CSS 343 Assignment #3

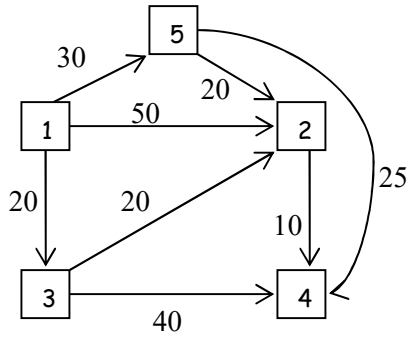
Part 1, Programming Graph ADT (emphasis on Dijkstra's shortest path algorithm)
Implement Dijkstra's shortest path algorithm, including recovering the paths. You will find the lowest cost paths and display the cost and path from every node to every other node. Another display routine will output one path in detail.

In the data, the first line tells the number of nodes, say n (assume nonnegative integer). Following is a text description of each of the 1 through n nodes, one description per line (assume 50 chars max length). After that, each line consists of 3 ints representing an edge. (Assume correctly formatted data, int data.) If there is an edge from node 1 to node 2 with a label of 10, the data is: `1 2 10`. A zero for the first integer signifies the end of the data for that one graph. There are several graphs, each having at most 100 nodes. For example:

Sample Input

```
5
Aurora and 85th
Green Lake Starbucks
Woodland Park Zoo
Troll under bridge
PCC
1 2 50
1 3 20
1 5 30
2 4 10
3 2 20
3 4 40
5 2 20
5 4 25
0 0  0
```

picture (not part of data)



Sample Output to display all (use this **exact** format, but blanks need not be exact):

```
Description               From node  To node  Dijkstra's  Path
Aurora and 85th
                              1        2       40          1 3 2
                              1        3       20          1 3
                              1        4       50          1 3 2 4
                              1        5       30          1 5

Green Lake Starbucks
                              2        1       ---
                              2        3       ---
                              2        4       10          2 4
                              2        5       ---

Woodland Park Zoo
                              3        1       ---
                              3        2       20          3 2
                              3        4       30          3 2 4
                              3        5       ---

Troll under bridge
                              4        1       ---
                              4        2       ---
                              4        3       ---
                              4        5       ---

PCC
                              5        1       ---
                              5        2       20          5 2
                              5        3       ---
                              5        4       25          5 4
```

Sample Output to display one path, for: `G.display(1,4);`

```
   1       4       50          1 3 2 4
Aurora and 85th
Woodland Park Zoo
Green Lake Starbucks
Troll under bridge
```

Part 1 Notes

-- For this lab (including part 2,) you may assume the input data file has correctly formatted data, e.g., that there will be 3 ints on one line of data for an edge (not 4 ints or chars, etc.). You must always do data error checking, e.g., that an int you get is a valid value for the problem. Ignore invalid data, meaning don't use in graph. You may assume the first int, the number of nodes in the graph, is valid.

-- Class includes the adjacency matrix, number of nodes, TableType array, and an array of NodeData. You do not need to implement a complete Graph class. The only methods you must have are the constructor, buildGraph (put in edge costs), insertEdge(), removeEdge(), findShortestPath(), displayAll() (not general output, uses couts to demonstrate that the algorithm works properly as shown), and display (to display one shortest distance with path). (Some utility functions are needed.) For insertEdge() and removeEdge(), you figure out reasonable parameters, return type and document their use.

```
class GraphM {
public:
   ...
private:
   struct TableType {
      bool visited;          // whether node has been visited
      int dist;              // currently known shortest distance from source
      int path;              // previous node in path of min dist
   };

   NodeData data[MAXNODES];              // data for graph nodes information
   int C[MAXNODES][MAXNODES];            // Cost array, the adjacency matrix
   int size;                             // number of nodes in the graph
   TableType T[MAXNODES][MAXNODES];      // stores Dijkstra information
};
```

-- The T in Dijkstra's algorithm (has dist, visited, and path), is a 2-dimensional array of structs. T is used to keep the current shortest distance (and associated path info) known at any point in the algorithm. In lecture it was shown as one-dimensional since the source was fixed at one. This is row one in the 2D array. To adjust in use, add [source] as the row. For example, T[w].dist becomes T[source][w].dist .

The data member T is initialized in the constructor (or a routine that the constructor calls): sets all *dist* for the source node (source row) to infinity, sets all *visited* to false, and sets all *path* to 0.

-- The pseudocode given is for only one source node. Another loop, controlling the source, allows for the shortest distance from all nodes to all other nodes. (The nodes start at array element one. The zero element is not used, or used otherwise.)

```
for (int source = 1; source <= nodeSize; source++) {
   T[source][source].dist = 0;

   // from lecture, finds the shortest distance from source to all other nodes
   for (int i = 1; ...) {
      ...
   }
}
```

-- Part 1 in main will look similar to the following:

```
// for each graph in file data31.txt, find the shortest path from every node to all others
for (;;) {
   GraphM G;
   G.buildGraph(infile1);
   if (infile1.eof()) break;
   G.findShortestPath();            // find shortest distance and path
   G.displayAll();                  // display shortest distance and path
   G.display(3, 1);
}
```

An example driver is supplied that tests only the basic functionality of your code.

-- The design of this graph class needs discussion. The findShortestPath() might not be a member function. Since there are data members having to do with Dijkstra's, it is best if it runs for every graph and run again if the graph changes (via insertEdge() or removeEdge()). A bool could keep track of the graph changing (you do not need to implement this). At any rate, the design is for simplicity, to easily test the algorithm with less of a focus on good object-oriented design.