Colombian Collegiate Programming League
CCPL 2013

Contest 4 -- April 27

Simulation and Math-Numeric

Problems

This set contains 8 problems; pages 1 to 12.

(Borrowed from several sources online.)

A - Tennis Set

Source file name: `tennis.c`, `tennis.cpp` *or* `tennis.java`

In tennis, the two players play games, alternating which player is the server in each game. When a player has won at least 4 points and has won at least 2 more points than his opponent, he has won the game. When a player has won at least 6 games and has won at least 2 more games than his opponent, he has won the set.
We have a Sequence of chars; the points on the game that tells us the result of each point in a recent tennis match. Each character is either `S` indicating that the server won that point, or an `R` indicating that the returner (the non-server) won that point. All together they represent one sequence of points played in the match.
Call the player who serves in the first game Player A and his opponent Player B. We want to know the score in the first set, with the number of games won by Player A, then a hyphen (`-`), then the number of games won by Player B. The data in points may go beyond the end of the first set, in which case the extra data can be ignored. The first set may not yet be completed, in which case we want the score to include only the completed games.
You must find and print the score of the first set. The printed outcome must not contain any extra characters (such as spaces or leading zeros). We are NOT using the tiebreaker which is sometimes used when a set reaches $6 - 6$.

Input

The input will contain several test cases, each test case will be in a single line containing a sequence of `R` and `S`, its length will be at least 1 and at most 2500.

The input must be read from standard input.

Output

Print a line with the answer, follow the format below.

The output must be written to standard output.

Sample input	**Sample output**
SSSSSSSSSSSSSSS	2-1
SSSRSSRRRRR	2-0
SSSSRRRRSSSSRRRRSSSSRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR	6-0

B - Geometric Progressions

Source file name: `geoprog.c`, `geoprog.cpp` *or* `geoprog.java`

You are given two geometric progressions $S_1 = b_1 \cdot q_1^i$ $(0 \leq i \leq n_1 - 1)$ and $S_2 = b_2 \cdot q_2^i$ $(0 \leq i \leq n_2 - 1)$. Find and print the number of distinct integers that belong to at least one of these progressions.

Input

The input will contain several test cases; each test case will be in a single line containing six integers: b_1, q_1, n_1, b_2, q_2, n_2. $0 \leq b_1, b_2, q_1, q_2 \leq 500'000.000$. $1 \leq n_1, n_2 \leq 100.500$.

The input must be read from standard input.

Output

Print a line with the answer, follow the format below.

The output must be written to standard output.

Sample input	**Sample output**
3 2 5 6 2 5	6
3 2 5 2 3 5	9
1 1 1 0 0 1	2
3 4 100500 48 1024 1000	100500
15724 19169 26501 6334 18467 42	26543

C - Water Pressure

Source file name: `water.c, water.cpp` *or* `water.java`

You have been given a *layout* containing a map of an underground cavern. The jth character of the ith element of the given layout will correspond to row i column j in the map. Water has begun flowing in at the upper left-hand corner (row 0, column 0), and will soon makes its way throughout the cavern. Each character in layout will either be a digit $(0-9)$ or `'X'` (except for the upper left-hand corner, which will always be `'.'`, and can be thought of as water). An `'X'` means that water will never penetrate the solid rock present in that area. A digit describes how much water pressure is required to penetrate that area. The water will travel according to the following rules:

1. Each second, beginning with second 0, the total amount of water increases by 1 in the squares where water has reached.
2. The total water pressure is equal to the total amount of water divided by the total number of squares the water has reached. This need not be an integral amount.
3. At the end of each second, after the total amount of water has been incremented, each square adjacent to the water (not diagonally) is checked for strength. Every adjacent square, whose digit value is strictly smaller than the total water pressure, will become covered with water.

Above, the total amount of water, and total amount of water pressure are values corresponding to the map as a whole, and not individual squares. Note that the water will never leave the map. When calculating which squares become covered during a given second, all squares adjacent to water are considered simultaneously. In other words, one square getting flooded will not then change the pressure, and prevent another square from getting flooded that same second. In addition, only squares directly adjacent to the water may be flooded during any second. Before second 0 occurs, the square in the upperleft hand corner is already flooded, and the total water is 1. This implies, after the increment occurs during second 0, the total water will be 2, and the total water pressure will be 2. You must find and print the second during which the last square containing a digit is flooded. If this will never occur, print -1.
Note: Total water pressure need not be an integer.

Input

The input will contain several test cases; each test case will be in several lines, the first line will contain R and C; the numbers of rows and columns of the given layout. $1 \leq R, C \leq 50$. The next R lines will contain the given layout; each line will contain C characters. The 0th character in the 0th element of layout will be (quotes for clarity) `'.'`. Every other character in layout will be a digit (`'0'-'9'`) or `'X'`.

The input must be read from standard input.

Output

Print a line with the answer, follow the format below.

The output must be written to standard output.

Sample input	**Sample output**
2 2 .0 0X 2 2 .0 00 2 2 .5 44 1 9 .23456789 2 3 .X0 X00	0 1 14 71 -1

D - Two Kings

Source file name: `twokings.c`, `twokings.cpp` *or* `twokings.java`

A white queen and two black kings are placed on a 100×100 chessboard. White and black sides move by turns. The white queen moves first (only one of the two black kings is moved on each turn). Given three positions: q (position of queen), k_1 (position of first king) and k_2 (position of second king), find and print the minimal number of white moves needed to capture one of the kings (kings are not allowed to capture the queen, and may not move to the queen's location). q, k_1 and k_2 will each contain the 0-based row and column of the corresponding piece.
Notes:

- The black side tries to avoid capturing as long as possible.
- Kings move one cell in any direction, horizontally, vertically, or diagonally.
- The queen moves any number of cells in any direction, horizontally, vertically, or diagonally.
- The queen captures a king if it moves to the cell that the king occupies.
- Neither side can skip a turn.
- Pieces may not move off the board.
- q, k_1 and k_2 will each contain two space-delimited integers between 0 and 99, inclusive.
- The numbers in q, k_1 and k_2 will not contain leading zeroes.
- No two pieces will occupy the same cell.

Input

The input will contain several test cases; each test case will be in a line containing 6 integers; q-`row`, q-`col`, k_1-`row`, k_1-`col`, k_2-`row`, k_2-`col`.

The input must be read from standard input.

Output

Print a line with the answer, follow the format below.

The output must be written to standard output.

Sample input	**Sample output**
0 0 1 1 99 99	1
0 0 99 0 0 99	1
1 1 90 0 0 30	2
98 98 0 97 99 0	2
99 0 1 1 0 98	2
99 10 80 80 50 1	4

E - Triangle Count

Source file name: `triangle.c, triangle.cpp` *or* `triangle.java`

Given an equilateral triangle with side length N divided into a triangular grid of triangles with side length 1, count the total number of triangles present in the grid.

Input

The input will contain several test cases; each test case will be in a line containing an integer N ($1 \leq N \leq 500$).

The input must be read from standard input.

Output

Print a line with the answer, follow the format below.

The output must be written to standard output.

Sample input	**Sample output**
2	5
3	13
4	27
5	48

F - Flatland

Source file name: `flatland.c`, `flatland.cpp` *or* `flatland.java`

Once upon a time there was *Flatland*, a world whose inhabitants believed was a 2D rectangular region. Flatlanders (the people inhabiting Flatland) assumed that if somebody traveled long enough in one direction, he/she would fall down over the edge of Flatland. Animated Caribbean Movies (ACM) plans to produce a film about Flatland. Before the script of the film is approved, ACM wants to become familiar with life as it was in Flatland by simulating how the world could evolve from given initial situations and some conditions determining life and death.
Flatlanders were nomads by nature as they were always traveling: all of them traveled at the same speed rate on straight lines but each individual had its own direction. As you may imagine, if one observed the life of a lonely Flatlander, he/she would eventually reach Flatland's edge and die. Nevertheless, if two Flatlanders collided, then their fate was improved as their directions changed: the resulting directions were as the former ones reflected in a mirror bisecting the angle between the former directions of the crashing inhabitants. So, a Flatlander survived because a collision with another one could change his/her direction.
However, there were bad news when more than two Flatlanders collided in a single crash: in that case all of them died, disappearing right there. Note that some Flatlanders could die at the same time. If this was the case, the last name of the list of deads was remembered as the *Last Dead One* in that moment (to simplify, we assume they used our modern English alphabet and lexicographic order). The survivors venerated the name of the Last Dead One until a new last dead appeared (and some Flatlander disappeared).
ACM's film begins with a given population in Flatland, where names, positions, and directions of every single individual are known. ACM wants you to help them to determine which would be the name of the last Last Dead One in the whole Flatland's life.

Input

There are NC test cases to solve, $0 < NC < 100$. The first line of the input file has NC. After that, for each testcase, a set of lines:

- the first line contains a number n, the number of Flatlanders in the initial world ($1 \leq n \leq 100$);
- the second line contains two positive integer numbers B and H separated by a space, representing the dimensions of Flatland ($2 \leq B, H \leq 100$). Coordinates in Flatland are points (i, j), with $0 \leq i \leq B$, $0 \leq j \leq H$. Flatland's edges are points with coordinates of the form $(0, j), (i, 0), (B, j)$ or (i, H);
- n lines (one per Flatlander) with four numbers and one string: x, y, d_1, d_2 and *name* separated by a blank. (x, y) represents the position of the Flatlander ($0<x<B$, $0<y<H$, and two Flatlanders cannot start in the same position), (d_1, d_2) represents the direction: (d_1, d_2) is a point on some Flatland's edge, so that the Flatlander is moving towards it; *name* is a string of one to 10 alphabetical uppercase characters, which represents the name of the Flatlander. You may assume that every Flatlander has a unique name.

The input must be read from standard input.

Output

For each given case, output one line with the name of the Last Dead One.

The output must be written to standard output.

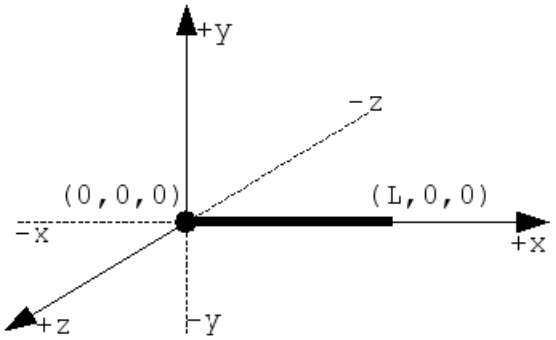
Sample input	**Output for the sample input**
2 2 20 23 1 1 0 0 BOB 3 3 3 0 ALICE 3 20 23 2 2 4 0 ALICE 4 2 2 0 BOB 1 3 0 3 CHARLES	ALICE BOB

G - Bender B. Rodríguez Problem

Source file name: `bender.c`, `bender.cpp` *or* `bender.java`

Bender is a robot built by *Mom's Friendly Robot Company* at its plant in Tijuana, Mexico in 2996. He is a `Bending-Unit 22`, serial number 2716057 and chassis number 1729. He was created for the task of bending metal wires.

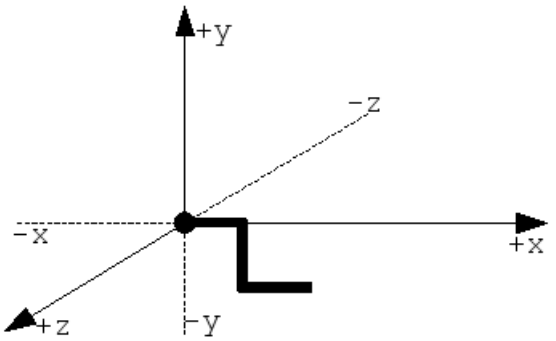
Bender needs to bend a wire of length `L` (`L`$\geq$`2` an integer). The wire is represented in the Bender's brain (a `MOS Technology 6502` microprocessor) as a line stucked in the origin of a tridimensional cartesian coordinate system, and extended along the `x` positive axis (`+x`), so that the fixed extreme of the wire is in the coordinate `(0,0,0)` and the free extreme of the wire is in the coordinate `(L,0,0)`.



Bender bends the wire at specific points, starting at the point `(L-1,0,0)` and ending at the point `(1,0,0)`. For each `i` from `L-1` to `1`, Bender can take one of the following decisions:

- Not to bend the wire at point `(i,0,0)`.
- To bend the wire at point `(i,0,0)` an angle of $\frac{\pi}{2}$ to be parallel to the axis `+y`, `-y`, `+z` or `-z`.

For example, if `L=3` and Bender bends the wire at `(2,0,0)` on the `+y` axis direction, and at `(1,0,0)` on the `-y` axis direction, the result would be:



Given a sequence of bends, you must determine what direction is pointed by the last segment of the wire (`+x` in the example). You can suppose that the wire can intercept itself, after all it is the future!

Input

The first line of each test case gives an integer `L` (`2`≤`L`≤`100000`) indicating the length of the wire.

The second line of each test case contains the `L-1` decisions taken by Bender at each point, separated by spaces. The `j`-th decision in the list (for each `1`≤`j`≤`L-1`) corresponds to the decision taken at the point `(L-j,0,0)`, and must be one of the following:

- `No` if the wire isn't bended at point `(L-j,0,0)`.
- `+y` if the wire is bended at point `(L-j,0,0)` on the `+y` axis.
- `-y` if the wire is bended at point `(L-j,0,0)` on the `-y` axis.
- `+z` if the wire is bended at point `(L-j,0,0)` on the `+z` axis.
- `-z` if the wire is bended at point `(L-j,0,0)` on the `-z` axis.

The end of the input is indicated when `L=0`.

The input must be read from standard input.

Output

For each case in the input, print one line with the direction pointed by the last segment of the wire, `+x`, `-x`, `+y`, `-y`, `+z` or `-z` depending on the case. *The output must be written to standard output.*

Sample input	**Output for the sample input**
3 +z -z 3 +z +y 2 +z 4 +z +y +z 5 No +z No No 0	+x +z +z -x +z

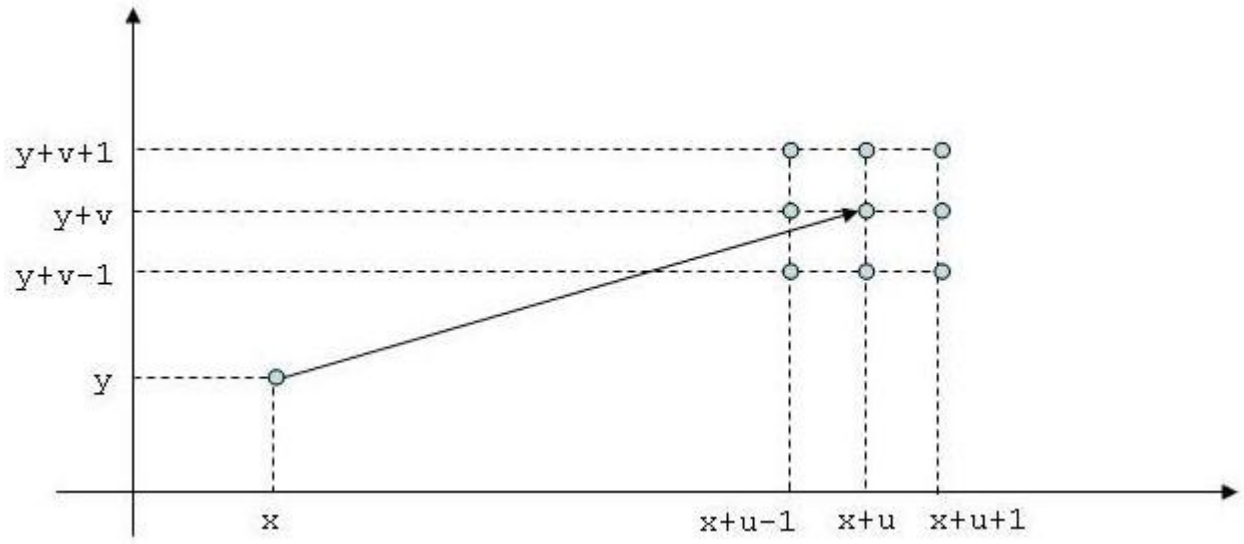
H - Discrete Pursuit

Source file name: `pursuit.c`, `pursuit.cpp` *or* `pursuit.java`

Robocops Inc. is a toy manufacturer company that develops a robot game in which a cop tries to catch a thief in a field that is simulated with a rectangular grid whose coordinates are denoted by pairs of integers. The resulting pursuit occurs in a discrete time scale: time is modeled with non-negative integers $0, 1, 2, \ldots$.

An object on the grid has a speed that determines how the object moves during the simulation. A *speed* is modeled with a pair of integers. The first one is the *horizontal speed* and the second one is the *vertical speed*. Additionally, horizontal and vertical speeds may vary in one unit (each one) to simulate slowing or accelerating.

More exactly: if at time k the object is at location (x, y) with speed (u, v), then, at time $k + 1$, the object may appear at location $(x', y') = (x + u + \epsilon, y + v + \delta)$, for some $\epsilon, \delta \in \{-1, 0, 1\}$. The speed at time $k + 1$ is $(x' - x, y' - y)$. For an object that moves with constant speed the values ϵ and δ are always 0.



At time 0, the cop is located at the origin of the grid, and the thief is at coordinates $(a, 0)$. The thief is moving with a constant speed; on the other hand, the cop starts still, but may vary his speed according to the given rules. Clearly, the cop catches the thief at time k if at this time the positions of both coincide.

Your task is to develop a program to control the cop in order to catch the thief in an efficient way. Your algorithm should determine the minimum time in which the cop may catch the thief.

Input

The problem input consists of several cases, each one defined by a line with three integer values, separated by blanks, that stand for the initial position a of the thief in the X-axis ($0 \leq a \leq 1000$), the horizontal speed u ($0 \leq u \leq 10$) and the vertical speed v ($0 \leq v \leq 10$) at which he moves. The end of the input corresponds to the end of the input file.

The input must be read from standard input.

Output

Output texts for each input case preserve the order in the input file.

For an input case, the output should be an integer that is the minimum time at which the cop may catch the thief.

The output must be written to standard output.

Sample input	**Output for the sample input**
1 1 1 3 1 0	2 3