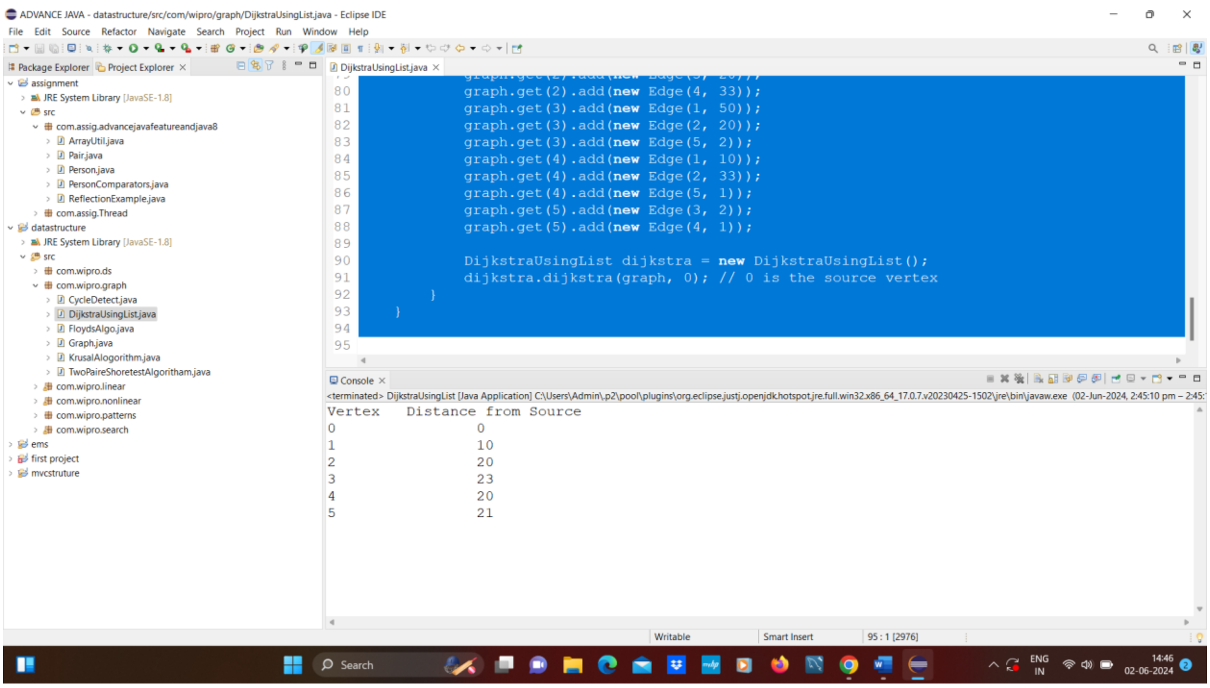
NAME:VISHAKHA AVINASH KALE
ASSIGEMENT:DAY9 AND 10
Task 1: Dijkstra's Shortest Path Finder Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

```
package com.wipro.graph;
import java.util.*;
public class DijkstraUsingList {

	static class Edge {
		int target;
		int weight;

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

	static class Node implements Comparable<Node>
{
		int vertex;
		int distance;

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

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

	public void dijkstra(List<List<Edge>> graph,
int src) {
		int V = graph.size();
		int[] dist = new int[V];
		Arrays.fill(dist, Integer.MAX_VALUE);
		dist[src] = 0;
```

```java
            PriorityQueue<Node> pq = new
PriorityQueue<>();
            pq.add(new Node(src, 0));

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

                for (Edge edge : graph.get(u)) {
                    int v = edge.target;
                    int weight = edge.weight;

                    if (dist[u] != Integer.MAX_VALUE
&& dist[u] + weight < dist[v]) {
                        dist[v] = dist[u] + weight;
                        pq.add(new Node(v, dist[v]));
                    }
                }
            }

            printSolution(dist, V);
        }

        private void printSolution(int dist[], int V)
{
            System.out.println("Vertex \t Distance
from Source");
            for (int i = 0; i < V; i++) {
                System.out.println(i + " \t\t " +
dist[i]);
            }
        }

        public static void main(String[] args) {
            int V = 6;
            List<List<Edge>> graph = new
ArrayList<>(V);

            for (int i = 0; i < V; i++) {
                graph.add(new ArrayList<>());
            }

            // Add edges
```

```java
        graph.get(0).add(new Edge(1, 10));
        graph.get(0).add(new Edge(2, 20));
        graph.get(1).add(new Edge(0, 10));
        graph.get(1).add(new Edge(3, 50));
        graph.get(1).add(new Edge(4, 10));
        graph.get(2).add(new Edge(0, 20));
        graph.get(2).add(new Edge(3, 20));
        graph.get(2).add(new Edge(4, 33));
        graph.get(3).add(new Edge(1, 50));
        graph.get(3).add(new Edge(2, 20));
        graph.get(3).add(new Edge(5, 2));
        graph.get(4).add(new Edge(1, 10));
        graph.get(4).add(new Edge(2, 33));
        graph.get(4).add(new Edge(5, 1));
        graph.get(5).add(new Edge(3, 2));
        graph.get(5).add(new Edge(4, 1));

        DijkstraUsingList dijkstra = new DijkstraUsingList();
        dijkstra.dijkstra(graph, 0); // 0 is the source vertex
    }
}
```

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.

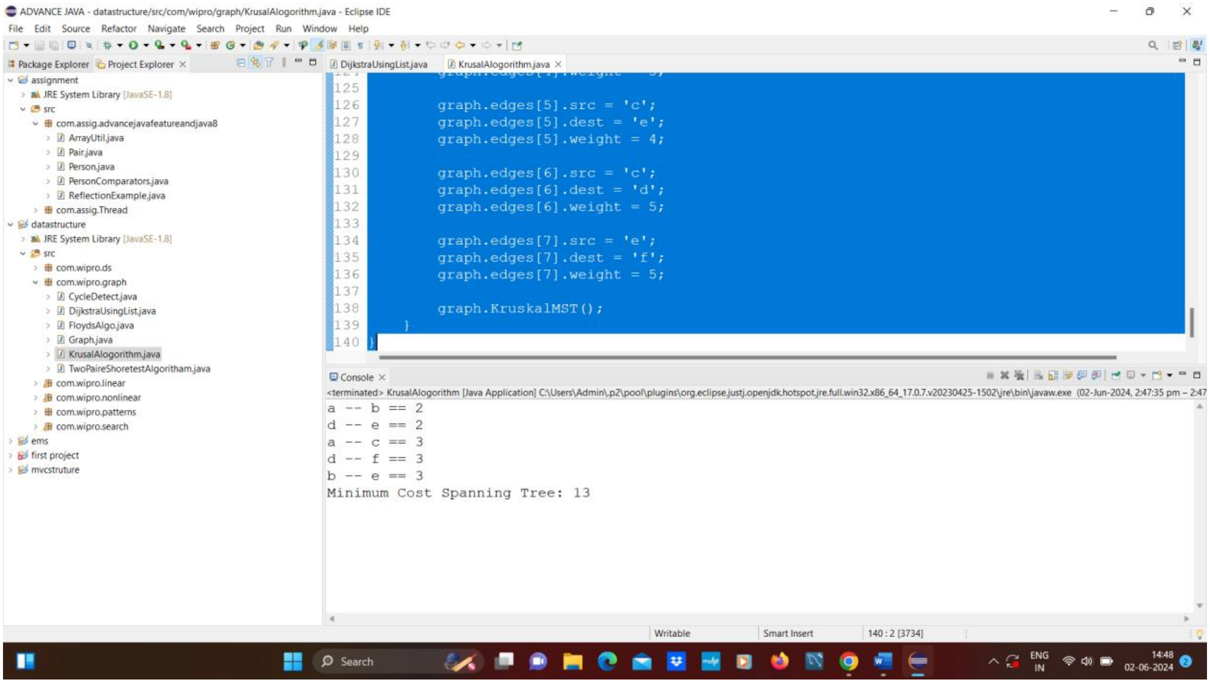
```
package com.wipro.graph;
import java.util.*;

class Edge implements Comparable<Edge> {
    char src, dest;
    int weight;

    public int compareTo(Edge compareEdge) {
        return this.weight - compareEdge.weight;
    }
}

class Subset {
    int parent, rank;
}

public class KrusalAlogorithm {
    int V, E;
    Edge[] edges;
    Map<Character, Integer> charToIndex = new
HashMap<>();
    Map<Integer, Character> indexToChar = new
HashMap<>();

    KrusalAlogorithm(int v, int e) {
        V = v;
        E = e;
        edges = new Edge[E];
        for (int i = 0; i < e; ++i) {
            edges[i] = new Edge();
        }

        char[] nodes = {'a', 'b', 'c', 'd', 'e',
'f'};
        for (int i = 0; i < nodes.length; i++) {
            charToIndex.put(nodes[i], i);
            indexToChar.put(i, nodes[i]);
        }
    }
```

```java
    int find(Subset[] subsets, int i) {
        if (subsets[i].parent != i) {
            subsets[i].parent = find(subsets,
subsets[i].parent);
        }
        return subsets[i].parent;
    }

    void union(Subset[] subsets, int x, int y) {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        if (subsets[xroot].rank <
subsets[yroot].rank) {
            subsets[xroot].parent = yroot;
        } else if (subsets[xroot].rank >
subsets[yroot].rank) {
            subsets[yroot].parent = xroot;
        } else {
            subsets[yroot].parent = xroot;
            subsets[xroot].rank++;
        }
    }

    void KruskalMST() {
        Edge[] result = new Edge[V];
        int e = 0;
        int i = 0;
        for (i = 0; i < V; ++i) {
            result[i] = new Edge();
        }

        Arrays.sort(edges);

        Subset[] subsets = new Subset[V];
        for (i = 0; i < V; ++i) {
            subsets[i] = new Subset();
        }

        for (int v = 0; v < V; ++v) {
            subsets[v].parent = v;
            subsets[v].rank = 0;
        }
```

```java
        i = 0;

        while (e < V - 1) {
            Edge nextEdge = edges[i++];

            int x = find(subsets,
charToIndex.get(nextEdge.src));
            int y = find(subsets,
charToIndex.get(nextEdge.dest));

            if (x != y) {
                result[e++] = nextEdge;
                union(subsets, x, y);
            }
        }

        int minimumCost = 0;
        for (i = 0; i < e; ++i) {
            System.out.println(result[i].src + " -- "
+ result[i].dest + " == " + result[i].weight);
            minimumCost += result[i].weight;
        }
        System.out.println("Minimum Cost Spanning
Tree: " + minimumCost);
    }

    public static void main(String[] args) {
        int V = 6;
        int E = 8;
        KrusalAlogorithm graph = new
KrusalAlogorithm(V, E);

        graph.edges[0].src = 'a';
        graph.edges[0].dest = 'b';
        graph.edges[0].weight = 2;

        graph.edges[1].src = 'd';
        graph.edges[1].dest = 'e';
        graph.edges[1].weight = 2;

        graph.edges[2].src = 'a';
        graph.edges[2].dest = 'c';
        graph.edges[2].weight = 3;
```

```
            graph.edges[3].src = 'd';
            graph.edges[3].dest = 'f';
            graph.edges[3].weight = 3;

            graph.edges[4].src = 'b';
            graph.edges[4].dest = 'e';
            graph.edges[4].weight = 3;

            graph.edges[5].src = 'c';
            graph.edges[5].dest = 'e';
            graph.edges[5].weight = 4;

            graph.edges[6].src = 'c';
            graph.edges[6].dest = 'd';
            graph.edges[6].weight = 5;

            graph.edges[7].src = 'e';
            graph.edges[7].dest = 'f';
            graph.edges[7].weight = 5;

            graph.KruskalMST();
      }
}
```

OUTPUT:



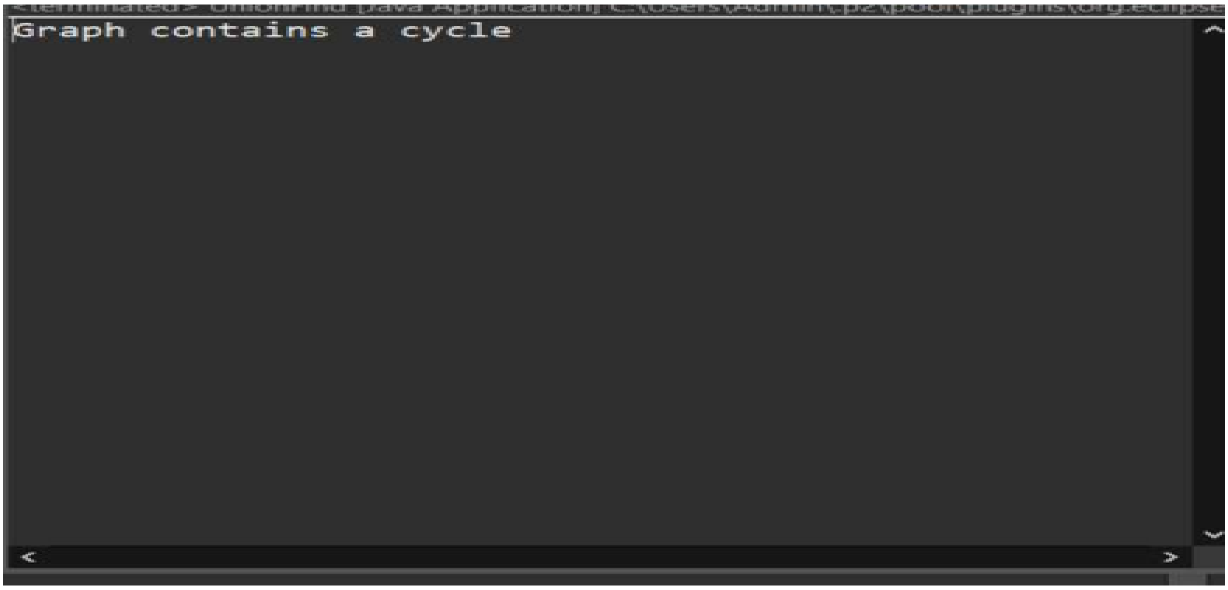
Task 3: Union-Find for Cycle Detection

Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

```
    package com.wipro.graph;
import java.util.*;
public class UnionFind {
 private int[] parent;
 private int[] rank;{
 // Constructor to initialize the Union-Find data
structurepublic UnionFind(int size) {
 parent = new int[size];
 rank = new int[size];
 for (int i = 0; i < size; i++) {
 parent[i] = i;
 rank[i] = 0;
 }
 }
// Find with path compression
public int find(int p) {

 if (parent[p] != p) {
 parent[p] = find(parent[p]);
 }
 return parent[p];
 }
 // Union by rank
 public void union(int p, int q) {
 int rootP = find(p);
 int rootQ = find(q);
 if (rootP != rootQ) {
 if (rank[rootP] > rank[rootQ]) {
 parent[rootQ] = rootP;
 } else if (rank[rootP] < rank[rootQ]) {
 parent[rootP] = rootQ;
 } else {
 parent[rootQ] = rootP;
 rank[rootP]++;
 }
 }
 }
 // Method to detect cycle in an undirected graph
 public boolean hasCycle(List<int[]> edges) {
```

```
for (int[] edge : edges) {
int u = edge[0];
int v = edge[1];
int rootU = find(u);
int rootV = find(v);
if (rootU == rootV) {
return true; // Cycle detected
} else {
union(u, v);
}
}
return false; // No cycle detected
}
public static void main(String[] args) {
int numberOfVertices = 5;
UnionFind uf = new UnionFind(int numberOfVertices);
List<int[]> edges = new ArrayList<>();
edges.add(new int[]{0, 1});
edges.add(new int[]{1, 2});
edges.add(new int[]{2, 3});
edges.add(new int[]{3, 4});
edges.add(new int[]{4, 0}); // Adding this edge
creates a cycle
if (uf.hasCycle(edges)) {
System.out.println("Graph contains a cycle"); } else
{
System.out.println("Graph does not contain a
cycle"); }
}
}
```

OUTPUT:


Graph contains a cycle