Dijkstra's Algorithm(lazy implementation)

- Single Source shortest path algorithm
- Works for a graph G(V,E) with non negative edge weights!
- Why: To ensure that once a node has been visited it's optimal distance cannot be improved!
- So it acts in a **greedy** way: by selecting the most promising node ever time.

- Logic:
  > Maintain a **Priority Queue** of key-value pairs (node,distance) which tells you which node to visit next.
  > So we maintain a **distance array** where the distance to every node = oo
  > Kickstart the algorithm:
    > Insert [s,0] into the Priority Queue.
    > Loop while(P.Q. is not empty {
        pulling out the next most promising [node,distance] pair
      }
    > for every node we visit:
      > Iterate over all edges outwards from the current node & relax each edge appending a new [node, distance] pair to the P.Q. for every relaxation.
    > Having duplicate key entries in P.Q. is what makes this specific implementation lazy!

- Pseudo Code:

```
# Returns an array that contains the shortest distance to every node from source(s)
# g: adjacency list of our weighted graph
# n: num of nodes in our graph
# s: index of source node (0<= s <=n)

dijkstra(g,n,s):
   vis = [false, false, …,false]; # size = n
   dist = [oo, oo, …, oo]; # size = n
   dist[s] = 0;
   init(pq); # initialize an empty priority queue
   pq.insert([s,0]); # kickstart the algorithm

   while(pq.size() != 0){
      index, minValue = pq.remove(); # remove most promising index, distance
      vis[index] = true; # mark index as visited

      for(edge: g[index]){
         if(vis[edge.to]){
```

```
            continue; # so we won’t visit them again!
        }
        newDist = dist[index] + edge.cost;
        if(newDist < dist[edge.to]){
            dist[edge.to] = newDist;
            pq.insert([edge.to, newDist]);
        }
    }
}
return dist; // return array with optimal distances
```

- Finding the **optimal path** :

If we want to not only find the optimal distance to a specific node but alse “what sequence of nodes were taken” to get there, we need to track additional info: <The index if the previous node> we took to get to our current node.

```
Prev = [null, null, …, null]; # array of size n
  .
  .
  .
if(newDist < dist[edge.to]) {  # Relaxation
   prev[edge.to] = index;
   dist[edge.to] = newDist;
   pq.insert([edge.to, newDist]);
}
```

- find the **shortest path between 2 nodes:**

```
# g: adjacency list of our weighted graph
# n: number of nodes in the graph
# s: index of source(starting) node
# e: index of end node
```

findShortestPath(g, n, s, e):

```
    dist, prev = dijkstra(g,n,s);
    path = [] ;  # empty list, will be in reverse order
    if (dist[e] == oo){
        return path;
    }
    for(at=e; at!=null; at=prev[at]){
        path.add(at);
    }
    path.reverse();
    return path;
```