1 Solving Mazes (10 pts)

In this assignment you are expected to develop a Python module, namely `maze`, that comprises classes and functions that collectively solves a maze that is represented in a text file. In doing so, you need to follow the steps that are described in the following sections.

1.1 `MazeSolver` Class

```
class MazeSolver()
```

The module `maze` should include an implementation for the `MazeSolver` class. `MazeSolver` should have the following data members and methods.

`__init__(path)`

The constructor will accept a string `path` and create a `Maze` instance by utilizing `text_to_array` method, and the new `Maze` instance will be assigned to class member `_maze`.

`_maze`

The class member `_maze` will identify the `Maze` object for which the solution will be performed.

`_cellstack`

The class member `_cellstack` will identify a `Stack` object onto which maze cell coordinates will be pushed as tuples of `row_index` and `column_index`. As the maze is being explored, the cells to be explored will be pushed onto the stack. At each exploration iteration, the next cell to be investigated will be read from the top of the `_cellstack` with a `top` operation. If the cell being investigated does not have a valid neighbor, it is popped out from the `_cellstack`. At any moment, if the cell at the top position is the exit cell, that means `MazeSolver` reached to a solution. On the other hand, if somehow stack becomes empty, this will mean that there is no solution for the maze.

`_explored`

`_explored` will hold the list of the explored cells. For this class member, you may prefer to use a `list` object. During the execution of the solution algorithm, this data member will be accessed frequently to check whether a given cell is visited before.

[Bonus (2.5pts)] As you have already learned, containment checking with a `list` object takes $O(n)$ time in the worst case. In order to make these checks, maintain an ordered `_explored` list and implement a custom containment check operation that performs a binary search on the list so that this frequent operation can be done in $O(logn)$.

`text_to_array(self, path)`

`text_to_array` should read the file whose path is `path` and return a `Maze` object. In order to create a `Maze` object, the content of the input text file should be read and a `list` of `list`s should be constructed such that it can be indexed as a two-dimensional array. For example, for a text file such as the one shown below, it should create and return a list like this: `[['X','X','S']['X','O','O']['X','E','X']]` so that it should be able to be indexed such that, for example, the character at `(2,1)` would be `'E'`.

[Bonus (1 pt)] Write a single line of code that converts the content of the input file to `list` of `list`s. Please note that obtaining the file handle object should not be included in this one-liner code.

```
1 XXS
2 XOO
3 XEX
```

`get_a_neighbor(a_tuple)`

Given a tuple of row and column id, `get_a_neighbor(a_tuple)` computes the location of a **valid** adjacent up, right, down, or left cell and returns a coordinate tuple. Valid cells have characters `0` or `E` and does not exist in `_explored` list.

`solve_maze()`

`solve_maze` will provide a solution to the maze which is identified by `_maze` by adopting the following algorithm.

```
* Clear and initialize _cellstack and _explored.
* Push the starting cell to _cellstack.
* As long as _cellstack is not empty, loop:
  + Get the top cell c from the _cellstack
  + If c is not in _explored, add c to _explored.
  + If c is the exit cell, then the maze is solved, quit the algorithm
    and return the path (content of _cellstack) from start to exit.
  + Get a neighbor of c that is not visited before and push it to
    _cellstack, if there is no such neighbor, pop out c from the
    _cellstack.
 * return [(-1,-1)] indicating that there is no solution for the maze.
```

Also note that, if a solution is found, the list object having the path to the solution should start with the starting cell coordinates and end with the exit cell coordinates.

1.1.1 Usage of the `maze` Module

A typical usage of your `maze` module would be as follows.

```
import maze
ms = maze.MazeSolver('testcase.txt')
ms.solve_maze()
# Example Output:
# [(1,0), (1,1), (2,1), (3,1)]
# where (1,0) is start and (3,1) is the end cell
```

1.2 `Maze` Class

Build a `Maze` class that internally maintains a two-dimensional array of characters which represents maze structure. In such maze representation, there exists four distinct cell types: Moveable cells, walls, starting point, exiting point.

`Maze` class should have a constructor accepting a list object satisfying maze structure rules:

- List object has to be comprised of `list`s of characters.
- It should represent a mxn array where $m > 3$ and $n > 3$, and each of the internal `list`s should be of length n, and each cell of lists has to hold a single character (technically speaking, they have to be strings of length one).
- Maze structure have to include only `'O'` (movable path), `'X'` (wall), `'S'` (starting cell), and `'E'` (exit cell) characters. No other characters should be allowed. There should be exactly one `'S'` and one `'E'` character in the array.
- `'S'` and `'E'` characters should be at the *boundry* of the maze, in other words, at least one of the coordinates of cells holding either `'S'` or `'E'` characters has to be either 0, $n-1$, or $m-1$.

Constructor should accept only those `list` objects that satisfy these rules, otherwise it should raise an `InvalidMazeException`.

Indexing of the maze should start from the upper left corner of the array, in other words, the cell (0,0) should be the upper left corner of the two-dimensional array.

`Maze` class should have the following accessor functions.

`get_start()`

`get_start()` should return the starting cell (i.e., a tuple) of the maze. Each cell should be represented as a tuple `(row_index, column_index)`.

`get_exit()`

`get_exit()` should return the list of exit cell (i.e., a tuple) of the maze. Each cell should be represented as a tuple `(row_index, column_index)`.

1.3 `Stack` Class

Develop `Stack` class that implements the Stack ADT that we covered in the lecture. The implementation provided in the textbook could be adopted.

1.4 Example Test Cases

Below are some of the example text cases that might be helpful to you during the development. Text file including these examples is available on ODTUClass page. Please note that there will be many more test cases that we will use during the evaluation of your module.

1.4.1 Test Case 1 (Solution Exists)

```
2 XOOOOOOOXOOOX
3 SOXXXOXOXXOXX
4 XOXXOOOXX000X
5 OOOOOXOOOOXOX
6 XXOXOXXOXOXXX
7 XXOXOXXOXXOOE
8 XXOXOOOOOOOXX
```

1.4.2 Test Case 2 (No Solution Exists)

```
11 XOOOOOOOXOOOX
12 SOXXXOXOXXOXX
13 XOXXOOOXX000X
14 OOOOOXOOOOXOX
15 XXOXXXXXXOXXX
16 XXOXOXXOXXOOE
17 XXOXOOOOOOOXX
```

1.4.3 Error Cases

```
20 XOOOOOOOXOOOX
21 SOXXXOXOXXOXX
22 XOXXOOOXX000X
23 OOOOOXOOOOXOX
24 XXOXOXXOXOXXX
25 XXOXOXXOXXOOO
26 XXOXOOOOOOOXX
27
28 XOOOOOOOXOOOX
29 SOXXXOXOXXOX
30 XOXXOOOXX000X
31 OOOOOXOOOOXO
32 XXOXOXXOXOXXX
33 XXOXOXXOXXOOE
34 XXOXOOOOOOOXX
35
36 XOOOOOOOXOOOX
37 SOXXXOXOXXOXX
38 XOXXOOOXX000X
39 OOOOOXOOOOXOX
40 XXOXOX-OXOXXX
41 XXOXOXXOXXOOE
42 XXOXOOOOOOOXX
```

2 Delivery Instructions

Please hand in your module as a single file named as `maze.py` over ODTUClass by 11:59pm on due date. An Assignment-01 page will be generated soon after the start date of this assignment. Should you have any questions pertaining to this assignment, please ask them in advance (rather than on the due date) for your own convenience. Whatever IDE you use, you have to make sure that your module could be run on a Python interpreter (`$> python.exe maze.py`) or iPython (`%run maze.py`).