COMS 3137 Data Structures and Algorithms. Homework 6

Submit your electronic files via `http://courseworks.columbia.edu`. Do the written assignments electronically if possible, since it is easier for all parties involved. Use the provided makefile for the programming problems.

Directory structure to be followed for submission:

```
Root folder : <Your UNI>_homework6
<all Java source files>
Readme File (describing all your files and notes to the graders)
Theory solutions (optional)
Makefile (optional if you modified the provided Makefile)
```

I.e., this directory structure should mean that typing "make" in your submission directory will compile your code with no errors. Archive your submission directory using the command (in the containing directory):

```
tar -czvf <Your UNI>_homework6.tar.gz <Your UNI>_homework6
```

1. If you are creating your classes within packages, please maintain the package directory structure during submission.
2. Include the makefile if you need to change the makefile provided to you.
3. The code should be commented as appropriate or the details should be explained in a readme.
4. If you are handing your theory on paper, do not print out your code.

Multiple Submissions: You can submit multiple times, but we will only consider the latest submission based on the timestamp in courseworks. Please give at least 1-2 minutes between two submissions.

Late Submissions: Any unexcused late homework submissions will be penalized 10% each day it is late. The penalty begins at the beginning of class the day the assignment is due. The 10% penalties will continue for 3 full days at which point no more late submissions will be graded. If you are submitting late, do not use courseworks and mail your submission to all the TAs.

Upload your archive to the Class Files section of courseworks in the Homework 6 subdirectory. It is also recommended that you keep a pristine copy of your submission folder in case there is any submission error.

1 Written Problems (32 points)

Make sure your solutions are clear. Diagrams and math are often insufficient to convey exactly what you mean, so supplement with some text. Either pseudocode or Java are acceptable when asked to provide algorithms. Nevertheless, clear, concise English is often preferable.

1. (4 points) **Weiss 7.4** Show the result of running Shellsort on the input 9, 8, 7, 6, 5, 4, 3, 2, 1 using the increments $\{1, 3, 7\}$. [Obviously, you will be graded on intermediary steps, so show your work; i.e., don't just submit 1 2 3 4 5 6 7 8 9]

2. **Weiss 7.17 a and b** Determine the running time of mergesort for

   (a) (2 points) sorted input

   (b) (2 points) reverse-ordered input

3. (4 points) **Weiss 7.19** Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quicksort with median-of-three partitioning and a cutoff of 3. [Show intermediary steps and use the convention that you choose the first, middle, and last elements for the median-of-three pivot.]

4. (4 points) **Weiss 7.38** Suppose arrays A and B are both sorted and both contain N elements. Give an $O(\log N)$ algorithm to find the median of $A \cup B$.

5. **Weiss 8.1 b and c** Show the result of the following sequence of instructions: `union(1,2)`, `union(3,5)`, `union(1,7)`, `union(3,6)`, `union(8,9)`, `union(1,8)`, `union(3,10)`, `union(3,11)`, `union(3,12)`, `union(3,13)`, `union(14,15)`, `union(16,0)`, `union(14,16)`, `union(1,3)`, `union(1,14)` when the unions are:

   (a) (0 points; **Don't do this**) Performed arbitrarily

   (b) (4 points) Performed by height

   (c) (4 points) Performed by size

   [Draw the trees, not the arrays]

6. (2 points each) **Weiss 8.2** For each of the trees [just b and c] in the previous exercise, perform a `find` with path compression on the deepest node.

7. (4 points) **Weiss 8.6** Prove that for the mazes generated in Section 8.7, the path from the starting point to ending points is unique.

2 Programming Problems

1. (14 points) **Weiss 8.15** Write a program that generates mazes of arbitrary size. Use Swing to generate a maze similar to that in Figure 8.19.

   - Your program should be called with `java MazeGUI <size>`, which will build and display a maze of size `<size>`.
   - Use the disjoint set class provided by the textbook (or write your own if you prefer).

- One way to interpret the maze-building algorithm is to think of an initial grid-graph such that each square in the maze is a node and each node is connected to its north, south, east and west neighbors. Then build a *random* spanning tree, such that there is some path from the start to the end. However, since your program will just be building a maze, it is up to you if you want to write code in terms of graphs or specifically for the maze.
- The provided files Maze.java and MazeGUI.java provide minimal starting points and demonstration code of how to draw lines (which is all you'll need to do). The code in Maze.java currently draws the complete grid, so you will need to modify this to draw only the remaining walls after you have built your maze.
- The textbook uses the upper left corner as the start and the lower right corner as the end of the maze. You may do this, but you are also free to choose random start and end squares.

2. (14 points) Maze solving. Add a function that solves the maze. Using depth first search starting from the start square, find the path to the end square and display the path in a GUI output of the maze.

   - One way you could do this is to draw red dots in the middles of the squares along the path. See the included starter code for an example.
   - Make sure your final output shows only the unique path from start to finish.
   - You may want to display the unsolved maze in one window and the solved maze in another, but you are free to organize this however you wish (you may even want two separate programs); If your maze is built correctly and it is solved correctly, you'll receive full credit on both programming problems.