COSC-211: DATA STRUCTURES
HW10: DIJKSTRA'S ALGORITHM

Due Monday, April 30, 11:59pm

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's code, and do not show anyone your code (except for me and the course TAs).

1 Introduction

In this assignment, you will implement Dijkstra's algorithm to find the shortest path from a single vertex to all other vertices in a graph. In doing so, you will reinforce your understanding of Dijkstra's algorithm, you'll get practice working with a graph represented as an adjacency matrix, and you'll revisit the heap data structure from earlier in the course.

1.1 Setup

Begin by downloading some starter code:

```
wget -nv -i https://goo.gl/Yc7Doy
```

You should now have three Java files:

- `Node.java`: represents a vertex in the graph. Right now all `Node` contains is an index, a `compareTo` method that doesn't do anything interesting, and a constructor that sets the index. You likely will find that you want to add more to `Node`.
- `Heap.java`: some code that you'll recognize from earlier assignments as a heap implementation of a priority queue. I'm giving you the entire, working heap this time! The heap relies on the `compareTo` method in `Node` to give a priority ordering of `Node` objects, so it won't do anything interesting until you rewrite `compareTo` to make it do something meaningful.
- `Dijkstra.java`: some starter code for your Dijkstra's algorithm implementation. Specifically, right now the code does the following:
    1. Creates a new $n \times n$ array of `int`s, which will be an adjacency matrix representing the graph.*
    2. Calls the `fillRandomGraph()` method to put some edges in the graph, with weights in the range 0-9. Right now each edge is created with probability 0.8 and the edge weights are selected uniformly at random. Feel free to change what this method does if you want to test your code on different types of graphs.
    3. Creates an array of `Node` objects, one for each vertex in the graph.

  *Here n is an input parameter to the program. For example, if you run the program using the command `java Dijkstra 5`, this will generate a graph with 5 nodes. If you don't specify a command-line argument to the program, the default is a 10-node graph.

2 Your Tasks

Your job in this assignment is to implement Dijkstra's algorithm to find the shortest path from vertex 0 (i.e., the node with index 0, which is in position 0 of the array created in the starter code) to every other vertex in the graph. Specifically, you will do the following:

1. In `Node.java`, add any fields that you might need to keep track of information in the course of your Dijkstra search.

2. In `Node.java`, change the `compareTo` method to make it do something meaningful. The method takes a `Node other` as input, and should return a negative number if `this` is smaller than `other`, 0 if they are equal, and a positive number if `this` is greater than `other`. This method is used to compare nodes in the heap that you'll use for your Dijkstra search, so your `compareTo` method should provide an answer that means the right thing in the context of the Dijkstra search.

3. In `Heap.java`, write a method called `decreaseKey()` that takes a `Node` and an `int` as input and, if the input `int` is smaller than the input `Node`'s current key, updates the `Node`'s key and re-heapifies the heap.

4. In `Dijkstra.java`, write a method called `dijkstra` that takes as input an `int[][]` representing a graph's adjacency matrix and a `Node[]` containing a list of the graph's nodes, and runs a Dijkstra search to find the shortest path from node 0 to every other node in the graph. By the end of this method, each `Node` should be storing its own information about its distance from node 0.

5. In `Dijkstra.java`, write a method called `printPath` takes as input a `Node` and prints out the indices of all nodes along the shortest path from node 0 to that node. (This is similar to the task, in HW9, of finding the path from exit back to entrance once you have solved the maze. In the mazes, it was less clear why we need to do that given that there's a clear visualization of the solution path. Here, there's not a good way of visualizing what path we took in the graph, so it's useful to have a method to reconstruct solution paths.)

3 Submit your work

Make sure you **test your code thoroughly** before submitting it. Code that does not compile will not receive credit.

Submit all of your Java files using either the submission web site or (from `remus/romulus`) the command line:

```
$ cssubmit *.java
```

This assignment is due on Monday, April 30, 11:59pm.