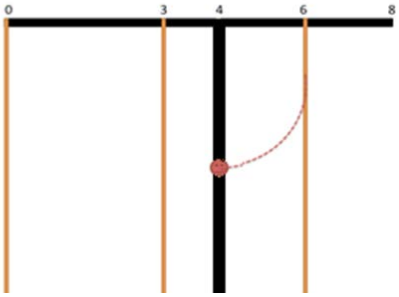
CIRCUS

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.

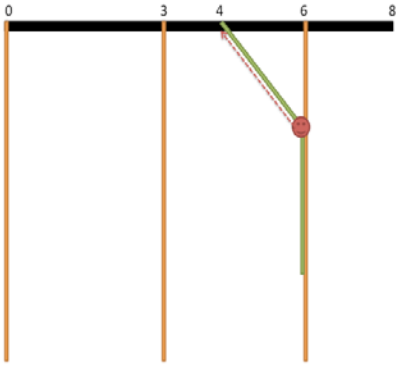
There are N ropes in total. The i-th rope is hanging from a location that is P_i units from the entrance, as measured along the hall ceiling.

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.

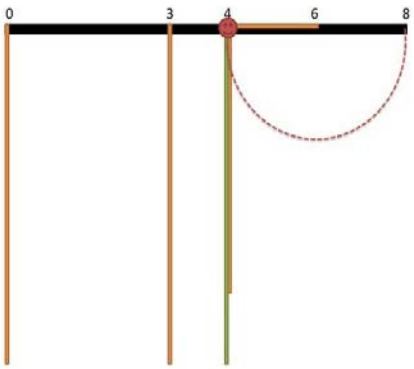
The circus manager wants to add an additional, temporary rope, hung on a position that is *D* units away from the entrance, measured on the ceiling. This is the rope where the performance will start. The performer's task is to make her way from this rope to the far end of the circus hall: at least *M* 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

$1 \le M \le 10^9$

$0 \le P_i < M$

$0 \le D < M$

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	$0 \le N \le 100\,000$	**1**	Only one minLength call with D = 0
2	17	$0 \le N \le 100$	**100**	$M \le 2000$
3	23	$0 \le N \le 2\,000$	**1 000**	
4	9	$0 \le N \le 2\,000$	**1 000 000**	
5	31	$0 \le N \le 100\,000$	**1 000**	
6	9	$0 \le N \le 100\,000$	**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 $\mathbf{2 + i}$ ($0 \le i \le N-1$): P_i
- line $N + 2$: Q - the number of times `minLength` will be called
- lines $\mathbf{N + 3 + i}$ ($0 \le i \le Q-1$): 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 0 3 6 2 4 5	2 3

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,…,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,……,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 $a_1 + a_2 + ... + a_N$. 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,……,a_N in ascending order. Let us denote $S_i = 1+a_1+ a_2+…….+a_i$. Necessary and sufficient condition to be able to represent each number between 1 and $a_1+ a_2+…….+a_N$ as a sum of elements from the multiset a_1, a_2,……,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 *Nc* 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:

$0 \le Nc \le 200\,000$
$0 \le Q \le 100\,000$
$1 \le M \le 10^{12}$
$1 \le K \le 5$

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	Nc	M	Q
1	10	≤ 300	≤ 100	≤ 100
2	20	≤ 20000	$\le 10^{12}$	≤ 1000
3	30	≤200000	≤ 1000000	≤ 100000
4	40	≤ 200000	$\le 10^{12}$	≤ 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 4 8 1 2 16 2 -1 2 2 8 1 3 7 9 2	1 0 1

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:

<table>
<tr>
<td>. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .</td>
<td>. . . . . .
. . . . . .
. . . . . .
. . . X . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .</td>
<td>. . . . .
. . . . .
. . X . .
. . . . .
. . . . .</td>
<td>. . . . .
. . . . .
. . X . .
. . . . .
. . 0 . .</td>
</tr>
<tr>
<td>. . X . .
. . . . .
. . 0 . .</td>
<td>. . X X .
. . . . .
. . 0 . .</td>
<td>. X X .
. . . .
. 0 . .</td>
<td>. X X 0
. . . .
. 0 . .</td>
</tr>
<tr>
<td>X X 0
. X .
0 . .</td>
<td>X X 0
. X .
0 . 0</td>
<td>X X 0
. X .
0 X 0</td>
<td>X | 0
. | .
0 | 0</td>
</tr>
</table>

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

- $3 \leq \mathbf{N}, \mathbf{M} \leq 5$
- 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.

CLARKSON

"Jeremy Clarkson Beatbox" is a popular YouTube video montage of the former Top Gear host performing beatbox. The video consists of a series of scenes from the show, stitched together to produce an entertaining musical performance. The video was published on YouTube in 2009 and it has been viewed almost three million times since then.

This year Jeremy Clarkson departed from the BBC, following a disciplinary ruling. To commemorate the iconic show, we want to produce a sequel to the popular YouTube montage, using scenes from more recent episodes. The lyrics of the new video are already chosen by the fans of the show, but making the montage is not an easy thing.

The problem is that, no matter how skillfully the separate pieces of the video are stitched together, the transitions are always noticeable to the viewers. If two transitions occur within a short period of time, it can even be irritating for some viewers. Therefore, the goal is to keep the transitions as far apart as possible by maximizing the length of the shortest piece in the montage.

Task

Write a program that takes two lines of input: the lyrics of the song on the first line, and the script of an episode of Top Gear on the second line. The program should then find the best way to construct the lyrics out of substrings of the script by maximizing the length of the shortest substring. It should output the length of the shortest substring in that optimal variant of the song. If it's impossible to construct the song out of the given episode, it should output −1.

Input

Your program should read from the standard input two lines of text, containing only English uppercase and lowercase letters.

Output

Your program has to write to the standard output one integer, which is equal to the length of the shortest substring in the optimal variant of the song, or -1 if construction is impossible.

Constraints

Each line in the input will be at most 100000 characters long.

Example

Input

```
JusticeAndTrust
CarsAndTrucksAreJustNice
```

Output

```
3
```

Explanation

```
Just|ice|AndTr|ust
Min(4, 3, 5, 3) = 3
CarsAndTrucksAreJustNice
```

Subtasks

Let us denote the length of the first string by *N*, and the length of the second string by *M.*

subtask	points	*Max N, M is less than or equal to*	note
1	11	**10**	
2	17	**2 000**	
3	19	**10 000**	
4	23	**30 000**	The answer to the problem is less or equal to 20
5	30	**100 000**	

RADIO

The border between Bulgaria and Romania is the Danube River. In fact, you have probably already seen it – it's hard to miss.

Since it is possible to cross the river by boat, people in the past and now do it. In order to prevent accidents, the Rescue Service built **N** security towers on the Bulgarian bank. Their positions are distanced X_1, X_2, …, X_N meters downstream from the point where the river enters Bulgaria. For simplicity we will consider the Danube to be a line segment and the towers – points with integer coordinates on it.

There is a radio transmitter in each tower. The transmitter in tower *i* has power P_i. Using these transmitters the towers might be able to communicate between one-another. Two towers *i* and *j* can communicate if the distance between them is less than or equal to the sum of their transmitters' powers (in other words: if $|X_i - X_j| \leq P_i + P_j$).

In order to reduce costs, a decision was made to decommission some of the towers. The new plan states that only **K** towers must remain and the rest will be sold. Selling tower *i* brings the government S_i leva (lev, plural leva, is the Bulgarian currency), but the tower can no longer be used.

However, to have a well-functioning system, a new requirement was introduced: each two of the remaining **K** towers must be able to communicate each other directly (when only one tower remains, we do not mind communications). Since this is potentially impossible with their current powers, they can be upgraded. Upgrading any tower increases its power for the price of one lev per one unit power.

Task

Ms. Elly recently started a summer internship at the Ministry of Finance. Now she wants to shine by proposing the most cost-efficient plan for selling **N** − **K** towers and upgrading the rest, so that each pair of the remaining ones can communicate directly between one-another. You decide to help the girl by writing a program which finds the minimal price (or maximal gain, if the sale of towers brings more money than are being spent on upgrading) for the task.

Input

On the first line of the standard input, two integers **N** and **K** are given – the number of towers in the beginning and at the end, respectively. On each of the following **N** lines are given three integers $\mathbf{X_i}$, $\mathbf{P_i}$, $\mathbf{S_i}$ – the position of the respective tower, its initial power, and the price at which it can be sold. The towers are given in ascending order of their positions $\mathbf{X_i}$. No two towers have equal coordinates $\mathbf{X_i}$.

Output

On a single line of the standard output, your program should print out one integer – the minimal price (or maximal gain) to do the task. In case of a gain, the printed number should be negative.

Constraints

- $1 \le \mathbf{K} \le \mathbf{N} \le 100\,000$
- $1 \le \mathbf{X_i}, \mathbf{P_i}, \mathbf{S_i} \le 1\,000\,000\,000$

Subtasks

- Subtask 1 (15 points): $\mathbf{N} \le 15$;
- Subtask 2 (15 points): $\mathbf{N} \le 1\,000$, $\mathbf{N} = \mathbf{K}$;
- Subtask 3 (15 points): $\mathbf{N} \le 1\,000$, $\mathbf{S_i} = 1$;
- Subtask 4 (15 points): $\mathbf{N} \le 100\,000$, $\mathbf{N} = \mathbf{K}$;
- Subtask 5 (15 points): $\mathbf{N} \le 100\,000$, $\mathbf{S_i} = 1$;
- Subtask 6 (25 points): $\mathbf{N} \le 100\,000$.

Examples

Input	Input
5 3 4 63 3 13 2 4 87 3 9 121 6 15 159 5 2	9 5 5 8 4 10 10 7 11 9 7 13 6 6 19 20 9 20 2 1 23 1 3 26 13 11 28 4 2

Output	Output
42	-24

Explanation

In the first example, one optimal variant is to keep the towers with indices {1, 3, 4}. This way we must increase the power of tower 1 with 17 and the power of 4 with 31, paying 17 + 31 = 48 leva. Selling the rest of the towers (2 and 5) we gain 4 + 2 = 6 leva. The total price is 48 – 6 = 42 leva.

In the second example we can choose, for example, to keep the towers with indices {2, 3, 6, 7, 9}. In order for the towers to be able to communicate with each other we must increase the power of 7 with 2 and the power of 9 with 4, for a price of 2 + 4 = 6. With this set of towers we can sell {1, 4, 5, 8} for 4 + 6 + 9 + 11 = 30 leva. The total "price" is 6 – 30 = −24 leva. Thus, we have gain of 24 leva.

TILING

Let us consider the positive integers k, l, and n. A rectangular room has floor dimensions of $k \cdot n \times l \cdot n$ units (the sign "centered dot" (·) means multiplication of integers). The floor is tiled with $k \cdot l \cdot n$ rectangular tiles of size $1 \times n$. No tile has been cut during the tiling. We can consider the floor as a grid with $k \cdot n$ columns and $l \cdot n$ rows.

A mentally disturbed robot has locked himself in the room and threatens to blow it up if you don't guess the exact way of floor tiling: what is the position of every tile in the covering. Fortunately, the robot is inclined to give you some information, on which basis you can try to reveal the exact tiling. You can ask the robot a set of q questions of the type $(x\ y)$, where $1 \le x \le k \cdot n$ and $1 \le y \le l \cdot n$ are numbers, respectively of a column and a row (counting starts at 1 from the "top left corner" – a cell in first column and first row).

You will get back a set of q ***correct*** answers exactly what tile is covering each of the queried cells: the coordinates of the top left corner ("the beginning") of the tile and the way it is laid (its "direction") – "horizontally" (along a row) or "vertically" (in one column).

Task

Write a function `tiling()`, which will be compiled with a jury's program and will lead a short dialog with the robot to reconstruct the tiling out of the received information.

You are maybe planning to ask a question for each cell, and voilà! Alas, the number q of questions asked should be as small as possible, otherwise you take the risk to annoy the robot, and although your program has guessed the right tiling, an explosion can occur (that is – a zero for the test: see the grading rules).

Implementation details

You should submit to the grading system a file **tiling.cpp**, which contains the function `tiling()`. Your file may contain any other necessary code and names, but should use as a global neither one of the predefined names below, nor the name `main`.

At the beginning of the file there should exist one line with the preprocessor instruction

```
#include "tiling.h"
```

The jury's program declares and implements the following items:

```
struct Data
{int x,y;
 char d;
};
void getArea(int *k, int *l, int *n);
bool getData(int q, Data *quest);
void setResult (const char *res);
```

Description of the defined functions for communication with the robot:

1. Function **`getArea`** will return the room parameters *k*, *l* and *n* (the column count will be *k*·*n*, and the row count will be *l*·*n*).
2. You can call **once** the function **`getData`** with the prototype above. Parameters have the following meaning:
   - *q* is the number of questions;
   - the input/output array of records `quest` contains:
     - before the call – the questions themselves (*x* and *y* are respectively the column and the row of the queried cell). The value of the member named *d* doesn't matter here;
     - after the call – the answers of the robot, namely: *x* and *y* are respectively the column and the row of the beginning of the tile, covering the respective asked cell, and the member *d* has for its value one of these two symbols: `h`, meaning horizontal direction, or `v`, denoting vertical direction. If the function returns **`false`**, that means that either there are incorrect data in the input/output array (say, out of the borders), or the function has already been called. In this case the array will contain zeroes as output values.
3. Before the exit, `tiling()` should call the defined function **`setResult`**, where the symbol array `res` describes the tiling. These are *k*·*l*·*n* symbols (with no delimiter), each being `h` or `v`. Here follows the algorithm for creating `res`:

- Start with no symbols in `res`;
- Consider walking the floor (the grid) by rows from the first to the *l*·*n*-th, and every row by columns from the first to the *k*·*n*-th. If you step on an upper left corner of a tile, you add one symbol to `res`: `h` for horizontal direction of the tile, or `v` for vertical direction. Cells which are not upper left corners (beginnings) are being simply jumped over.

Constraints

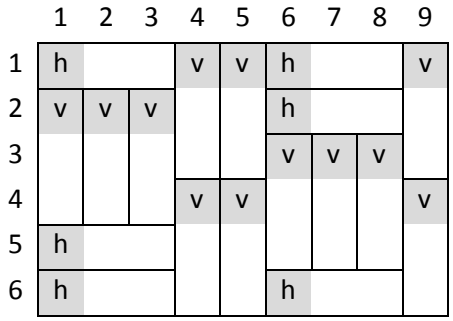
In 20% of the tests, $1 \le k, l \le 10$.
In 30% of the tests, $n = 2$.
$1 \le k, l \le 57$, $2 \le n \le 10$.

Grading

If a registered tiling description is missing or wrong, the test is given no points. A correct description is granted points according to the proximity of the asked questions' count q to the theoretically necessary questions' count. More precisely, if the theoretical minimum is Q questions and the test example is designed for P points, a correct description will be given $\min(P, P(Q/q)^{30})$ points. The final sum is rounded to the nearest integer.

Example

This example is made with respect to the tiling in the picture. You can see in grey the beginnings of the tiles, and a letter in each beginning, denoting the direction of the tile.



If, for example, you have defined variables

```
int k,l,n;
```

a call of the function

```
getArea(&k,&l,&n);
```

will set the values of these variables respectively to `k=3, l=2` and `n=3.`

For an illustration about posing a set of questions, look at the following fragment:

```
Data data[128] = {{1,1,0},{4,1,0},{7,1,0},{1,2,0},
                  {2,3,0},{3,4,0},{4,5,0},{5,6,0},{7,5,0}};
if (getData(9,data)) {//interpret the returned data}
else {//there is an error or this is not the first call}
```

In the considered example after the first call of **`getData`**, as the data before the call are in the border limits, the array `data` will look like this:

```
{{x:1,y:1,d:'h'},{x:4,y:1,d:'v'},{x:6,y:1,d:'h'},
 {x:1,y:2,d:'v'},{x:2,y:2,d:'v'},{x:3,y:2,d:'v'},
 {x:4,y:4,d:'v'},{x:5,y:4,d:'v'},{x:7,y:3,d:'v'}}
```

The result should be registered by calling `setResult`. In this example, say:

```
char r[128]="hvvhvvvvhvvvvvvhhh";
setResult(r);
```

Example comments

Nine correct questions are posed (q=9). The returned nine correct answers allow tiling's determination, which is registered in accordance with the rules. Posed question's count exceeds the theoretical minimum needed for this configuration, which is Q=6. So a solution like this would be granted a part of the provided P points, namely $\{P(2/3)^{30}\}\approx\{0.000005\ P\}$. As you can guess, an answer like this, although correct, will practically be useless.

Local testing

In order to be able to test your function *tiling* on your local computer, you will get files *Lgrader.cpp* and *tiling.h*. Compile *Lgrader* together with your file **tiling.cpp** and you will receive a program that you can use to test your function.

Lgrader reads the standard input in the following format:

- Line 1 contains three integers: k, l and n.
- It follows a matrix of integers. The integer in each cell of this matrix denotes a tile number. The matrix consists of $l \cdot n$ lines; each line contains $k \cdot n$ integers.

Example for local testing (corresponds to the picture above).

Input	Output
3 2 3 1 1 1 2 3 4 4 4 5 6 7 8 2 3 9 9 9 5 6 7 8 2 3 10 11 12 5 6 7 8 13 14 10 11 12 15 16 16 16 13 14 10 11 12 15 17 17 17 13 14 18 18 18 15	hvvhvvvvhvvvvvvhhh