Data 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.

```
package com.wipro.graph;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;

public class Dijkstra {
   private HashMap<String, HashMap<String,Integer>> adjList = new HashMap<>();

   private HashMap<String, String> previous = new HashMap<>();

   public static void main(String[] args) {
      Dijsktra myGraph = new Dijkstra ();
      myGraph.addVertex("A");
      myGraph.addVertex("B");
      myGraph.addVertex("C");
      myGraph.addVertex("D");
      myGraph.addVertex("E");
      myGraph.addVertex("F");

      myGraph.addEdge("A", "B", 2);
      myGraph.addEdge("A", "D", 8);
      myGraph.addEdge("B", "E", 6);
      myGraph.addEdge("B", "D", 5);
      myGraph.addEdge("E", "D", 3);
      myGraph.addEdge("E", "F", 1);
      myGraph.addEdge("E", "C", 9);
      myGraph.addEdge("D", "F", 2);
      myGraph.addEdge("F", "C", 3);

      myGraph.printGraph();

      myGraph.dijkstra("A");

      ArrayList<String> shortestPathToC = myGraph.getShortestPathTo("C");
      System.out.println("Shortest path from A to C: " + shortestPathToC);
   }

   private void addVertex(String vertex) {
         adjList.putIfAbsent(vertex, new HashMap<>());
```

```
}

public boolean addEdge(String vertex1, String vertex2, int weight) {

    if (adjList.containsKey(vertex1) && adjList.containsKey(vertex2)) {

        adjList.get(vertex1).put(vertex2, weight);
        adjList.get(vertex2).put(vertex1, weight);
        return true;
    }
    return false;
}

private void printGraph() {
    System.out.println("Graph:");
    for (Map.Entry<String, HashMap<String, Integer>> entry : adjList.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
    }

}

private void dijkstra(String start) {
    HashMap<String, Integer> distance = new HashMap<>();
    PriorityQueue<VertexDistancePair> pq = new PriorityQueue<>();

    for (String vertex : adjList.keySet()) {
        distance.put(vertex, Integer.MAX_VALUE);
        previous.put(vertex, null);
    }


     distance.put(start, 0);
    pq.offer(new VertexDistancePair(start, 0));

    while (!pq.isEmpty()) {
        VertexDistancePair currentPair = pq.poll();
        String current = currentPair.vertex;

        for (Map.Entry<String, Integer> neighbourEntry : adjList.get(current).entrySet()) {
            String neighbour = neighbourEntry.getKey();
            int weight = neighborEntry.getValue();
            int newDistance = distance.get(current) + weight;

            if (newDistance < distance.get(neighbour)) {
                distance.put(neighbour, newDistance);
                    previous.put(neighbour, current);
                pq.offer(new VertexDistancePair(neighbour, newDistance));
            }
```

```
            }
        }
    }

    private ArrayList<String> getShortestPathTo(String destination) {
        ArrayList<String> path = new ArrayList<>();
        String current = destination;
        while (current != null) {
            path.add(0, current);
            current = previous.get(current);
        }
        return path;
    }

    private static class VertexDistancePair implements Comparable<VertexDistancePair> {
        String vertex;
        int distance;

        VertexDistancePair(String vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }

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

}
```

Output:

```
<terminated> Dijsktra [Java Application] C:\Users\Asus\.p2\pool\plugi
Graph:
A -> {B=2, D=8}
B -> {A=2, D=5, E=6}
C -> {E=9, F=3}
D -> {A=8, B=5, E=3, F=2}
E -> {B=6, C=9, D=3, F=1}
F -> {C=3, D=2, E=1}
Shortest path from A to C: [A, B, D, F, C]
```

Task 2: Kruskal's Algorithm for MST

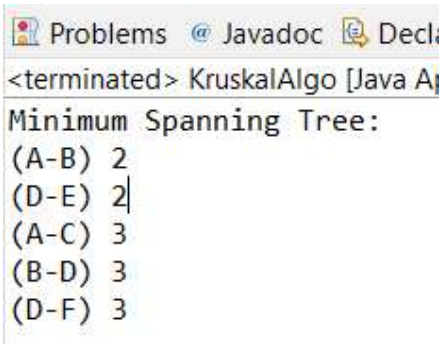
Implement Kruskal's algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

```
package com.wipro.graph;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
class Edge implements Comparable<Edge> {
	String source;
	String destination;
	int weight;
	Edge(String source, String destination, int weight) {
		this.source = source;
		this.destination = destination;
		this.weight = weight;
	}
	@Override
	public int compareTo(Edge other) {
		return this.weight - other.weight;
	}
}
public class KruskalAlgo {
	private final HashMap<String, HashMap<String, Integer>> adjacencyList = new
HashMap<>();
	public static void main(String[] args) {
		KruskalAlgo kruskal = new KruskalAlgo();
		kruskal.addVertex("A");
		kruskal.addVertex("B");
		kruskal.addVertex("C");
		kruskal.addVertex("D");
		kruskal.addVertex("E");
		kruskal.addVertex("F");
		kruskal.addEdge("A", "B", 2);
		kruskal.addEdge("B", "D", 3);
		kruskal.addEdge("A", "C", 3);
		kruskal.addEdge("C", "E", 4);
		kruskal.addEdge("B", "C", 5);
		kruskal.addEdge("D", "F", 3);
		kruskal.addEdge("E", "F", 5);
		kruskal.addEdge("D", "E", 2);
		kruskal.addEdge("B", "E", 4);
		kruskal.findMinimumSpanningTree();
	}
	private boolean addEdge(String vertex1, String vertex2, int weight) {
		if (adjacencyList.containsKey(vertex1) && adjacencyList.containsKey(vertex2)) {
			adjacencyList.get(vertex1).put(vertex2, weight);
			adjacencyList.get(vertex2).put(vertex1, weight);
			return true;
		}
		return false;
	}
	private void addVertex(String vertex) {
		if (!adjacencyList.containsKey(vertex)) {
```

```
                adjacencyList.put(vertex, new HashMap<>());
            }
        }
        private void findMinimumSpanningTree() {
            List<Edge> allEdges = getAllEdges();
            Collections.sort(allEdges);
            DisjointSet disjointSet = new DisjointSet();
            for (String vertex : adjacencyList.keySet()) {
                disjointSet.makeSet(vertex);
            }
            List<Edge> minimumSpanningTree = new ArrayList<>();
            for (Edge edge : allEdges) {
                String srcParent = disjointSet.find(edge.source);
                String destParent = disjointSet.find(edge.destination);
                if (!srcParent.equals(destParent)) {
                    minimumSpanningTree.add(edge);
                    disjointSet.union(edge.source, edge.destination);
                }
            }
            printMinimumSpanningTree(minimumSpanningTree);
        }
        private List<Edge> getAllEdges() {
            List<Edge> edges = new ArrayList<>();
            for (String vertex : adjacencyList.keySet()) {
                for (String neighbour : adjacencyList.get(vertex).keySet()) {
                    int weight = adjacencyList.get(vertex).get(neighbour);
                    edges.add(new Edge(vertex, neighbour, weight));
                }
            }
            return edges;
        }
        private void printMinimumSpanningTree(List<Edge> minimumSpanningTree) {
            System.out.println("Minimum Spanning Tree:");
            for (Edge edge : minimumSpanningTree) {
                System.out.println("(" + edge.source + "-" + edge.destination + ") " +
edge.weight);
            }
        }
        static class DisjointSet {
            private final HashMap<String, String> parent = new HashMap<>();
            void makeSet(String vertex) {
                parent.put(vertex, vertex);
            }
            String find(String vertex) {
                if (!parent.get(vertex).equals(vertex)) {
                    parent.put(vertex, find(parent.get(vertex)));
                }
                return parent.get(vertex);
            }
            void union(String vertex1, String vertex2) {
                parent.put(find(vertex1), find(vertex2));
            }
```

```
        }
}
```

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.

```
    import java.util.Arrays;
class UnionFind {
        int[] parent;
        int[] rank;
        UnionFind(int n) {
                parent = new int[n];
                rank = new int[n];
                Arrays.fill(rank, 1);
                for(int i=0; i<n ;i++) {
                        parent[i] =i;
                }

        }
        int find(int i) {
                if (parent[i] != i) {
                        parent[i] = find(parent[i]);
                }
                return parent[i];
        }
        void union(int x, int y) {
                int rootX = find(x);
                int rootY = find(y);
                if (rootX != rootY) {
                        if (rank[rootX] < rank[rootY]) { // 1<2
                                parent[rootX] = rootY;
                        } else if (rank[rootX] > rank[rootY]) {
                                parent[rootY] = rootX;
                        } else {
                                parent[rootY] = rootX;
```

```
                                        rank[rootX]++;
                                }
                        }
                }
        }
class Graph1 {
        int V, E;
        Edge[] edges;
        class Edge {
                int src, dest;
        }
        Graph1(int v, int e) {
                this.V = v;
                this.E = e;
                this.edges = new Edge[E];
                for (int i = 0; i < e; i++) {
                        edges[i] = new Edge();
                        System.out.println(edges[i].src + "  -- " + edges[i].dest);
                }
        }

        public boolean isCycleFound(Graph1 graph) {
                UnionFind uf = new UnionFind(V);
                for(int i=0; i< E ; ++i) {
                        int x = find(uf, graph.edges[i].src);
                        int y = find(uf, graph.edges[i].dest);

                        if(x==y) {
                                return true;
                        }
                        uf.union(x, y);
                }
                return false;
        }
        private int find(UnionFind uf, int i) {

                return uf.find(i);
        }

}
public class CycleDetect {
        public static void main(String[] args) {
                //int V = 3, E = 3;
                int V = 3, E = 2;
                Graph1 graph = new Graph1(V, E);
                graph.edges[0].src = 0;
                graph.edges[0].dest = 1;
                graph.edges[1].src = 1;
                graph.edges[1].dest = 2;
                //graph.edges[2].src = 0;
                //graph.edges[2].dest = 2;
                System.out.println(graph.V + " -- " + graph.E);
```

```
                for (int i = 0; i < E; i++) {
                        System.out.println(graph.edges[i].src + "  -- " + graph.edges[i].dest);
                }


                if(graph.isCycleFound(graph)) {
                        System.out.println("Cycle Found");
                }else {
                        System.out.println("Cycle Not Found...");
                }
        }
}
```

Output: