Labyrinth

English version Wersja polska

The task requires writing a program able to steer a certain character – a player who is looking for the labyrinth's exit. Communication with the system has an interactive character: program is outputting the commands (marked blue in the description) and receives the information and statements in return from the system (marked red in the description).

Rules

The labyrinth's structure. Labyrinth consists of empty fields (marked with '.') and parts of the walls ('X' character). Only one element of external wall that surrounds the whole labyrinth is marked as an exit ('E' character). Player's position is marked as a plain field ('O' character). You can assume that the walls are made in a very elegant way which means that each element of the wall is adjacent to no more than 2 other wall elements; the parts of the wall which meet with their corners, have to have a common neighboring element of the wall on their edges. There is a complete darkness in the labyrinth. Player does not have a map of the labyrinth.

The beginning of the game. At the begging of the game player is at an unknown empty field inside the labyrinth. He has a torch with him and a certain amount of dynamite (n>=0), which is stated by the system's greeting statement:

You wake up with a headache, completely lost in a dark maze.
In your pockets you find a torch and *n* stick[s] of dynamite.

Actions allowed. Player is allowed to move around the labyrinth and use the torch or dynamite giving one of the following commands (to realize the "basic" functionality you can only use the first two ones):

- walk (north|south|east|west) *k* – it moves the player by k fields in a chosen direction. Each field that a player crosses should be empty or be an exit from the labyrinth. The return statement can be:
  - You are still lost in a dark maze. – if you did not find an exit
  - Against all odds, you have found the exit. Congratulations! – in case you entered a field marked 'E' (the game is considered to be over – you have completed the task and are granted victory)
- look around – it lights up eight fields surrounding you and returns a proper statement from the system, for example:
  The maze around you looks like this:
  X.X
  .OX
  .XX
- look (north|south|east|west) *k* – this command works very similarly to 'look around', but with a difference that looking to the chosen direction will have a k range (instead of 1). In case there is a wall that blocks the view in the row or column we have chosen, consecutive fields will not be displayed. An example return statement for 'look north 4' command can look like that:

The maze around you looks like this:
.XX
..X
X..
X.X
.OX
.XX
In that case the ‘look north 10’ command would give an identical effect because of the wall that blocks the view.

- use dynamite – this command will replace the eight fields surrounding the player with empty fields with no side effects for the player (although the number of dynamites he possesses is decreased by one). The return system’s statement would look like that:
  You light up a stick of dynamite.
  The explosion knocks you to the ground, but amazingly you survive.

Restrictions. Player cannot give more than 10^4 commands to the system in one game. You also have to be really careful about spending your resources: the amount of oxygen used cannot exceed 10^7, and crossing one field costs you 10 units of oxygen, using the torch costs you 1 unit of oxygen for every field you highlight and using the dynamite will cost you 1000 units of oxygen. If you exceed any of the limits or violate the rules (hitting the wall, using too much dynamite, giving an incorrect command) the system will output the following statement: STOP. *Problem description.* and game is considered to be over.

Input/Output

In one execution of the program we are playing t (t<20) games in different labyrinths. At the beginning of the game system outputs the t value and then all the games are played one after each other in a way mentioned earlier.

Program should clear the output buffer after printing each line. It can be done using fflush(stdout) command or you by setting the proper type of buffering at the beginning of theexecution - setlinebuf(stdout).

An example of the scenario can be found here.

Test data

Functionality for 0 to 2 points. Labyrinth covers the 50 by 50 board and contains only the outside wall. None of the labyrinth’s fields is located north or west from the player. The only instructions you need to use to fulfill those requirements are ‘walk … 1’ and ‘look around’. You have no dynamite.

Functionality for 2 to 5 points. Labyrinth covers the array 50 by 50 board. The only instructions you need to use to fulfill those requirements are ‘walk … 1’ and ‘look around’. You have no dynamite. Basic strategy is described below.

Functionality for 5 to 7 points. Labyrinth covers the 500 by 500 board. The labyrinth’s walls are made of no more than 500 segments and their summary length does not exceed 2500. The only instructions you need to use to fulfill those requirements are ‘walk … 1’ and ‘look around’. You

have no dynamite.

Functionality for 7 to 8 points. Labyrinth covers the 5000 by 5000 board. The labyrinth's walls are made of no more than 500 segments and their summary length does not exceed 25000. You have no dynamite.

Functionality for 8 to 9 points. Labyrinth covers the 50000 by 50000 board. The labyrinth's walls are made of no more than 500 segments and their summary length does not exceed 250000. You have no dynamite.

Functionality for 9 to 10 points. Labyrinth covers the 50000 by 50000 board. The labyrinth's walls are made of no more than 500 segments and their summary length does not exceed 250000. You have dynamite but only in the amount you really need.

Simple strategy – depth first search (up to 5 points)

DFS:
if (*current field is exit*) **exit the program**;
mark the current field as visited;
output 'look around';
read *information about the neighborhood*;
for each direction **from the set** {'west', 'north', 'east', 'south'}
if (field in the direction direction was not visited and is not a wall)
{
output 'walk direction 1';
execute recursive DFS;
output 'walk *opposed to* direction 1';
}

Example strategy – omitting obstacles (up to 10 points)

Pseudocode is less detailed; the outcome depends on the way you implemented it. We arbitrary choose one direction (for example: north) and try to move that way.

iteratively execute:
if (you can go north) go north;
else
{
try to go around the encountered obstacle.
if (*an obstacle cannot be omitted it means the player is trapped inside*)
use dynamite.
else *put the player at the edge of the obstacle that is farthest to the north*
}

Judge

To test your program you can use an arbiter used in the task. You can download a source code for g++ compiler here (Unix/Linux system or Cygwin for Windows).

Typical execution can look like that (of course in the SPOJ system it occurs very differently)

ppp07d-judge input_file tested_program – testing the playing program (outcome on the display)
ppp07d-judge input_file tested_program 2> log_file - testing the playing program (outcome in the log file)
ppp07d-judge input_file 2> /dev/null – manual game with an arbiter

where: input_file – file with a labyrinth description, tested_program – path to the playing program‘s executable.

The format of the input file looks like that: first there is t – the number of games. For each game there is the first line that contains six integers: *k* - number of segments, *x* and y – location of the exit, *x* and *y* – location of the player,*n* – number of available dynamite. Each of the following k lines contains the description of one segment which is represented by four integers *x1 y1 x2 y2* where x1<=*x2* and *y1*<=y2 and at least one of those conditions is an equality. The input file to generate a labyrinth from the example can be downloaded here.