Maze Challenge Homework

Your task, if you choose to accept it, is to write a maze solver to help a video game character find the shortest path through a maze. Your program will read a maze definition from a file, solve the maze and output the directions that our character needs to follow to go from the start location all the way to the end location.

But there's a twist. The maze is littered with mines and stepping on such a mine would result in the inevitable death of the character who is dutifully following your instructions. Thankfully, the character has three lives and, having blown up the mine, will be able to resurrect and resume his walk through the maze from where he died at most twice.

What we expect from you

Your program will need to read data from a file and print out a few lines of output on the screen. You can choose the language of your choice for this challenge as long as we can run and/or compile your code on a Windows, Mac or a Linux machine.

This challenge should take you around four to six hours to complete. When you are done, please send us your code and instructions on how to compile (if needed) and run the code.

This is an opportunity for you to show us not only your problem solving and adaptation skills, but also your coding and design skills. We hope to see some comments, unit tests and how you design your code.

The steps

1. Loading a maze

The character must walk through three mazes, each of which is fully defined by a string. All three mazes can be read from the file in the homework package: `mazes.txt`. Each line in the file represents a single maze.

A maze definition string is made of two parts: the maze size and the maze structure. Here is an example:

```
(3,3)-[34,14,12,6,77,5,1,19,9]

size        structure
```

The size represents the height and the width of the maze. The first number inside the parentheses is the height, and the second number is the width.

The structure part of the maze definition is the list of cells, or grid elements that make up the maze. You must read the numbers from left to right and position them on a maze grid starting from the top left corner of the grid and going from left to right, top to bottom. Loading the maze above should result in a maze where the cells are positioned like this:

```
34  14  12
 6  77   5
 1  19   9
```

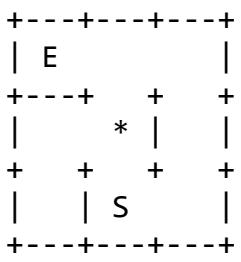
Each cell must now be converted into something meaningful by your program. Each number representing a cell holds the entire state of the cell. A cell can have multiple characteristics: it can be a start or end cell. It can also have openings to the top, left, bottom or right. Finally, it may or may not have a mine. Your program should take each cell code and extract from it each cell feature by using a *bitwise AND* operation against the feature code.

Here are the feature codes:

UP = 1
RIGHT = 2
DOWN = 4
LEFT = 8
START = 16
END = 32
MINE = 64

If you have never performed a *bitwise AND* operation before, please use whatever information you have at your disposal to help you. If you were to extract the features from code 77 using the list above, you would find that this particular cell would have a mine (64) and have openings to the left (8), bottom (4) and top (1).

Doing this for all the cells in the maze above, you would get something like this:



where **S** means the start of the maze, **E** means the end of the maze and ***** represents a mine. Your program does not have to display the maze.

2. Solving the maze

Having loaded a maze into memory, your program must now solve the maze and find the its shortest path that will keep our character alive. You can rest assured that all three mazes in the maze file have a solution.

Solving a maze is not trivial and we realize you may not want to spend a lot of time coming up with your own algorithm. Please feel free to use the Internet or books to find any algorithm of your choice as a starting point and tell us in a code comment which one you chose and why.

3. Show us the directions from start to finish

After you have solved the maze, please print on the screen a list of directions. Each direction must be written as ‘up’, ‘left’, ‘down’ and ‘right’ and be separated by commas. The whole string must also be surrounded by square brackets ‘[‘ and ‘]’. You can use single quotes or double quotes around the direction words.

For example, the solution to the maze drawn in the previous page is:

```
['up', 'up', 'left']
```

If our character had only one life, however, the solution would be:

```
['right', 'up', 'up', 'left', 'left']
```

Since the mazes.txt file has three mazes, we should see three lines of directions from your program on the screen.

3. Send us your work

Please send us your code or project back with clear instructions on how to compile it (if needed) and run it.

Good luck!