Day 9 and 10:

Task 1: Dijkstra's Shortest Path Finder

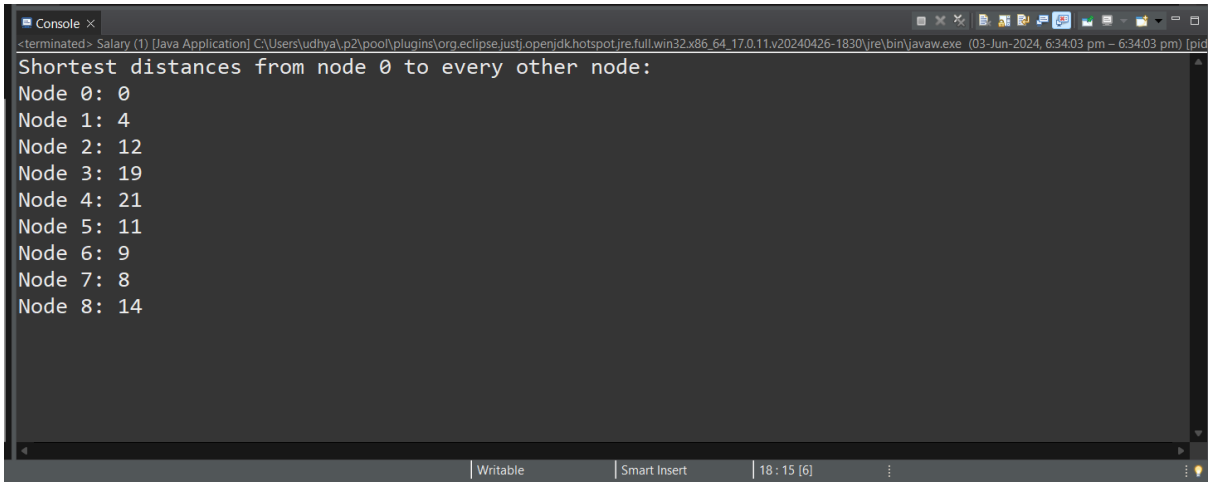
Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

```
import java.util.*;


public class DijkstraShortestPath {


   private static final int INF = Integer.MAX_VALUE;


   public static void dijkstra(int[][] graph, int start) {
      int V = graph.length;
      boolean[] visited = new boolean[V];
      int[] dist = new int[V];
      Arrays.fill(dist, INF);
      dist[start] = 0;


      for (int count = 0; count < V - 1; count++) {
         int u = minDistance(dist, visited);
         visited[u] = true;


         for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] != 0 && dist[u] != INF &&
                  dist[u] + graph[u][v] < dist[v]) {
               dist[v] = dist[u] + graph[u][v];
            }
         }
      }
```

```
    printSolution(dist, start);
}

private static int minDistance(int[] dist, boolean[] visited) {
    int min = INF;
    int minIndex = -1;

    for (int v = 0; v < dist.length; v++) {
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            minIndex = v;
        }
    }

    return minIndex;
}

private static void printSolution(int[] dist, int start) {
    System.out.println("Shortest distances from node " + start + " to every other node:");
    for (int i = 0; i < dist.length; i++) {
        System.out.println("Node " + i + ": " + dist[i]);
    }
}

public static void main(String[] args) {
    int[][] graph = {
            {0, 4, 0, 0, 0, 0, 0, 8, 0},
            {4, 0, 8, 0, 0, 0, 0, 11, 0},
```

```
            {0, 8, 0, 7, 0, 4, 0, 0, 2},
            {0, 0, 7, 0, 9, 14, 0, 0, 0},
            {0, 0, 0, 9, 0, 10, 0, 0, 0},
            {0, 0, 4, 14, 10, 0, 2, 0, 0},
            {0, 0, 0, 0, 0, 2, 0, 1, 6},
            {8, 11, 0, 0, 0, 0, 1, 0, 7},
            {0, 0, 2, 0, 0, 0, 6, 7, 0}
        };

        int startNode = 0;
        dijkstra(graph, startNode);
    }
}
```

OUTPUT:



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.

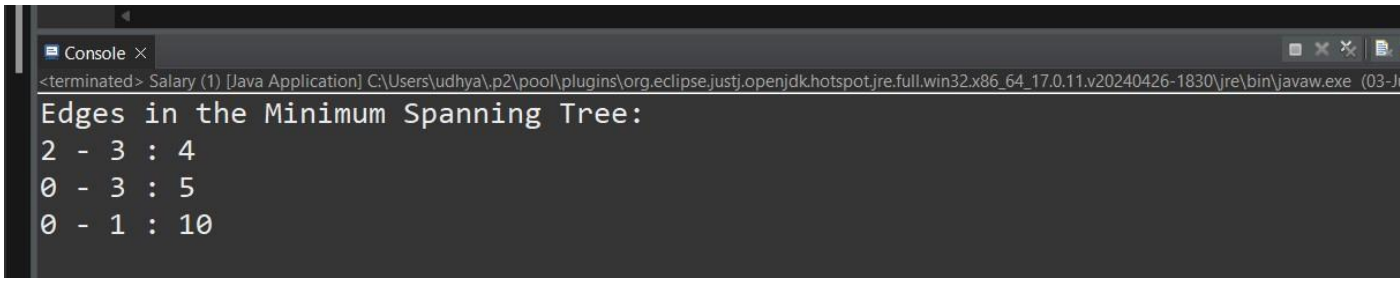
```
import java.util.*;


class Edge implements Comparable<Edge> {
```

```
    int src, dest, weight;

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

    @Override
    public int compareTo(Edge other) {
        return this.weight - other.weight;
    }
}

public class KruskalMST {
    private int V;
    private List<Edge> edges;

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

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

    public List<Edge> kruskalMST() {
        List<Edge> mst = new ArrayList<>();
```

```
    Collections.sort(edges);

    DisjointSet disjointSet = new DisjointSet(V);
    for (Edge edge : edges) {
        int srcParent = disjointSet.find(edge.src);
        int destParent = disjointSet.find(edge.dest);

        if (srcParent != destParent) {
            mst.add(edge);
            disjointSet.union(srcParent, destParent);
        }
    }

    return mst;
}

public static void main(String[] args) {
    int V = 4;
    KruskalMST graph = new KruskalMST(V);

    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.kruskalMST();

    System.out.println("Edges in the Minimum Spanning Tree:");
```

```
        for (Edge edge : mst) {
            System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight);
        }
    }
}

class DisjointSet {
    int[] parent;

    public DisjointSet(int n) {
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    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 xParent = find(x);
        int yParent = find(y);
        parent[yParent] = xParent;
    }
}
```

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.

```
package com.wipro.graphalgo;

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 Graph {
	int V, E;
	Edge[] edges;

	class Edge {
		int src, dest;
	}

	Graph(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(Graph 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 = 2;
                Graph graph = new Graph(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: