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 Graph {
  private int V;
  private List<List<WeightedEdge>> adjList;

  public Graph(int V) {
    this.V = V;
    adjList = new ArrayList<>(V);
    for (int i = 0; i < V; i++) {
      adjList.add(new ArrayList<>());
    }
  }

  public void addEdge(int u, int v, int weight) {
    adjList.get(u).add(new WeightedEdge(v, weight));
  }

  public static class WeightedEdge {
    public final int dest;
    public final int weight;

    public WeightedEdge(int dest, int weight) {
      this.dest = dest;
      this.weight = weight;
    }
```

```
}
public int[] dijkstra(int source) {
  int[] distances = new int[V];
  Arrays.fill(distances, Integer.MAX_VALUE);
  distances[source] = 0; // Distance to source is 0

  PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(Node::getDistance));
  pq.add(new Node(source, 0));

  while (!pq.isEmpty()) {
    Node current = pq.poll();

    for (WeightedEdge neighbor : adjList.get(current.vertex)) {
      int alt = distances[current.vertex] + neighbor.weight;
      if (alt < distances[neighbor.dest]) {
        distances[neighbor.dest] = alt;
        pq.add(new Node(neighbor.dest, alt));
      }
    }
  }
  return distances;
}

public static class Node {
  public final int vertex;
  public final int distance;

  public Node(int vertex, int distance) {
    this.vertex = vertex;
    this.distance = distance;
```

```
    }

    public int getDistance() {
      return distance;
    }
  }
}
```

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.*;

public class Graph {
  private int V;
  private List<WeightedEdge> edges;

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

  public void addEdge(int u, int v, int weight) {
    edges.add(new WeightedEdge(u, v, weight));
  }

  public static class WeightedEdge {
    public final int src;
    public final int dest;
```

```
    public final int weight;

    public WeightedEdge(int src, int v, int weight) {
      this.src = src;
      this.dest = v;
      this.weight = weight;
    }
  }
  private int find(int[] parent, int i) {
    if (parent[i] == -1) {
      return i;
    }
    return find(parent, parent[i]);
  }
private void union(int[] parent, int x, int y) {
    int xSet = find(parent, x);
    int ySet = find(parent, y);
    parent[xSet] = ySet;
  }
  public List<WeightedEdge> kruskalMST() {
    List<WeightedEdge> mst = new ArrayList<>();

    Collections.sort(edges, Comparator.comparingInt(WeightedEdge::getWeight));

    int[] parent = new int[V];
    Arrays.fill(parent, -1);

    for (WeightedEdge edge : edges) {
      int x = find(parent, edge.src);
      int y = find(parent, edge.dest);
```

```
      if (x != y) {
        mst.add(edge);
        union(parent, x, y);
      }
    }

    return mst;
  }
}
```

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.

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

  public UnionFind(int n) {
    parent = new int[n];
    rank = new int[n];
    for (int i = 0; i < n; i++) {
      parent[i] = i;
      rank[i] = 0;
    }
  }
  private int find(int x) {
    if (parent[x] != x) {
      parent[x] = find(parent[x]);
    }
```

```
  return parent[x];
}

private void union(int x, int y) {
  int xRoot = find(x);
  int yRoot = find(y);

  if (xRoot == yRoot) return;

  if (rank[xRoot] < rank[yRoot]) {
    parent[xRoot] = yRoot;
  } else if (rank[xRoot] > rank[yRoot]) {
    parent[yRoot] = xRoot;
  } else {
    parent[yRoot] = xRoot;
    rank[xRoot]++;
  }
}
public boolean detectCycle(int[][] edges) {
  for (int[] edge : edges) {
    int x = edge[0];
    int y = edge[1];

    int xRoot = find(x);
    int yRoot = find(y);

    if (xRoot == yRoot) {
      return true;
    }
```

```
      union(xRoot, yRoot);
    }
    return false;
  }
}
```