**Due Date:**
Wednesday, February 26, 2014

**Delivery**
Create the archive, pa2.tar from your files. Deliver the archive to csc3320@orion.ucdenver.edu as an attachment to an email. Put PA2 in the subject field and your name in the body.

**Program objectives**
The objectives of this assignment are as follows.
An ability to use current techniques, skills, and tools necessary for computing practice (ABET i).

**Value**
This program is worth 15 points. The distribution of points will be as follows.

| Criterion | Value |
|---|---|
| Templates | 2 |
| Inclusion model | 1 |
| Program style (see below) | 3 |
| Annotated output | 7 |
| Usage of pseudocode | 2 |

**Background**
Suppose you are travelling between Aurora, CO and Broomfield, CO using the RTD. Given an RTD map of Denver and surrounding communities, you notice that there are several routes you could take, but you are interested in taking one that costs the least (there may be more than one). You could, of course, enumerate all routes and pick the least costly. As you soon discover that this can take an inordinate amount of time, you call your favorite computer scientist and ask if there is a faster way to determine a minimum-cost route. The answer you get is encouraging—YES!

**Problem**
You just finished shopping at Macy's at the Aurora mall, but they didn't have your size in one of the shirts that you fell in love with. The sales clerk checked inventory on a computer and discovered that the shirt was available in your size at the Macy's in the Flat Irons Crossing mall. Since you have been shopping all day, you are famished and want to stop at your favorite Italian restaurant, Maggie's, which is within walking distance of the Flat Irons Crossing mall. Today is Wednesday and Maggie is featuring Lasagna ($14.25), Spaghetti ($13.95), and Gnocchi's ($13.49). All prices include tax. Your mode of transportation is The Ride (RTD). You have $77.90 in your pocket, and the shirt cost $60 (tax included). After you take the bus to the Flat Irons Crossing mall and pay for the shirt, what is the most expensive meal you will be able to afford that includes a 15% gratuity? See below for the bus routes and their costs in cents.

Using the picture below, model the problem with a graph $G = (V, E)$ and determine a lowest cost bus route from Aurora to Broomfield using the algorithmic method shown below. The

graph $G = (V, E, W)$ is weighted and undirected. The edgeweights $w_i$, $i$=1, 2, 3…..$|E|$ represent the costs to travel along each subpath $p_i \sim p_{i+1}$ where the path $p$ (source to destination) is the path $p_1 \sim p_2 \sim p_3 \ldots \sim \ldots p_n$ and each subpath $p_i \sim p_{i+1}$ represents an edge $e_i$ with a weight $w_i$.

**Input**

1. A file listing a set of weighted edges $e = (u, v, w)$, $e \in E$, $u$ and $v \in V$ and $w$ is the weight (cost) associated with the edge $(u, v)$. The edges are listed in the data file, one triple per line with space-delimiting between elements of the triple. The first line of the file specifies the number of verticies and the number of edges, space-delimited. The second line contains a list of space-delimited verticies. The third line and thereafter contains the edge triples.
2. Command line args to specify the start and destination verticies

**Output**

1. A shortest path from source to destination as a list of verticies in order from start to finish.
2. The amount of pocket money remaining after paying the bus fare, and the meal you were able to purchase at Maggie's.
3. A representation of the original graph, either as an *adjacency list* or an *adjacency matrix*.

**Minimum program requirements**

1. Display a greeting (pause the display, then prompt to continue).
2. Write global function templates and non-templates.
3. You may have as many classes/structs as you like.
4. The compilation of the function templates should follow the inclusion model.
5. The output should be annotated.
6. Error checking where appropriate.
7. Use the provided pseudocode for computing the shortest path. These algorithms should be templates.
8. Create a priority queue as specified below.

**Requirements for Priority Queue<T>**

A *priority queue* (*PQ*) is a data structure that maintains a collection of elements such that the element with the highest priority is always at the front or top of the *PQ*. A *priority* is an attribute of the type T. For pa2, a priority will be based on an edge weight—the lower the weight, the higher the priority. You may use *any* data structure you like to represent the *PQ*. The *PQ* must support the following operations.

**push(const T& elem)**

Insert *elem* into the *PQ* based on its priority attribute.

**pop()**

Remove the element with the highest priority (always at the top). The next highest priority element moves to the top after the pop.

**top()**

Returns, but does not remove, the element at the front of the *PQ*.

**decrease_key(const T& elem, int new_key)**
If elem.key < new_key, then elem.key = new_key. Otherwise, do nothing. Notice that a *decrease_key* operation may corrupt the ordering (possible that new_key is smaller than the element's key at the top of the queue.

**empty()**
Returns *true* if the *PQ* is empty, *false* otherwise.

**size()**
Returns the number of elements in the *PQ*.

**Pseudocode for single-source, shortest path**

```
SSSP(G, s)          // G is the input graph and s is the source vertex
                    // s is a vertex, s ∈ V. Each vertex has the following
                    // attributes: Distance field designated as d, predecessor
                    // field designated as π, and label field which represents a
                    // vertex name. G may be represented as an adjacency matrix or
                    // an adjacency list. The priority queue will be ordered based
                    // on the value of the d parameter of each vertex. Notice that
                    // the call to Initialize orders the PQ so that the source
                    // vertex is at the front. The value of d for a particular vertex
                    // can be adjusted by the Relax algorithm while it is still in
                    // the PQ.
    Initialize(G, s)
    S = ∅           // An empty set to store the shortest path verticies
    Q = V           // A min priority queue where priority is based on edge weight
                    // with lower weights having higher priority. The verticies are
                    // stored arbitrarily in V, but when inserted into Q, will have
                    // priority queue order, based on the d attribute of each vertex.
    while ! Q.empty( )      // Loop as long as Q is not empty
        u = Q.top( )        // Get element at the front of the queue
        Q.pop( )            // Remove front element
        S.insert(u)         // Insert u into S which stores the verticies in a
                            // a shortest path as it is being computed.
        for each vertex v on the adjacency list of u
            Relax(u, v, w) // Relax edge (u, v, w). Notice that this could require
                           // a Q.decrease_key( ) operation


Initialize(G, s)
    for each v ∈ V
        v.d = ∞
        v.π = NIL
    s.d = 0
```

**Relax( *u*, *v*, *w*)** // $(u, v, w)$ is a weighted edge, $e \in E$
**if** $v.d > u.d + w$
$v.d = u.d + w$
$v.\pi = u$

**Notes**
1. You may use any data structure you like to build the priority queue.
2. I expect to see new types in addition to a priority queue.
3. At a minimum, parameterize the priority queue and shortest path algorithms.