CS2040S: Data Structures and Algorithms

Problem Set 7 Part 3

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so by leaving a comment at the start of your .java file. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 7. Mazes are Just Dynamite! Part 3

Unfortunately, it turns out that the walls you've so thoughtlessly broken using your superpowers in part 2 of this problem set, were actually sentient. Angered by your actions, you were attacked by the Greater Walls, and they managed to seal your superpowers somewhere deep inside their inner labyrinths.

Luckily, you somehow managed to escape before they could do you mortal harm. However, without your superpowers, you are nothing. You need to get them back, but to do so you need to traverse another maze, now full of the spirits of the walls you've broken. Being now powerless and helpless, you feel cowardly and afraid, and are scared to death by the Wall Spirits as well as the all-enveloping darkness of the inner labyrinths (maze).

Your fear level starts off at **0**, and every step you take in the maze increases your fear level by **1** (due to the all-enveloping darkness), unless you pass through a Wall Spirit while taking that step, in which case your fear level increases by scariness level of the Wall Spirit. This fear level increase happens every single time you pass through a Wall Spirit, even if the Wall Spirit is one you have encountered before.

The goal of this problem set is to get to your superpowers with the minimum possible fear level, lest you go mad with terror and let the darkness consume you.

Preliminaries. We generally adopt the same conventions for the maze as part 1 (please refer back to part 1's pdf if you have forgotten). The only change is that now our maze can now have characters other than ' ' (spaces) and `#` (hashes).

Here is a pictorial example of a 2 x 6 maze:

```
   0 1 2 3 4 5
  #############
0 # W a l l   #
  ### ##### ###
1 # M a r i a #
  #############
```

In this example, each Wall Spirit is represented by a character (we mostly used letters inside this example to keep it less confusing, but do note that there can be characters other than letters). The integer value of the character represents the scariness level of the Wall Spirit haunting that spot. The `#` (hash) character is reserved and represents a True Wall (not a Wall Spirit) that cannot be passed. The ' ' (space) character is also reserved and represents an empty space without any Wall Spirit or True Walls.

Now, using the first row of the example, we will trace through a small example of traversing the maze. Assume we start from (0, 1) and traverse to (0, 5), and the position of the player is represented by an asterisk (*):

```
   0 1 2 3 4 5
  #############
0 # W*a l l   #
  ### ##### ###
  Location: (0, 1). Fear level: 0

   0 1 2 3 4 5
  #############
0 # W a*l l   #
  ### ##### ###
  Location: (0, 2). Fear level: 0 + 97 = 97

   0 1 2 3 4 5
  #############
0 # W a l*l   #
  ### ##### ###
  Location: (0, 3). Fear level: 97 + 108 = 205

   0 1 2 3 4 5
  #############
0 # W a l l*  #
  ### ##### ###
  Location: (0, 4). Fear level: 205 + 108 = 313

   0 1 2 3 4 5
  #############
0 # W a l l  *#
  ### ##### ###
  Location: (0, 5). Fear level: 313 + 1 = 314
```

We provide you with two classes `Maze` and `Room` that represent the maze that your program will solve. The size of the maze is represented by the number of *rows* and *columns* in the maze. The maze itself is represented by a matrix of rooms. You will be able to check the four directions of the room through the public methods `getNorthWall()`, `getSouthWall()`, `getEastWall()` and `getWestWall()`. These functions return the integer value of the wall in each direction - if the wall is a True Wall (`#`), it returns `Integer.MAX_VALUE`, else if the wall is an empty space (' '), it returns 0, else it returns a **positive integer value** representing the scariness level of the Wall Spirit in that direction.

In this problem set, you will implement the class `MazeSolver` that will implement the provided interface `IMazeSolver`.

Please read the FAQ section at the end of this document before starting to code.

Problem 7.e. Exploring the Maze

Now implement the class `MazeSolver` that correctly implements the `IMazeSolver` interface. First, implement the method:

```
Integer pathSearch(int startRow, int startCol, int endRow, int endCol)
```

Your implementation for this function should return `null` if no path from the specified start room to the specified end room is found. If there are paths, then return the minimum possible fear level upon reaching the end room among all possible paths. There is no need to set the `onPath` variable for a room for this problem set, and we have removed it from the `Room` class.

Your `pathSearch` method should be **efficient** and **fast**, as you want to get your superpowers back as soon as possible.

Problem 7.f. Runtime Analysis

What is the asymptotic worst-case runtime complexity of your `pathSearch` method?

Take the number of columns in the maze (the number returned by `Maze.getColumns()`) to be `C` and the number of rows in the maze (the number returned by `Maze.getRows()`) to be `R`.

You are required to state the steps and reasoning you took to arrive at your final answer. If you require any other variables for your analysis, please define them clearly.

The following problems are bonus problems that award bonus marks, and you may choose not to do them.

Problem 7.g. A Change of Heart (Bonus Part 1)

For this bonus question, we make the following changes to what happens to your fear level when you pass through a Wall Spirit:

1. If the Wall Spirit's scariness level is strictly higher than your fear level right before you meet it, then your fear level increases to match that Wall Spirit's scariness level.

   Example: If you are at (0, 0) with fear level 2 and move to (0, 1) by passing through a Wall Spirit with a scariness level of 9, then your fear level at (0, 1) will be 9.

2. Otherwise, your fear level remains the same, since the Wall Spirit is not scary enough to increase your fear level.

   Example: If you are at (0, 1) with fear level 9 and move to (1, 1) by passing through a Wall Spirit with a scariness level of 2, then your fear level at (1, 1) will remain at 9.

Note that everything else remains the same - if you don't meet a Wall Spirit, your fear level still increases by 1 due to the darkness. Implement the following method:

```
Integer bonusSearch(int startRow, int startCol, int endRow, int endCol)
```

Your implementation for this function should return `null` if no path from the specified start room to the specified end room is found. If there are paths, then return the minimum possible fear level upon reaching the end room among all possible paths.

Problem 7.h. You are Filled with Determination (Bonus Part 2)
You receive information that there is a certain special room in the maze that contains a warm light. By reaching this room, your fear level gets set to **-1**, due to the darkness being dispelled and causing you to be filled with courage and determination. The modified rules for your fear level mentioned in Bonus Part 1 remain unchanged. Implement the overloaded method:

```
Integer bonusSearch(int startRow, int startCol, int endRow, int endCol, int sRow, int sCol)
```

The `sRow` and `sCol` parameters refer to the location of the special room. Your implementation for this function should return `null` if no path from the specified start room to the specified end room is found. If there are paths, then return the minimum possible fear level upon reaching the end room among all possible paths.

Note: There is no guarantee that there exists a path to the special room. There is also no requirement to stop at the specified end room if you pass by it, i.e. the path: start room $\rightarrow$ end room $\rightarrow$ special room $\rightarrow$ end room is still a valid path. For this problem, we also assume that start room $\neq$ special room, and end room $\neq$ special room.

Problem 7.i. Does it work? (Bonus Part 3)
Are you able to use the same algorithm behind your `pathSearch` to implement your `bonusSearch` methods? If yes, why? If no, why not? If the rules were changed such that your fear level becomes `min(fear level, scariness level)` while fear level still increases by 1 when encountering an empty space, would your algorithm still work?

In general, what conditions have to be fulfilled for your algorithm to work? Explain your thought process clearly.

Frequently Asked Questions

1. Are we allowed to use Java's PriorityQueue/TreeSet/TreeMap? They don't seem to have the `decreaseKey` functionality.

   Yes, you are allowed to use them. In place of `decreaseKey`, we allow you to re-insert the same object into the Collection, which means that there will now be two copies of the object in your collection, possibly with different priorities, but you know the one with the higher priority will be polled first. This incurs some overhead, but we will not penalize for it. For the purposes of this problem set, you may assume that this operation is completely equivalent to `decreaseKey`.

   For a slightly more complex but potentially better solution, you may consider using multiple data structures. For example, we can use a TreeSet to maintain priority order, together with a HashMap to store information that allows you to find a previously inserted object in the TreeSet. This allows you to compare priorities and replace the object in the TreeSet by deleting it and inserting your higher priority object.

   Finally, if you feel up to the challenge, you can implement your own custom priority queue that has the `decreaseKey` functionality. It's a good exercise! :)

2. How do I properly use the PriorityQueue/TreeSet/TreeMap?

   Have your objects implement the Comparable interface. More details can be found here.

   In case you are unfamiliar with the Comparable interface, here is an example for a Pair class that implements Comparable, where we order by ascending `x` then by ascending `y`:

```
class Pair implements Comparable<Pair> {
    int x;
    int y;

    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public int compareTo(Pair p) {
        // This function can be simplified, but for clarity's
        // sake we are more explicit about the checks
        if (x < p.x) {
            return -1;
        } else if (x > p.x) {
```

```
            return 1;
        } else {
            if (y < p.y) {
                return -1;
            } else if (y > p.y) {
                return 1;
            } else {
                return 0;
            }
        }
    }
}
```

Alternatively, create a Comparator class to pass to your PriorityQueue/TreeSet/TreeMap. More details can be found here.

3. Are we allowed to modify and upload files other than `MazeSolver` for our submission?

   No, you can complete the problems without modifying any of these other files.

4. Why are there so many bonus questions for this Problem Set?

   We made these bonus questions in hopes that students will rise to the challenge and try them for the purposes of learning and a bit of fun. We hope that you don't feel pressured to complete them, as they are additional. Also, the bonuses may look intimidating, but they are not especially hard.

5. Why is my code not working?

   - Check if you have implemented the `equals` methods if you're using custom objects, and make sure to compare objects using their `equals` method.
   - Check through your code for possible `NullPointerException`.
   - Think through the logic of your code, and test your code on edge cases.
   - Remove all `println`, unused imports, and your own testing code before uploading to Coursemology.
   - If all else fails, ask for help on the forums.