GRAPH-SEARCH-ALGORITHMS ASSIGNMENT

(DAY-8,9,10)

Task 4: Graph Edge Addition Validation

Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

GraphAddEdge.java

```java
package assignment;
import java.util.List;
import java.util.ArrayList;
class Graph {
    private int V;
    private List<Integer>[] adj;
    Graph(int V) {
        this.V = V;
        adj = new ArrayList[V];
        for (int i = 0; i < V; ++i)
            adj[i] = new ArrayList<>();
    }
    void addEdge(int u, int v) {
        adj[u].add(v);
    }
    boolean isCyclicUtil(int v, boolean[] visited, boolean[] recStack) {
        if (!visited[v]) {
            visited[v] = true;
            recStack[v] = true;

            for (int neighbor : adj[v]) {
                if (!visited[neighbor] && isCyclicUtil(neighbor, visited, recStack))
                    return true;
                else if (recStack[neighbor])
                    return true;
            }
        }
        recStack[v] = false; // remove the vertex from recursion stack
        return false;
    }
    boolean isCyclic() {
        boolean[] visited = new boolean[V];
        boolean[] recStack = new boolean[V];
        for (int i = 0; i < V; i++)
            if (isCyclicUtil(i, visited, recStack))
                return true;
                return false;
    }
```

Console

```
<terminated> GraphAddEdge (1) [Java Application] C:\Pro
Adding edge 3 -> 0 creates a cycle? false
```

GraphAddEdge.java

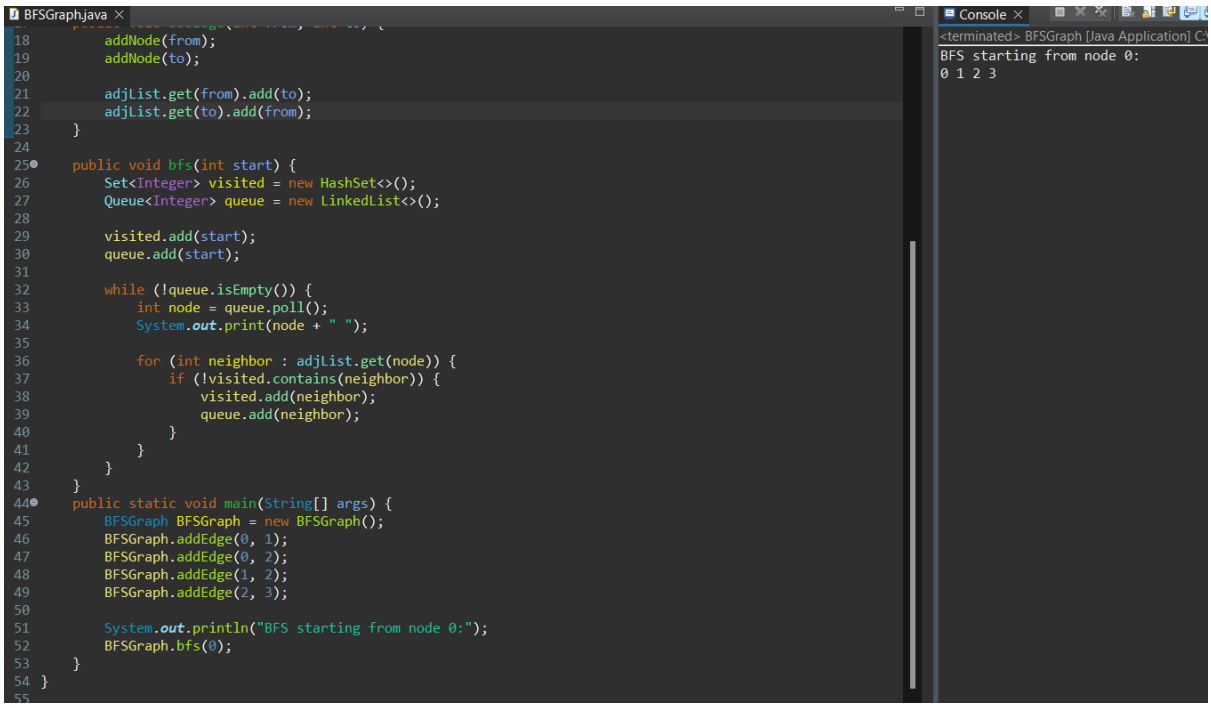
```java
                if (!visited[neighbor] && isCyclicUtil(neighbor, visited, recStack))
                    return true;
                else if (recStack[neighbor])
                    return true;
            }
        }
        recStack[v] = false; // remove the vertex from recursion stack
        return false;
    }
    boolean isCyclic() {
        boolean[] visited = new boolean[V];
        boolean[] recStack = new boolean[V];
        for (int i = 0; i < V; i++)
            if (isCyclicUtil(i, visited, recStack))
                return true;
                return false;
    }
    boolean addEdgeAndCheckCycle(int u, int v) {
        addEdge(u, v);
        return !isCyclic();
    }
}

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

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

        System.out.println("Adding edge 3 -> 0 creates a cycle? " + graph.addEdgeAndCheckCycle(3, 0));
```

Console

```
<terminated> GraphAddEdge (1) [Java Application] C:\P
Adding edge 3 -> 0 creates a cycle? false
```

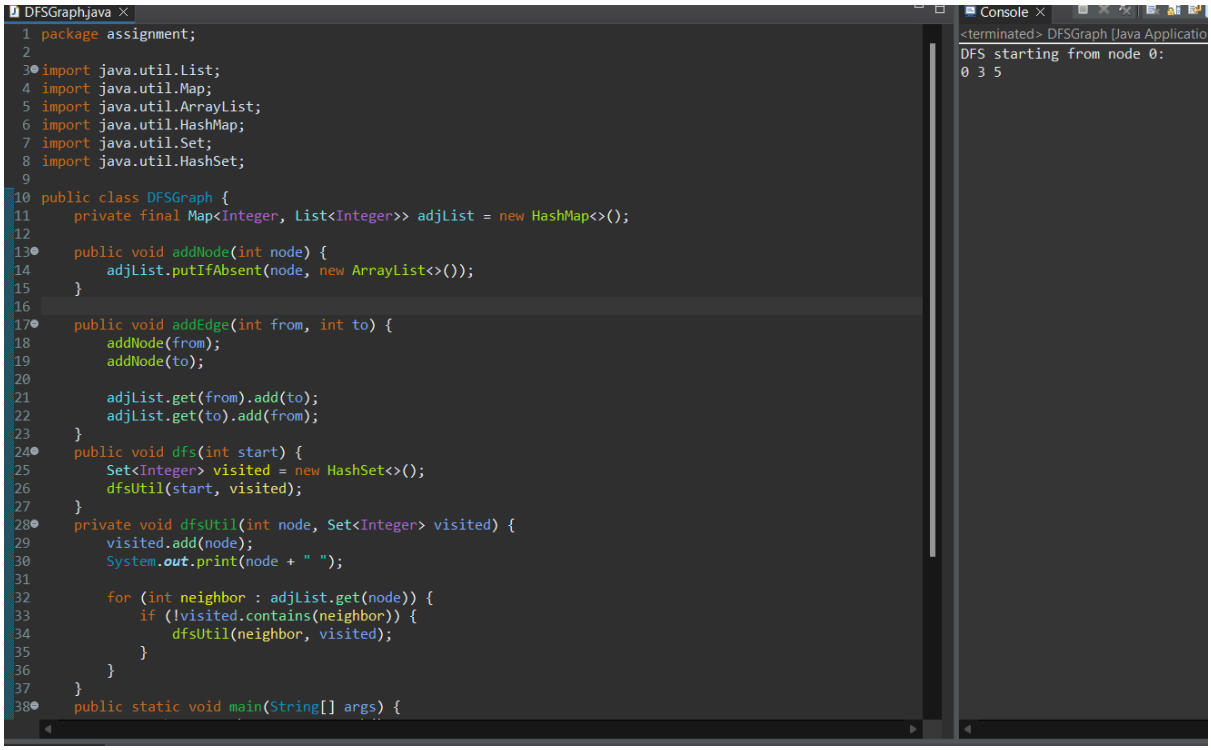
Task 5: Breadth-First Search (BFS) Implementation

For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

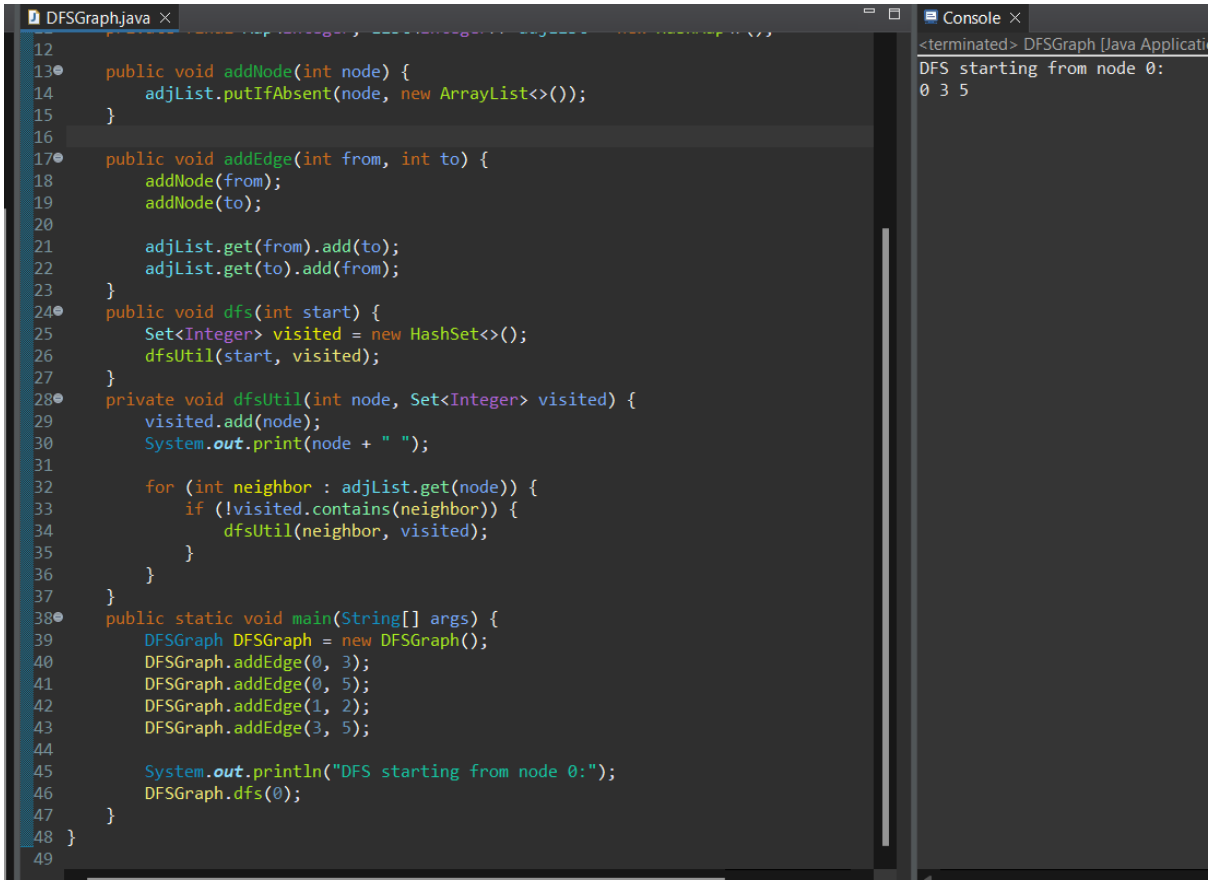


Task 6: Depth-First Search (DFS) Recursive

Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.




```java
package assignment;

import java.util.List;
import java.util.Map;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;

public class DFSGraph {
    private final Map<Integer, List<Integer>> adjList = new HashMap<>();

    public void addNode(int node) {
        adjList.putIfAbsent(node, new ArrayList<>());
    }

    public void addEdge(int from, int to) {
        addNode(from);
        addNode(to);

        adjList.get(from).add(to);
        adjList.get(to).add(from);
    }
    public void dfs(int start) {
        Set<Integer> visited = new HashSet<>();
        dfsUtil(start, visited);
    }
    private void dfsUtil(int node, Set<Integer> visited) {
        visited.add(node);
        System.out.print(node + " ");

        for (int neighbor : adjList.get(node)) {
            if (!visited.contains(neighbor)) {
                dfsUtil(neighbor, visited);
            }
        }
    }
    public static void main(String[] args) {
```




```java

    public void addNode(int node) {
        adjList.putIfAbsent(node, new ArrayList<>());
    }

    public void addEdge(int from, int to) {
        addNode(from);
        addNode(to);

        adjList.get(from).add(to);
        adjList.get(to).add(from);
    }
    public void dfs(int start) {
        Set<Integer> visited = new HashSet<>();
        dfsUtil(start, visited);
    }
    private void dfsUtil(int node, Set<Integer> visited) {
        visited.add(node);
        System.out.print(node + " ");

        for (int neighbor : adjList.get(node)) {
            if (!visited.contains(neighbor)) {
                dfsUtil(neighbor, visited);
            }
        }
    }
    public static void main(String[] args) {
        DFSGraph DFSGraph = new DFSGraph();
        DFSGraph.addEdge(0, 3);
        DFSGraph.addEdge(0, 5);
        DFSGraph.addEdge(1, 2);
        DFSGraph.addEdge(3, 5);

        System.out.println("DFS starting from node 0:");
        DFSGraph.dfs(0);
    }
}

```

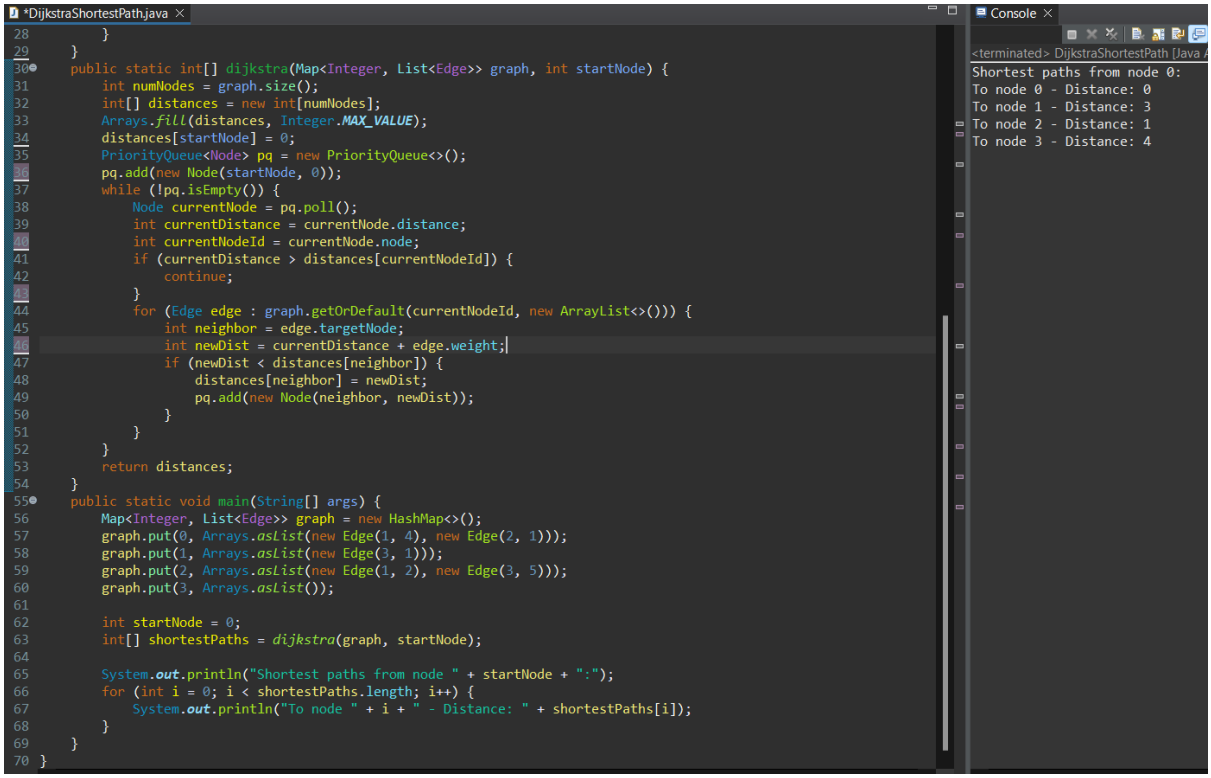
Day 9 and 10:

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.

```java
package assignment;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.ArrayList;


public class DijkstraShortestPath {

    static class Edge {
        int targetNode;
        int weight;

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

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

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

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

```
Console
<terminated> DijkstraShortestPath [Java
Shortest paths from node 0:
To node 0 - Distance: 0
To node 1 - Distance: 3
To node 2 - Distance: 1
To node 3 - Distance: 4
```

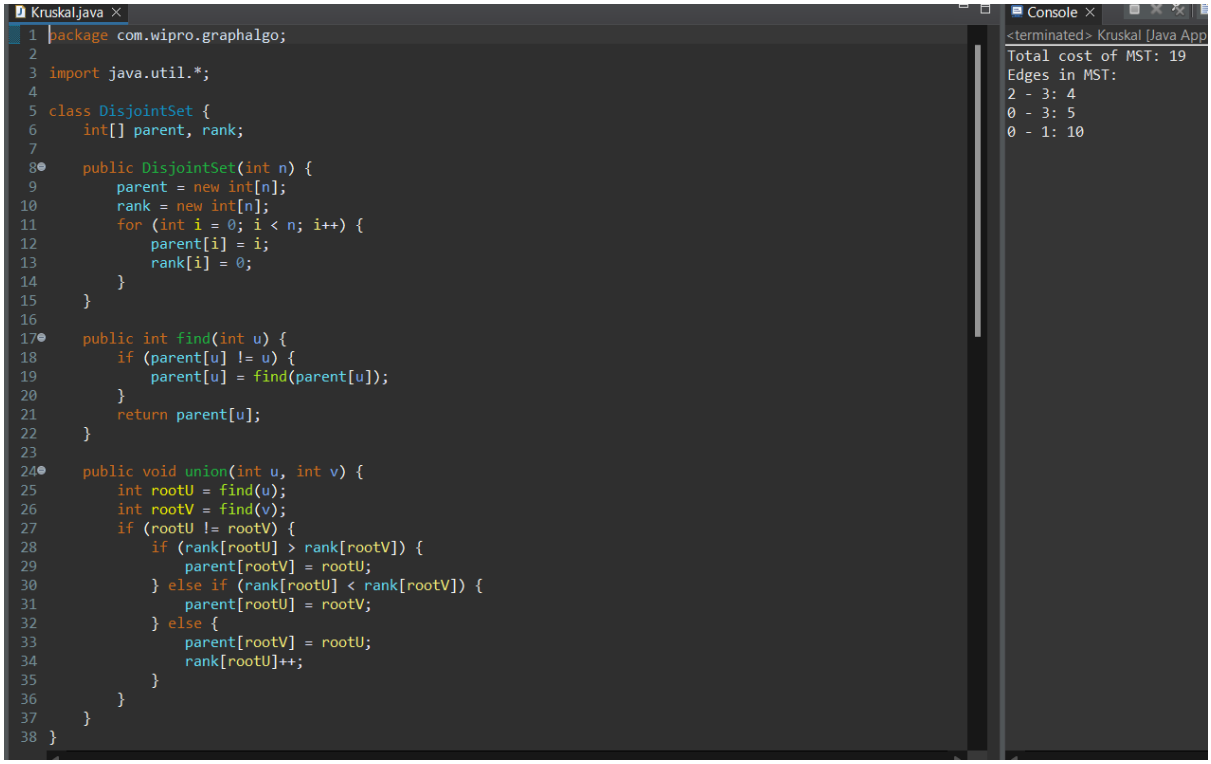


```java
        }
    }
    public static int[] dijkstra(Map<Integer, List<Edge>> graph, int startNode) {
        int numNodes = graph.size();
        int[] distances = new int[numNodes];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[startNode] = 0;
        PriorityQueue<Node> pq = new PriorityQueue<>();
        pq.add(new Node(startNode, 0));
        while (!pq.isEmpty()) {
            Node currentNode = pq.poll();
            int currentDistance = currentNode.distance;
            int currentNodeId = currentNode.node;
            if (currentDistance > distances[currentNodeId]) {
                continue;
            }
            for (Edge edge : graph.getOrDefault(currentNodeId, new ArrayList<>())) {
                int neighbor = edge.targetNode;
                int newDist = currentDistance + edge.weight;
                if (newDist < distances[neighbor]) {
                    distances[neighbor] = newDist;
                    pq.add(new Node(neighbor, newDist));
                }
            }
        }
        return distances;
    }
    public static void main(String[] args) {
        Map<Integer, List<Edge>> graph = new HashMap<>();
        graph.put(0, Arrays.asList(new Edge(1, 4), new Edge(2, 1)));
        graph.put(1, Arrays.asList(new Edge(3, 1)));
        graph.put(2, Arrays.asList(new Edge(1, 2), new Edge(3, 5)));
        graph.put(3, Arrays.asList());

        int startNode = 0;
        int[] shortestPaths = dijkstra(graph, startNode);

        System.out.println("Shortest paths from node " + startNode + ":");
        for (int i = 0; i < shortestPaths.length; i++) {
            System.out.println("To node " + i + " - Distance: " + shortestPaths[i]);
        }
    }
}
```

```
Console
<terminated> DijkstraShortestPath [Java
Shortest paths from node 0:
To node 0 - Distance: 0
To node 1 - Distance: 3
To node 2 - Distance: 1
To node 3 - Distance: 4
```

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.java

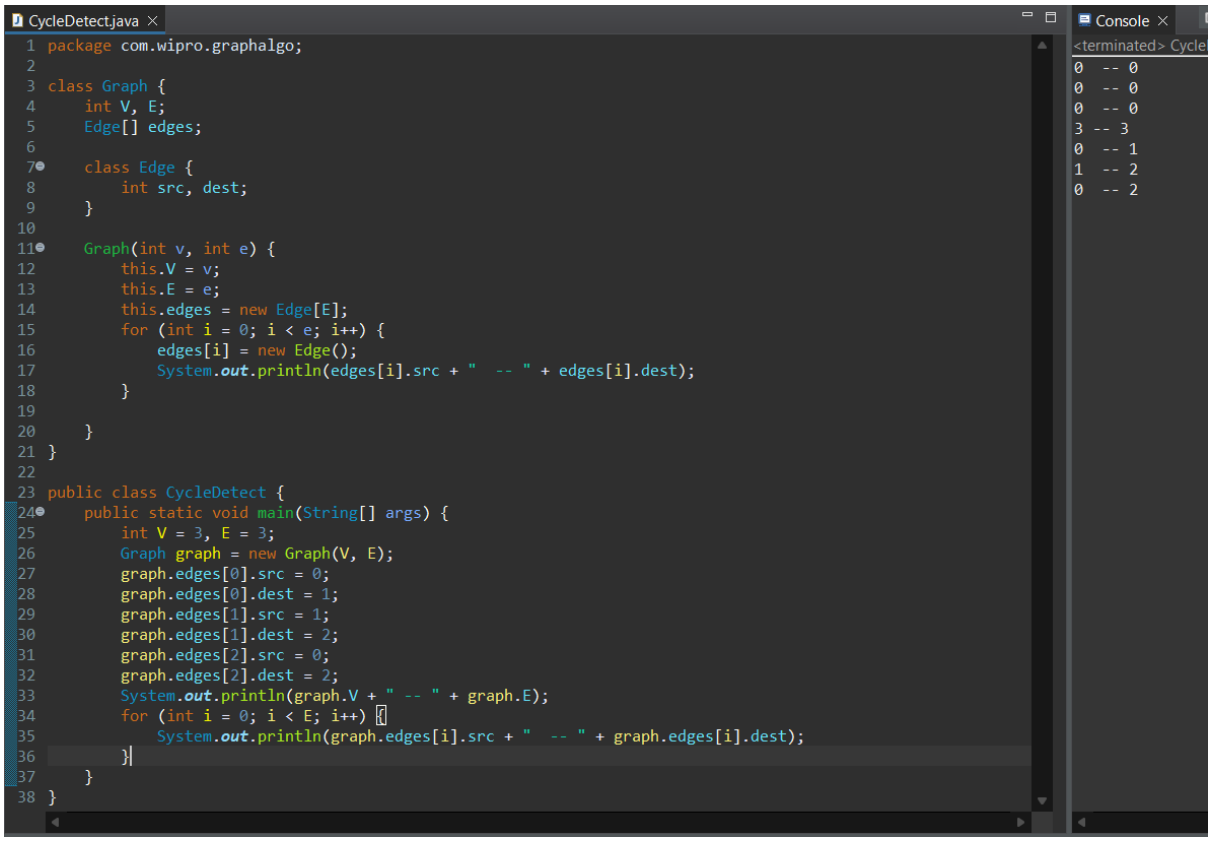
```java
package com.wipro.graphalgo;

import java.util.*;

class DisjointSet {
    int[] parent, rank;

    public DisjointSet(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 u) {
        if (parent[u] != u) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    public void union(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);
        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
}
```

Console

```
<terminated> Kruskal [Java App
Total cost of MST: 19
Edges in MST:
2 - 3: 4
0 - 3: 5
0 - 1: 10
```

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.



CycleDetect.java

```java
package com.wipro.graphalgo;

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 class CycleDetect {
    public static void main(String[] args) {
        int V = 3, E = 3;
        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);
        }
    }
}
```

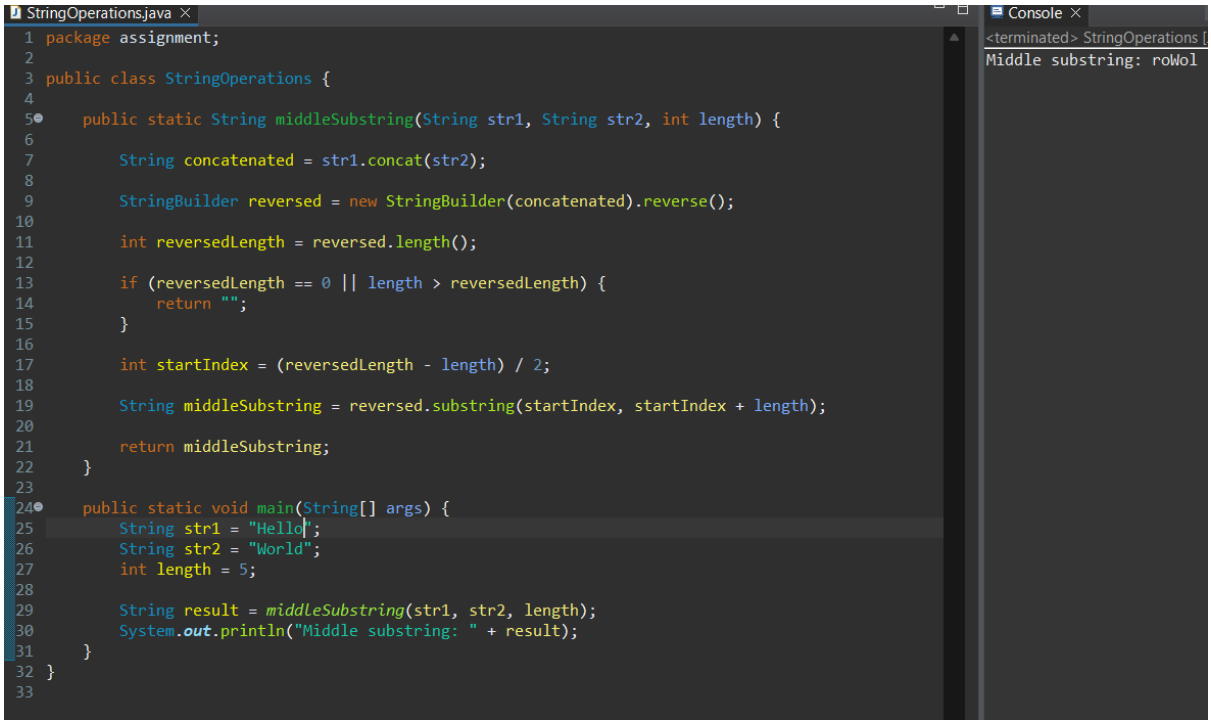
Console

```
<terminated> Cycle
0  -- 0
0  -- 0
0  -- 0
3 -- 3
0  -- 1
1  -- 2
0  -- 2
```

Day 11:

Task 1: String Operations

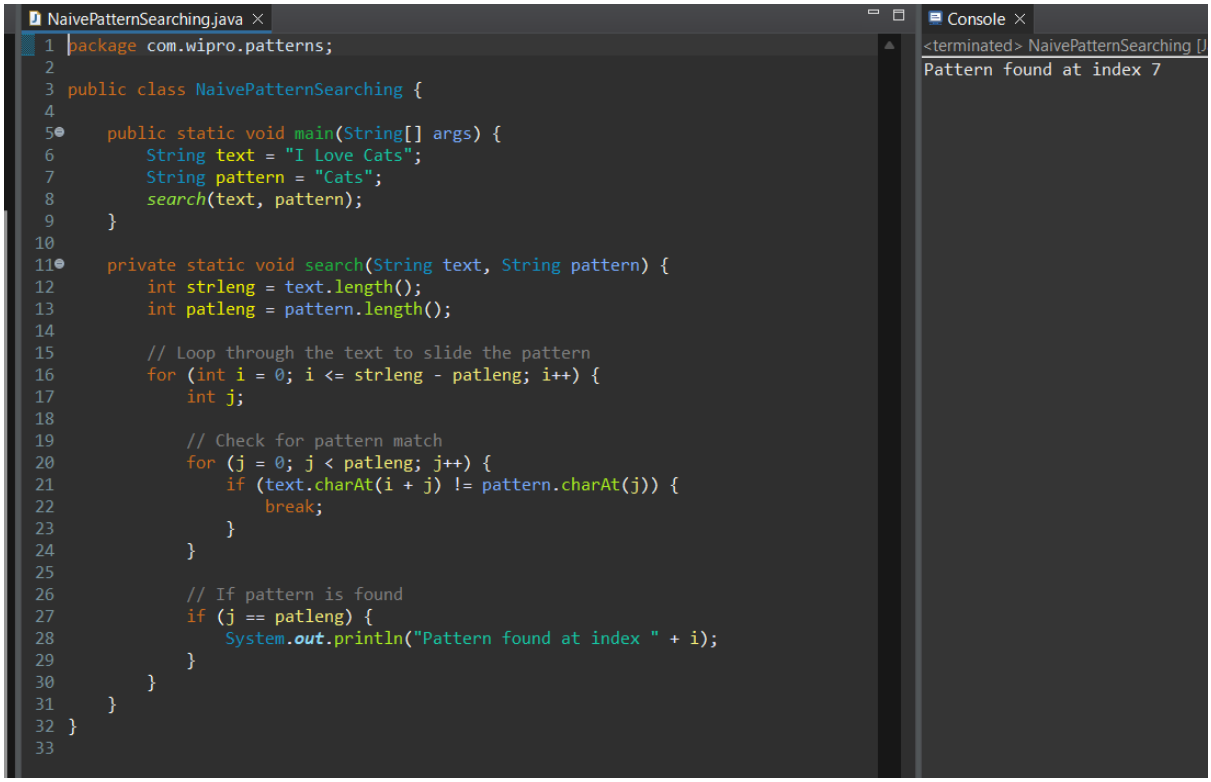
Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.




```java
package assignment;

public class StringOperations {

    public static String middleSubstring(String str1, String str2, int length) {

        String concatenated = str1.concat(str2);

        StringBuilder reversed = new StringBuilder(concatenated).reverse();

        int reversedLength = reversed.length();

        if (reversedLength == 0 || length > reversedLength) {
            return "";
        }

        int startIndex = (reversedLength - length) / 2;

        String middleSubstring = reversed.substring(startIndex, startIndex + length);

        return middleSubstring;
    }

    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";
        int length = 5;

        String result = middleSubstring(str1, str2, length);
        System.out.println("Middle substring: " + result);
    }
}
```

Task 2: Naive Pattern Search

Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.




```java
package com.wipro.patterns;

public class NaivePatternSearching {

    public static void main(String[] args) {
        String text = "I Love Cats";
        String pattern = "Cats";
        search(text, pattern);
    }

    private static void search(String text, String pattern) {
        int strleng = text.length();
        int patleng = pattern.length();

        // Loop through the text to slide the pattern
        for (int i = 0; i <= strleng - patleng; i++) {
            int j;

            // Check for pattern match
            for (j = 0; j < patleng; j++) {
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }

            // If pattern is found
            if (j == patleng) {
                System.out.println("Pattern found at index " + i);
            }
        }
    }
}
```