a) **Dijkstras(Graph, root)**

/\* First, I initialize all the vertices to have a minDistance to infinity (Integer.MAX\_VALUE), set its parent to null, and visited to false. \*/

**For each vertex (node) v in the Graph**

distance[v] = infinity (Integer.MAX\_VALUE);

parent [v] = null;

visited = false;

distance[root] = 0;

Q = buildMinHeap(Graph); //this is done in linear time, comments on runtime below

**while**(Q is **NOT** empty)

u = findMin(); //node in Q with the smallest distance

u.visited = true; //mark the node visited

**for each neighbor v of u**

combinedDistance = distance[u] + weight (u,v);

**if**(combinedDistance < distance[v])

distance[v] = combinedDistance;

parent[v] = u;

extractMin(); // remove u from the minHeap and heapifyDown on root node

b) My pseudocode runs in O((E+V)(log(V)). The initialization stage of the Dijkstra’s algorithm takes linear time O(V) because it has to initialize the distance and parent of every vertex in V.

When I build the heap after initialization, building a minHeap take linear time O(V) as well, and is explained in further detail below.

We will run Dijkstra’s until the Q is empty, which means you will run it on every vertex in V. This takes O(V) time. For every vertex v in Q we need to check each edge. We find a vertex v that has the smallest distance in Q, which is done in constant time O(1), since we are maintaining a minHeap structure. Relaxing the edge and comparisons of distance are also done in constant time. After we do our comparisons on v and mark it visited, we remove it from the Q (minHeap). By removing it from the minHeap, we are removing the root value of the minHeap and inserting the last value of the minHeap at the root. This could, worst case take log(V) times because the node at the root could bubble down at most the height of the heap tree number of times. Overall, my algorithm runs in worst case O((E+V)(log(V)). Dijkstra’s algorithm will run in worst case O((E+V)(log(V)) time, because for every V and every neighbor that V has (# of edges E), they will swap at most log(V) number of times, which is the height of the tree.

/\* in buildMinHeap \*/

First, we set queue Q as the graph Graph and note that Graph is maintaining a heap structure.

In a heap structure, there are n layers (height of the tree, log(V)). Starting from the n-1 layer and working backwards until you reach the root node, go to the first node that has a leaf (which would be the midpoint of the heap/array structure) and call heapifyDown on that node. This maintains the linear running time complexity of O(V), since for every node starting from the midpoint we call heapify down starting from the bottom layer up to the top layer. We start from the midpoint, because calling heapify down on the nodes at the bottom layer is done in constant time. We will do the least work at the bottom layer where there are the most nodes, and the most work at the top layer with the fewest node. At most each node would only have to bubble once.

heapifyDown(int i){

int index = i;

**while**(the vertex v at index i in the heap has a left child){

int smallest\_child = find the smallest child of v at index i

**if**(if the value of vertex v is larger than the value of its smallest\_child){

swap positions

index = smallest\_child; //update the new index to run heapifyDown on

}

else {break;}

}

}

c) If an adjacency matrix was used my algorithm’s runtime complexity would change to O(n^2). Instead of checking each neighbors’ weights through an ArrayList of nodes, I would have to go through an ArrayList of ArrayList of nodes to find the neighbors. This adjacency matrix would be of size N x N. If our graph was sparse, indicating that the number of edges is around the number of vertices, then checking for neighbors would take O(n^2), because you would be iterating over a matrix of size NxN to check the vertices of every vertex. There would be a lot of memory and time wasted traversing the adjacency matrix the graph was sparse (# vertex approximately equal to # edges), because each entry you’re iterating over would be mostly 0’s. If we used a heap implementation for a sparse Graph, the time complexity would be O(nlogn), since each node would only have to bubble at most the height of the tree number of times. If the graph G was dense, indicating that the number of vertices << number of edges, then the adjacency matrix implementation would still maintain O(n^2), while the heap would take O(n^2logn). This is because, for each neighbor we would have to iterate over every single edge, and if there are a lot of edges, you would have to traverse a very long linked list before you got to the edge you wanted. If the graph is sparse, use an adjacency list, if the graph is dense, use an adjacency matrix.