COMPLEXITY OF US13

kruskal algorithm

```
procedure Kruskal(sortedGraphEdges: array of Edge, verticesGraph: array of
Vertex) returns Graph
    A := new Graph()
    parent := new array[verticesGraph.size()]
    rank := new array[verticesGraph.size()]

    for i := 0 to verticesGraph.size() - 1 do
        parent[i] := i
        rank[i] := 0
    end for

    nA := 0
    i := 0

    while nA < verticesGraph.size() - 1 and i < sortedGraphEdges.size() do
        e := sortedGraphEdges[i]
        u := indexOf(verticesGraph, e.getOrigin())
        v := indexOf(verticesGraph, e.getDestiny())
        rootU := find(u, parent)
        rootV := find(v, parent)

        if rootU != rootV then
            A.addEdge(e)
            union(rootU, rootV, parent, rank)
            nA := nA + 1
        end if

        i := i + 1
    end while

    return A
```

find method

```
procedure find(vertex: integer, parent: array of integer) returns integer
    if parent[vertex] != vertex then
        parent[vertex] := find(parent[vertex], parent)
    end if
    return parent[vertex]
```

addEdge **method**

union **method**

```
procedure union(rootU: integer, rootV: integer, parent: array of integer,
rank: array of integer)
    if rank[rootU] > rank[rootV] then
        parent[rootV] := rootU
    else if rank[rootU] < rank[rootV] then
        parent[rootU] := rootV
    else
        parent[rootV] := rootU
        rank[rootU] := rank[rootU] + 1
    end if
```

Lines	**Kruskal**
1ª	1A
2ª	1A
3ª	1A
4ª	$(n + 1)$ A
5ª	n * 1A
6ª	n * 1A
7ª	1A
8ª	1A
9ª	m * 2C
10ª	m * 1A
11ª	m * O (n)
12ª	m * O (n)
13ª	m * T(find) (the complexity of this algorithm is analysed below)
14ª	m * T(find) (the complexity of this algorithm is analysed below)
15ª	m * 1C
16ª	m * 1A
17ª	m * T(union) (the complexity of this algorithm is analysed below)
18ª	m * (1A + 1Op)
19ª	m * (1A m + 1Op)
20ª	1R
Total	$9 + 2n + 3m$
O estimate	O $(m \log m)$

Subtitle:

- **A**: Assignment

- $\boldsymbol{n}$**:** Number of Vertices
- $\boldsymbol{m}$**:** Number of Edges
- **C**: Comparison
- **Op**: Arithmetic Operation
- **I**: Increment
- **R**: Return

COMPLEXITY OF US17

dijkstra algorithm

```
procedure Dijkstra(start: String, MP: String, edges: array of Edges)
    shortestPaths := new Graph()
    if MP is  equal to start then
        print("The vertex is already te Metting Point!")
    else
        initialGraph := new Graph()
        for each Edge in the set of Edges do
            Add Edge to initialGraph
        end for
        vertices := get vertices from inicialGraph
        numVertices :=  vertices.size()
        dist :=  new array[numVertices]
        prev := new array[numVertices]
        visited := new array[numVertices]
        for each index i from 0 to numVertices - 1 do
            dist[i] := -1
            prev[i] := null
            visited[i] := false
            create a PriorityQueue named Queue with a comparator based on
the distance of vertices
             MPVertex := new Vertex(MP)
             startVertex := new Vertex(start)
             dist[vertices.indexOf(startVertex)] := 0
            queue.add(startVertex)
            while (not queue.isEmpty())
                u := queue.poll()
                uIndex := vertices.indexOf(u)
                 visited[uIndex] := true
```

```
            neighbors := initialGraph.getVerticesConnectedTo(u)
            for each neighbor in neighbors
                  vIndex := vertices.indexOf(neighbor)
                 weight := initialGraph.getEdgeCost(u, neighbor)


                 if (not visited[vIndex] and dist[uIndex] ≠ -1 and
(dist[vIndex] = -1 or dist[uIndex] + weight < dist[vIndex])) then
                      queue.remove(neighbor)
                     dist[vIndex] := dist[uIndex] + weight
                     prev[vIndex] := u
                     queue.add(neighbor)
                end if
            end for
        end while
        path := new ArrayList()
        at := MPVertex
        while (at ≠ null)
              path.add(at)
              at := prev[vertices.indexOf(at)]
        end while
        path.reverse()
        i := 0
        while (i < path.size() - 1)
              origin := path.get(i)
              destiny := path.get(i + 1)
              cost := initialGraph.getEdgeCost(origin, destiny)
              shortestPaths.addEdge(new Edge(origin, destiny, cost))
              i := i + 1
        end while
return shortestPaths
```

Lines	Dijkstra
1ª	1A
2ª	1C
3ª	1A
4ª	1A
5ª	1A
6ª	m * 1A
7ª	m * 1A
8ª	n * 1A
9ª	1A
10ª	n * 1A
11ª	n * 1A
12ª	n * 1A
13ª	n * 1C
14ª	n * 1A
15ª	n * 1A
16ª	n * 1A
17ª	1A
18ª	1A
19ª	1A
20ª	1A
21ª	1A+log(n)
22ª	n ∗ log(n)
23ª	n ∗ log(n)
24ª	n ∗ 1A
25ª	n ∗ 1A
26ª	n ∗ 1A
27ª	m ∗ 1C
28ª	m * 1A
29ª	m * 1A
30ª	m ∗ 3C
31ª	m ∗ log(n)
32ª	m ∗ 1A
33ª	m ∗ 1A
34ª	m ∗ log(n)
35ª	m ∗ 1A
36ª	m ∗ 1A
37ª	n * log(n)
38ª	1A
39ª	1A
40ª	n * 1C
41ª	n * 1A
42ª	n * 1A
43ª	n * 1A

44ª	n * 1A
45ª	1A
46ª	n * 1C
47ª	n * 1A
48ª	n * 1A
49ª	n * 1A
50ª	n * 1A
51ª	n * 1A
52ª	1A
53ª	1A
54ª	1R
Total	9+2n+3m+2mlog(n)+6n+5m+5n+1
O estimate	O (m log(n))

Subtitle:

- **A**: Assignment
- **n:** Number of Vertices
- **m:** Number of Edges
- **C**: Comparison
- **R:** Return

COMPLEXITY OF US18

```
            neighbors := initialGraph.getVerticesConnectedTo(u)

            for each neighbor in neighbors do

                vIndex := vertices.indexOf(neighbor)

                weight := initialGraph.getEdgeCost(u, neighbor)

                if not visited[vIndex] and dist[uIndex] != -1 and
(dist[vIndex] = -1 or dist[uIndex] + weight < dist[vIndex]) then

                    queue.remove(neighbor)

                    dist[vIndex] := dist[uIndex] + weight

                    prev[vIndex] := u

                    queue.add(neighbor)

                end if

            end for

        end while

        path := new ArrayList<>()

        for Vertex at := MPVertex; at != null; at :=
prev[vertices.indexOf(at)] do

            path.add(at)

        end for

        path.reverse()

        shortestPath := new Graph()

        for each index i from 0 to path.size() - 2 do

            Vertex origin := path.get(i)

            Vertex destiny := path.get(i + 1)

            int cost := initialGraph.getEdgeCost(origin, destiny)

            shortestPath.addEdge(new Edge(origin, destiny, cost))

        end for

        shortestPaths.add(shortestPath)

    end if

end for

return shortestPaths
```

Lines	Dijkstra
1ª	1A
2ª	k * 1C
3ª	k * 1C
4ª	k * 1A
5ª	k * 1A
6ª	k * 1A
7ª	k * m * 1A
8ª	k * m * 1A
9ª	k * n * 1A
10ª	k * 1A
11ª	k * n * 1A
12ª	k * n * 1A
13ª	k * n * 1A
14ª	k * n * 1C
15ª	k * n * 1A
16ª	k * n * 1A
17ª	k * n * 1A
18ª	k * 1A
19ª	k * 1A
20ª	k * 1A
21ª	k * 1A
22ª	k * (1A + log(n))
23ª	k * n ∗ log(n)
24ª	k * n ∗ log(n)
25ª	k* n ∗ 1A
26ª	k* n ∗ 1A
27ª	k* n ∗ 1A
28ª	k* m ∗ 1C
29ª	k* m ∗ 1A
30ª	k* m ∗ 1A
31ª	k* n ∗ 3C
32ª	k* m ∗ log(n)
33ª	k * m ∗ 1A
34ª	k * m ∗ 1A
35ª	k* m ∗ log(n)
36ª	k * m ∗ 1A
37ª	k * m ∗ 1A
38ª	k * n * log(n)
39ª	k * 1A
40ª	k * 1A

41ª	k * n * 1C
42ª	k * n * 1A
43ª	k * n * 1A
44ª	k * n * 1A
45ª	k * n * 1A
46ª	k * 1A
47ª	k * n * 1C
48ª	k * n * 1A
49ª	k * n * 1A
50ª	k * n * 1A
51ª	k * n * 1A
52ª	k * 1A
53ª	k * n * 1A
54ª	k * 1A
55ª	k * 1A
56ª	1R
Total	k * (18 + 4m + 11n + (n*m)) * log(n)
O estimate	O (k * n2 * log(n))

Subtitle:

- **A**: Assignment
- **k:** Meeting points.
- **n:** Number of Vertices
- **m:** Number of Edges
- **C**: Comparison
- **R**: Return