HW 5:   Graph Algorithms
100 points (85 points deliverables, 15 points for design)
Due 11pm May 11, 2021
This is an individual assignment
Instructions:

1. **Read and follow the contents of Spring 2021 CSCI 335 Programming Rules document on Blackboard (in the 'Course Information' Section).**
2. **Read the assignment description below.**
3. **Submit only the files requested in the deliverables at the bottom of this description to Gradescope by the deadline.**

Learning Outcome: The goal of this assignment is to become familiar with heaps, disjoint sets and graph algorithms. Acknowledge the sources you use in the README.GRAPH file.

Question 1: Adjacency   **25 points**

You will read a directed graph from a text file. Below is an example, found in Graph1.txt:

```
5
1 2 0.2 4 10.1 5 0.5
2 1 1.5
3 2 100.0 4 50.2
4
5 2 10.5 3 13.9
```

The first line is the number of vertices. Each vertex is represented by an integer from 1 to N. Each line is of the form:

```
<vertex> <connected vertex 1> <weight 1> <connected vertex 2> <weight 2> …
```

For each vertex, you have a list of the adjacent vertices with positive edge weights. For instance, in the above example, vertex 1 is connected to vertex 2 (edge weight 0.2), to vertex 4 (edge weight 10.1) and to vertex 5 (edge weight 0.5). Vertex 2 is connected to vertex 1 (edge weight 1.5), vertex 4 has no outgoing edges, etc.

Represent the graph using an underlined adjacency list.
Create your graph class inside **graph.h**

In order to test your implementation, you will read a second text file (let us call it **AdjacencyQueries1.txt**) that will contain a set of pair of vertices. Your program (name it **CreateGraphAndTest.cc**) will have to first create the graph by reading it from text file **Graph1.txt**. It

will then open the file **AdjacencyQueries1.txt** and for each pair of vertices in it, your program will print whether the vertices are adjacent or not, and if they are, your program will print the weight of the edge that connects them.

For example, if the file AdjacencyQueries1.txt contains:

```
4 1
3 4
1 5
5 1
1 3
```

Then the output should be:

```
4 1: not_connected
3 4: connected 50.2
1 5: connected 0.5
5 1: not_connected
1 3: not_connected
```

where the numbers to the right of "connected" are the weight of this edge.

*Note: This example is consistent with Graph1.txt. It is very important that you follow this format exactly. Make sure you have the "_" underscore in "not_connected". **Make sure there is nothing extra printed.***

So, your program can be called for example as:
./CreateGraphAndTest <GRAPH_FILE> <ADJACENCY_QUERYFILE>

Exact deliverables are described at the bottom of the file.

Question 2: Dijkstra's Algorithm Implementation **(60 points)**

Using your graph implementation in Q1, implement Dijkstra's Algorithm, **using a priority queue (i.e. heap).** Write a program that runs as follows:

./FindPaths <GRAPH_FILE> <STARTING_VERTEX>

This program should use Dijkstra's Algorithm to find the shortest paths from a given starting vertex to all vertices in the graph file.

The priority queue data structure is provided for you under the file, **binary_heap.h,** and included inside **graph.h. Do not modify binary_heap.h, we will use the original automatically.**

You should then print out the paths to every destination. For example, if you run the program having as input Graph2.txt (provided) starting from vertex 1, i.e.

```
./FindPaths Graph2.txt 1
```

Then the output should be:

```
1: 1 cost: 0.0
2: 1 2 cost: 2.0
3: 1 4 3 cost: 3.0
4: 1 4 cost: 1.0
5: 1 4 5 cost: 3.0
6: 1 4 7 6 cost: 6.0
7: 1 4 7 cost: 5.0
```

The first number is the target vertex. (aka: the last line beginning with 7: is the shortest path from the input starting vertex 1, to the target vertex 7.) **Your output should always be ordered this way, starting from target vertex 1, to the maximum target vertex for that graph.**
Following the target vertex, display the path taken, **inclusive of the starting and ending vertices**. Finally, display "cost:" followed by the cost of that path.

Note: This is an example using the inputs provided in Graph2.txt. It is very important that you follow the output format exactly. Do not print any extra lines or characters.

If there does NOT exist a path to a particular vertex in the graph, the output for the path to that vertex should display (note the underscore!): **not_possible**

Example: Vertex 3 has no vertices pointing to it. Under the command .**/FindPaths <graph_file> 2** the result for the path to 3 should be:

3: not_possible

If a vertex does not point to any other vertex, and that vertex is used as the starting vertex argument, your program should output **not_possible** for every path, besides the path to itself.

Example: Vertex 3 is a vertex that does not point to any other vertex. Under the command, **./FindPaths <graph_file> 3** the output should display:

1: not_possible
2: not_possible
3: 3 cost: 0
4: not_possible

These examples do not represent every possible graph, but this is enough to determine the output for every possible situation.

Deliverables: You should submit these files:

MAKE SURE TO SUBMIT ALL RELEVANT FILES. CHECK YOUR INCLUDE STATEMENTS. DO NOT PRINT ANYTHING OTHER THAN WHAT IS ASKED

- **README.GRAPH file**

Question 1

- **CreateGraphAndTest.cc**
  - `testGraphDriver()` – All functionality should be in here, NOT in the main function.
- **graph.h**
  - Use this for the relevant graph class and its routines.

Question 2

- **FindPaths.cc**
  - `findPathDriver()` – All functionality should be in here, NOT in the main function.
- **graph.h**
  - Use this for the relevant graph class and its routines. Modify with the relevant functions needed by `findPathDriver()`

Do not submit or modify binary_heap.h and dsexceptions.h. The originals will be included automatically in our testing when you submit.