RPS DAY 9- 10 Assignments

Assignment 3

Name: Akshada Baad
Batch - CPPE

Day 9 and 10:

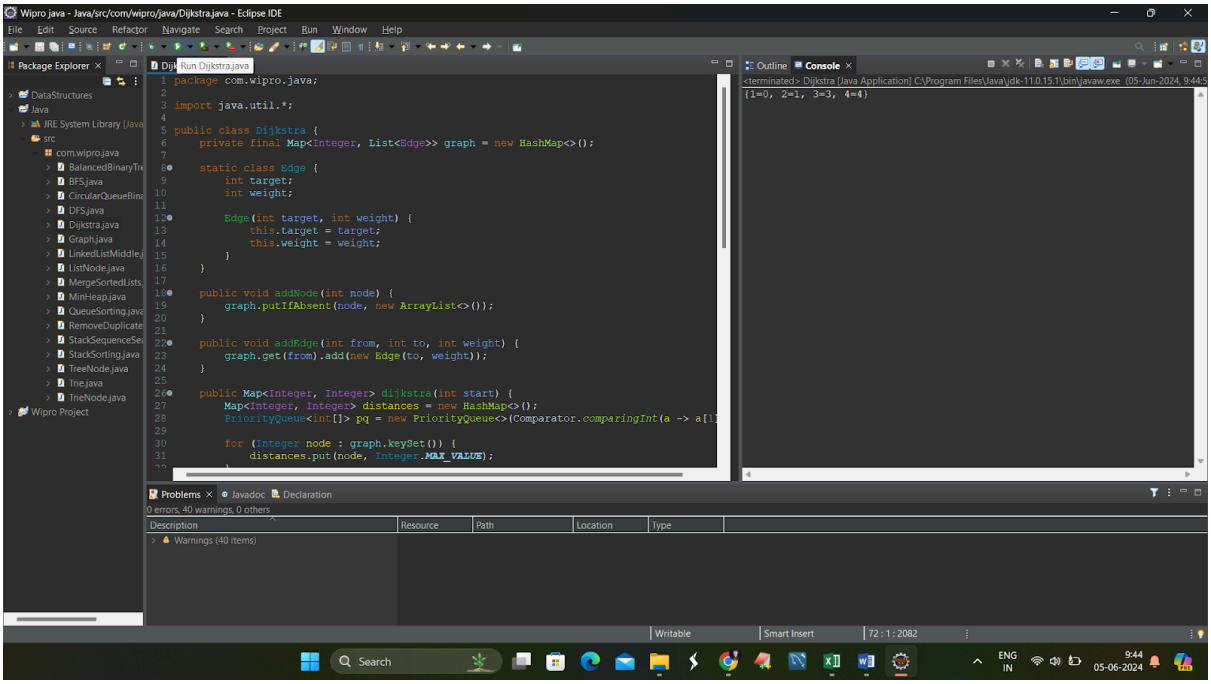
Task 1: Dijkstra's Shortest Path Finder

Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

```
import java.util.*;


public class Dijkstra {
    private final Map<Integer, List<Edge>> graph = new HashMap<>();

    static class Edge {
        int target;
        int weight;

        Edge(int target, int weight) {
            this.target = target;
            this.weight = weight;
        }
    }

    public void addNode(int node) {
        graph.putIfAbsent(node, new ArrayList<>());
    }

    public void addEdge(int from, int to, int weight) {
```

```
    graph.get(from).add(new Edge(to, weight));
}

public Map<Integer, Integer> dijkstra(int start) {
    Map<Integer, Integer> distances = new HashMap<>();
    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));

    for (Integer node : graph.keySet()) {
        distances.put(node, Integer.MAX_VALUE);
    }
    distances.put(start, 0);
    pq.add(new int[]{start, 0});

    while (!pq.isEmpty()) {
        int[] current = pq.poll();
        int currentNode = current[0];
        int currentDistance = current[1];

        if (currentDistance > distances.get(currentNode)) {
            continue;
        }

        for (Edge edge : graph.get(currentNode)) {
            int newDist = currentDistance + edge.weight;
            if (newDist < distances.get(edge.target)) {
                distances.put(edge.target, newDist);
                pq.add(new int[]{edge.target, newDist});
            }
        }
    }
```

```
        return distances;
    }

    public static void main(String[] args) {
        Dijkstra graph = new Dijkstra();
        graph.addNode(1);
        graph.addNode(2);
        graph.addNode(3);
        graph.addNode(4);

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

        System.out.println(graph.dijkstra(1)); // Output: {1=0, 2=1, 3=3, 4=4}
    }
}
```



Task 2: Kruskal's Algorithm for MST

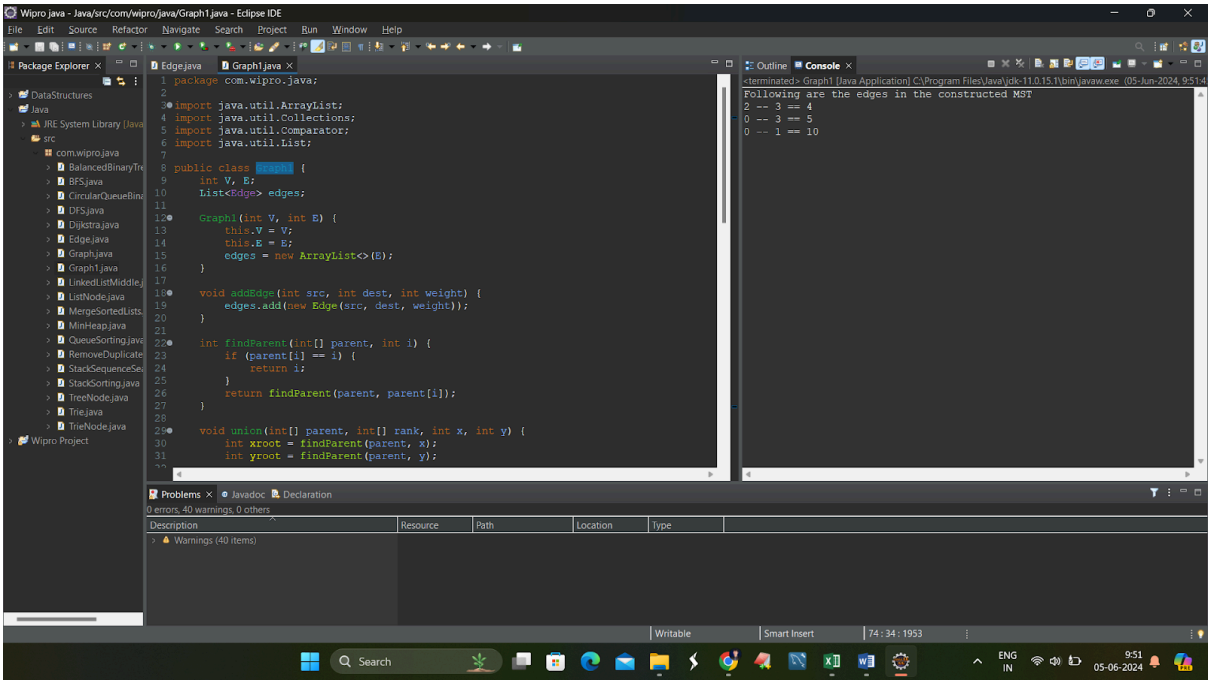
Implement Kruskal's algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

```
    import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;


class Edge {
   int src, dest, weight;
```

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

class Graph {
    int V, E;
    List<Edge> edges;

    Graph(int V, int E) {
        this.V = V;
        this.E = E;
        edges = new ArrayList<>(E);
    }

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

    int findParent(int[] parent, int i) {
        if (parent[i] == i) {
            return i;
        }
        return findParent(parent, parent[i]);
    }

    void union(int[] parent, int[] rank, int x, int y) {
        int xroot = findParent(parent, x);
```

```
    int yroot = findParent(parent, y);

    if (rank[xroot] < rank[yroot]) {
        parent[xroot] = yroot;
    } else if (rank[xroot] > rank[yroot]) {
        parent[yroot] = xroot;
    } else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}

void KruskalMST() {
    List<Edge> result = new ArrayList<>();
    int[] parent = new int[V];
    int[] rank = new int[V];

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

    Collections.sort(edges, Comparator.comparingInt(o -> o.weight));

    for (Edge edge : edges) {
        int x = findParent(parent, edge.src);
        int y = findParent(parent, edge.dest);

        if (x != y) {
            result.add(edge);
```

```
                union(parent, rank, x, y);
            }
        }

        System.out.println("Following are the edges in the constructed MST");
        for (Edge edge : result) {
            System.out.println(edge.src + " -- " + edge.dest + " == " + edge.weight);
        }
    }

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

        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(1, 3, 15);
        graph.addEdge(2, 3, 4);

        graph.KruskalMST();
    }
}
```



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.

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

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

```
    }

    int find(int i) {
        if (parent[i] != i) {
            parent[i] = find(parent[i]); // Path compression
        }
        return parent[i];
    }

    void union(int x, int y) {
        int xroot = find(x);
        int yroot = find(y);

        if (rank[xroot] < rank[yroot]) {
            parent[xroot] = yroot;
        } else if (rank[xroot] > rank[yroot]) {
            parent[yroot] = xroot;
        } else {
            parent[yroot] = xroot;
            rank[xroot]++;
        }
    }
}

class GraphCycle {
    int V, E;
    Edge[] edges;

    class Edge {
        int src, dest;
```

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

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

boolean isCycle() {
    UnionFind unionFind = new UnionFind(V);

    for (int i = 0; i < E; ++i) {
        int x = unionFind.find(edges[i].src);
        int y = unionFind.find(edges[i].dest);

        if (x == y) {
            return true;
        }
        unionFind.union(x, y);
    }
    return false;
}
```

```
    public static void main(String[] args) {
        int V = 3;
        int E = 3;
        GraphCycle graph = new GraphCycle(V, E);

        graph.edges[0] = graph.new Edge(0, 1);
        graph.edges[1] = graph.new Edge(1, 2);
        graph.edges[2] = graph.new Edge(0, 2);

        if (graph.isCycle()) {
            System.out.println("Graph contains cycle");
        } else {
            System.out.println("Graph doesn't contain cycle");
        }
    }
}
```