Problem A. Unimodal or Multimodal?

Input file:	`unimulti.in`
Output file:	`unimulti.out`
Time limit:	2 seconds
Memory limit:	512 megabytes

You are given a function of two integer variables, defined on the $[-10; 10] \times [-10; 10]$ square. Determine whether it has:

- multiple local maxima;
- multiple local minima;
- some plateaus.

For the discrete domain, a function f is said to have a *local minimum* at (x, y) if the following statements hold simultaneously:

- $f(x, y) < f(x - 1, y)$ or the function is not defined at $(x - 1, y)$;
- $f(x, y) < f(x + 1, y)$ or the function is not defined at $(x + 1, y)$;
- $f(x, y) < f(x, y - 1)$ or the function is not defined at $(x, y - 1)$;
- $f(x, y) < f(x, y + 1)$ or the function is not defined at $(x, y + 1)$.

A *local maximum* is defined in a similar way, only the inequalities are reversed. A function f is said to have a *plateau* at (x, y) if at least one of the following statements holds:

- $f(x, y) = f(x - 1, y)$;
- $f(x, y) = f(x + 1, y)$;
- $f(x, y) = f(x, y - 1)$;
- $f(x, y) = f(x, y + 1)$.

Input

The input file consists of one line, which contains the definition of the function to test.

The function is a polynomial of two variables, x and y. It consists of one or several monomials, which are separated by either a minus sign or a plus sign. The first monomial can also be prepended by a plus or a minus sign.

Each monomial consists of an integer (ranging from 1 to 9) representing the constant factor, followed by `x`, followed by a hat `^`, followed by an integer (ranging from 0 to 8) representing the degree of x, followed by a similar description of the second variable y.

A power of 1 might be omitted together with the hat. A power of 0 might be omitted together with the hat and the variable name. A constant factor of 1 might be omitted, except for the cases when both variables are omitted. Please check the examples for what is possible.

The number of monomials will not exceed 100.

Output

In the first line, print "`Multiple local maxima: Yes`" if the function has multiple local maxima, otherwise print "`Multiple local maxima: No`".

In the second line, print "`Multiple local minima: Yes`" if the function has multiple local minima, otherwise print "`Multiple local minima: No`".

In the third line, print "`Plateaus: Yes`" if the function has plateaus, otherwise print "`Plateaus: No`".

Sample input and output

unimulti.in	unimulti.out
1	Multiple local maxima: No Multiple local minima: No Plateaus: Yes
x^2y^4	Multiple local maxima: Yes Multiple local minima: No Plateaus: Yes
-1+5x^8y^7-9+x^2-y+5xy^4-3x^8y^0	Multiple local maxima: Yes Multiple local minima: Yes Plateaus: No

Problem B. Magic Treasure

Input file:	`standard input`
Output file:	`standard output`
Time limit:	2 seconds
Memory limit:	512 megabytes

In a far away country, a magic treasure is hidden in a labyrinth surrounded by magic walls, which is, in turn, lost in jungles. Centuries ago, mighty magicians and wealthy dictators tried to take the treasure away, but the labyrinth resisted. The treasure still waits for someone really powerful and clever to come.

Fortunately for the brave souls, an ancient artifact exists, which, when delicately asked, can tell the distance to the labyrinth or, when inside the labyrinth, the distance to the treasure if using a shortest way through the labyrinth. However, it was used so many times that only 2967 attempts has remained: after the last attempt is answered, this artifact will turn into a latest iPhone, or whatever is appropriate for the current civilization.

Sometimes magic is forceless once technology comes. When a similar labyrinth, which was known to hide an oilfield, was approached by military forces, it was found that the outer magic walls of the labyrinth are infinitely thin and lose their defensive power at four kilometers of altitude. So a brave soul can enter the labyrinth, provided it can fly high.

The Pnacotic Manuscripts hint that the labyrinth is organized as a lattice of size $M \times N$ ($1 \le M, N \le 50$), where each cell is a 10×10 meter square. Some cells are free, and some are occupied by infinitely heavy magic cubic blocks. The treasure is invisible and lies in one of the free cells, at integer coordinates.

You now reside in a tiny village somewhere near the labyrinth. You are equipped by a modern quadcopter that can fly 10 kilometers horizontally away from your location, and can, of course, fly high enough to surpass the magic walls. The ancient artifact is now mounted at the bottom of the quadcopter, and at one click of the remote controller you can activate the artifact and get to know:

- if outside the labyrinth: the distance to the closest outer wall of the labyrinth, in meters.
- if inside the labyrinth: the length of the shortest path to the treasure, avoiding the cubic blocks. Unfortunately, the artifact does not know that it is activated in the air, and returns the distance to the treasure as if it fell down and lies on the ground. That's how magic works, you know.

Find the treasure!

Interaction protocol

First, the testing system tells you the values of M and N. Then, you choose the coordinates to activate the artifact at, and pass them to the testing system. The testing system, in turn, flies your quadcopter to that point, then activates the artifact, gets the reply and passes it to you. In particular, it tells whether the artifact is inside or outside the labyrinth, and the distance. You will also know if the artifact is still usable or not.

Once you get to know where the treasure is, you send a special request to the testing system, which then moves the quadcopter to the location of the treasure, lands it, and activates the grip installed on the quadcopter. The testing system replies whether the treasure is captured or not. You have only one attempt here. After that, your program should terminate.

You will receive "Wrong Answer" if either:

- your query is misspelled;
- you try to activate the artifact which is not usable;
- you supply the wrong coordinates of the treasure;
- your program terminated without supplying the coordinates of the treasure.

If your program crashed, or used too much resources, you will receive an appropriate outcome: "Runtime Error", "Time Limit Exceeded", "Memory Limit Exceeded".

The labyrinth is guaranteed to be located completely within 5 kilometers from your location, which is $(0; 0)$. Although the treasure is at integer coordinates, the labyrinth is not guaranteed to be aligned with coordinate axes.

Input

The first line will contain two integers, M and N.

All other lines, except the last one, will contain answers to your queries, in the following form:

- `outside` D_o `active`: the artifact is outside, the distance to the closest outer wall is D_o, you can still use the artifact;
- `outside` D_o `iphone`: same as above, but you cannot use the artifact anymore (this happens after the query number 2967);
- `inside` D_i `active`: the artifact is inside and is **not** above the cubic block, the shortest path along the labyrinth to the treasure is D_i, you can still use the artifact;
- `inside` D_i `iphone`: same as above, but you cannot use the artifact anymore;
- `blocked active`: the artifact is inside and is **above** the cubic block, so the shortest path is not defined, you can still use the artifact;
- `blocked iphone`: same as above, but you cannot use the artifact anymore.

The testing system will compute D_o and D_i using 64-bit floating-point values, and the precision of these values will be reasonably high.

The last line is either `OK` or `FAIL`, depending on whether you specified the right coordinates of the treasure.

Output

Your queries should be either:

- `activate` X Y: you request to activate the artifact at coordinates (X, Y), which can be any floating-point values such that $\sqrt{X^2 + Y^2} \le 10^4$;
- `found` X Y: you think you found the treasure, and it is located at integer coordinates (X, Y). This shall be the last query.

When printing `activate` X Y in languages such as C or C++, please pay attention to the precision. The testing system will take these values literally, and they might be different to the ones in your program.

Please flush the output stream after printing a query (for example, `fflush` in C, `cout.flush()` or using `std::endl` in C++), or you might get "Idleness Limit Exceeded".

Sample input and output

standard input	standard output
1 1 outside 32.7566787083184 active outside 2.0 active inside 6.708203932499369 active inside 2.5495097567963922 active inside 2.1633307652783933 active inside 0.1414213562373093 active OK	activate 42 17 activate 5 12 activate 0 0 activate 3.5 3.5 activate 4.2 4.2 activate 3.1 5.9 found 3 6

Note

This problem is much simpler than you might think.

Problem C. Traveling Salesperson Problem

Input file:	`tsp.in`
Output file:	`tsp.out`
Time limit:	3 seconds
Memory limit:	512 megabytes

In this problem, you are asked to solve a *symmetric* traveling salesperson problem for a complete graph. More precisely, given a graph with lengths associated with its edges, you need to find a shortest route visiting every vertex exactly once. The route must finish in the same vertex where it started.

We use tests taken from the real world problems (except for the example test, of course). For each such problem, the exact value A_{best} of the answer is somehow known, and we give it to you. We want you to produce a route which does not exceed $1.05 \cdot A_{\text{best}}$.

Input

The first line of the input file contains two integers N and A_{best}, the number of vertices in the graph and the best known answer for this problem, correspondingly. $4 \le N \le 500$.

The following $N - 1$ lines contain the adjacency matrix of the graph. The i-th line of the input file, counting the first line, contains $i - 1$ integers $L_{i,1} \dots L_{i,i-1}$, the lengths of edges between the vertex i and all vertices with smaller indices. $1 \le L_{i,j} \le 25\,000$.

Output

Output N integers: indices of the vertices in the order which defines the (nearly) optimal route. Any route is accepted as long as its length does not exceed $1.05 \cdot A_{\text{best}}$.

Sample input and output

tsp.in	tsp.out
4 60 5 10 25 20 30 15	1 2 4 3

Problem D. Non-Dominated Sorting

Input file:	`sorting.in`
Output file:	`sorting.out`
Time limit:	2 seconds
Memory limit:	512 megabytes

In a D-dimensional space, a point $A = (A_1, \ldots, A_D)$ is said to *dominate* a point $B = (B_1, \ldots, B_D)$, if the following relations hold:

- $\forall i \in [1; D]\ A_i \le B_i$;
- $\exists j \in [1; D]\ A_j < B_j$.

Non-dominated sorting takes a set P of points and assigns *ranks* to these points in the following way:

- All points from P, which are not dominated by any point in P, get rank 0 and form a set R_0.
- All points from $P \setminus R_0$, which are not dominated by any point in $P \setminus R_0$, get rank 1 and form a set R_1.
- All points from $P \setminus (R_0 \cup R_1)$, which are not dominated by any point in $P \setminus (R_0 \cup R_1)$, get rank 2 and form a set R_2.
- ... and so on.

You are given N points in a D-dimensional space. Find their ranks assigned by non-dominated sorting.

Input

The first line of the input file contains two integers N and D, the number of points and the dimension correspondingly ($1 \le N \le 10^5, 2 \le D \le 4$).

The following N lines contain D integers each, which are coordinates of the points. All coordinates do not exceed 10^9 by the absolute value.

Output

Output N integers. The i-th integer must be the rank of the i-th point given in the input file.

Sample input and output

sorting.in	sorting.out
6 2 1 0 0 1 0 0 1 1 0 1 1 1	1 1 0 2 1 2
6 3 1 1 3 2 1 3 1 2 1 3 2 3 2 3 1 3 3 2	0 1 0 2 1 2
4 4 4 1 2 3 5 2 4 3 6 3 5 3 7 4 6 1	0 1 2 0

Note

For your convenience, dimensions of tests depend on their numbers:

- $T \bmod 3 \equiv 1 \leftrightarrow D = 2$;
- $T \bmod 3 \equiv 2 \leftrightarrow D = 3$;
- $T \bmod 3 \equiv 0 \leftrightarrow D = 4$.

Problem E. Bugs

Input file:	`not applicable`
Output file:	`not applicable`
Time limit:	`not applicable`
Memory limit:	`not applicable`

In this problem, you need to design a finite state machine that will guard the bug on a cellular field through the obstacles towards the target or determine that the target is unreachable.

The bug has a direction, so it can rotate and move forward. However, if it moves forward and hits an obstacle, it dies. In order not to die young, the bug can see whether there are obstacles around. The finite state machine accesses these sensors through the following predicates:

- `M0 ::` Can move forward?
- `M1 ::` Can move right?
- `M2 ::` Can move backward?
- `M3 ::` Can move left?

The bug has a rudimentary "sense of target", so it can feel whether it gets closer to the target (using the Manhattan metric, that is, the sum of distances by axes X and Y) if it moves to an adjacent cell. The finite state machine accesses these sensors through the following predicates:

- `C0 ::` Is move forward cool?
- `C1 ::` Is move right cool?
- `C2 ::` Is move backward cool?
- `C3 ::` Is move left cool?
- `?T ::` Is at target?

Note, however, that the move can be cool even if it is impossible! The last predicate tests whether the bug is already at the target. This predicate is not strictly necessary, but it makes the life a little bit easier.

There is one more component, without which the life of the bug would be too hard. The bug can remember one position (the memory cell and the direction) where it has already been, and it can compare whether it is exactly at that position, or it is closer to the target than the saved cell. **Initially, this position stores the starting position.** This data is accessible to the finite state machine as the following predicates:

- `?S ::` Is at saved position?
- `?B ::` Is better than saved?

As the number of predicates is quite large, each state of the finite state machine, which is the brain of the bug, is a decision tree. Each leaf of the decision tree is the action, which the bug should perform, and the number of the next state. The actions that do not terminate the process are:

- `N ::` Do nothing
- `F ::` Move forward
- `S ::` Save position
- `[ ::` Rotate counter-clockwise
- `] ::` Rotate clockwise

There are also two actions which correspond to the final decisions of the bug:

- `+ ::` Report reached
- `- ::` Report unreachable

The first of these actions must be issued when the bug is at the target, and the second when the target is impossible to reach without hitting an obstacle.

The internal node of the decision tree are the predicates. Each decision tree is written as a single line of characters, the grammar for which is given in the input description.

The bug starts from state 0. Your task is to find the finite state machine which acts correctly on **every imaginable field**. The testing system will try its best to validate this property by running your finite state machine on approximately 100 millions of tests having various sizes and properties.

Input

You submit your input file using a `*.txt` extension. This file should contain **five** lines, the i-th line (counting from 0) represents the decision tree for the i-th state of the finite state machine. The grammar is as follows:

Derivation	Semantics
P `::= +`	Report that the target is reached
P `::= -`	Report that the target is unreachable
P `::= N`x	Do nothing, go to state x
P `::= F`x	Move forward, go to state x
P `::= S`x	Save position, go to state x
P `::= [`x	Rotate counter-clockwise, go to state x
P `::= ]`x	Rotate clockwise, go to state x
P `::= M`x{P_1}{P_2}	If can move in direction x, execute P_1, otherwise execute P_2
P `::= C`x{P_1}{P_2}	If moving in direction x would decrease the Manhattan distance to the target, execute P_1, otherwise execute P_2
P `::= ?S{`P_1`}{`P_2`}`	If at the saved location, execute P_1, otherwise execute P_2
P `::= ?B{`P_1`}{`P_2`}`	If closer to the target (by the Manhattan distance) than at the saved location, execute P_1, otherwise execute P_2
P `::= ?T{`P_1`}{`P_2`}`	If at the target, execute P_1, otherwise execute P_2

Each line must not exceed 100 characters. The codes for the directions are 0 for forward, 1 for right, 2 for backward, 3 for left.

Your submission will be tested on fields of size up to 15×15. On each field, it should make at most 10^6 steps before performing action "`+ ::` Report reached" or "`- ::` Report unreachable".

Output

The outcomes of the testing will be as follows:

- Compilation Error: your bug description is syntactically incorrect.
- Runtime Error: your bug crashed into a wall.
- Time Limit Exceeded: your bug made 10^6 steps and did not reach the conclusion.
- Wrong Answer: your bug said "Report reached" not at the target, or it said "Report unreachable" when the target is actually reachable.
- Accepted: all tests are passed, your bug is likely correct.

There should be no other verdicts. In case such verdict arrived, please notify the teacher.

The visualizer to help you with validation is available online[1]. The Javascript code in the visualizer should behave exactly in the same manner as the code in the testing system. If you find any discrepancy, please notify the teacher.

Sample input

```
?T{+}{N1}
M0{F0}{[1}
N2
N3
N4
```

The finite state machine above moves forward as long as it is possible, otherwise turns counter-clockwise, and if it hits the target, it says so. Note that each unused state, from 2 to 4, has a trivial code, which does nothing and remains in that state.

[1] `https://ctlab.itmo.ru/~mbuzdalov/ec-2019/autobug.html`

Problem F. OneMax

Input file:	`standard input`
Output file:	`standard output`
Time limit:	10 seconds
Memory limit:	512 megabytes

The OneMax optimization toy problem is defined as follows. The testing system creates a bit string S of length N and hides that string from you, so you know only its length N. Your task is to find what S is. The only way to get some information about it is to ask the testing system a string Q and receive its *fitness*, which is:

$$F(Q) = |\{i \mid Q_i = S_i\}|.$$

At some point of time, you need to make a query Q' such that $Q' = S$. To make your life harder, you are not allowed to make more than $\lceil 3N/4 + 100 \rceil$ queries in total, including the last one.

To make your life easier, you don't need to provide a complete bit string for each query. Instead, you need to say which bits to flip in your previous query to create your next query. The first query is pre-created for you by the testing system. Its fitness will be given to you along with N.

Interaction protocol

First, the testing system tells your solution the length of the bit string N and the fitness of the first query F_1 pre-created by the testing system. Then, your solution tells the testing system the indices of bits to flip. The testing system flips the bits you specified in the current query bit string and tells you the fitness of this new string. If this fitness is N, the testing system terminates, so your solution must also gracefully terminate when it receives N as a fitness.

The limit on the number of queries is $\lceil 3N/4+100 \rceil$, where the first pre-created query also counts. If your solution asks a $\lceil 3N/4 + 101 \rceil$-st query, the system will hang, and you will receive the "Wrong Answer" outcome. You will also receive this outcome if your solution disconnects too early.

You will receive the "Time Limit Exceeded" outcome if your solution takes too long (i.e. more than ten seconds of processor time) to process the queries. If your solution crashes, you will receive the "Runtime Error" outcome.

If your query contains something other than the indices of bits to flip, separated by whitespaces, or if one index is specified multiple times, the testing system terminates, and you will receive the "Presentation Error" outcome.

Finally, if everything is OK (e.g. your solution finds the hidden string in a reasonably small number of queries) on every test, you will receive the "Accepted" outcome, in this case you will solve the problem.

Input

The first line of the input stream contains two numbers N and F_1 ($1 \le N \le 100\,000$, $0 \le F_1 \le N$), the size of the hidden bit string and the fitness of the first query. The next lines of the input stream consist of the answers to the corresponding queries. Each answer will be an integer number from 0 to N. Each answer will be on its own line.

Output

To make a query, print a line which contains the one-based indices of bits to flipped in the previous query to get the new query. Don't forget to put a newline character and to flush the output stream after you print the query.

Sample input and output

standard input	standard output
3 2 1 1 3	1 1 2 2 3