Task 1: Dijkstra's Shortest Path Finder
Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.
Ans:

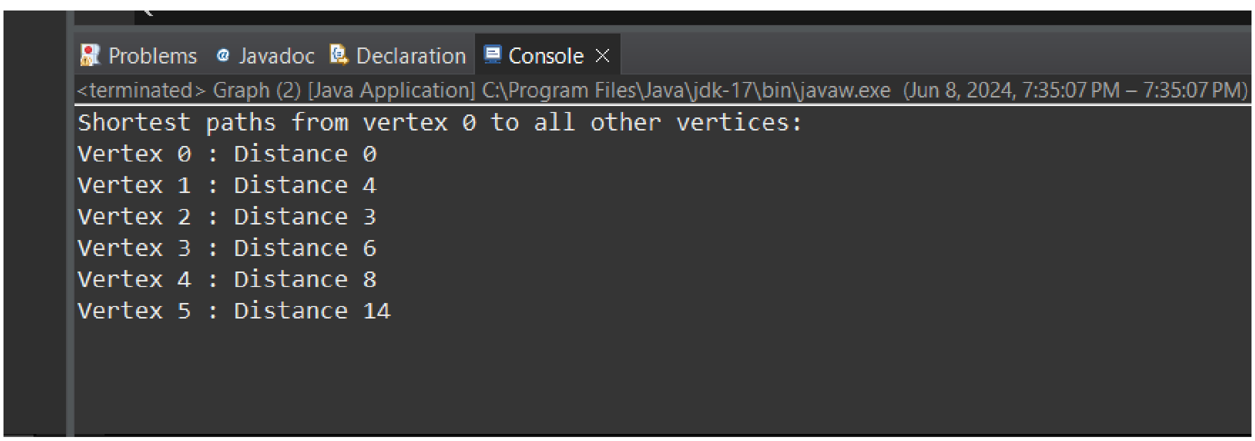
```
package Assignment9_10;


import java.util.*;


public class Graph {
    private int vertices;
    private List<List<Node>> adjList;


    static class Node implements Comparable<Node> {
        int vertex;
        int weight;


        Node(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }


        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.weight, other.weight);
        }
    }


    public Graph(int vertices) {
        this.vertices = vertices;
        adjList = new ArrayList<>(vertices);
```

```
    for (int i = 0; i < vertices; i++) {
        adjList.add(new LinkedList<>());
    }
}

public void addEdge(int source, int destination, int weight) {
    adjList.get(source).add(new Node(destination, weight));
    adjList.get(destination).add(new Node(source, weight)); // For undirected graph
}

public void dijkstra(int startVertex) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    int[] distances = new int[vertices];
    boolean[] visited = new boolean[vertices];

    Arrays.fill(distances, Integer.MAX_VALUE);
    distances[startVertex] = 0;
    pq.add(new Node(startVertex, 0));

    while (!pq.isEmpty()) {
        Node node = pq.poll();
        int currentVertex = node.vertex;

        if (visited[currentVertex]) {
            continue;
        }

        visited[currentVertex] = true;
```

```java
        List<Node> neighbors = adjList.get(currentVertex);
        for (Node neighbor : neighbors) {
            int neighborVertex = neighbor.vertex;
            int edgeWeight = neighbor.weight;

            if (!visited[neighborVertex]) {
                int newDist = distances[currentVertex] + edgeWeight;
                if (newDist < distances[neighborVertex]) {
                    distances[neighborVertex] = newDist;
                    pq.add(new Node(neighborVertex, newDist));
                }
            }
        }
    }

    printShortestPaths(distances, startVertex);
}

private void printShortestPaths(int[] distances, int startVertex) {
    System.out.println("Shortest paths from vertex " + startVertex + " to all other vertices:");
    for (int i = 0; i < distances.length; i++) {
        System.out.println("Vertex " + i + " : Distance " + distances[i]);
    }
}



public static void main(String[] args) {
    int vertices = 6;
```

```
        Graph graph = new Graph(vertices);

        graph.addEdge(0, 1, 4);
        graph.addEdge(0, 2, 3);
        graph.addEdge(1, 2, 1);
        graph.addEdge(1, 3, 2);
        graph.addEdge(2, 3, 4);
        graph.addEdge(3, 4, 2);
        graph.addEdge(4, 5, 6);

        graph.dijkstra(0);
    }
}
```

Output:




Task 2: Kruskal’s Algorithm for MST
Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

Ans:

```
package Assignment9_10;
```

```
import java.util.*;

public class Kruskal {
    class Edge implements Comparable<Edge> {
        int src, dest, weight;

        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight;
        }
    };

    class Subset {
        int parent, rank;
    };

    private int V, E;
    private Edge[] edges;

    public Kruskal(int v, int e) {
        V = v;
        E = e;
        edges = new Edge[E];
        for (int i = 0; i < e; ++i) {
            edges[i] = new Edge();
        }
    }

    private int find(Subset[] subsets, int i) {
```

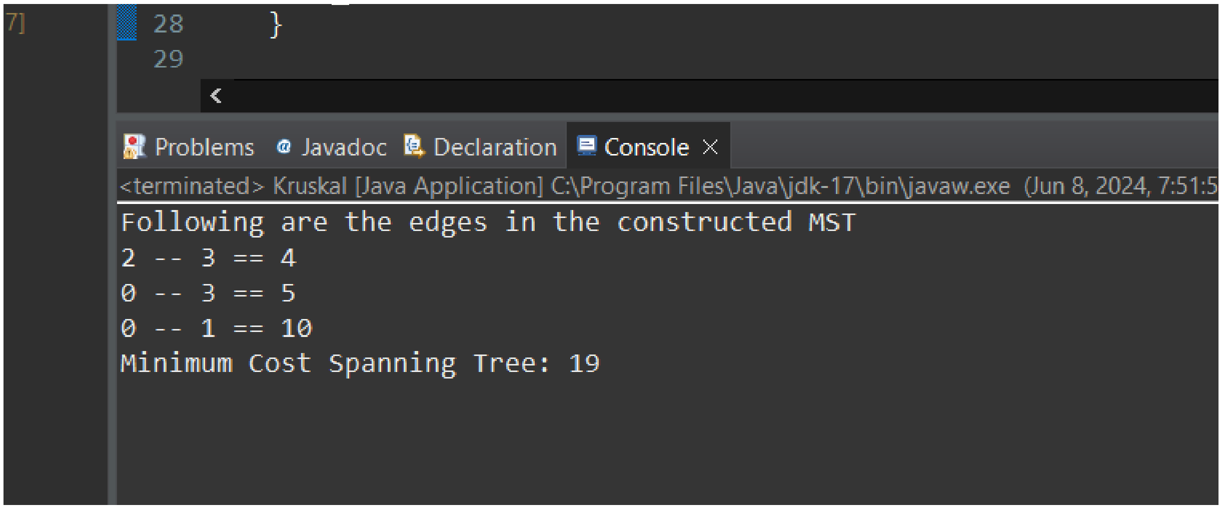
```
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

private void union(Subset[] subsets, int x, int y) {
    int rootX = find(subsets, x);
    int rootY = find(subsets, y);

    if (subsets[rootX].rank < subsets[rootY].rank) {
        subsets[rootX].parent = rootY;
    } else if (subsets[rootX].rank > subsets[rootY].rank) {
        subsets[rootY].parent = rootX;
    } else {
        subsets[rootY].parent = rootX;
        subsets[rootX].rank++;
    }
}

public void kruskalMST() {
    Edge[] result = new Edge[V];
    int e = 0;
    int i = 0;
    for (i = 0; i < V; ++i)
        result[i] = new Edge();

    Arrays.sort(edges);

    Subset[] subsets = new Subset[V];
```

```java
    for (i = 0; i < V; ++i)
        subsets[i] = new Subset();

    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    i = 0;
    while (e < V - 1) {
        Edge nextEdge = edges[i++];
        int x = find(subsets, nextEdge.src);
        int y = find(subsets, nextEdge.dest);

        if (x != y) {
            result[e++] = nextEdge;
            union(subsets, x, y);
        }
    }

    printMST(result, e);
}

private void printMST(Edge[] result, int e) {
    System.out.println("Following are the edges in the constructed MST");
    int minimumCost = 0;
    for (int i = 0; i < e; ++i) {
        System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);
        minimumCost += result[i].weight;
```

```java
    }
    System.out.println("Minimum Cost Spanning Tree: " + minimumCost);
  }

  public static void main(String[] args) {
    int V = 4;
    int E = 5;
    Kruskal graph = new Kruskal(V, E);

    // add edge 0-1
    graph.edges[0].src = 0;
    graph.edges[0].dest = 1;
    graph.edges[0].weight = 10;

    graph.edges[1].src = 0;
    graph.edges[1].dest = 2;
    graph.edges[1].weight = 6;

    graph.edges[2].src = 0;
    graph.edges[2].dest = 3;
    graph.edges[2].weight = 5;

    graph.edges[3].src = 1;
    graph.edges[3].dest = 3;
    graph.edges[3].weight = 15;


    graph.edges[4].src = 2;
    graph.edges[4].dest = 3;
```

```
        graph.edges[4].weight = 4;


        graph.kruskalMST();
    }
}
```

Output:




Task 3: Union-Find for Cycle Detection
Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

Ans:

```
package Assignment9_10;


import java.util.*;


public class UnionFind {
    private int[] parent;
    private int[] rank;


    public UnionFind(int size) {
```

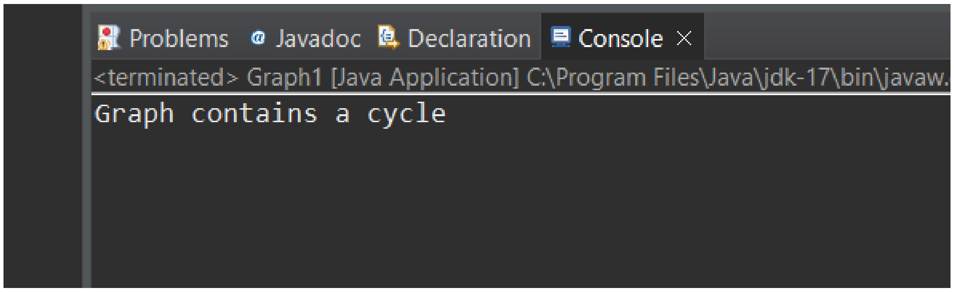
```
    parent = new int[size];
    rank = new int[size];
    for (int i = 0; i < size; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

public int find(int node) {
    if (parent[node] != node) {
        parent[node] = find(parent[node]);
    }
    return parent[node];
}

public void union(int node1, int node2) {
    int root1 = find(node1);
    int root2 = find(node2);

    if (root1 != root2) {
        if (rank[root1] < rank[root2]) {
            parent[root1] = root2;
        } else if (rank[root1] > rank[root2]) {
            parent[root2] = root1;
        } else {
            parent[root2] = root1;
            rank[root1]++;
        }
    }
```

```
    }
}

package Assignment9_10;

import java.util.*;

public class Graph1 {
    private int V;
    private List<Edge> edges;

    class Edge {
        int src, dest;

        Edge(int src, int dest) {
            this.src = src;
            this.dest = dest;
        }
    }

    public Graph1(int V) {
        this.V = V;
        edges = new ArrayList<>();
    }

    public void addEdge(int src, int dest) {
        edges.add(new Edge(src, dest));
    }
```

```java
public boolean hasCycle() {
    UnionFind uf = new UnionFind(V);

    for (Edge edge : edges) {
        int root1 = uf.find(edge.src);
        int root2 = uf.find(edge.dest);

        if (root1 == root2) {
            return true;
        }

        uf.union(root1, root2);
    }

    return false;
}

public static void main(String[] args) {
    Graph1 graph = new Graph1(3);

    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(0, 2);

    if (graph.hasCycle()) {
        System.out.println("Graph contains a cycle");
    } else {
        System.out.println("Graph does not contain a cycle");
    }
```

```
    }
}
```

Output:


Problems
Javadoc
Declaration
Console
<terminated> Graph1 [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.
Graph contains a cycle