Day 9&10

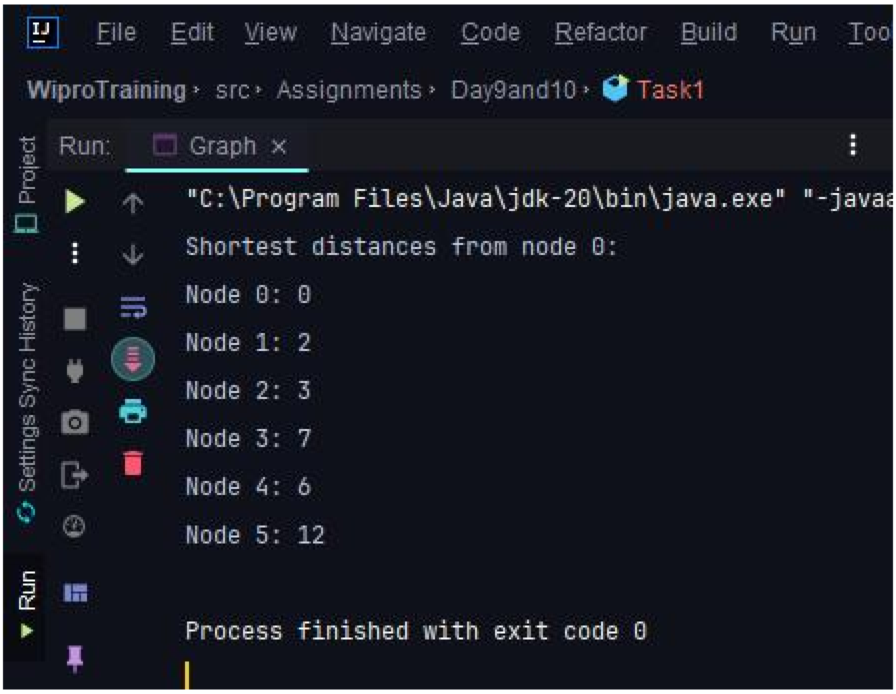
Task 1: Dijkstra's Shortest Path Finder
Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

Program:

```
package Assignments.Day9and10;

import java.util.*;

class Task1 {
   private int V;
   private List<Edge>[] adj;

   static class Edge {
      int dest;
      int weight;

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

   public Task1(int v) {
      V = v;
      adj = new ArrayList[v];
      for (int i = 0; i < v; ++i) {
         adj[i] = new ArrayList<>();
      }
   }

   public void addEdge(int u, int v, int weight) {
      adj[u].add(new Edge(v, weight));
      adj[v].add(new Edge(u, weight));
   }

   public void dijkstra(int start) {
      PriorityQueue<Edge>                pq                =                new
PriorityQueue<>(Comparator.comparingInt(e -> e.weight));
```

```
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[start] = 0;
        pq.add(new Edge(start, 0));

        while (!pq.isEmpty()) {
            Edge curr = pq.poll();
            int u = curr.dest;

            for (Edge neighbor : adj[u]) {
                int v = neighbor.dest;
                int weight = neighbor.weight;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.add(new Edge(v, dist[v]));
                }
            }
        }

        System.out.println("Shortest distances from node " + start + ":");
        for (int i = 0; i < V; ++i) {
            System.out.println("Node " + i + ": " + dist[i]);
        }
    }

    public static void main(String[] args) {
        Task1 graph = new Task1(6);
        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 2, 4);
        graph.addEdge(1, 2, 1);
        graph.addEdge(1, 3, 7);
        graph.addEdge(2, 4, 3);
        graph.addEdge(3, 4, 1);
        graph.addEdge(3, 5, 5);

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

Output:



```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaa
Shortest distances from node 0:
Node 0: 0
Node 1: 2
Node 2: 3
Node 3: 7
Node 4: 6
Node 5: 12

Process finished with exit code 0
```

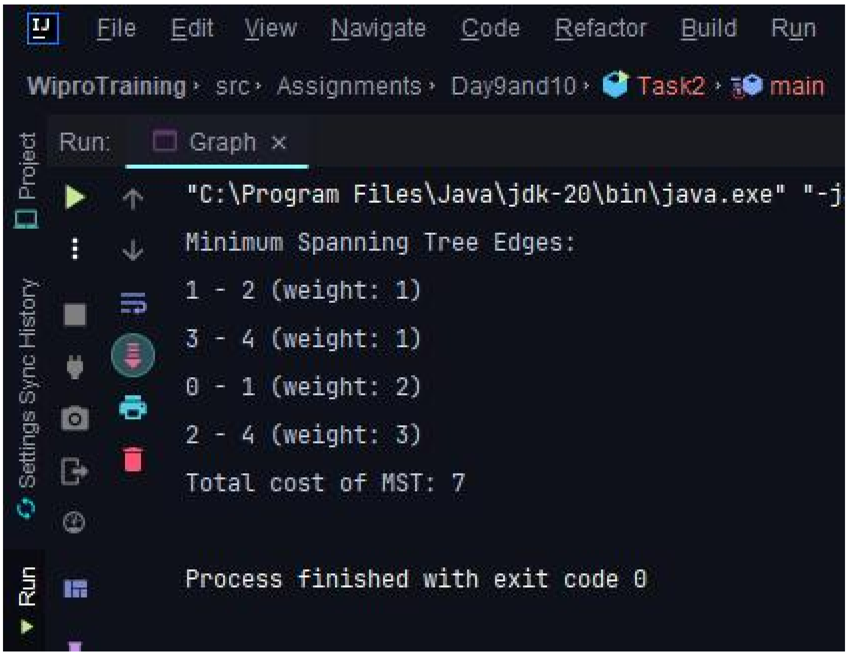
Task 2: Kruskal’s Algorithm for MST
Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

Program:

```
package Assignments.Day9and10;

import java.util.*;

class Task2 {
    private final int V;
    private final List<Edge> edges;

    static class Edge {
        int src;
        int dest;
        int weight;

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

```
public Task2(int v) {
    V = v;
    edges = new ArrayList<>();
}

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

// Find set of vertex i
private int find(int i, int[] parent) {
    if (parent[i] != i)
        parent[i] = find(parent[i], parent);
    return parent[i];
}

// Does union of i and j. Returns false if i and j are already in the same set.
private boolean union(int i, int j, int[] parent) {
    int a = find(i, parent);
    int b = find(j, parent);
    if (a == b)
        return false;
    parent[a] = b;
    return true;
}

// Finds MST using Kruskal's algorithm
public void kruskalMST() {
    edges.sort(Comparator.comparingInt(e -> e.weight));
    int[] parent = new int[V];
    for (int i = 0; i < V; i++)
        parent[i] = i;

    int minCost = 0;
    List<Edge> mstEdges = new ArrayList<>();

    for (Edge edge : edges) {
        if (union(edge.src, edge.dest, parent)) {
            mstEdges.add(edge);
            minCost += edge.weight;
        }
    }

    System.out.println("Minimum Spanning Tree Edges:");
    for (Edge edge : mstEdges) {
            System.out.println(edge.src + " - " + edge.dest + " (weight: " +
edge.weight + ")");
        }
        System.out.println("Total cost of MST: " + minCost);
    }

    public static void main(String[] args) {

        Task2 graph = new Task2(6);

        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 2, 4);
        graph.addEdge(1, 2, 1);
        graph.addEdge(1, 3, 7);
        graph.addEdge(2, 4, 3);
        graph.addEdge(3, 4, 1);

        graph.kruskalMST();
    }
}
```

Output:



```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-j
Minimum Spanning Tree Edges:
1 - 2 (weight: 1)
3 - 4 (weight: 1)
0 - 1 (weight: 2)
2 - 4 (weight: 3)
Total cost of MST: 7

Process finished with exit code 0
```

Task 3: Union-Find for Cycle Detection

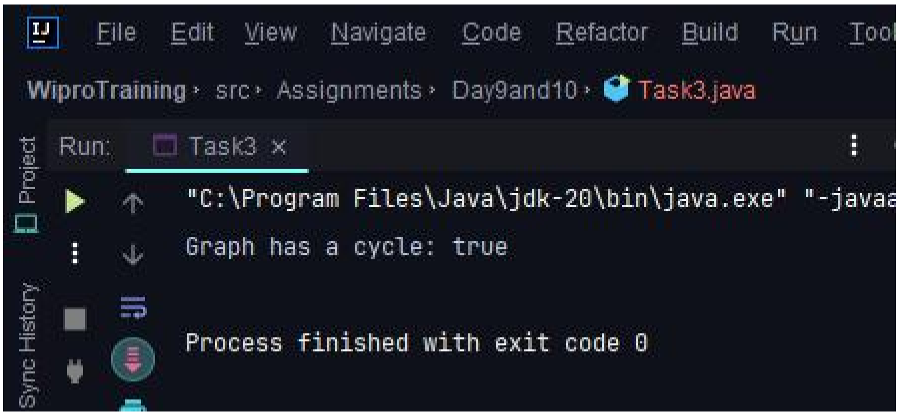
Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

Program:

```
package Assignments.Day9and10;

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;
        }
    }

    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        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[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
}
```

```java
package Assignments.Day9and10;

import java.util.ArrayList;
import java.util.List;

class Graph {
    private final int V;
    private final List<int []> edges;

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

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

    public boolean hasCycle() {
        UnionFind uf = new UnionFind(V);
        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            if (uf.find(u) == uf.find(v)) {
                return false;
            }
            uf.union(u, v);
        }
        return true;
    }
}


public  class Task3 {
    public static void main(String[] args) {
        Graph graph = new Graph(4);
        graph.addEdge(0, 1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);

        boolean hasCycle = graph.hasCycle();
        System.out.println("Graph has a cycle: " + hasCycle);
    }
}
```

Output:



```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaa
Graph has a cycle: true

Process finished with exit code 0
```