**1. Suppose all edge weights in a graph are integers in the range from 1 to |V| where V is the set of vertices. How fast can you make Kruskal’s algorithm run? Explain.**

In Kruskal's algorithm, after sorting the edges in non-decreasing order of weight, we iterate over these sorted edges and add them to the minimum spanning tree (MST) if they do not create a cycle. To detect cycles efficiently, we use a disjoint-set data structure (such as the union-find data structure). The time complexity of Kruskal's algorithm depends on the time complexity of sorting the edges and the time complexity of performing union-find operations. Sorting the edges takes O (E log E) time, where E is the number of edges in the graph.

However, in the given scenario where the edge weights are integers in the range from 1 to |V|, we can exploit the fact that there are only |V| distinct edge weights. This allows us to use Counting-Sort instead of a general-purpose sorting algorithm like quicksort or merge sort.

Counting-Sort has a time complexity of *O (V + E)* in this case because it correctly sorts n integers in the range 0 to k in O (n + k) time. Here, n refers to the total number of elements to be sorted, which is E in this context, and k is the range of distinct edge weights, which is |V|. Therefore, sorting the edges using Counting-Sort will take O (V + E) time. After sorting the edges, the subsequent operations involve performing union-find operations to detect cycles and add edges to the MST. The time complexity of union-find operations can be bounded by O (V log V) using path compression and union by rank heuristics.

Combining the time complexities, we have O (V + E) for sorting the edges and O (V log V) for the union-find operations. Since the sorting dominates the time complexity, the overall time complexity of Kruskal's algorithm in this scenario is O (E + V log V).

**2. Given an MST for an edge-weighted graph G, suppose that an edge in G that does not disconnect G is deleted. Design an algorithm using which we could find the minimum spanning tree for the modified graph easily without constructing the new tree from scratch again. What is its running time?**

*Assuming that the edge was in the original minimum spanning tree (MST) of the edge weighted graph G* (which would be redundant because it would make the MST to have a cycle), in order to find the MST for the modified graph after deleting an edge that does not disconnect the graph, we can follow these steps:

* Identify the two connected components: Start by deleting the edge from the original MST. This deletion will result in two disconnected components, Sub-MST1 and Sub-MST2.
* Find the minimum weight edge between the components: Iterate through all the remaining edges in the graph and find the edge with the minimum weight that connects Sub-MST1 and Sub-MST2. This can be done by checking if the endpoints of an edge belong to different components. The cut property guarantees that this edge will be part of the MST for the modified graph.
* Replace the deleted edge with the minimum weight edge: Replace the deleted edge with the minimum weight edge found in the previous step to obtain the new MST for the modified graph.

The running time of this algorithm depends on the number of edges in the graph, denoted as E. The first step of identifying the two components can be done in O (1) time since the deleted edge does not disconnect the graph. The second step requires iterating through all the remaining edges (BFS), which has a worst-case complexity of O(E). Finally, replacing the deleted edge with the minimum weight edge can be done in O (1) time. Therefore, the overall running time of this algorithm is **O(E)**.

*Assuming that the edge was not in the original minimum spanning tree (MST) of the edge weighted graph G*, in order to find the MST for the modified graph after deleting an edge that does not disconnect the graph, we can follow these steps:

* Let the edge that was deleted be denoted as e = (u, v), where u and v are the endpoints of the edge. Add the edge e to the MST, creating a new graph T' = T ∪ {e}, where T is the original MST. This addition introduces a unique cycle C in the graph T'.
* To find the cycle C, we can perform a breadth-first search (BFS) starting from vertex u (or v) in the graph T' while ignoring the edge weights. During the BFS, keep track of the parent of each vertex encountered. Once the BFS completes, we will have identified the cycle C and obtained the parent array, which represents the path from u to v in the cycle.
* Traverse the parent array starting from v until we reach u. At each step, keep track of the edge with the maximum weight encountered. Let this edge be denoted as maxEdge. Remove the edge maxEdge from the MST T' to obtain the modified MST for the graph G after deleting the original edge.

The running time of this modified algorithm is as follows:

- The BFS step takes O (|V| + |E|) time because it visits each vertex and edge once.

- Traversing the parent array takes O(|V|) time.

- Removing the edge maxEdge takes O (1) time.

Therefore, the overall running time of this modified algorithm **is O (|V|+|E|)**.

**3. ﻿ Show how to solve the single source shortest path problem of the graph in the lecture slides on Dijkstra’s algorithm, slide 11 using Dijkstra’s algorithm, when the source node is g. Give the initial values of D(v) for each node v. Each time a node is pulled into the cloud, give the D values which have changed as a result.**

**A drawing of a triangle with lines and dots

Description automatically generated**

**A piece of paper with writing on it

Description automatically generated*Choosing ‘g’ as source node, the single source shortest path algorithm has been solved below:***

**A piece of paper with writing on it

Description automatically generated**

**5. Design an algorithm for the single source shortest path problem (SSSP) on directed acyclic graphs (DAGs) in O (m + n) time.**

function shortestPathDAG (graph, source):

// Perform topological sort

stack = empty stack

visited = array of boolean values initialized as false

dist = array of distances initialized as infinity

dist[source] = 0

for each vertex v in graph:

if visited[v] is false:

topologicalSortUtil (v, visited, stack)

// Process vertices in topological order

while stack is not empty:

u = stack.pop ()

// Update distances of adjacent vertices

for each adjacent vertex v of u:

if dist[v] > dist[u] + weight (u, v):

dist[v] = dist[u] + weight (u, v)

return dist

function topologicalSortUtil (vertex, visited, stack):

visited[vertex] = true

for each adjacent vertex v of vertex:

if visited[v] is false:

topologicalSortUtil (v, visited, stack)

stack.push(vertex)

To design an algorithm for the single source shortest path problem (SSSP) on directed acyclic graphs (DAGs) in O (m + n) time:

* We first perform a topological sort of the DAG. Starting from the source vertex, we can use depth-first search (DFS) to traverse the graph. We maintain a stack to store the vertices in the order of their finishing times. Then we initialize an array called "dist" to store the distances from the source vertex to all other vertices. Setting the distance of the source vertex to 0, then we initialize the distances of all other vertices as infinity.
* We then process the vertices in the topological order. While the stack is not empty, we pop a vertex from the stack. For each adjacent vertex of the popped vertex, we update its distance if it can be reached with a shorter path. We compare the current distance of the adjacent vertex with the sum of the distance to the popped vertex and the weight of the edge between them.
* We repeat this process for all vertices in the topological order until all shortest paths are calculated. When we are done, we can return the array "dist" containing the shortest distances from the source vertex to all other vertices.

By using the topological sort order, we ensure that we process each vertex after all its preceding vertices, allowing us to calculate the shortest paths efficiently. This algorithm has a time complexity of O (m + n), where m is the number of edges and n is the number of vertices in the DAG.