ASSIGNMENT:DAY_09_&_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.

ANS:

```
import java.util.*;


class Djikstras {

   private Map<Integer, List<Edge>> adjList;


   public Djikstras() {

      this.adjList = new HashMap<>();

   }


   public void addNode(int node) {

      adjList.putIfAbsent(node, new ArrayList<>());

   }


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

      adjList.putIfAbsent(from, new ArrayList<>());

      adjList.putIfAbsent(to, new ArrayList<>());

      adjList.get(from).add(new Edge(to, weight));

   }


   public Map<Integer, String> dijkstra(int start) {

      Map<Integer, String> distances = new HashMap<>();
```

```
    PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(edge -> edge.weight));

    for (int node : adjList.keySet()) {
        distances.put(node, Integer.MAX_VALUE+"");
    }
    distances.put(start, 0+"");
    pq.add(new Edge(start, 0));

    while (!pq.isEmpty()) {
        Edge edge = pq.poll();
        int currentNode = edge.to;
        int currentDistance = edge.weight;

        for (Edge neighbor : adjList.get(currentNode)) {
            int newDist = currentDistance + neighbor.weight;
            if (newDist < Integer.parseInt(distances.get(neighbor.to))) {
                distances.put(neighbor.to, newDist+"");
                pq.add(new Edge(neighbor.to, newDist));
            }
        }
    }

    return distances;
}
```

```
class Edge {
    int to;
    int weight;

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

public static void main(String[] args) {
    Djikstras graph = new Djikstras();
    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, 6);
    graph.addEdge(3, 4, 3);

    Map<Integer, String> distances = graph.dijkstra(4);
    for (Map.Entry<Integer,String> entry : distances.entrySet()) {
```

```
        System.out.println("Distance from 4 to " + entry.getKey() + " is " + (Integer.parseInt(entry.getValue())==Integer.MAX_VALUE? "Infinity":entry.getValue()));
      }
  }
}

//code-by-RUBY
```

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:

```
import java.util.*;

class Kruskal {
  private int vertices;
  private List<Edge> edges;

  public Kruskal(int vertices) {
    this.vertices = vertices;
    this.edges = new ArrayList<>();
  }

  public void addEdge(int from, int to, int weight) {
    edges.add(new Edge(from, to, weight));
```

```
}

public List<Edge> kruskal() {
    List<Edge> mst = new ArrayList<>();
    Collections.sort(edges, Comparator.comparingInt(edge -> edge.weight));
    UnionFind uf = new UnionFind(vertices);

    for (Edge edge : edges) {
        if (uf.find(edge.from) != uf.find(edge.to)) {
            mst.add(edge);
            uf.union(edge.from, edge.to);
        }
    }

    return mst;
}

class Edge {
    int from, to, weight;

    Edge(int from, int to, int weight) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }
```

```java
}

class UnionFind {
    private int[] parent, 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 x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
```

```java
            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else {
                    parent[rootY] = rootX;
                    rank[rootX]++;
                }
            }
        }
    }

    public static void main(String[] args) {
        Kruskal graph = new Kruskal(5);
        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);

        List<Edge> mst = graph.kruskal();
        for (Edge edge : mst) {
            System.out.println("Edge: " + edge.from + " - " + edge.to + " with weight: " + edge.weight);
        }
```

```
    }
}
```

//code_by_RUBY

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:

```
public class UnionFindCycleDetection {
    private int[] parent, rank;

    public UnionFindCycleDetection(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 x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
```

```java
    return parent[x];
}

public void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

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

public boolean detectCycle(int[][] edges) {
    for (int[] edge : edges) {
        int from = edge[0];
        int to = edge[1];

        if (find(from) == find(to)) {
```

```
            return true;
          }
          union(from, to);
        }
        return false;
    }

    public static void main(String[] args) {
        int vertices = 4;
        int[][] edges = {{0, 1}, {1, 2}, {2, 3}, {3, 0}};

        UnionFindCycleDetection uf = new UnionFindCycleDetection(vertices);
        boolean hasCycle = uf.detectCycle(edges);

        System.out.println("Graph contains cycle: " + (hasCycle==true? "YES":"NO"));
    }
}
//code_by_RUBY
```