Kruskal’s algorithm To Find MST

1. Sort all edges in **ascending order** of weight.
2. Initialize an empty MST and a **Disjoint Set Union (DSU)** (also called Union-Find).
3. Loop through sorted edges:
   - If the current edge’s **nodes are in different sets**, **add it** to the MST and **merge their sets**.
   - Else, skip the edge (because it creates a cycle).
4. Repeat until MST contains **(V-1) edges**, where V = total vertices.

```
Function KRUSKAL(G):
    A = EMPTY_SET

    FOR EACH vertex v IN G.V DO
        CALL MAKE_SET(v)
    END FOR

    SORT G.E BY weight ASCENDING

    FOR EACH edge (u, v) IN G.E DO
        IF FIND_SET(u) != FIND_SET(v) THEN
            A = A U {(u, v)}
            CALL UNION(u, v)
        END IF
    END FOR

    RETURN A
END Function
```

Prim’s Algorithm Explanation

1. Start from any vertex (arbitrary).
2. Keep track of vertices included in MST.
3. At each step, select the smallest weight edge that connects a vertex in the MST to a vertex outside the MST.
4. Add this edge and the new vertex to the MST.
5. Repeat until all vertices are included.

```
Function PRIM(G):
    A = EMPTY_SET

    FOR EACH vertex v IN G.V DO
        key[v] = INFINITY
        visited[v] = FALSE
        parent[v] = NULL
    END FOR

    key[0] = 0

    FOR i FROM 1 TO |G.V| - 1 DO
        u = SELECT vertex WITH MINIMUM key[u] WHERE visited[u] = FALSE
        visited[u] = TRUE

        FOR EACH vertex v ADJACENT TO u DO
            IF visited[v] = FALSE AND weight(u, v) < key[v] THEN
                parent[v] = u
                key[v] = weight(u, v)
            END IF
        END FOR
    END FOR
    FOR EACH vertex v FROM 1 TO |G.V| - 1 DO
        A = A U {(parent[v], v)}
    END FOR

    RETURN A
END Function
```

Single-Source Shortest Paths

INITIALIZE−SINGLE−SOURCE

```
FOR EACH vertex v IN G.V DO
    d[v] = INFINITY
    parent[v] = NULL
END FOR

d[s] = 0
```

```
Function RELAX(u, v, w, d, parent):
    IF d[v] > d[u] + w(u, v) THEN
        d[v] = d[u] + w(u, v)
        parent[v] = u
    END IF
END Function
```

Dijkstra Algorithm

```
Function DIJKSTRA(G, src):
    DECLARE dist[]
    DECLARE visited[]

    FOR EACH vertex v IN G.V DO
        dist[v] = INFINITY
        visited[v] = FALSE
    END FOR

    dist[src] = 0

    FOR count FROM 1 TO |G.V| - 1 DO
        u = SELECT vertex WITH MINIMUM dist[u] WHERE visited[u] = FALSE
        IF u = NULL THEN
            BREAK
        END IF

        visited[u] = TRUE
        FOR EACH vertex v IN G.V DO
            IF G.edge(u, v) EXISTS AND visited[v] = FALSE AND dist[u] + weight(u, v) < dist[v] THEN
                dist[v] = dist[u] + weight(u, v)
            END IF
        END FOR
    END FOR

    RETURN dist
END Function
```

Dijkstra’s Algorithm Complexity Note

Aspect	Simple Array Implementation	Min-Priority Queue Implementation
Time Complexity	$O(V^2)$	$O((V+E)\log V)$
Explanation	- Selecting min vertex: $O(V)$ each iteration \n- V iterations total	- Extract-min: $O(\log V)$ per vertex \n- Relax edges: $O(E \log V)$ total
Space Complexity	$O(V^2)$ (for adjacency matrix) \n$O(V)$ for arrays (dist, visited, parent)	$O(V+E)$ (for adjacency list and priority queue) \n$O(V)$ for dist and parent arrays

The Bellman-Ford algorithm

```
Function BELLMAN_FORD(G, w, s):
    FOR EACH vertex v IN G.V DO
        d[v] = INFINITY
        parent[v] = NULL
    END FOR

    d[s] = 0

    FOR i FROM 1 TO |G.V| - 1 DO
        FOR EACH edge (u, v) IN G.E DO
            IF d[u] + w(u, v) < d[v] THEN
                d[v] = d[u] + w(u, v)
                parent[v] = u
            END IF
        END FOR
    END FOR
```

```
    FOR EACH edge (u, v) IN G.E DO
        IF d[u] + w(u, v) < d[v] THEN
            RETURN "Negative-weight cycle detected"
        END IF
    END FOR

    RETURN d, parent
END Function
```

Explanation of Bellman-Ford Algorithm

1. **Initialization:**
   - Set the distance $d[v]$ for every vertex v in the graph to infinity.
   - Set the predecessor (or parent) of each vertex to NULL.
   - Set the distance $d[s]$ of the source vertex s to 0, because the distance from the source to itself is zero.
2. **Relaxation Loop:**
   - Repeat the following step $|V| - 1$ times (where $|V|$ is the number of vertices):
     - For every edge (u, v) in the graph:
       - Check if the current known distance to v can be improved by going through u.
       - If $d[u] + w(u, v) < d[v]$, update:
         - $d[v] = d[u] + w(u, v)$
         - $parent[v] = u$

   This step ensures that the shortest path to each vertex is gradually improved, considering paths of increasing length.
3. **Negative Cycle Detection:**
   - After the relaxation loops, check once more every edge (u, v):
     - If any edge can still be relaxed (i.e., $d[u] + w(u, v) < d[v]$), it means there is a **negative-weight cycle** reachable from the source.
     - In this case, the algorithm reports the presence of a negative-weight cycle (returns an error or special value).
4. **Result:**
   - If no negative-weight cycle is detected, the algorithm returns:
     - The distance array d, containing the shortest distances from s to every vertex.
     - The parent array, which can be used to reconstruct the shortest paths.