Section 7.4: Dijkstra's Algorithm (Version 1)

Determines the length of a shortest path from a source vertex **`S`** in a connected weighted graph *G* to each of the other vertices of *G*. The vertices in the graph G will be processed and the set of vertices can be stored either as an adjacency matrix or as an adjacency list.

We will also store the weights of the edges as an `n` × `n` weight matrix `w`:

```
     w[i][j] = weight of edge from vertex i to vertex j
```

The vertices will be processed one at a time. Each vertex will be an object of a class `Node`:

```
Node
     vertex        (type integer)  // index or label of vertex (1..n)
     weight        (type integer)  // total min weight on path from S
     predecessor   (type Node)     // predecessor on min weight path
```

The set of vertices yet to be processed will be stored in a structure L which must support operations:

- initialize structure
- check if empty
- findMin (determine which vertex in L has smallest current weight)
- remove vertex from structure

```
algorithm  Dijkstra(G, start)    // graph G, source vertex start

     for each vertex  u      // to be stored in the data structure L
     {
          u.weight = ∞       // use maxInteger or other flag
          u.predecessor = null
     }
     start.weight = 0
     start.predecessor = start

     L.initialize(V(G))      // initially L contains all vertices

     while !L.empty()
     {
          x = L.findMin()    //  x is vertex in L with min weight
          for each vertex  y  in  L  that is adjacent to  x
          {
               if  x.weight + w[x.vertex][y.vertex]  <  y.weight
               {
                    y.weight = x.weight + w[x.vertex][y.vertex]
                    y.predecessor = x
               }
          }
          L.remove(x)
     }
```

running time (depends on data structure):

http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html

http://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html

https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

What design technique does Dijkstra's algorithm use? How do you know?
Greedy.
At each step, the algorithm picks the vertex of smallest total weight and sticks with that choice (does not backtrack).

Running time: Suppose G has n vertices and m edges.

first loop, setting fields for each vertex is $O(n)$
call to **L.initialize** depends on data structure used for L
if L is an array or linked list, then L.initialize is $O(n)$
call to **L.empty** is likely $O(1)$ (just check if pointer is null or if count of L is 0)
while loop:
in each iteration of the while loop, one vertex is selected (using findMin), processed (scan all vertices adjacent), and then removed; so the while loop iterates n times

n calls to findMin (first with n vertices in L, then n-1, then n-2, …)
if L is an unsorted array or linked list
worst case: $n + n - 1 + n - 2 + \ldots + 2 + 1 = O(n^2)$ □ "expensive" part
if L is a sorted array or linked list, then there would be an extra cost to sort during each iteration (the weights change for the vertices, so you would have to re-sort)
worst case: n*lg n + (n-1)*lg (n-1) + (n-2)*lg(n-2) + … + 2*lg 2 □n^2 * lg n

processing each vertex (scanning the adjacent vertices)
while and for loop combined
if adjacency matrix is used => $O(n^2)$ since it scans through every row and every column
if adjacency list is used => $O(m)$ since only the adjacencies are stored

n calls to **remove**
unsorted array: shifting of elements in array is required,
worst case: $n + n - 1 + n - 2 + \ldots + 2 + 1 = O(n^2)$
(doubly) linked list: $O(1)$ for each call (total is $O(n)$)

Total time, worst case for **unsorted array or unsorted linked list: $O(n^2)$**

Can we improve on this running time?

Yes! With better data structures!!!