CMPSC 250
Analysis of Algorithms
Fall 2012
Bob Roos

Lab 8
Tues., 13 November 2012
Hand In by 2:45pm Tues., 20 Nov.

From now on I will be deducting points from programs that do not contain header comments for each method you write (including their purpose, their return value if any, and the meanings of the parameters) and descriptive comments for the program as a whole (including comments on what it does and how to use it, e.g., command line arguments and their meanings), descriptions of any variables that you declare, etc. I will also be deducting points for any documents, including programs, that fail to contain your name and the lab number.

The `EdgeWeightedGraph` Class

Make sure you know how to create edge-weighted graphs either by reading from a file or by explicitly adding edges. The process is a little bit different from the `Graph` and `Digraph` examples since we have a brand new class to deal with: the `Edge` class. Here's a comparison between the `addEdge` methods for `Graph` and `EdgeWeightedGraph`:

Creating a graph:

```
...
Graph g = new Graph(2);
g.addEdge(0,1);
...
```

Creating an edge-weighted graph:

```
...
EdgeWeightedGraph g = new EdgeWeightedGraph(2);
g.addEdge(new Edge(0,1,.874));
...
```

Enumerating the neighbors of a vertex is also different, since an adjacency list consists of a collection of `Edge`s, not vertex numbers. For instance, here is how we would print the neighbors of a vertex `v` and compute the degree of `v` in a `Graph` and in an `EdgeWeightedGraph`:

Graph:

```
...
int degree = 0;
for (int w : g.adj(v)) {
   StdOut.println(w);
   degree++;
}
...
```

Edge-weighted graph:

```
 ...
 int degree = 0;
for (Edge e : g.adj(v)) {
    StdOut.println(e.other(v));
    degree++;
}
 ...
```

The `Edge` class has a method named `other` that tells you the name of the "other" endpoint if you give it an endpoint. In the code above, we are processing edges leaving `v`, so we use `e.other(v)` to ask what vertex is at the other end of edge `e`. (There is also a method named "`either()`" that returns one of the two endpoints of an edge.)

Programs `WeightDemo1.java` and `WeightDemo2.java` and data file `wg` give examples of creating edge-weighted graphs.

1. Write a Java program that reads in an edge-weighted graph, prints it back out, and computes and prints the total weight of all the edges in the graph. Test it with the `wg` file I provided and with an input file of your own that describes a connected weighted graph with 6 vertices and 10 edges. **CAUTION:** Remember that these are *undirected* graphs, so each edge appears twice in the adjacency lists, once for each endpoint.

2. Write a Java program that creates and prints a *complete* edge-weighted graph where every edge has a randomly-assigned weight of either 1 or 2. The 1s and 2s should be assigned with equal probability (which is not quite the same as saying that there should be equal numbers of 1s and 2s). Recall that a graph is *complete* if every pair of vertices is connected by an edge.

   The size of the graph should be specified by a command-line argument, e.g., "`java Lab8Prob2 5`" should output a complete graph with 5 vertices and random weights of 1 or 2 on each edge.

   Run it for sizes 2, 5, and 8.

3. **Special Instructions!** If you are stumped by this—not even sure where to begin—see me. The goal of this exercise is to help you put some things together from other parts of the course; it is not "hard," but even so, **I expect you to ask for help if you get stuck!**

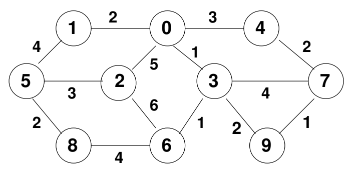
   Write a Java program that inputs an edge-weighted graph from a file, prints it out, and then performs a depth-first search starting at vertex 0, printing out each vertex as it is marked and the weight of the edge that was followed to reach that vertex. (See next problem for an example of an output format.) Each vertex should be visited exactly once.

   Test your program on file `wg` and on two files of your own; your graphs should have at least 8 vertices, should be connected, and should have at least 16 edges apiece.

   There are probably a few choices you will need to make for this—for instance, do you want to create a static boolean `marked` array, or do you want to pass it into the dfs as a parameter? I don't have a preference! Please program this in any way that "works for you." Just be sure to document how your program works.

4. **[OPTIONAL.]** This is a bit more challenging. Write a Java program that inputs an edge-weighted graph from a file, prints it out, and then performs a *greedy depth-first search* starting at vertex 0, printing out each vertex as it is marked and the weight of the edge that was followed to reach that vertex.

   A *greedy* depth-first search is like an ordinary depth-first search, but one in which the unmarked neighbor connected with the largest edge weight is chosen first. For instance, in the following weighted graph, the order of visitation would be 0, 2, 6, 8, 5, 1, 3, 7, 4, 9.



Ask me if you don't see why this is the only possible correct output. This graph is described in file `wg2`.

One way to format your output might be:

```
Visiting 0
Using edge of weight 5.0
Visiting 2
Using edge of weight 6.0
Visiting 6
Using edge of weight 4.0
Visiting 8
 ... etc. ...
```

Test your program on file `wg` and on two files of your own; your files should have at least 8 vertices, should be connected, and should have at least 16 edges apiece.

HINT: Efficiency is not a concern here, so you have lots of options. For instance, during the DFS you can make a preliminary pass through the adjacent edges of the current vertex `v` and put the edges into a max priority queue, then remove them one by one. There is a class named `MaxPQ` in the files accompanying your textbook; you would initialize, add to, and remove from such a queue using:

```
MaxPQ<Edge> pq = new MaxPQ<Edge>();
 ...
pq.insert(e);
 ...
Edge e2 = pq.delMax();
 ...
```

5. Move up one level from your `lab8` directory and type "`zip -r lastnamelab8.zip lab8`" (if you named it something else, use the something else). Now upload this to your drop box.