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.

Dijkstra's algorithm is a famous algorithm used to find the shortest path from a starting node to all other nodes in a weighted graph with non-negative weights. Here's how we can implement Dijkstra's algorithm in Java using a priority queue (min-heap) for efficient retrieval of the next vertex with the smallest tentative distance.

Steps and Implementation:

1. **Graph Representation**: We'll use an adjacency list to represent the graph, where each vertex points to a list of its neighboring vertices along with the edge weights.
2. **Priority Queue**: To efficiently retrieve the vertex with the smallest tentative distance, we'll use a PriorityQueue from Java's standard library.
3. **Distance Array**: Maintain an array to store the shortest known distance from the start node to each node in the graph.
4. **Visited Array**: Track which nodes have already been processed.
5. **Algorithm**:
    - Initialize distances from the start node to all other nodes as infinity (or a very large number) except for the start node itself (distance to itself is 0).
    - Use a priority queue to select the node with the smallest distance (initially the start node).
    - For the selected node, update the distances to its neighboring nodes if a shorter path is found through the selected node.
    - Mark the selected node as processed and continue until all nodes have been processed or the priority queue is empty.

```
import java.util.*;

public class DijkstraAlgorithm {

    private static final int INF = Integer.MAX_VALUE; // Infinity value for unreachable nodes

    public static void dijkstra(List<List<Node>> graph, int start) {
        int V = graph.size(); // Number of vertices in the graph
```

```
        // Array to store the shortest distance from start to each vertex
        int[] dist = new int[V];
        Arrays.fill(dist, INF);
        dist[start] = 0; // Distance from start to itself is 0

        // Priority queue (min-heap) to select the vertex with the smallest distance
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node ->
node.distance));
        pq.offer(new Node(start, 0)); // Start with the start node

        // Array to track visited nodes
        boolean[] visited = new boolean[V];

        while (!pq.isEmpty()) {
            int u = pq.poll().vertex;

            // If u is already processed, continue to the next iteration
            if (visited[u]) continue;

            visited[u] = true;

            // Update distances to all neighboring vertices of u
            for (Node neighbor : graph.get(u)) {
                int v = neighbor.vertex;
                int weight = neighbor.distance;

                // If a shorter path to v is found through u, update dist[v]
                if (!visited[v] && dist[u] != INF && dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.offer(new Node(v, dist[v]));
                }
            }
        }

        // Print shortest distances from start to all vertices
        System.out.println("Shortest distances from node " + start + ":");
        for (int i = 0; i < V; i++) {
            if (dist[i] == INF) {
                System.out.println("Node " + i + ": unreachable");
```

```
            } else {
                System.out.println("Node " + i + ": " + dist[i]);
            }
        }
    }

    public static void main(String[] args) {
        int V = 5; // Number of vertices in the graph

        // Create adjacency list representation of the graph
        List<List<Node>> graph = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            graph.add(new ArrayList<>());
        }

        // Adding edges to the graph (undirected graph)
        graph.get(0).add(new Node(1, 9));
        graph.get(0).add(new Node(2, 6));
        graph.get(0).add(new Node(3, 5));
        graph.get(0).add(new Node(4, 3));
        graph.get(2).add(new Node(1, 2));
        graph.get(2).add(new Node(3, 4));

        // Perform Dijkstra's algorithm starting from node 0
        dijkstra(graph, 0);
    }

    // Helper class to represent a node with its distance from the source
    static class Node {
        int vertex;
        int distance;

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

Kruskal's algorithm is a popular algorithm used to find the Minimum Spanning Tree (MST) of a connected, undirected graph with non-negative edge weights. The MST is the subset of edges that connects all vertices together without any cycles and with the minimum possible total edge weight.

Steps and Implementation:

1. **Graph Representation**: Use an edge list to represent the graph. Each edge contains two vertices and a weight.
2. **Union-Find Data Structure**: Utilize a Union-Find (Disjoint Set Union, DSU) structure to efficiently manage and merge sets of vertices. This helps in detecting cycles during the construction of the MST.
3. **Kruskal's Algorithm**:
    - Sort all the edges in the graph in non-decreasing order of their weights.
    - Initialize an empty MST.
    - Iterate through the sorted edges and add each edge to the MST if it does not form a cycle with the edges already in the MST (checked using Union-Find operations).

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.

To implement a Union-Find data structure with path compression and use it to detect cycles in an undirected graph, we'll follow these steps:

Union-Find Data Structure with Path Compression

The Union-Find (Disjoint Set Union, DSU) data structure supports two primary operations efficiently:

- **Union**: Merge two sets.
- **Find**: Determine which set a particular element is in.

Path compression is a technique used in the Find operation to flatten the structure of the tree whenever find is called, ensuring that future operations are faster.

Steps to Implement Union-Find with Path Compression:

1. **Initialization**:

    - Maintain two arrays: parent[] to store the parent of each node and rank[] to keep track of the depth of trees.
2. **Find Operation with Path Compression**:
    - Traverse up the parent pointers until you reach the root of the set.
    - During this traversal, make each node point directly to the root (path compression).
3. **Union Operation**:
    - Merge two sets by attaching the root of one tree under the root of another tree.
    - Attach the tree with lower rank under the tree with higher rank to keep the tree flat.
4. **Cycle Detection**:
    - For each edge in the graph, check if both vertices belong to the same set (using find).
    - If they do, then including this edge would form a cycle.