COSC 222 Lab 3 – Solving Mazes and Queues and Stacks

In this lab you will practice using Queues and Stacks by completing an implementation of a maze solver. In addition to starter code, ten maze files are given to you, `maze1.txt` to `maze10.txt`. Your source file is in the form of an exported eclipse project. Please import it by right-clicking in your package explorer, choosing **`import->general->archive file`**.

The content of the maze files looks like this:

```
10
0 5
9 2
#####.####
#####.####
##......##
##.####.##
.....#....
.########.
.####.....
#####.####
##....####
##.#######
```

The `'#'` symbols are walls or obstacles, and the `'.'` locations are positions you are free to move in.

The first number is the size (in this case, 10. Which means the maze is on a 10-by-10 board). The maze is a 0-indexed array, and it reads left-to-right, top-down (like a cursor on the screen). So the top row is row 0, the bottom row is row 9. The leftmost column is column 0.

The next two numbers are the position of the start of the maze. In this case, 0 5 refers to the top row, and the 6th position in the row. The next two numbers are the target coordinates. In this example, 9 2 refers to the last row and the 3rd position.

The code is set up for you to queue-up locations as you encounter them. A **`location`** is a single number that represents a position on the map. The top row

of this map can be described by locations 0, 1, 2, …, 9 and the second row will be 10, 11, 12, … , 19. The last row will be 90, 91, 92, …, 99.

In `Solver.java`, you are to implement functions to turn a (row,col) pair into a `location`, and to extract the row and column from a `location`.

You are given a `Maze.java` class which stores a maze object from which your code will copy its map and get its dimension, and its start and end coordinates. You are also given a file reader function that will read these maze files and create a maze object for you.

As you process locations from the map, you will make modifications to your map (by marking positions with an `'x'`) to keep track of what locations have been visited and queued. Also given is a `displayMyMap()` method that will display your map every iteration of the program. Please see

```
QueueingSolution1.txt QueueingSolution2.txt
StackSolution1.txt StackSolution2.txt
```

for sample executions that use `displayMyMap()`. This display method is initially commented out, and you are free to turn it on to watch your code run (once it can run) but it should be commented-out once again when submitting your work.

Rather than checking all your maze-solving paths, your program's execution will be tested by seeing if your number of iterations reported matches the expected solution, and also whether the largest size your queue (or stack) reaches during the execution matches the expected number. For these numbers to match up, you will be required to test locations in the order of North, East, South, West. That is, every time you pop a location off the queue, you check if North is a valid location on the map and it is an available location (by checking if it is equal to `'.'`) and if it is, add it to the queue. Repeat this for east, south, west. Remember that the South direction from your current position is where the row index is increased by 1, while the North direction is where the row index is smaller than your current position.

Please submit your code in the form of an **exported java project archive file (.zip).** Please do not submit **`.rar`** or **`.jar`** or **`.tar`** files.