Assignment 3
COMP 250, Winter 2022

Prepared by Prof. Langer and T.A.s Yingzi, Hector, Rahul

Posted: Wed. March 2, 2022
Due: Wed. March 16 at 23:59 (midnight)

[document last modified: March 4, 2022]

General instructions

- Search for the keyword *updated* to find any places where the PDF has been updated.
- T.A. office hours will be posted on mycourses under the "Office Hours" tab. For zoom OH, if there is a problem with the zoom link or it is missing, please email the T.A. and the instructor.
- If you would like a TA to look at your solution code outside of office hours, you may post a *private* question on the discussion board.
- We will provide you with some examples to test your code. See file ExposedTests.java which is part of the starter code. If you pass all these tests, you will get at least 50/100. These tests correspond exactly to the exposed tests on Ed. We will also use private tests for the remaining points out of 100.

  We strongly encourage you to come up with more creative test cases, and to share these test cases on the discussion board. There is a crucial distinction between sharing test cases and sharing solution code. We encourage the former, whereas the latter is a serious academic offense.

  Ed should be used for code submission only. You should write, compile and test your code locally (e.g. in Eclipse or IntelliJ) for development.
- **Policy on Academic Integrity:** See the Course Outline Sec. 7 for the general policies. You are also expected to read the posted checklist PDF that specifies what is allowed and what is not allowed on the assignment(s).
- **Late policy:** Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Please see the submission instructions at the end of this document for more details.

Introduction

There are two parts to this assignment. The two parts are unrelated, so feel free to start with either one. This assignment is designed to give you practice with recursion. So your solutions must use recursion.

Part I: Finding the closest antenna pair (50 points)

Suppose there are two organizations A and B who each own some antenna towers on a grid of land. To communicate with each other, organization A needs to pick one of their towers to send signals to a tower owned by organization B, and vice versa. In order to minimize signal loss during transmission, they hope the distance between the two towers is minimized. Your goal is to find the closest pair of antenna towers for the two organizations to communicate with.

We use **`Point2D`** objects to store the locations of the antenna towers. You will need to write a class **`ClosestAntennaPair`** that finds the closest antenna tower pair among two sets of antenna towers belonging to the organizations A and B respectively. From here on, we will let A and B refer to the sets of points from each of the two towers.

The *brute force* way to solve this problem is to consider each pair of towers – one from each of A and B – compute the distance between them, and then choose the pair with the minimum distance. If there are n_A and n_B towers in A and B, respectively, then the number of pairs of towers to consider would be $n_A n_B$. However, depending on the arrangement of towers, it may be possible to avoiding checking all pairs. The solution for doing so involves recursion.

To understand how recursion can be used to solve this problem, consider a simpler version of the problem which is classical in computer science – the *closest pair problem*. In this problem, we have *one* set of 2D points in a plane and we wish to find the pair of points in that set that has the smallest distance. The solution to this problem is to *recursively* partition the set of n points into two sets of size $\frac{n}{2}$ – those with x coordinate less than or equal to the x-median in the set and those with x coordinate greater than the x-median. Then, recursively find the closest pair *within* each of these two sets. Let these minimum distances of the closest pairs in the two sets be δ_1 and δ_2 and let $\delta = min(\delta_1, \delta_2)$. The only remaining case to check is that there is a point in the left set and a point in the right set that are a distance less than δ apart. Such pairs can be found efficiently (see slide references below), such that the overall solution in the worst case is $O(n \log n)$. Note this worst case behavior is much faster than $O(n^2)$ which would be the time complexity for a brute force testing of all pairs in the original set.

Let's now return to the **`ClosestAntennaPair`** problem. You will need to come up with a similar recursive approach for this problem. The idea is again to choose an x value for splitting the A and B points into two sets each, and to recursively solve the problem for the left sets (A and B) and the right sets (A and B). The same x value should be used to split the two sets; however, note that this cannot guarantee that the A and B sets will be each split into equal size subsets. Indeed, if all the A points are to the left of all the B points, then the recursive approach will have similar time complexity as a brute force approach! The win for the recursive approach comes when the A and B points are well mixed, and in that case the time complexity will be closer to $O(n \log n)$ in practice, which is similar to the worst case in the classical problem.

To get you started on the **`ClosestAntennaPair`** problem, we provide you with solution code of the classic *closest pair problem* described above. You will need to read more about the classic problem on your own. For example, see slides here (p. 25-34) for details, but please share on the discussion board any other PDF/video resources that you find. The solution code for the classic problem is in the given file **`ClosestPair.java`**.

A code template for **`ClosestAntennaPair`** class is provided to you. This class has the following fields:

- **`closestDistance`**: a private double value that stores the closest distance
- **`counter`**: this field is for grading purposes and you don't need to worry about it.

It has implemented **`merge`** method which is exactly the same method as in **`ClosestPair`**. You can use this method as it is for the new problem. We also give you the **`distance()`** and **`getCounter()`** methods, which you should not change at all. Finally, we give you a template for the constructor and **`closest()`** method, which you need to implement:

- the constructor which takes two **`Point2D`** **arrays** as input; the constructor should follow exactly the same pattern as in **`ClosestPair`** class, except that you will have two sets of points rather than one set of points.
- the function **`closest`**, which computes for the best distance among points given in a certain range and updates the corresponding fields.

Do not remove any of these methods or modify the method signatures.

Hints/Remarks

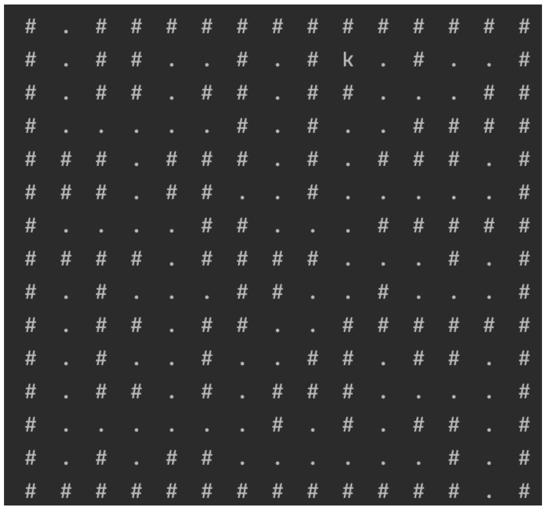
- We expect your solution to be very similar to the **`ClosestPair`** class provided. Your main task is to modify the provided code so that it works for the new problem. In order to do so, you will need to carefully read and understand the provided code. Indeed *we expect you to spend the majority of your time* understanding the given code. Moreover, you will need to understand some parts of the code more deeply than others. Figuring out which is part of your task, so please do not post questions on Ed asking which parts can be skipped. In general when programming, you should always know what a chunk of code is doing before using it in your own work; otherwise you might use the code incorrectly, which can lead to errors. If you are really stuck at understanding parts of the code, please come to office hours for help or post specific questions on the discussion board.
- We provide you with the class **`Point2D`** which is used to store 2-dimensional coordinate information of a point. [Updated Mar. 3] ~~Use this class for both parts of this assignment.~~ **Do not** modify this class - any modification on this class will be ignored when grading. You are free to call any public methods from this class in your solution.
- The **`Point2D`** class uses the **`Comparator`** interface, which has not been covered in the lectures but is straightforward. A **`Comparator<T>`** object can compare two objects of type `T` which can be used for making "greater/equal/less than" decisions and for sorting. For more details you can check the Java API and also here. You will also see how it works from reading the provided **`ClosestPair`** code. For this assignment, the main purpose of the **`Comparator`** is to give an ordering for sorting. We would suggest that you read the Java API for Arrays, specifically the various **`Arrays.sort()`** methods, and get some idea on how to use them with a **`Comparator`**. In particular, you can sort part of an array; be careful with indexing here!

Part II: Maze Solver (50 points)

For this second part of the assignment, you will implement a recursive solution to a maze. This is a specific instance of *graph traversal* problems that we will address later in the course.

Maze Setup

Each maze in our problem is represented by a 15×15 character array. At the initial phase, each cell of the maze will have one of two characters in it - a hash symbol (#) or a dot (.). There will also be a cell marked k. The hash symbol represents a wall, the dot symbol represents a valid step, and the k can be thought of as a "key" which the solution needs to reach. We will use the $(row, column)$ indices to represent a cell of maze so that **`maze[x][y]`** is row x and column y. Each maze will have a start position of $(0, 1)$ and a final position of $(14, 13)$ [Updated: Mar. 4.]. This means you can always assume both these positions will be marked initially with a dot symbol. Fig. 1 shows an example of a valid maze.



```
# . # # # # # # # # # # # # #
# . # # . . # . # k . # . . #
# . # # . # # . # # . . . # #
# . . . . . # . # . . # # # #
# # # . # # # . # . # # # . #
# # # . # # . . # . . . . . #
# . . . . # # . . . # # # # #
# # # # . # # # # . . . # . #
# . # . . . # # . . # . . . #
# . # # . # # . . # # # # # #
# . # . . # . . # # . # # . #
# . # # . # . # # # . . . . #
# . . . . . . # . # . # # . #
# . # . # # . . . . . . # . #
# # # # # # # # # # # # # . #
```

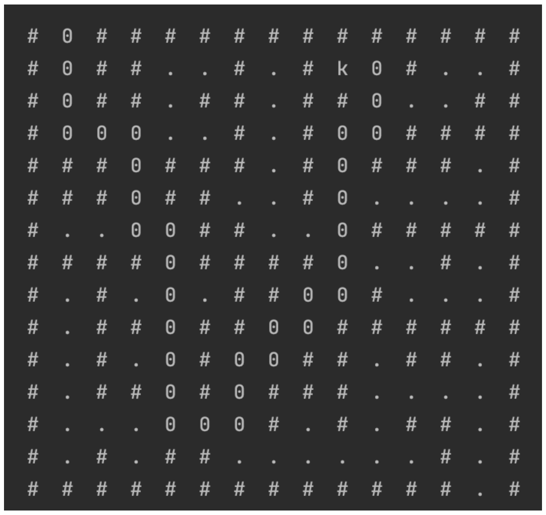
Figure 1: Valid Maze

Solving a Maze

We say a maze is solved if there are two paths marked. The first path is from the start position to the key k. The second path is from the key to the final position. Paths can only be over the valid cells, i.e, cells that were initially marked with a dot rather than hash symbol.

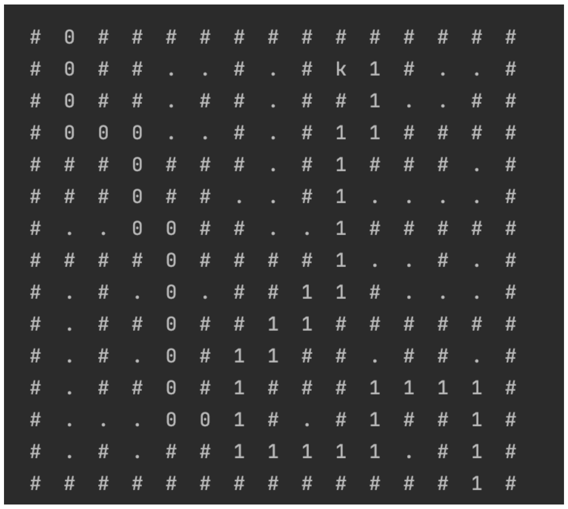
To solve the maze, the first path from the start cell to the key should be marked with a 0, and the second path from the key to the end should then be marked with a 1. See Fig. 2 for an example of how the array should be marked after the two stages. Note that after you mark the first path to the key, it is okay to overwrite any 0s with 1s, for the path from the key to the exit. Do not overwrite the key k; it remains part of the solved maze.

The goal is to find **any** valid solution to the maze. It is possible for there to be more than one solution. You just need to find any one valid solution. You will do this using recursion.



```
# 0 # # # # # # # # # # # # #
# 0 # # . . # . # k 0 # . . #
# 0 # # . # # . # # 0 . . # #
# 0 0 0 . . # . # 0 0 # # # #
# # # 0 # # # . # 0 # # # . #
# # # 0 # # . . # 0 . . . . #
# . . 0 0 # # . . 0 # # # # #
# # # # 0 # # # # 0 . . # . #
# . # . 0 . # # 0 0 # . . . #
# . # # 0 # # 0 0 # # # # # #
# . # . 0 # 0 0 # # . # # . #
# . # # 0 # 0 # # # . . . . #
# . . . 0 0 0 # . # . # # . #
# . # . # # . . . . . . # . #
# # # # # # # # # # # # # . #
```

(a) Path to key is marked



```
# 0 # # # # # # # # # # # # #
# 0 # # . . # . # k 1 # . . #
# 0 # # . # # . # # 1 . . # #
# 0 0 0 . . # . # 1 1 # # # #
# # # 0 # # # . # 1 # # # . #
# # # 0 # # . . # 1 . . . . #
# . . 0 0 # # . . 1 # # # # #
# # # # 0 # # # # 1 . . # . #
# . # . 0 . # # 1 1 # . . . #
# . # # 0 # # 1 1 # # # # # #
# . # . 0 # 1 1 # # . # # . #
# . # # 0 # 1 # # # 1 1 1 1 #
# . . . 0 0 1 # . # 1 # # 1 #
# . # . # # 1 1 1 1 1 . # 1 #
# # # # # # # # # # # # # 1 #
```

(b) Both paths are marked

Figure 2: Marking of each path

Starter Code

You are provided with a class **`Maze`** which has the following methods already defined for you:

- **`char[][] load_maze(String directory)`** : This method takes in the name of a txt file that has a valid maze and returns a char array representing it.
- **`void printMaze(char[][] maze)`** : This method takes in a char array representing a maze and prints it out.
- **`solveMaze(char[][] maze`**) : This is driver method that calls the recursive method **`solveMazeUtil()`**.
- **`boolean canMove(char[][] maze, boolean found, int x, int y)`** :
  This method should return true if (x, y) is a valid cell to move into and false otherwise.

  The boolean variable **`found`** represents if they key has been found yet or not, namely it indicates if we are either in the stage of finding the first path to the key (false) or the second path from the key to the exit (true). If the key k has already been found, you can move and overwrite cells marked as 0, i,e, any cell used in the first path to the key.

Your task

Write the following method:

- **`boolean solveMazeUtil(char[][] maze, boolean found, int x, int y)`**
  This method *must be written recursively*. It solves the maze problem by marking the 0's and 1's appropriately on the maze. It returns true if the maze could be solved and false otherwise. See below for hints of how to go about writing this.

Here is the main strategy for writing the recursive method **`solveMazeUtil`**. At the current position, mark this position as part of the path and then recursively call the method up to four times, namely by moving one step to the right, left, up, or down. (Diagonal is not allowed.) In calling the method recursively from each of these new positions, you are trying to find a valid path from these position. If you can find a valid path, then return true; if you cannot find a path, then the current position must be appropriately (re)marked and you should return false.

We are not specifying the base cases. You need to figure them out for yourself. Please do not post them publicly on the discussion board.

Submission

Please follow the instructions on Ed Lessons Assignment 3.

- Ed does not want you to have a package name. Therefore you should remove the package name before uploading to Ed.
- We will check that your file imports only what is in the starter code.
- You may submit multiple times. Your assignment will graded using your most recent submission.
- If you submit code that does not compile, you will automatically receive a grade of 0. Since we grade only your latest submission, you must ensure that your latest submission compiles!
- The deadline is midnight Wed. March 16. On Ed, this deadline is coded as 12:00 AM on Thurs. March 17. Similarly, on Ed, the two day window for late submission ends at 12:00 AM on Sat. March 19.

Good luck and happy coding!