Programming Problem: Jonhson's Algorithm on a Sparse Graph

[12 points for the programs (no analysis part this time)]

For this problem, you get to implement Johnson's algorithm for a sparse graph (where it might perform much better than, say, the Floyd-Warshall algorithm). Johnson's algorithm has two stages. In the first stage, it uses the Bellman-Ford algorithm to re-weight the graph edges so as to eliminate negative edge weights. For the second stage, it runs Dijkstra's algorithm $|V|$ times, each time with a different vertex as the source. The first stage should be easy to implement; the Bellman-Ford algorithm isn't particularly complicated. Likewise, the second stage shouldn't be too difficult, since you've already implemented Prim's algorithm, and Dijkstra's algorithm is almost the same.

Implementation Language

You get to implement your solution in Java, C or C++, whichever language you prefer. The name of your source file will depend on your language:

- For Java : johnsons.java
- For C : johnsons.c
- For C++ : johnsons.cpp

When we test your solutions, we'll compile and run them on an EOS Linux machine using the following commands. Be sure to try these out yourself to make sure you program will compile and work when we try it out. Your program will read its input from standard input (i.e., from the terminal), but, like in the following examples, we'll use I/O redirection to get it to read from a file instead.

- For Java

```
javac johnsons.java
java johnsons < input-2.txt
```

- For C

```
gcc -Wall -std=c99 -g -O2 -o johnsons johnsons.c
./johnsons < input-2.txt
```

- For C++

```
g++ -Wall -g -O2 -o johnsons johnsons.cpp
./johnsons < input-2.txt
```

Input Format

The input will describe a weighted, directed graph (remember, for Prim's algorithm, we assumed the input was an undirected graph). The first line will contain two values. The first is a positive integer, V, giving the number of vertices in the graph. The second is a non-negative integer, E giving the number of edges.

The next E lines of input each describe one of the (directed) edges. An edge is given as three integers a b and w, where w is the weight of an edge from vertex a to b. Vertices are indexed from 0 to $V-1$. The w value may be positive or negative.

After the list of edges, the input will have a non-negative integer, k, giving the number of queries. This will be followed by k lines, giving the indices of a source vertex followed by a destination vertex.

The following shows sample input, `input-2.txt`. This file describes a graph with 10 vertices and 16 edges. Some edges have negative weight (for example, the edge from vertex 3 to vertex 0). After the graph, there are 5 query pairs (for example, the first query is asking about the shortest path from vertex 6 to vertex 7).

```
10 16
0 1 5
0 2 3
1 4 4
2 5 1
3 0 -2
3 1 1
3 6 6
4 7 -1
5 3 4
5 8 1
6 5 -3
6 4 -2
7 6 4
7 9 3
8 6 5
9 8 -1
5
6 7
4 5
9 0
7 4
5 1
```

The number of vertices, the number of edges and the total length of any simple path fit in a signed, 32-bit integer. For any pair of vertices, i and j, there will be at most one edge from i to j (although there may be an edge from vertex i to j and a different edge, with different weight going from i back to j).

Program Output

For each query pair, your solution should print a line of output to standard output. For query i j, report the length of the shortest path from i to j. Report this in the following format, where *len* is the length of the shortest path.

i `->` *j* `=` *len*

If there is no path from i to j, print the following line as output.

i `->` *j* `= x`

If the input graph has a cycle with a negative total edge weight, then don't respond to any of the queries. Instead, just print the following error message and exit immediately. Recall that Bellman-Ford includes a step for detecting cases like this, so your solution can report this message before it even starts running Dijkstra's algorithm.

```
Negative edge weight cycle
```

Graph Representation

You should assume the input is a sparse graph, so represent it as an adjacency list (rather than an adjacency matrix). This should help make both the Bellman-Ford and the Dijkstra's algorithm

stages of the algorithm faster (faster than what you'd get with a dense representation).

Computing Shortest Paths

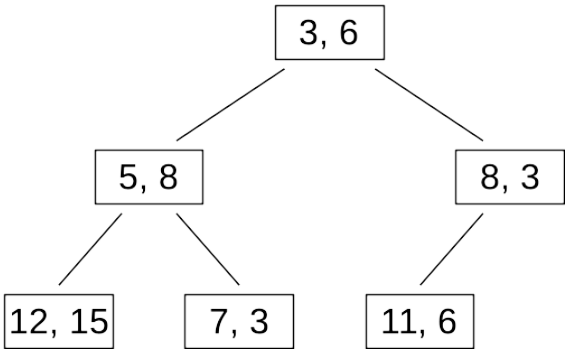
Compute the lengths of all shortest paths before you start responding to any of the queries. You'll do this by filling in a $|V| \times |V|$ table of shortest path lengths, where the element at row i and column j is the length of the shortest path from vertex i to j. Each time you run Dijkstra's algorithm with a new source vertex, you'll be filling in one row of this table.

Remember that Johnson's algorithm re-weights the graph to eliminate negative-weight edges. This changes the total weight of each shortest path, but it doesn't change the sequence of edges along that path (i.e., any path that was a shortest path before the re-weighting will still be a shortest path after the re-weighting). To report the length of a shortest path, you have to take the re-weighting into consideration (to figure out what the total weight of the path would have been before the re-weighting of the graph). Given the definition of the re-weighting operation, this part is easy to do (so, you get to figure it out yourself).

Heap Implementation

If you're happy with your Prim's algorithm implementation from homework 4, you should be able to re-use much of that code for this problem, with just a few changes to turn it into Dijkstra's algorithm. If you use your heap from homework 2, you can just set the branching factor to 2 (rather than making it depend on the size of the graph).

If you're not happy with your solution, or you'd just like to try something new, you can use a standard heap implementation for your Dijkstra's algorithm (e.g., the STL priority_queue or PriorityQueue from the Java collections framework). These heap implementations generally won't support an efficient technique for performing the decrease-key operations, so you'll have to use an alternative technique. Instead of performing a decrease-key, you can just insert a new heap entry with a new, lower key value. The figure below shows what your heap might look like. Each entry will contain a key / vertex pair, but some vertices may have multiple entries in the heap. For example, vertex 6 has two entries in the heap. The entry with a key of 11 must have been inserted earlier and the entry with a key of 3 must have been inserted later, when we found a shorter path to vertex 6.



If some vertices have multiple entries in the heap, your program will need to be able to ignore the out-of-date entries. Fortunately, this is easy to do. For any vertex, the lowest-key value is the only one you need to consider when you're growing a spanning tree[1]. In the heap above, vertex 6 has the minimum key, so it's the next one that will go in the spanning tree. Later, its out-of-date entry, with a key of 11, will also be removed from the heap. Your program needs to be able to notice that vertex 6 has already been added to the spanning three, so this out-of-date entry for vertex 6 should be ignored.

[1]For Dijkstra's algorithm, we're growing a spanning tree, much like we do with Prim's algorithm. It's just that Dijkstra's algorithm defines the spanning tree by minimum path length rather than minimum total weight.

For fun, you can consider how this technique affects the asymptotic runtime of Dijkstra's algorithm (but we're not asking you to write this up). Permitting multiple heap elements for each vertex will give us a larger heap, so all the heap operations will be more expensive.

Sample Execution

We're providing five sample inputs with this problem. These include problem instances with only positive edge weights, some with some negative edge weights, a large test case and even one with with a negative edge weight cycle. If you run your program on these inputs, here's what you should expect:

- For Java

```
java johnsons < input-1.txt
2 -> 5 = 1
2 -> 3 = 15
6 -> 4 = 21
1 -> 5 = 15
0 -> 6 = 8
java johnsons < input-2.txt
6 -> 7 = -3
4 -> 5 = 0
9 -> 0 = 3
7 -> 4 = 2
5 -> 1 = 5
java johnsons < input-3.txt
Negative edge weight cycle
java johnsons < input-4.txt
0 -> 4 = -3
0 -> 5 = -2
5 -> 0 = x
java johnsons < input-5.txt
68 -> 959 = 47
560 -> 525 = 42
31 -> 614 = 7
879 -> 520 = 61
947 -> 912 = 118
972 -> 603 = -3
... Lots of output lines omitted ...
738 -> 942 = 2
58 -> 176 = 108
```

- For C or C++

```
./johnsons < input-1.txt
2 -> 5 = 1
2 -> 3 = 15
6 -> 4 = 21
1 -> 5 = 15
0 -> 6 = 8
./johnsons < input-2.txt
6 -> 7 = -3
4 -> 5 = 0
9 -> 0 = 3
7 -> 4 = 2
5 -> 1 = 5
```

```
./johnsons < input-3.txt
Negative edge weight cycle
./johnsons < input-4.txt
0 -> 4 = -3
0 -> 5 = -2
5 -> 0 = x
./johnsons < input-5.txt
68 -> 959 = 47
560 -> 525 = 42
31 -> 614 = 7
879 -> 520 = 61
947 -> 912 = 118
972 -> 603 = -3
... Lots of output lines omitted ...
738 -> 942 = 2
58 -> 176 = 108
```

Submitting Your Work

In Moodle, you'll see an assignment named `Assignment 5`. Submit the source code to your program using this link. Of course, you'll still need to submit your write-ups for the other, formal problems via GradeScope.