Problem A. Um_nik's Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	4 seconds
Memory limit:	512 mebibytes

Can you replicate my bachelor thesis in 5 hours?

I give you an undirected bipartite graph. Let K be the size of its maximum cardinality matching. Devise an *algorithm* to find a matching of size at least $0.95 \cdot K$.

If you want to get Accepted, I suggest you to optimize your code as good as you can.

Input

The first line contains three positive integers n_1, n_2 and m ($1 \le n_1, n_2, m \le 2 \cdot 10^6$) — the number of vertices in the first part, the number of vertices in the second part and the number of edges in the graph, respectively.

The next m lines describe edges, one per line. Description of each edge is two integers u and v ($1 \le u \le n_1$, $1 \le v \le n_2$) — the ids of vertices in first and second parts that are connected by the edge. There is no pair of edges connecting the same vertices.

Output

In the first line print one integer L — the size of the matching you found. The inequality $0.95 \cdot K \le L$ should hold.

In the next L lines print the ids of the edges in your matching. Edges are numbered from 1 to m in the order they are given in input.

Examples

standard input	standard output
3 2 4 1 1 2 1 3 1 3 2	2 1 4
20 20 20 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19 19 20 20	19 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Problem B. String Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	20 seconds
Memory limit:	512 mebibytes

We give you a string s of length n.

Let's fix some k ($1 \le k \le n$). Create $m = \lfloor \frac{n}{k} \rfloor$ strings of length k, the i-th of them being a substring of s starting with position $(i-1)k+1$: $p_i = s_{(i-1)k+1} s_{(i-1)k+2} \dots s_{ik}$.

In other words, we cut the string s into strings of length k and discard leftovers. Let $f(k) = |\{(i,j) \mid 1 \le i < j \le m,\ dist(p_i, p_j) \le 1\}|$, where $dist$ denotes the Hamming distance. In human words, $f(k)$ is the number of pairs of strings p that are different in at most 1 position.

We ask you to devise an *algorithm* to compute $f(k)$ for all k from 1 to n.

Input

The first line contains one positive integer n ($1 \le n \le 2 \cdot 10^5$) — the length of the string.

The second line contains the string s of length n, consisting of lowercase English characters.

Output

Print n numbers, the k-th of them being $f(k)$.

Examples

standard input	standard output
7 kkekeee	21 2 1 0 0 0 0
10 babaiskeke	45 2 0 0 0 0 0 0 0 0
11 aaabaaabaaa	55 10 2 1 0 0 0 0 0 0 0

Problem C. StalinSort Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	3 seconds
Memory limit:	512 mebibytes

There is an esoteric sorting *algorithm* called StalinSort. It goes as follows.

Go through the elements of the given array from left to right, starting from the second. If the current element is not less than previous, then do nothing, otherwise erase it. In the end you get a sorted array.

One can implement StalinSort in $O(n)$ time, which is pretty cool. There is a catch though: you can lose many elements in the process. To fix this I've come up with an improved nondeterministic version of this algorithm.

Go through the elements of the given array from left to right, starting from the second. If the current element is not less than previous, then do nothing, otherwise you have options. You can either erase it, or erase the previous element, but only if the current prefix still becomes non-decreasing. In the end you get a sorted array.

Depending on its choices, this algorithm can erase different number of elements. I wonder, what is the minimum number of erased elements I can get applying this improved version of StalinSort to the given permutation?

Input

The first line contains one positive integer n ($1 \le n \le 5 \cdot 10^5$) — the size of the permutation.

The second line contains n integers p_i ($1 \le p_i \le n$) — the permutation itself. It is guaranteed that all p_i are distinct.

Output

Print one number — the minimum number of elements improved StalinSort can erase.

Examples

standard input	standard output
6 6 1 2 3 4 5	1
6 5 6 1 2 3 4	4
6 6 4 5 1 2 3	3

Problem D. FFT Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1.5 seconds
Memory limit:	256 mebibytes

When I want to apply FFT *algorithm* to polynomial of degree less than 2^k in modular arithmetics, I have to find ω — a primitive 2^k-th root of unity.

Formally, for two given integers m and k, I should find any integer ω such that:

- $0 \le \omega < m$,
- $\omega^{2^k} \equiv 1 \pmod{m}$,
- $\omega^p \not\equiv 1 \pmod{m}$ for all $0 < p < 2^k$.

In this task, I ask you to find ω for me, or determine that it does not exist. Since we talk about application of FFT, I've set some reasonable **limitations for** k: for smaller k naive polynomial multiplication is fine, and for larger k FFT takes more than 1 second (we are competitive programmers after all).

Input

The only line of input contains two integers m and k ($2 \le m \le 4 \cdot 10^{18}$, $15 \le k \le 23$).

Output

Print any ω satisfying the criteria, or print -1 if there is no such ω.

Examples

standard input	standard output
998244353 23	683321333
1048576 15	64609
3 23	-1

Problem E. Binary Search Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

This is an interactive problem.

I have a hidden permutation $p_1, p_2, \ldots, p_n$. You are not to guess it. Your task is to devise a *data structure* (that's against the rules!) that supports the following operations on a set S, which is initially empty:

- "`add` x" — put element x in S,
- "`delete` x" — delete element x from S,
- "`getMin`" — print the element x from S such that p_x is the smallest among x in S.

You will have to perform "`getMin`" after each operation of other types.

You don't know the permutation, but you can make queries. In one query you can choose k distinct indices $x_1, x_2, \ldots, x_k$ for some value of k, and in return I will tell you the permutation of these indices $y_1, y_2, \ldots, y_k$ such that $p_{y_1} < p_{y_2} < \ldots < p_{y_k}$. In other words, I will sort the indices according to p.

Note that all x_i should be present in S at the moment of query.

It is easy to perform "`getMin`" in 1 query — just sort everything in S. It is also not hard to perform it using several queries with sum of k up to $O(\log n)$. Can you flex your *algorithm* (this is lame) muscles and satisfy both?

Note that since you don't know p and my task is to make your solution fail, I **can** change p depending on your queries, but only in such a way that all my previous responses are correct. I **can** also choose the order of operations you have to perform depending on your queries.

Input

Initially you are given a single line with one integer n ($1 \le n \le 8000$) — the number of elements. Each element will be inserted and deleted exactly once.

Interaction Protocol

Then there will be exactly $2n$ rounds of interaction.

Each round of interaction consists of 4 phases:

1. you read the next operation on a separate line: either "`add` x" or "`delete` x" for some $1 \le x \le n$;
2. you choose some $0 \le k \le \min(|S|, 30)$ and print $k+1$ numbers on a separate line: k first, then $x_1, x_2, \ldots, x_k$: the k elements you want to sort. Elements you choose should be between 1 and n, should be **distinct**, and should be in S at this time. Note that S is already changed according to phase 1;
3. you read k integers $y_1, y_2, \ldots, y_k$ on a separate line: y is a permutation of x you just printed, and $p_{y_1} < p_{y_2} < \ldots < p_{y_k}$;
4. you print a single integer x on a separate line, such that x is in S, and p_x is the smallest possible. Print -1 if S is empty.

It is guaranteed that all $2n$ possible operations ("`add` x" and "`delete` x" for all $1 \le x \le n$) will occur exactly once, and for each x operation "`add` x" will precede "`delete` x".

Do not forget to print end of line and flush your output before you read anything.

Example

standard input	standard output
3 add 1 1 add 3 3 1 delete 1 3 add 2 3 2 delete 3 2 delete 2	 1 1 1 2 1 3 3 1 3 3 2 2 3 3 1 2 2 0 -1

Note

In the example $p = [2, 3, 1]$.

Problem F. Face Recognition Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

I have drawn n dots on a plane and m straight segments connecting these dots. No segment goes through dots other than its endpoints, and no two segments intersect in any point other than their common endpoint. Also, if you start in one dot and move only by segments, you can go to any other dot. All n dots are in distinct positions.

Yes, that's a planar embedding of some connected graph with n vertices and m edges. Your task is to check if each face of this graph, including the outer face, is a triangle. Face is a triangle if and only if it is bounded by exactly 3 edges. If face is on both sides of some edge, this edge is counted twice.

Strive for excellence! Allow yourself nothing less than the best possible complexity for your *algorithm*.

Input

The first line contains two integers n and m ($3 \le n \le 10^5$, $n - 1 \le m \le 3 \cdot 10^5$) — the number of dots and the number of segments, respectively.

The next n lines contain coordinates of dots. All the coordinates are not greater than 10^9 by absolute value. All n dots are in distinct positions.

Each of the next m lines contains two integers u and v ($1 \le u, v \le n$, $u \ne v$) — the ids of dots connected by a segment. It is guaranteed that there are no two segments connecting the same pair of dots.

It is guaranteed that the input describes a valid planar embedding of a connected graph with all edges drawn as straight segments.

Output

If each face of the given graph, including its outer face, is a triangle, print "`YES`", otherwise print "`NO`".

Examples

standard input	standard output
3 3 0 0 1 0 0 1 1 2 2 3 3 1	YES
4 4 0 0 3 0 0 3 1 1 1 2 2 3 3 1 1 4	NO
4 6 0 0 3 0 0 3 1 1 1 2 2 3 3 1 1 4 2 4 3 4	YES
4 5 0 0 2 0 1 1 0 2 1 2 1 3 1 4 2 3 3 4	NO
4 3 0 0 0 1 1 1 1 2 1 2 2 3 3 4	NO

Problem G. Petr's Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1 second
Memory limit:	256 mebibytes

Petr is well-known for his unusual contests which shuffle well-established standings a lot. Each of his contests has a positive integer parameter k: its *unusualness*.

To predict results of such a contest with n participants, we can use the following *algorithm*: take an identity permutation of length n: $p_1 = 1$, $p_2 = 2$, ..., $p_n = n$ and then sequentially shuffle all segments of length k from left to right.

In other words, we perform $(n - k + 1)$ operations, where on the i-th operation we permute elements p_i, p_{i+1}, ..., p_{i+k-1} in random order so that all the permutations of these elements are equiprobable.

Given the resulting permutation p, can you recover the *unusualness* parameter k of this particular Petr's contest? To make things easier, we will only give you such tests that $20k \le n$ holds.

Input

The first line contains a single integer n ($40 \le n \le 10^5$) — the length of the permutation.

The second line contains n distinct integers p_1, p_2, ..., p_n ($1 \le p_i \le n$) — the resulting permutation. It is guaranteed that this permutation was generated using the algorithm described above for some k such that $20k \le n$.

Output

Print a single integer — the *unusualness* parameter k of this particular Petr's contest.

Example

standard input	standard output
40 2 3 4 1 6 5 8 9 7 11 10 12 14 13 15 17 18 16 19 20 21 23 22 25 26 24 28 27 30 29 32 33 31 35 36 37 38 34 40 39	2

Note

The line breaks in the example are added for clarity and do not exist in real tests.

Problem H. Greedy Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1 second
Memory limit:	256 mebibytes

I'm playing Super Mario Galaxy 3 (thanks for the copy, Nintendo). Most of the levels are small planets in a shape of sphere, but bonus levels are something different. They are in a shape of **torus**. You can imagine it as rectangle with both pairs of opposite sides glued together. As a tribute to 8-bit predecessors, the surface of the planet is a small rectangular grid. Each cell in the grid has its own height.

Playing the bonus level consists of two parts. In the first part I can terraform the level by applying zero or more operations. In one operation I choose a row or a column in the grid, and increase the heights of all the cells in that row or column by one. I can perform this operation any number of times with same or different rows and columns.

After the terraforming a coin appears at each common side of two cells of equal height, and I can collect them. I'm good at platforming, so collecting all the coins is not a problem. Designing *algorithm* for terraforming the level so that the maximum possible number of coins appear — that's the problem. The problem for you, actually.

Input

The first line contains two integers n and m ($2 \le n, m \le 50$) — the dimensions of the rectangular grid.

The next n lines describe the initial heights of all cells. The i-th of them contains m integers h_{i1}, h_{i2}, ..., h_{im}, where h_{ij} denotes the height of the cell with coordinates (i, j).

All the heights are between 0 and 500, inclusive. It is **not required** that this holds after the terraforming.

Output

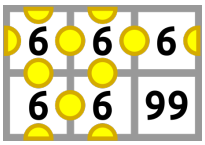
Print a single integer — the maximum number of coins can appear if I terraform the level optimally.

Examples

standard input	standard output
2 3 1 2 3 4 5 99	8
3 3 3 2 4 2 2 3 5 4 6	14
5 4 3 6 10 8 0 6 8 8 2 4 5 6 1 5 9 6 3 6 11 12	16

Note

The level from the first example after an optimal terraforming:



Problem I. Euclid's Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1 second
Memory limit:	256 mebibytes

Euclid's *algorithm* is one of the oldest algorithms known to mankind. It is used to find greatest common divisor of two given numbers. Your program should also take two numbers and find a greatest common divisor. And may Euclid be with you.

I give you two positive integers d and k. Your task is to find the largest integer that divides $(a+d)^k - a^k$ for every positive integer a.

Input

The only line contains two integers d and k ($1 \le d, k < 10^{100}$).

Output

Print a single integer — the largest integer that divides $(a+d)^k - a^k$ for every positive integer a.

Example

standard input	standard output
2 2	4

Problem J. Closest Pair Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	10 seconds
Memory limit:	512 mebibytes

There is a classic problem about finding two closest points amongst a set of points on a plane. There is also a classic randomized *algorithm* to solve it, which goes like this:

```
rotate the plane around the origin by a random angle phi
let p be the array of points
sort p by x coordinate in increasing order
ans = INF;
for (int i = 0; i < n; i++) {
    for (int j = i - 1; j >= 0; j--) {
        if (p[i].x - p[j].x >= ans) break;
        ans = min(ans, dist(p[i], p[j]));
    }
}
```

Here "`INF`" is some number greater than all distances. The function "`dist`" returns Euclidean distance between the given points. Note that all x coordinates are distinct after rotation with probability 1.

I know that the algorithm works well in practice, but how well exactly? I ask you to compute the expected number of calls of the function "`dist`", assuming that the angle "`phi`" is chosen uniformly at random from the range $[0; 2 \cdot \pi)$.

Input

The first line contains a single integer n ($2 \le n \le 250$) — the number of points.

The next n lines contains coordinates of points, one per line.

All the coordinates are not greater than 10^6 by absolute value.

It is guaranteed that all points are distinct. It is guaranteed that no three points lie on the same line.

Output

Print one number — the expected number of calls of the function "`dist`".

Your answer is considered correct if its absolute or relative error does not exceed 10^{-6}.

Formally, let your answer be a, and the jury's answer be b. Your answer is accepted if and only if $\frac{|a-b|}{\max(1,|b|)} \le 10^{-6}$.

Examples

standard input	standard output
4 0 0 0 1 1 0 1 1	5.0000000000000
3 0 0 0 1 1 0	2.5000000000000

Problem K. Interactive Algorithm

Input file:	*standard input*
Output file:	*standard output*
Time limit:	5 seconds
Memory limit:	256 mebibytes

This is an interactive problem.

I have a hidden permutation $p_1, p_2, \ldots, p_n$. You are to guess it.

You can make some queries. In one query you tell me a permutation $q_1, q_2, \ldots, q_n$ of length n, and I reply you with *similarity* of permutations p and q.

The *similarity* of two permutations is defined as follows. Let $w_1, w_2, \ldots, w_n$ be a permutation, then define $N(w)$ as the set of unordered pairs of adjacent elements in w. For example, $N([4,1,3,2]) = \{\{1,4\},\{1,3\},\{2,3\}\}$. This way, the *similarity* of p and q is the size of $N(p) \cap N(q)$.

You can make at most 25 000 queries. Note that no *algorithm* in the world can distinguish between p and reversed p, so both of these permutations will be accepted as correct answer.

This time I will not mess with you and **will not** change the hidden permutation. Though I could. You should be thankful, really.

Input

Initially you get a single line with a single integer n ($2 \le n \le 400$) — the size of the hidden permutation.

Output

When you know the hidden permutation, print an exclamation mark "`!`" and then n integers $p_1, p_2, \ldots, p_n$, or $p_n, p_{n-1}, \ldots, p_1$.

This does not count towards query limit.

Interaction Protocol

To make a query, print a question mark "`?`" and then n distinct integers $q_1, q_2, \ldots, q_n$ on a single line ($1 \le q_i \le n$).

In response read one integer s ($0 \le s < n$) on a single line, the similarity of p and q.

Do not forget to print end of line and flush your output after each query. You can make at most 25 000 queries.

Example

standard input	standard output
5 1 3 4	 ? 1 2 3 4 5 ? 2 4 3 5 1 ? 3 4 2 5 1 ! 3 4 2 5 1

Problem L. Not Our Problem

Input file:	*standard input*
Output file:	*standard output*
Time limit:	2 seconds
Memory limit:	256 mebibytes

Someone (not me!) calls an array of length n consisting of non-negative integers *good* if and only if $a_i \cdot a_{i+1} \cdot \min(a_i, a_{i+1}) \le C$ holds for all $1 \le i < n$.

Someone else (we have nothing to do with them!) gives you an array of length n with some blank positions. Calculate the number of ways to fill in the blank positions so that the array is good, or determine that there are infinitely many ways. If the answer is finite, print it modulo 998 244 353.

Input

The first line contains two integers n and C ($1 \le n \le 10^6$, $0 \le C \le 10^{18}$) — the length of the array and the constraint on the product.

The second line contains the array a. Each element is either `-1`, which means that it needs to be filled, or between 0 and 10^{18}, inclusive.

Output

If the number of ways is infinite, print -1, otherwise print the number of ways modulo 998 244 353.

Examples

standard input	standard output
4 100 -1 7 2 -1	104
4 100 10 -1 -1 1	240
1 0 -1	-1
2 10 10 10	0