bubble cup

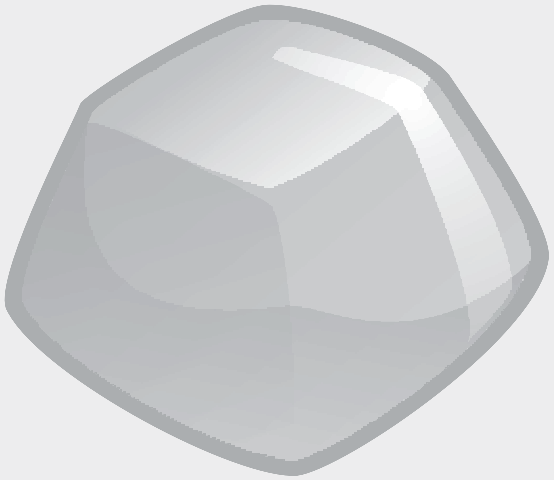
booklet

Content:

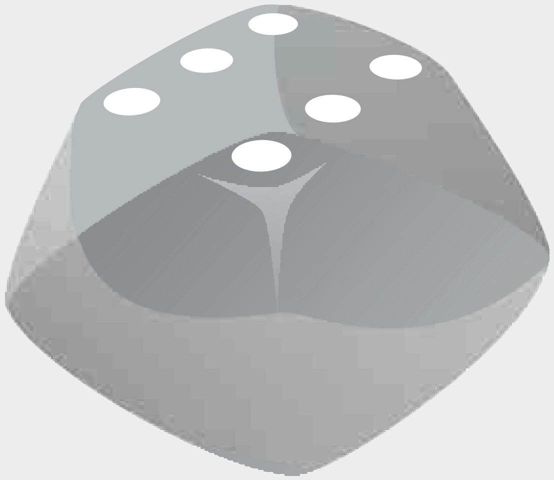
Welcome

This book contains all problems from Bubble Cup finals from 6 to 10. 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 6



Problem A: Game

It is given a tree with N nodes and number K. Lou is playing a game on that tree in such way:
In one move, he chooses one node and some K neighbors of that node (he can choose only node with at least K neighbors) and destroys them all. Destroyed nodes can't be used later.
Since he wants that his game lasts as long as possible, tell him what is the maximal number of nodes, that he can destroy.

Input:

The first line contains two integer numbers N and K. Each of the next $N - 1$ lines contain two integer numbers A and B (in the range $1..N$) which represents that there is the edge in the tree between nodes A and B.

Output:

Output should contain one integer number which represents the maximal number of nodes that Lou can destroy.

Constraints:

- $1 \le N \le 10^5$
- $0 \le K \le 10^5$
- $1 \le A, B \le N$

Example input:

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

Example output:

```
9
```

Example explanation:

In the first move, he can choose node 4 and his neighbors 8 and 9 and destroy them all. In the second move he can chose node 3 and his neighbors 6 and 7, and finally, he can choose node 2 and nodes 1 and 5.

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

The task is to find how many disjunctive (K+1) stars there are in a tree.
Usually, one of the first things that come to the mind in the problem like this is dynamic programming in a tree, which leads to the solution in this task as well.
For each vertex (node) x, we will calculate the maximum number of disjunctive stars that we can make in its subtree. To do that, we will need several values:

- *d[x].Zero – the maximum number of stars that we can make if we do not take vertex x at all*
- *d[x].Kdown – the maximum number of stars that we can make if we take vertex x and its K children*
- *d[x].Onedown – the maximum number of stars that we can make if we take vertex, one of his children and K-2 children of chosen children*
- *d[x].Kminus1down – the maximum number of stars that we can make if we take vertex x, his father and his K-1 children*
- *d[x].MaxWithoutFather – max(d[x].Zero, d[x].Kdown, d[x].Onedown).*

We will calculate these values in a following way:

- *For d[x].Zerodown is easy, just sum up all d[u].MaxWithoutFather for all u where u is child of x.*
- *It is a bit harder for d[x].Kdown and d[x].Kminus1down. First, we will show how to calculate d[x].Kdown and then d[x].Kminus1down can be computed in a similar way.*

Suppose that the set S is a set of K vertices that we take to form a star with node x (their father). The number of stars in a subtree of node x will then be:
In all formulas below, u is every child of x

$$\sum_{u}^{u\in S} d[u].Zero + \sum_{u}^{u\notin S} d[u].MaxWithoutFather$$

Therefore, we want to find the set S for which that sum is maximal. The sum above can be written in a following way as well:

$$\sum_{u} d[u].MaxWithoutFather - \sum_{u}^{u\in S} (d[u].MaxWithoutFather - d[u].Zero)$$

Because first sum is constant for all sets S, we should just choose K children (u) for which $d[u].MaxWithoutFather - d[u].Zero$ *are minimal. We can do that simply by sorting children by that value. For d[u]. Kminus1down it's all the same, just instead of K children, we will choose first K-1 children.*
For d[x].Onedown, we should choose node y for which this sum is maximal:

$$\sum_{u} d[u].MaxWithoutFather - d[y].MaxWithouthFather + d[y].Kminus1down$$

And that is the node with maximal $d[y].Kminus1down - d[y].MaxWithouthFather$.
After all these values calculated, the answer is simple: d[root].MaxWithoutFather.

Problem B: Turing

You are given a Turing machine. Turing machine in this task consists of

1. A **state** register that stores the state of Turing machine. There are 52 different states, labeled with lowercase and uppercase letters of English alphabet (‘a’ – ‘z’ and ‘A’ – ‘Z’). At the beginning, the machine is in the state ‘a’. State ‘F’ is the final state – the execution of the machine halts if the machine gets into state ‘F’.
2. A **tape** divided into 16 cells, one next to the other (a row of cells). Each cell contains either a symbol 0 or a symbol 1. At the beginning, symbol of each cell is set to 0.
3. A **head** that can read and write symbols on the tape. The head is a pointer to one cell. The head can move to adjacent cells (left or right), but it must not move outside of tape bounds. At the beginning, the head points to the first (leftmost) cell.
4. A **program** for the machine – set of instructions (transition functions). Instruction is a 5-tuple: $Q_c\ b_c\ \rightarrow Q_n\ b_n\ D$ that, given the *state*(Q_c) the machine is currently in and the *symbol*(b_c) it is currently reading (symbol of a cell which the head is currently pointing to) tells the machine to do the following in sequence:
   a. Write the new *symbol* (b_n) to a cell which the head is currently pointing to
   b. Move the head, which is described by *direction* (D) (‘L’ – left; ‘R’ – right; ‘N’ – stay at the same cell)
   c. Change the state of the machine to the new *state* (Q_n)

One operation is one execution of the instruction. Your task is to write a valid program for this Turing machine that does at least 1024 operations.
Program is valid if it halts (reaches the final state), head does not move outside of tape bounds, instruction set does not contain duplicate $Q_c\ b_c$ pairs and machine never reaches a pair $Q_c\ b_c$ such that no instruction is defined for the pair (unless Q_c is the final state).

Input:

There is no input.

Output:

The first line of output should contain one positive integer N – the number of instructions. Each of the next N lines should contain 5 characters separated by spaces representing one instruction – $V\ W\ X\ Y\ Z$
V represents Q_c; W represents b_c; X represents Q_n; Y represents b_n; Z represents D;

$V, X \in \{'a', \dots, 'z', 'A', \dots, 'Z'\};$
$W, Y \in \{0, 1\};$
$Z \in \{L, R, N\}$

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

Turing machine was invented in 1936 by Alan Turing. It is a hypothetical device representing a computing machine. Turing machine is important in computational complexity theory, because it is easy to analyze mathematically, and it is believed it is as powerful as any other model of computation. The Church-Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine. There are different types of Turing machines, and some are used to define complexity classes, such as deterministic Turing machines, non-deterministic Turing machines, probabilistic Turing machines etc. However, Turing machine in this task is not a real Turing machine, because it does not have infinite resources like the usual Turing machine. The tape is not unlimited and there is only 52 *possible states of the machine.*
This solution to this task is intended to be made "by hand". While it should be possible to write a program that will write a Turing machine program, it is much easier to use our human intelligence and intuition to come up with the solution to this task. There are several different ideas for programs that would execute at least 1024 *operations. We are going to present one that is rather simple, has potential to do a lot more than* 1024 *operations and does not need a lot of instructions.*
Since the cells of the tape can have either symbol 0 *or symbol* 1*, we can view the tape as a binary number, which is the core of the idea. By counting numbers on the tape (incrementing by one), we can count* 2^{16} *numbers, since the tape has* 16 *cells. This alone is greater than* 1024 *operations, and we have not even taken into account the operations needed to increment each number. Of course, it might not be possible to count all the* 2^{16} *numbers because we have to stop somehow, but it is not needed.*
Our program could count up to $2^9 - 1$*, which should be more than enough when we take into account the operations to increment each number. There are several different ways to achieve this. The more significant bits will be to the right on the tape, so the least significant bit of the number will be the leftmost bit. It is possible to use the leftmost cell as the least significant, but it is better to use it for something else, so we can shorten our program.*
At the beginning, we will set the value of the leftmost cell to 1*. This will be an indicator that it is the end of the tape. It will be used for the loop, which is going to be explained later on. We are going to use a different state for each cell when going to the right. So, for the first bit we will use* $state_1$*, for the second* $state_2$*, etc. (states can be mapped to English alphabet letters at the end). The most significant bit is marked with* $state_9$ *(because we are using* 9 *cells to represent a number). When incrementing a number, we start incrementing from the lowest bit.*
If the bit is incremented from 1 *to* 0*, we carry one to the next bit to the right. When moving the head to the cell right, we go from the* $state_i$ *to* $state_{i+1}$*. The special case is when we reach* $state_9$ *and increment from* 1 *to* 0 *and it means that we have counted all the numbers and can enter the final state and finish the execution.*
If the current bit is incremented from 0 *to* 1*, it means that we do not have to carry one and we have finished the incrementation phase. Now we have to return to the least significant bit. This is a part where the* 1 *in the leftmost cell comes in handy. All the cells left from the one the head is currently pointing at will have a value* 0*, except the one in the leftmost cell. We can write a simple loop that will move us to the leftmost cell, and then we can just use an additional instruction to move to the cell for the least significant bit (one to the right).*
The loop is simple – while there is a 0 *in the current cell, stay in the same state and move to the left. The loop ends when we read the symbol* 1*. We then change the state and move to the right (lowest bit).*
Example of this Turing machine loop:

```
X 0 X 0 L
X 1 Y 1 R
```

When we reach the least significant bit, we can start with the incrementation phase again. This program needs only about 10 *different machine states and it is quite short. The full program is given below:*

```
21
a 0 x 1 R
x 0 x 1 N
x 1 b 0 R
b 0 L 1 L
b 1 c 0 R
c 0 L 1 L
c 1 d 0 R
d 0 L 1 L
d 1 e 0 R
e 0 L 1 L
e 1 f 0 R
f 0 L 1 L
f 1 g 0 R
g 0 L 1 L
g 1 h 0 R
h 0 L 1 L
h 1 i 0 R
i 0 L 1 L
i 1 F 0 R
L 0 L 0 L
L 1 x 1 R
```

Problem C: Code

Gennady just passed lection Recursion on BubbleBee site, and then, he easily solved one task writing just a few lines of code:

```
int_64 a[N];
int_64 sum(int x, int y) {
    int_64 s;
    s = 0;
    for(int i = x; i <= y; i++) s+=a[i];
    return s;
}
int_64 solve(int_64 left, int_64 right, int_64 index) {
    int_64 tmp,res;
    if (left == right) return a[left];
    res = solve(left, index, left) + sum(left, index) + solve(index+1, right, index+1) + sum(index+1, right);
    if (index+1 < right) {
        tmp = solve(left, right, index+1);
        if (tmp < res) res = tmp;
    }
    return res;
}
int_64 get_Answer() {
    read N;
    read array A of N elements;
    return solve(1, N, 1);
}
```

Unfortunately, he hasn't learned about complexity of algorithms yet, so he didn't know that his code is very inefficient. Please, help him to efficiently solve this task for all test cases.

Input:

The first line contains one integer N. Next line contains N integers that represents array A.

Output:

Output the same number that function get_Answer() from Gennady's code returns.

Constraints:

- $1 \leq N \leq 3{,}000$
- $0 \leq A_i \leq 10^9$

Example input:

```
5
3 5 1 2 7
```

Example output:

```
57
```

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

First we should notice that the function solve() calculates the solution for an interval $[left, right]$, while the index is used to iterate through that interval as a loop. Then, we could make an equivalent solve2() function:

```
int_64 solve2(int_64 left, int_64 right) {
    int_64 res;
    if (left == right) return a[left];
    res = infinity;
    for(int index = left+1; index < right; index++) {
        res = min(res, solve2(left, index) + sum(left, index) + solve2(index+1, right) + sum(index+1, right));
    }
    return res;
}
```

Now we can use a matrix $d(NxN)$ to store the solution for each $[left, right]$ interval, and not make further recursive calls (this method is also known as memoization). The code complexity would then be $O(N^3)$, as the sum() function could be calculated in $O(1)$ in a well-known way, storing prefix sums.
As $O(N^3)$ is not good enough, we should further optimize the code.
Let $mid_{point[left,right]}$ be the index for which the solution of function solve2(left, right) has been found. Then, it can be proven that the inequalities $mid_{point[left,right]} \geq mid_{point[left,right-1]}$ and $mid_{point[left+1,right]} \leq mid_{point[left,right]}$ are always valid. This is also known as the Knuth optimization1.

[1] The Art of Computer Programming, Donald Knuth

Now, if we keep track of mid_{point} *for each interval in a separate matrix, we could adjust the range in the solve2() function loop in which we search the* $mid_{point[left,right]}$ *index, as shown in the following equivalent solve3() function:*

```
int_64 d[N][N] = {infinity}, mid_point[N][N];
int_64 solve3(int_64 left, int_64 right) {
    int_64 res = infinity;
    if (d[left, right] != infinity) return d[left, right];
    if (left == right) {
        mid_point[left, left] = left;
        d[left, left] = a[left];
        return d[left, left];
    }
    for(int index = mid_point[left, right-1]; index < mid_point[left+1, right]; index++) {
        tmp = solve3(left, index) + sum(left, index) + solve3(index+1, right) + sum(index+1, right);
        if (tmp < res) {
            mid_point[left, right] = index;
            res = tmp;
        }
    }
    d[left,right] = res;
    return res;
}
```

As each element is checked exactly once while increasing the right interval limit, as well as exactly once while increasing the left interval limit, the overall complexity is $O(N^2)$*. The exact mathematical proof of this is left to the reader to devise.*

Problem D: Jumping

Egor has written an array which consists of N natural numbers and in which all numbers are smaller than or equal to N. Now he is playing with it in a following way:
In the beginning, he circles the first number in the array.
Whenever he circles a number X, he moves to the X^{th} element in the array, circles it and repeats the procedure.
Egor always makes exactly 94294032599379535 steps and then stops playing. However, he has noticed that, sometimes when he finishes the game, some elements of the array have never been circled, and he does not like that. Because of that, he wants to change some elements of the array before he starts playing in such a way that, in the end, every number has been circled at least once. Help him to calculate the minimum number of elements that he needs to change in order to reach his aim.

Input:

The first line contains one integer N number of elements of the array. Second line contains N integers – elements of the array.

Output:

Output contains one integer representing the minimum number of elements that Egor needs to change in order to reach his goal – to circle every element in the array.

Constraints:

- $1 \leq N \leq 10^6$
- $1 \leq A_i \leq N$

Example input:

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

Example output:

```
1
```

Example explanation:

If Egor doesn't change any of the elements, then he will first circle the first number (2), then the second number (3), then the third number (2), then the second number (3), then the third number… and in the end only the first, the second and the third number will be circled.
If he changes the third number into 5, all elements will be circled.

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

This problem is not difficult, but there is a number of edge cases we have to be careful about.
First, let's rephrase the problem statement in terms of graph theory. We have a directed graph with N vertices, and with exactly one edge going out of each vertex. Our goal is to transform it into a graph from which, starting at node 1, we can traverse the entire graph – and we have to do it with the minimal number of changed edges. In most cases this will have to be a full cycle. The only other option is a path starting from vertex 1, traversing the entire graph but not looping back into vertex 1.

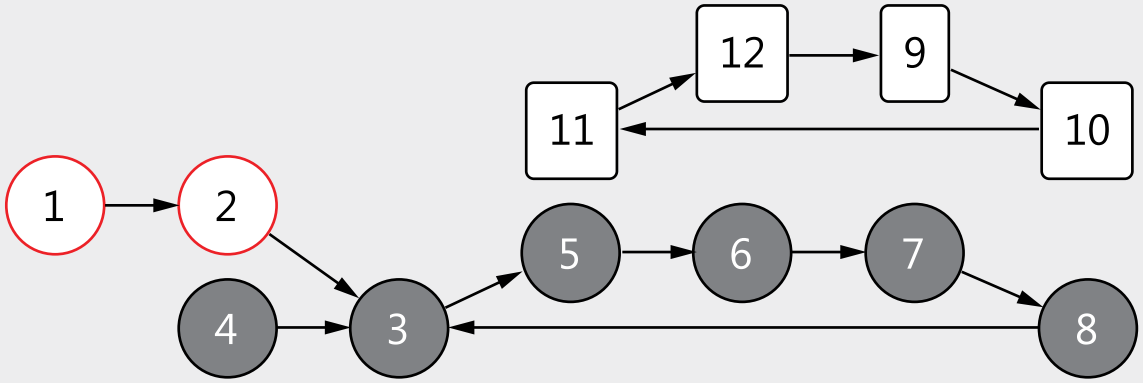


Figure 1. An example of dividing the graph into three types of shapes – a line (1, 2), tail (3-8) and cycle (9-12)

What is the shape of our original graph? It is easy to see that the weakly connected components of the graph are cycles, some of which have trees attached to them (like vertices 1, 2 and 4 in Figure 1). We will split each of these graph-tree components into "tail" and "line" shapes, by picking a vertex with an in-degree of 0 and following the path from it until we close a cycle or hit a vertex that we have already assigned to another line or tail.
Obviously, the split into lines, tails and cycles will in general not be uniquely determined for a graph, but the total number of shapes will. This is because each weakly connected component will contain either exactly one shape (if it is a cycle) as many shapes as it has vertices with an in-degree of zero (if it is not). We will denote this number with K.
Let us first concentrate on the case where we construct a full cycle. We can claim the following:

Lemma 1:

If the graph is not already a full cycle, the number of alterations needed to make it a full cycle cannot be less than K - the number of shapes in the graph.

Outline of proof:

If the graph consists of just a single tail shape, we obviously need at least one alteration. Otherwise, we can notice that none of the shapes contain the entire set of vertices. This means that each shape will need at least one edge which connects it to the other shapes, so it cannot remain unaltered.
Now let's construct an algorithm that makes exactly K alterations. For each shape we will select a start vertex and an end vertex. For tails and lines there is a unique pick, while for cycles we can pick a vertex at random and make it both the start and the end. We then apply the following algorithm:

1. *Initialize a set of selected shapes $S = \{\emptyset\}$*
2. *Pick a shape s and add it to S*
3. *Select a shape $t \notin S$, and alter the edge going out of the end vertex of s so that it points to the beginning vertex of t*
4. *If there is no such shape t, connect the end vertex of s to the beginning vertex of the shape which was picked first and terminate the algorithm*
5. *Repeat the steps 2 and 3 with the shape t in place of s*

It is clear that this algorithm ends up in the correct state after K iterations, each iteration altering one edge. So, since there is always a way to solve the problem with K changes, and by lemma 1 we know that we can never solve it with less than K, this means that the answer is exactly K.

Here we can notice that the change to the algorithm in order to cover our other end state (a path that begins with vertex 1 but does not necessarily loop back to it in the end) is minimal. Namely, if the vertex 1 was the starting point of a tail or a line, we will pick that shape first, and omit the step in which we connect the end vertex of the last shape to it. And if vertex 1 was part of a cycle, we will pick it for both the start and end vertex, pick that cycle first and proceed as in the previous case.

This can be done in the following manner:

1. *First, check if the graph already satisfies the conditions of the problem; if it does, output 0 and terminate.*
2. *Find all vertices that do not have any incoming edges. Denote their number with B. These vertices are beginnings of tails and lines. If vertex 1 is among them, set a flag F.*
3. *Mark all vertices that belong to shapes beginning with vertices from the previous step. All unmarked vertices are parts of cycles. If vertex 1 is unmarked, set the F flag.*
4. *Count the number of cycles among the unmarked vertices. Denote this number with C.*
5. *If F is not set, output $B + C$ as the solution and terminate. Else, output $B + C - 1$ as the solution and terminate.*

Each of the first four steps takes $O(N)$ time – notice that no vertex needs to be visited more than once in any individual step – so the whole solution takes $O(N)$ time as well. The memory complexity is also $O(N)$.

Problem E: Hacker

Bob Bubbles and his brother Bub Bubbles always competed with each other. Since Bub Bubbles was making a lot of bubblars (money currency in Bubbleland) with his scamming machine, Bob Bubbles thought about hacking into a bank, so he can have more bubblars than his brother. In order to do that, he has opened N programs on his computer. Each program is a rectangular window with sides parallel to the sides of his high-resolution screen. Windows can overlap. He also needed a special program Bubble Tracker to show him how close is he to be caught by cyber police, so he could prevent it. The window of Bubble Tracker is a square and Bob wanted it always to be visible (it cannot intersect with other windows and it must be inside the screen, but its sides can lie on the sides of other windows). He also needed it to be as big as possible, but he couldn't resize any of the already opened windows because it would lower his performance, so he asked you for help. Help Bob Bubbles find the maximal length of the side of Bubble Tracker that can be fit in the screen, so he can beat his brother.

Input:

The first line of the standard input contains two numbers W and H, representing width and height of the screen, respectively. Bottom-left corner of the screen is $(0, 0)$ and top-right corner is (W, H). Next line contains number N, number of programs Bob Bubbles has opened. Each of the next N lines contains four integers $X_i, Y_{i,}, W_i, H_i$, coordinates of bottom-left corner, width and height of the i_{th} window respectively.

Output:

On the first and only line of the standard output print the maximal length of the side of Bubble Tracker window.

Constraints:

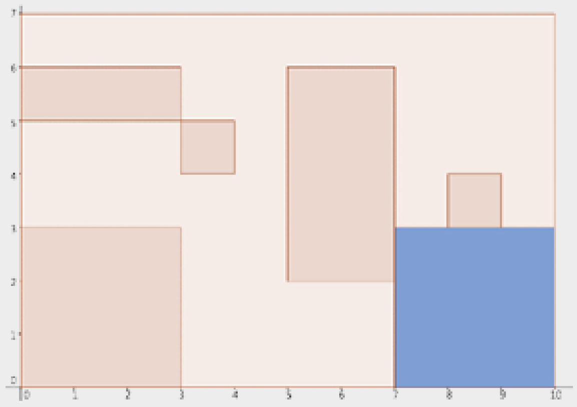
- $1 \le H, W \le 10^9$
- $1 \le N \le 40\,000)$
- $1 \le X_i, Y_i, W_i, H_i \le 10^9$
- $X_i + W_i \le W, Y_i + H_i \le H$

Example input:

```
10 7
5
0 0 3 3
3 4 1 1
0 5 3 1
5 2 2 4
8 3 1 1
```

Example output:

```
3
```



Example explanation:

One of the possible solutions of the sample test is shown in the picture on the right with bottom-left corner in (7, 0).

> Time and memory limit: 1.0s / 256MB

Solution and analysis:

We are given starting rectangle with coordinates (0, 0, width, height) and N rectangles in it. Our task is to find length of the biggest square that can be placed in the starting rectangle so that it doesn't overlap with any other of the N given rectangles. Suppose there is only one rectangle R(0, 0, x, y). We are going to resize it by some value k so it becomes R1(0, 0, x + k, y + k). We can notice that for any point A that is not in the resized rectangle R1 (it can lie on the edge of rectangle R1 because sides of windows in the task can touch each other), we can put square S with side of length k and top-right corner in A so that S doesn't overlap with starting rectangle R. Also, for each point B that is inside rectangle R1 we cannot put square S with top-right corner in B because it will overlap with starting rectangle R. For example, if B is in the resized part of the rectangle, either left or bottom edge of S will be inside rectangle R because length of the resized part is k.

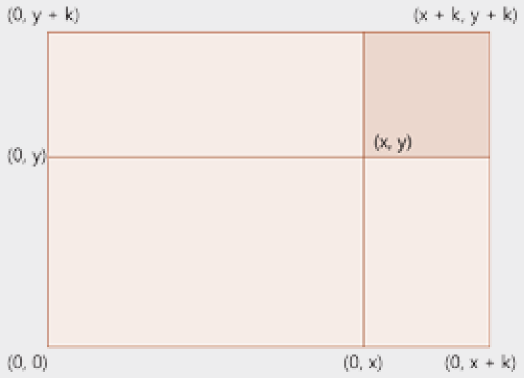


Image 1. Rectangle (0, 0, x, y) and resized rectangle (0, 0, x + k, y + k)
If we resize each rectangle by k (if rectangle resized by k goes out from the starting rectangle then it should be trimmed so it stays inside) and there exists point A which is not inside of any resized rectangle, we can put square S with top-right corner in A such that it doesn't overlap with any of the original (non-resized) rectangles. Notice that bottom and left edges of the starting rectangle (0, 0, 0, height) and (0, 0, width, 0) should be considered as rectangles too. We should only check if some point A exists (we don't need to know its location).
Solution will always be an integer and that comes from the fact that if there exists solution k' for which opposite sides don't lie on two rectangles then there exists some solution k > k' (we can expand square until it touches at least two rectangles). And since all the inputs are integers then the distance between two rectangles that are "touched" is integer and so is the solution.
*How to check if there is a point which is not inside any rectangle? Point must be inside of the starting rectangle (0, 0, width, height) and rectangles are inside of the starting rectangle, so we are going to find area of union of rectangles and check if it equals to area of the starting rectangle (height * width). If it differs then there exists some point A which is not in any of the rectangles, but also not lying on the edges of rectangles, which should be allowed. This issue can be solved using the fact that all numbers are integers thus if we expand rectangles by value k-1 and there is a point A(x, y) which is not inside any rectangle and not lying on the edges of any rectangle then there is a point B(ceil(x), ceil(y)) which in the worst case would lie on one of the sides of some rectangle and be the top-right corner of square S with side length equal to k.*
Since we know how to check if solution with value k is possible, we could use binary search to iterate through solutions and check whether or not the solution is possible and output the highest possible solution.

Union of the rectangles:

Given N axis-aligned rectangles, calculate the area of their union. We can solve it using sweep line algorithm where events are left and right edges of the rectangles. When we cross the left edge, the rectangle is added to the active set. When we cross the right edge, it is removed from the active set. We now know which rectangles are cut by the sweep line, but we actually need to know what is the length L of the sweep line parts which are intersected by these active rectangles. After processing the event, we can add distance between last two sweep line events multiplied by L to union area.
Every time we add rectangle to the active set, we actually insert vertical segment [A, B] which is representing the rectangle, in some data structure. Length of the sweep line parts which are intersected by active rectangles is length of the union of segments in the data structure.
We need a structure with three main methods addSegment(A, B), removeSegment(A, B) and getUnionOfSegments(). It can be solved using segment tree data structure. Let's create base segments first, intervals such that no horizontal edge crosses them, except in endpoints, as shown in the image 2.

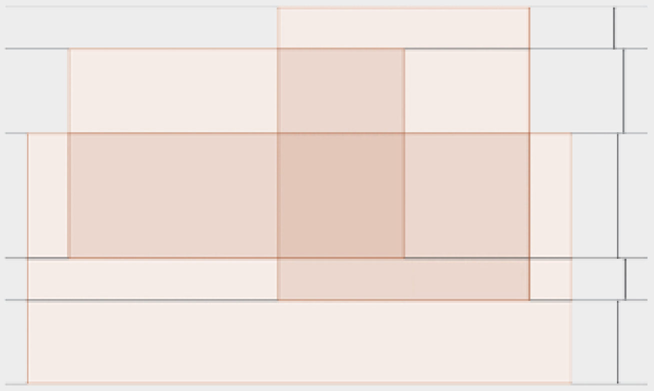


Image 2. Segments on the right are representing base segments

Each leaf node of the segment tree will represent one base segment. Every other node will represent segment of the left child merged with segment of the right child, [A, B] = merge([A, C], [C, B]). Also, for each node we will save two more variables, length of the union of all segments in the data tree which intersect with segment [A, B] and number of whole segments [A, B] in the segment tree at the moment.

Updating when segment is added in the tree is done using the following algorithm:
1. *We are starting from root node and inserting segment [A, B]*
2. *If segment [C, D] which is covered by current node equals to [A, B] then whole segment [C, D] is covered for use so length of union of segments under the current node is equal to [A, B] and number of whole segments [A, B] is increased by one*
3. *Otherwise, find intersection [C, D] of segment [A, B] with segment of left child. Call the function recursively for left child and segment [C, D]. Do the same for right child.*
   3.1. *Update length for the current node so that it equals to sum of length for both children.*

Updating when segment is removed from the tree is done using the similar algorithm, except here length is updated after number of whole segments [A, B] is equal to zero after removing. If node is leaf then it is zero, otherwise it is the sum of lengths for both children.
We should notice that number of nodes updated on each level will be at most four due to the fact that parent node represents merged segments for children nodes. When updating root node, and segment we are inserting is intersecting both left and right parts of the tree, we will visit both children nodes. If not, then we will visit appropriate child node and do the same. After that for the left part of the tree, if there exists an intersection with right child then it will be segment [C, D] which is whole segment covered by the right child thus no nodes in right child sub-tree will be visited. It is analogous for the right part of the tree.
Time complexity of the update methods is $O(log\ N)$ *with constant of four. We are iterating through each of* $2N$ *events for sweep line and calling update method for each event which gives us total complexity, for checking whether or not the solution* k *exists, of* $O(N\ log\ N)$.
Time complexity: $O(N\ log\ N\ log\ min(width, height))$, *where* $log\ min(width, height)$ *is binary search complexity for iterating through solutions.*
Memory complexity: $O(N)$.

Problem F: Scammer

Bub Bubbles is a scammer and criminal who specializes in money counterfeiting. He lives in Bubbleland. Bubbleland's currency is bubblar, famous for being impossible to counterfeit. Bub was aware of that, so he had to come up with a different technique to get bubblars. He made a machine that can copy any other currency perfectly and print large quantities of it. He could then go to the bank with the fake money and convert it to bubblars.
However, due to economic crisis, banks in Bubbleland exchange currencies in a non-standard way. The exchange rate can be both positive and negative, and they give a fixed bonus amount of money regardless of the amount of money being changed. Different currencies usually have different exchange rates, but not necessarily. Fixed bonuses are, however, always different. For the benefit of their customers who are not good with mathematics, banks do not allow exchanges where the amount of money bank has to pay to the customer is zero or negative (it would mean that the customer would only lose money). Also, the bank has a limited amount of bubblars. Customer cannot exchange amount of money in foreign currency that would result in amount of bubblars exceeding the bank limit.
Bub's machine is powerful. Bub can just enter some positive real number which represents the amount of money machine should print and the machine would calculate what would be the best currency to print the money in. The best currency is, of course, the one that would get him the highest amount of bubblars (not higher than the bank limit). But, for some amounts of money, there might exist multiple best currencies. Bub forgot about that when he made the machine, so when he enters a number for which there is more than one best currency, the machine breaks down.
Your task is to calculate how many different amounts of money would break the machine down.

Input:

The first line contains one integer N and one real number M – number of different currencies and the bank limit. Each of the next N lines contains two real numbers r_i and b_i , representing the exchange rate and fixed bonus for the ith currency. All the real numbers in the input contain exactly two fractional digits.

Output:

Output contains one integer representing the number of different amounts of money for which there exists more than one currency that Bub could convert in the bank to get the highest amount of bubblars.

Constraints:

- *$1 \leq N \leq 10^5$*
- *$5 \leq M \leq 10^6$*
- *The largest amount of money machine can print is 10^5*
- *$-10^3 \leq r_i \leq 10^3$*
- *$0 \leq b_i \leq 10^6$*
- *No test case will have more than two best currencies for any amount of money*

Example input:

```
4 6.00
2.00 3.00
-1.00 9.00
0.00 1.40
4.50 0.50
```

Example output:

```
2
```

Example explanation:

Printing and converting 1 in the first and fourth currencies gives the same highest amount of bubblars ($2.00 \cdot 1 + 3.00 = 4.50 \cdot 1 + 0.50 = 5$). Also, printing and converting 7.6 in the second and third currencies gives the same highest amount of bubblars ($-1.00 \cdot 7.6 + 9.00 = 0.00 \cdot 7.6 + 1.40 = 1.40$). Note that while printing and converting 0.2 in the third and the fourth currencies gives the same amount of bubblars, the first currency gives a higher amount of bubblars not higher than the bank limit.

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

This task is a geometry problem. It is not hard to see that the currencies are actually straight lines. Money exchange is a linear equation $y = kx + b$, where k is the exchange rate, b is the fixed bonus, x is amount of money given to bank and y is amount of bubblars. Translated to geometry, k is the slope of the line and b is the y-intercept. The bank limit is a horizontal line of the form $y = M$. The best currency for some amount of money x is the line that has the largest y such that it is less than or equal to M. Obviously, we are asked to count certain intersection points of lines. The intersection point (x, y) that should be counted is the one that satisfies $y > 0,\ y \le M,\ x > 0,\ x \le 10^5$ and the line segment linking the intersection point and the point (x, M) does not intersect any other lines.
Brute force solution is straightforward – find the intersection point of each pair of lines and test if it satisfies the necessary conditions. There are $O(N^2)$ intersection points and testing if there is a third line that intersects the corresponding line segment has complexity $O(N)$ in the worst case, so the total complexity is $O(N^3)$, which is not nearly good enough.
The appropriate solution to this problem uses divide and conquer technique. To explain the algorithm, we first define what is a $chain$. $Chain$ for a set of lines is a set of points such that a point (x, y) lying on any of the lines in the set is in the $chain$ if and only if $y > 0,\ y \le M,\ x > 0,\ x \le 10^5$ and the line segment linking the point (x, y) and the point (x, M) does not intersect any line from the set (except at the point (x, y)). $Chain$ for some set of lines is shown in the Figure 1.

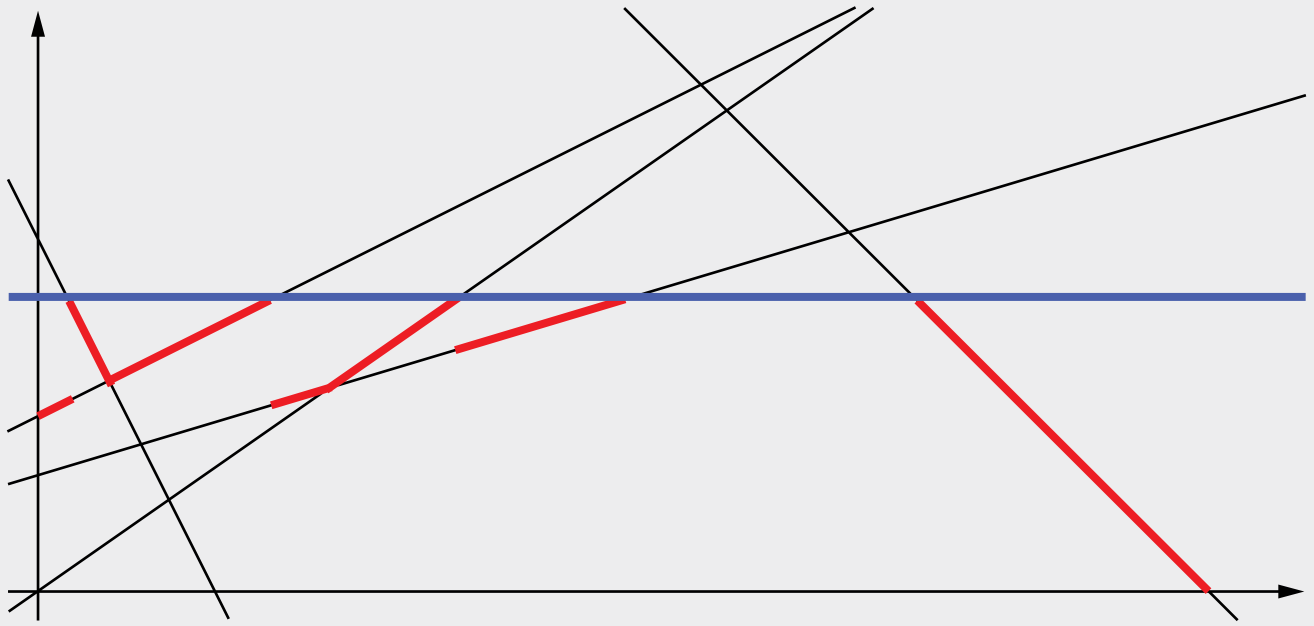


Figure 2. Bank limit is shown in blue and chain is shown in red color

Obviously, it is a set of line segments that can be sorted by the x coordinate of the start point (or end point) in a unique way. Number of intersections of two line segments, which can happen at the endpoint of one and start point of the other segment, in $chain$ is the solution to the problem for that specific set of lines.

If we are given two chains, we can merge them and get a new chain. Let's call this operation mergeChains. To merge two chains, assume that the start and end points of line segments in each chain are sorted with respect to x coordinate. Just like in mergesort, we are going keep two pointers, pointing to the current point in each chain. The pointers move parallelly – at each step, we are going to increment the pointer that points to the point with lower x coordinate. By keeping track of which chain is currently higher (larger y coordinate) and carefully processing intersections of line segments of these two chains, we can build a new chain. Notice that the start and end points of this new chain are already sorted after this operation is finished. The complexity of mergeChains operation is $O(K)$, where K is number of points in the chain. An important fact (which you can prove as an exercise) is that $K = O(N)$, where N is number of lines forming a chain. Therefore, the complexity of this operation is $O(N)$. This also means that the solution for the whole problem, the number of relevant intersections, is $O(N)$.

Now, the goal of our algorithm is to build the $chain$ for the whole set of lines. At the beginning, each line forms a $chain$ containing at most one line segment. Therefore, we start with N chains. Using mergeChains to merge first and second chain, then third and fourth, then fifth and sixth and so on, we end up with $\left\lceil\frac{N}{2}\right\rceil$ chains. Repeating this process, after $O(logN)$ steps we end up with one final chain which we can use to count the total number of relevant intersections. Each step has the complexity $O(N)$, so the total complexity of this algorithm is $O(NlogN)$.

Problem G: Unary

Unary numeral system (base-1) is a numeral system where a number K is represented with an arbitrarily chosen symbol repeated K times. The chosen symbol for this task is digit 1. Digit 0 will be used as a separator between two numbers. Writing positive integers in unary system consecutively (using 0 to separate them), you get a sequence of digits:

1011011101111011111011111110 …

By removing every second digit 1 from this sequence, you obtain a new sequence:

10~~1~~10~~1~~1~~1~~01~~1~~1~~1~~01~~1~~1~~1~~10~~1~~1~~1~~1~~1~~10 … → 10101011011101110 …

Given N, determine the Nth digit of the resulting sequence.

Input:

The first and only line of input contains one positive integer N.

Output:

Output contains one character, either '0' or '1', representing the Nth digit of the sequence.

Constraints:

- $1 \le N \le 10^{18}$

Example input:

```
6
```

Example output:

```
0
```

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

First, let's see how we determine the Nth digit of the sequence without removing every second digit 1. After writing exactly K numbers in the unary system consecutively, we are going to have exactly K zeros as seperators, while the number of ones is going to be

$$\sum_{i=1}^{K} i = \frac{K(K+1)}{2}.$$

It follows that the Kth zero in this sequence is at the position $K + \frac{K(K+1)}{2}$. Therefore, if the positive solution for K in the following quadratic equation is integer, the Nth digit is 0, otherwise it is 1.

$$K + \frac{K(K+1)}{2} = N$$

When we remove every second digit 1, the number of ones decreases by: $\left\lfloor \frac{\frac{K(K+1)}{2}}{2} \right\rfloor$.

Including the previous term in the quadratic equation, we get: $K + \frac{K(K+1)}{2} - \left\lfloor \frac{K(K+1)}{4} \right\rfloor = N$.

This equation can also be solved for K to check if the Nth digit is either 0 or 1, but it is a bit trickier. Also, the standard form for solving quadratic equation uses $sqrt$ function. This is an implementation problem because we get a term containing N in the $sqrt$ and N can be as large as 10^{18}. The precision of double data type is not good enough to perfectly represent such large numbers.
One way of solving this is to use the solution to quadratic equation as an approximation, and then to test a few integers smaller and a few integers larger than the approximated solution if they fit in the equation. If one of those integers is a solution, the Nth digit is 0, otherwise it is 1. If we ignore the complexity of the $sqrt$ function, this solution is $O(1)$.
Another solution that is safer than the previous and uses only integers is to use binary search. By using binary search on K, we can find the largest K such that: $K + \frac{K(K+1)}{2} - \left\lfloor \frac{K(K+1)}{4} \right\rfloor \leq N$.

If such K satisfies the equation $K + \frac{K(K+1)}{2} - \left\lfloor \frac{K(K+1)}{4} \right\rfloor = N$, the Nth digit is 0, otherwise it is 1. The complexity of binary search is logarithmic, which is a bit more complex than the previous solution. However, this method is stable and does not deal with floating-point arithmetic. Note that it is important to set the upper limit of the search to a number that is not much larger than $2 \cdot 10^9$, so that it it is large enough to find the solution, but not too large to cause overflow when using the 64-bit integer data types.

Problem H: Graffiti

Young artist Petr likes to draw graffiti. He just has found a long wall which consist of N consecutive empty places, numbered from 1 to N, where he can draw graffiti, and he wants to fill the whole wall with his masterpieces. Every hour he wants to draw a new graffiti on another place on the wall, and he has already decided in which order he is going to do that. When he draws a graffiti at place i on the wall, and he wants to draw next graffiti at place j, he walks from place i to the place j and he passes by all places between i and j.
Interesting about young Petr is that he loves his work, and every time he passes by his graffiti, he has to take picture of it. He doesn't take a picture of graffiti that he has just drawn.
As said, he has already decided in what order he is going to draw graffiti and now he asks you to help him to find out how many times he is going to take picture of each graffiti.

Input:

The first line of standard input contains number N, number of places on the wall. Next line contains N distinct numbers from interval $[1..N]$ where i-th number represent the place on the wall where young Petr is going to draw graffiti at i-th hour.

Output:

On the first and only line of standard output you should print N numbers where i-th number represent the number of times Petr is going to take a picture of the graffiti that is going to be drawn at i-th place on the wall.

Constraints:

- $1 \le N \le 100{,}000$

Example input:

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

Example output:

```
0 2 0 2 0
```

```
> Time and memory limit:  1.0s / 256MB
```

Solution and analysis:

If Petr walks from position i to position j, we are going to consider interval $[i, j]$. For given input we are getting $N - 1$ intervals. Now for each position i we have to find out how many intervals contains number i, but we take in consideration only intervals after the one that starts with number i.
If there weren't this last condition, if we were only asked to calculate for every position i how many intervals contain that number, we could solve it using well known data structure Segment Tree where each node would represent how many times we crossed that whole interval, but we haven't crossed whole interval represented by its father's node. After updating segment tree with each interval, the result for number i would be sum of values in nodes on the path in segment tree from root to the leaf that represents number i, let's call that sum $SegmentTreeQuery(i)$. This is known as lazy propagation in Segment Tree.
After updating segment tree with i-th interval (let's say that interval is $[l, r]$), we should find $SegmentTreeQuery(r)$ and remember that in array $before[r]$. After inserting all intervals in segment tree, results for i-th position is $SegmentTreeQuery(i) - before[i]$.

bubble cup 7

Problem A: Forest Snake

Forest Snake lives somewhere in the big forest of Rudnik and he loves to travel a lot. This summer he decided to visit the famous Micro forest which has many tourist attractions. The most popular among those attractions is Soft tree because of its interesting property. This tree contains a lot of different types of fruits and every fruit has a letter on it. Forest Snake loves palindromes and he decided to make palindrome from the letters of Soft tree. Amount of his happiness is equal to the length of palindrome he makes.
Soft tree can be represented as a connected acyclic graph with N nodes and $N - 1$ edges where each node is one fruit and some fruits are connected with edges. Forest snake can choose some node and walk to some other node, but he can visit every edge exactly once. Help him and determine the tour with maximum amount of happiness.

Input:

The first line contains number of nodes N. The second line contains string of length N where i^{th} character is written on node with index i. Each of the next $N - 1$ lines contain two integers u and v, indicating that there is an edge between nodes u and v.

Output:

Output should contain a single integer which represents the maximum amount of Forest Snake's happiness.

Constraints:

- *1 ≤ N ≤ 5,000*
- *1 ≤ u, v ≤ N*
- *All characters are lowercase English letters*

Example input:

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

Example output:

```
4
```

```
> Time and memory limit:  3s / 256MB
```

Solution and analysis:

Solution 1:

The first solution is using trie data structure.
Suppose that the longest palindrome has odd length and its middle is at the current node. Now problem consist of finding two node-disjoint chains which are adjacent to the current node and the strings they form are the same. We build trie from every subtree rooted at node adjacent to the current node. The trie is built such that it contains all strings which start at the root and end at some leaf. If two tries contain the same strings they are candidates for the solution because they are node-disjoint. We can check whether a string occurs in more than one trie by merging all tries and keeping track of how many times each node occurred. This can be done efficiently by keeping only current trie and union of all tries so far. When the current trie is built we merge it with union. The whole procedure has linear time complexity and when we pick every node as middle overall complexity is quadratic on number of nodes. Memory complexity is linear. Solution is similar for the strings with even length: for every edge build two tries from its endpoints, merge them and check if some string occurs in both tries.

Solution 2:

The second solution is using dynamic programming.
For every two nodes u and v calculate

$$dp_{u,v} = \begin{cases} 1, if\ u\ and\ v\ are\ equal \\ 0, if\ corresponding\ letters\ differ \\ dp_{f(u,v),f(v,u)} + 2\ if\ corresponding\ letters\ are\ equal\ and\ dp_{f(u,v),f(v,u)} > 0 \end{cases}$$

where $f(u,v)$ is the first node on path from u to v. Calculating all values of f has quadratic time complexity, and all dp values can be calculated in quadratic time using memoization. Overall, this solution has quadratic memory and time complexity.

Summary:

Although the first solution has better memory complexity the second one has smaller time constant and it is far easier to implement.

Problem B: Calculator

Your task is to implement a special calculator. Like an ordinary calculator, the user can type in a mathematical expression, but that is where similarities end. This calculator lacks some features compared to ordinary calculators - for example, it cannot do subtraction or division. But it can also do some things ordinary calculators are not able to do: after the user types in an expression, she can choose to calculate only a part of it between a given starting and ending point, and she can do it as many times as she likes for the same expression. The calculator supports the following elements of input:

- Non-negative integers
- Arithmetical operators: +, *
- Brackets: (,)

Input:

The first line contains one integer p – the number of elements in the expression. The second line contains the expression E, comprised of elements listed in the text above. Each element of the expression is separated by a single space character on both sides. The third line contains one integer N – the number of requests for calculation on E. Each of the next N lines contain 2 integers, representing the starting and ending element in the expression that should be calculated. Each number, operator or bracket is a single element. It's not important how many keys presses the user needs to make to obtain it).

Output:

The output contains N lines – in every line there should be one integer, representing the result of the calculation. Since the numbers can get very large, the output should be calculated modulo $10^9 + 7$.

Constraints:

- *$1 \le p \le 1{,}000{,}000$*
- *For each integer k in the expression E, $1 \le k \le 10{,}000$*
- *$1 \le N \le 100{,}000$.*
- *$1 \le ai \le bi \le p$, for each $i \in \{1 \ldots n\}$*
- *It is guaranteed that E will be a valid mathematical expression.*
- *It is guaranteed that all subexpressions of E that need to be calculated will be valid.*
- *It is guaranteed that all subexpressions of E that need to be calculated will be valid. None of the subexpressions will begin or end with a + or * sign, and all brackets will be properly matched.*

Example input:

```
17
99 + ( 25 * ( 3 + 7 ) * 10 + 50 ) * 2
2
6 14
3 17
```

Example output:

```
150
5100
```

```
> Time and memory limit:  2s / 128MB
```

Solution and analysis:

The obvious algorithm that solves the problem is the following:

For each query:

1. *Extract the subexpression that needs to be calculated*
2. *Calculate the value of the subexpression*

Step 2 is not completely trivial and we won't go into details on how exactly to implement it, but it should be clear that it requires $O(p)$time, and the solution as a whole then requires $O(N \cdot p)$ time in the worst case, which is clearly not fast enough to finish under the time limit.
Intuitively, we should be able to do this faster because the algorithm described above computes certain expression fragments over and over again. But how do we make use of this insight?
Let's first solve an easier subset of the problem – we'll assume that our expression does not contain any brackets. This means that the expression can be written as a sum of products:

$$E = a_{11} \cdot a_{12} \cdot \ldots \cdot a_{1k_1} + a_{21} \cdot \ldots \cdot a_{2k_2} + \cdots + a_{l1} \cdot a_{l2} \cdot \ldots \cdot a_{lk_l}$$

When the expression comes in but before we answer any queries, we can precompute some things.
For each number a_{ij} we will keep track of the following data:

1. *i and j*
2. *$PL_{ij} = a_{i1} \cdot \ldots \cdot a_{ij}$ – the left part of the product a_{ij} belongs to (including a_{ij})*
3. *$PR_{ij} = a_{ij} \cdot \ldots \cdot a_{ik_i}$ – the right part of the product a_{ij} belongs to (including a_{ij})*
4. *$SL_{ij} = a_{11} \cdot a_{12} \cdot \ldots \cdot a_{1i_1} + \cdots + a_{(i-1)1} \cdot a_{(i-1)2} \cdot \ldots \cdot a_{(i-1)k_{i-1}}$ – the left part of the total sum, up to the product a_{ij} belongs to*
5. *$SR_{ij} = a_{(i+1)1} \cdot a_{(i+1)2} \cdot \ldots \cdot a_{(i+1)k_{i+1}} + a_{l1} \cdot a_{l2} \cdot \ldots \cdot a_{lk_l}$ – the right part of the total sum, starting with the product after the product a_{ij} belongs to.*

We will also calculate the value of the entire expression E (which is just $SL + PL$ for the last element in the expression).
How does this help us? We get a calculation request for the subexpression between elements a_{ij} and a_{pq}. Assuming that $i \neq p$, the formula

$$E - SL_{pq} - SR_{ij} + PR_{ij} + PL_{pq}$$

gives us the correct result. This is easy to verify: $E - SL_{pq} - SR_{ij}$ calculates the sum of all products fully contained in the subexpression, and $PR_{ij} + PL_{pq}$ adds the parts of the two products split by a_{ij} and a_{pq} respectively.
Since this is not correct if $i = p$, we need a different formula for that case:

$$PR_{ij} \cdot a_{pq} \Big/ PR_{pq}$$

Again, it is not difficult to see why this is correct.

How fast is this solution? The precompute step can be performed in $O(p)$ time. We can go through the expression from left to right, saving cumulative sums and products and arrays as we go along to get PL and SL. For PR and SR, we do the same thing from right to left. Answering queries can now be done in constant time – we only need to do a couple of array lookups and arithmetic operations. (Note: not all constant-time operations are created equal. The division modulo 10^9+7 needs to be implemented through exponentiation by squaring. You could write a naïve implementation that makes 10^9+7 steps, which is still theoretically $O(1)$ but the time limit checker will not be convinced by that argument ☺). The overall time complexity of the solution is $O(p+N)$.

Let us now solve the full problem. The presence of brackets changes the complete structure of the task, right? Well, not really. Notice that, for each pair of brackets, it is impossible to construct a valid query that contains only one of the brackets and not the other. This means that each query has to either fully reside within the pair of brackets, or fully cover the brackets and everything inside them. This immediately gives us a way to reduce the problem to the solved case:

1. *If the subexpression is fully contained within the brackets, we can safely ignore everything outside the brackets*
2. *If the subexpression generated by the query covers the brackets, we can replace the contents of the brackets with a single number – the value of the expression delimited by the brackets*
3.

Let's do this on the example from the problem statement:

```
99 + ( 25 * ( 3 + 7 ) * 10 + 50 ) * 2
6 14
```

The subexpression is completely inside the outer pair of brackets, so we ignore everything else:

```
25 * ( 3 + 7 ) * 10 + 50
```

The remaining pair of brackets is covered by the subexpression, so we calculate it:

```
25 * 10 * 10 + 50
```

Now we have reduced the problem to the already solved case and we can solve it using the same algorithm. The only remaining question is how to do the bracket handling work without impacting the algorithmic complexity. A straightforward way to do this is with the help of a stack structure:

1. *Set the number of encountered brackets so far to $b=0$.*
2. *Set the values we want to keep track of (current accumulated sum, current product, current values of i and j) to their initial values.*
3. *Go through the expression from left to right, keeping track of cumulative sums and products and saving values of PL, SL, i, j, b for each element to an array.*
    a. *If the current element is an open bracket, take the current state (current sum, current product, current values of i and j), put it on the stack, then reinitialize the state. Increase the number of encountered brackets by 1.*
    b. *If the current element is a closed bracket, record the value of the whole expression SB within the bracket (which should already be calculated as the accumulated sum within the bracket), then pop the previous state from the stack. Keep calculating PL and SL for the remaining elements, as if everything within the brackets was a single element with the value SB.*
4. *Repeat steps 1-3, going from right to left this time and filling in values of PR and SR.*

The formulas for answering queries remain the same as in the previous case. The only difference is that, due to i and j being "bracket-relative" now, we also have to check whether $b_{ij} = b_{pq}$ to know when to use the first formula and when the second. The complexity of this algorithm is the same as the complexity of the algorithm described earlier. We still keep track of just one set of values for each expression element. The only additional operations during the precompute step are related to handling of the stack, but there can be at most $O(p)$ of them. Answering queries is still done in $O(1)$ time per query.

Notes:

1. *The memory limit is 128MB, so there is more than enough memory to store all data. The memory complexity of the solution is $O(p)$, but some amount of care has to be taken to not let the constant factor get out of hand.*
2. *All inputs and outputs can be handled as 32-bit integers, but using 32-bit numbers everywhere can overflow during multiplication. So you either need to be careful and cast to 64-bit when it's needed, or work with 64-bit numbers all the way and try not to hit the memory limit.*

Problem C: ForEST

This year, ForEST (traditional festival for inhabitants of forests) is organized in Forest Snake's forest. Best beer manufacturers present their products and top jungle musicians have live performances in the middle of wood. Traditionally, monkeys take care of security, and this year they decided to organize ForEST slightly different – there will be multiple fan pits in the forest. The first pit is nearest to the stage, the second is immediately after the first and so on… Also it is known that first pit is inside the second, which is inside the third… Fan pits are separated by long straight ribbons which are fixed to some trees in the forest. Ticket price for the first fan pit is N coins (forest coin is official currency in every forest in the world), for the second $N - 1$,… For i^{th} fan pit price is $N - i + 1$. Being outside of any fan pit is free.

Position of every tree in forest can be represented with two integer coordinates, and every security ribbon can be represented as straight line segment starting at one tree and ending at some other tree. So, one fan pit is actually convex polygon with trees as vertices and ribbons as edges. Forest Snake is interested in following problem: given coordinates of some animals in the forest, what price each of them paid for being at that place during ForEST?

Input:

The first line contains one integer N – the number of fan pits. Each of the next N lines start with the number of nodes of fan pit M, followed by M pairs of integers x and y – coordinates of nodes. The next line contains number Q – number of Snake's question. Each of the next Q lines contain pair of integers x and y representing the position of animals.

Output:

For each of the Q animals output a single integer per line which is equal to its ticket price.

Constraints:

- *There is at least one fan pit*
- *Sum of number of nodes of all polygons doesn't exceed* $3 \cdot 10^5$
- *Every fan pit has at least three nodes*
- $1 \leq Q \leq 3 \cdot 10^5$
- $-10^9 \leq x, y \leq 10^9$
- *It is guaranteed that the first polygon is inside the second, the second inside the third… N - 1th inside the Nth*
- *All polygons are convex and there is no animal standing on any ribbon or tree*
- *Nodes of the polygons are given in counterclockwise order*

Example input:

```
2
3 -3 2 2 -3 3 5
4 10 10 -10 10 -10 -10 10 -10
3
0 0
100 100
6 3
```

Example output:

```
2
0
1
```

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

Solution and analysis:

For every convex polygon make upper and lower chain, such that upper chain contains all points on clockwise path from the leftmost point of the polygon to the rightmost one. Similar, lower chain contains all points on counterclockwise path from the leftmost to the rightmost point. Make array of all points from input (points from queries and polygons) and sort them by x coordinate ascending. Also, make two stacks, one for upper chains and the other for lower chains. Stack should keep only index of the last point visited so far. Traverse the array and check if current point is from polygon or from some query:

1. *If the point is from polygon:*
   a. *If it's the leftmost point of some chain push its polygon on stack*
   b. *If it's the rightmost point of some chain pop its polygon from stack*
   c. *Otherwise make current point be the last point of its chain*

2. *If the point is query point:*
   a. *Check if it's under all upper chains and then do binary search on lower chains, otherwise do the binary search on upper chains. Binary search finds between which two chains the current point is located, and from that we can easily calculate the number of polygons in which the point is located inside of.*

Sorting all the points requires $O(K \log K)$ where K is total number of points including those on polygon and from queries. Every binary search requires $O(\log N)$, which leads to total complexity $O(K \log K + Q \log N)$.

Problem D: Search

Alice is looking for a job and she has heard that one software company is hiring experts in run-length encoding (RLE). It is a very simple form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. For example, you can compress:

`“WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWW”`
into: `“12W1B12W4B14W”`

`“11522666”`
into `“21152236”`
Alice is playing around with encoding of positive integers. One unit of data is a single digit, so she encodes each sequence of the same digit as that digit and its count.
We can define a function Alice uses for encoding as $RLE(A) = B$, where A and B are positive integers. However, since this is a compression algorithm, Alice doesn’t consider an encoding valid if B has more digits than A.
Alice encodes a number multiple times in a row, until the encoding is not valid. For example, number 333 can be encoded two times in a row before the encoding gets invalid: $RLE(333) = 33$; $RLE(33) = 23$; $RLE(23) = 1213$ – number 1213 has more digits than number 23 so the last encoding is not valid.
After playing with RLE she found one interesting number - 22. Number 22 can be encoded infinitely many times, because $RLE(22) = 22$.
Now she wants to find a positive integer different than 22, with no more than 100 digits, that can be encoded at least 5 times in a row before the encoding gets invalid. Help her!

Input:

There is no input for this problem.

Output:

Output a single positive integer Alice is looking for.

Constraints:

- *Number of digits of the positive integer in the output must be ≤ 100*
- *Output must not be 22*

Example input:

```
No example input
```

Example output:

```
No example output
```

```
> Time and memory limit:  0.1s / 64 MB
```

Solution and analysis:

The idea is to find the number which can be encoded 4 *times, and then manually construct the number which can be encoded into that number (so ultimately it can be encoded* 5 *times). To find the first number which can be encoded* 4 *times, we can use brute force solution. Straight forward brute force solution can be optimized with some heuristics. For example, we don't need to consider numbers that contain digits* 5, 6, 7, 8 *and* 9 *in itself, since they will not satisfy both condition to have less than* 100 *digits, and to be encodable* 5 *times.*
Smallest number different than 22 *that can be encoded 4 times is* 2233322211. *Now, number* 2233333333333333333333333333333333322222222222222222222221 *(two times* 2, *thirty-three times* 3, *twenty-two times* 2 *and one* 1*) can be encoded* 5 *times. Since this number has less than* 100 *digits, it is one of the solutions. Another solution can be obtained from number* 22333222112, *and it will have less digits,* 31 *total. Number is* $[2x2,\ 3x3,\ 3x2, 22x1, 1x2]$.

Problem E: Cycles

You are given a graph G with two spanning trees that share no edges. A cycle in G is a connected subgraph whose vertices have degree 2. Your task is to find a collection of cycles in G such that every edge is in precisely two of your cycles. Such a collection will always exist.

Input:

The first line contains two integers separated by an empty space: n – the number of vertices and m – the number of edges. Every of the next n lines contains 3 integers u, v, l, separated with empty spaces, which represent the edge between vertices u and v. The remaining integer l can take values $0, 1, 2$. Value 0 means that it is not in either of the spanning trees, 1 means that it is in the first one, and 2 that it is in the second tree.

Output:

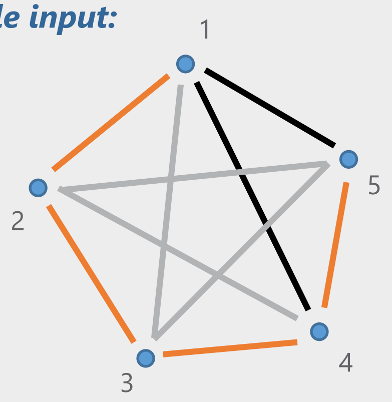
The first line of the output should contain a single integer C – the number of cycles you produced. Every line that follows should describe one of the C cycles and should start by integer m which is the size of the cycle and should then contain m integers that specify the cycle (so that the edges are between the 1st and 2ndvertex, 2nd and 3rd vertex, etc. , and between mth and 1st). Any collection of cycles that contains every edge precisely twice is considered to be a valid solution.

Constraints:

- *$1 \le n \le 500{,}000$, $1 \le m \le 1{,}000{,}000$, $2n - 2 \le m$*
- *There are no loops or repeated edges*

Example input:

```
5 10
1 2 1
2 3 1
3 4 1
4 5 1
1 5 0
1 3 2
3 5 2
5 2 2
2 4 2
1 4 0
```



Example output:

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

```
> Time and memory limit:  10s / 256MB
```

Solution and analysis:

Since we're already given spanning trees in the graph and we're looking for cycles, it is natural to recall that given a tree T and an edge e not in the tree, there is a unique path in T that joins the endpoints of e. In turn, this gives a cycle that contains edge e. Write T_1 for the first spanning tree that is given, and T_2 for the second one. By applying this procedure to edges not in T_1 and the tree T_1, we get a collection of cycles that contain each edge not in T_1 precisely once, while the edges of T_1 can possibly be contained in multiple cycles. If an edge is contained in more than two cycles, this is certainly a problem, given our goal. The key observation now is that we can actually turn the current collection of cycles into a useful one by applying the symmetric difference to the cycles. In other words, we pick the edges in an odd number of cycles in the current collection. It is easy to see that this gives us an even subgraph H (i.e. all vertices have even degrees), where we may perform Euler tour to produce a new collection of cycles, that contain each edge of H precisely once. Crucially, the new collection of cycles contains each edge not in T_1 precisely once, while the edges of T_1 are contained at most once.
We can repeat the same procedure to tree T_2. As the trees have no common edges, by taking a union of the two collections produced so far, we obtain a collection of cycles that contains each edge of G once or twice. Finally, we observe that taking all edges that appear precisely once gives another even subgraph and performing Euler tour once more and adding these cycles, produces a solution – a collection of cycles containing each edge precisely twice.
As far as the implementation is concerned, there is an important point – how to find the cycles coming from a spanning tree efficiently? Recall that we actually do not need all the cycles given by paths along the tree, but actually only the resulting even subgraph given by symmetric difference. For this it is actually sufficient to write at each node of a tree the number of non-tree edges that are adjacent to it and then to pick tree edges whose subtrees have odd sum of numbers at nodes. Thus, a simple tree traversal suffices.

To sum up, the algorithm looks as follows:

- *For T_1, for each node v, write the number of non-tree edges adjacent to v, and then perform a DFS traversal that sums the numbers in subtrees and chooses the edges with odd subtree sum.*
- *Take the edges chosen above and add edges not in T_1 to form an even subgraph.*
- *Perform Euler tour on this subgraph and add the resulting cycles to solution.*
- *Repeat all the steps above to T_2*
- *Finally, form an even subgraph of edges appearing only once the cycles chosen so far.*
- *Perform another Euler tour to complete the solution.*
- *The algorithm has linear time and memory complexity.*

Problem F: Compression

A software company is making tools for data compression. One group of engineers is working on algorithms for compression of really long textual data. Currently, they are implementing some variations of compression algorithm called run-length encoding. Run-length encoding is a technique where consecutive runs (sequences) of same data are stored as a run value and count. In this case specifically, algorithm takes a string S as an input and compresses it into a new string C_s using run-length encoding. C_s is of the form

$$< run_1 >< count_1 >< run_2 >< count_2 > \cdots < run_k >< count_k >$$

where $< run_i >$ is a non-empty string value and $< count_i >$ is a positive integer value for each $i = 1..k$. Decompressing C_s back to S works by repeating value of $< run_i >$ exactly $< count_i >$ times for each $i = 1..k$. Obviously, C_s may not be unique because it might be possible to split S into runs in many different ways, so engineers are experimenting with different approaches of splitting S into runs. All of these approaches are optimized for compression speed and low memory consumption of the algorithm, and not the compression efficiency, because in reality input data can be several terabytes large and it is more important that compression is fast and does not take a lot of resources. To measure compression efficiency, engineers will use smaller inputs and compare their approaches of splitting S into runs to the optimal way of splitting S into runs – one that results in C_s of minimum length. Help them by writing a program that will calculate the minimum length of C_s.

Input:

The first line contains one integer N – length of string S. The second line contains the string S.

Output:

Output contains one integer – minimum length of string C_s.

Constraints:

- *1 ≤ N ≤ 3,000*

All letters of S are lowercase letters of English alphabet

Example input:

```
12
aaaababababc
```

Example output:

```
7
```

Example explanation:

Compressed string of the smallest length is $a3ab4c1$. Some other possibilities for C_s are $a4ba3bc1$, $aa2ba2babc1$, $aaaababababc1$ etc. but none of them have length less than 7.

```
> Time and memory limit:  1.5s / 128MB
```

Solution and analysis:

We will start by describing a dynamic programming part of the solution.
Let $DP[i]$ represent the minimum compression length for the string $S[1..i]$ (substring of S starting at the position 1 and ending at the position i). The solution is then $DP[N]$.
So, how do we calculate $DP[i]$, for $i = 1..N$? We initialize by setting $DP[0] = 0$. When we are at the position i, we take any substring of S ending at the position i as a run. There are exactly i such substrings: $S[i..i], S[i-1..i], S[i-2..i], \ldots, S[1..i]$. For each run, we will try to repeat it k times, where $k = 1..M$, and M is $\left\lfloor \frac{i}{length(run)} \right\rfloor$.
Then, if the substring $S[i - k \cdot length(run) + 1..i]$ is equal to the substring $S[i - length(run) + 1..i]$ repeated exactly k times, one possible value for $DP[i]$ would be $DP[i - k \cdot length(run)] + length(run) + number_{of_{digits(k)}}$. $DP[i]$ obviously takes a minimum of all such values.
We do not need to check the whole first substring – if the equality of substrings holds for $k-1$ and $S[i - k \cdot length(run) + 1..i - (k-1) \cdot length(run) + 1]$ is equal to the $S[i - length(run) + 1..i]$, then it holds for k also. Actually, we can also see that when we get to k that doesn't satisfy the equality, we can stop for this run, because the condition would not be satisfied for any number larger than k either.
Let's take a string $S =' aabab'$ as an example. For $i = 5$, the runs can be $'b'$, $'ab'$, $'bab'$, $'abab'$ and 'aabab'. Taking 'b' as a run and $k = 1$, we take $DP[4] + length('b') + number_{of_{digits(1)}}$ as one possible value for $DP[5]$. For $k = 2$, we see that substring $S[4..4]$ is not equal to $'b'$ so we can stop. For $'ab'$, one possible value for $DP[5]$ is when $k = 1$, and the value is $DP[3] + length('ab') + number_{of_{digits(1)}}$. When $k = 2$, $S[2..3] = S[4..5]$ so possible value for $DP[5]$ is $DP[1] + length('ab') + number_{of_{digits(2)}}$. We cannot try with $k = 3$ because of length of S. Similarly, we can check runs $'bab'$, $'abab'$ and $'aabab'$, with k only equal to 1, due to their length.
We haven't mentioned the complexity of the algorithm yet, but so far we can see that this doesn't seem efficient enough in the worst case. We can speed this up by precomputing if the substring $S[i - j..i] = S[i + 1..i + j + 1]$ for all possible values of i and j and storing the results in a matrix of size $O(N^2)$. This will help us improve the dynamic programming part of the solution by avoiding checking if two substrings are equal character by character every time, and instead have a $O(1)$ check. We can precompute this in several ways, such as using hashing or trie data structure. When using hashing, we can compute the hash values of all substrings in $O(N^2)$, using rolling hash, as in Rabin-Karp algorithm.
Let's see what the total complexity of the solution is. For each index i, we check all the runs ending at i. This has time complexity $O(N^2)$. Then, for each run, we try to repeat it k number of times. In the worst case, where the characters of the input string are all the same, we would need to try for each $k = 1..\left\lfloor \frac{i}{length(run)} \right\rfloor$. For each k, we need to check if the substrings are the same, and without any precomputation this has time complexity of $O(length(run))$, so the total complexity of the algorithm in that case would be $O(N^3)$. However, with $O(1)$ check, the total complexity is $O(H_N N^2)$, where H_N is N^{th} harmonic number, and $H_N = \sum_{i=1}^{N} \frac{1}{i}$. Harmonic series grows very slowly, so for $N = 3000$, $H_N \leq 10$. With the memory complexity of $O(N^2)$, this algorithm is efficient enough, given the constraints in this task.

Problem G: Sticks

You are given an array of N sticks. First stick is at the position 1, second at the position 2, etc. Last one is at the position N. You can pair two sticks if they are not paired already and there are exactly two sticks between them, either paired or unpaired. When you pair two sticks, you move one to the position of other – you can choose which one you move. Only the stick that is moved changes its position, all the other sticks remain at their previous position.
Given N, determine if it is possible to pair all the sticks and if it is, output the pairing process. If there are multiple solutions, output any solution.

Input:

The first and only line contains one integer N – number of sticks.

Output:

If it is not possible to pair all the sticks, output -1. If it is possible, output the pairing process, one pairing step per line. Format of each step is "a b" (without quotation marks), which means that the stick at the position a is paired with the stick at the position b and that the stick at a is moved to b ($1 \leq a, b \leq N$).

Constraints:

- $1 \leq N \leq 50$

Example input:

```
5
```

Example output:

```
-1
```

Example explanation:

It is not possible to pair all the sticks. We can pair 4 sticks in different ways, but one stick would end up without a pair. One way to pair 4 out of those 5 sticks would be

```
1 4
5 3
```

After the first step, stick at the position 1 is paired with the stick at the position 4 and moved to that position. Notice that there are now two sticks between position 3 and 5. In the next step, stick at the position 5 is paired with the stick at the position 3 and moved to that position. Stick at the position 2 is left without a pair.

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

Solution and analysis:

Let's make a few observations first.
If N is odd, it is obvious that there is no solution because at least one stick would end up without a pair.
If N is even, we can reduce the problem to $N-2$ sticks by pairing the fourth and first stick and moving the fourth to the first position. We are then left with two sticks less, with the two paired sticks being at the beginning of the array and having no effect in the further pairing process. That means that if the pairing of all sticks is possible when N equals some even number K, it is also possible when N equals any even number larger than K. Now we just need to find minimum such number.
We can see that it is impossible to pair all the sticks when N equals 2, 4 or 6 by manually trying all the possibilities (there aren't many of them). We can also try to solve the problem when $N = 8$ manually. It is not hard to come up with the full pairing process for $N = 8$, so we can conclude that there is a solution when N is even and $N \geq 8$.
One solution when $N = 8$ is to first pair 5^{th} and 2^{nd} stick, moving 5^{th} to the 2^{nd} position. Then, we can pair 3^{rd} and 7^{th} stick, moving the 3^{rd} to 7^{th} position, since there is exactly two unpaired sticks between them after the first step. Third step should be pairing and moving the 8^{th} stick to the 6^{th} position (there is exactly one pair between them from the second step). Final step is pairing the 1^{st} and 4^{th} stick.
So, the final algorithm is to pair the first and the fourth unpaired stick from the beginning of the array in each step, by moving the fourth unpaired stick to the position of the first, until we are left with exactly 8 unpaired sticks. Then those 8 sticks can be paired as previously described. All of this can be done in linear time, so the final algorithm has complexity $O(N)$.

Problem H: Vectors

A set of m vectors $\{v_1, v_2, \ldots, v_m\}$ in $\mathbb{R}^d$ (the set of d-tuples of real numbers) is said to be *linearly independent* if the only reals $\lambda_1, \lambda_2, \ldots, \lambda_m$ that satisfy $\lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_m v_m = 0$ are $\lambda_1 = \lambda_2 = \cdots = \lambda_m = 0$. For example, in $\mathbb{R}^2$ the set of vectors $\{\binom{1}{0}, \binom{0}{1}\}$ is linearly independent. However, $\{\binom{1}{0}, \binom{0}{1}, \binom{1}{1}\}$ is not since $1\binom{1}{0} + 1\binom{0}{1} + (-1)\binom{1}{1} = \binom{0}{0}$.
In this task, you are given n vectors in $\mathbb{R}^d$, and every vector has some weight. Your job is to find a linearly independent set of vectors with maximal sum of weights.

Input:

The first line contains two integers d and n. The next n lines contain $d + 1$ integers each, separated with one empty space between any two integers. The first d numbers in the line $i + 1$ are coordinates of the i^{th} vector, and the last number is its weight.

Output:

The output should consist a single integer: the sum of weights of vectors in your set.

Constraints:

- *1 ≤ d ≤ 200*
- *1 ≤ n ≤ 500*
- *The coordinates of the vectors are integers in the range [-10^3, 10^3]*
- *The weights of the vectors are integers in the range [-10^6, 10^6]*

Example input:

```
4 4
1 0 0 0 30
0 0 1 0 30
1 0 1 0 100
0 0 0 1 1
```

Example output:

```
131
```

```
> Time and memory limit: 0.5s / 16MB
```

Solution and analysis:

We claim that the greedy algorithm works, i.e. that it suffices to sort the vectors, start with an empty set of vectors and then at each step add the heaviest vector to the set if the set remains linearly independent.
Observe the following basic lemma from linear algebra.
Lemma 1. Suppose that vectors $\{v_1, v_2, \dots, v_n\}$ *are linearly independent, and that* $\{u_1, u_2, \dots, u_{n+1}\}$ *are also linearly independent. Then we can find some* k *such that* $\{v_1, v_2, \dots, v_n,\ u_k\}$ *is also linearly independent.*
We postpone the proof for later, the fact above should at least be intuitively obvious.
Proof that the greedy algorithm is correct. We prove by induction on $s \geq 1$ *that the greedy algorithm produces a set of* s *vectors of maximal weight. For* $s = 1$ *this is clear, as we choose a non-zero vector of maximal weight.*
Suppose now that $s \geq 2$ *and that the statement holds for smaller values of* s. *Suppose however that the statement fails for* s, *that is, the greedy algorithm finds* $\{v_1, v_2, \dots, v_s\}$, *but* $\{u_1, u_2, \dots, u_s\}$ *has higher weight. Still, by induction hypothesis,* $\{v_1, v_2, \dots, v_{s-1}\}$ *is optimal for* $s - 1$. *This means that any subset of* $s - 1$ *elements of* $\{u_1, u_2, \dots, u_s\}$ *has weight at most that of* $\{v_1, v_2, \dots, v_{s-1}\}$, *and in particular,* v_s *has smaller weight than any* u_i. *By Lemma 1 applied to* $\{v_1, v_2, \dots, v_{s-1}\}$ *and*$\{u_1, u_2, \dots, u_s\}$, *we have some* k *such that* $\{v_1, v_2, \dots, v_{s-1}, u_k\}$ *is linearly independent, and* u_k*has higher weight than* v_s, *so it would have been added to our set before* v_s, *which is contradiction. This finishes the proof that the algorithm works.* □
Proof of Lemma 1. Suppose contrary, so every u_k *can be written is in the span* V *of* $\{v_1, v_2, \dots, v_n\}$. *But* $\{u_1, u_2, \dots, u_{n+1}\}$ *are linearly independent in* V *which contradicts Steinitz exchange lemma. (See* *http://en.wikipedia.org/wiki/Steinitz_exchange_lemma*). □
Finally, we may observe that a simple way to implement the algorithm above is to sort the vectors by weights and put them as rows in a matrix in the sorted order. Then one pass of Gaussian elimination gives the solution, by picking the non-zero rows. This gives an algorithm of time complexity $O(dn^2)$ *and memory complexity* $O(dn)$.

Problem I: Queries on an array

You are given an array a of N elements. Array a is 0-indexed. There are two types of queries that you should perform on the array.

- $INVERT\ i\ j\ k$: Invert the k^{th} bit on each element in the range $[a_i, a_j]$
- $SUM\ i\ j$: Output the sum of the elements in the range $[a_i, a_j]$

Note that 0^{th} bit is the least significant bit and 31^{st} bit is the most significant bit.

Input:

The first line contains one integer N – size of the array. Second line contains N integers that are initial values of the elements in the array. Third line contains one integer Q – number of the queries. Following Q lines contain one query per line.

Output:

Output contains Q_{sum} lines, where Q_{sum} represents the number of SUM queries, and each line contains the answer to the corresponding query.

Constraints:

- *1 ≤ N ≤ 100,000*
- *1 ≤ Q ≤ 100,000*
- *0 ≤ i ≤ j ≤ N-1*
- *0 ≤ k ≤ 31*

Elements of the array are 32 bit unsigned integers.

Example input:

```
4
1 2 3 1
5
SUM 0 2
INVERT 0 2 0
SUM 0 2
INVERT 3 3 10
SUM 3 3
```

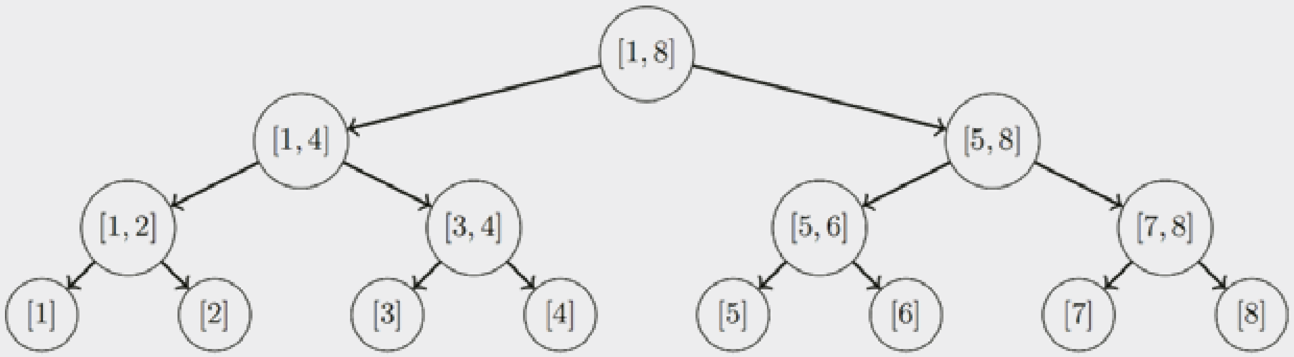
Example output:

```
6
5
1025
```

```
> Time and memory limit: 4s / 16MB
```

Solution and analysis:

Here we are faced with an interesting and relatively straightforward problem, assuming familiarity with the required data structure. The obvious brute-force solution would involve manually updating each array element upon an inversion command and performing the summing up in a similar fashion. This has a worst-case complexity of $O(N)$ per operation, which is too slow for the given query time. Clearly, we need to utilize a more clever approach here.
*A common pattern of thought when up against a problem involving dynamically updating an array and then querying over its intervals is to try to get the complexity down to $O(\log N)$; one of the most common data structures to consider for achieving this is called a **segment tree**, which we can use for this problem as well. A segment tree is a binary tree constructed over an array (a more general version is constructed over a set of points on the real number line) where each node is "responsible" for a certain subinterval of the array – the root is responsible for the entire array, leaves are responsible for each of the individual elements, while nodes in between are responsible for the union of the intervals held responsible by their children – the figure below represents an example layout of the structure for an array of size 8, with the intervals noted in each node:*



*The operations of updating and querying a single element are clearly of logarithmic time complexity, as they require recursively descending down the tree, halving the interval being considered in each step. Querying over a range is also of logarithmic cost – if we query by aggregating the values stored in the minimal set of nodes covering the entire range; the proof that there will always be $O(\log N)$ nodes in this minimal set is left as an exercise to the reader. Updating a range requires us to be more clever; we can only update the minimal set of nodes covering the range – but as we might need to propagate this to the nodes deeper in the tree, we use a technique called **lazy propagation,** where we store pending updates in nodes and propagate them to their children whenever they are accessed.*
*To see how a segment tree could be used for solving this problem, let us consider an easier variant: assume we are only dealing with 1-bit values (as in, all members of the array are lesser than 2). This problem can easily be solved by using a segment tree, having each node store the **sum** of the array values in the interval it's responsible for (i.e. each node's value is equal to the sum of the values of its children). Once we have a structure like this, querying for the sum of a range simply involves summing over the nodes in the minimal set as discussed in the previous paragraph. Inversions are first performed by "inverting" all the nodes in the minimal set, then propagating the operation to their children when necessary, and so on. "Inverting" a node is simple: if a node is responsible for an interval of size l, and it had stored a sum of k previously, then after inverting that interval, the sum stored will become $l - k$ (as all the 1s in the interval become 0s and vice versa). Hence, we have successfully reached a logarithmic-time solution per operation for this version of the problem.*

To expand this to d-bit values, we just need to construct d segment trees of 1-bit values as outlined in the previous paragraph, each responsible for an individual bit of the integers. An inversion operation involves updating the appropriate segment tree (given to us in the problem with the input parameter k). When summing up, we can just add up all the powers individually with separate summations on each segment tree:

$$SUM(l,r) = \sum_{i=0}^{d-1} segTree[i].SUM(l,r) * 2^i$$

This gives us an overall time complexity of $O(\log N)$ per invert operation and $O(d \log N)$ per summation – overall the worst-case time complexity of the algorithm (when only performing sum queries) of $O(Q \cdot d \log N)$ and the space complexity is $O(dN)$. As in this version of the problem we have fixed $d = 32$, we can ignore it from our analysis, giving us the required $O(Q \log N)$ behaviour.
Another thing to note is that the query result may overflow a 32-bit integer, and as such, using a 64-bit type (long long in C++/Int64 in Pascal) is necessary to completely solve this problem.

Problem A: Fibonotci

Statement:

Fibonotci sequence is an integer recursive sequence defined by the recurrence relation

$$F_n = c_{n-1} \cdot F_{n-1} + c_{n-2} \cdot F_{n-2}$$

with:

$$F_0 = 0,\ F_1 = 1\,.$$

Sequence c is infinite and *almost cyclic* sequence with a cycle of length N. A sequence s is *almost cyclic* with a cycle of length N if $s_i = s_{i \bmod N}$, for $i \geq N$, except for a finite number of values s_i, for which $s_i \neq s_{i \bmod N}$ $(i \geq N)$.
Following is an example of an *almost cyclic* sequence with a cycle of length 4.

$$s = (5, 3, 8, 11, 5, 3, 7, 11, 5, 3, 8, 11, \dots)$$

Notice that the only value of s for which the equality $s_i = s_{i \bmod 4}$ does not hold is s_6 ($s_6 = 7$ and $s_2 = 8$).
You are given $c_0,\ c_1, \dots, c_{N-1}$ and all the values of sequence c for which $c_i \neq c_{i \bmod N}$ $(i \geq N)$.
Find $F_K \bmod P$.

Input:

The first line contains two numbers K and P. The second line contains a single number N. The third line contains N numbers separated by spaces, that represent the first N numbers of the sequence c. The fourth line contains a single number M, the number of values of sequence c for which $c_i \neq c_{i \bmod N}$. Each of the following M lines contain two numbers j and v, indicating that $c_j \neq c_{j \bmod N}$ and $c_j = v$.

Output:

Output should contain a single integer equal to $F_K \bmod P$.

Constraints:

- *$1 \leq N, M \leq 50{,}000$*
- *$0 \leq K \leq 10^{18}$*
- *$1 \leq P \leq 10^9$*
- *$1 \leq c_i \leq 10^9$, for $i = 0, 1, \dots, N - 1$*
- *$N \leq j \leq 10^{18}$*
- *$1 \leq v \leq 10^9$*
- *All values are integers*

Example input:

```
10 8
3
1 2 1
2
7 3
5 4
```

Example output:

```
4
```

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

Solution and analysis:

Let's first solve a simpler problem – when the sequence c is cyclic, i. e. when $c_i = c_{i \bmod N}$, for $i \geq 0$.
This simpler version is similar to Fibonacci sequence. Actually, for $N = 1$ and $c_0 = 1$, it is the Fibonacci sequence. To find the K^{th} number of these kind of recursive sequences fast we should first write them in their matrix form.

Matrix form of this sequence is:

$$\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} = \begin{pmatrix} c_{i-1} & c_{i-2} \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{i-1} \\ F_{i-2} \end{pmatrix}$$

Expanding this, we can see that:

$$\begin{pmatrix} F_K \\ F_{K-1} \end{pmatrix} = C_{K-1}C_{K-2} \ldots C_2C_1 \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}, \text{ where } C_i = \begin{pmatrix} c_i & c_{i-1} \\ 1 & 0 \end{pmatrix}.$$

How do we calculate this efficiently?
For relatively small K, and we will take $K < N$ for this case, we can do this just by multiplying all the matrices.
For large K ($K \geq N$), we will take advantage of the fact that c is cyclic. Since c is cyclic with cycle of length N, we know that $C_{N-1}C_{N-2} \ldots C_1C_0 = C_{iN+(N-1)}C_{iN+(N-2)} \ldots C_{iN+1}C_{iN}$, for $i \geq 0$ (note that $C_0 = \begin{pmatrix} c_0 & c_{N-1} \\ 1 & 0 \end{pmatrix}$). Let's define this product of matrices as $S = C_{N-1}C_{N-2} \ldots C_1C_0$.

Now, we can write a formula for F_K that can be calculated quickly:

$$\begin{pmatrix} F_K \\ F_{K-1} \end{pmatrix} = C_{a-1}C_{a-2} \ldots C_1C_0S^bC_{N-1}C_{N-2} \ldots C_2C_1 \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}, \text{ where } b = (K-N) div\ N \text{ and } a = K \bmod N.$$

We can calculate S^b in $O(logb)$ steps using exponentiation by squaring, and then we can just multiply everything in the expression to get F_K quickly.
Let's get back to the full problem, when c is almost cyclic. In this case, we cannot just use S^b in the formula above, because some matrices in S^b may not respect the cyclic property. Instead of S^b, we will have something like

$$S \cdot S \cdot \ldots \cdot S \cdot E_1 \cdot S \cdot S \cdot \ldots \cdot S \cdot E_2 \cdot \ldots = S^{t_1} \cdot E_1 \cdot S^{t_2} \cdot E_2 \cdot S^{t_3} \cdot \ldots$$

where E_i denotes the product of matrices of the cycle, with some matrices being different than the matrices of the original cycle. Also, $i \leq 2M$ since each of the M values of c different than values of the original cycle appears in exactly two matrices, so at most $2M$ of cycles are affected.
We can still calculate each S^{t_i} quickly, using exponentiation by squaring. Since there are at most $2M$ of those, total complexity of this would be $O(MlogK)$.

Now, we only need to calculate each E_i. Naive way would be to just multiply all matrices of E_i. Since the number of matrices is N, this would be $O(N \cdot M)$ worst case, which is too slow. To quickly calculate E_i, we will initially create a segment tree of matrices, with matrices of original cycle in the leaves. Internal nodes of the tree will represent the product of their children. This means that the root will represent the product of all matrices in the cycle. To calculate E_i, we will just update our segment tree with those values that are different than the original values of the cycle. We will do this by updating corresponding leaves of the tree, moving up to the root and updating the products in the internal nodes. After we're done updating the tree with all of the matrices that are different than matrices of the original cycle, we will just use the product in the root of the tree. Finally, we will update the tree back with matrices of the original cycle in order to reuse the segment tree for E_{i+1}.

Since there are $O(N)$ nodes in the segment tree, the complexity of updating is $O(logN)$. The total complexity is then $O(MlogN + MlogK)$. We should also mention that the constant factor is not very small, since we operate on matrices and not just integers.

Note that we need to find $F_K \ mod \ P$ and P may not even be a prime number. However, this does not affect us since we only deal with operations of addition and multiplication throughout the whole procedure and we can just do them all modulo P.

Problem B: Bribes

Statement:

Ruritania is a country with a very badly maintained road network, which is not exactly good news for lorry drivers that constantly have to do deliveries. In fact, when roads are maintained, they become *one-way*. It turns out that it is sometimes impossible to get from one town to another in a legal way – however, we know that all towns *are reachable*, though *illegally*!

Fortunately for us, the police tend to be very corrupt and they will allow a lorry driver to break the rules and drive in the wrong direction provided they receive 'a small gift'. There is one patrol car for every road and they will request 1000 Ruritanian dinars when a driver drives in the wrong direction. However, being greedy, every time a patrol car notices the same driver breaking the rule, they will charge *double* the amount of money they requested the previous time on that particular road.

Borna is a lorry driver that managed to figure out this bribing pattern. As part of his job, he has to make K stops in some towns all over Ruritania and he has to make these stops in a certain order. There are N towns (enumerated from 1 to N) in Ruritania and Borna's initial location is the capital city i.e. town 1. He happens to know which ones out of the $N-1$ roads in Ruritania are currently unidirectional, but he is unable to compute the least amount of money he needs to prepare for bribing the police. Help Borna by providing him with an answer and you will be richly rewarded.

Input:

The first line contains N, the number of towns in Ruritania. The following $N-1$ lines contain information regarding individual roads between towns. A road is represented by a tuple of integers (a, b, x), which are separated with a single whitespace character. The numbers a and b represent the cities connected by this particular road, and x is either 0 or 1: 0 means that the road is bidirectional, 1 means that only the $a \to b$ direction is legal. The next line contains K, the number of stops Borna has to make. The final line of input contains K positive integers $s_1, \ldots, s_K$: the towns Borna has to visit.

Output:

The output should contain a single number: the least amount of thousands of Ruritanian dinars Borna should allocate for bribes, modulo $10^9 + 7$.

Constraints:

- *$1 \le N \le 10^5$*
- *$1 \le K \le 10^6$*
- *$1 \le a, b \le N$ for all roads*
- *$x \in \{0, 1\}$ for all roads*
- *$1 \le s_i \le N$ for all $1 \le i \le K$*

Example input:

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

Example output:

```
4
```

Explanation:

Borna first takes the route $1 \to 5$ and has to pay 1000 dinars. After that, he takes the route $5 \to 1 \to 2 \to 3 \to 4$ and pays nothing this time. However, when he has to return via $4 \to 3 \to 2 \to 1 \to 5$, he needs to prepare 3000 $(1000 + 2000)$ dinars. Afterwards, getting to 2 via $5 \to 1 \to 2$ will cost him nothing. Finally, he doesn't even have to leave town 2 to get to 2, so there is no need to prepare any additional bribe money. Hence, he has to prepare 4000 dinars in total.

```
> Time and memory limit:  1.5s / 64MB
```

Solution and analysis:

Let us first provide a suitable interpretation of the task description. The country can obviously be modelled as an undirected graph, where vertices are towns and edges are roads. We see that it is connected (the description mentions that every city is reachable), but we also know that there are $N-1$ edges, where N is the number of vertices. From this it follows that the graph is, in fact, a tree. For the sake of simplicity, let us make this tree rooted at node 1.
Let us consider just an $a \rightsquigarrow b$ transfer. From the previous assertion it follows that the cheapest path from a to b will always be the shortest path from a to b – which is, in fact, the only path from a to b that does not have any repeated vertices. Borna's trip is thus uniquely defined by all of his stops. Getting from town a to town b requires that Borna first goes to the lowest common ancestor (LCA) node of a and b, and then descends to b (note that the LCA can also be any of the nodes a and b!).
Computing the LCA of two vertices is a well-known problem and may be solved in several different ways.
One possible approach is to use the equivalence between LCA and the range minimum query (RMQ) problem and then compute the LCA of any two vertices in constant time.
Another one is based on heavy path decomposition. In any case, we need to be able to compute the LCA in $O(1)$ time.
Let us now define the notion of a banned (directed) edge: a directed edge $a \to b$ is banned if it requires paying a bribe. If a is the parent of b for a banned edge $a \to b$, then we call $a \to b$ a down-banned edge. Similarly, we may define up-banned edges. If Borna traveled along a banned edge p times, then he will have to prepare $1 + 2 + \cdots + 2^{p-1} = 2^p - 1$ thousands of dinars for bribing the police. Hence we need to determine the number of times every edge was traversed. This depends on whether the edge is down-banned or up-banned.
Before delving into these two cases, we need to compute the following three properties for every town x:

- *$ends_{down}$: the number of times x was the final stop in a path, this is equal to the number of occurrences of x in the array of stops;*
- *$ends_{up}$: the number of times x was the highest stop in a path, this is equal to the number of times x was the LCA of two consecutive stops;*
- *$gone_{up}$: the number of times x was the first stop in a path.*

Now we consider the cases:

- *If an edge $a \to b$ is up-banned, then the number of times it was traversed is equal to the number of times any vertex in a's subtree was an initial stop, minus the number of times any vertex in a's subtree was the highest stop (i.e. sum of all $gone_{up}$'s minus the sum of all $ends_{up}$'s). We may compute these parameters for all vertices at once using just one post-order tree traversal. Thus, we can compute the 'bribe contributions' of all up-banned edge in linear time.*
- *If an edge $a \to b$ is down-banned, then the number of times it was traversed is equal to the number of times any vertex in b's subtree was a final stop, minus the number of times any vertex in b's subtree was the highest stop (i.e. sum of all $ends_{down}$'s minus the sum of all $ends_{up}$'s). Similar to the previous case, we can compute the 'bribe contributions' of all down-banned edges using only one post-order tree traversal.*
- *Hence, by first computing $ends_{down}$, $ends_{up}$ and $gone_{up}$ for every vertex, and then traversing the tree, we are able to compute the answer. The final complexity depends on the implementation of LCA. The asymptotically optimal solution to this problem has $O(N+K)$ time complexity, but even an $O(N \log N + K)$ approach is acceptable given these constraints.*

Problem C: Party

Statement:

People working in MDCS (Microsoft Development Center Serbia) like partying. They usually go to night clubs on Friday and Saturday.
There are N people working in MDCS and there are N clubs in the city. Unfortunately, if there is more than one Microsoft employee in night club, level of coolness goes infinitely high and party is over, so club owners will never let more than one Microsoft employee enter their club in the same week (just to be sure).
You are organizing night life for Microsoft employees and you have statistics about how much every employee likes Friday and Saturday parties for all clubs.
You need to match people with clubs maximizing overall sum of their happiness (they are happy as much as they like the club), while half of people should go clubbing on Friday and the other half on Saturday.

Input:

The first line contains integer N – number of employees in MDCS.
Then an NxN matrix follows, where element in i^{th} row and j^{th} column is an integer number that represents how much i^{th} person likes j^{th} club's Friday party.
Then another NxN matrix follows, where element in i^{th} row and j^{th} column is an integer number that represents how much i^{th} person likes j^{th} club's Saturday party.

Output:

Output should contain a single integer – maximum sum of happiness possible.

Constraints:

- *$2 \leq N \leq 20$*
- *N is even*
- *$0 \leq$ level of likeness $\leq 10^6$*
- *All values are integers*

Example input:

```
4
1 2 3 4
2 3 4 1
3 4 1 2
4 1 2 3
5 8 7 1
6 9 81 3
55 78 1 6
1 1 1 1
```

Example output:

```
167
```

Explanation:

Here is how we matched people with clubs:
Friday: 1^{st} person with 4^{th} club (4 happiness) and 4^{th} person with 1^{st} club (4 happiness).
Saturday: $2nd$ person with 3^{rd} club (81 happiness) and 3^{rd} person with 2^{nd} club (78 happiness)
$4 + 4 + 81 + 78 \ = \ 167$

```
Time and memory limit: 1.5s / 6MB
```

Solution and analysis:

This problem is a variation of the well-known assignment problem. More about the problem can be found on the Wikipedia article - https://en.wikipedia.org/wiki/Assignment_problem.
First, notice that the memory limit is very low. This makes it almost impossible to write a dynamic programming solution. So, let's look at a different approach.
The most naïve solution would be going through $\binom{N}{\frac{N}{2}}$ *combinations of assignments – half of people on Friday and the other half on Saturday. For every combination, we run Hungarian algorithm and find the best answer among all of the combinations. Although the memory complexity of the algorithm is only* $O(N^2)$*, the time complexity of this solution is* $O\left(\binom{N}{\frac{N}{2}}N^3\right)$*, since Hungarian algorithm has* $O(N^3)$ *time complexity.*
This is too slow.

To explain the solution of this problem let's describe the scheme of Hungarian algorithm:

```
HungarianAlgorithm(…)
{
          for(int i=0; i<n; i++)
          {
            hungarian_iteration();
          }
}
```

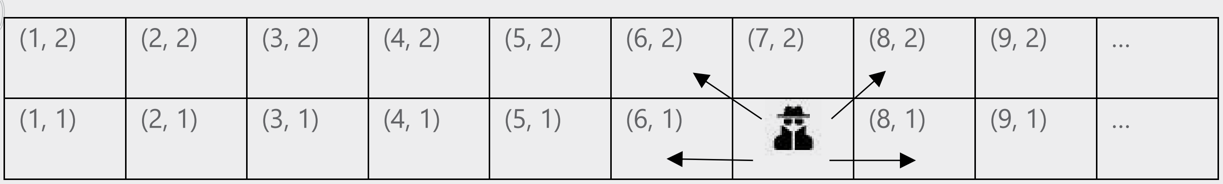
Hungarian algorithm allows rows of the assignment matrix to be added one by one, in $O(N2)$ *time complexity each. To solve our problem, we will recursively go through all sets of binary masks with equal number of* 0s *and* 1s*, where* 0 *in* i*th position means that the* i*th person would go partying on Friday, while 1 denotes a person going partying on Saturday. In each recursive call, we will add a row to the solution for the binary mask we are currently generating. Time complexity of the solution is* $O\left(\binom{N}{\frac{N}{2}}N^2\right)$*, because the number of recursive calls is* $O\left(\binom{N}{\frac{N}{2}}\right)$*. This is enough to solve the task within the constraints.*

Curiosity here is that the number of states we have to visit during our recursion is closely related to problem H: Bots. You can find a detailed explanation in the analysis of that problem.

Problem D: Tablecity

Statement:

There was a big bank robbery in Tablecity. In order to catch the thief, the President called none other than Albert – Tablecity's Chief of Police. Albert does not know where the thief is located, but he does know how he moves.
Tablecity can be represented as $1000 \, x \, 2$ grid, where every cell represents one district. Each district has its own unique name "(X, Y)", where X and Y are the coordinates of the district in the grid. The thief's movement is as follows:
Every hour the thief will leave the district (X, Y) he is currently hiding in, and move to one of the districts: $(X-1, Y)$, $(X+1, Y)$, $(X-1, Y-1)$, $(X-1, Y+1)$, $(X+1, Y-1)$, $(X+1, Y+1)$ as long as it exists in Tablecity.
Below is an example of thief's possible movements if he is located in district $(7, 1)$:



Albert has enough people so that every hour he can pick any two districts in Tablecity and fully investigate them, making sure that if the thief is located in one of them, he will get caught. Albert promised the President that the thief will be caught in no more than 2015 hours and needs your help in order to achieve that.

Input:

There is no input for this problem.

Output:

The first line of output contains integer N – duration of police search in hours. Each of the following N lines contains exactly 4 integers X_{i1} Y_{i1} X_{i2} Y_{i2} separated by spaces, that represent 2 districts (X_{i1}, Y_{i1}) and (X_{i2}, Y_{i2}) which got investigated during i^{th} hour. Output is given in chronological order (i^{th} line contains districts investigated during i^{th} hour) and should guarantee that the thief is caught in no more than 2015 hours, regardless of thief's initial position and movement.

Constraints:

- *1 ≤ N ≤ 2015*
- *1 ≤ X ≤ 1000, 1 ≤ Y ≤ 2*

Example input:

```
No example input
```

Example output:

```
2
5 1 50 2
8 1 80 2
```

Explanation:

Example output is not guaranteed to catch the thief and is not correct. There exists a combination of an initial position and a movement strategy such that the police will not catch the thief.

Consider the following initial position and thief's movement:

In the first hour, the thief is located in district $(1, 1)$. Police officers will search districts $(5, 1)$ and $(50, 2)$ and will not find him.

At the start of the second hour, the thief moves to district $(2, 2)$. Police officers will search districts $(8, 1)$ and $(80, 2)$ and will not find him.

Since there is no further investigation by the police, the thief escaped!

```
> Time and memory limit:  0.1s / 64MB
```

Solution and analysis:

Notice that parity of thief's X coordinate changes every time he moves.
Assume that at the beginning X coordinate of thief's position is odd, and check districts $(1,1)$ and $(1,2)$. The next day check districts $(2,1)$ and $(2,2)$ and so on until 1000^{th} when you check districts $(1000,1)$ and $(1000,2)$. What is achieved this way is that if starting parity was as assumed, thief could have never moved to district with X coordinate i on day $i+1$, hence he couldn't have jumped over the search party and would've been caught. If he wasn't caught, his starting parity was different than we assumed, so on 1001^{st} day we search whatever (1 and 1001 are of the same parity, so we need to wait one day), and then starting on 1002^{nd}day we do the same sweep from $(1,1)$ and $(1,2)$ to $(1000,1)$ and $(1000,2)$ and guarantee to catch him.
Shortest possible solution is by going from $(2,1)$ and $(2,2)$ to $(1000,1)$ and $(1000,2)$ twice in a row, a total of 1998 days, which is correct in the same way. First sweep catches the thief if he started with even X coordinate, and second sweep catches the thief if he started with odd X coordinate.

Problem E: Spectator riots

Statement:

It's riot time on football stadium Ramacana! Raging fans have entered the field and the police find themselves in a difficult situation. The field can be represented as a square in the coordinate system defined by two diagonal vertices in $(0,0)$ and $(10^5,10^5)$. The sides of that square are also considered to be inside the field, everything else is outside.

In the beginning, there are N fans on the field. For each fan we are given his speed, an integer v_i as well as his integer coordinates (x_i, y_i). A fan with those coordinates might move and after one second he might be at any point $(x_i + p,\ y_i + q)$ where $0 \leq |p| + |q| \leq v_i$. p, q are both integers.

Points that go outside of the square that represents the field are excluded and all others have equal probability of being the location of that specific fan after one second.

Andrej, a young and promising police officer, has sent a flying drone to take a photo of the riot from above. The drone's camera works like this:

It selects three points with integer coordinates such that there is a chance of a fan appearing there after one second. They must not be collinear, or the camera won't work. It is guaranteed that not all of the initial positions of fans will be on the same line.

Camera focuses those points and creates a circle that passes through those three points. A photo is taken after one second (one second after the initial state).

Everything that is on the circle or inside it at the moment of taking the photo (one second after focusing the points) will be on the photo.

Your goal is to select those three points so that the expected number of fans seen on the photo is maximized. If there are more such selections, select those three points that give the circle with largest radius among them. If there are still more suitable selections, any one of them will be accepted.

Input:

The first line contains the number of fans on the field, N. The next N lines contain three integers: x_i, y_i, v_i. They are the x-coordinate, y-coordinate and speed of fan i at the beginning of the one second interval considered in the task.

Output:

You need to output the three points that camera needs to select. Print them in three lines, with every line containing the x-coordinate, then y-coordinate, separated by a single space. The order of points does not matter.

Constraints:

- *$3 \leq N \leq 10^5$*
- *$0 \leq x_i, y_i \leq 10^5$*
- *$0 \leq v_i \leq 1{,}000$*
- *All numbers will be integers*

Example input:

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

Example output:

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

Explanation:

The circle defined in output will catch all of the fans, no matter how they move during one second.

```
> Time and memory limit: 0.5s / 128MB
```

Solution and analysis:

First, we simplify the problem by replacing the initial set of points with the set of all points where some fan might appear after one second, call that set S. Every point in S has a probability that a specific player will appear there. Consequently, for every point P in S we know the expected number of fans at it after one second, call it P_e .
Now, for some arbitrary circle that passes through some three points of S (which doesn't violate the rules of the problem), the expected number of fans caught on camera is the sum of all T_e, where T is a point on or inside the circle. Our goal is to find a circle that maximizes that sum.
After drawing a few examples, we can notice that we can always catch most of the points that are possible locations of some fan or even all of them. We can write a brute-force solution that will increase our suspicion that all fans can be caught, no matter how they move.
Now let's try to find the largest circle of those that surely catch all fans and don't violate the rules in the problem. It is easy to see that three fixed points that determine the circle must lie on the convex hull of S (otherwise we surely wouldn't catch all points of S with that circle).
Convex hull can be computed in $|S|\, log(|S|)$ which might be too slow if unnecessary points are not eliminated from S.
Notice that for every fan in input, if his speed is v, he might appear at $O(v^2)$ points, so convex hull algorithm would have $O(N \cdot v^2 \cdot log(N \cdot v))$ complexity, which is too slow.
The trick is to take only convex hull of those $O(v^2)$ points, which will have $O(1)$ points. All other points should be eliminated from S as they don't have a chance of appearing on convex hull of S. Contestants need to be careful with edge cases when a fan potentially goes out of the field.
After computing the convex hull of S (call it $H(S)$), we hope to find the circle that will pass through some three points on that hull and contain all other points inside it or on it.
These two claims can be proven geometrically:

1. *For a convex polygon, the largest circle among all circumcircles of triangles determined by the polygon vertices will surely contain all vertices of the polygon on it or inside it.*
2. *For a convex polygon, the largest circumcircle of some triangle that is determined by vertices of the polygon is a circumcircle of a triangle that contains three consecutive vertices of a polygon.*

With 1) and 2) we conclude:
The largest circle among those that are circumscribed around triangles that are composed of three consecutive vertices of $H(S)$ contains all of the points of $H(S)$ (and then obviously of S) and no other circle that contains all those points can be larger.
This means that we can finish the problem easily in linear time (with respect to the size of convex hull).

Problem F: Bulbo

Statement:

Bananistan is a beautiful banana republic. Beautiful women in beautiful dresses. Beautiful statues of beautiful warlords. Beautiful stars in beautiful nights.
In Bananistan people play this crazy game – Bulbo. There's an array of bulbs and each player has a position, which represents one of the bulbs. Distance between two neighboring bulbs is 1. Each turn one contiguous set of bulbs lights-up, and players have the cost that's equal to the distance from the closest shining bulb. Then all bulbs go dark again. Before each turn players can change their position with $|pos_{new} - pos_{old}|$ cost, and players know three next light-ups. The goal is to minimize your summed cost. I tell you, Bananistanians are spending their nights playing with bulbs.
Banana day is approaching, and you are hired to play the most beautiful Bulbo game ever. A huge array of bulbs is installed, and you know your initial position and all the light-ups in advance. You need to play the ideal game and impress Bananistanians, and their families.

Input:

The first line contains number of turns n and initial position x. Next n lines contain two numbers l_{start} and l_{end}, which represent that all that bulbs from $[l_{start}, l_{end}]$ interval are shining this turn.

Output:

Output should contain a single number which represents the best result (minimum cost) that could be obtained by playing this Bulbo game.

Constraints:

- $1 \le n \le 5{,}000$
- $1 \le x \le 10^9$
- $1 \le l_{start} \le l_{end} \le 10^9$

Example input:

```
5 4
2 7
9 16
8 10
9 17
1 6
```

Example output:

```
8
```

Example-play:

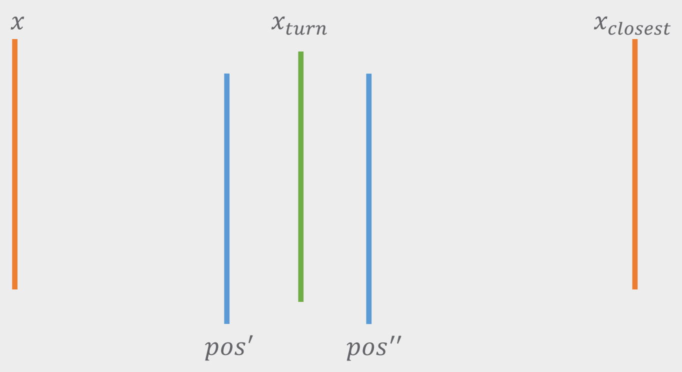
Before 1. turn move to position 5
Before 2. turn move to position 9
Before 5. turn move to position 8

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

Solution and analysis:

Let's start by creating a solution in $O(n \cdot maxX)$ complexity. It's a simple dynamic programming approach.
We have an array dyn (size $maxX$), which should say how costly would it be to be in this position at the end of the turn. At initialization step each position is set to $maxValue$ (some big number), except the initial position which is set to be 0. This should represent that in the initial moment it's impossible to be anywhere except in the initial position.
Before each turn we can change position, so before each turn we could calculate the best cost of being in that position when the bulbs light up. We can do this by passing this array twice – once from the left and once from the right. Consider the case we're passing from left to right (smaller index i to larger index i, the other case could be explained in the similar fashion). While passing, we can keep the $bestVal$, which would be initiated to $dyn[0]$, and updated as $bestVal = min(bestVal + 1, dyn[i])$, $dyn[i] = bestVal$ for each $i > 0$. Rationale: if we came from the $i - 1$ position we have the same cost of that position $+1$, if we didn't we have the same cost we had before this exercise. Now we only need to add distance to the closest bulb for each position and we finished this turn. When we finish each turn we pick the lowest value in the array, and that's our solution. Simple enough.

But this solution is too slow for us. We want more.
Statement: We never have to move to the position which is not beginning or the end position of one of the light-ups.
Let's consider following situation: We're at the position x. If x is going to be inside of the next turn shining bulbs, there's no point of moving at all (it would be the same as moving after the turn). So, consider x is outside those bulbs, and $x_{closest}$ is the closest bulb to x that will be lighten-up. Also, consider $x < x_{closest}$ (the other case could be explained in the similar fashion). Consider all the remaining light-up points (light-up beginning and end positions) are in the array pos, which is sorted. Take a look at the following picture:



First thing we could notice is the fact that our cost for this turn is going to be $x_{closest} - x$, if we finish the turn anywhere between x and $x_{closest}$ inclusive. Going left from x or right from $x_{closest}$ doesn't make any sense, because we would have same or bigger cost than staying in x or $x_{closest}$ and moving from it in the next turn.

Next, let's consider we haven't ended our turn on some light-up endpoint, but between two neighboring endpoints, pos' and pos''. Let's call that position x_{turn}. Let's also introduce $x_{closest2}$, which is the closest bulb from the x_{turn} in the next turn light-up.
If x_{turn} is shining, then pos' is shining as well, so we could have finished our turn there. If $x_{closer2} \leq pos'$ we would be better off or equally well if we finished our turn in pos'. In that case we would have $x_{turn} - pos'$ smaller cost for the next turn. If afterwards we need to go to x_{turn}, total cost would not exceed the cost of going straight to x_{turn} in the initial turn. If $x_{closer2} \geq pos''$ we would be better off or equally well if we finished our turn in pos'', similar to the explanation for pos'. So, in each turn we could stay in the place or go to the closest light-up endpoint and we could still get the optimal solution.
We can use this fact to make a $O(n^2)$ solution – instead of each position we should take consider only light-up endpoints and initial position. Everything else is the same as in original solution.
Dynamic programming solution is enough to pass within the constraints for the program, but this problem can be solved in linear time as well.
Let's look at the values of array dyn. We can notice that this array actually has only one local minimum at each turn. What this means is that we have a range $[l, r]$ and that all of the values from $dyn[0]$ to $dyn[l]$ are monotonically decreasing, all values from $dyn[r]$ to $dyn[10^9]$ are monotonically increasing, while all of the values $dyn[l], dyn[l+1], \dots, dyn[r]$ have the same value and represent the minimum summed cost until this turn. We can use this property to create a linear time algorithm.
Our linear algorithm will be as follows:
At the beginning, our optimal range will be $[x_{start}, x_{start}]$ and minimum cost will be 0. At each turn, we will update this optimal range and minimum cost.
If the range of shining bulbs in the next turn intersects with our optimal range, we can easily see that our new optimal range will be this intersection. The minimum cost will stay the same as cost for previous turn, since we don't need to move if we are located somewhere in this intersection, as we will already be located at a bulb that is shining. Anywhere outside of this intersection, the cost would increase since either the distance to the closest shining bulb would be larger than 0, or because of moving from our optimal range to somewhere outside of it, or both.
If the range of shining bulbs in the next turn does not intersect our optimal range and is left from our it, we will set that our optimal range is from the rightmost shining bulb to the left end of our previously optimal range. Our minimum cost will increase for exactly the distance between these positions – if we don't move from the left end of our previously optimal range, our cost increases for this distance. If we move from the left end of our previously optimal range by one position to left, we decrease our distance in the next turn by 1, but increase the cost by 1 because of the move. Same goes for moving two positions to left, and so on until movement to the rightmost shining bulb in the next turn (at which point our distance to shining bulb will be 0, but our cost for moving will be the same as distance when we didn't move at all). It is easily seen that the minimum cost for a position that is left from this new optimal range is larger by 1, and the minimum cost for a position that is right from this new optimal range is also larger by 1.
Moving further to the left or right, this cost increases more and more, resulting in only one local minimum as we described previously.
If the range of shining bulbs does not intersect our optimal range and is right from it, we can do a similar thing as we do when the range is left from our optimal range.
After all the turns, we have our optimal range and the minimum cost possible. The total complexity is $O(n)$, since in each turn we update the range in $O(1)$.

Problem G: Run for beer

Statement:

People in BubbleLand like to drink beer. Little do you know, beer here is so good and strong that every time you drink it your speed goes 10 times slower than before you drank it.
Birko lives in city Beergrade, but wants to go to city Beerburg. You are given a road map of BubbleLand and you need to find the fastest way for him. When he starts his journey in Beergrade his speed is 1. When he comes to a new city he always tries a glass of local beer, which divides his speed by 10.
The question here is what the minimal time for him to reach Beerburg is. If there are several paths with the same minimal time, pick the one that has least roads on it. If there is still more than one path, pick any.
It is guaranteed that there will be at least one path from Beergrade to Beerburg.

Input:

The first line of input contains integer N – number of cities in Bubbleland and integer M – number of roads in this country. Cities are enumerated from 0 to $N-1$, with city 0 being Beergrade, and city $N-1$ being Beerburg.

Each of the following M lines contain three integers a, b ($a \neq b$) and len. These numbers indicate that there is a bidirectional road between cities a and b with length len.

Output:

The first line of output should contain minimal time needed to go from Beergrade to Beerburg.
The second line of the output should contain the number of cities on the path from Beergrade to Beerburg that takes minimal time.
The third line of output should contain the numbers of cities on this path in the order they are visited, separated by spaces.

Constraints:

- *$2 \leq N \leq 10^5$*
- *$1 \leq M \leq 10^5$*
- *$0 \leq len \leq 9$*
- *There is at most one road between two cities*

Example input:

```
8 10
0 1 1
1 2 5
2 7 6
0 3 2
3 7 3
0 4 0
4 5 0
5 7 2
0 6 0
6 7 7
```

Example output:

```
32
3
0 3 7
```

```
> Time and memory limit:  0.5s / 64MB
```

Solution and analysis:

The problem can be restated as follows: given an undirected graph. Consider any path with edge lengths $l_0, l_1, \dots, l_{len-1}$. It's cost is $l_0 + 10l_1 + \dots + 10^{len-1}\{l_{len-1}\}$. We need to find the cost of the cheapest path from 0 to $n - 1$ and output it.
For now, let's ignore the leading zeros. Notice that the distance after walking the edge (u, v) can be obtained by putting the length of this edge in front of the distance to get to u. So, the trivial solution is to run BFS from vertex 0 and store for each vertex the smallest number to get to it. But it will run in $O(N^2)$ time because we will have to compare large number in order to get the best one for each vertex.
In order to avoid it we can store the equivalence classes instead of actual numbers, so that we could compare their classes, not actual numbers. The vertex 0 will have class 0. We will split the graph into layers. The k^{th} layer will have vertexes with length of distance exactly k. Now process the graph layer by layer. For k^{th} layer we know all equivalence classes, let's obtain the $\{k + 1\}^{st}$ layer and all the equivalence classes for it. Imagine we walked some edge (u, v) with length l. The distance to v, $d(v)$ will be $d(u)$. If we need to compare two numbers with the same length, the equivalence classes allow us to replace it with just a pair $\left(l, class(u)\right)$. Store the minimal pair for each vertex of the next layer, sort and shrink them obtaining the equivalence class for new layer. This is easily done in $O\left(nlog(n)\right)$, but also a $O(n \cdot alphabet)$ solution exists.
So how do we handle zeros? They behave almost the same way as other digits with only one exception: when they are in front of distance (leading zeros). For example, the path with length 001 is smaller than 11 despite being longer. The problems in the middle of the path don't cause any problems to solution described above. To solve this problem, let's process them in different manner: calculate for each vertex the smallest number of zeros to get to it from $n - 1$. Now the answer can be obtained as follows: walk from 0 to vertex x by the shortest path and go from x to $n - 1$ only by zero edges.

Problem H: Bots

Statement:

Sasha and Ira are two best friends. But they aren't just friends, they are software engineers and experts in artificial intelligence. They are developing an algorithm for two bots playing a two-player game. The game is cooperative and turn based. In each turn, one of the players makes a move (it doesn't matter which player).
Algorithm for bots that Sasha and Ira are developing works by keeping track of the state the game is in. Each time either bot makes a move, the state changes. And, since the game is very dynamic, it will never go back to the state it was already in at any point in the past.
Sasha and Ira are perfectionists and want their algorithm to have an optimal winning strategy. They have noticed that in the optimal winning strategy, both bots make exactly N moves each. But, in order to find the optimal strategy, their algorithm needs to analyze all possible states of the game (they haven't learned about alpha-beta pruning yet) and pick the best sequence of moves.
They are worried about the efficiency of their algorithm and are wondering what is the total number of states of the game that need to be analyzed?

Input:

The first and only line contains integer N.

Output:

Output should contain a single integer – number of possible states modulo $10^9 + 7$.

Constraints:

- *$1 \le N \le 10^6$*

Example input:

```
2
```

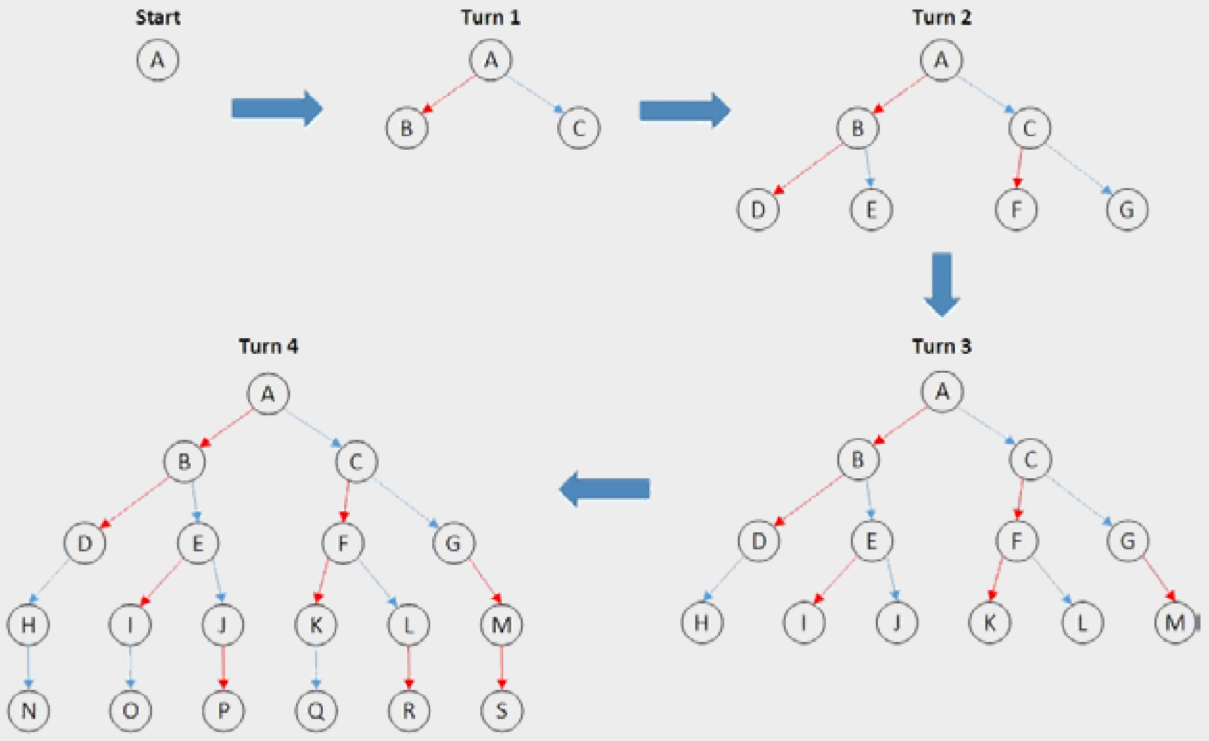
Example output:

```
19
```

Explanation:

Start: Game is in state A.
Turn 1: Either bot can make a move (first bot is red and second bot is blue), so there are two possible states after the first turn – B and C.
Turn 2: In both states B and C, either bot can again make a turn, so the list of possible states is expanded to include D, E, F and G.
Turn 3: Red bot already did $N = 2$ moves when in state D, so it cannot make any more moves there. It can make moves when in state E, F and G, so states I, K and M are added to the list. Similarly, blue bot cannot make a move when in state G, but can when in D, E and F, so states H, J and L are added.
Turn 4: Red bot already did $N = 2$ moves when in states H, I and K, so it can only make moves when in J, L and M, so states P, R and S are added. Blue bot cannot make a move when in states J, L and M, but only when in H, I and K, so states N, O and Q are added.

Overall, there are 19 possible states of the game their algorithm needs to analyze.



> Time and memory limit: 2s / 64MB

Solution and analysis:

Problem naturally can be transformed to a more formal way: how many vertices will a trie contain if we add all possible strings with length $2 * N$ with equal number of zeros and ones to it. So, it is obvious that upper half of this tree would be a full binary tree.
Let's take a look on $N = 3$:

- *level 0 - 1 vertex,*
- *level 1 - 2 vertices,*
- *level 2 - 4 vertices,*
- *level 3 - 8 vertices.*

Starting from N^{th} level not every vertex will duplicate: only those that haven't spent theirs 0s or 1s will.

So, here is how to calculate how many vertices will be there on level $i + 1$:
Lets' assign Number_of_duplicating_vertices_from_level to $PD(i)$

$$Count_{i+1} = PD(i) * 2 + (Count_i - PD(i)).$$

And PD can be calculated pretty easily with binomial coefficients: $PD(i) = 2 * C(i, N)$.
Everything else is implementation techniques: inverse module arithmetic's + some fast way to calculate these $C(i, N)$.

Problem I: Robots protection

Statement:

Company "Robots industries" produces robots for territory protection. Robots protect triangle territories – right isosceles triangles with catheti parallel to North-South and East-West directions.
Owner of some land buys and sets robots on his territory to protect it. From time to time, businessmen want to build offices on that land and want to know how many robots will guard it. You are to handle these queries.

Input:

The first line contains integer N – width and height of the land, and integer Q – number of queries to handle.
Next Q lines contain queries you need to process.
Two types of queries:
$1\ dir\ x\ y\ len$ – add a robot to protect a triangle. Depending on the value of dir, the values of x, y and len represent a different triangle:
$dir\ =\ 1$: Triangle is defined by the points $(x, y), (x + len, y), (x, y + len)$
$dir\ =\ 2$: Triangle is defined by the points $(x, y), (x + len, y), (x, y - len)$
$dir\ =\ 3$: Triangle is defined by the points $(x, y), (x - len, y), (x, y + len)$
$dir\ =\ 4$: Triangle is defined by the points $(x, y), (x - len, y), (x, y - len)$
$2\ x\ y$ – output how many robots guard this point (robot guards a point if the point is inside or on the border of its triangle)

Output:

For each second type query output how many robots guard this point. Each answer should be in a separate line.

Constraints:

- *1 ≤ N ≤ 5,000*
- *1 ≤ Q ≤ 10^5*
- *1 ≤ dir ≤ 4*
- *All points of triangles are within range [1, N]*
- *All numbers are positive integers*

Example input:

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

Example output:

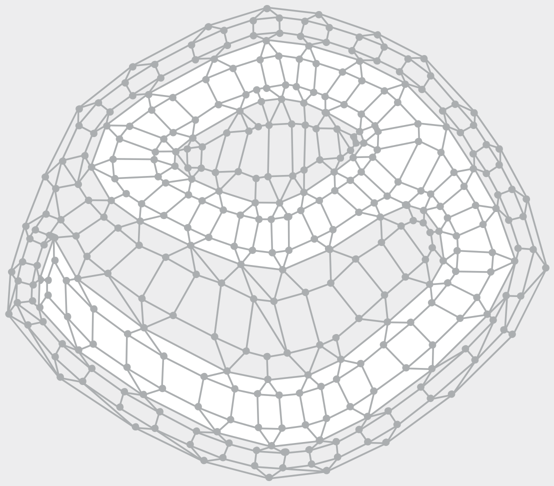
```
2
2
2
0
1
```

```
> Time and memory limit: 1s / 512MB
```

Solution and analysis:

To solve this problem, we should first take a look at one easier problem. Consider having the same problem statement, but with rectangles instead of triangles. The solution to this problem is straightforward.
We will store a matrix representing points in our coordinate system. For each type 1 *query, given the rectangle* $((x_{ul}, y_{ul}), (x_{ur}, y_{ur}), (x_{ll}, y_{ll}), (x_{lr}, y_{lr}))$*, we would add -1 to the points* $(x_{ul}, y_{ul} + 1)$ *and* $(x_{lr} + 1,\ y_{lr})$*, and 1 to the points* (x_{ll}, y_{ll}) *and* $(x_{ur} + 1, y_{ur} + 1)$*. Notice that we expanded the given rectangle when adding* $+1s$ *and* $-1s$*, this is because the point is considered in the rectangle even when it is on the border! This way, when type 2 query is received, to find the answer we simply sum every value in the rectangle* $(0, 0)$ *to* (x, y)*. Since simply summing the points in rectangles is* $O(n^2)$*, we should use binary indexed tree for this, hence getting the sufficient time complexity* $O(log^2 n)$ *per query.*
We will use this approach with some modification to solve the original problem. For handling the triangles, we need to introduce new coordinate systems. Depending on the type of triangle (types 1 and 4 are analogous, so are types 2 and 3) we will make two new coordinate systems, where the corresponding hypotenuses are parallel to one axis. For types 2 and 3, point (x, y) *in original coordinate system would map to* $(x + y, y)$*, and for types 1 and 4, point* (x, y) *would map to* $(x + n - y - 1, y)$*, where* n *is the size of coordinate plane given in the input.*
The problem is now somewhat abstracted to the simple rectangle problem. This time we will need three matrices, one representing original coordinate system and two representing the introduced coordinate systems. For each triangle we need to border it with +1s and -1s, similarly as in the rectangle case, only using 2 matrices for each triangle. When adding border for the cathetus we use the original coordinate system and for the hypotenuse we need one of the two introduced coordinate systems, depending on the type of the triangle. Remember, the point belongs to the triangle even when it is on one of catheti or hypotenuse, so be careful.
For calculating the answer for type 2 query, we need to sum the values in the rectangle from $(0, 0)$ *to* (x, y) *in all three matrices (of course,* (x, y) *needs to be mapped accordingly to the coordinate system each matrix represents). Again, to not exceed time limit we need to use binary indexed trees.*
Since we are using binary indexed tree, the time complexity of this solution is $O(Q \cdot log^2 N)$*.*

bubble cup 9



Problem A: Cowboy Beblop at his computer

Cowboy Beblop is a funny little boy who likes sitting at his computer. He somehow obtained two elastic hoops in the shape of $2D$ polygons, which are not necessarily convex. Since there's no gravity on his spaceship, the hoops are standing still in the air. Since the hoops are very elastic, Cowboy Beblop can stretch, rotate, translate or shorten their edges as much as he wants.

For both hoops, you are given the number of their vertices, as well as the position of each vertex, defined by its X, Y and Z coordinates. The vertices are given in the order they're connected: the 1st vertex is connected to the 2nd, which is connected to the 3rd, etc., and the last vertex is connected to the first one. The hoops are connected if it's impossible to pull them to infinity in different directions by manipulating their edges, without having their edges or vertices intersect at any point – just like when two links of a chain are connected. The polygons' edges do not intersect or overlap.

Cowboy Beblop is fascinated with the hoops he has obtained, and he would like to know whether they are connected or not. Since he's busy playing with his dog, Zwei, he'd like you to figure it out for him. He promised you some sweets if you help him!

Input:

The first line of input contains an integer N, which denotes the number of edges of the first polygon.

The next N lines each contain the integers X, Y and Z - coordinates of the vertices, in the manner mentioned above.

The next line contains an integer M, denoting the number of edges of the second polygon, followed by M lines containing the coordinates of the second polygon's vertices.

Output:

Your output should contain only one line, with the words "YES" or "NO", depending on whether the two given polygons are connected.

Constraints:

- $3 \le N \le 10^5$
- $3 \le M \le 10^5$
- $-10^6 \le X, Y, Z \le 10^6$

It is guaranteed that both polygons are simple (no self-intersections), and in general that the obtained polygonal lines do not intersect each other. Also, you can assume that no 3 consecutive points lie on the same line.

Example input:

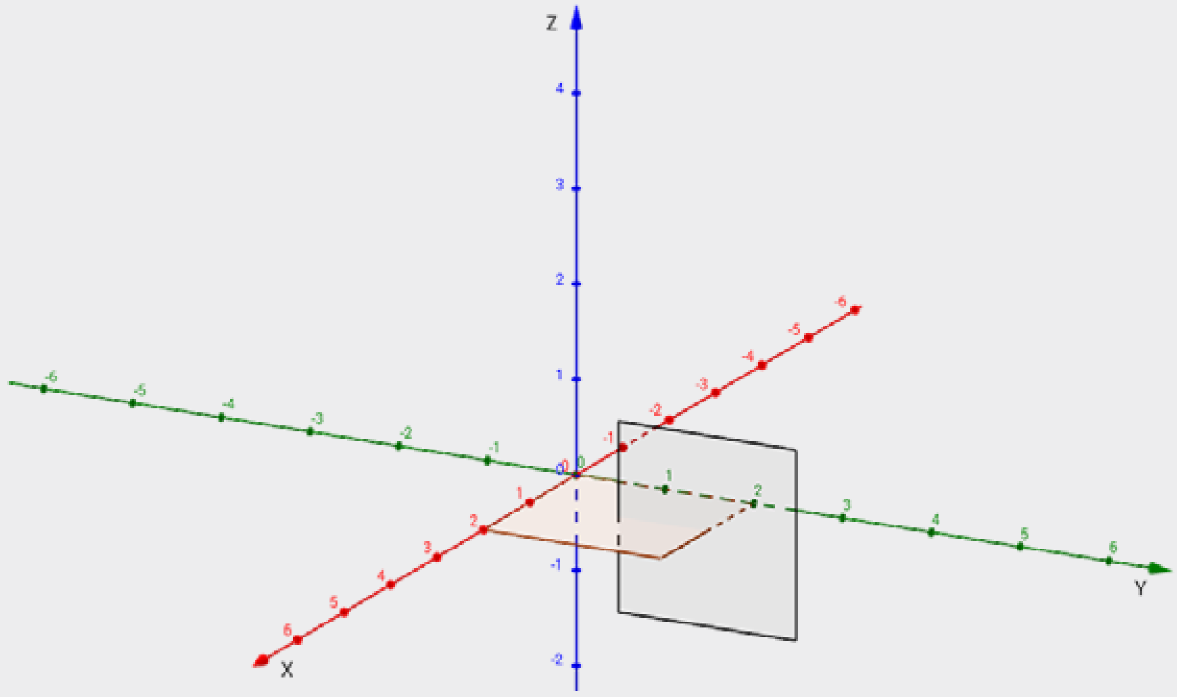
```
4
0 0 0
2  0  0
2  2  0
0 2 0
4
1  1 -1
1  1  1
1  3  1
1  3 -1
```

Example output:

```
YES
```

Explanation:

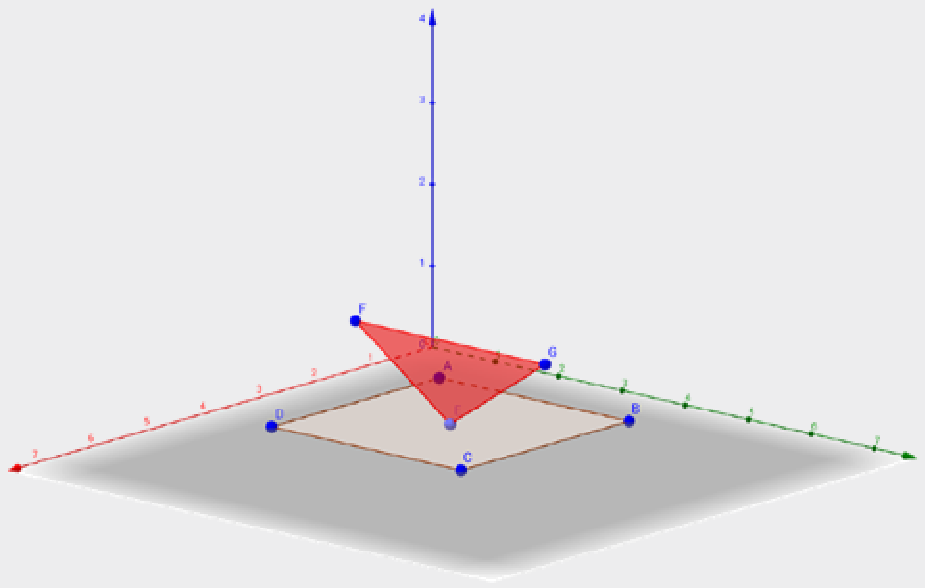
In the picture below, the two polygons are connected, as there is no way to pull them apart (they are shaped exactly like two square links in a chain). Note that the polygons do not have to be parallel to any of the $xy-, xz-, yz-$ planes in general.



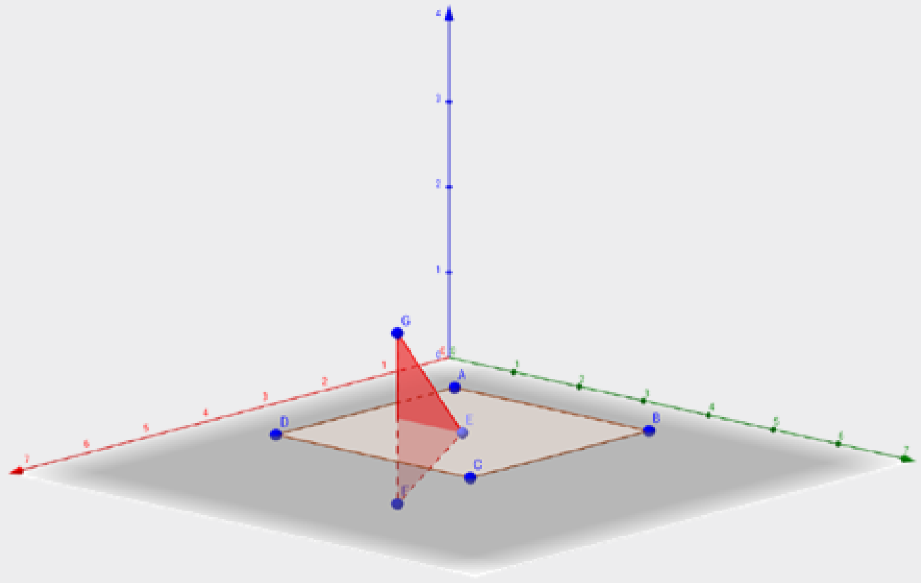
```
> Time and memory limit: 1s / 64MB
```

Solution and analysis:

The solution to this task effectively consists of two parts: analyzing the geometry and the relations between the two polygons and deriving whether they intersect or not.
For the second part, we need to find all intersections between each of the polygons and the common line of their two planes. Once that is done, and the intersection points along the line are sorted, we can simply go through them and count the number of intersection of, say, polygon P1 with the inside of polygon P2, as well as keep track of the directions in which it passes through P2. Whenever we reach a point belonging to polygon P2, our position (inside or outside of polygon P2) changes. Now we simply count points belonging to polygon P1, which we reach while being inside of the polygon P2. This solution has the complexity of $O(N * logN)$, *where* N *is the sum of the edges of the two polygons.*
As for the geometry – it seems that an approach using vectors (the mathematical ones, not arrays with variable length) is much easier than the others. It relieves the coder from having to solve complicated equations, but instead uses relatively simple calculus, once all the vector operations have been defined. Thus, we first define the vectors of the two polygons' planes as the vector product of two consecutive edges' vectors. The two consecutive vectors will be used as the base of the $2D$ *vector space defined by the plane. Then, for each edge of both polygons, we need to see whether it has intersections with the other polygon's plane or not. Let's observe the case when the edge's end points (let's label them* A *and* B*) are on the opposite sides of the plane (*α*). If the angle between* α*'s direction vector and the vector from* A *to a random point in* α *is acute, then the angle between* α*'s direction vector and the vector from* B *to the same point will be obtuse, and vice-versa. Thus, if the dot products of the two vectors for both* A *and* B *are of different sign, they are on the opposite side of* α *and there is an intersection. If either of the products is zero, then the corresponding point is inside* α *(this case will be discussed later on).*
Finding the point of intersection (point X*) between the line* AB *and* α *can be done in several ways. One solution is to pick a random point in* α *(point* C*) and observe the vector* CA*. It is easy to see that one can represent it as a linear combination of* α*'s base vectors and the vector* AB*. Once the parameters of the linear combination have been found by solving the system of equations (determinants seem easiest), one can simply ignore the component associated with the vector* AB*. The rest will sum up to the vector* CX*, and thus* X *is found. It can also be done with even simpler expressions using vectors (left to the reader to find out), but this solution seemed very intuitive and easy to code, so I stuck with it.*
The only special case happens when a polygon's vertex lies exactly on the other polygon's plane. In that case, one simply observes the previous and the next vertex and whether they are on the opposite sides of the plane or not. If they are – the intersection is taken as the "problematic" point and if they are not, we assume there is no intersection. See Picture 1 and Picture 2. Similar applies when two consecutive points are lying on the plane. Please note that simply ignoring the point lying on the other polygon's plane is wrong.



Picture 1: Point E is on plane of A-B-C-D polygon. FG segment doesn’t intersect A-B-C-D polygon, so E is not point of intersection.



Picture 2: Point E is on plane of A-B-C-D polygon. In this case FG segment intersect A-B-C-D polygon, so E is point of intersection.

Problem B: Underfail

You have recently fallen through a hole and, after several hours of unconsciousness, you realized you are in an underground city. On one of your regular daily walks through the unknown, you have encountered two unusually looking skeletons called Sanz and P'pairus, who decided to accompany you and give you some puzzles for seemingly unknown reasons.
One day, Sanz has created a crossword for you. Not any kind of crossword, but a $1D$ crossword! You are given a string of length N and M words, none of which is longer than K. You are also given an array $P[\,]$, which designates how much each word is worth – the i^{th} word is worth $P[i]$ points.
Whenever you find one of the M words in the string, you are given the corresponding number of points. Each letter in the crossword can be used at most X times. A certain word can be counted at different places, but you cannot count the same appearance of a word multiple times. If a word is a substring of another word, you can count them both (presuming you haven't used the letters more than X times).
In order to solve the puzzle, you need to tell Sanz what's the maximum achievable number of points in the crossword.

Input:

The first line of input will contain one integer N– the length of the crossword, and the second line will contain the crossword string. The third line will contain the integer M – the number of given words, and the next M lines will contain descriptions of words: each line will have a word string and an integer p. The last line of the input will contain X – the maximal number of times a position in the crossword can be used.

Output:

Output a single integer – the maximal number of points you can get.

Constraints:

- $1 \le N \le 500$
- $1 \le M \le 100$
- $1 \le X \le 100$
- $1 \le K \le 500$
- $0 \le p \le 100$

Example input:

```
6
abacba
2
aba 6
ba 3
3
```

Example output:

```
12
```

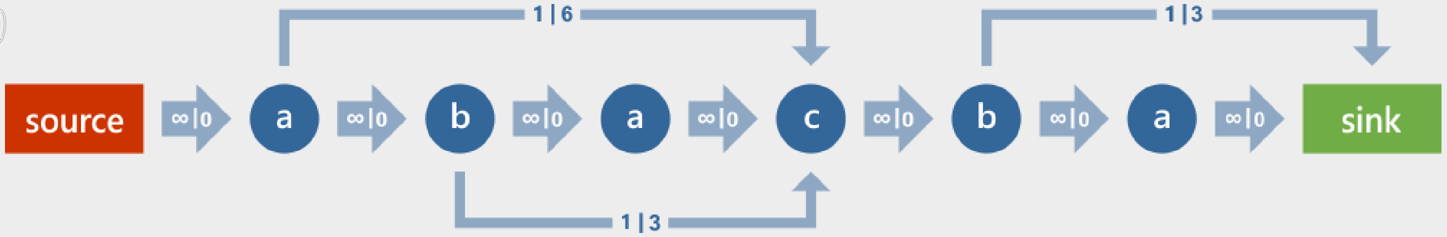
Explanation:

For example, with the string “$abacba$”, words “aba” (6 points) and “ba” (3 points), and $X = 3$, you can get at most 12 points - the word “aba” appears once (“$abacba$”), while “ba” appears two times (“$abacba$”). Note that for $X = 1$, you could get at most 9 points, since you wouldn’t be able to count both “aba” and the first appearance of “ba”.

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

In the basic case (when $X = 1$) we have a common DP problem. Unfortunately, for larger X it is much more complicated, so the basic case cannot be generalized.
Instead, we can look at this problem as a graph problem. We represent the crossword as a directed weighted graph, where edges have both a cost and a capacity. First, we represent every letter from the crossword as a vertex and add an extra source and sink node. Then, we connect each letter's vertex with the next one with an edge of capacity ∞ and cost 0. Also, we connect the source node with the first letter and we connect the last letter with the sink node using the same kind of edge. Finally, for every position where one of the given words matches a substring, we connect the first letter of the match with a letter just after the end of the match with an edge of capacity 1 and a cost equal to the score we get for the word. By doing it for an example input we get this graph:



Any flow with total capacity X is equivalent to a set of selected words that satisfies the given constraint, since we can have at most X "overlapping" 1-capacity edges. The problem now reduces to maximizing the cost of that flow, which can be done by adapting an algorithm that solves the min-cost flow problem (by using negative values). Using an adaptation of Bellman-Ford algorithm we can get a complexity of $O(E^2 V loq V)$.

Problem C: Paint it really, really black

I see a pink boar and I want it painted black. Black boars look much more awesome and mighty than the pink ones. Since Jaggy became the ruler of the forest, he has been trying his best to improve the diplomatic relations between the forest region and the ones nearby.

Some other rulers, however, have requested too much in return for peace between their two regions, so he realized he has to resort to intimidation. Once a delegate for diplomatic relations of a neighboring region visits Jaggy's forest, they might suddenly change their mind about attacking Jaggy, if they see a whole bunch of black boars. Black boars are really scary, after all.

Jaggy's forest can be represented as a tree (graph without cycles) with N vertices. Each vertex represents a boar and is colored either black or pink. Jaggy has sent a squirrel to travel through the forest and paint all the boars black. The squirrel, however, is quite unusually trained and while it traverses the graph, it changes the color of every vertex it visits, regardless of its initial color: pink vertices become black and black vertices become pink.

Since Jaggy is too busy to plan the squirrel's route, he needs your help. He wants you to construct a walk through the tree starting from vertex 1 such that in the end all vertices are black. A walk is any alternating sequence of vertices and edges, starting and ending with a vertex, such that every edge in the sequence connects the vertices before and after it.

Input:

The first line of input contains the integer N, denoting the number of nodes of the graph. The following line contains N integers, which represent the color of each node.

If the i^{th}integer is:

1, then the corresponding node is black

-1 , then the node is pink.

Each of the next $N-1$ lines contain two integers, which represent the indexes of the nodes which are connected (one-based).

Output:

Output the path of a squirrel: output a sequence of visited nodes' indexes in order of visiting. In case all of the nodes are initially black you should print 1.

Constraints:

- $2 \leq N \leq 2 \cdot 10^5$

Example input:

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

Example output:

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

Explanation:

At the beginning squirrel is at node 1 and its color is black. Next steps are as follows:
From node 1 we walk to node 4 and change its color to **pink**
From node 4 we walk to node 2 and change its color to **pink**
From node 2 we walk to node 5 and change its color to **black**
From node 5 we return to node 2 and change its color to **black**
From node 2 we walk to node 4 and change its color to **black**
Finally, we visit node 3 and change its color to **black**

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

Root the tree at node 1. Now notice that if we had a function F such that $F(n)$ colors the whole subtree of node n black (except maybe n itself) and returns us to n in the process, we can easily solve the problem. Let p be the parent of n. Then after entering n do $F(n)$. If n is black, go to p and never return to that subtree again. Otherwise, go from n to p then to n then to p. Now n is black and we can continue to do the same for other children of p and so on.
Now it is easy to see that all that we need to do is to make DFS-like tour of the tree and upon returning from a node to a corresponding parent we check the color of the child node. If it is black, continue normally, otherwise visit it again and again return to the parent and then continue normally. The only exception is the root node as it has no parent. After finishing the tour and ending up in node 1, if it is black, we are done. If not, it is the only white one and we can select any child $\boldsymbol{c}$ of 1 and do $1 - c - 1 - c$. Now all nodes are black.
It is recommended to do the implementation with stack. Complexity is $O(N)$.

Problem D: Potions homework

Ronaldo, Her-my-oh-knee and their friends have started a new school year at their MDCS School of Speechcraft and Misery. At the time, they are very happy to have seen each other after a long time. The sun is shining, birds are singing, flowers are blooming, and their Potions class teacher, professor Snipe is sulky as usual. Due to his angst fueled by disappointment in his own life, he has given them a lot of homework in Potions class.
Each of the N students have been assigned a single task. Some students do certain tasks faster than others. Thus, they want to redistribute the tasks so that each student still does exactly one task, and that all tasks are finished.
Each student has their own laziness level, and each task has its own difficulty level. Professor Snipe is trying hard to improve their work ethics, so each student's laziness level is equal to their task's difficulty level.
Both sets of values are given in the array A, where $A[i]$ represents both the laziness level of the i^{th} student and the difficulty of their task. The time a student needs to finish a task is equal to the product of their laziness level and the task's difficulty.
They have asked you what is the shortest possible (total) time they must spend to finish all tasks.

Input:

The first line of input contains the integer N, which represents the total number of tasks. The next N lines contain exactly one integer each, which represents the difficulty of the task and the laziness of the student who initially received the task.

Output:

Your output should consist of only one line – the minimum time needed to finish all tasks, modulo 10007.

Constraints:

- *1 ≤ N ≤ 100,000*
- *1 ≤ A[i] ≤ 100,000*

Example input:

```
2
1
3
```

Example output:

```
6
```

Explanation:

If the students switch their tasks, they will be able to finish them in 3+3=6 time units.

```
> Time and memory limit: 0.1s / 64MB
```

Solution and analysis:

Everything is pretty simple here. What should be done here is to give the laziest student the easiest task, because it is always optimal to do so. If we do that continuously, it becomes obvious that solution is to sort the given array and get the following formula:

$$\sum_{i=1}^{N} A[i]^{*}\, A[N-i-1]$$

Problem E: Festival organization

The Prodiggers are quite a cool band and for this reason, they have been the surprise guest at the ENTER festival for the past 80 years. At the beginning of their careers, they weren't so successful, so they had to spend time digging channels to earn money; hence the name.
Anyway, they like to tour a lot and have surprising amounts of energy to do extremely long tours. However, they hate spending two consecutive days without having a concert, so they would very much like to avoid it.
The Prodiggers would like to hold K tours of length of at least L days, and at most R days. Since they are quite superstitious, they want all their tours to have the same length and different schedules (regarding playing concerts and skipping days). Additionally, they would absolutely hate to skip two consecutive days in a single tour. Since their schedule is quite busy, they want you to tell them in how many ways can they hold the K tours, modulo M.

Input:

The first and only line of input will contain 3 numbers: K, L, R

Output:

Output a single number: in how many ways can they hold the K tours, modulo M=1,000,000,007.

Constraints:

- $1 \le K \le 200$
- $1 \le L \le R \le 10^{18}$

Example input:

```
1 1 2
```

Example output:

```
5
```

```
> Time and memory limit: 1s / 256MB
```

Solution and analysis:

It is easy to prove that the answer for the query is $\sum_{n=l}^{r}\binom{F_{n+2}}{k} = \sum_{n=0}^{r}\binom{F_{n+2}}{k} - \sum_{n=0}^{l-1}\binom{F_{n+2}}{k}$*, where* F_n *are Fibonacci numbers. Let's notice, that* $\binom{x}{k}$ *is a polynomial for x and can be expressed in the form* $c_0 + c_1x + \ldots + c_kx^k$*. Knowing this we can express the answer in a different way* $\sum_{n=l}^{r}\binom{F_{n+2}}{k} = \sum_{n=0}^{r}\sum_{m=0}^{k} c_m F_{n+2}^m = \sum_{m=0}^{k} c_m \sum_{n=0}^{r} F_{n+2}^m$.
Thus, we reduced our problem to computing the sum of m*th powers of Fibonacci numbers.*

To do so we will refer to Binet's formula $F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$ *or* $F_n = \frac{\sqrt{5}}{5}(\phi^n - \psi^n)$.

$$F_n^m = \left(\frac{\sqrt{5}}{5}\right)^m (\phi^n - \psi^n)^m = \left(\frac{\sqrt{5}}{5}\right)^m \sum_{j=0}^{m}(-1)^{m-j}\binom{j}{m}\phi^{nj}\psi^{n(m-j)}$$

It reduces to the following:

$$\sum_{n=0}^{r} F_n^m = \left(\frac{\sqrt{5}}{5}\right)^m \sum_{n=0}^{r}\sum_{j=0}^{m}(-1)^{m-j}\binom{j}{m}\left(\phi^j\psi^{(m-j)}\right)^n = \left(\frac{\sqrt{5}}{5}\right)^m \sum_{j=0}^{m}(-1)^{m-j}\binom{j}{m}\sum_{n=0}^{r}\left(\phi^j\psi^{(m-j)}\right)^n$$

The inner sum is almost always a geometric progression with $b_0 = 1$ *and,* $q = \phi^j\psi^{(m-j)}$ *except for the cases when,* $\phi^j\psi^{(m-j)} = 1$ *but we may avoid any special cases by computing it in a way similar to binary exponentiation.*
Indeed, in order to compute $\sum_{n=0}^{r} q^n$*, we may start with computing* $\sum_{n=0}^{2^k-1} q^n$ *and* q^{2^k}*.Two sums with m1 and m2 elements can be merged together in the following way:* $\sum_{n=0}^{m1+m2-1} q^n = \sum_{n=0}^{m1-1} q^n + q^{m1}\sum_{n=0}^{m1+m2-1} q^n$.

Thus, we can compute the sum of m*th powers only with additions and multiplications. The only difficulty is that there is* $\sqrt{5}$ *present in these formulas (in* ϕ *and* ψ*) and there is no such number modulo* $10^9 + 7$*. The solution to this is simple: let's never specify an exact value for it. This way we will always work with numbers of the form* $a + \sqrt{5}b$*.The addition and multiplication of these numbers is fairly easy:* $(a + \sqrt{5}b) + (c + \sqrt{5}d) = (a + c) + \sqrt{5}(b + d)$,
$(a + \sqrt{5}b)(c + \sqrt{5}d) = (ac + 5bd) + \sqrt{5}(ad + bc)$;
These are the only operations that we require. The sum of Fibonacci numbers (which are integers) is also integer, so the final result will never contain any $\sqrt{5}$*. Thus, we have solved this problem.*

Problem F: Pokermon League challenge

Welcome to the world of Pokermon, yellow little mouse-like creatures, who absolutely love playing poker!
Yeah, right...
In the ensuing Pokermon League, there are N registered Pokermon trainers, and T existing trainer teams. Since there is a lot of jealousy between trainers, there are E pairs of trainers who hate each other. Their hate is mutual, there are no identical pairs among these, and no trainer hates himself (the world of Pokermon is a joyful place!). Each trainer has a wish-list of length L of teams he'd like to join. All the teams are divided into two conferences.
Your task is to divide players into teams and **the teams into two conferences, so** that:

- each trainer belongs to exactly one team
- no team is in both conferences
- total hate between conferences is at least $\frac{E}{2}$
- every trainer is in a team from his wish-list

Total hate between conferences is calculated as the number of pairs of trainers from teams from different conferences who hate each other.

Input:

The first line of input contains 2 non-negative integers:

- N - total number of Pokermon trainers
- E - number of pairs of trainers who hate each other

Each Pokermon trainer is represented by a number between $[1, N]$
The next E lines contain 2 integers A and B indicating that Pokermon trainers A and B hate each other
The next $2N$ lines are in a following format:
Starting with Pokermon trainer 1, for every trainer in consecutive order:

- first number L - a size of Pokermon trainers wish list
- in the next line are positive integers $t[i]$ - the teams Pokermon trainer would like to be on.

Each trainer's wish list will not contain repeating teams.
Teams on the wish lists are numbered in such a way that the set of all teams that appear on at least $\mathbf{1}$ wish list is set of consecutive positive integers $\{1, 2, 3, \ldots, T\}$.

Output:

Contains 2 lines:
The first line contains N numbers, specifying the team every trainer is in. First for trainer 1, then 2, etc.
The second line contains T numbers where for every team, starting with team 1, there is an integer specifying the conference (1 or 2) of that team.

Constraints:

- $4 \le N \le 50{,}000$
- $2 \le E \le 100{,}000$
- $16 \le L \le 20$
- $1 \le t[i] \le T$
- $1 \le T \le 1{,}000{,}000$
- $1 \le A, B \le N$

Example input:

```
4 3
1 2
2 3
4 1
16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 15
16
2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 18
16
2 3 4 5 6 7 8 9 10 11 12 13 14 15 18 19
16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 19
```

Example output:

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

Explanation:

Conference 1 contains only team 1, and conference 2 contains all other teams. Total hate between conferences is 2 which is greater than $\frac{E}{2} = \frac{3}{2} = 1.5$.

Pokermon trainer 1 belongs to team 1, trainers 2 and 3 to team 2 and trainer 4 to team 3. Other teams are empty but they have been assigned a conference.

> Time and memory limit: 5s / 256MB

Solution and analysis:

The key idea is to first put players into conferences and then assign them teams. We go through all of the players and put them into conferences so that the first condition is satisfied. We take them one by one and select a conference for each player such that he hates more (or equally many) players in the other conference that the one we put him in. That means that at any point of this process, there will be more hate edges connecting players in different conferences than those connecting players in the same conferences. Thus, in the end, we have satisfied first condition. Complexity is $O(N + E)$ for this part. Now we will try to satisfy the second condition by assigning teams to the players. Let $\boldsymbol{S}$ be the set of all of the teams that appear on at least one wish-list. Let's select some subset $\boldsymbol{A}$ of $\boldsymbol{S}$. We select it in a way that every element from $\boldsymbol{S}$ will be thrown in it with 50% chance. Now, we will assign teams from $\boldsymbol{A}$ to players in conference 1 and teams from $\boldsymbol{A}^c$ to players in conference 2. If such assigning is possible while satisfying the second condition, we will be done. Let's see what is the chance that a particular player in conference 1 can't be assigned a valid team from $\boldsymbol{A}$. That means none of at least 16 teams on his wish-list are in $\boldsymbol{A}$. The chance for that is $\frac{1}{2^{16}}$. Same goes for all players in conference 1 and equivalently for conference 2. So, the chance that at least one player can't be assigned a team is at most $N \cdot \frac{1}{2^{16}} \leq 0.77$. This means that there is at least 23% chance that this method will give us the final solution which fulfills both problem conditions. This means that after a several steps, we will very likely solve the problem. Complexity is $O(NL)$ for this part. For example, if we do 30 steps, the probability that we will find a correct solution is $1 - 0.7730 = 0.9996$.

Problem G: Heroes of Making Magic III

I'm strolling on sunshine, yeah-ah! And don't it feel good!
Well, it certainly feels good for our Heroes of Making Magic, who are casually walking on a one-directional road, fighting imps. Imps are weak and feeble creatures and they are not good at much. However, Heroes enjoy fighting them. For fun, if nothing else.
Our Hero, Ignatius, simply adores imps. He is observing a line of imps, represented as a zero-indexed array of integers $A[\,]$ of length N, where $A[i]$ denotes the number of imps at the i^{th} position. Sometimes, imps can appear out of nowhere.
When heroes fight imps, they select a segment of the line, start at one end of the segment, and finish on the other end, without ever exiting the segment. They can move exactly one cell left or right from their current position and when they do so, they defeat one imp on the cell that they moved to, so, the number of imps on that cell decreases by one. This also applies when heroes appear at one end of the segment, at the beginning of their walk.
Their goal is to defeat all imps on the segment, without ever moving to an empty cell in it (without imps), since they would get bored. Since Ignatius loves imps, he doesn't really want to fight them, so no imps are harmed during the events of this task. However, he would like you to tell him whether it would be possible for him to clear a certain segment of imps in the abovementioned way if he wanted to.
You are given Q queries, which have two types:

1. a b k - denotes that k imps appear at each cell from the interval $[a, b]$
2. a b - asks whether Ignatius could defeat all imps in the interval $[a, b]$ in the way described above

Input:

The first line contains a single integer, N, the length of A. The following line contains N integers, $A[i]$, the initial number of imps in each cell. The third line contains a single integer Q, the number of queries. The remaining Q lines contain one query each, with a, b and k.

Output:

For each second type of query output 1 if it is possible to clear the segment, and 0 if it is not.

Constraints:

- *$1 \le N \le 200{,}000$*
- *$1 \le Q \le 300{,}000$*
- *$0 \le A[i] \le 5{,}000$*
- *$0 \le a \le b < N$*
- *$0 \le k \le 5{,}000$*

•

Example input:

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

Example output:

```
0
1
```

Explanation:

For the first query, one can easily check that it is indeed impossible to get from the first to the last cell while clearing everything. After we add 1 to the second position, we can clear the segment, for example by moving in the following way: $0 \to 1 \to 2 \to 1 \to 0 \to 1 \to 2$

```
> Time and memory limit: 5s / 64MB
```

Solution and analysis:

For queries of type 2, we are only interested in the elements A_k with n in the interval $[i, j]$, and nothing else. For the sake of convenience, label those elements as $a_0, a_1, \dots, a_n$, where $n = j - i + 1$.
It can be easily seen that in order to clear a segment, we can just move back and forth between positions 0 and 1 until a_0 becomes 0, move to 1 and alternate between 1 and 2, etc – until we zero out the last element. If we want this procedure to succeed, several (in)equalities must hold:

- *$a_0 \geq 1$ (since we have to decrease the first element of the segment in the first step)*
- *$a_1 - a_0 \geq 0$ (after following this procedure, a_1 is decreased exactly by a_0, and we end up on a_1)*
- *$a_2 - (a_1 - a_0 + 1) = a_2 - a_1 + a_0 - 1 \geq 0$, or, $a_2 - a_1 + a_0 \geq 1$*
- *$a_3 - a_2 + a_1 - a_0 \geq 0, \dots$*
- *In general, $a_m - a_{m-1} + a_{m-2} - \dots + (-1)^m a_0$ has to be greater or equal than 0, if m is odd, and 1, if m is even*

Equivalently, if we define a new sequence d', such that $d'_0 = a_0$, and $d'_m = a_m - d'_{m-1}$, these inequalities are equivalent to stating that $d'_0, d'_2, d'_4, \dots \geq 1$ and $d'_1, d'_3, d'_5, \dots \geq 0$.
Of course, we do not have to store the d array for each pair of indices i, j, but it is enough to calculate it once, for the entire initial array A. Then, we can calculate the appropriate values of d'_k for any interval $[i, j]$ (k is the zero-based relative position of an element inside the segment): Let $c = d_{i-1}$. Then,

- *For even values of k, $d'_k = c + d_{i+k}$*
- *For odd values of k, $d'_k = d_{i+k} - c$*

Now we only need a way to maintain the array d after updates of the form „$1\ i\ j\ v$“. Fortunately, we can see that this array is updated in a very regular way:

- *Elements on even positions inside the interval $[i, j]$ are increased by v*
- *If $j - i$ is even, elements on positions $j + 1, j + 3, j + 5, \dots$ are decreased by v, and elements on positions $j + 2, j + 4, j + 6, \dots$ are increased by v*

All of this can be derived from our definition of d.
Both of these queries can be efficiently implemented using a lazy-propagation segment tree, where each internal node stores two numbers, the minimal of its odd- and even-indexed leaves.
Using the approach described above, we arrive at a solution with the complexity of $O(q \log n)$.

Problem H: Dexterina's Lab

Dexterina and Womandark have been arch-rivals since they've known each other. Since both are super-intelligent teenage girls, they've always been trying to solve their disputes in a peaceful and nonviolent way. After god knows how many different challenges they've given to one another, their score is equal and they're both desperately trying to best the other in various games of wits. This time, Dexterina challenged Womandark to a game of Nim.
Nim is a two-player game in which players take turns removing objects from distinct heaps. On each turn, a player must remove at least one object, and may remove any number of objects from a single heap. The player to remove the last object wins.
By their agreement, the sizes of piles are selected randomly from the range $[0, X]$. Each pile's size is taken from the same probability distribution.
Womandark is coming up with a brand new and evil idea on how to thwart Dexterina's plans, so she hasn't got much spare time. She, however, offered you some tips on looking fabulous in exchange for helping her win in Nim. Your task is to tell her what is the probability that the first player to play wins, given the rules as above and assuming that both players play optimally.

Input:

The first line of input contains 2 integers: N and X. The second line contains $X + 1$ real numbers, given to 6 decimal places each: $P(0), \dots, P(X)$.

Output:

Output a single real number, the probability that the first player wins. The answer will be judged as correct if it differs from the correct answer by at most 10^{-6}.

Constraints:

- *$1 \leq N \leq 10^9$*
- *$1 \leq X \leq 100$*

Example input:

```
2 2
0.5 0.25 0.25
```

Example output:

```
0.625
```

Explanation:

The correct answer is exactly 0.625. The checker will also accept, for example, outputs like 0.625000, 0.625001 and 0.625000000.

```
> Time and memory limit: 0.5s / 256MB
```

Solution and analysis:

Dynamic programming – $\mathcal{O}(xn^2)$

A well-known result from a game theory states that the winning player of a game of Nim is determined only by the bitwise XOR of the piles' sizes. The first player has a winning strategy if (and only if) this value is not zero.
Let P_i be the probability that a pile contains i objects. Define $d_{m,x}$ as the probability that the bitwise XOR of m random sizes is equal to x. The values of d can be expressed with the recurrence relation

$$d_{m,x} = \begin{cases} 1 \text{ if } m = 0 \wedge x = 0 \\ 0 \text{ if } m = 0 \wedge x \neq 0 \\ \sum_{i=0}^{N} d_{m-1,i} P_{i \oplus x} \text{ otherwise} \end{cases}$$

where N is the maximal possible value of the XOR (one less than the first power of 2 greater than the maximal pile size n) and $\oplus$ the bitwise XOR operator.
The probability that the second player wins is $d_{n,0}$. Because the first player wins if the second does not, we can calculate the values of d in $\mathcal{O}(xn^2)$ and output $1 - d_{n,0}$.

Matrix exponentiation – $\mathcal{O}(x^3 \log n)$

Let D_i be the vector $(d_{i,0} \quad d_{i,1} \quad \cdots \quad d_{i,N})^T$. The transformation that produces D_{i+1} from D_i is linear, and can therefore be expressed as $D_{i+1} = MD_i$, where M is a matrix given by $M_{i,j} = P_{i \oplus j}$.
Since the matrix multiplication is associative, $D_n = M^n D_0$ we can calculate M^n using $\mathcal{O}(\log n)$ matrix multiplications, for a total complexity to calculate D_n (which contains the solution $d_{n,0}$) of $\mathcal{O}(x^3 \log n)$.

This problem can also be solved in:

- *Vector exponentiation – $\mathcal{O}(\boldsymbol{x^2 \log n})$*
- *Faster multiplication – $\mathcal{O}(\boldsymbol{x \log x \ \log n})$*
- *Even faster multiplication – $\mathcal{O}(\boldsymbol{x \log x + \ x \log n})$*
- *But $\mathcal{O}(\boldsymbol{x^3 \log n})$ is enough to get AC.*

Problem I: R3D3's summer adventure

$R3D3$ spent some time on an internship in MDCS. After earning enough money, he decided to go on a holiday somewhere far, far away. He enjoyed sun tanning, drinking alcohol-free cocktails and going to concerts of popular local bands. While listening to "The White Buttons" and their hit song "Dacan the Baker", he met another robot for whom he was sure was the love of his life. Well, his summer, at least.
Anyway, $R3D3$ was too shy to approach his potential soulmate, so he decided to write her a love letter. However, he stumbled upon a problem. Due to a terrorist threat, the Intergalactic Space Police was monitoring all letters sent in the area. Thus, R3D3 decided to invent his own alphabet, for which he was sure his love would be able to decipher.
There are N letters in $R3D3$'s alphabet, and he wants to represent each letter as a sequence of $0s$ and $1s$, so that no letter's sequence is a prefix of another one's sequence. Since the Intergalactic Space Communications Service has lately introduced a tax for invented alphabets, R3D3 must pay a certain amount of money for each bit in his alphabet's code. He is too lovestruck to think clearly, so he asked you for help.
Given the costs C_0 and C_1 for each 0 and 1 in $R3D3$'s alphabet, respectively, you should come up with a coding for the alphabet (with properties as above) with minimal total cost.

Input:

The first line of input contains 3 integers:
N - the number of letters in the alphabet
C_0- cost of 0s
C_1 - cost of 1s

Output:

Output a single number - the minimal cost of the whole alphabet.

Constraints:

- $2 \leq N \leq 10^8$
- $0 \leq C_0 \leq 10^8$
- $0 \leq C_1 \leq 10^8$

Example input:

```
4 1 2
```

Example output:

```
12
```

Explanation:

The alphabet is "00", "01", "10", "11". So minimal total cost is 12.

> Time and memory limit: 1s / 256 MB

Solution and analysis:

Basically, the problem can be formulated in the following way: Let's build a binary tree with edges labeled '0' and '1'with N leaves so that the sum of costs of paths to all leaves is minimal. This tree is also called 'Varn code tree'*. Varn code tree is generated as follows. Start with a tree consisting of a root node from which descend 2 leaf nodes, the costs associated with the corresponding code symbols. Select the lowest cost node, let c be its cost, and let descend from it 2 leaf nodes $c + c(0)$ and $c + c(1)$. Continue, by selecting the lowest cost node from the new tree, until N leaf nodes have been created.
This greedy approach can be done in $O(NlogN)$ if we actually construct a tree. Basically, we do the following thing N times: out of all codes we have selected the lowest, deleted it and created 2 new codes by adding '0' and '1' to the one we have deleted. If all of this was done with some standard data structure this is $O(NlogN)$.
If we actually don't need the tree and the coding itself, just the final cost, we can improve to $O(log^2(N))$. Let's define an array F to represent this tree, where $F[i]$ will be a number of paths to the leaves in this tree with cost equal to i. While sum of all values in F is lower than N we do the following procedure: find an element with lowest 'i' where $F[i] > 0$. Then, $F[i + c(0)] += F[i]$; $F[i + c(1)] += F[i]$; $F[i] = 0$. Basically, we did the same thing as in the previous algorithm, just instead of adding 2 edges to the leaf with the lowest cost we did that to all of the leafs that have the lowest cost in the tree. Numbers in this array rise at least as fast as Fibonacci numbers, just the array will be sparse. So, if instead of array we use some data structure like a map, we get $O(log^2(N))$ complexity.

Problem A: Digits

John gave Jack a very hard problem. He wrote a very big positive integer A_0 on a piece of paper. The number is less than 10^{200000}. In each step, Jack is allowed to put '+' signs in between some of the digits (maybe none) of the current number and calculate the sum of the expression. He can perform the same procedure on that sum and so on. The resulting sums can be labeled respectively by A_1, A_2 etc. His task is to get to a single digit number.
The problem is that there is not much blank space on the paper. There are only three lines of space, so he can't perform more than three steps. Since he wants to fill up the paper completely, he will perform exactly three steps.
Jack must not add leading zeros to intermediate results, but he can put '+' signs in front of digit 0. For example, if the current number is 1000100, $10 + 001 + 00$ is a valid step, resulting in number 11.

Input:

In the first line, a positive integer N, representing the number of digits of number A_0.
In the second line, a string of length N representing number A_0. Each character is a digit. There will be no leading zeros.

Output:

Output exactly three lines, the steps Jack needs to perform to solve the problem. You can output any sequence of steps which results in a single digit number (and is logically consistent).

Constraints:

- *1 ≤ N ≤ 200,000*

Example input 1:

```
2
10
```

Example output 1:

```
10
1+0
1
```

Example input 2:

```
3
992
```

Example output 2:

```
99+2
1+01
2
```

Example input 3:

```
4
1234
```

Example output 3:

```
123+4
1+2+7
1+0
```

Explanation:

Example 1: In the first step, we use zero '+' signs, so $A_1 = 10$. In the second step, we place a '+' sign between 1 and 0, so $A_2 = 1 + 0 = 1$. In the third step, we don't need to (and we can't) put any '+' signs, so we get $A_3 = 1$.

Example 2: In the first step, we only put a '+' between the last two digits, so we get $A_1 = 99 + 2 = 101$. In the second step, we place the only '+' sign between the first and the second digit, so $A_2 = 1 + 01 = 1 + 1 = 2$. In the third step, we don't need to (and we can't) put any '+' signs, so we get $A_3 = 2$.

Example 3: In the first step, we use a '+' sign between the last two digits, so $A_1 = 123 + 4 = 127$. On the second step, we place a '+' sign between every two digits, so $A_2 = 1 + 2 + 7 = 10$. In the third step, we place a '+' sign between 1 and 0, so $A_3 = 1 + 0 = 1$.

```
> Time and memory limit: 1s / 256MB
```

Solution and analysis:

Let $ds(x)$ be the sum of digits of the number x.
Notice that if $ds(A_0) \leq 198$, then $ds\left(ds\left(ds(A_0)\right)\right)$ is a single digit number, therefore we have solved the case $ds(A_0) \leq 198$.
After a little more thinking, we can extend it to $ds(A_0) \leq 288$, because 289 is the smallest number which can't be reduced to a single digit number in at most two steps. This is because number 199 can be transformed to single digit in 2 steps by using following transformations $1 + 99$, $1 + 0 + 0$. Now we know how to solve $ds(A_0) <= 288$. Now we will solve $288 \leq ds(A_0) \leq 999$.
Let $A_0 = a_0 a_1 \dots a_{n-1} a_n$ be the decimal representation of A_0.
Consider:

$$X = a_0 a_1 + a_2 a_3 + \dots + a_{n-1} a_n$$
$$Y = a_0 + a_1 a_2 + \dots + a_{n-2} a_{n-1} + a_n$$

(this is for an odd n, the following results would be the same with an even n).
Now look at: $X + Y = 11ds(A_0) - 9a_n$. (this is easy to be seen by addition).
Since $ds(A_0) \geq 288$, we have $X + Y > 10ds(A_0)$. Thus, we trivially conclude that for example: $X > 5ds(A_0)$.
That means that $X > 1000$. Now look at the following sequence of numbers:

$$a_0 + a_1 + a_2 + a_3 + \dots + a_{n-1} + a_n,$$
$$a_0 a_1 + a_2 + a_3 + \dots + a_{n-1} + a_n,$$
$$a_0 a_1 + a_2 a_3 + \dots + a_{n-1} + a_n,$$
$$\dots,$$
$$X = a_0 a_1 + a_2 a_3 + \dots + a_{n-1} a_n\,.$$

It is increasing, and the first element has 3 digits, the last has at least 4, so the number of digits has increased at some point. Since the numbers in the sequence are increasing by at most 81 (trivial check), the sequence element, when digit skipping occurred, is at most 1080. Thus, our first step is to put '+' signs as they are in that sequence element. After that, we apply $ds(\,)$ two more times and we will be done (trivial check).
Now we solve $ds(A_0) \geq 1000$.
Consider:

$$X = a_0 a_1 a_2 + a_3 a_4 a_5 + \dots + a_{n-2} a_{n-1} a_n$$
$$Y = a_0 + a_1 a_2 a_3 + a_4 a_5 a_6 + \dots + a_{n-4} a_{n-3} a_{n-2} + a_{n-1} + a_n$$
$$Z = a_0 + a_1 + a_2 a_3 a_4 + a_5 a_6 a_7 + \dots + a_{n-3} a_{n-2} a_{n-1} + a_n.$$

(this is for the n of the form $n = 3k + 2$, the following results would be the same with the other n)
As before, we can easily see that $X + Y + Z > 98ds(A_0)$ ($90ds(A_0)$ is enough), so, for example, $X > 30ds(A_0)$.
Now look at the following sequence of numbers:

$$a_0 + a_1 + a_2 + a_3 + \dots + a_{n-1} + a_n,$$
$$a_0 a_1 a_2 + a_3 + a_4 + \dots + a_{n-1} + a_n,$$
$$a_0 a_1 a_2 + a_3 a_4 a_5 + \dots + a_{n-2} + a_{n-1} + a_n,$$
$$\dots,$$
$$X = a_0 a_1 a_2 + a_3 a_4 a_5 + \dots + a_{n-2} a_{n-1} a_n$$

It is increasing and since the first number is $ds(A_0)$ and the last one is at least $30ds(A_0)$, the digit skipping occurred somewhere.

Let $M = a_0a_1a_2 + a_3a_4a_5 + \ldots + a_{3k}a_{3k+1}a_{3k+2} + a_{3k+3} + \cdots + a_{n-2} + a_{n-1} + a_n$ *be the sequence element right before the digit skipping occurred. Therefore, it is less than* $10ds(A_0)$ *and at most* 999 *away from power of* 10*. Now we continue by 'steps of two', so the next element is:*

$a_0a_1a_2 + a_3a_4a_5 + \ldots + a_{3l}a_{3k+1}a_{3k+2} + (a_{3k+3}a_{3k+4} + a_{3k+5}) + \cdots + a_{n-2} + a_{n-1} + a_n,$

and the next one:

$a_0a_1a_2 + a_3a_4a_5 + \ldots + a_{3l}a_{3k+1}a_{3k+2} + (a_{3k+3}a_{3k+4} + a_{3k+5}) + (a_{3k+6}a_{3k+7} + a_{3k+8}) + \cdots + a_{n-2} + a_{n-1} + a_n,$

Remember that $M < 10ds(A_0)$*. Since* $X > 30ds(A_0)$*, if we continued making steps of three, we would increase our number by at least* $20ds(A_0)$ *before the end of the procedure. Our steps are steps of two, and each step increases the number by at least* $1/11$ *of what it would be increased by if we were making steps of three (trivial check). Thus, before the end of our procedure, we will increase M by more than* $ds(A_0)$*. Since* $ds(A_0) \geq 1000$*, and remember that* M *was at most* 999 *less than the power of* 10*, the digit skipping will indeed occur.*

We thus reach the similar situation as in the previous case and we know what our first step is. We only need to apply $ds()$ *two more times after that.*

Problem B: Neural Network Country

Due to the recent popularity of the *Deep learning* new countries are starting to look like *Neural Networks*. That is, the countries are being built *deep* with many layers, each layer possibly having many *cities*. They also have *one* entry, and *one* exit point. There are exactly L layers, each having N cities. Let us look at the two adjacent layers L_1 and L_2. Each city from the layer L_1 is connected to each city from the layer L_2 with the travelling cost c_{ij} for $i, j \in (1, 2, .., N)$, and each pair of adjacent layers has the same cost in between their cities as any other pair (they just stacked the same layers, as usual). Also, the travelling costs to each city from the layer L_2 are same for all cities in the L_1, that is c_{ij} is the same for $i \in (1, 2, .., N)$, and fixed j. *Doctor G.* needs to speed up his computations for this country so he asks you to find the number of paths he can take from *entry* to *exit* point such that his travelling cost is divisible by given number M.

Input:

The first line of input contains N – the number of cities in each layer, L – the number of layers, and M. Second, third and fourth line contain N integers denoting costs from *entry* point to the first layer, costs between adjacent layers as described above, and costs from the last layer to the *exit* point.

Output:

Output a single integer, the number of paths *Doctor G.* can take which have total cost divisible by M, modulo $10^9 + 7$.

Constraints:

- $1 \le N \le 10^6$
- $2 \le L \le 10^5$
- $2 \le M \le 100$
- $0 \le costs < M$

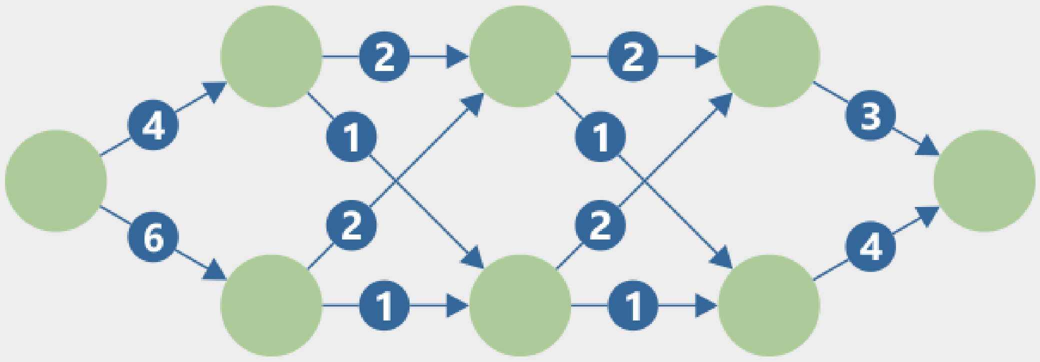
Example input:

```
2 3 13
4 6
2 1
3 4
```

Example output:

```
2
```

Explanation:



This is a country with 3 layers, each layer having 2 cities. Paths 6 → 2 → 2 → 3, and 6 → 2 → 1 → 4 are the only paths having total cost divisible by 13. Notice that input edges for layer cities have the same cost, and that they are same for all layers.

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

Let's ignore constraints for a second and try to solve the problem using a classical dynamic approach. We can easily see that if we know in how many ways we can reach some layer for every modulo up to M, we can simply calculate number of ways we can reach the next layer for every modulo by iterating over weights in $O(N)$. That means we have matrix $Dp_{L\times M}$, where $Dp[i][j]$ is equal to the number of roads up to i-th layer whose total cost is equal to j modulo m. The transition between the layers is calculated as follows:

$$Dp[i][j] = \sum_{k=0}^{N-1} Dp[i-1][(j-w[k]) \bmod M],$$

where w is a zero-base indexed array of weights between the layers. Given the array of costs between the starting point and the first layer, a, the base is calculated as $Dp[0][a[k]] \mathrel{+}= 1$, for every k from 0 to $N-1$. The last layer is a bit tricky. In order to calculate the final result, we need to know not only $Dp[L][0..N-1]$, but also which of those roads end in which nodes. We can achieve that by calculating Dp up to the layer $L-1$, and computing the last step manually, by "merging" the costs of the last two sets of edges, by simply adding $w[i]$ and $b[i]$, where b is the array of edge costs between the last layer and the finish point, for every i from 0 to $N-1$, and outputting the following result modulo Q:

$$res = \sum_{i=0}^{N-1} Dp[L-1][(-w[i]-b[i]) \bmod M],$$

giving the overall time complexity of $O(L\cdot N\cdot M)$, and memory complexity of $O(L\cdot M)$, both of them exceeding the given limits. Of course, memory is not a problem if we see that we need not to save the whole matrix, but only the last column. We will reduce time complexity step by step. If we notice that it doesn't matter in which node we are currently in (except in the last step), we can speed up the computation by saving the number of occurrences of each $w[i]$ modulo M into an array $num[i]$. Now Dp matrix is calculated in $O(L\cdot M^2)$ using formula:

$$Dp[i][j] = \sum_{k=0}^{M-1} num[k]\cdot Dp[i-1][(j-k) \bmod M].$$

Now, a keen eye can spot multiplication of a matrix and a vector by looking at this formula. Indeed, we can compute a transition matrix of dimensions $M\times M$ which, multiplied by a vector of the current state (number of roads which have the total cost $i = 0..M-1$, modulo M up to some layer), gives us the next state vector, corresponding to the next layer, resulting in time complexity $O(M^3\cdot \log L)$, using modular matrix exponentiation. The last layer still has to be treated separately, in the same way as described before.

Problem C: Property

Bill is a famous mathematician in BubbleLand. Thanks to his revolutionary math discoveries he was able to make enough money to build a beautiful house. Unfortunately, for not paying property tax on time, court decided to punish Bill by making him lose a part of his property.

Bill's property can be observed as a convex regular $2n$-sided polygon $A_0A_1 \dots A_{2n-1}A_{2n},\ A_{2n} = A_0$ with sides of the **exactly** 1 meter in length.

Court rules for removing part of his property are as follows:

Split every edge $A_kA_{k+1}, k = 0\dots 2n-1$ in n **equal parts** of size $\frac{1}{n}$ with points $P_0, P_{1,} \dots P_{n-1}$

On every edge $A_{2k}A_{2k+1}, k = 0\dots n-1$ court will choose **one point** $B_{2k} = P_i$ for some $i = 0, \dots, n-1$ such that $\bigcup_{i=0}^{n-1} B_{2i} = \bigcup_{i=0}^{n-1} P_i$

On every edge $A_{2k+1}A_{2k+2}, k = 0\dots n-1$ Bill will choose **one point** $B_{2k+1} = P_i$ for some $i = 0, \dots, n-1$ such that $\bigcup_{i=0}^{n-1} B_{2i+1} = \bigcup_{i=0}^{n-1} P_i$

Bill gets to keep property inside of $2n$-sided polygon $B_0B_1 \dots B_{2n-1}$

Luckily, Bill found out all B_{2k} points the court chose. Even though he is a great mathematician, his house is very big and he has a hard time calculating. Therefore, he is asking you to help him choose points in order to maximize his property area.

Input:

The first line contains one integer number n representing number of edges of Bill's $2n$-sided polygon house.

The second line contains n distinct integer numbers $B_{2k}, k = 0\dots n-1$, separated by a single space, representing points the court chose. If $B_{2k} = i$, the court chose point P_i on side $A_{2k}A_{2k+1}$.

Output:

Output contains n distinct numbers separeated by a single space representing points Bill shoud choose in order to maximize the property area. If there are multiple solutions that maximize the area, return any solution which maximizes the area.

Constraints:

- $2 \le n \le 50{,}000$
- $0 \le B_{2k} \le n-1,\ k = 0 \dots n-1$
- $\bigcup_{i=0}^{n-1} B_{2i} = \{0, 1, 2, \dots, n-1\}$

Example input:

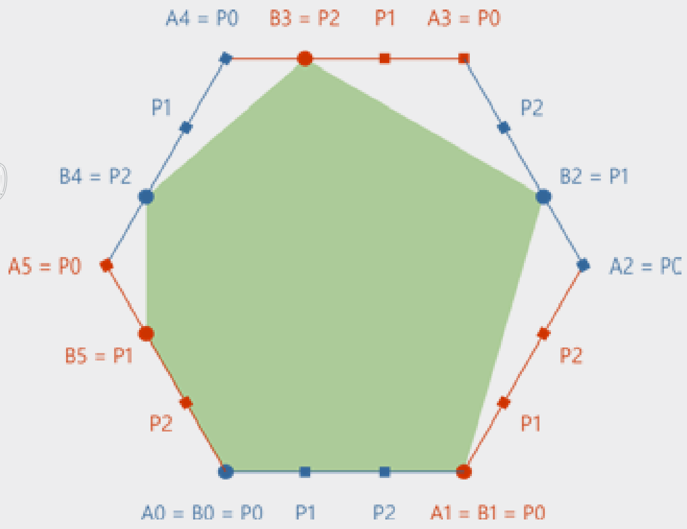
```
3
0 1 2
```

Example output:

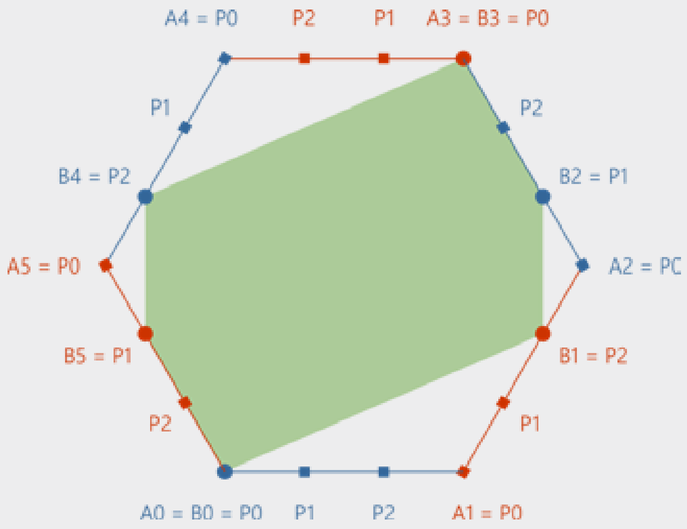
```
0 2 1
```

Explanation:

Court chose points $B_0 = P_0$, $B_2 = P_1$ and $B_4 = P_2$ as described in the image below.
If Bill chooses points $B_1 = P_0$, $B_3 = P_2$ and $B_5 = P_1$ he will achieve the maximum area as shown in the image below:



On the other hand, if Bill chooses points $B_1 = P_2$, $B_3 = P_0$ and $B_5 = P_1$, the area of his green property will be smaller:



```
> Time and memory limit: 1s / 256MB
```

Solution and analysis:

Let's first figure out what is the easiest way to calculate the area of the $B_0B_1 \dots B_{2n-1}$.
We can easily determine the area of of $B_0B_1 \dots B_{2n-1}$ polygon by subtracting a purple triangle area from the initial $A_0A_1 \dots A_{2n-1}$ polygon (Figure 1).

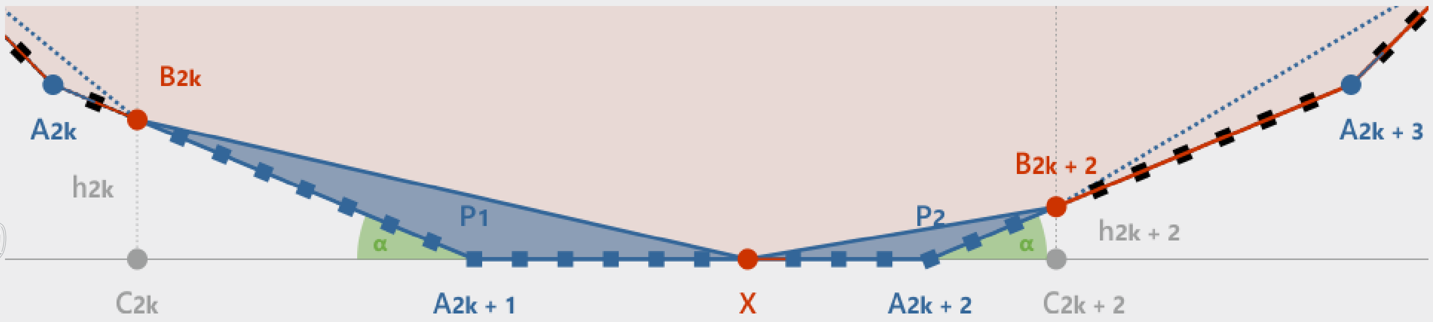


Figure 3

Determining the maximum polygon area $B_0B_1 \dots B_{2n-1}$ we can achieve is equivalent to determining the minimum purple triangle area we can achieve, which is a much easier task.
Let's say the court chose points B_{2k} and B_{2k+2} (Figure 1). We now need to determine our point $X = B_{2k+1}$ to minimize sums of areas P_1 and P_2.

$$P_1 + P_2 = \frac{A_{2k+1}X * B_{2k}C_{2k}}{2} + \frac{XA_{2k+2} * B_{2k+2}C_{2k+2}}{2}$$
$$P_1 + P_2 = \frac{x * B_{2k}C_{2k}}{2} + \frac{(s-x) * B_{2k+2}C_{2k+2}}{2}$$
$$P_1 + P_2 = \frac{s * B_{2k+2}C_{2k+2}}{2} + \frac{x * (B_{2k}C_{2k} - B_{2k+2}C_{2k+2})}{2}$$
$$P_1 + P_2 = const + \frac{x * (h_{2k} - h_{2k+2})}{2}$$

Therefore, to minimize the total triangle area, we need to choose right x for $h_{2k} - h_{2k+2}$. Because our points represent a permutation of $\{P_0, P_1, \dots, P_n\}$, best way to choose points will be to take minimum x when $h_{2k} - h_{2k+2}$ is maximal, and maximum x, when $h_{2k+1} - h_{2k+2}$ is minimal.
Writing algorithm is now easy. For polygon create the array $diffh[k] = (h_{2k} - h_{2k+2}, k)$, sort it in an ascending order and give to $sortedDiffh[k].edge$ point P_{n-k-1}.

The overall algorithm complexity is $O(NlogN)$.

Problem D: Exploration plan

The competitors of Bubble Cup X gathered after the competition and discussed what is the best way to get to know the host country and its cities.
After exploring the map of Serbia for a while, the competitors came up with the following facts: the country has V cities which are indexed with numbers from 1 to V, and there are E bi-directional roads that connect the cites. Each road has a weight (the time needed to cross that road). There are N teams at the Bubble Cup and the competitors came up with the following plan: each of the N teams will start their journey in one of the V cities, and some of the teams share the starting position.
They want to find the **shortest** time T, such that every team can move in these T minutes, and the number of **different** cities they **end up in** is at least K (because they will only get to know the cities they end up in). A team doesn't have to be on the move all the time, if they like it in a particular city, they can stay there and wait for the time to pass.
Please help the competitors to determine the shortest time T so it's possible for them to end up in at least K different cities or print -1 if that is impossible no matter how they move.
Note that there could exist multiple roads between some cities.

Input:

The first line contains four integers: V, E, N and K, the number of cities, the number of roads, the number of teams and the smallest number of different cities they need to end up in. The second line contains N integers, the cities where the teams start their journey. The next E lines contain information about the roads in following format: $A_i\ B_i\ T_i$, which means that there is a road connecting cities A_i and B_i, and you need T_i minutes to cross that road.

Output:

Output a single integer that represents the **minimal** time the teams can move for, such that they **end up in** at least K different cities or output -1 if there is no solution.

Constraints:

- *$1 \le V \le 600$*
- *$1 \le E \le 20{,}000$*
- *$1 \le N \le 200$*
- *$1 \le K \le N$*
- *$1 \le A_i, B_i \le V$*
- *$1 \le T_i \le 10{,}000$*
- *The result will be no greater than 1731311 if the solution exists*

Example input:

```
6 7 5 4
5 5 2 2 5
1 3 3
1 5 2
1 6 5
2 5 4
2 6 7
3 4 11
3 5 3
```

Example output:

```
3
```

Explanation:

Three teams start from city 5, and two teams start from city 2. If they agree to move for 3 minutes, one possible situation would be the following: Two teams in city 2, one team in city 5, one team in city 3 , and one team in city 1. And we see that there are four different cities the teams end their journey at.

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

The first step we should do is to precompute the All-pairs shortest path table. We can do this with Dijkstra's algorithm from every node in time $O(V^2 logE)$ or with Floyd Warshall in time $O(V^3)$.
The next step is to notice that we can use a binary search to find the answer, because if the teams end up in at least K cities in some time T, they can do that in every greater time (they can just remain in the same cities as in the time T).
The final step to our solution is to create a function $bool\ can(int\ T)$, that for some time T, which we guess in our binary search, tells if the requirements are met (at least K different cities). This can be solved as follows: We will create a bipartite graph, where on the left side we have every starting city of our teams and on the right side the remaining cities. Then for every starting city on the left, we will create an edge to every city on the right that can be reached within time T (here we use our APSP table). Now for some fixed time T, we have a bipartite graph with starting cities on the left, that have an edge to every city they can reach within time T. After we have this graph, it's not hard to see that we now have a maximum bipartite matching problem. We just have to check whether the MBP is greater than or equal to K. Our bipartite graph will have at most N vertices on the left, and at most NV edges, so if we use Ford Fulkerson algorithm for matching the time complexity of this part will be $O(N^2V)$.
The total time complexity is: $O(V^2 logE + N^2V\ log(maxAnswer))$

Problem E: Casinos and travel

John has just bought a new car and is planning a journey around the country. Country has N cities, some of which are connected by bidirectional roads. There are $N - 1$ roads and every city is reachable from any other city. Cities are labeled from 1 to N.
John first has to select from which city he will start his journey. After that, he spends one day in a city and then travels to a randomly choosen city which is directly connected to his current one and which he has not yet visited. He does this until he can't continue obeying these rules.
To select the starting city, he calls his friend Jack for advice. Jack is also starting a big casino business and wants to open casinos in some of the cities (max 1 per city, maybe nowhere). Jack knows John well and he knows that if he visits a city with a casino, he will gamble exactly once before continuing his journey.
He also knows that if John enters a casino in a good mood, he will leave it in a bad mood and vice versa. Since he is John's friend, he wants him to be in a good mood at the moment when he finishes his journey. John is in a good mood before starting the journey.
In how many ways can Jack select a starting city for John and cities where he will build casinos such that no matter how John travels, he will be in a good mood at the end? Print answer $mod\ 10^9 + 7$.

Input:

In the first line, a positive integer N, the number of cities.
In the next N - 1 lines, two numbers a, b separated by a single space meaning that cities a and b are connected by a bidirectional road.

Output:

Output one number: the number of ways Jack can make his selection mod $10^9 + 7$.

Constraints:

- *$1 \le N \le 200{,}000$*
- *$1 \le a, b \le N$*

Example input 1:

```
2
12
```

Example output 1:

```
4
```

Example input 2:

```
3
1 2
2 3
```

Example output 2:

```
10
```

Explanation:

Example 1: If Jack selects city 1 as John's starting city, he can either build 0 casinos, so John will be happy all the time, or build a casino in both cities, so John would visit a casino in city 1, become unhappy, then go to city 2, visit a casino there and become happy and his journey ends there because he can't go back to city 1. If Jack selects city 2 for start, everything is symmetrical, so the answer is 4.

Example 2: If Jack tells John to start from city 1, he can either build casinos in 0 or 2 cities (total 4 possibilities). If he tells him to start from city 2, then John's journey will either contain cities 2 and 1 or 2 and 3. Therefore, Jack will either have to build no casinos, or build them in all three cities. With other options, he risks John ending his journey unhappy. Starting from 3 is symmetric to starting from 1, so in total we have $4 + 2 + 4 = 10$ options.

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

After examining the problem, it is easy to see that the statement can be reduced to:
Given a tree, in how many ways can you select a root and color every node black or white such that all paths from root to any leaf node (except the root) have even number of black nodes.
Assume we have selected a root and color the tree arbitrarily. Select any path from root to some leaf. Notice that whatever the number of black nodes is on this path, the parity can be adjusted by changing the color of the corresponding leaf if needed. Therefore, for every path mentioned in the problem, its parity is ultimately determined by the leaf node color and all the other nodes can be colored in any way. Therefore, the number of colorings for a fixed root is $2^{N-|leaves|}$*. We only need to count, for every root, how many leaves are there. That is easy. If a root is a leaf itself, the number of leaves of such rooted tree is the number of leaves of our unrooted tree, minus the root, otherwise it is just the number of leaves of the unrooted tree.*
The explicit formula is (N *is the number of nodes and* L *the number of leaves in the rooted tree):*

$$S = L * 2^{N-L+1} + (N - L) * 2^{N-L}$$

Problem F: Product transformation

Consider an array A with N elements, all being the same number a. Define the product transformation as a *simultaneous* update $A_i = A_i * A_{i+1}$, that is multiplying each element to the element right to it for $i \in (1, 2, .., N-1)$, with the last number A_N remaining the same. For example, if we start with an array A with $a = 2$ and $N = 4$ after one product transformation $A = [4, 4, 4, 2]$, and after two product transformations $A = [16, 16, 8, 2]$. Your simple task is to calculate the array A after M product transformations. Since the numbers can get quite big you should output them modulo Q.

Input:

The first and only line of input contains four integers N, M, a, Q.

Output:

You should output the array A from left to right, space separated.

Constraints:

- *$7 \leq Q \leq 10^9 + 123$*
- *The multiplicative order of a number a modulo Q, $\phi(a, Q)$ is prime.*
- *$1 \leq M, N < \phi(a, Q) \leq 10^6 + 123$*
- *$2 \leq a \leq 10^6 + 123$*

Example input:

```
2 2 2 7
```

Example output:

```
1 2
```

Explanation:

After 2 transformations A = [8, 2] mod 7 = [1, 2].

Note:

The multiplicative order of a number a modulo Q $\phi(a, Q)$, is the smallest natural number x such that a^x mod Q = 1. For example, $\phi(2, 7) = 3$.

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

For example, let's consider an array $A = [a, a, a, a, a], (N = 5)$. *After four product transformations that array becomes* $A = [a^{16}, a^{15}, a^{11}, a^5, a]$. *By writing only one example, one cannot see the pattern that easily. If we write out multiple examples for different* $M's$, *we might notice that the differences of exponents have somewhat familiar structure. In this case, written from left to right, we have* $1\ 4\ 6\ 4$, *and that resembles binomials, right? Indeed, it is not hard to prove mathematically, relying on recurrent formula* $C_k^n = C_{k-1}^{n-1} + C_k^{n-1}$, *that after* M *product transformations* i*-th element of zero-base indexed array* A *is:*
$A_i = a^{C_0^M + C_1^M + \cdots + C_{N-i-1}^M}$.

Given that the multiplicative order of number a *modulo* Q, $p = \phi(a, Q)$ *is prime, we can speed up the computation by using Sieve of Eratosthenes for finding primes in* $O(p \cdot \log\log p)$, *and asking if* $a^x \bmod Q = 1$ *in* $O(\log x)$. *If we approximate the prime-counting function* $\pi(x)$ *with* $\pi(x) \approx \frac{x}{\ln x}$, *it gives us an overall complexity of* $O(p + p \cdot \log\log p)$ *which is a bit faster than a solution which doesn't use Sieve of Eratosthenes, which runs in* $O(p \cdot \log p)$ *with a big multiplicative constant, knowing that calculating modulus is expensive.*

Now, let us denote $D_i = C_0^M + C_1^M + \cdots + C_{N-i-1}^M$, *for simplicity. Given the number* p *which we have previously determined as* $p = \phi(a, Q)$, *it stands:*
$a^{D_i} \bmod Q = a^{D_i \bmod p} \bmod Q$.

Since p *is also a prime number, we can find each* C_k^n *in* $O(1)$ *if we precompute all factorials and their inverses up to* N *(which can be done in* $O(N \cdot \log N)$, *given that* p *is prime and bigger than* N, *as stated in the constraints). Now we can find all* $D_i's$ *easily, in linear time. We also do a standard modular exponentiation algorithm to output final result in each iteration, yielding total complexity of*
$O(p \cdot \log\log p + N \cdot \log N + N \cdot \log p)$.

Problem G: Bathroom terminal

Smith wakes up at the side of a dirty, disused bathroom, his ankle chained to pipes. Next to him is tape-player with a hand-written message “Play Me”. He finds a tape in his own back pocket. After putting the tape in the tape-player, he sees a key hanging from a ceiling, chained to some kind of a machine, which is connected to the terminal next to him. After pressing a Play button a rough voice starts playing from the tape:

“Listen up Smith. As you can see, you are in pretty tough situation and in order to escape, you have to solve a puzzle. You are given N strings which represent words. Each word is of the maximum length L and consists of characters 'a'-'e'. You are also given M strings which represent patterns.
Pattern is a string of length $\leq\ L$ and consists of characters 'a'-'e' as well as the maximum 3 characters $'?'$. Character $'?'$ is an unknown character, meaning it can be equal to any character 'a'-'e', or even an empty character.
For each pattern find the number of words that matches with the given pattern. After solving it and typing the result in the terminal, the key will drop from the ceiling and you may escape.
Let the game begin.”

Help Smith escape.

Input:

The first line of input contains two integers N and M, representing the number of words and patterns, respectively. The next N lines represent each word, and after those N lines, following M lines represent each pattern.

Output:

Output contains M lines and each line consist of one integer, representing the number of words that match the corresponding pattern.

Constraints:

- *1 ≤ N ≤ 100,000*
- *1 ≤ M ≤ 5,000*
- *0 ≤ L ≤ 50*

Example input:

```
3 1
abc
aec
ac
a?c
```

Example output:

```
3
```

Explanation:

If we switch '?' with 'b', 'e' and with empty character, we get 'abc', 'aec' and 'ac' respectively.

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

Let's first put all given words in trie of maximum depth L in $O(N * L)$ time. Once this is done every node in the trie will contain how many words end at that node.
Now for each pattern we need to check how many words inside the trie satisfy the pattern. For every character 'a'-'e' in pattern we iterate though trie character by character. When we reach '?' we need to recursively sum the count of all possible letter choices, by this you have to recursively process all children of the current trie node, by continuing iteration. For the empty character, you have to stay at current trie node, but process the next character in the pattern, also beware of patterns with multiple consecutive '?' such as 'a???b' or 'a??' and patterns like '?aaaa?', with same caracters between two '? ', because while searching the trie, you may find same words multiple times. To solve this, for each pattern make a set of trie node pointers, which point to end nodes of found words for current pattern, so when you find a word next time, first check if the word is in the set, before counting it.
As there can be maximum 3 '?' character in a pattern, calculating how many strings in a trie satisfy a pattern will be performed $O(5\char`^3 * M * L)$.
Also, there is a bit slower solution. Use lexicographic sort on input words, generate all possible words from every pattern, and search every generated word in the sorted word vector. This solution is slower by O(logN).

Problem H: Bob and stages

The citizens of BubbleLand are celebrating thier 10th anniversary so they decided to organize a big music festival. Bob got a task to invite N famous singers who would sing on the fest. He was too busy placing stages for their performances that he totally forgot to write the invitation e-mails on time, and unfortunately, he only found K available singers. Now there are more stages than singers leaving some of the stages empty. Bob would not like if citizens of BubbleLand noticed empty stages and found out that he was irresponsible.
Because of that he decided to choose exactly K stages that form a **convex** set, make large posters as edges of that convex set and hold festival inside. While those large posters will make it impossible for citizens to see empty stages outside Bob still needs to make sure they don't see any of the empty stages inside that area. Since lots of people are coming, he would like that the festival area is as large as possible. Help him calculate the **maximum** area that he could obtain respecting the conditions. If there is no such area, the festival cannot be organized, and the answer is 0.00.

Input:

The first line of input contains 2 integers N and K, separated with one empty space, representing number of stages and number of singers, respectively. Each of the next N lines contain 2 integers X_i and Y_i, the coordinates of the stages.

Output:

Output contains only one line with one number, rounded to two decimal points: the maximal festival area.

Constraints:

- *$3 \le N \le 1{,}000$*
- *$3 \le K \le \min(N, 50)$*
- *$0 \le X_i, Y_i \le 10^9$*
- *There are no three or more collinear points*

Example input:

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

Example output:

```
10.00
```

Explanation:

From all possible convex polygon with 4 vertices and no other vertex inside, the largest is one with points (0, 0), (2, 1), (4, 4) and (1, 5).

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

For each point p from the given points we will find the maximal area of all convex K-gons which have p as leftmost vertex. After setting point p and removing all points to the left of p, we sort the rest of the points by angle around p. If we connect the points in the order they are sorted in, and the first and the last point with p, we get a star-shaped polygon P_p. Each convex polygon that has p as leftmost vertex must lie inside P_p. Next, we compute the visibility graph VG_p in such a polygon and use dynamic programming to find the polygon with maximum area.
We will construct the visibility graph during one counter-clockwise scan around the polygon. Let's say that we have M points in P_p different from p. We won't include point p in visibility graph. When we visit p_i we construct all incoming edges of p_i. With each vertex p_i we maintain a queue Q_i that stores the starting points of some of the incoming edges of p_i in counter-clockwise order. It contains those points p_j such that ji is an edge of the visibility graph and we have not yet reached another point p_k with $k > i$ such that jk is an edge of the visibility graph. The following pseudo code describes the algorithm for computing the visibility graph in a star-shaped polygon.

```
procedure CreateVisibilityGraph
        for i := 1 to M do Qᵢ  := ∅ end
        for i := 1 to M - 1 do Proceed(i, i + 1) end
procedure Proceed(i, j)
        while Qᵢ ≠ ∅ and IsLeftTurn(Front(Qᵢ)i, ij) do
                Proceed (FRONT(Qᵢ), j)
                Pop(Qᵢ)
        end
        Connect(i, j)
        Push(i, Qⱼ)
```

Time complexity for computing visibility graph is O(|VG|) since every call of the Proceed adds one edge to the VG.

Next, we will use dynamic programming over the visibility graph for computing the maximal area of all polygons with p as leftmost vertex. Each of these polygons is uniquely determined by one convex chain, a subset of the VG, which has K-2 edges (obtained by removing two edges from p). For each edge e of the VG, and for each d, $1 \le d \le m$-2 we will determine $DP[e][d]$ - the maximum area of all polygons whose corresponding convex chain starts with e and which is d edges long. We will treat the vertices clockwise. Assume that we are at some vertex p_i. Let the incoming edges of p_i be in_1,..., in_{inmax} and the outgoing edges out_1,..., out_{outmax} both ordered counter-clockwise by angle. Note that the algorithm for computing the visibility graph inside P gives us the edges in this order. For all outgoing edges we know the maximal areas of polygons which corresponding chain starts there.

We will treat the incoming edges in the reversed order, starting at $inmax$. *For this first incoming edge we look at all outgoing edges that form a convex angle with it. Let* m_{d-1} *be the maximal value of* $DP[o][d-1]$ *among them. Then* $DP[inmax][d] = m + AreaOfTriangle(p, inmax)$. *Clearly, all outgoing edges that form a convex angle with* (i, j) *form a convex angle with* $(i, j\text{-}1)$. *Hence, we don't have to check them again. Finding the maximal area takes time* $O(K * |VG|)$, *since we look at each edge twice. The overall complexity of the algorithm is* $O(KN^3)$.

References:
[1] David P. Dobkin, Herbert Edelsbrunner, Mark H. Overmars, Searching for Empty Convex Polygons (1988)

Problem I: Dating

This story is happening in a town named BubbleLand. There are n houses in BubbleLand. In each of these n houses lives a boy or a girl. People there really love numbers, and everyone has a favorite number f. That means that the boy or the girl that lives in the $i-th$ house has a favorite number equal to f_i .
The houses are numerated with numbers from 1 to n.
The houses are connected with $n-1$ bi-directional roads and you can travel from any house to any other house in the town. There is exactly one path between every pair of houses.
A new dating agency had opened their offices in BubbleLand and the citizens were very excited. They immediately sent q questions to the agency and each question was in the following format:
a b – asking how many ways there are to choose a couple (boy and girl) that have the same favorite number and live in one of the houses on a unique path from house a to house b.
Help the dating agency answer the questions and grow their business.

Input:

The first line contains an integer n, the number of houses in the town.
The second line contains n integers, where the i-th number is 1 if a boy lives in the i-th house or 0 if a girl lives in the i-th house.
The third line contains n integers, where the i-th number represents the favorite number f_i of the girl or the boy that lives in the i-th house.
The next $n-1$ lines contain information about the roads and the i-th line contains two integers a_i and b_i which means that there exists a road between these two houses.
The following line contains an integer q, the number of questions.
Each of the following q lines represents a question and consists of two integers a and b.

Output:

For each of the q questions output a single number, the answer to the citizens' questions.

Constraints:

- $1 \le n \le 10^5$
- $1 \le q \le 10^5$
- $1 \le f_i \le 10^9$

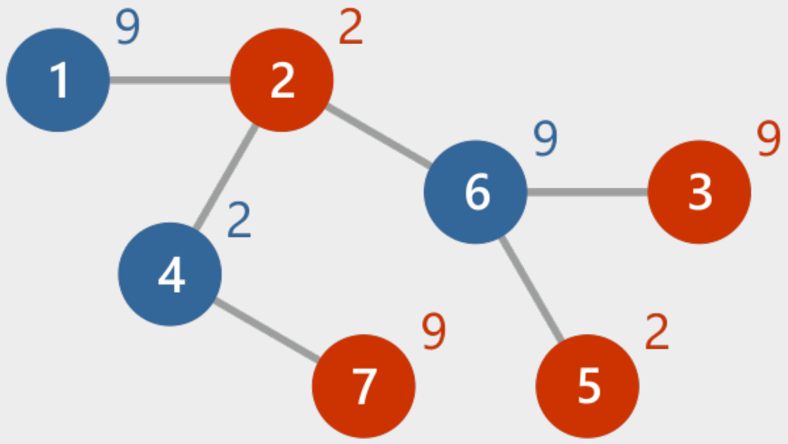
Example input:

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

Example output:

```
2
3
```

Explanation:



Blue nodes represent houses where boys live and pink those where girls live, and the numbers beside nodes represent their favorite numbers.
In the first question from house 1 to house 3, the potential couples are: (1, 3) and (6, 3).
In the second question from house 7 to house 5, the potential couples are: (7, 6), (4, 2) and (4, 5).

```
> Time and memory limit: 2s / 256MB
```

Solution and analysis:

This problem is solved with Mo's algorithm on tree.
Flatten the tree in an array by doing a modified DFS preorder traversal. For every node, we must calculate $ST[u]$ and $EN[u]$. $ST[u]$ represents the start time when we entered the node u in our DFS and $EN[u]$ represents the time when we finished exploring the node u and its subtree.
$ST[u]$ and $EN[u]$ will give us the position of the node u in the flattened array.
Now when we must answer query $u - v$, we split the path $u - LCA$ and $LCA - v$. Core of the algorithm is to see what ranges in our arrays ST and EN we should consider:
Case 1: $LCA == u$:
In this case, our query range would be $[\,ST[u],\ ST[v]\,]$.
Case 2: $LCA\,! = u$
In this case, our query range would be $[\,EN[u],\ ST[v]\,]\ +\ [\,ST[LCA],\ ST[LCA]\,]$.
We should ignore every node that appears twice or zero on that range (if it appears twice that means we finished processing that node and it isn't a part of that path). With all this considered we can solve the problem with Mo's algorithm by decomposing the queries in buckets of the size $sqrt(n)$.
While moving L and R pointers we add and remove nodes and calculate answer on the fly with a counter array for nodes with girls and for nodes with boys. For example, when we add a node with a boy, we can do the following: $res \mathrel{+}= countGirl[favoriteNumber]$. Similarly, for removing nodes from range. Time complexity $O(n\,sqrt(n))$

References:
[1] Mo's Algorithm on Trees: http://codeforces.com/blog/entry/43230