

CIRCUS

A new circus performance involves using really long ropes, hanging from the flat ceiling of the circus hall. The circus hall entrance and all the ropes are positioned in a straight line. The ceiling is 10^{18} units high and the ropes are hanging freely and they reach the ground. The circus performers must move from one rope to another, so that they can get at least M units away from the entrance.

There are N ropes in total. The i -th rope is hanging from a location that is P_i units from the entrance, as measured along the hall ceiling.

The circus performers are careful and don't jump crazily from one rope to another. Imagine a circus performer is holding the i -th rope at a distance of S units from the ceiling. The performer can swing on the rope she is holding. If swinging on a rope, the performer reaches a point not less than M units away from the entrance, we can consider her task accomplished. While swinging, the performer can grab another rope, which is hanging from a location that is at most S units away from the first rope. Formally, she can grab the j -th rope if $|P_i - P_j| \leq S$. Now, the performer holds both the i -th and j -th ropes. At this time, she would start climbing up the i -th rope, while still keeping in hands the j -th rope. When the performer reaches the point where the i -th rope touches the ceiling, she would make sure the j -th rope is pulled tightly along the ceiling. Only at this point, the performer can continue moving, holding on to the j -th rope, at distance from the ceiling based on how the rope was pulled along the ceiling. Formally, the performer continues her movement holding on to the j -th rope at a distance $|P_i - P_j|$ from the ceiling.

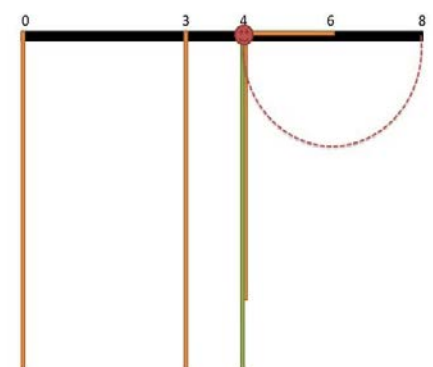
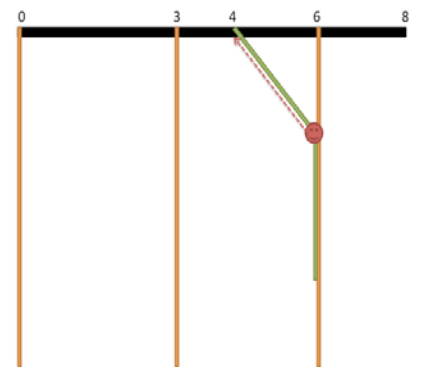
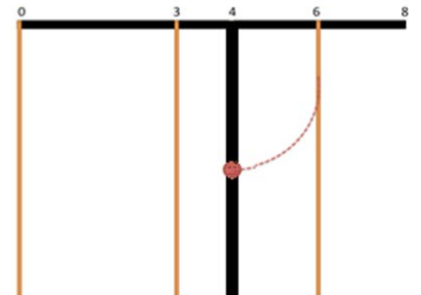
The circus manager wants to add an additional, temporary rope, hung on a position that is D units away from the entrance, measured on the ceiling. This is the rope where the performance will start. The performer's task is to make her way from this rope to the far end of the circus hall: at least M units away from the entrance. During the moving from one rope to another, the temporary rope is indistinguishable from a regular rope. Your program should answer the question: what is the minimal distance from the ceiling that the performer should hold the temporary rope at, so that she can accomplish her task.

Consider an example with 3 permanent ropes, positioned respectively 0, 3 and 6 units away from the entrance. The aim is to reach $M = 8$.

Consider a temporary rope, 4 units away from the entrance. Imagine a circus performer holding the temporary rope (bold) 3 units away from the ceiling. The performer can reach the rope that is 6 units from the entrance by swinging.

Once the performer grabs the rope that is 6 units away from the entrance, she starts climbing up the first rope (here – the temporary one).

After climbing up the first rope, she is holding the second rope $6 - 4 = 2$ units from its base. Now the performer is ready to swing and just able to reach the target that is 8 units away from the entrance.



Task

You need to implement two functions: `init`, which is called once at the beginning of your program's execution and processes the initial input parameters, and `minLength`, which answers one query given a temporary rope's location. Your function `minLength` will be called multiple times from the grader program.

- `init(N, M, P[])`
 - `N`: the number of ropes
 - `M`: the target distance in units
 - `P[]`: array of size `N` that holds the locations of all ropes, measured in units along the ceiling from the entrance
- `minLength(D)`
 - `D`: distance in units along the ceiling from the entrance where the temporary rope will be located

Implementation details

You have to submit exactly one file, called `circus.cpp`. This file implements the `init`, and `minLength` functions as described above, using the following signatures.

- `void init(int N, int M, int P[])`
- `int minLength(int D)`

File `circus.cpp` should NOT contain function `main()`, but can contain other declarations and functions, necessary for correct working of functions `init` and `minLength`. Your program should contain `#include "circus.h"` in the beginning

Constraints

$$1 \leq M \leq 10^9$$

$$0 \leq P_i < M$$

$$0 \leq D < M$$

Example

$N = 3, M = 8, P = \{0, 3, 6\}$

`minLength(4)` should return 2. It is possible to jump from the temporary rope to the rope at position 6, holding it 2 units from the ceiling, and then reach $M = 8$. Another possible sequence of jumps would be to jump from the temporary to the rope at position 0, then to 3, then to 6, and then to reach $M = 8$. However, the second sequence of jumps would require the performer to hold the temporary rope 4 units from the ceiling in the beginning.

`minLength(5)` should return 3. The performer is allowed to reach the exit even straight from the temporary rope.

Subtasks

subtask	Points	N	minLength calls	note
1	11	$0 \leq N \leq 100\,000$	1	Only one <code>minLength</code> call with $D = 0$
2	17	$0 \leq N \leq 100$	100	$M \leq 2000$
3	23	$0 \leq N \leq 2\,000$	1 000	
4	9	$0 \leq N \leq 2\,000$	1 000 000	
5	31	$0 \leq N \leq 100\,000$	1 000	
6	9	$0 \leq N \leq 100\,000$	1 000 000	

Local testing

In order to be able to test your functions *init* and *minLength* on your local computer you will get files *Lgrader.cpp* and *circus.h*. Compile them together with your file **circus.cpp** and you will receive a program that you can use to test your functions.

Lgrader reads the input in the following format:

- ❑ line 1: two integers N and M
- ❑ lines $2 + i$ ($0 \leq i \leq N-1$): P_i
- ❑ line $N + 2$: Q - the number of times *minLength* will be called
- ❑ lines $N + 3 + i$ ($0 \leq i \leq Q-1$): numbers D - parameters for each *minLength* call

Lgrader will print Q numbers, one per line, showing the return values of the *minLength* calls.

Example for local testing

Input	Output
3 8	2
0	3
3	
6	
2	
4	
5	

HAPPINESS

Monetary system in X-land is a bit strange. There are banknotes with values of all integer numbers from 1 to M . There is another strange rule in the shops of X-land – the customer can never receive change, but also cannot leave a tip – in other words the customer must always pay the exact value of his purchases. If he does not have the exact sum for his purchase, then he cannot buy. Imagine what inconvenience this creates for the customers.

Niya is a girl from X-land. Like all other persons, she constantly fights against the rules described above. She always knows her set of banknotes – let's assume their values are a_1, a_2, \dots, a_N . All those values are between 1 and M and she may possess more than one banknote of a kind. Also the sequence of values a_1, a_2, \dots, a_N , is not ordered in any way. Niya feels happy, when entering a shop, she may buy any combination of goods with total price equal to any number between 1 and the total sum of her banknotes $a_1 + a_2 + \dots + a_N$. In that case, when she is shopping, she must only consider her total amount of money without making complicated calculations of whether she can buy (or not) with her banknotes.

Remark: Let us sort a_1, a_2, \dots, a_N in ascending order. Let us denote $S_i = 1 + a_1 + a_2 + \dots + a_i$. Necessary and sufficient condition to be able to represent each number between 1 and $a_1 + a_2 + \dots + a_N$ as a sum of elements from the multiset a_1, a_2, \dots, a_N , is that the following inequality $S_i \geq a_{i+1}$ held true for each $i > 1$ and $a_1 = 1$.

As expected, Niya's set of banknotes is changing after each purchase and also after each wage she receives – that's why her happiness is variable. You can help the girl with a program. Your program will receive as an input the initial set of Niya's banknotes and all the events that happen – purchases and wages. The program should be able to determine if Niya is happy in the beginning and after each event.

We should note that Niya feels happy also when she doesn't have any money – then she just skips shopping and goes jogging.

Task

Write functions *init()* and *is_happy()*, which will be compiled with jury's grader. These functions should serve to determine Niya's happiness at the beginning and after each event. The functions will receive as parameters the starting set of banknotes and the sets of banknotes that are removed from the set (on purchases) and added to the set (on receiving wage).

Implementation details

You should submit to the grading system a file **happiness.cpp**, which contains functions:

bool init(int coinsCount, long long maxCoinSize, long long coins[]).

bool is_happy(int event, int coinsCount, long long coins[]).

Parameters description:

coinsCount – number of banknotes that are received (starting set or wage) or discarded (shopping).

maxCoinSize – maximum value of one banknote.

coins[] – array, in which in random order are given values of the banknotes (index starts from 0).

event – event's type :

–1 – Shopping;

1 – Receiving wage.

The function *init* is called once by the grader at the beginning to set the starting set of Niya's banknotes and then grader calls *Q* times function *is_happy* with *event* = –1 (shopping) or *event* = 1 (wage). After each call the called function should return *true*, if Niya feels happy with her current set of banknotes or *false* if she doesn't.

File **happiness.cpp** should NOT contain function *main()*, but can contain other declarations and functions, necessary for correct working of functions *init* and *is_happy*. Your program should contain `#include "happiness.h"` in the beginning

Constraints

Let N_c denote the number of Niya's banknotes at any given moment and K – the number of banknotes, used in any purchase or wage. Then we have:

$$0 \leq N_c \leq 200\,000$$

$$0 \leq Q \leq 100\,000$$

$$1 \leq M \leq 10^{12}$$

$$1 \leq K \leq 5$$

It is guaranteed that in any call of *is_happy* with *event* = -1 (shopping) the set of banknotes given in *coins[]* is a subset of current Niya's set of banknotes.

Example

called function	event	coinsCount	maxCoinSize	coins[]	function returns
init		5	100	4 8 1 2 16	true
is_happy	-1(shopping)	2		2 8	false
is_happy	1(receiving wage)	3		7 9 2	true

Subtasks

Subtask	Points	N_c	M	Q
1	10	≤ 300	≤ 100	≤ 100
2	20	≤ 20000	$\leq 10^{12}$	≤ 1000
3	30	≤ 200000	≤ 1000000	≤ 100000
4	40	≤ 200000	$\leq 10^{12}$	≤ 100000

Local testing

In order to be able to test your functions *init* and *is_happy* on your local computer, you will get files *lgrader.cpp* and *happiness.h*. Compile them together with your file **happiness.cpp** and you will receive a program that you can use to test your functions.

The program expects following input format:

Single positive integers N and M are given on the first row – initial count of Niya's banknotes and maximum value of one banknote.

N positive integers are given on the second row, separated by spaces – values of banknotes in the initial set.

Non-negative integer Q is given on the third row – event's count.

On each of the next Q rows one event is described – first, a value for the event is given: -1 (shopping) or 1 (receiving of a wage). After that a positive integer K is given – number of banknotes that are removed or added to Niya's set. Last K integers are given, separated by intervals – values for the banknotes which are removed or added.

On the standard output the program will print $Q+1$ lines with 0 or 1 – "happiness" statuses of Niya at the beginning and after each event.

Example for local testing

Input	Output
5 100	1
4 8 1 2 16	0
2	1
-1 2 2 8	
1 3 7 9 2	

Ultimate TTT

Tic-Tac-Toe is a game for two persons with very simple rules. The game is played on an (initially empty) board with three rows and three columns. The first player chooses one of the cells and fills it with her sign ('X'). After that the second player chooses an empty cell and fills it with her sign ('O'). On the third turn the first player ('X') chooses an empty cell again and so on. The players alternate turns and cannot skip a turn (it doesn't make sense to do so either). The game continues until one of the players has three of her signs in the same row, column, or one of the two diagonals. If the board gets filled without any of the players achieving a winning configuration, the game is considered as draw.

Elly and Kris are playing a modified version of the game. Instead of the standard 3-by-3 board, they start at infinitely large board. After the first player makes a move (it actually doesn't matter where), the next move can only be made on a cell, which can potentially be on 3x3 board with the first one. In other words, the second move must be at most 2 columns and/or rows away from the first one. The next moves follow a similar logic: the players must always make such choice of a blank cell that all of the already played ones can fit on a 3-by-3 board. Please note that the more the game progresses, the smaller the board of valid cells gets, until finally reaching a standard 3-by-3 board (assuming none of the players wins before that). Please see the example below for clarification.

As in the original game, the winner is the player who first has three neighboring cells on the same row, same column, or diagonally. Similarly, if all the valid cells get filled before anyone wins, the game is considered a tie.

Consider the following example:

. X X X O
. . X O X X O . .	. X X O . .	. X X O O . .
X X O . X . O . .	X X O . X . O . O	X X O . X . O X O	X O . . O O

In this example the first player ('X') won, but using better strategy of the second player ('O') this wouldn't be so.

We will call *optimal play* the making of such moves, so the player wins, if the game can be won, makes a draw, if the game cannot be won, but can be tied, or loses, if none of the two options above are available. We assume that the opponent also plays optimally.

Task

Write a program that, given the current state of the board, finds which cell a player would choose if playing optimally.

Input

The first line of the standard input contains two integers **N** and **M** – the number of remaining valid rows and columns, respectively. Each of the following **N** lines contains a string of length **M**, describing the next row of the board. All lines form a correct description of the current state of the board. It is guaranteed, that at least one move has been made (so the remaining valid cells are finite), and the game hasn't finished yet.

Output

The program writes on a single line of the standard output two space separated integers **R** and **C**: respectively a row and a column number (indexed from 1). They locate a cell which the next player should choose to play optimally. If there are more options, the output can describe any of them.

Constraints

- $3 \leq N, M \leq 5$
- All symbols, describing the board, are from the alphabet {'.' (dot), 'X' (the capital letter), 'O' (the capital letter)}.
- In test cases, worth 50% of the points for the task, $N = M = 3$ (i.e. the board is a standard Tic-Tac-Toe board).

Grading

The problem tests are grouped into 4 subtasks. Each subtask gets 25 points, if all tests in this subtask are passed.

Example

Input

```
3 4  
.XX.  
....  
.O..
```

Output

```
1 1
```

Clarification

It is second player's ('O') turn. If she chooses (1, 4), then 'X' can win by playing in (2, 3) and putting 'O' into a "fork" – no matter which cell she chooses, 'X' would have a winning move.

By playing in (1, 1) the next move the board will be limited to the columns 1..3, thus it would be impossible for 'X' to win in one move by playing in (1, 4). By using this move there is a strategy, with which 'O' can lead the game to a tie.