 14
```

Example output:

```
21
```

```
> Time and memory limit: 1s / 64MB
```

Solution and analysis:

Firstly, we can see that a cutting exists if all elements of the given sequence are smaller than or equal to m. This is the first thing that we are going to check. From now on, we are assuming that all elements are not greater than m.

Let's start thinking backwards – not from the sequence itself but from the subsequences. If we denote subsequence $a_i, a_{i+1}, \dots, a_j$ as $a[i,j]$, then in the final cutting the last subsequence has the form $a[k,n]$ for some $1 \leq k \leq n$. Now we can say that the final solution is $max\{a_k, a_{k+1}, \dots, a_n\} + d[k-1]$, where $d[k-1]$ is the optimal cutting for the first $k-1$ elements of a. This cutting has to be optimal, because otherwise the cutting for the whole sequence would not be optimal either.

This smells like dynamic programming. Let's define array d of length n as:

$$d\,[k] = \texttt{optimal value of cutting sequence } a[1,k]$$

We have the following recurrent relation between the elements of d:

$$d\,[k] = \min_{bound[k] \leq i \leq k} \{d\,[i-1] + \max\{a[i,k]\}\},$$

where $bound[k]$ is the minimal index such that the sum of elements of $a[bound[k],k]$ is less than or equal to m. In other words, if the element a_k is the right boundary of some subsequence, then its left boundary has to be in the above segment. For the base of the dynamic programming algorithm, we can define $d[0] = 0$ and $d[1] = a[1]$. The final solution is stored in the element $d[n]$.

Implementation:

The tricky part of this problem was implementation. Let's see how we can initialize bounds fast. Array $bound$ is non-decreasing ($bound[k] \leq bound[k+1]$). When we want to initialize the element $bound[k]$, we only have to look in the segment $[bound[k-1], k]$. If we accumulate the current sum in this segment, the initialization of the whole array can be implemented in linear time.

```
==========================================================================================

bound [1] = 1;
currentSum = a [1];
for k = 2 to n do
        bound [k] = bound [k - 1];
        currentSum = currentSum + a [k];
        while (currentSum> m)
                currentSum = currentSum - a [bound [i]];
                bound [i] = bound [i] + 1;

==========================================================================================
```

Algorithm for bound initialization

What about array d? Naive implementation of the above recurrent relation leads to time complexity of $O(n^2)$, which is very slow for our constraints. The key observation is that we do not need all indices from the segment $[bound[k], k]$ when we want to find a minimum. We only need indices from the set

$$I_k = \{k, bound[k]\} \cup \{i \in [bound[k]-1, k-1] | a_i > a_{i+1}, \dots, a_k\}$$

Therefore, we have to check for boundaries and only for elements that are strictly greater than ones before them. This is intuitively clear, because if we have some maximum in a subsequence we want to stretch to the left as long as we can.
We can store these indices in a list. When we move from k-th to $(k+1)$-th element, we remove only some elements from the head and some elements from the tail of this list. From the beginning we are going to remove indices that are smaller than $bound[k+1]$. Since every index in the list represents an element that is greater than the ones before, from the end of the list we are going to remove indices if the corresponding elements are less than or equal to $a[k+1]$. After that we are going to add new element k at the end of the list. All of this is possible because both the indices in the list and their corresponding elements are sorted in a strictly increasing order.
And what about the minimum of these elements? Theoretically, this list can be very long. Well, we are going to store values $d[i] + max\{a[i+1, k]\}$ in a heap structure (all of them except for boundaries). When we move to a new element, as we remove something from the list, we remove the corresponding element from the heap. In the end, only for the last element of the list, which had the value $max\{a[last+1, k-1]\}$ before adding the new one, is going to change – it becomes $a[k] = max\{a[last+1, k]\}$.

```
d [0] = 0 and d [1] = a [1];
add in heap (1, a [1]);       // index and value
add in list 1;                // index of elements in heap
for k = 2 to n do
        while first index in list is less than bound [k]
                remove it from heap;
                remove it from list;

        while last element in list is less than or equal to a [k]
                remove it from heap;
                remove it from list;
        if (heap is not empty)
                remove last element lastElement from heap;
                add inheap(lastElement, d [lastElement] + a [k]);

        maxInBound = max (a [k], a [firstElementInList]);
        d [k] = max (a [k] + d [k - 1], d [bound [k] - 1] + maxInBound, min in heap);
        add in heap (k, d [k]);
        add in list (k);
```

Pseudo-code for described algorithm

Complexity:

Initialization of bounds is linear (as we have seen). Every element from the sequence is going to be added to the heap (and list) only once and removed from it at most once. Initialization of elements $d\ [k]$ requires one call for finding a minimum in the heap. This leads us to total time complexity of $O(n \log n)$.
Memory complexity is $O(n + MaxElement)$, because we must store information of positions in heap structure. Also, we have to pay attention to cumulative sums and use int64 for storing this information.

Test data:

Test corpus for this problem contains 30 test cases. Short description of test cases is given in Table 1.

Num	n	m	maximal $a[i]$	Solution	Description
01	10	20	15	37	By hand
02	20	1000	46	46	You don't need to cut
03	100	10000	978	5621	Random
04	1000	100000	32746	-1	-1
05	100	1000	452	452	Sum is equal to M
06	10000	100000	100000	834993296	Every element is one subsequence
07	50000	100000000	132767	7830539	~ 50 subsequences of 1.000 elements
08	99999	100000000	132867	15540259	~ 100 subsequences of 10.00 elements
09	99999	98765432	19753	296266	~ 10 subsequences of 10.000 elements
10	99999	99999999	40007	1159863	~ 10 subsequences of 10.000 elements
11	99999	99999999	32768	556893	Random
12	99999	99999999	532767	273208529	~ 1000 subsequences of 100 elements
13	99999	99999999	32768	556910	changing big - small subsequence
14	100000	7654321	123456	12345600000	Every element is equal to M
15	1	100	50	50	One element
16	100000	98765432	999987	479320035	Random monotonic subsequences
17	99999	100000000	9999	9999	Many zeros
18	100000	10000	9999	999900000	Monotonic down subsequences
19	99999	67834589	987655	-1	One element M + 1 and all ones
20	80000	100000000	532767	221021568	~ 10 subsequences of 10.000 elements
21	90000	10000	6012	92936424	Monotonic up subsequences
22	999999	9999999	999998	4064318963217351	Monotonic down subsequences
23	80000	100000000	999982	306422910	Random monotonic subsequences
24	80000	100000000	100	100	Random small
25	80000	100000000	49999	1449530	~ 20 subsequences of 10.000 elements
26	30000	100	0	0	All zeros
27	10000	1000	999	9990000	All equal to M – 1
28	100000	100000000	999996	355237902	Random monotonic subsequences
29	100000	100000000	504243	528239	~ 3 subsequences of 30.000 elements
30	100000	100000000	532767	275873972	~ 1000 subsequences of 1000 elements

Problem C: Extrema

Statement:

Let's define a function f as $f(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i$, where $a_i \in [0,1]$ and $\sum_{i=1}^{n} a_i = 1$.
Given two points X and Y from R^n and value $f(X) = C$, find minimum and maximum value for $f(Y)$.

Input:

The first line of input contains the number n. The second line contains numbers $x_1, \dots, x_n$ separated by a space. The third line contains $y_1, \dots, y_n$. The final line contains the number C. It is guaranteed that there will always be coefficients a_i for which $f(X) = C$ satisfying the above conditions.

Output:

The first line of output should contain minimum value for $f(Y)$ rounded to two decimal places, and the second line should contain the maximum value for $f(Y)$, also rounded to two decimal places.

Constraints:

- *$1 \le n \le 100{,}000$*
- *It is guaranteed that there will always be coefficients a_i for which $f(X) = C$ satisfying the above conditions.*

Example input:

```
3
0 2 1
0 0 1
0.75
```

Example output:

```
0.00
0.75
```

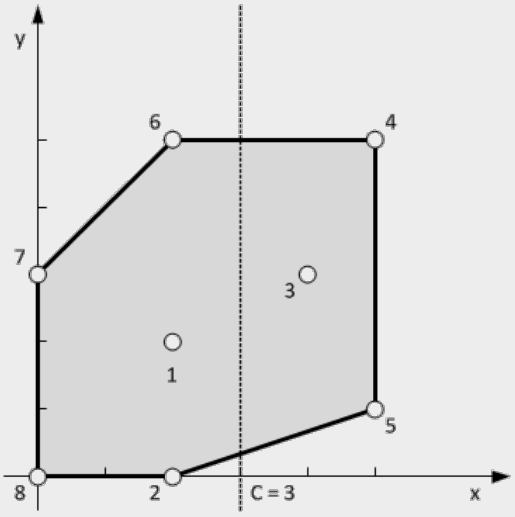
> Time and memory limit: 1s / 64MB

Solution and analysis:

This problem requires some math skills. At first sight, it seems to be a kind of linear programming problem, but it can be solved quite elegantly.
We have a function f and we know that it has the form $f(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i$, where $a_i \in [0,1]$ and $\sum_{i=1}^{n} a_i = 1$. The coefficients satisfying these conditions are called barycentric. We can easily spot the following property in one-dimensional space: given $x_1, \dots, x_n$ and x, $x \in [min\{x_i\}, max\{x_i\}]$ there exists a function f as defined above such that $x = f(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i$. It is enough to vary coefficients for minimum and maximum of x_i, the rest can be 0. Now we know that C must be between these two values.
Extending this to the two-dimensional case is harder. This is stated by the following theorem.

Theorem:

Given points $(x_1, y_1), \dots, (x_n, y_n)$ and $(x, y)$from R^2, (x, y) are in the convex hull of $(x_1, y_1), \dots, (x_n, y_n)$ iff there exists a function f as defined above such that $(x, y) = \big(f(X), f(Y)\big)$, where $X = (x_1, \dots, x_n)$ and $Y = (y_1, \dots, y_n)$.
Again, we can accomplish this by varying just the coefficients a_i for points which are vertices of the convex hull, the rest can be 0.



From the theorem, we conclude that the possible values for $f(Y)$ are just projections of the points which belong to the convex hull to the y-axis. The additional constraint $f(X) = C$ restricts the set of possible points to the ones which lie on the intersection of the convex hull and the line $x = C$. This intersection is a segment (or just a point in extreme case), so the final solution will be the boundaries of this segment.

Figure 1. Point and corresponding convex hull for points $X = (2,2,4,5,5,2,0,0)$ and $Y = (2,0,3,5,1,5,3,0)$ and value $C = 3$

Implementation and complexities:

The described idea can be implemented easily: find the convex hull for the given points and find where the line $x = C$ intersects it. Finding the convex hull has the complexity of $O(n \log n)$. Finding the intersections is linear, because we only need to check for consecutive vertices of the convex hull. This leads to the final complexity of $O(n \log n)$.

Problem D: Interval Graph

Statement:

For a set of closed intervals on real line, one the can construct an interval graph. Represent each interval with a different graph vertex and connect two vertices if and only if two corresponding intervals have common points.
Does the given tree represent an interval graph for some set of intervals?

Input:

The first line contains positive integer n - the number of nodes in a tree. The nodes are numbered by IDs: $0, 1, 2, \ldots, n-1$.
The node 0 is the root node of the tree.
The next n lines describe children for all nodes.
Line i (each od n lines) lists all children of the node with ID i.
The first integer in the line is c_i, the number of child nodes of node i. The next c_i integers in the same line are IDs of those child nodes.

Output:

The output consists of one line:
"yes" (without quotes) if the given tree represents an interval graph
"no" if it doesn't

Constraints:

- *1 ≤ n ≤1,000,000*

Example input:

```
3
1 2
0
1 1
```

Example output:

```
yes
```

Example input:

```
7
31 2 3
14
15
16
0
0
0
```

Example output:

```
no
```

> Time and memory limit: 3s / 64MB

Solution and analysis:

First, note that no three intervals can have a common point. If that were the case, the interval graph would have a triangle and wouldn't be a tree.
Now consider one interval and all intervals that have common points with it.
The intervals that are nested in that interval cannot therefore have any more common points with other intervals. They generate only one edge in the interval graph.
Intervals that are not nested contain one or both end points of the interval we are considering, therefore we can have no more than two intervals that have common points with considered interval and are not nested in it.
So, if we prune all one-edge sub-graphs corresponding to nested intervals, each vertex in the remaining graph can have at most degree two. In other words, the graph on Figure 1. can't be a sub-graph of the pruned graph.

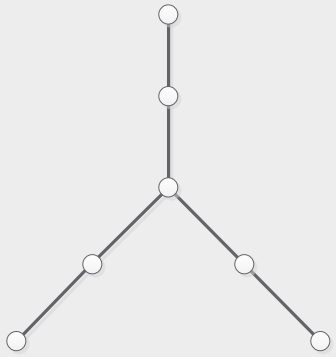


Figure 1. Forbidden structure for interval tree

We can see that if the tree satisfies this property, it is an interval graph. In the pruned graph every edge has degree one or two (if the graph is connected), therefore it is just a sequence of edges. We can therefore construct a sequence of intervals that correspond to this graph and include nested intervals for additional one-edge sub-graphs that were pruned.
Conclusion is that not having the graph on Figure 1. for a sub-graph is a necessary and sufficient condition for the tree to be an interval graph.
Interval trees are a very important subclass of intersection graphs and perfect graphs. The generalization of above statement is a famous result of Lekkerkerker and Boland given below:
Theorem. A graph is an interval graph if and only if it contains none of the graphs shown in Figure 2. as an induced sub-graph.

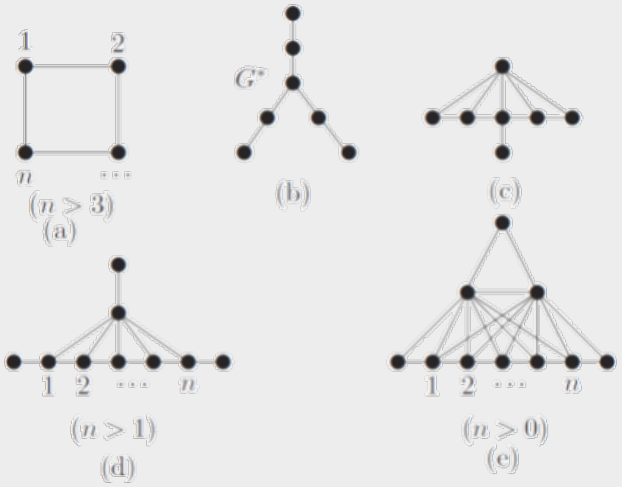


Figure 2. Forbidden structures for interval graphs

Implementation:

For the nodes on the first two levels, the described condition is equivalent to not having more than two sub-trees of depth one or more.
For nodes on deeper levels, the described condition is equivalent to not having more than one sub-tree of depth one or more.
This check could be easily done by depth-first search in linear time.
For simplicity, this could be broken into the following steps:

1. *Traverse tree and for each node calculate maximal depth of the sub-tree under it;*
2. *Traverse tree and for each node calculate the number of children sub-trees with depth one or more;*
3. *Traverse tree and for each node, check the condition taking in consideration the level of the node.*

These steps could be done in one tree traversal.
Since depth of the tree can be up to 1,000,000, recursive DFS cannot be used due to stack limitations. Iterative DFS is not much harder to implement. For techniques on how to refactor recursion to iteration, a good resource is The Art of Computer Programming.

Complexity:

Time and memory complexity for DFS are $O(n)$and $O(log(n))$ respectively but storing the tree structure requires $O(n)$ memory. Overall, both memory and time complexity are linear: $O(n)$.

Problem E: Nice Subsequence

Statement:

Given an array a of n integers, find the longest nice subsequence of consecutive elements.
The subsequence $a[i], \ldots, a[j]$, $i < j$, is *nice* if:

1. $a[i] \leq a[j]$
2. $a[i] \leq a[k] \leq a[j]$, for all $k \in [i, j]$

Input:

First line contains one positive integer n, where n is the number of elements in the given array. Each of the next n lines contain one integer which represents an element of the array.

Output:

The output consists of one integer number:
"-1" (without quotes) if nice subsequence doesn't exist
Length of the longest nice subsequence

Constraints:

- *$2 \leq n \leq 1{,}000{,}000$*
- *$0 \leq a[i] \leq 2 \cdot 10^9$*

Example input:

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

Example output:

```
5
```

> Time and memory limit: 3s / 64MB

Solution and analysis:

First, we can see that a nice subsequence doesn't exist if the array is monotonically decreasing.
A naïve solution would be to find the longest nice subsequence ending with each index j and then find the longest of those, but that's too slow. We need to somehow use the information we have already obtained to speed up the search. For that purpose, we will create a stack of nice subsequences we have obtained (we will call it S). It will initially be empty.

We traverse the array, starting from the right. We want to keep some properties of S invariant:

- *All subsequences on the stack will either be nice or have length 1.*
- *The subsequences on the stack will always be sorted so that their left boundary values (minimums) are decreasing (the largest value is at the top), and their right boundary values (maximums) are increasing (the smallest value is at the top).*
- *The subsequences will be sorted by their left boundary and they will not overlap each other.*

We will always keep a "current" monotone subsequence, which we will denote c, and as we traverse the array from end to beginning, as long as the values of the elements are decreasing we can keep adding them to c. When we arrive to an element that breaks monotonicity (it is larger than its neighbor on the right), we want to push c to the stack (monotone sequences are nice by definition) - but first we have to perform some operations in order to ensure that S will continue to have the desired properties. Namely:
If $min(c) \leq min(top(S))$ and $max(c) \leq max(top(S))$, the subsequence which spans from the left boundary of c to the right boundary of $top(S)$ is also nice, so we can expand c to match this subsequence and pop from S.
If $max(c) > max(top(S))$, we can discard $top(S)$, because it means that any subsequence we find in the future cannot be nice if it stretches further than the right boundary of c.
Either of these steps can be repeated several times. Finally, when $min(c) > min(top(S))$d $max(c) \leq max(top(S))$, we push c to the stack and continue the traversal.
It is not hard to check that the properties of S we have highlighted will continue to hold after any of these steps are performed. Of course, we will keep a variable holding the best result we have found so far, and if we come across a nice subsequence longer than that value during any of these steps, we update the result.
Let's now try to sketch a proof that, after the entire array is processed, we will have found the correct result. We are only looking at subsequences on the stack, and we know that they will always be nice, so we will obviously never return a result larger than the correct one. What remains to be shown is that the longest subsequence will always be found by this algorithm.

If the longest nice subsequence is the subsequence $[i, j]$, we know that $a[i-1] > a[i]$ (or i is the first element) and $a[j] > a[j+1]$ (or j is the last element). So, both $a[i-1]$ and $a[j]$ will cause breaks in monotonicity, although in general they won't be in the same monotone subsequence. This means that we need to make sure that the sequence starting with i (s_i) will eventually merge with the sequence ending with j (s_j). But that is simple: $min(s_i) \leq min(top(S))$ because otherwise $[i, j]$ would not be nice at all, so the subsequences between s_i and s_j on the stack will either get merged into s_i or be discarded. Finally, when s_j becomes the top of S, s_i will merge with it, because, again, $[i, j]$ being nice implies that $max(s_i) < max(s_j)$. This means that $[i, j]$ will definitely be processed at some point, which means that the proof is finished.

Complexity:

Since the number of monotone subsequences in the array cannot be larger than n, the main part of the algorithm essentially consists of $O(n)$ "push" and $O(n)$ "pop" operations on the stack, making the overall time complexity of the solution linear.

Buxkdop F: Panuql

Tnanopozn:

A boufoin panuql qt a panuql jqnc qznomou odopoznt qz jcqic oaic njx zoqmckxuqzm odopoznt auo uodanqyodw buqpo (ix-buqpo), azv nco aktxdrno yadro xf oaic odopozn qt muoanou ncaz xzo. Oaic odopozn cat rb nx fxru zoqmckxut.
Wxr auo mqyoz a panuql $A_{m\times n}$ jqnc qznomou odopoznt $a_{ij}, 1 \le i \le m, 1 \le j \le n$.
Icoie qf ncouo olqtnt a boufoin panuql $B_{m\times n}$ jcouo b_{ij} vqyqvot a_{ij} fxu oyouw $1 \le i \le m, 1 \le j \le n$.

Qzbrn:

Nco fqutn dqzo ixznaqzt bxtqnqyo qznomout m, n — nco vqpoztqxzt xf nco panuql $A_{m\times n}$. Oaic xf nco zoln m dqzot ixznaqzt a tohrozio xf n qznomout tobauanov kw tbaiot, uobuotoznqzm odopoznt a_{ij} xf nco panuql $A_{m\times n}$.

Xrnbrn:

Xrnbrn ixztqtnt xf xzo vqmqn: :
"1" (jqncxrn hrxnot) qf a boufoin panuql olqtnt
"0" qf qn vxotz'n olqtn

Ixztnuaqznt:

- *$1 \le m, n \le 80$*
- *$2 \le |a_{ij}| \le 1{,}000$*

Olapbdo qzbrn:

```
2 2
6 4
10 9
```

Olapbdo xrnbrn:

```
1
```

Olapbdo qzbrn:

```
1 3
4 6 9
```

Olapbdo xrnbrn:

```
0
```

DON'T PANIC ☺

```
> Nqpo azv popxuw dqpqn: 0.5t / 64PK
```

Solution and analysis:

The statement of this problem was put through a cipher and presented to the competitors in encrypted form, which they had to decipher before they could start solving the actual problem.
Since the other eight problems were unencrypted and the texts had the same basic shape, starting by trying to compare the ciphertext with them was a good idea. It is noticeable that certain words repeat multiple times and looking at the other problems reveals that they probably correspond to certain important phrases, for example "input", "output", "integer", with unchanged number of letters in a word. This implies that the cipher is a simple substitution cipher, and reveals the encrypted values for many of the letters. Going in this direction starts producing text that is already somewhat intelligible, so we are encouraged to continue: we know where keywords such as "statement" and "problem" are located, and we can guess the remaining letters in frequently occurring words: "the", "line". The rest is easy: most words will have only one of two letters left encrypted, and simple common sense should be enough to finish the job. Also, a cool thing is that when you translate "DON'T PANIC" you get "LET' MATCH".

Translated problem looks like this:

Statement:

A perfect matrix is a matrix with integer elements in which each two neighboring elements are relatively prime (co-prime), and the absolute value of each element is greater than one. Each element has up to four neighbors.
You are given a matrix $A_{m\times n}$ with integer elements $a_{ij}, 1 \le i \le m, 1 \le j \le n$.
Check if there exists a perfect matrix $B_{m\times n}$ where b_{ij} divides a_{ij} for every $1 \le i \le m, 1 \le j \le n$.

Input:

The first line contains positive integers m, n — the dimensions of the matrix $A_{m\times n}$. Each of the next m lines contain a sequence of n integers separated by spaces, representing elements a_{ij} of the matrix $A_{m\times n}$.

Output:

Output consists of one digit:

- *"1" (without quotes) if a perfect matrix exists*
- *"0" if it doesn't exist*

The condition that neighboring elements are relatively prime can be stated in the form that neighboring elements cannot have the same prime factors. Since b_{ij} divides a_{ij}, candidates for prime factors for b_{ij} are prime factors of a_{ij}. Clearly, if there exists a perfect matrix, we can transform it to another perfect matrix where all b_{ij} are prime numbers, by omitting all but one prime factors of b_{ij}. Resulting perfect matrix has all prime elements and neighboring elements are different. Therefore, it is sufficient to check if a perfect matrix with prime elements and described properties exists to get the final answer.
In view of these conclusions, problem can be restated in the following way: For every element in the original matrix A, pick one of its prime factors, so that two neighboring elements have different factors picked. This is closely related to the graph coloring problem, which is NP, so backtrack is the way to go.

Constraints:

- *1 ≤ m, n ≤ 80*
- *2 ≤ |a_{ij}| ≤ 1,000*

Implementation:

The most naive backtrack implementation is too slow. A few improvements can be made. Prime numbers up to 1,000 can be pre-computed and included in the source file.
All numbers up to 1,000 can also be factored to prime factors offline and included in the source file. If some element a_{ij} has a unique prime divisor among its neighbors, this prime divisor can be picked for the perfect matrix.
If some element a_{ij} has only one prime divisor, this prime divisor cannot be picked from its neighbors.
The last observation is the crucial one for speeding up the backtrack algorithm. What should be noted here is that once a prime divisor is removed from the list of possible primes for all neighboring elements, the process can continue if one of the neighboring elements is left with only one prime divisor. This can propagate as long as there are changes made.

Complexity:

Time complexity is exponential. Memory complexity in most implementations shouldn't be larger than $O(mnp)$, where p is the number of different prime factors.

Problem G: Operations

Statement:

You are given an array of n characters in the form of $DSDSDS \dots DSD$ (n is odd) where:

- $D \in$ {'0'...'9'} and
- S ∈ {'+', '-', '=', '>', '<'};

Find out the maximal number of non-overlapping correct expressions (a correct expression is a substring of the given string which starts and ends with a digit, has exactly one comparison operator ('=' or '>' or '<'), and is mathematically correct).

Input:

The first line contains a positive integer n. The next line consists of n characters in the form described above (without any spaces between characters).

Output:

The output consists of the integer number which represents the maximal number of non-overlapping correct expressions.

Constraints:

- *n is odd*
- *n < 5,000,000*

Example input:

```
7
7-5<3=5
```

Example output:

```
1
```

Example input:

```
11
2+5<6-4<5=3
```

Example output:

```
2
```

> Time and memory limit: 3s / 64MB

Solution and analysis:

It is not hard to check that the maximal number of correct expressions can be achieved using the following algorithm (Greedy algorithm):

- *Find the first comparison operator for which we can obtain the correct expression, take the correct expression which contains that comparison operator and for which the rightmost character has the smallest index in the original array.*
- *Start looking for new correct expression from the position of first digit after the previously found correct expression.*

Checking if it exists and finding the optimal correct expression (optimal in the meaning described above) that contains some fixed comparison operator and that starts from some particular position (on the left side of comparison operator) can be done in the following way:
First, we calculate all possible values on the left side of the comparison operator. Then we scan digit by digit on the right side of the comparison operator, calculating the value of the expression on the right side by considering the digit and checking if that value, with the comparison operator and any value from the left side, gives a true statement.
Checking if a value from right side, the comparison operator and any value from the left give a true statement can be done by sorting values from the left side in non-decreasing order. Then, if the comparison operator is '=', use binary search to check if that value exists in a set from left. If the comparison operator is '<', check if the value from the right is greater than the first value from left in the sorted array, and in case when the comparison operator is '>', check if the value from the right is less than the last value from the left in the sorted array. Alternatively, since the minimal possible value on the left is $-9 \cdot \frac{n-1}{2}$ *and the maximal is* $9 \cdot \frac{n-1}{2}$*, another approach would be to have an array of* $9 \cdot (n-1) + 1$ *elements, so that for each value calculated on the left we join one element of the array, and while filling that array we can calculate the minimal and maximal value from all those that we were using in filling, so we can easily check if a value from the right exists on the left, just by looking in the corresponding place in the new array. If the value from the right is greater than some value from the left we can check using min, and whether the value from the right is less than some value from left we can check using max. We can also use min and max for initializing array for each new comparison operator.*

Example:

`3 + 8 - 6 = 2 + 4 < 3 + 2` *First, we calculate values on the left side of '='. These are 6,2,5. Then we go from '=' to the right. The first potential value is 2. Check if it is found in the set from the left. If it is, meaning that we found one correct expression, start looking for a new one from '4'. The next comparison operator is '<'. The only value on its left side is 4. The first digit on the right is 3. Considering that 3 is not greater than any value from the left (in this case just 4) we continue. The next digit is 2, so the next potential value is 5. Since 5 is greater than 4, we do have a new correct expression. We get to the end of the array, so the maximal number of non-overlapping correct expressions is two.*

Complexity:

It is obvious that the complexity of the solution which uses a sorted array is $O(n \log(n))$ *and that the complexity of the second solution is* $O(n)$.
Memory complexity is $O(n)$ *in either case.*

Problem H: Travel 'n' sleep

Statement:

You are the manager of a company and you want to send some of your employees to a big company meeting, which starts t days from now. The city where the meeting will be held is very far away from your headquarters, so they will have to travel for a couple of days, passing through some other cities and making pauses to sleep and rest during the journey. You have a map that assigns numbers between 1 and n to the cities and shows which of these cities have direct routes between each other. All the employees start from your headquarters (city 1) on the first day. On any given day, each employee can choose either to travel between two connected cities or to stay where he is and rest, and they all have to reach the meeting place (city n) and must not be late for the meeting.

There is just one small problem: your employees hate each other, so you can never allow two or more of them to be in the same city at the same time (except at the start and the end of their journeys, of course). It is allowed for someone to enter a city on the same day when someone else is leaving, however. You kind of hate all of them too, so you don't want to allow anyone to stay in your headquarters or to return there during the journey.

The meeting is quite important, so you would like to send as many people there as possible, and now you want to calculate exactly how many is that.

Input:

The first line contains three numbers, n, t and m. Each of the following m lines contain two different integers, the numbers of connected towns. All routes are two-way.

Output:

The output consists of exactly one non-negative integer, the maximal number of people that can reach town n from town 1 in t or less days.

Constraints:

- $2 \leq n \leq 50$
- $1 \leq t \leq 30$
- $1 \leq m \leq 500$

Example input:

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

Example output:

```
2
```

Explanation:

On the first day, the first person can go to city 2 and the second can go to city 3, and they will both reach city 4 on the second day.

> Time and memory limit: 1s / 64MB

Solution and analysis:

Looking at the problem statement carefully, it is noticeable that this problem is fairly similar to the problem of finding maximum flow in a graph. There are a few difficulties, however. Paths in our graph depend on a time component, while maximum flow assumes that edges have constant capacities. Also, we need to make sure that only one path can include any single vertex at a point in time. So what we need to do is try to find a way to transform our graph into one that is more suited to the max-flow constraints.

First, we will transform every vertex v of the original graph (except one!) into t vertices of the form (v, t_i), with the idea that one vertex of the new graph will represent a single point in space and time. With this, the original graph is turned into a graph in which we always know whether a particular route is available or not. To be more precise, for each edge uv in the original graph, the new graph will have directed edges from vertex (u, t_i) to $(v, t_i + 1)$ and from (v, t_i) to $(u, t_i + 1)$ for t_i between 0 and $t - 1$. We also have to account for the possibility of staying in the same city on a particular day, so every vertex (u, t_i) should also have an edge towards $(u, t_i + 1)$ for $t_i < t$. We will not do this for city 1, however, in order to eliminate staying in this city or returning to it later.

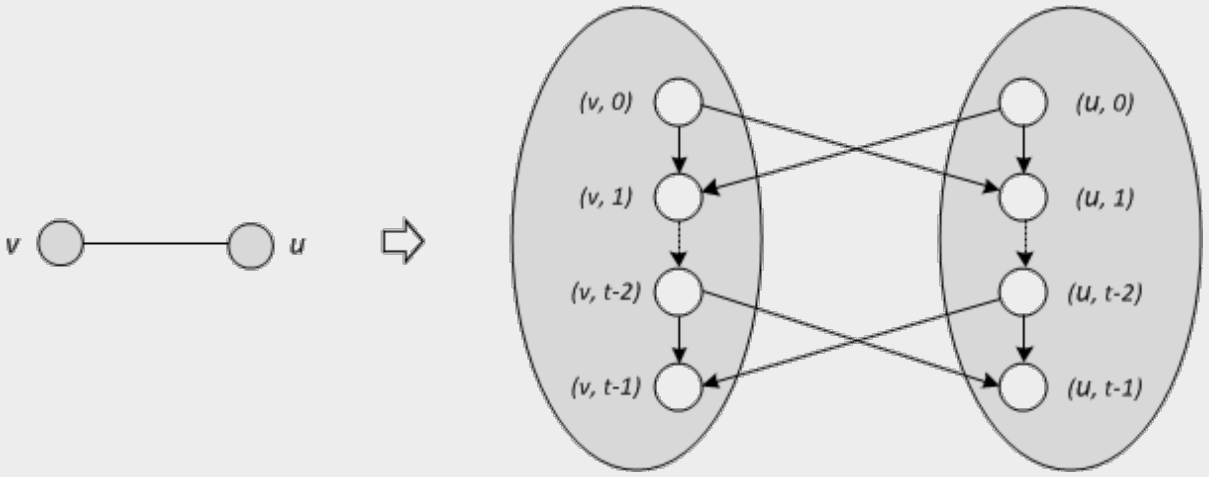


Figure 1. Vertex transformation with time parameter

Now we have a graph we can traverse without paying special attention to the time component. The other problem is ensuring that no two people can be in the same city on the same day, and we can do this using another easy trick to transform the graph: we split every vertex $w = (v, t_i)$ into an "in" vertex w_{in} and an "out" vertex w_{out}, and add an edge of capacity 1 from w_{in} to w_{out}. All edges that went to v should be redirected to w_{in}, while all edges that went out from w should now start from w_{out}.

The only thing remaining is to define the starting and the finishing vertex (the source and the sink) for the flow in our new graph. The source is easy: it is the out vertex corresponding to city 1, where everyone has to start. We do not have a single finishing vertex, however - all vertices corresponding to city n are valid finishing points. We will get around this by adding yet another (!) vertex, v_{sink}, and adding edges of unlimited capacity from all vertices $(v, t_i)_{in}$ to v_{sink}. (If "unlimited" is a problem for the computer to understand, any relatively large number will do.) It is now ensured that the solution to the problem is the maximum flow of the transformed graph.

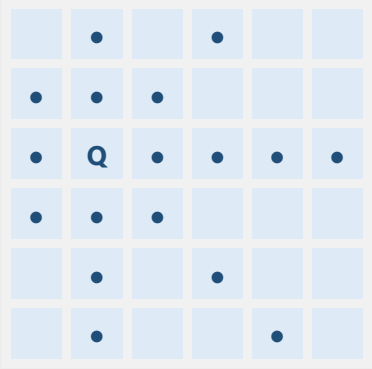
Complexity:

The constraints are such that most standard algorithms for finding maximum flow will work, so feel free to pick your favorite one. For example, the Edmonds–Karp algorithm is relatively easy to implement, and its complexity is $O(V \cdot E^2)$for a graph with V vertices and E edges. The number of vertices in the transformed graph is approximately $V = 2nt$, while the number of edges is approximately $E = (m + n) \cdot t$. This looks like bad news, but a closer look at the algorithm reveals that it consists of iterating breadth-first searches ($O(E)$ time), and that each iteration augments the flow. In the general case the number of iterations can be $O(V \cdot E)$, but here it is easy to see that the flow can never be larger than $n - 1$, since only $n - 1$ people have a city to go to on any given day. This means that the overall time complexity is actually $O(E \cdot n)$, which reduces to $O(mnt)$, and that should be well within the time limit. The space complexity primarily depends on the size of the transformed graph, which, if we use lists of edges for storage, is $O(V + E) = O\big((m + 3n) \cdot t\big)$.

Problem I: Queen

Statement:

Compute how many squares on average does a queen attack on a generalized chess board $n \times n$.
The queen attacks a square if it is on the same row, column or diagonal. For example, the queen denoted by the letter Q in the image bellow attacks 17 squares marked with dots:



Input:

The first line contains positive integer n, the number of lines and columns of a board.

Output:

The output consists of one real number rounded to exactly three decimal places, the average number of fields attacked by a queen.

Constraints:

- $1 \leq n \leq 1{,}000{,}000$

Example input:

```
3
```

Example output:

```
6.222
```

> Time and memory limit: 0.5s / 64MB

Solution and analysis:

From each of $n \cdot n$ fields, a queen attacks $n-1$ fields in its row, and $n-1$ in its column.
Let's now count the fields on the same up-left to down-right diagonal. Counting diagonal by diagonal, we get:

$$D = 2 \cdot 1 + 3 \cdot 2 + \cdots + n(n-1) + (n-1)(n-2) + \cdots + 3 \cdot 1 + 2 \cdot 1$$

For up-right to down-left diagonals we obviously obtain the same result. This gives the total number of fields that are attacked from all positions of the queen:

$$S = 2D + n^2(2n-2)$$

After summation, we obtain:

$$S = \frac{10n^3 - 12n^2 + 2n}{3}$$

and the average number of attacked fields is:

$$M = \frac{S}{n^2} = \frac{10n - 12 + \frac{2}{n}}{3}$$

Summation can be done by using math, or a computer program. In the latter case, time complexity will be linear (instead of constant) and care should be taken of overflow / precision.

Implementation:

Trivial: just read n and write $\frac{10n-12+\frac{2}{n}}{3}$.

Complexity

As mentioned before, time complexity is $O(1)$ if computation is done mathematically, and $O(n)$ if computation is done programmatically.
Memory complexity is $O(1)$ in any case.

Test data:

The case $n = 1$ is an interesting example, so it should be included. Other tests should include odd and even numbers, as well as small and big numbers, to check different cases, time complexity and computation accuracy.

bubble cup 4

Problem A: Card

Statement:

Mike often needs to know if he could place a rectangular card of size $a \times b$ into an envelope of size $c \times d$. In order to be faster, Mike doesn't really try to put a card into an envelope, he just places a card on the table and then tries to cover it with an envelope. Of course, both the card and the envelope can be rotated, but they cannot be folded.
Now, Mike wants to be even faster. He decided to find the answers for all sizes of cards and envelopes he operates with. That's where you jump in. Your program should compute the answer for one particular case. The program should work the same way Mike does his tests, so in boundary cases the answer is "yes".

Input:

The first line contains four integers $a, b\ c$, and d delimited by a space.

Constraints:

- $a, b, c, d < 2 \cdot 10^9$

Output:

The output contains only one string: "yes" or "no" (without quotes).

Example input:	Example output:
2 3 3 4	yes

> Time and memory limit: 0.5s / 64MB

Solution and analysis:

All we need to do is to distinguish between several cases. To simplify the analysis, let's first sort pairs (a, b) *and* (c, d) *so that* $a \leq b, c \leq d$.

- *Case 1:* $\boldsymbol{a > c}$
- *In this case the answer is clearly no, since any* y*-projection of the card is bigger than* c.
- *Case 2:* $\boldsymbol{a \leq c,\ b \leq d}$
- *In this case card is easily covered with the envelope, for example by matching centers and aligning card and envelope axes, so the answer is yes;*
- *Case 3:* $\boldsymbol{a \leq c,\ b > d,\ a^2 + b^2 > c^2 + d^2}$
- *In this case the card diagonal* $D_C = \sqrt{\boldsymbol{a^2 + b^2}}$ *cannot be covered with the envelope, because the envelope diagonal* $D_E = \sqrt{\boldsymbol{c^2 + d^2}}$ *is shorter than* D_C. *Therefore, the answer is no.*
- *Case 4:* $\boldsymbol{a \leq c,\ b > d,\ a^2 + b^2 \leq c^2 + d^2}$
- *This is the remaining case. Now we have* $d < b < D_C \leq D_E$ *and we need to try to put the envelope's diagonal over the card. Consider the circle centered at envelope center and having radius* $\frac{D_C}{2}$. *It intersects all four sides of the envelope and we need to check if the distance between the nearest two intersection points is bigger or equal to* a. *If so, the answer is yes, otherwise no.*

Time complexity of this algorithm is constant – $O(1)$.

Test data:

Test corpus for this problem contains 10 test cases constructed with following methods:

- *several tests with different orders of side sizes*
- *tests with boundary conditions (for example card and envelope being of equal size)*
- *test in which the card tightly fits into the envelope diagonally*
- *test in which the card doesn't fit diagonally, but it would if it was just a bit smaller*

Problem B: Rook

Statement:

There is a generalized chess board of size (n, n). A rook should move from square $(1, 1)$ to square (n, n). In every move, exactly one coordinate must increase by 1 or more. There are also m occupied squares on the board, so the rook cannot be placed on any of them and cannot jump over them. Squares $(1, 1)$ and (n, n) are not occupied.

In how many ways can the rook reach the square (n, n)?

Input:

The first line contains two positive integers n and m delimited by a space,. In each of the next m lines there are two positive integers, x_i and y_i, coordinates of i^{th} occupied square, $i = 1, 2, \ldots m$.

Constraints:

- $n \leq 5{,}000$
- $m \leq 100{,}000$
- $1 \leq x_i, y_i \leq n$

Output:

The output contains number of different rook paths, as described above. If this number is greater than 1 million, you should only output its last 6 digits.

Example input:

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

Example output:

```
48
```

> Time and memory limit: 2s / 64MB

Solution and analysis:

Let's denote $w_{x,y}$ the number of ways in which the rook can reach the square (x, y). Then $w_{1,1} = 1$ and

$$w_{x,y} = \sum_{i=x_0}^{x-1} w_{i,y} + \sum_{j=y_0}^{y-1} w_{x,j}$$

where:

$$x_0 = \begin{cases} \max\{\, i : 1 \le i < x, and \;\; square\ (i,y)\ is\ occupied\} & if\ there\ is\ such\ square, \\ 1 & otherwise \end{cases}$$

$$y_0 = \begin{cases} \max\{\, j : 1 \le j < y, and \;\; square\ (x,j)\ is\ occupied\} & if\ there\ is\ such\ square, \\ 1 & otherwise \end{cases}$$

Using the formula for each square directly gives an algorithm that works in $O(n^3)$ time, which is too slow for limitations given in the problem statement.
Introducing two new matrices:

$$a_{x,y} = \sum_{i=x_0}^{x-1} w_{i,y} \qquad\qquad b_{x,y} = \sum_{j=y_0}^{y-1} w_{x,j}$$

for each square (x, y) we can compute $a_{x,y}, b_{x,y}, w_{x,y}$ in $O(1)$ time, which gives the following $O(n^2)$ algorithm:

```
================================================================================
w[1][1] = 1
for  i = 1..n
   for j = 1..n
      if (square (i, j) is occupied
           a[i][j] = 0;
           b[i][j] = 0;
           w[i][j] = 0;
      else
           a[i][j] = (a[i-1][j] + w[i-1][j]) mod 1000000;
           b[i][j] = (b[i][j-1] + w[i][j-1] ) mod 1000000;
           w[i][j] += (a[i][j] + b[i][j]) mod 1000000;

================================================================================
```

We are assuming here that all elements with at least one zero coordinate are initialized to 0.

Complexity:

For this solution, there are a couple of variations regarding time and space complexity:

a) We can put info about occupied squares into a matrix (for example w), and use a, b, w as matrices. In that case both time and space complexity is $O(n^2)$.

b) We could also put info about occupied squares into a separate array of length m and sort it in order in which the squares are being visited. Also, instead of matrices a, b, w, it is enough to use the last two rows of each of them. That gives us time complexity $O(m \log m + n^2)$, and space complexity $O(m + n)$.

Test data:

Test cases should include:

- *An example where it is not possible to move by the rules and reach the square (n, n);*
- *A big example with a large table and lots of occupied squares (up to the limit).*

Problem C: Tree game

Statement:

You are playing a simple game. You are given an undirected connected graph which does not have cycles. There is also one coin with is in the beginning located at vertex x. One step consists of moving the coin from the vertex at which it is currently located to any adjacent vertex (two vertexes are adjacent if there is an edge connecting them). Every edge has an associated number of points you gain if you move the coin from one of its vertexes to another. Your task is to calculate the maximal number of points you can gain in k steps. You can move the coin along some edges more than once.

Input:

The first line contains number n, which is the number of vertexes of the tree (number of vertexes n). The following $n-1$ lines contain information for $n-1$ edges of the tree. Each of the following $n-1$ lines has three numbers (i-th of these lines describes i-th edge) – the first two numbers are vertexes connected by the edge and the third number is the number of points that you gain if you move the coin along that edge. The number of points associated with an edge is less or equal to $1{,}000$. The vertexes are labeled with numbers from 1 to n.
The next line contains the number k.
The last line contains the vertex x, vertex at which the coin is located in the beginning.

Constraints:

- *2 ≤ n ≤ 100,000*
- *1 ≤ k ≤ 100,000*
- *The number of points associated with an edge is less or equal to 1,000*

Output:

You should output one number which is the maximal number of points you can gain in k steps with the coin located in the beginning at vertex x.

Example input:

```
6
1 2 3
4 3 5
4 1 2
3 6 6
5 1 9
3
4
```

Example output:

```
20
```

> Time and memory limit: 1s / 64MB

Solution and analysis:

This is a graph problem. On first sight, it looks like this problem requires the standard dynamic programming approach for trees - bottom-up from leaves to the root. But if we play a little bit with this problem, we will see that the greedy approach will find an optimal path.

Assume that we use following edges in k steps path: $p = e_1, e_2, \dots, e_k$. Edges can be used more than once so e_i and e_j can be the same edge for some $1 \leq i < j \leq k$. If there is an edge e_m $(1 \leq m < k)$ that has more points than every edge in the path p used after e_m, then the path p can't be optimal. Namely, in that case we can use the first part $e_1, e_2, \dots, e_m - 1$ of the path p and after that we just use edge e_m for the remaining $n - i + 1$ steps. This way we will get more points than in the original path.

Because of this, in the optimal path the edge with the maximal number of points among the edges that constitute the optimal path must be the edge which was used last (or in a last couple of steps) in the optimal path. This is the main idea for the algorithm.

Let's say that in the optimal path the edge e is used last in a couple of steps. We can see that the number of edges we used prior to using edge e should be as small as possible; otherwise the path would not be optimal because we can make a path which uses less edges prior to using edge e, and this path will then get us more points.

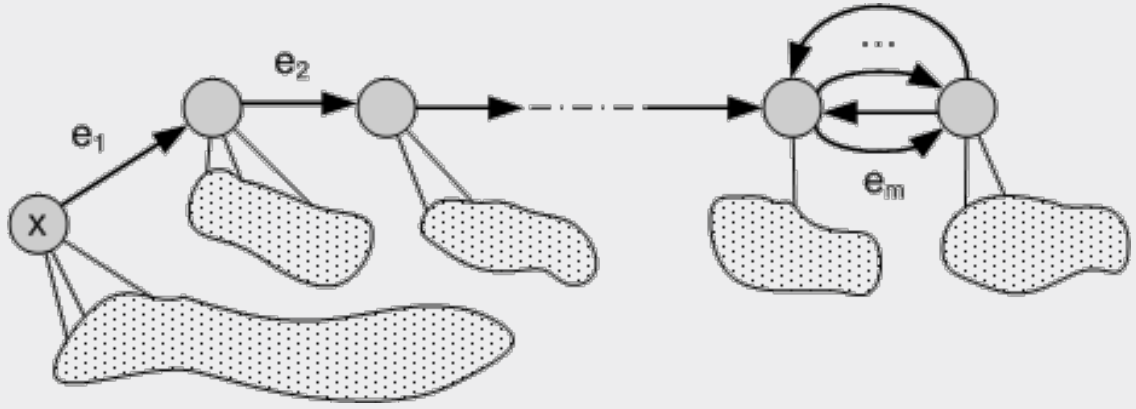


Figure 1. Optimal path

The solution consists of the following: for every edge of the tree we try to go to that edge using the minimal possible number of edges and then use that edge for every available step left and we choose among those paths the path with the maximal number of points. Of course, we try this for every edge to which the minimal number of edges used is less than the available number of steps. Because this is a tree, we can accomplish all this with one traversal using some standard graph traversal algorithms – DFS or BFS.

Time and memory complexity of this solution are both $O(n)$.

Problem D: Transformations

Statement:

You are given n different transformations of integers $1, 2, \ldots, n$, one for each of these n numbers. Using the first transformation you can transform number 1 to some group of numbers, using the second transformation you can transform number 2 to some other group of numbers etc. Numbers that can be derived using given transformations are also integers between 1 and n.

If you have a group of elements, which are numbers from 1 through n (there can be multiple instances of the same number in the group), in one step can you can transform any element of the group to new elements that are produced using the transformation of the selected element. You start with a group which has only one element, which is a number between 1 and n, and you can choose which number is the starting element of the group. Your goal is to have after s steps a group with as much elements as possible.

Input:

The first line contains one positive integer n.

The following n lines contain information for transformations of numbers from 1 to n. Each of the following n lines consists of the following integers (i-th of these lines describes transformation of number i) – the first number, denote it with e_i is the number of elements to which number i is transformed and the following e_i numbers are the numbers to which number i is transformed.

The last line contains number s which is number of available steps.

Output:

You should output one number which is the maximal number of elements your group can possibly have after s steps.

Constraints:

- $1 \leq n \leq 1{,}000$
- $1 \leq s \leq 50$
- $1 \leq e_i \leq 30$

Example input:

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

Example output:

```
10
```

Explanation:

There are 4 numbers. The transformations are:

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

The optimal solution is choosing the initial element of the group to be 2, then after transforming it the group will have elements 1 1 3 4 4, after that one instance of number 4 is transformed and the group will have elements 1 1 1 2 3 4. Finally, the number 2 is transformed and the group has 10 elements after 3 steps.

> Time and memory limit: 1s / 64MB

Solution and analysis:

We can solve the task using dynamic programming. This is a very nice problem, because we have some kind of two-step dynamic programming where these steps communicate with each other.

Firstly, let us introduce labels that we are going to use:

- $k \to (t[k][1], t[k][2], \dots, t[k][num[k]]) \equiv T[k]$, $k \in [1, n]$, *for the transformations. Number* k *can be transformed in the above group, where* $e[k]$ *represents cardinality of this list.*
- $F(numStep, (a_1, a_2, \dots a_m))$ *– maximal number of elements that can be obtained starting from the group* $(a_1, a_2, \dots a_m)$ *and performing* $numStep$ *transformations in some order.*
- s *- number of steps (transformations)*
-

The final solution can be computed as:

$$solution = max\{F(s, (1)), F(s, (2)), \dots, F(s, (n))\}$$

The main observation for this problem is following: when we perform transformation $a_k \to T[a_k]$ *on the group* $(a_1, a_2, \dots, a_m)$ *we obtain a new group*

$$newGroup = (a_1, a_2, .., a_{k-1}, a_{k+1}, \dots, a_m) + (t[k][1], t[k][2], \dots, t[k][e[k]num[k]]) = A + T[k]$$

Pay attention that the plus sign in the above formula is not a union. From here, we can look at these two groups independently. The only question is to find how many transformations to "give" to each group - partition of the number of steps. Without loss of generality, we can calculate the value $F(numStep, (a_1, a_2, \dots a_m))$ *by checking all possible number of steps for transformation* $a_m \to T[a_m]$.

Formally:

$$F(numStep, (a_1, a_2, \dots a_m)) = \max_{k \in [0, numStep]} F(numStep - k, (a_1, .., a_{m-1}) + F(k, a_m)\}$$

We can think of these transformations and groups as some kind of tree of deep s. Basically, we start from any group with one element – which is going to represent a root of this tree. We want to find a leaf which holds the set with maximal cardinality.

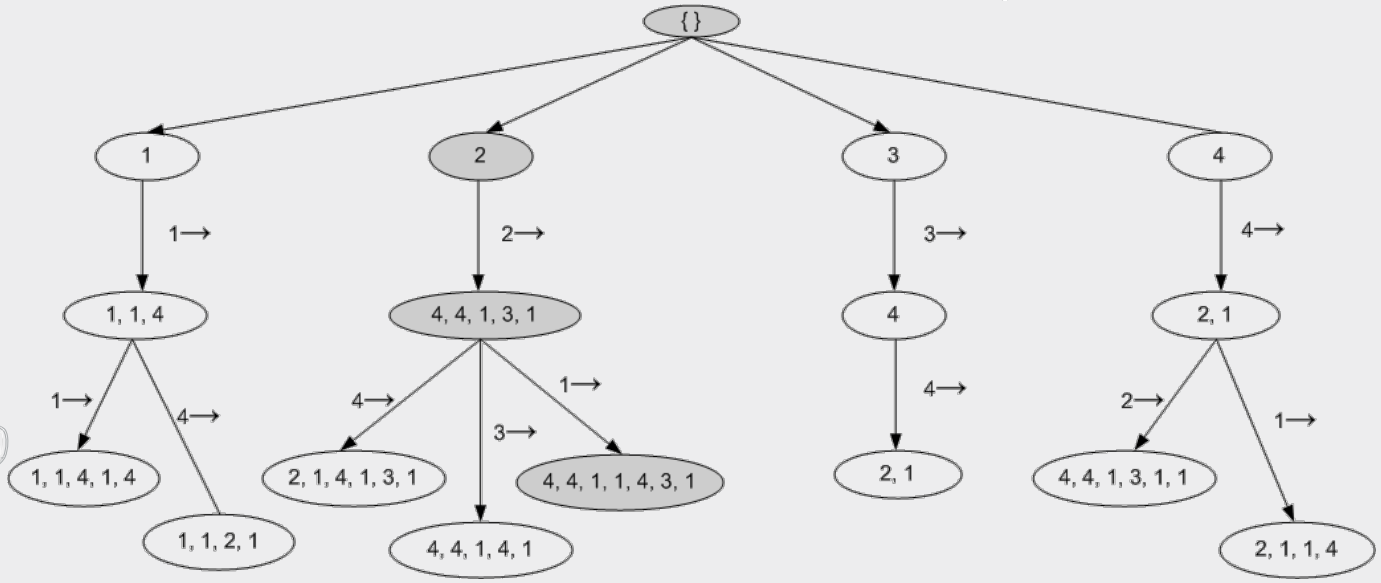


Figure 1. Example of the tree mentioned in the problem analyses from example in the problem statement with changed condition $s = 3$

Implementation:

This can be implemented in many ways. We will describe one of them. First let us define matrix d as:

- *$d[numStep, x]$ = maximal number of elements that can be obtained starting*
- *from only one element x in $numStep$ steps*

When we are computing some particular element $d[k, x]$, we are going to use:

- *$q[numStep, v]$ = maximal number of elements that can be obtained starting from the group $(t[x][1], \dots, t[x][v])$ in $numStep$ steps*

From here we have that $d[numStep, x] = q[numStep - 1, e[x]]$ (here we have -1 because we used one transformation $x \to T[x]$). We can play with elements of the matrix q with following relation:

$$q[numStep, 1] = e[x]$$
$$q[numStep, v] = \max_{k \in [0, numStep]} \{q[k, v-1] + d[numStep - k, t[x][v]]\}, \quad \texttt{for } v \in [2, e[x]]$$

The complexity of this solution is $O(n \cdot s^2 \cdot m)$, where m represents maximal group cardinality in the given transformations.

Problem E: LIS

Statement:

You are given an integer sequence a of length n and an integer w, $1 \leq w \leq n$. Let us denote with L_k the length of the longest increasing subsequence (LLIS) for subarray: $a_k, a_{k+1}, \ldots, a_{k+w-1}$. You have to write a program that computes values L_k for every k, $1 \leq k \leq n - w + 1$.
Assume that the sum of values L_k does not exceed $3 \cdot n \sqrt{n}$.
The longest increasing subsequence of a given sequence a is the subsequence of strictly increasing elements containing the largest number of elements. Elements of the subsequence do not need to be consecutive.

Input:

The first line contains two positive integers n and w and, where n is the number of elements in the given array and w is the width of subarray that have to be examined. Next line contains n integers, separated with one space, which represents the elements of array a. The elements are in range $[0, 2 \cdot 10^9]$.

Output:

The output contains $n - w + 1$ numbers, one per line. The number in the k-th line is the length of the longest increasing subsequence for $a_k, a_{k+1}, \ldots, a_{k+w-1}$.

Constraints:

- *1 ≤ n ≤ 100,000*
- *1 ≤ w ≤ n*
- *0 ≤ a[i] ≤ 2·10 ^{9}*

Example input:

```
6 4
1 4 2 5 6 7
```

Example output:

```
3
3
4
```

Explanation:

For this example, we have three subsequences of width 4 in given array a:

- *(1, 4, 2, 5), where LLIS is equal to 3; one possible LIS is (1, 2, 5)*
- *(4, 2, 5, 6), where LLIS is equal to 3; one possible LIS is (4, 5, 6)*
- *(2, 5, 6, 7), where LLIS is equal to 4; LIS is the whole subsequence*

> Time and memory limit: 2s / 64MB

Solution and analysis:

This problem considers finding the length of the longest increasing subsequence in a sliding window (of width w), over a given sequence a. In the problem statement it is noted that the sum of lengths does not exceed $n\sqrt{n}$. This is a very interesting fact and it might be confusing. Here we are going to present an output-sensitive data structure that solves this problem with time complexity $O(n \log n + OUPUT)$ or in our case $O(n \log n + n\sqrt{n})$.

Within this framework, several related questions can be posed regarding this problem, each with potentially different time complexity.

- *Local Max Value - For each window report the length of the longest increasing subsequence in that window;*
- *Local Max Sequence - Explicitly list a longest increasing subsequence for each window;*
- *Global Max Sequence - Find the window with the longest increasing subsequence among all windows;*

Here we deal with the Local Max Value. This algorithm solves the other two versions of the problem described above. Its optimality in our case is an open question and left for contestants to improve it ☺
A naïve approach is to consider finding LIS for every window separately. The standard dynamic programming algorithm for finding LIS has time complexity of $O(n^2)$, which will lead to complexity of $O(n \cdot w^2)$ for our problem. This approach can be sped up with algorithms which date back to Robinson [1] and Schensted [2] with a generalization due to Knuth [3]. These algorithms have time complexity $O(n \log n)$, which is optimal in the comparison model. Hunt and Szmanski [4] gave an algorithm with time complexity $O(n \log \log n)$ using the van Emde Boas data structure [5]. In any case, this naïve approach has time complexity $O(n \cdot w \log \log w)$ in the best case.
Without loss of generality we can assume that a given array a is a permutation of the set $\{1, 2, \dots, n\}$ (if not we can simply sort the array and rename the numbers in it with corresponding index). As we have seen in the previous paragraph, we have to find some way to use the LIS (or some other information) from the previous window when examining the current one. For this purpose, we will use Young tableaux or the Robinson–Schensted–Knuth algorithm. We will not explain these algorithms in detail, because only a part of them will be needed here.
Above we have stated that the length of LIS for a given array can be found in $O(n \log n)$ time. How can we do this? Let us introduce a new list d. Initially this list will be empty. We will insert elements from array a one at a time into the list d. When inserting number $value$ into d we have two cases:

1. *$value$ is greater than all elements from the list d - In this case we add $value$ to the end of list*
2. *$value$ is not greater than all elements from the list d - In this case there exists an element that is greater than $value$. Let us denote with t the first one from the left. Remove the element t from the list d and put $value$ in its place.*

With this algorithm list d will be monotonically increasing. It can be shown (how?) that the length of list d is the length of the longest increasing subsequence. It should be noted that list d is not a LIS for array a, because it may not be a subsequence (see example on Figure 1). The main idea behind this method is that the element $d[k]$ is the smallest element from array a for which there exists an increasing subsequence in a of length k ending with that element. We will call d the principal row of array a and denote it with $R(a)$.

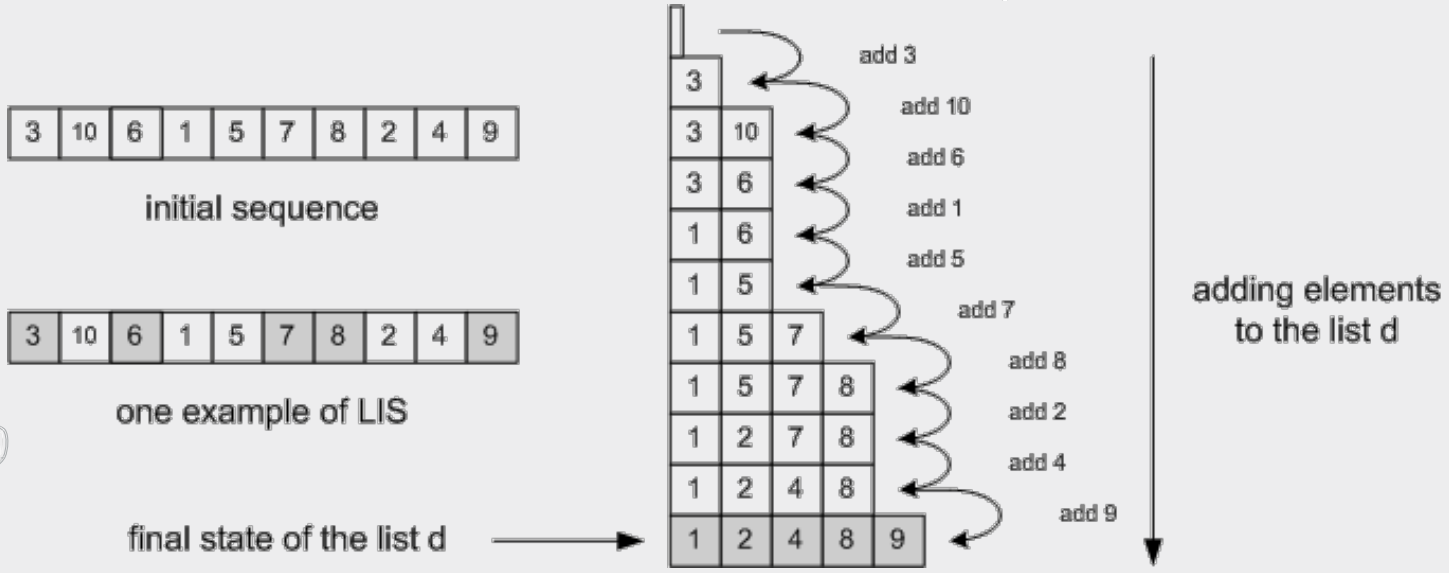


Figure 1. Example of algorithm for finding the LIS in array $\{3,10,6,1,5,7,8,2,4,9\}$.

In order to deal with the problem, we will consider a slightly more general question. We want to define some kind of structure that will maintain information about the LIS of a sequence in such a way that it supports the following operations:

- *adding a new element at the end of a sequence*
- *removing the first element from a sequence*
- *querying the data structure for the LLIS*

For this purpose, we are going to store the principal row for every suffix of the current sequence. If we denote with a^k *the suffix* $a_k a_{k+1} \dots a_n$*, our structure will maintain* $R(a^1), R(a^2), \dots, R(a^n)$ *(note that in our case this sequence has length w). This collection of principal rows is called a row tower.*

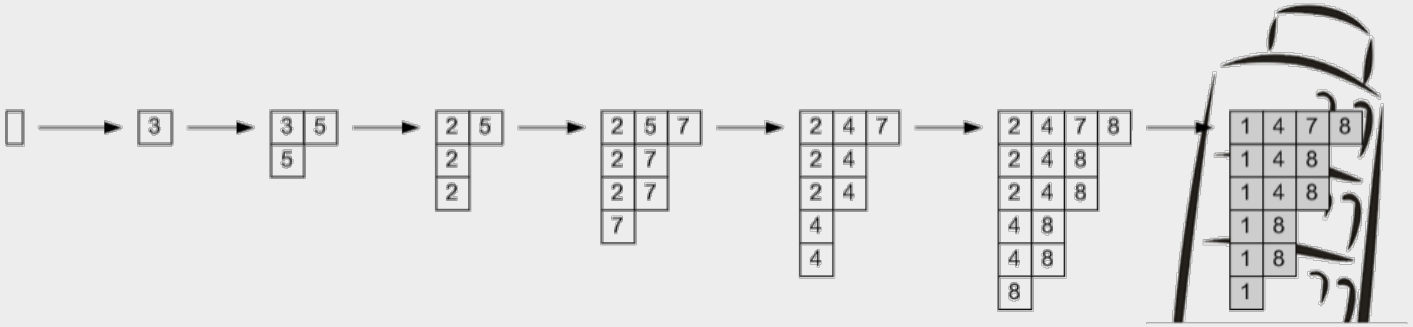


Figure 2. Example of a row tower for the array $(3,5,2,7,4,8,1)$ *and how it is generated.*

Removing the first element from a sequence can be implemented easily – delete the first principal row $R(a^1)$. The length of the first principal row is the length of LIS. Adding a new element corresponds to inserting it in every row and adding a new row containing only this element. A naïve implementation of this method will also lead to time complexity of $O(n \cdot w \log w)$. If we want to speed this up, we must store this tower in some compressed way.
Something that we can notice in Figure 2 is that $R(a^k)$ is either the same as $R(a^{k-1})$ or can be obtained from it by deleting a single element. This can be proven by induction (how?). From this we can state a generalization:
Lemma Let sequence A be a suffix of sequence B. Then $R(A)$ is a subsequence of $R(B)$ and

$$|R(B)| - |R(A)| \leq |B| - |A|$$

Because of this nice property, we can store the whole row tower in the following way:

- $R = R(a) = R(a^1)$ *– the principal row of whole sequence*
- *Drop out sequence d with the length $|R(A)|$. Element $d[index]$ represents the suffix at which the element $R(a)[index]$ drops out of the principal row.*

For our example of Figure 2 we have $d = (6,3,1,5)$. With these two sequences we can reconstruct the whole tower. The main problem here is to see how we can efficiently update this representation of the row tower. The expire operation simply subtracts one from each element of d and deletes the element with expiry time 0 (if there is one) from R. The add operation for an element b requires that b should bump an element out of each row of the row tower (unless it is appended to all of them). Since the rows form an inclusion chain, if b bumps a certain element s out of a row, then it bumps the element s out of all further rows to which s belongs. In other words, the drop out time for s changes to the index of the first row from which it is bumped out by b.
Now consider the next row of the tower (if one exists) after s has dropped out. In this row there may or may not be elements larger than s. If there are such elements, then b bumps out the smallest one. If not, then b is appended to the end of this and all subsequent rows. We can find a sequence of indices $i_1 < i_2 < \cdots < i_k$ for the sequence d such that:

- *i_1 is the least index of an element in the principal row R which is larger than b*
- *i_{x+1} is the least index larger than i_x for which $d(i_{x+1}) > d(i_x)$ (the element is larger than the prior one and it is still in the current principal row)?*

Now we can simply update the drop out sequence d according to:

- $d(i_1) = w$
- $d(i_{x+1}) = d(i_x)$, *for* $x \in [1, k-1]$

Implementation of this algorithm is pretty straightforward, and we will leave it to the reader.

Complexity:

In this way we managed to implement operations for adding and removing one element in linear time of the LLIS problem (querying is still in constant time). In the problem statement we denoted the length of LIS in k-th window with L_k. From this the overall time complexity of our algorithm is $O(\sum L_k)$. The described algorithm computes the lengths of LIS in the sliding window in total time of

$$O(n \log n + \sum L_k) = O(n \log n + OUTPUT) = O\left(n \log n + n\sqrt{n}\right)$$

Test data:

The test corpus for this problem consists of 15 test cases.
Test cases were generated with a couple of algorithms which (except those for special cases) were based on random sequences and following theorem [9]:
Theorem Let π_n be an uniform random permutation of set the $\{1,2,3,\dots,n\}$ and L_n an integer-valued random variable $L_n = LLIS(\pi_n)$. As $n \to \infty$ we have

$$E[L_n] \approx 2\sqrt{n} \quad \text{and} \quad \sigma[L_n] = o(\sqrt{n})$$

A short description of test cases is given in Table 1.

ID	n	w	min LLIS	max LLIS	solution sum	Description
01	10	5	2	3	16	By hand
02	100	10	3	7	395	Random
03	1000	100	12	21	15.333	Random
04	1000	900	54	57	5.675	Random
05	10000	100	70	91	802.603	Increasing sequence
06	99000	1000	2	825	39.315.222	"Saw" sequence
07	100000	50000	427	446	21.829.042	"Saw" sequence
08	100000	90000	587	597	5.908.135	Random
09	100000	100	12	25	1.671.330	Random
10	100000	1	1	1	100.000	Special case - Random
11	1	1	1	1	1	By hand
12	99999	99999	618	618	618	Special case - Random
13	99888	65432	1	1	34.457	Decreasing sequence
14	99999	1000	23	61	4.159.326	Random, $P_d = 95\%$
15	99999	77777	3024	3101	67.945.385	Random, $P_d = 95\%$

Table 1. Description of the test data

References:
[1] G. de B. Robinson, On representations of the symmetric group, Am. J. Math. 60 (1938) 745–760.
[2] C. Schensted, Longest increasing and decreasing subsequences, Can. J. Math. 13 (1961) 179–191.
[3] D. E. Knuth, Permutations, matrices, and generalized Young tableaux, Pacific J. Math. 34 (1970) 709–727.
[4] J. Hunt, T. Szymanski, A fast algorithm for computing longest common subsequences, Comm. ACM 20 (1977) 350–353.
[5] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1976/77) 99–127.
[6] M. H. Albert at al., Longest increasing subsequences in sliding windows, Theor. Comp. Sci. 321 (2004) 405 – 414.
[7] D. E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison–Wesley, Reading, Mass, 1973.
[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT Press (2009)

Problem F: Padlock

Statement:

You are stuck in a room with n doors. On every door there is a padlock with a 10-digit rolling lock combination. You can roll any digit either up or down, where rolling up at digit 9 will make the digit 0, and rolling down at digit 0 will make the digit 9. The padlock will be open when the combination is matched with the key for that padlock. The goal is to open all doors with the minimal number of rolling operations.
Initially all padlocks are set to 0000000000. Doors can be opened in any order. Besides rolling digits there is one very cool button on the padlocks. This button can turn the digits on padlock to the same combination as a different padlock that is already open (you cannot jump to a combination of the padlock for some door that is not open yet). This transformation does not count as a rolling operation.

Input:

The first line contains one positive integer n, where n is the number of doors. The next n lines contain 10-digit integers (some of them can have leading zeros), which represent the keys for padlocks.

Output:

The output should contain only one integer – minimal number of rolling necessary to open all doors.

Constraints:

- *1 ≤ n ≤ 1,000*

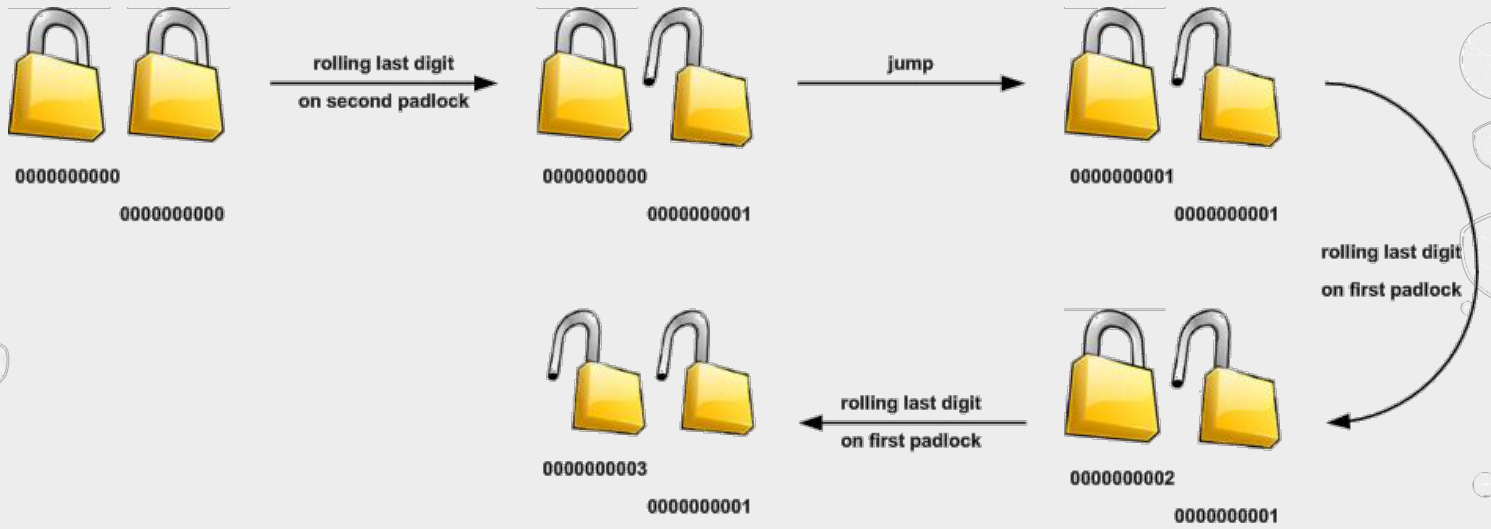
Example input:

```
2
0000000003
0000000001
```

Example output:

```
3
```

Explanation:



> Time and memory limit: 1s / 64MB

Solution and analysis:

We will first give the algorithm description, and then prove its correctness. We can use a simple greedy strategy:

1. *Find a lock that needs the least number of rollings, from the initial state, to open. Add that number to the overall cost and put that lock into the set of open locks.*
2. *Repeat until all locks are open:*
   a. *Among locks that are still closed, find the one that requires the least number of rollings to unlock, considering we can set it to state of any of the locks already open using zero rollings, or we can roll the numbers from the initial state.*
   b. *Update the overall cost and put the minimal lock into set of open locks.*

To show that this approach does indeed yield the minimal overall number of rollings, we can consider a graph whose vertices are locks, and weight of each edge $(u, v)$is equal to number of rollings needed to open lock v once it is set to the combination of lock u. We can extend this graph with a lock z, whose key is all zeros, so that weights of edges (z, i) represent numbers of rollings necessary to open lock i from its initial state. We also notice that weights of edges (u, v) and (v, u) must be equal, thus we have a complete undirected graph.

When opening lock v, we can either set it to a key of a previously open lock u and then roll the numbers to get the right key or roll the numbers from the initial position to v's key. So, unlocking v increases the overall cost either by weight of edge (z, v)) or by weight of edge (u, v). If we consider the subgraph with only these used edges, we see that it is actually a spanning tree of the original graph. So, in order to find the least number of rollings necessary to open all locks, we need to find a minimum spanning tree of our graph.

The proposed greedy approach is actually Prim's algorithm for finding minimum spanning trees of graphs and is easily implemented to run in $O(n^2)$ time. We can also precalculate numbers of rollings between all pairs of locks, and store the graph in matrix form, which requires additional $O(n^2)$ time and memory.

Problem G: LR primes

Statement:

A number $a = \overline{c_n c_{n-1} \dots c_2 c_1}$ is called L prime if its every non-empty suffix is a prime number and all its digits are different from zero. In other words, numbers $\overline{c_1}, \overline{c_2 c_1}, \dots, \overline{c_{n-1} \dots c_2 c_1}$ and $\overline{c_n c_{n-1} \dots c_2 c_1}$ must be primes. For example, the number 113 is L prime number.
A number $a = \overline{c_n c_{n-1} \dots c_2 c_1}$ is called R prime if its every non-empty prefix is a prime number. In other words, numbers $\overline{c_n}, \overline{c_n c_{n-1}}, \dots, \overline{c_n c_{n-1} \dots c_2}$ and $\overline{c_n c_{n-1} \dots c_2 c_1}$ must be primes. For example, number 311 is R prime.
You are given an integer segment $[a, b]$. How many integers from this segment are L or R prime numbers (including numbers a and b)?

Input:

The first line contains two positive integers a and b, which represent the given segment.

Output:

The output contains only one integer – the number of integers from given segment that are L or R primes.

Constraints:

- $1 \leq a \leq b \leq 10^{18}$

Example input:

```
10 30
```

Example output:

```
4
```

Explanation:

From the segment $[10, 30]$ L primes are: 13, 17, 23; R primes are 23, 29. Number 23 is both L and R prime, so we are going to count it only once.

> Time and memory limit: 0.5s / 64MB

Problem analysis:

L and R primes are also known as left-truncated and right-truncated primes. Codes of their sequences in the On-Line Encyclopedia of Integer Sequences are A024785 and A024770. We found them interesting for a programming problem because of two facts:

- *they are finite*
- *they have some kind of recursive property*

We need to find a method for generating consecutive right and left primes. Here we are going to explain the algorithm for right primes. The same algorithm, with small modifications because of the special digit 0, can be used for the left primes.

As we mentioned, these numbers have some kind of recursive structure: every right prime number having at least two digits is an extension of another right prime number (i.e. the least significant digit is added). This is the main fact on which we are going to base our iterative algorithm.

Let Q_R be an empty queue, which will store the right primes. We start by inserting the one-digit right primes (just primes). Then in every step we extract the first element s from the queue and check if any of the numbers $10 \cdot s + k$, $k \in [1,3,7,9]$ is prime. We excluded the digits $\{0,2,4,5,6,8\}$, because if the last digit is from this set, the new number will not be prime. If this number is also a right prime - put it at the end of the queue.

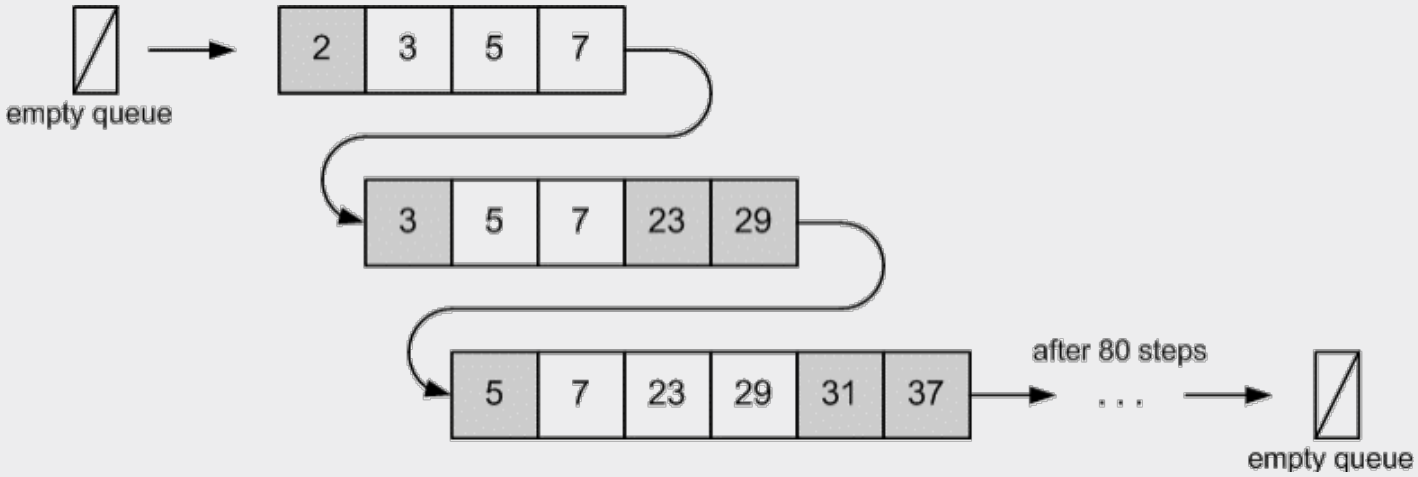


Figure 1. The queue states in the right prime construction process.

Complexity and implementation:

An interesting feature that we need to address for this algorithm is its time complexity. The complexity is $O(k\sqrt{m})$, where k is the number of the right prime numbers and m is the greatest among them (the same thing holds for the left primes). This fact is left for contestants to find out. Namely, these are finite sequences and after running this algorithm it appears that the algorithm terminates with an empty queue. There are only 83 right prime numbers and only 4260 left prime numbers. The largest of them are 73.939.133 and 357.686.312.646.216.567.629.137, respectively.
Another interesting fact is that if zeros are permitted, the sequence of left primes is infinite.
Because of this fact, the described algorithm with a reasonable implementation works very fast. For this we must use some other technique for the primality testing. In our case the Fermat test will do the work. Of course, some other algorithm, like the Miller-Rabin test, would also work. Here we will briefly describe the Fermat test.
Firstly, recall Fermat's little theorem: if p is a prime number and a is an integer relatively prime to p, then

$$a^{p-1} \equiv_p 1$$

Experimentation shows that this typically fails when p is composite. This is the fact which is going to be the core of our test. Complexity of this algorithm is $O(k \cdot log^2 n)$, where k is the number of times we test a random number a with above theorem.

```
==================================================================================================
Function: Fermat's primality test
Input:  n - a value to test for primality
Output: false if n is composite
         true if n is probably prime
--------------------------------------------------------------------------------------------------
repeat k times
   pick random integer a from set {2,3,…,n-1}
   d = gcd(a,n);
   if (d != 1)
     return false;
   tmp = a^(n-1) mod n;
   if (tmp != 1)
      return false;
return true;
==================================================================================================
```

Pseudo code for the second algorithm

Another option is to hardcode all left and right primes in the code. Such solution works in linear time. Here we have to pay attention to the size of the file. If we hardcode this in a relatively smart way, we will get the source file of the size ~ 60KB, and 64KB is the maximum allowed size for source files on the finals.

Problem H: Hashed strings

Statement:

You are an evil hacker and your current evil mission is to impersonate your target by sending messages that look like they came from them but that are actually from you. You have worked out the entire operation except for one small detail: every string that your target sends is followed by a 32-bit hash value, which is used for error checking. You know the algorithm, and it goes like this:

The strings are composed of lowercase letters of the English alphabet, and every letter corresponds to a unique 16-bit code. All 32 bits of the hash value are initialized to zero. The hash is then calculated by passing through every character of the string in order and performing the following steps:

Do a binary left rotation of the entire hash value (by one place)

Take the code for the character and the least significant 16 bits of the hash and do a binary XOR of these two values

Write the result from the previous step to the least significant 16 bits of the hash

Unfortunately, to implement the algorithm you need to know the 16-bit codes for letters of the alphabet, and those codes are secret. Not all is lost though! You have already intercepted many pairs of strings with their hash values. Now all you have to do is find some way to use that information to crack the codes.

Input:

The first line contains one positive integer n, the number of strings. Each of the following n lines contains one string consisting exclusively of letters 'a'-'z' and one integer in the range $[0, 2^{32} - 1]$. Writing this integer in 32-bit binary gives the hash value of the string.

Output:

The first line of output should be one of three words: "IMPOSSIBLE", "UNIQUE" or "MULTIPLE" (without quotes), if there are respectively no solutions, exactly one solution and more than one solution. If the first line is "IMPOSSIBLE" or "MULTIPLE", nothing else should be written to output. If the first line is "UNIQUE", each following line of output should contain exactly one letter and one number, separated by a space. Every number is in the range $[0, 2^{16} - 1]$, and when written in 16-bit binary represents the code for the letter. There should be as many lines as there are different letters that appear in input. The lines should be sorted alphabetically by letter.

Constraints:

- *1 ≤ n ≤ 400*
- *Each string is at most 100 characters long*

Example input:

```
2
a 4
ab 12
```

Example output:

```
UNIQUE
a 4
b 4
```

> Time and memory limit: 3s / 64MB

Solution and analysis:

This task is based on a problem that the author actually had to solve for his real-life job (it didn't involve any hackers though – that part is made up ☺), and we thought it was interesting enough to be used for competitive purposes.
We have n strings. Let's denote them with $s_i = c_{i,l_i-1}c_{i,k_i-2} \dots c_{i,0}$ ($i \in \{0,1 \dots n-1\}$;l_i is the length of s_i). (We'll use indexes that increase right to left throughout the text, so don't say you weren't warned.) A 32-bit integer corresponds to each string: $h_i = b_{i,31}b_{i,30} \dots b_{i,0}$ $(b_{i,j} \in \{0,1\})$. And finally, each character c corresponds to a code, which is the 16-bit integer $x_c = x_{c,15}x_{c,14} \dots x_{c,0}$.
Let's observe the rightmost (index 0) bit of the hash value h_i. How is it calculated? Obviously, the rightmost bit of the rightmost character of string s_i (which we have denoted with $x_{c_{i,0},0}$) can change it in the last step. But before that, the bit $x_{c_{i,17},15}$ was initially added on index 15 and then made half a circle to get to our bit $b_{i,0}$. And before that, bits $x_{c_{i,18},14}$, $x_{c_{i,19},13}$,… $x_{c_{i,32},0}$ also ended up turning around and contributing to $b_{i,0}$. And another half-circle before that, bit $x_{c_{i,49},15}$, and so on. Obviously, this goes on until we run out of characters in s_i. The formula is

$$b_{i,0} = \sum_{0 \le j < l_i, 0 \le k < 16}^{32|(j+k)} x_{c_{i,j},k}$$

where the sum is an XOR sum (or, mathematically speaking, everything happens in $\mathbb{Z}_2$). Now let's try to generalize this observation. We have the q-th bit (from the right, zero-based – as above) of hash value h_i. Which bits of the original codes are important for this bit? The same line of thinking as in the previous paragraph leaves us with the formula

$$b_{i,q} = \sum_{0 \le j < l_i, 0 \le k < 16}^{j+k \equiv q \pmod{32}} x_{c_{i,j},k}$$

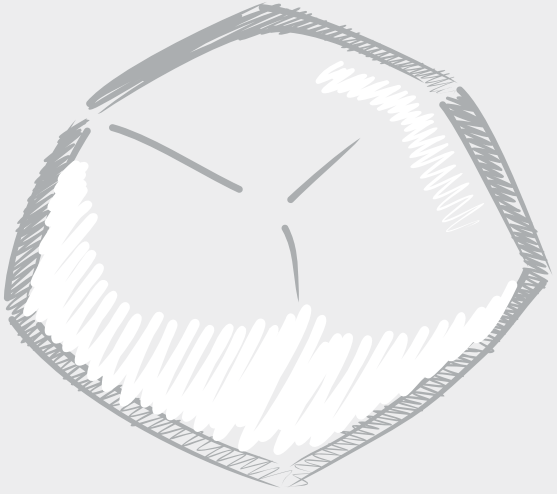
This means that the problem reduces to a system of linear equations. We have one equation per every bit of every hash sum, which is a total of $32n$. The number of variables is 16 times the number of letters that appear in the input.
Solving systems of linear equations is a well-known problem, and here it is made even easier by the fact that we are working in $\mathbb{Z}_2$ so the only values are 0 and 1 and there are no problems with precision. For example, we can solve the system in time $O(u^2 \cdot v)$, where u is the number of variables and v the number of equations, by the standard Gaussian method of eliminating the variables one by one. Of course, this algorithm is able to determine whether the system has a solution, whether it is unique and, if it is, to find it. After this, assembling the solution bit-by-bit into codes for every letter and sorting them alphabetically should present no trouble at all.

Complexity:

It is easy to see that the most expensive part of our algorithm is solving the system of equations, so the time complexity will be $O(u^2 \cdot v)$ per above. We have $u = 32n$ and, since we have a finite alphabet of known size, we could say that v is a constant but that would be slightly disingenuous as this constant is quite large. If w is the number of letters that appear in the input, we have $v = 16w$, and finally our complexity is $O(2^{13} \cdot w^2 \cdot n)$. It is interesting to note that time complexity is independent from the length of the strings.
We need $O(u \cdot v) = O(2^9 \cdot w \cdot n)$ space to store the equations, which gives us the memory complexity of this solution.

Test data:

ID	Description
01	Easy test (example from the problem statement)
02	1 string with 1 letter with valid hash value (result code for the letter is equal to the hash value)
03	1 string - 31 times one letter. Code invalid/valid (IMPOSSIBLE/UNIQUE)
04	
05	1 string - 32 times one letter (each bit from the code influences each bit of the hash value, so all bits of the hash value need to be equal) (IMPOSSIBLE/MULTIPLE)
06	
07	1 string - 33 times one letter. Code valid (UNIQUE)\
08	1 string - 64 times one letter (hash value doesn't depend on the code of the letter/hash value is always 0) (IMPOSSIBLE/MULTIPLE)
09	
10	Invalid hash value (larger than it could be calculated with given string) (IMPOSSIBLE)
11	
12	Contradiction (last bit of the code for a letter should be both 0 and 1)
13	Large test with a small number of letters
14	Less strings than the number of used letters but still UNIQUE solution
15	Large strings but not enough equations to calculate UNIQUE solution (MULTIPLE)
16	Large test. One bit changed so IMPOSSIBLE.
17	Large tests to calculate UNIQUE solution
18	
19	



Problem A: Good sets

Statement:

Let A be the set $\{1, 2, \dots n\}$, where n is a given natural number. Set B is called good if it has the following properties:
B is a subset of A;
For every x, if x belongs to B, then $2x$ doesn't belong to B;
No other set C can have properties a) and b) and a greater number of elements than B;
For example, if $n = 12$, then $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, and the set $\{1, 3, 4, 5, 7, 9, 11, 12\}$ is good, while $\{1, 4, 5, 6, 7, 9, 11\}$ is not good (note that set C from the third property doesn't have to be a superset of B).
Given positive integer numbers n and b compute the following:
The number of elements in every good set;
With how many zeros the total number of good sets ends, if written in base b.

Input:

The first and only line of input contains two integers n and b, separated with one empty space, representing cardinality of the set A and the given base b, respectively.

Output:

Output contains only one line with two integers, separated with one empty space: the number of elements in every good set and number of zeros at the end of the total number of good sets in base b, respectively.

Constraints:

- $1 \leq n \leq 4 \cdot 10^9$
- $2 \leq b \leq 100$
- *Number b is a prime number.*

Example input:

```
12 3
```

Example output:

```
8 1
```

Example explanation:

All good sets consist of 8 elements and there are 6 of them - 6(10) = 20(3).

> Time and memory limit: 0.5s / 64MB

Solution and analysis:

Divide set A into chains such that each chain starts with an odd number from A and contains repeatedly doubled values from A. For example, if $n = 12$, set A is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and the chains are $1 - 2 - 4 - 8$, $3 - 6 - 12$, $5 - 10$, 7, 9, 11. Here the first chain contains 4 elements, the second chain 3, the third 2, and there are three one-element chains. Now elements are chosen from each chain independently.

Let us denote with $d[n]$ the maximal number of elements from a chain of length n such that no two are consecutive. It is not hard to see that if a chain has an odd number of elements, there is only one way to pick the maximal number of elements from that chain (take every element with an odd index). So, for a chain with $2k - 1$ elements, that maximal number is k. In other words, we have that $d[2k - 1] = k$.

For chains with an even number of elements, say $2k$, maximal number of elements that can be taken is k. Can we apply some sort of induction here? For $2k + 2 = 2(k + 1)$ elements we see that we must use exactly one element out of the last two (because otherwise we would have to select $k + 1$ elements from $2k$, which is not possible by induction). If we choose the last one, then from the first $2k$ we must select k of them and this can be done in $d[2k]$ ways. In the second case, if we do not select the last one, then we would have to select $k + 1$ elements from first $2k + 1$. From prior discussion we have that there is only one way to do this. Now we have:

$$d[2k + 2] = d[2k + 1] + d[2k] = 1 + d[2k]$$

Finally, from induction we have that $d[2k] = k + 1$, because $d[2] = 2$.

It remains to count chains of each different length, add up maximal numbers of elements for each chain, and multiply ways to choose such elements from each chain. Actually, instead of computing the exact number of ways to form a good set, it is required only to compute how many times this number is divisible by a given prime number b.

Problem B: Wheel of Fortune

Statement:

You are on a quiz show playing the game Wheel of Fortune. The wheel has N fields of the same size, and each field $1, 2, \ldots, N$ is associated with a value: $D[1], D[2], \ldots, D[N]$. Each time you spin the wheel you have equal probability of hitting any of the N fields. You will spin the wheel K times. When you spin the wheel for the i-th time and it stops on field j, if it is your first time hitting that field, $D[j]$ dollars is added to your prize and field j gets marked. If the wheel stops at a marked field, meaning you've hit that field in some of your previous spins ($i-1, i-2, \ldots, 1$), your score does not increase.
What is the expected value of the prize you'll take home?

Input:

The first line contains two integers, N – number of fields on the wheel, and K – number of times you get to spin the wheel. The following N lines contain one integer each, representing values of the fields – $D[1], D[2], \ldots, D[N]$.

Output:

Output contains exactly one real number – expected value of your overall prize, rounded to 5 decimal places.

Constraints:

- $1 \leq N \leq 10^4$
- $1 \leq K \leq N$
- $1 \leq D[i] \leq 10^4$

Example input:

```
2 2
10
20
```

Example output:

```
22.50
```

> Time and memory limit: 0.5s / 64MB

Solution and analysis:

In order to calculate the expected prize value, we can observe the expected value that we can gain from each field. By the rules of the game, for each field j we can get either 0 points if we never hit that field in our K spins, or exactly $D[j]$ points if we hit j at least once (i.e. we get the same score for field j no matter how many times the wheel stops at that field). Thus, the expected overall prize can be calculated as the expected sum of prizes each field will give us, so we have:

$$E[Prize] = E\left[\sum_{j=1}^{N} Prize_j\right] = \sum_{j=1}^{N} E\left[Prize_j\right]$$

Given that we have N *fields of the same size, and we are making* K *random draws (i.e. wheel spins), it is obvious that for each field of the wheel we have the same binomial distribution over the number of hits after* K *spins:*

$$P_k(K,p) = \binom{K}{k} p^k (1-p)^{K-k}$$

Here P_k stands for the probability of exactly k hits after K spins, and p denotes probability of hitting the field in a single spin, so $p = \frac{1}{N}$.
Now we can write the distribution over prize value for each field:

$$Prize_j = \begin{pmatrix} 0 & D[j] \\ P_0(K,p) & \sum_{i=1}^{K} P_i(K,p) \end{pmatrix}$$

In order to avoid dealing with binomial coefficients, we can rewrite the above distribution in simpler terms:

$$Prize_j = \begin{pmatrix} 0 & D[j] \\ P_0(K,p) & 1 - P_0(K,p) \end{pmatrix}$$

so, we end up with :

$$Prize_j = \begin{pmatrix} 0 & D[j] \\ \left(1-\frac{1}{N}\right)^K & 1 - \left(1-\frac{1}{N}\right)^K \end{pmatrix}$$

Finally, we can calculate the expected overall prize as:

$$E[Prize] = \sum_{j=1}^{N} E\left[Prize_j\right] = \sum_{j=1}^{N} D[j] \cdot \left(1 - \left(1-\frac{1}{N}\right)^K\right)$$

which yields an easy $O(N)$ solution.

Problem C: MaxDiff

Statement:

You are given an array of integers A of length N. We will define $S(A)$ as the sum of absolute differences between all pairs of consecutive elements in A. More formally, assuming that A is zero-based:

$$S(A) = \sum_{i=1}^{N-1} |A[i] - A[i-1]|.$$

Your task is to find the permutation $p(A)$ of the array A for which the value $S(p(A))$ is maximized.

Input:

The first line of input will contain one integer N, representing the size of the array A. The second line will contain N space-separated integers, representing the elements of the array.

Output:

The first and only line of output should contain a single integer equal to the largest sum of differences of consecutive elements obtainable from A as described in the problem statement.

Constraints:

- $1 \le N \le 1{,}000{,}000$
- $-2^{32} \le A[i] \le 2^{32}-1$

Example input:

```
3
2 3 5
```

Example output:

```
5
```

Example explanation:

There are six possible ways to reorder the array: $(2,3,5)$; $(2,5,3)$; $(3,2,5)$; $(3,5,2)$; $(5,2,3)$; $(5,3,2)$. The sums of differences are then respectively $3, 5, 4, 5, 4$ and 3, and the largest among them is 5.

> Time and memory limit: 1.0s / 64MB

Solution and analysis:

It is obviously infeasible to generate all permutations of A, calculate the value S for each one and pick the maximum, so let's try to observe some things about the problem that will help us reduce the space of possible solutions.
We will assume that all elements in the array are distinct. The proofs for the case when equal elements are allowed are slightly more difficult and there is a number of corner cases that have to be taken care of, so they will be left to the reader as an exercise ☺
First, let's notice a relatively obvious but very important fact: there will always exist an optimal solution in which the elements are sorted in a "zig-zag" manner, i.e. it will not contain a triple of consecutive elements such that $A[i] < A[i+1] < A[i+2]$ *(or* $A[i] > A[i+1] > A[i+2]$*). Proving this is easy: if we have a triple satisfying this condition, we can just pull out its middle element and place it at the end of the array – it is trivial to verify that S cannot decrease after this transformation.*
The other fact is slightly harder to notice. Let's denote the median of A with m*. (A reminder: the median of an array is the middle element of the sorted array if the number of elements is odd, and the average of the two middle elements if the number of elements is even). Clearly* m *depends only on the elements of A and not on the permutation. We will prove the following:*
Lemma. There is an optimal solution in which there are no two consecutive elements that are either both larger or both smaller than the median.
Proof. The first thing to notice here is that, due to the "zig-zag" principle discussed above, an optimal solution can't contain a sequence of exactly two consecutive elements on the same side of the median. Let's assume that there are at least three such consecutive elements. It is easy to see that we can always pick exactly three consecutive elements from this sequence such that $A[i] > A[i+1] < A[i+2]$*. Since the rest of the array now has at least two more elements that are under the median than elements that are over it, we can use the same reasoning to conclude that somewhere else in the array there are three consecutive elements under the median, ordered as* $A[j] < A[j+1] > A[j+2]$*. Since* $A[i+1] > A[j+1]$ *(the former is over the median and the latter under it), we can swap these two elements and get a solution that preserves all the inequalities and is strictly better than the previous one.*
Now we have enough information to deduce the most important statement:
If the order of elements in A satisfies the two principles described above, its value of S is

$$S(A) = |A[0] - m| + |A[N-1] - m| + \sum_{i=1}^{N-1} 2 \cdot |A[i] - m|,$$

where m *is the median of A.*

It should be clear that this holds from the following argument: since for all i elements $A[i]$ and $A[i+1]$ aren't on the same side of the median, their absolute difference is $|A[i] - A[i+1]| = |A[i] - m| + |A[i+1] - m|$. For each element except the first and the last one the term $|A[i] - m|$ appears twice in the final sum, while for the two edge elements it appears just once. This gives us the final step in the solution: since all terms in the sum are non-negative, we just have to minimize the value $|A[0] - m| + |A[N-1] - m|$. If the total number of elements is even, we pick the two middle elements for the ends – otherwise the zig-zag property would not hold. If it is odd, we pick the median element at one end and the element closest to it by absolute value at the other.

Note that we don't even have to generate the exact permutation, since all permutations $p(A)$ constructed in this way will have the same value $S(p(A))$ and the above discussion gives us the guarantee that permutations which don't satisfy these conditions cannot possibly result in a better solution.

The implementation ends up being very simple: first we find the median of the array A, then we find the edge elements as described in the previous paragraph, and finally we sum up the absolute differences of the elements from the median, multiplying by two for all except the leftmost and the rightmost element.

There are still some traps that need to be avoided – edge cases with a very small number of elements need to be dealt with, the solution has to be kept in a 64-bit value, repeating values can pose a problem for certain implementations. However, none of that should present a serious challenge for any competitor with decent technique.

Complexity

There is a choice for the algorithm used to calculate the median. The simplest way is to sort all the elements and pick the one(s) in the middle, which takes $O(N \log N)$ time. We can do better – the well-known quickSelect algorithm gives expected $O(N)$ time. Although its running time depends on the pivot choice and its worst-case complexity is $O(N^2)$, median-of-three or just random pivot choice should be enough since none of the test cases targeted this scenario (at least not intentionally). For the more paranoid contestants, the pivot can be chosen using the median-of-medians algorithm, which guarantees $O(N)$ running time but is tricky to implement and slower on average than simple quickSelect.

The rest of the algorithm can be done in a single pass of the array, giving an overall $O(N)$ time complexity of the algorithm. The memory complexity is obviously $O(N)$.

Problem D: Cars

Statement:

There are n cars parked at the parking lot and a new car is arriving. The parking lot is a space between two walls and cars are parked along one line between those walls. The driver will park his car if there is a free parking spot that is long enough (at least as long as the car). Otherwise, he will have to move a few cars in order to make appropriate space for his car. The car can be moved to the left or to the right along the parking lot, but at most until it reaches a wall or another car.
Your task is to find the minimal total distance by which currently parked cars have to be moved in order to provide enough space for the arriving car.

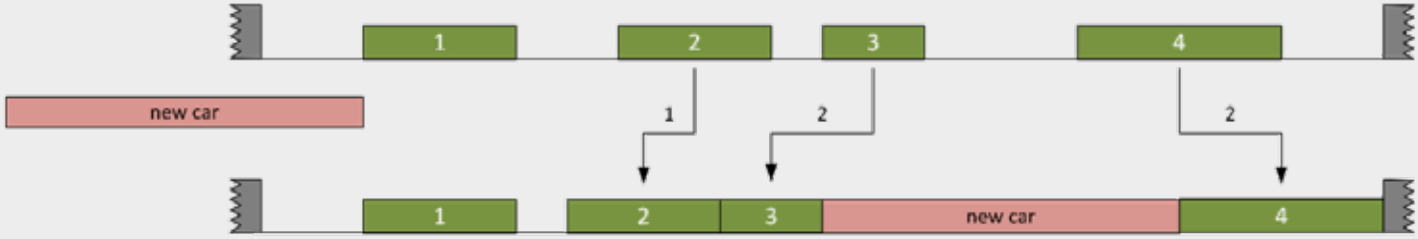


Figure 1. Optimal car moves for the given example.

Input:

The first line of input contains three space-separated integers n, lw and rw. They denote the number of cars already parked, the coordinate of the left wall and the coordinate of the right wall, respectively. Each of the following n lines contains two integers, describing a parked car: p – the coordinate of the leftmost point of the car, and l – the length of the car. The last line of the input contains one integer $lnew$, the length of the arriving car.

Output:

Output contains only one line with one integer – the sum of distances by which parked cars have to be moved to provide enough space for the arriving car. If a solution doesn't exist, the output should be -1.

Constraints:

- $0 \le n \le 10^5$
- $0 \le lw, rw, p[i] \le 2 \cdot 10^9$
- $1 \le l[i], lnew \le 10^6$

Example input:

```
4 0 22
2 3
7 3
11 2
16 4
7
```

Example output:

```
5
```

Example explanation:

The best way is to push the second car to the left by 1, the third car to the left by 2 and the fourth car to the right by 2. It will create an empty space of length 7, so the new car can be parked there. The sum of all movement lengths is 5 (= 1 + 2 + 2).

> Time and memory limit: 0.5s / 64MB

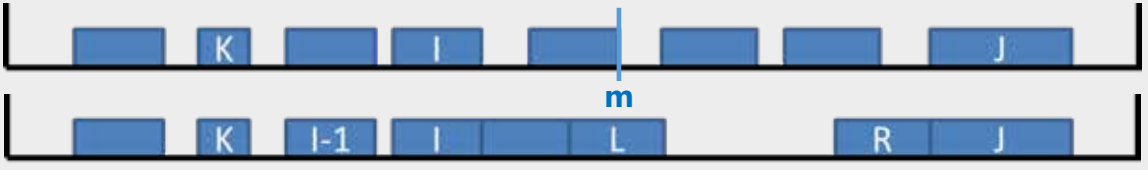
Solution and analysis:

Let's enumerate the cars with 1 to n from left to right. (To be able to do that, we will need to sort the array of cars first). Consider each car in turn. For each car K, find the first car J ($J > K$), such that the sum of free parking spots between K and J is greater than the length of the new car. For each such pair k, j we will find the optimal solution, and then use these to compute the global minimum.
Let's consider a given pair of cars (K, J). In the case that the pair $(K+1, J)$ also satisfies the total empty length constraint, we can narrow down the search space by removing car K from the set of cars for which we will consider moves. We will repeat this process as long as removing the leftmost car will still satisfy the total empty space length constraint. Let's denote the leftmost car remaining in this set with I.

Let's enumerate empty spaces between cars I and J with numbers $0..(J-I)$. We will find the first empty space m such that $\frac{\sum_{y=0}^{m} Length(y) \geq \sum_{y=0}^{J-I} Length(y)}{2}$. We will move every car in the direction of whichever car (I or J) is closer to it, in order to reduce the total distance covered. In this way, the upper bound on the distance any given car can move is $\frac{\sum_{y=0}^{j-i} Length(y)}{2}$, whereas if we moved any car in the other direction (to the car which is farther away from it) this upper bound would be greater.
Let's define four values for every car:

- $FS_L(I) = \sum Free\ space\ to\ the\ left\ of\ I$ – *Cost for moving car I to the leftmost position possible (if all cars before it were parked consecutively from the left wall, with no space between cars).*
- $CM_L(I) = \sum_{X=0}^{I} FS_L(I)$ – *total cost for moving all cars from 0 to I (inclusive) to the left wall.*
- *Another pair of arrays FS_R, CM_R , representing free space and total cost of moving to the right wall.*

These values can be pre-computed in $O(n)$ time with two passes through the array.
We would like to compute the cost of moving all cars between I and J away from the middle (m), without moving any other cars.



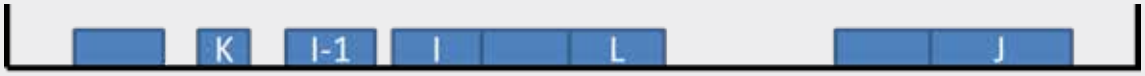
Consider only cars I, L (The rightmost car moved to the left) and $I-1$. We will separately calculate the cost of moving cars to the left and to the right. The cost can be calculated in the following way:
The cost of moving car L and all cars left of it to the left wall is $CM_L(L)$.

If we moved all cars left of car $I-1$ (inclusive) to the left wall and all cars from I to L to the car I, the cost difference between this configuration and the previous is $FS_L(I) * (L-I)$. (It is the cost of moving $L-I$ cars to the right by the sum of empty spaces left of I).

Now we can determine the cost of moving only cars from I to L to the left towards car I. Since the cost difference between this configuration and the previous is $CM_L(I-1)$,
This cost can be calculated as:

$$\text{Cost}(I,L) = CM_L(L) - CM_L(I-1) + FS_L(i) * (L-I)$$

In the same way (using pre-computed FS_R, CM_R), we can determine the cost of moving remaining cars between I and J to the right in the consecutive configuration without moving J.



The only issue here is that we may have created more space than what is needed for the new car. To reduce the cost, we should first compare $L - I$ and $J - L$ (the number of cars we moved to the left and the number of cars we moved to the right).
Let W be the sum of empty spaces between cars I and J and let S be the space we created for the new car:

$$W = FS_L(J) - FS_L(I)$$
$$S = Length(NewCar) - W$$

Let's say that:

$$L - I \geq J - L$$

In this case, the cost for creating the empty space to fit the new car should be reduced by $S \cdot (L-I)$:

$$OptimumCost = Cost - (Length(NewCar) - W) * (L-I)$$

In the case that we moved more cars to the right than to the left the same logic applies using the values from FS_R and CM_R. After calculating $OptimumCost$ for each car as a first car (I), we should easily be able to pick the best one. The time complexity of the solution is dominated by the initial sort – the rest of the algorithm is linear. This means that the overall time complexity is $O(N \log N)$.

Problem E: Triangles

Statement:

You are given an array of positive integers. Find the maximal substring (i.e. a subset of at least three consecutive elements of the array) so that any three distinct elements from that substring can form the sides of a triangle. Also, find the maximal subsequence (a subset consisting of at least three elements, not necessarily consecutive) with the same property.

Input:

The first line of input contains one integer n, the number of elements in the array. The next n lines contain the elements of the array.

Output:

Output consists of exactly two lines, each containing one integer– the length of the maximal substring and the maximal subsequence with the property described above, respectively. If such substring or subsequence doesn't exist, the corresponding value is zero.

Constraints:

- $1 \leq n \leq 100{,}000$

Example input:

```
5
60
30
20
40
60
```

Example output:

```
3
4
```

Elements of the array are positive integers, each less than or equal to 10^9.

> Time and memory limit: 1.5s / 64MB

Solution and analysis:

Input size limit 100000 suggests that any solution to this problem should work in $O(n \log n)$ time (or faster).

Part a (substring):

To verify that some substring fulfills the requirement, it is enough to check if the sum of two smallest numbers in a substring is greater than the largest number in that substring. Indeed, if the inequality holds for these three elements, it will hold for any three numbers in that substring.
There are several ways to find the length of the longest such substring, two of which will be explained here.

Solution 1:

Suppose that we want to check if there exists a substring of length k with the described property. We can divide input array into slots of length k (the last slot may have less than k elements). For each slot $a[0]..a[k-1]$, we can compute the following arrays:
Prefix maximum: $leftMax[i] = max\{a[0], a[1], \dots, a[i-1]\}$
Suffix maximum: $rightMax[i] = max\{a[i+1], a[i+2], \dots, a[k-1]\}$
Prefix minimum: $leftMin[i] = max\{a[0], a[1], \dots, a[i-1]\}$
Suffix minimum: $rightMin[i] = min\{a[i+1], a[i+2], \dots, a[k-1]\}$
Prefix second minimum: $leftMin2[i] = min(\{a[0], a[1], \dots, a[i-1]\}\{leftMin[i]\})$
Suffix second minimum: $rightMin2[i] = min(\{a[i+1], a[i+2], \dots, a[k-1]\}\{rightMin[i]\})$
With this pre-calculation, we can find the maximum, minimum and second minimum of any substring of input array of length k in constant time. Namely, any substring of length k covers entirely one slot or lies in two consecutive slots, so min and max are straightforward to compute using suffix arrays of the left slot and prefix arrays of the right slot, while computing second min requires several comparisons between minima and second minima of two parts of the substring.
Acting as described, it is possible to check all substrings of length k in linear time. Doing binary search on k gives us an $O(n \log n)$ algorithm for the original problem.

Solution 2:

We start with the substring consisting of the first three elements of the input array. If the current substring has the required property, we move the right boundary of the substring forward, introducing a new element into it; otherwise we move forward the left boundary of the substring, removing one element form the substring.
To check if the substring has the triangle property, we can use one heap that extracts maximum, and one that extracts minimum - let's call the heaps M and m respectively. When the right boundary moves, we just put a new element into both heaps. Moving the left boundary requires removing one particular element from both heaps. Instead of removing that element immediately, we can use two auxiliary heaps (again, one for max and one for min, call them M_a and m_a) and put the element that should have been removed there. As long as that element is not equal to max of the heap M, it doesn't matter if it is present in the heap or not. So every time we extract the max of heap M, we also extract the max of M_a; if they are not equal, the max is regular and we can use it (we just put back the max of M_a); if the two maxima are equal, we do the (delayed) removal from both M and M_a, and get another max from both until they differ. We do the same with heaps m and m_a for extraction of the minimal element.
Getting min/max from the heap and putting a new element into all 4 heaps requires $O(\log n)$ time, so we can check one particular substring in logarithmic time. Since after each check one of the substring's boundaries moves forward, there are $O(n)$ substrings to check, so the total running time is again $O(n \log n)$.

Part b (subsequence):

It is easy to prove that if some elements of a sorted array form a subset with the described property, then the entire segment (from minimal to maximal element of the subset) also has the property. So, , the maximal subset of the original array with the described property is a substring of the sorted array.
Therefore, to solve part b, it is enough to sort the input array and then search for the longest substring using (any) solution of part a. The running time of such algorithm would be $O(n \log n)$ for sorting and O(n log n) for finding the longest substring, which gives $O(n \log n)$ in total.
It is also possible to find the longest substring with the given property in a sorted array more directly. Obviously, the two smallest elements are the first two elements of a substring, and the largest is the last one, so there is no need to use heaps or auxiliary prefix/suffix arrays to find minima and maxima of a substring. The running time in this case is still $O(n \log n)$ due to sorting, even though finding the longest substring in a sorted array can be done in linear time.

Problem F: Olympic Games

Statement:

Young boy, Oliver, has watched the Olympic Games this year for the first time. The number of countries which participated in the Olympics is n. There are k different sports and each country had its own representative in some of the k sports. In each sport the gold medal is won by exactly one country among the ones that have representatives for that sport. And, of course, for every sport there is at least one country which competes in it.
Oliver noticed that a small number of countries won a huge number of gold medals, and that a lot of countries didn't win any. Now, he is wondering what could be the minimal difference in the number of gold medals between the country which took the most and the country which took the least. Oliver is still too young to figure out the answer to this question, so please help him.

Input:

The first line contains n, the number of participating countries. The second line contains k, the number of sports. The third line contains the total number of competitors, m. Each of the next m lines contains two integers, c and s, which mean that country c had a representative in sport s.

Output:

Output contains only one integer – the minimal possible difference in the course country by ne country wins a gold medal depending on the implementation, but not more than O(N)f the algorithm number of gold medals between the country which took the most gold medals and the country which took the least.

Constraints:

- $2 \le n \le 100$
- $1 \le k \le 500$
- $0 \le c_i \le n-1$, $0 \le s_i \le k-1$ for each i
- No pair (c_i, s_i) is contained in the input more than once

Example input:

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

Example output:

```
1
```

> Time and memory limit: 0.5s / 64MB

Solution and analysis:

Let's describe the algorithm for solving this problem. Initially, no sport is assigned to any country. Going circularly through all countries we try to assign new sport to the currently considered country, while maintaining the number of sports assigned to other countries (actually, we try to find an augmenting path in a bipartite graph which starts from the country being considered and ends at some yet unused sport). If at some moment no new augmenting path for a particular country exists, we can skip that country in all subsequent iterations in order to save time. Since for every sport there is at least one country which competes in it, after a certain number of iterations each sport will be assigned to some country.
By this algorithm we get a matching where the number of gold medals taken by country with the minimal number of gold medals is maximal and the number of gold medals taken by the country with the maximal number of gold medals is minimal. Therefore, this matching has the desired property that the difference between these two values is minimal.
As the number of iterations which can find augmenting paths is at most $k + n - 1$ (there are at most $n - 1$ unsuccessful findings) and each augmenting path can be found in $O(E)$ time, where E is the number of edges, the time complexity of the solution is $O((n + k) \cdot n \cdot k)$. The memory complexity is $O(n \cdot k)$.

Problem G: Matrix

Statement:

You are given a square binary matrix A of dimension $N \times N$. Elements on the main diagonal are all ones. We want to compute the 77,686,783th power of this matrix (MDCS written in ASCII codes is (77,68,67,83)). To make things more interesting, we will define binary operations + (addition) and · (multiplication) as the following:

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

So basically, addition is logical OR and multiplication is logical AND.
The input matrix is too big for normal time constraints, so it will be given by listing all positions of ones in it. Also, for the output only the number of ones in the 77,686,783th power of the matrix is sufficient.

Input:

The first line contains two integers, N and M - dimension of the square matrix A and the number of ones in it. Each of the next M lines contains two integers x and y – which means that $A_{x,y}$ is equal to 1.

Output:

Output contains only one integer– the number of ones in the 77,686,783th power of the given matrix.

Constraints:

- *1 ≤ N ≤ 5,000*
- *N ≤ M ≤ 200,000*
- *Indices x and y from the input satisfy the condition 1 ≤ x,y ≤ N and these pairs are unique.*
- *The matrix elements are 0 or 1 and the elements on the main diagonal all ones. All elements that are not listed in the input have zero value.*

Example input:

```
4 8
1 1
1 3
1 4
2 2
2 3
3 1
3 3
4 4
```

Example output:

```
11
```

Example explanation:

From the input we have that $A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Its 77,686,783th power is $\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, which has 11 ones in it.

Note:

For two square matrices A and B with dimensions $N \times N$, we say that matrix C, with the same dimension, is product of these two matrices if:

$$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{2,1} \cdot B_{2,j} + \cdots + A_{i,N} \cdot B_{N,j}, \text{ for every } 1 \leq i, j \leq N$$

> Time and memory limit: 2.0s / 64MB

Solution and analysis:

At first glance, this problem requires fast multiplication of the sparse binary matrices with a given definition for operations. This is known and very hard problem for implementation. The standard approach for matrix multiplication gives as time complexity $O(n \cdot n^3)$ which is very big for the problem constraints. Logarithmic powering is also too slow. But, in our case solution has nothing to do with this – this is a graph theory problem.

Before we start the analysis of this problem, let's look at the adjacency matrix A of an arbitrary directed graph $G = G(V, E)$. As we know, the adjacency matrix is a binary one and its element $A[v][u]$ is equal to 1 if and only if there is an edge from vertex v to vertex u (directed edge). The adjacency matrix is a square one, so it is allowed to consider powers of this matrix: A^k, for every $k \in N_0$. From now on we will assume that operations are defined as in the problem statement.

Can we, with some corresponding graph property, define the square of the matrix A? The element at position (v, u) is going to be equal to one if and only if there is a vertex k such that $A[v][k] = 1$ and $A[k][u] = 1$. This means that $A^2\ [v][u]$ is equal to 1 if and only if there is a path of length 2 in the starting graph G. Using mathematical induction, we can prove the following property:

- *$A^k[v][u] = 1$ if and only if there is a path from vertex v to vertex u of length k.*

We have an additional property of the start matrix: elements on the main diagonal are ones. This means that all of the vertices have loops. In other words, we can "circle" around any vertex for an arbitrarily long time. So, if there is a path of length k between vertices v and u, then there is path of length k_1 between them for every $k_1 > k$ (we can just append a "circle" of length $k_1 - k$ to path). This means that, with the above property of matrix A, we have

- ***$A^k[v][u] = 1$ if and only if there is a path from vertex v to vertex u of length less than or equal to k.***

Now we can go back to our original problem. In the input we have a directed graph G with n vertices and m edges, where every vertex has a loop. From the above definition of the element $A^k[v][u]$, we have that starting from $k > n$ power of matrix A is not going to change. This is very important, because in this way we have to calculate the n-th power, and of course $n <$ 77,686,783. The problem can be reformulated as:

- *find the number of edges in the transitive closure of the given graph G,*
- *i.e. for every vertex calculate the number of vertices that are reachable from it*

Naïve approach for the transitive closure leads to complexity $O(n \cdot m)$ – graph tour, DFS or BFS, from every vertex separately. A better idea is to find the strongly connected components (SCC) first. In this way, submatrices for every component have all elements equal to one (so we do not want to "waste" time there). After finding the SCCs, we can shrink every component to just one "vertex". In this way we can obtain a directed acyclic graph (DAG).

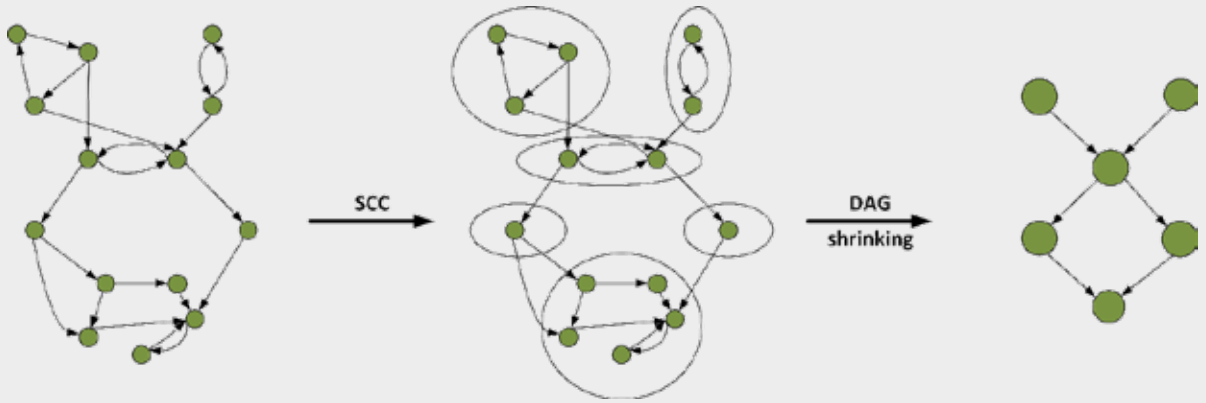


Figure 1. Creating DAG graph from SCC components

Things are a little bit easier. If we assume that for every component we have a list of all components that are reachable from it, we can easily transfer this to the start graph and calculate the final result. But how can we initialize these lists with the given time constraints?

For DAG we can find its topological sort order. We can initialize the lists for every component in this order, because it holds that by the time when we are examining some component, all components reachable from it are already initialized. Let us denote with $currentC$ the current component for which we want to initialize the list. Unfortunately, we cannot simply "connect" all lists from their neighbors, because there can be some duplicates (see Figure 2 for example). On the other hand, these lists can be $O(n)$ long, so the union of these sets must require passing through them multiple times.

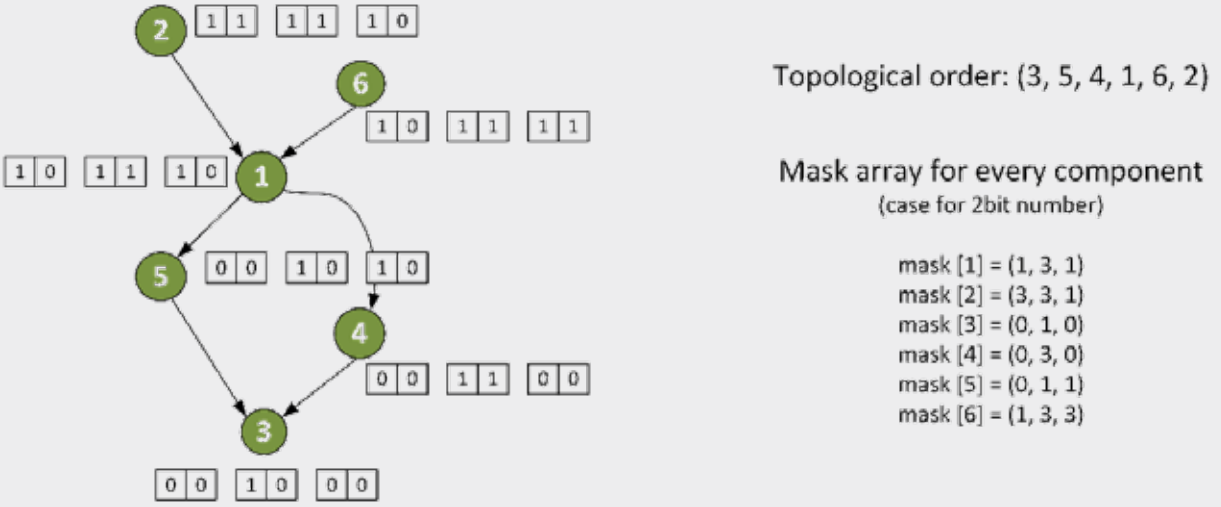


Figure 2. Example of a topological sort and mask arrays (for the case of 2-bit numbers)

The best way to maintain these "lists" is to store them in some sort of marked array (when a component is in the "list" we are going to mark the corresponding element). If we use simple boolean arrays, complexity will again be $O(n \cdot m)$. Idea is to use bit masks. For every component we are going to store an array $mask_{currentC}$ of length $\left\lceil \frac{numComponents}{64} \right\rceil$ of long type. This way we can mark some component c in this array as

$$mask_{currentC}[c\ DIV\ 64] = 2^{c\ MOD\ 64}$$

In other words, for each component there is a unique corresponding bit in every array. Now, we can initialize the array for $currentC$ by simply OR-ing the arrays for its neighbor's element-wise (which represents union). Note that although the complexity remains $O(n \cdot m)$, the reduction of the constant factor is very significant.

Complexity:

In this problem we have quite a pipeline of graph algorithms. First, the complexity of finding SCCs and building a DAG is $O(m)$. Finding the topological order has the same complexity. Finally, performing the dynamic programming approach for initialization of bit mask arrays as described above has $O\left(\frac{n \cdot m}{64}\right)$ complexity if we use a 64-bit integer type for bit masks. Indeed, every array is going to be iterated for every component which has an edge directed at the corresponding component for that array. So, for every edge we have one tour through some array. This brings us to the final complexity of $O\left(\frac{n \cdot m}{64}\right)$.

Test data:

The data corpus for this problem consists of 24 test cases. Test cases are created with one (or more) of the following methods:

- *Random generation of a binary matrix with given probability for* 1 *and* 0
- *Generation of a matrix that corresponds to a tree with some additional cross or / and up edges*
- *Generation of a matrix that corresponds to a path with some additional edges*
- *Creating a SCC graph from a tree with cross edges and blossoming a SCC components in every vertex*
- *Special cases (triangle, all ones, only loops…)*

Num	N	M	Comment
00	4	8	test case from problem statement
01	10	31	by hand
02	100	5187	Random with $p = 0.5$
03	1000	1000	All zeros except on the main diagonal
04	500	125250	One in the upper triangle
05	2000	161804	Random with $p = 0.0.3$
06	2500	127736	Random upper triangle with $p = 0.03$
07	3000	5999	Random tree structure
08	3000	36871	Path with down edges
09	3000	193412	Expanded SCC graph with $numComp = 450$
10	5000	161017	Expanded SCC graph with $numComp = 1000$
11	4000	67879	Random with $p = 0.0.3$
12	3000	182515	Random tree structure with cross edges
13	5000	9999	Random tree structure
14	5000	59870	Random tree structure with down cross edges
15	5000	194846	Expanded SCC graph with small $numComp$
16	5000	179948	Random with $p = 0.0.3$
17	5000	196411	Path with down edges
18	5000	189283	Random tree structure with cross edges
19	5000	188570	Random tree structure with down cross edges
20	5000	184753	Expanded SCC graph with $numComp = 1000$
21	1	1	Only one vertex
22	4500	118341	Components: SCC, Tree, Path, Random
23	5000	143657	Components: big SCC, Tree, Path, Random

Table 1. Test data description

Problem H: String covering

Statement:

We say that string B covers string A if A can be obtained by putting together several copies of string B, where overlapping between two successive copies of B is allowed but the overlapped parts must match. After connecting these copies, the whole generated string must match string A.
For example, string $B = abab$ covers string $A = ababab$, with two copies (see Figure 1). String $B = aba$ does not cover A, because the last character b cannot be covered.

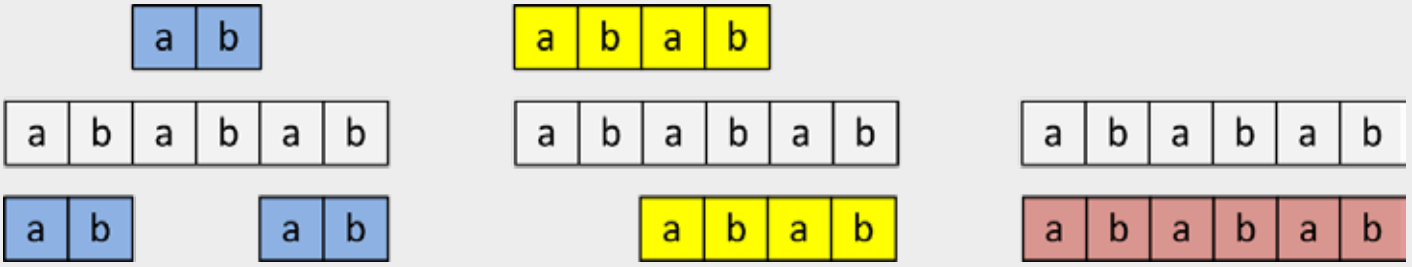


Figure 1. All possible coverings of string $A = ababab$.

You are given string A. Write a program that calculates how many strings B exist that cover string A in this way.

Input:

The first and only line of the input contains string A.

Output:

Output contains only one integer – the number of different strings that cover given string A.

Constraints:

- *1 ≤ length (A) ≤ 100,000*
- *String A consists exclusively of letters 'a' - 'z'*
- *Output the final solution modulo $10^9 + 9$*

Example input:

```
ababab
```

Example output:

```
3
```

> Time and memory limit: 1.0s / 64MB

Solution and analysis:

Problem H on this year's BubbleCup finals was a string problem. For a given string A it was asked to find how many strings there exist that covers this string. We say that string B covers string A if A can be obtained by putting several copies of string B where overlapping between two successive copies of B is allowed but the overlapped parts must match. After connection of these copies whole generated string must match string A. First thing that we can observe is that necessary condition for string B is that it is some prefix of the string A. So, the asked modulo in the task description is only a small trick.

Now we know that the final answer is going to be smaller or equal to n (solution is equal to n if and only if all char in A are equal). Naïve approach would be to check every prefix of the string A. When testing for the prefix $A^k = a_1 a_2 \dots a_k$ we have to find all occurrences of it as substrings in A and then see if these occurrences covers whole string A (here we can see that string B has to be an suffix as well). Complexity of this algorithm is approximately $O(n^3)$, but it can be reduced to quadratic with some hash functions for substrings. In any case this is too big for our constraints.

Can we, in some other way, track these occurrences for prefixes? Let us assume that for the prefix A^k occurrences in the string A starts at positions $1 = s_1 < s_2 < \dots < s_m$. For the prefix A^{k+1} start positions are subset of the start positions for A^k. Idea is to store these occurrences in some nice way that can be updated fast when adding new character on the end (moving to the next prefix).

For this we can use data structure: suffix array. Let us denote with s suffix array – $s[i]$ is equal to the start position of the i-th suffix in the lexicographic order. Then the positions for prefixes A^k are successive in the array s. For every prefix we can define some segment in the suffix array $[l_k, r_k]$ which means that suffices in this segment starts with given prefix. Nice thing is that these segments create an inclusive chain: $[l_{k+1}, r_{k+1}] \subset [l_k, r_k]$. In this way we can easily obtain segments.

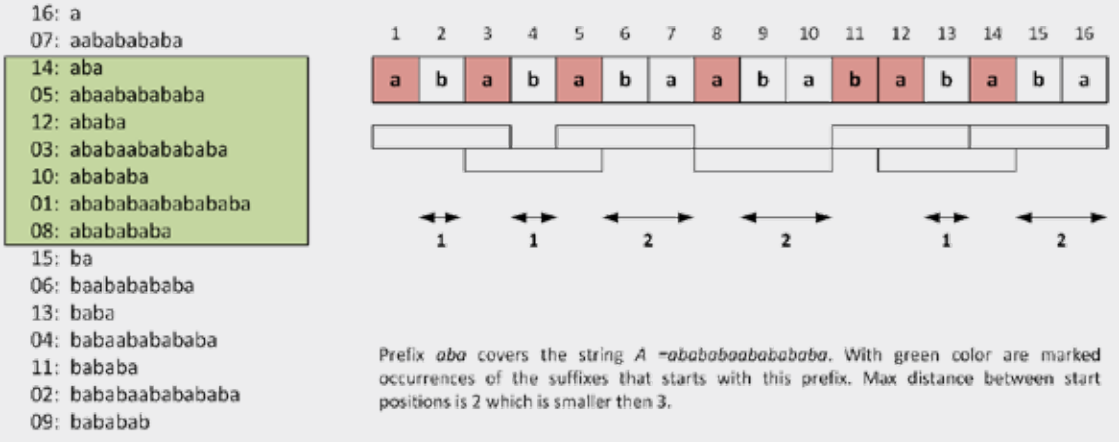


Figure 1. Example of suffix array and prefix check.

Now we have the start positions of occurrences for any prefix. Problem is to, in some efficient way, see if they cover the whole string A. For this we must use additional data structure – max heap. In heap we are going to store the lengths between two successive positions - gaps. When we remove some occurrence, we will remove two distances / gaps (from prior to current one and from current one to next occurrence) and add the new one which is the sum of the removed ones (from prior to next). Finally, we can state that current prefix covers the whole string if and only if the max element in the heap (max distance between two successive occurrences) is smaller or equal to the prefix length.

```
=============================================================================================
initialization of the suffix array S;
sol = 0;
for k = 1 to n - 1 do
   add in heap key-value pair (k, 1);
segmentLeft = 1; segmentRight = n;
for k = 1 to n do
   while (char at position k of segmentLeft-th suffix is different from A [k]) do
     remove from heap key prior and segmentLeft;
     add in heap key-value pair (prior, next – prior);
     inc(segmentLeft);
   while (char at position k of segmentRight-th suffix is different from A [k]) do
     remove from heap key prior and segmentLeft;
     add in heap key-value pair (prior, next – prior);
     dec(segmentRight);

   if (max in heap <= k)
      inc(sol);
endfor
=============================================================================================
```

Figure 1. Pseudo code for described algorithm

Complexity:

Sorting the suffixes with suffix array takes $O(n \log n)$ time. This can be implemented in the linear time but in our case, this is sufficient. After that for every prefix, in order as in the given string, we are going to maintain the above segments in linear time overall (in every step we are going to shrink current segment with only one compering of chars). For heap, every position is going to be added exactly one time and removed at most one time. Taking all of this in to account we get the final complexity:

$$O(n \log n + n \cdot 3 \log n) = O(n \log n).$$

Test data:

Test corpus consists of 15 test cases. In the random string we can expect only one covering string – the string itself. Many ideas for test generation is to create some "recursive" string that has many occurrences of prefixes in it. The "worst" case for this is to use small number of different chars. Description of the test data with some comments is given below.

Num	N	Solution	Comment
00	6	3	$ababab$
01	45690	9138	Concatenation of the string $aabcc$
02	90010	10	Concatenation of the string $a..ab..ba..ag$
03	90000	10000	Concatenation of the string $aaaabcaaa$ with random change
04	80.000	1	Many a with random char b
05	10.002	3334	Concatenation of the string aab
06	98.904	8242	Concatenation of the string $aaaaaabaaaaaaaaaaabaaaaa$
07	50.003	1	Random concatenation of strings aab and $aaab$
08	96.048	16	Concatenation of the string $acbacb$ with some xyz
09	99.999	1	String is of the form $a..ab\ldots ba..a$
10	99.999	99.999	All chars are the same
11	78950	3158	Concatenation $abcd\ldots xyz$
12	100.000	1	Many v with random char w
13	1	1	Only one char
14	90.000	5628	Concatenation of the string $abababaababababa$
15	100.000	100	Concatenation of the string $aa..ab$

Table 1. Test data description

Problem I: Polygons

Statement:

You are given n points with integer coordinates in the plane. After that you are given q queries.
Each query gives you a list of indices in the originally given set of points. The points in this list form a simple polygon (one that does not intersect itself). For each query you should output how many points from the original set are on the inside of the given polygon.

Input:

The first line of the input contains two integers n and q, separated with an empty space. Next n lines each contain two numbers, x and y - coordinates of a point. The q lines that come after that each start with a number p, the number of points that form that polygon. The rest of the line consists of p space-separated numbers that represent indices (indices are from 0 to $n - 1$) of originally given points that form the polygon.

Output:

You should output q lines, one for each query. Output for each query should be just one integer number: the number of points from the original set on the inside of the polygon given in that query.

Constraints:

- *$1 \leq n \leq 1{,}000$*
- *$1 \leq q \leq 10{,}000$*
- *Coordinates of a point are in the segment $[0, 10^6]$*
- *No two points are the same. Also, no three different points are on the same line.*
- *Vertices of the polygon do not count as being inside of it*

Example input:

```
7 2
0 0
0 4
4 0
4 4
2 1
3 2
2 3
3 0 1 3
3 0 2 3
```

Example output:

```
1
2
```

Example explanation:

First polygon is a triangle whose vertices are $(0, 0), (0, 4)$ and $(4, 4)$. There is one point on its inside: $(2, 3)$. Second triangle has the vertices $(0, 0), (4, 0)$ and $(4, 4)$. There are two points on its inside $(2, 1)$ and $(3, 2)$.

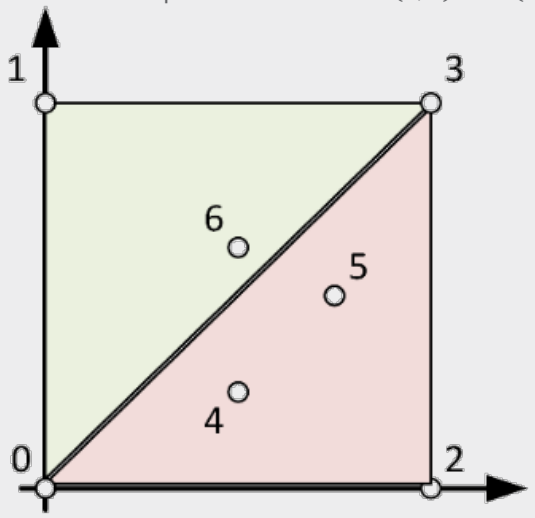


Figure 1. Visualization of the given example.

> Time and memory limit: 6.0s / 64MB

Solution and analysis:

Taking each query and checking each point is on the inside of that polygon in a straightforward manner would take $O(n^2 \cdot q)$ time, which is too slow for the given constraints.
We are going to deal with this by precalculating some things, which will allow us to answer each query in linear time with respect to the number of vertices of the polygon. For each two points from the original set, we calculate the number of points under the line segment connecting them (that are contained in the quadriteral formed from the endpoints of the line segment and their projections on the x axis; we don't count points on the edges of this quadriteral). Also, we calculate the number of points directly under each point from the set (i.e. the ones that have equal x and smaller y coordinates).

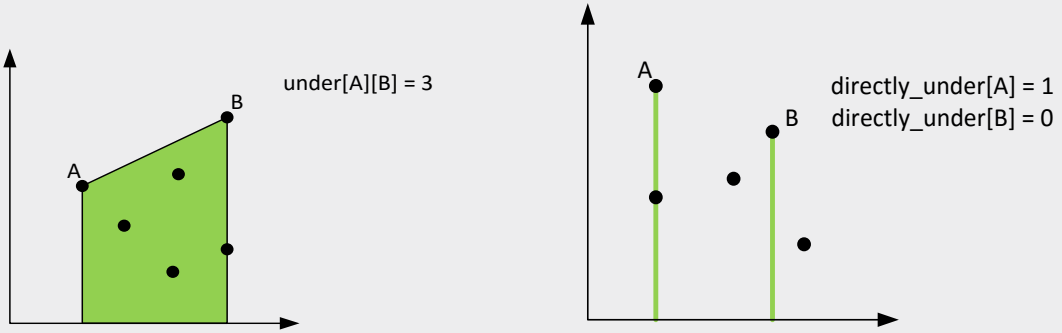


Figure 1. What is precalculated.

First, we sort the points by x coordinate, sorting points with equal x by y. The number of points directly under each point is easily calculated in linear time from this sorted array.
Now for each point (we'll call it point A) we take all the points after it in this sorted array (those that have larger x, or equal x and larger y coordinate) and sort them by angle in respect to point A. We go through these points in this order (we'll call the current point B). For each point A we keep an array that counts the number of times each x coordinate has appeared in points B that we went through until now. We keep this array as a cumulative table in order to be able to do insertions and calculations of the sum of the first k elements in $log(MAX_X)$ time for each operation, where MAX_X is the maximal value of x coordinates among the set of points. For each point B we do the following:

1. *We add 1 to the cumulative table in the position of the x coordinate of point B.*
2. *We calculate the number of points under the line segment A-B as sum in cumulative table to the position of the x coordinate of point B – 1.*

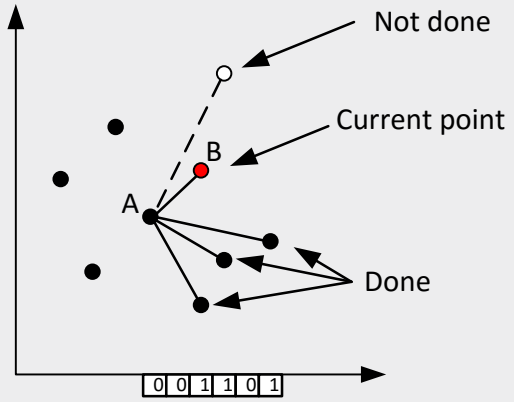


Figure 2. A typical state of a precalculation step.

We respond to each query in the following way. We go through the polygon in clockwise direction and for each edge, if it goes to the right (the x coordinate of its second point is larger than the x coordinate of its first point) we add the number of points under that edge to the sum, otherwise we substract this number from the sum. For each vertex, if we go through it going to the right (the edge coming into it and the one going out of it are both to the right) we add the number of points under it to the sum, if we go through it going to the left we subtract the number from the sum.

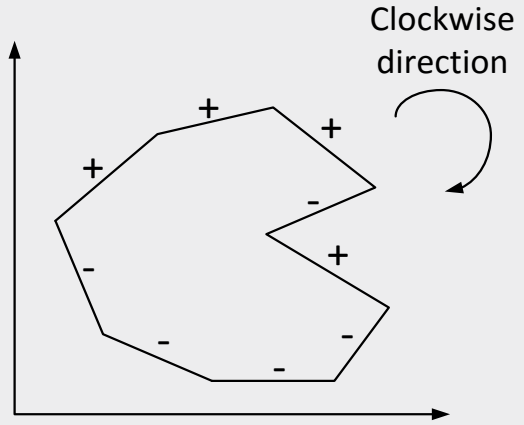


Figure 3. Responding to a query.

After going through all edges and vertices of the polygon, each point outside of the polygon is counted zero times and each point on the inside of the polygon is counted exactly once, which is exactly what we need.
Complexity:

Time complexity of initial calculation is $O(n^2 \cdot \log(MAX_X))$. *After that, each query is resolved in* $O(n)$ *time. The total complexity is therefore* $O(n^2 \cdot \log(MAX_X) + q \cdot n)$.