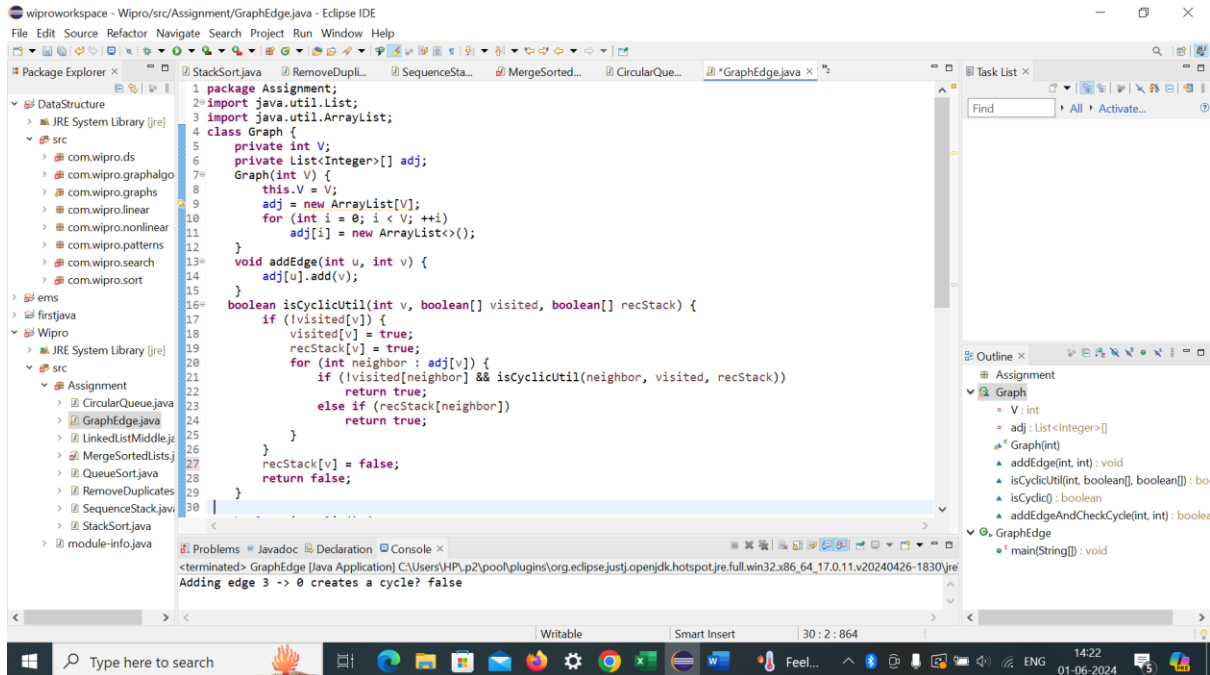


# Algorithms

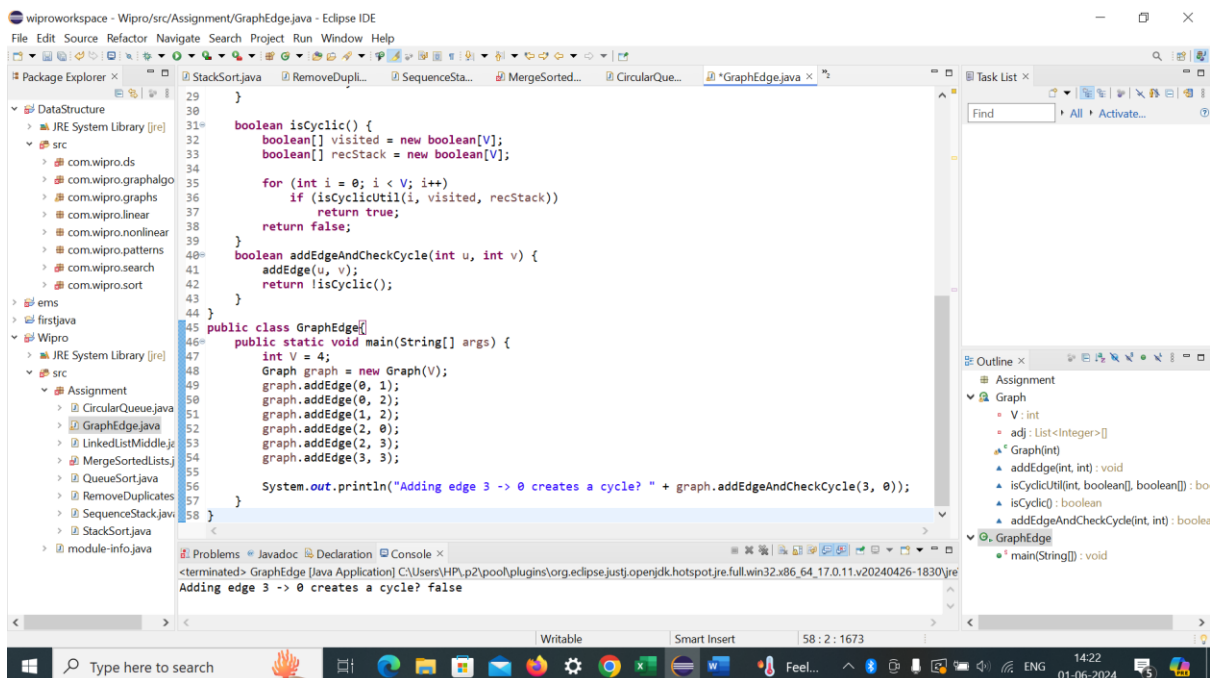
## Task 4: Graph Edge Addition Validation

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.



```
1 package Assignment;
2 import java.util.List;
3 import java.util.ArrayList;
4 class Graph {
5     private int V;
6     private List<Integer>[] adj;
7     Graph(int V) {
8         this.V = V;
9         adj = new ArrayList[V];
10        for (int i = 0; i < V; ++i)
11            adj[i] = new ArrayList<>();
12    }
13    void addEdge(int u, int v) {
14        adj[u].add(v);
15    }
16    boolean isCyclicUtil(int v, boolean[] visited, boolean[] recStack) {
17        if (!visited[v]) {
18            visited[v] = true;
19            recStack[v] = true;
20            for (int neighbor : adj[v]) {
21                if (!visited[neighbor] && isCyclicUtil(neighbor, visited, recStack))
22                    return true;
23                else if (recStack[neighbor])
24                    return true;
25            }
26            recStack[v] = false;
27            return false;
28        }
29    }
30 }
```

Console Output: Adding edge 3 -> 0 creates a cycle? false

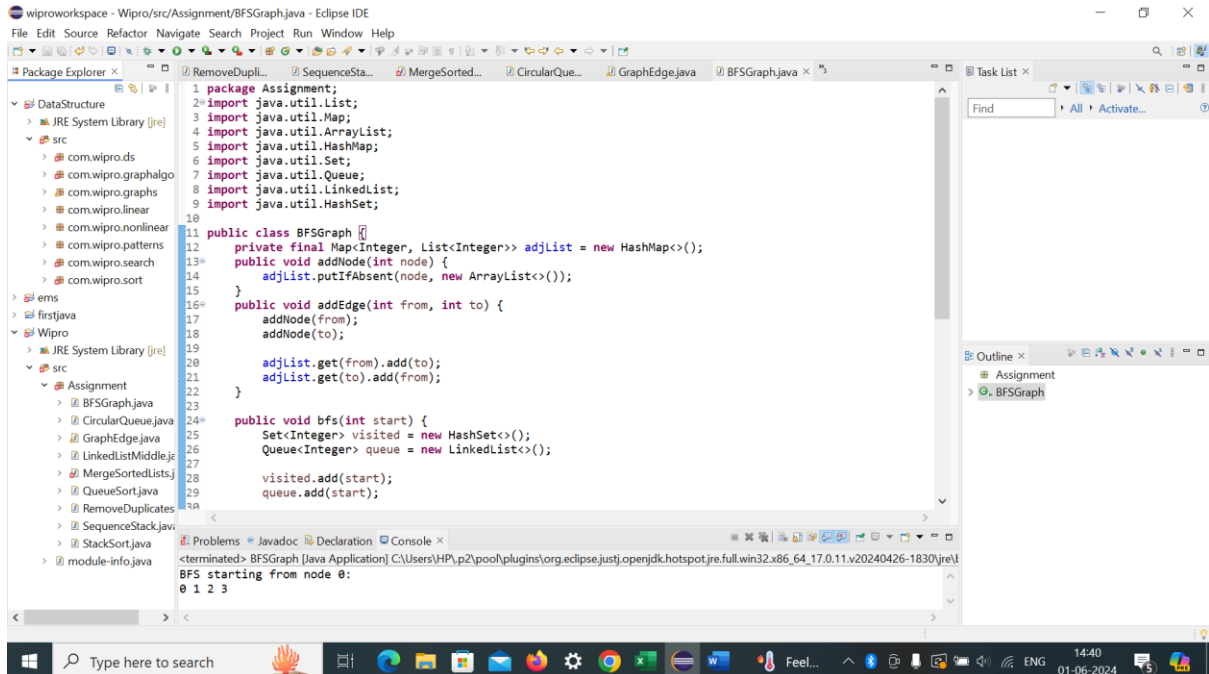


```
29 }
30
31 boolean isCyclic() {
32     boolean[] visited = new boolean[V];
33     boolean[] recStack = new boolean[V];
34
35     for (int i = 0; i < V; i++)
36         if (isCyclicUtil(i, visited, recStack))
37             return true;
38     return false;
39 }
40 boolean addEdgeAndCheckCycle(int u, int v) {
41     addEdge(u, v);
42     return !isCyclic();
43 }
44 }
45
46 public class GraphEdge {
47     public static void main(String[] args) {
48         int V = 4;
49         Graph graph = new Graph(V);
50         graph.addEdge(0, 1);
51         graph.addEdge(1, 2);
52         graph.addEdge(2, 0);
53         graph.addEdge(2, 3);
54         graph.addEdge(3, 3);
55
56         System.out.println("Adding edge 3 -> 0 creates a cycle? " + graph.addEdgeAndCheckCycle(3, 0));
57     }
58 }
```

Console Output: Adding edge 3 -> 0 creates a cycle? false

## Task 5: Breadth-First Search (BFS) Implementation

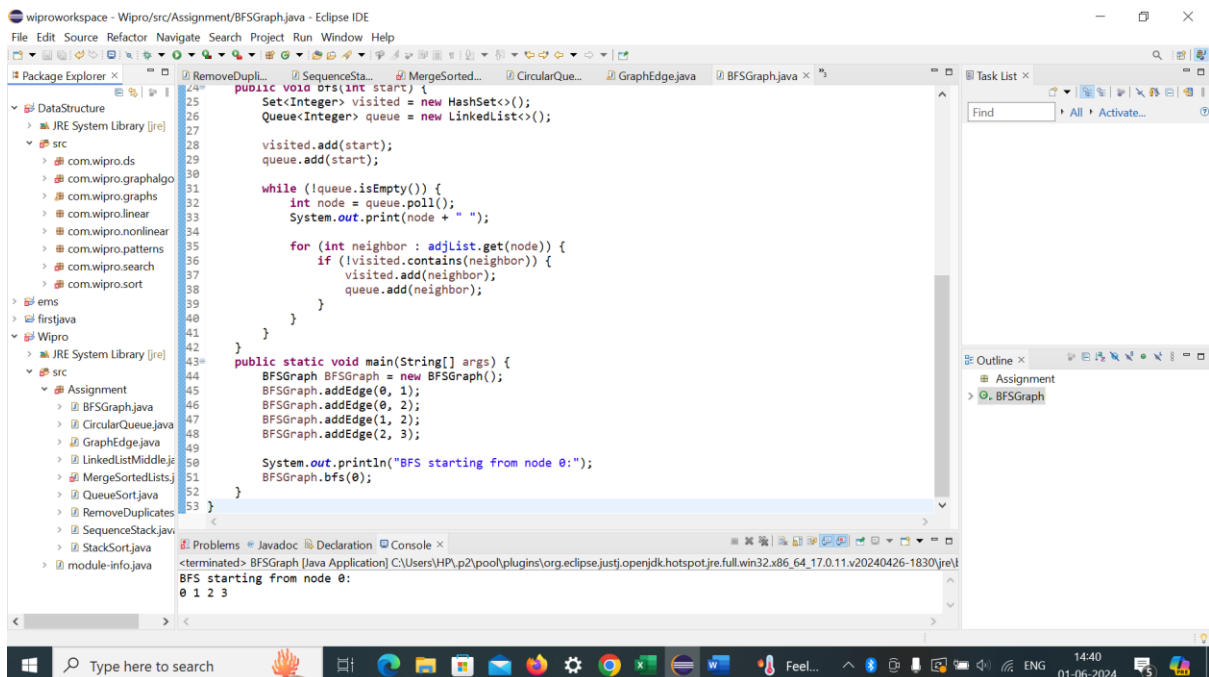
For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.



```
1 package Assignment;
2 import java.util.List;
3 import java.util.Map;
4 import java.util.ArrayList;
5 import java.util.HashMap;
6 import java.util.Set;
7 import java.util.Queue;
8 import java.util.LinkedList;
9 import java.util.HashSet;
10
11 public class BFSGraph {
12     private final Map<Integer, List<Integer>> adjList = new HashMap<>();
13     public void addNode(int node) {
14         adjList.putIfAbsent(node, new ArrayList<>());
15     }
16     public void addEdge(int from, int to) {
17         addNode(from);
18         addNode(to);
19         adjList.get(from).add(to);
20         adjList.get(to).add(from);
21     }
22     public void bfs(int start) {
23         Set<Integer> visited = new HashSet<>();
24         Queue<Integer> queue = new LinkedList<>();
25         visited.add(start);
26         queue.add(start);
27     }
28 }
29
30 public static void main(String[] args) {
31     BFSGraph bfs = new BFSGraph();
32     bfs.addEdge(0, 1);
33     bfs.addEdge(0, 2);
34     bfs.addEdge(1, 2);
35     bfs.bfs(0);
36 }
```

Console Output:

```
<terminated> BFSGraph [Java Application] C:\Users\HP\AppData\Local\Temp\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.11.v20240426-1830\jre\
BFS starting from node 0:
0 1 2 3
```



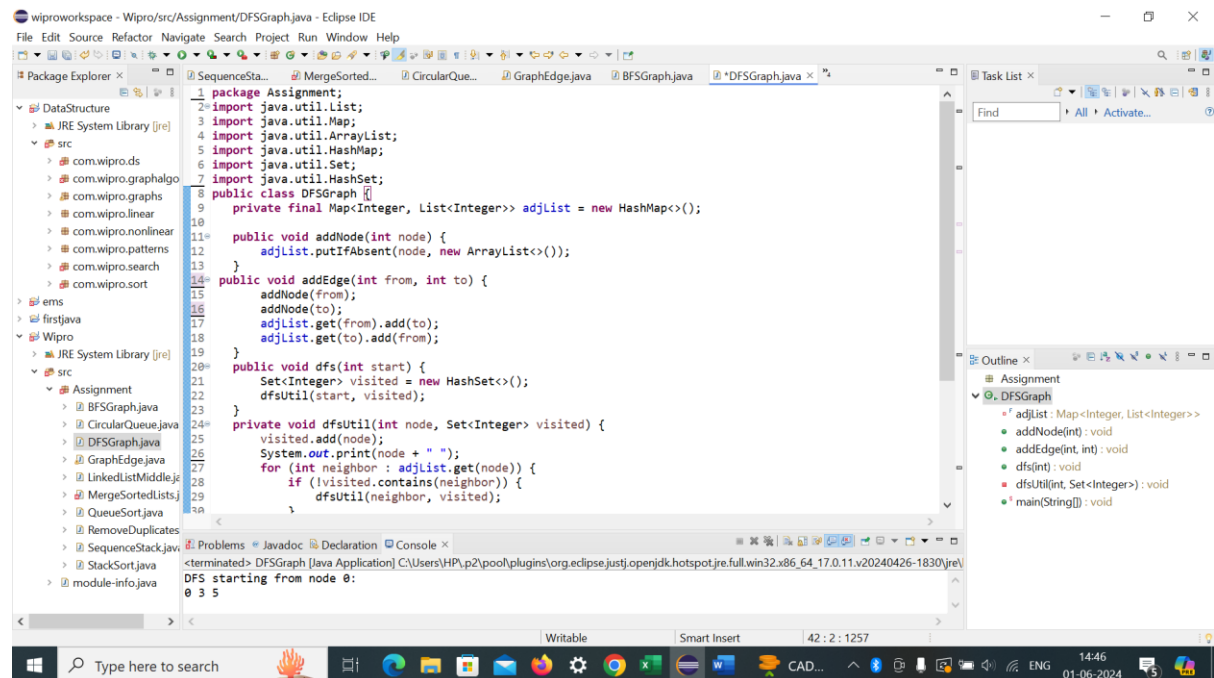
```
25     public void bfs(int start) {
26         Set<Integer> visited = new HashSet<>();
27         Queue<Integer> queue = new LinkedList<>();
28         visited.add(start);
29         queue.add(start);
30         while (!queue.isEmpty()) {
31             int node = queue.poll();
32             System.out.print(node + " ");
33             for (int neighbor : adjList.get(node)) {
34                 if (!visited.contains(neighbor)) {
35                     visited.add(neighbor);
36                     queue.add(neighbor);
37                 }
38             }
39         }
40     }
41 }
42
43 public static void main(String[] args) {
44     BFSGraph bfs = new BFSGraph();
45     bfs.addEdge(0, 1);
46     bfs.addEdge(0, 2);
47     bfs.addEdge(1, 2);
48     bfs.bfs(0);
49     System.out.println("BFS starting from node 0:");
50 }
51
52 public static void main(String[] args) {
53     BFSGraph bfs = new BFSGraph();
54     bfs.addEdge(0, 1);
55     bfs.addEdge(0, 2);
56     bfs.addEdge(1, 2);
57     bfs.bfs(0);
58     System.out.println("BFS starting from node 0:");
59 }
```

Console Output:

```
<terminated> BFSGraph [Java Application] C:\Users\HP\AppData\Local\Temp\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.11.v20240426-1830\jre\
BFS starting from node 0:
0 1 2 3
```

## Task 6: Depth-First Search (DFS) Recursive

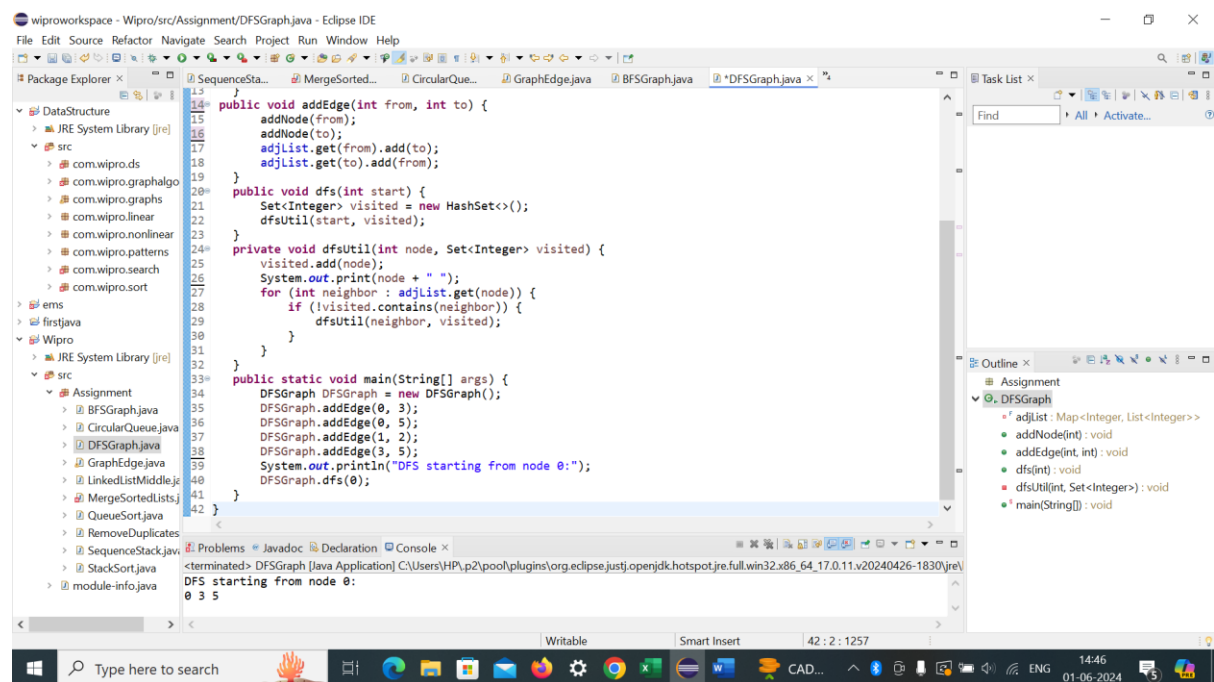
Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.



The screenshot shows the Eclipse IDE with the file `DFSGraph.java` open. The code implements a recursive Depth-First Search (DFS) algorithm for an undirected graph. The graph is represented by an adjacency list `adjList` of type `Map<Integer, List<Integer>>`. The `dfs` method starts the search from a given node, and `dfsUtil` is the recursive helper function that visits all nodes and prints them.

```
1 package Assignment;
2 import java.util.List;
3 import java.util.Map;
4 import java.util.ArrayList;
5 import java.util.HashMap;
6 import java.util.Set;
7 import java.util.HashSet;
8 public class DFSGraph {
9     private final Map<Integer, List<Integer>> adjList = new HashMap<>();
10
11     public void addNode(int node) {
12         adjList.putIfAbsent(node, new ArrayList<>());
13     }
14     public void addEdge(int from, int to) {
15         addNode(from);
16         addNode(to);
17         adjList.get(from).add(to);
18         adjList.get(to).add(from);
19     }
20     public void dfs(int start) {
21         Set<Integer> visited = new HashSet<>();
22         dfsUtil(start, visited);
23     }
24     private void dfsUtil(int node, Set<Integer> visited) {
25         visited.add(node);
26         System.out.print(node + " ");
27         for (int neighbor : adjList.get(node)) {
28             if (!visited.contains(neighbor)) {
29                 dfsUtil(neighbor, visited);
30             }
31         }
32     }
33
34     public static void main(String[] args) {
35         DFSGraph dfsGraph = new DFSGraph();
36         dfsGraph.addEdge(0, 3);
37         dfsGraph.addEdge(0, 5);
38         dfsGraph.addEdge(1, 2);
39         dfsGraph.addEdge(3, 5);
40         System.out.println("DFS starting from node 0:");
41         dfsGraph.dfs(0);
42     }
43 }
```

The console output shows the result of the DFS: `DFS starting from node 0: 0 3 5`. The Outline view on the right shows the structure of the `DFSGraph` class and its methods.

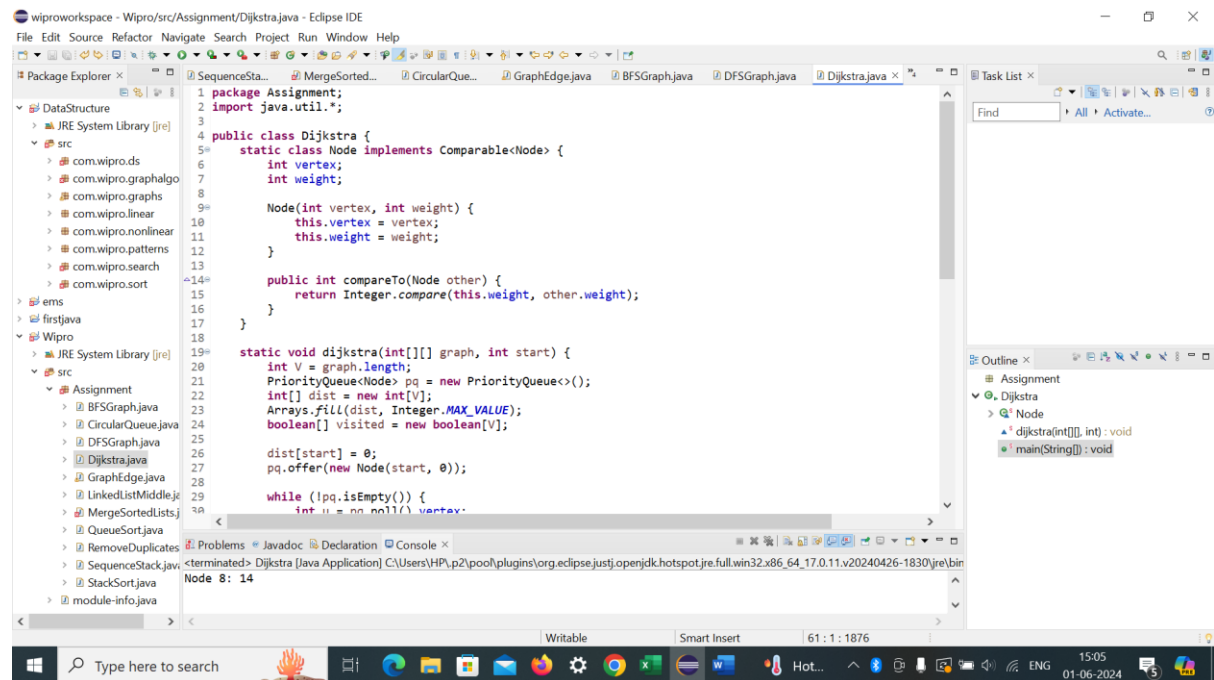


This screenshot shows the same Eclipse IDE environment, but with the `main` method added to the `DFSGraph` class. The `main` method creates a `DFSGraph` object, adds edges (0,3), (0,5), (1,2), and (3,5), and then calls `dfs(0)` to start the search from node 0. The console output remains the same: `DFS starting from node 0: 0 3 5`.

```
13 package Assignment;
14 import java.util.List;
15 import java.util.Map;
16 import java.util.ArrayList;
17 import java.util.HashMap;
18 import java.util.Set;
19 import java.util.HashSet;
20 public class DFSGraph {
21     private final Map<Integer, List<Integer>> adjList = new HashMap<>();
22
23     public void addNode(int node) {
24         adjList.putIfAbsent(node, new ArrayList<>());
25     }
26     public void addEdge(int from, int to) {
27         addNode(from);
28         addNode(to);
29         adjList.get(from).add(to);
30         adjList.get(to).add(from);
31     }
32     public void dfs(int start) {
33         Set<Integer> visited = new HashSet<>();
34         dfsUtil(start, visited);
35     }
36     private void dfsUtil(int node, Set<Integer> visited) {
37         visited.add(node);
38         System.out.print(node + " ");
39         for (int neighbor : adjList.get(node)) {
40             if (!visited.contains(neighbor)) {
41                 dfsUtil(neighbor, visited);
42             }
43         }
44     }
45
46     public static void main(String[] args) {
47         DFSGraph dfsGraph = new DFSGraph();
48         dfsGraph.addEdge(0, 3);
49         dfsGraph.addEdge(0, 5);
50         dfsGraph.addEdge(1, 2);
51         dfsGraph.addEdge(3, 5);
52         System.out.println("DFS starting from node 0:");
53         dfsGraph.dfs(0);
54     }
55 }
```

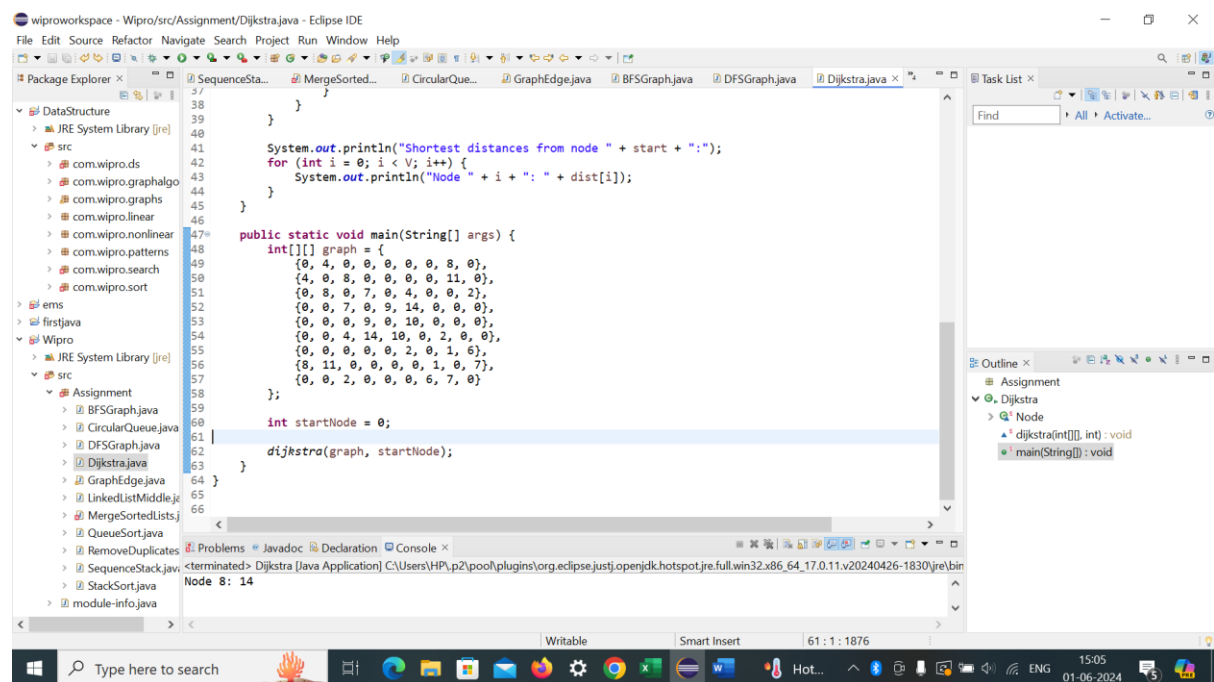
## Task 1: Dijkstra's Shortest Path Finder

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



The screenshot shows the Eclipse IDE with the file `Dijkstra.java` open. The code implements Dijkstra's algorithm. It starts with a package declaration and imports. A `Node` class is defined as a static inner class implementing `Comparable<Node>`. The `dijkstra` method takes a 2D array of graph weights and a start node, initializes a priority queue, and iterates through nodes to find the shortest paths. The console shows the error message: "terminated: Dijkstra [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64.17.0.11.v20240426-1830\jre\bin\node 8: 14".

```
1 package Assignment;
2 import java.util.*;
3
4 public class Dijkstra {
5     static class Node implements Comparable<Node> {
6         int vertex;
7         int weight;
8
9         Node(int vertex, int weight) {
10             this.vertex = vertex;
11             this.weight = weight;
12         }
13
14         public int compareTo(Node other) {
15             return Integer.compare(this.weight, other.weight);
16         }
17     }
18
19     static void dijkstra(int[][] graph, int start) {
20         int V = graph.length;
21         PriorityQueue<Node> pq = new PriorityQueue<>();
22         int[] dist = new int[V];
23         Arrays.fill(dist, Integer.MAX_VALUE);
24         boolean[] visited = new boolean[V];
25
26         dist[start] = 0;
27         pq.offer(new Node(start, 0));
28
29         while (!pq.isEmpty()) {
30             int u = pq.poll().vertex;
31             // ... (rest of the algorithm logic)
32         }
33     }
34 }
```

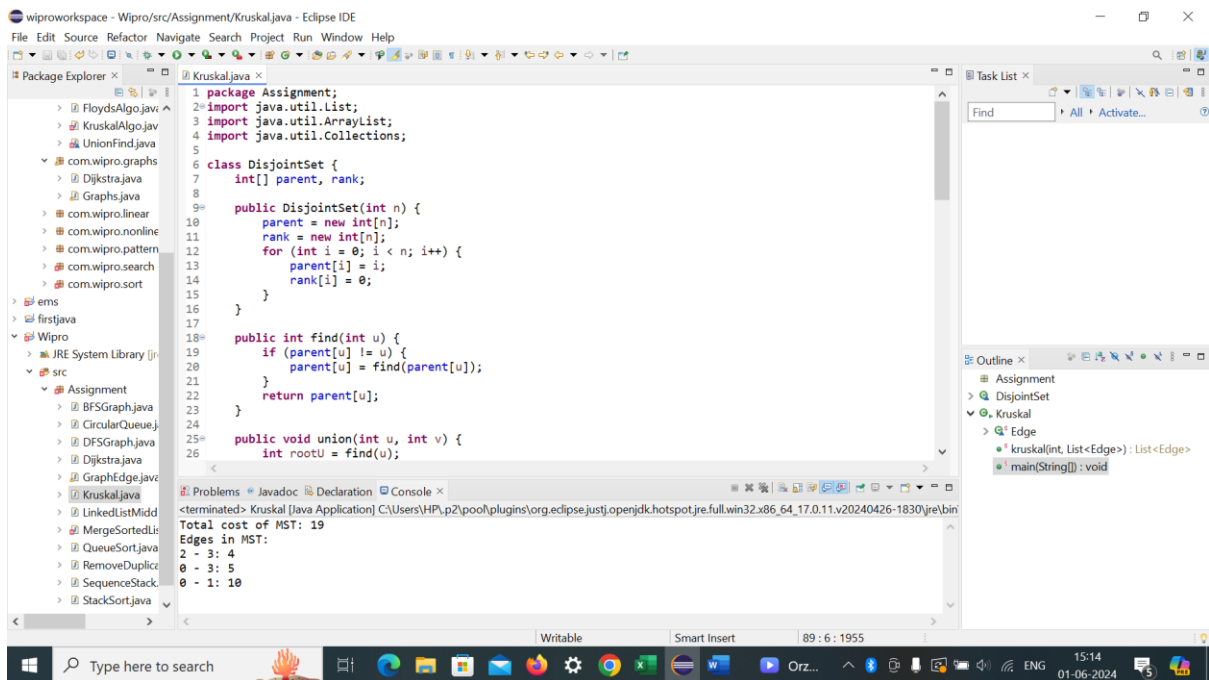


The screenshot shows the Eclipse IDE with the file `Dijkstra.java` open. The code is now complete, including a `main` method that defines a sample graph and calls the `dijkstra` method. The console shows the error message: "terminated: Dijkstra [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64.17.0.11.v20240426-1830\jre\bin\node 8: 14".

```
38 }
39
40 }
41
42 System.out.println("Shortest distances from node " + start + ":");
43 for (int i = 0; i < V; i++) {
44     System.out.println("Node " + i + ": " + dist[i]);
45 }
46
47 public static void main(String[] args) {
48     int[][] graph = {
49         {0, 4, 0, 0, 0, 0, 0, 8, 0},
50         {4, 0, 8, 0, 0, 0, 0, 11, 0},
51         {0, 8, 0, 7, 0, 4, 0, 0, 2},
52         {0, 0, 7, 0, 9, 14, 0, 0, 0},
53         {0, 0, 9, 0, 10, 0, 0, 0, 0},
54         {0, 0, 4, 14, 10, 0, 2, 0, 0},
55         {0, 0, 0, 0, 0, 2, 0, 1, 6},
56         {8, 11, 0, 0, 0, 0, 2, 0, 7},
57         {0, 0, 2, 0, 0, 0, 0, 6, 7, 0}
58     };
59
60     int startNode = 0;
61     dijkstra(graph, startNode);
62 }
63
64 }
65
66 }
```

## 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.



```
1 package Assignment;
2 import java.util.List;
3 import java.util.ArrayList;
4 import java.util.Collections;
5
6 class DisjointSet {
7     int[] parent, rank;
8
9     public DisjointSet(int n) {
10         parent = new int[n];
11         rank = new int[n];
12         for (int i = 0; i < n; i++) {
13             parent[i] = i;
14             rank[i] = 0;
15         }
16     }
17
18     public int find(int u) {
19         if (parent[u] != u) {
20             parent[u] = find(parent[u]);
21         }
22         return parent[u];
23     }
24
25     public void union(int u, int v) {
26         int rootU = find(u);
```

Task List

Find

All Activate...

Outline

- Assignment
- DisjointSet
- Kruskal
  - Edge
    - kruskal(int, List<Edge>): List<Edge>
    - main(String[]): void

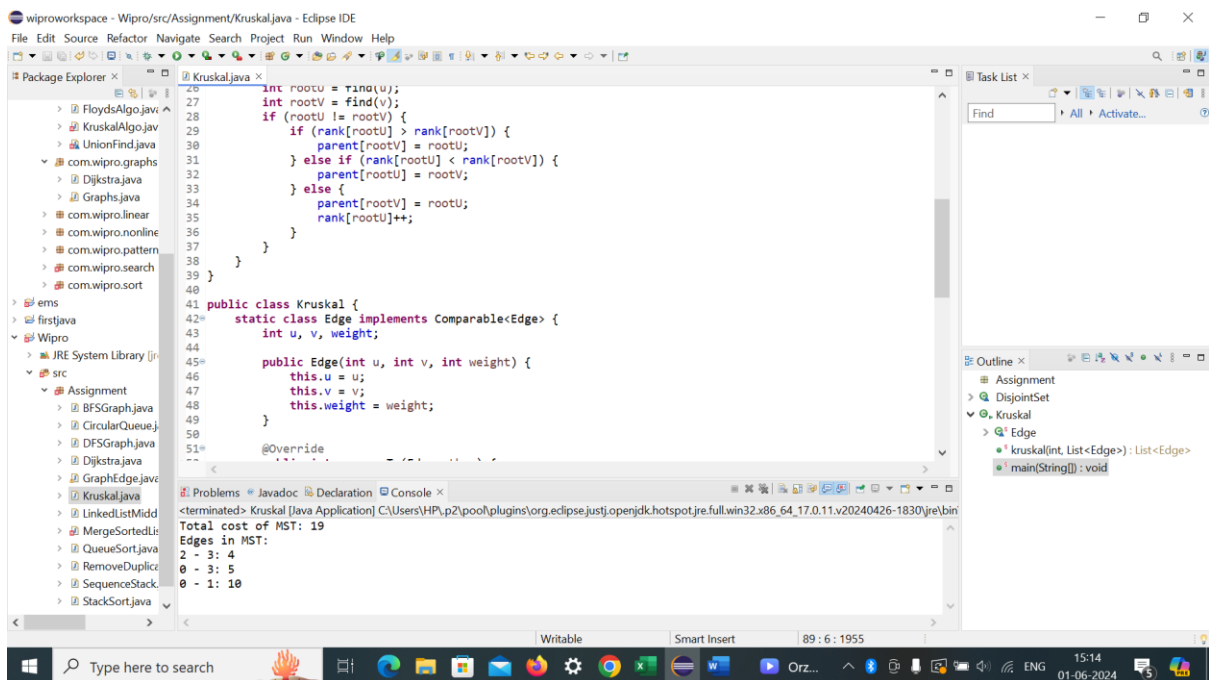
Problems Javadoc Declaration Console

<terminated> Kruskal [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64\_17.0.11.v20240426-1830\jre\bin

Total cost of MST: 19

Edges in MST:

- 2 - 3: 4
- 0 - 3: 5
- 0 - 1: 10



```
26     int rootU = find(u);
27     int rootV = find(v);
28     if (rootU != rootV) {
29         if (rank[rootU] > rank[rootV]) {
30             parent[rootV] = rootU;
31         } else if (rank[rootU] < rank[rootV]) {
32             parent[rootU] = rootV;
33         } else {
34             parent[rootV] = rootU;
35             rank[rootU]++;
36         }
37     }
38 }
39
40 public class Kruskal {
41     static class Edge implements Comparable<Edge> {
42         int u, v, weight;
43
44         public Edge(int u, int v, int weight) {
45             this.u = u;
46             this.v = v;
47             this.weight = weight;
48         }
49
50         @Override
51         public int compareTo(Edge other) {
52             return Integer.compare(this.weight, other.weight);
53         }
54     }
55
56     List<Edge> edges;
57     DisjointSet ds;
58
59     public Kruskal(List<Edge> edges) {
60         this.edges = edges;
61         ds = new DisjointSet(edges.size());
62     }
63
64     public List<Edge> kruskal() {
65         Collections.sort(edges);
66         List<Edge> mst = new ArrayList<>();
67         for (Edge e : edges) {
68             if (ds.find(e.u) != ds.find(e.v)) {
69                 ds.union(e.u, e.v);
70                 mst.add(e);
71             }
72         }
73         return mst;
74     }
75
76     public int totalCost() {
77         int cost = 0;
78         for (Edge e : kruskal()) {
79             cost += e.weight;
80         }
81         return cost;
82     }
83
84     public void printMST() {
85         System.out.println("Edges in MST:");
86         for (Edge e : kruskal()) {
87             System.out.println(e.u + " - " + e.v + ": " + e.weight);
88         }
89     }
90
91     public static void main(String[] args) {
92         // Example usage
93         List<Edge> edges = new ArrayList<>();
94         edges.add(new Edge(2, 3, 4));
95         edges.add(new Edge(0, 3, 5));
96         edges.add(new Edge(0, 1, 10));
97         Kruskal k = new Kruskal(edges);
98         List<Edge> mst = k.kruskal();
99         System.out.println("Total cost of MST: " + k.totalCost());
100        k.printMST();
101    }
102 }
```

Task List

Find

All Activate...

Outline

- Assignment
- DisjointSet
- Kruskal
  - Edge
    - kruskal(int, List<Edge>): List<Edge>
    - main(String[]): void

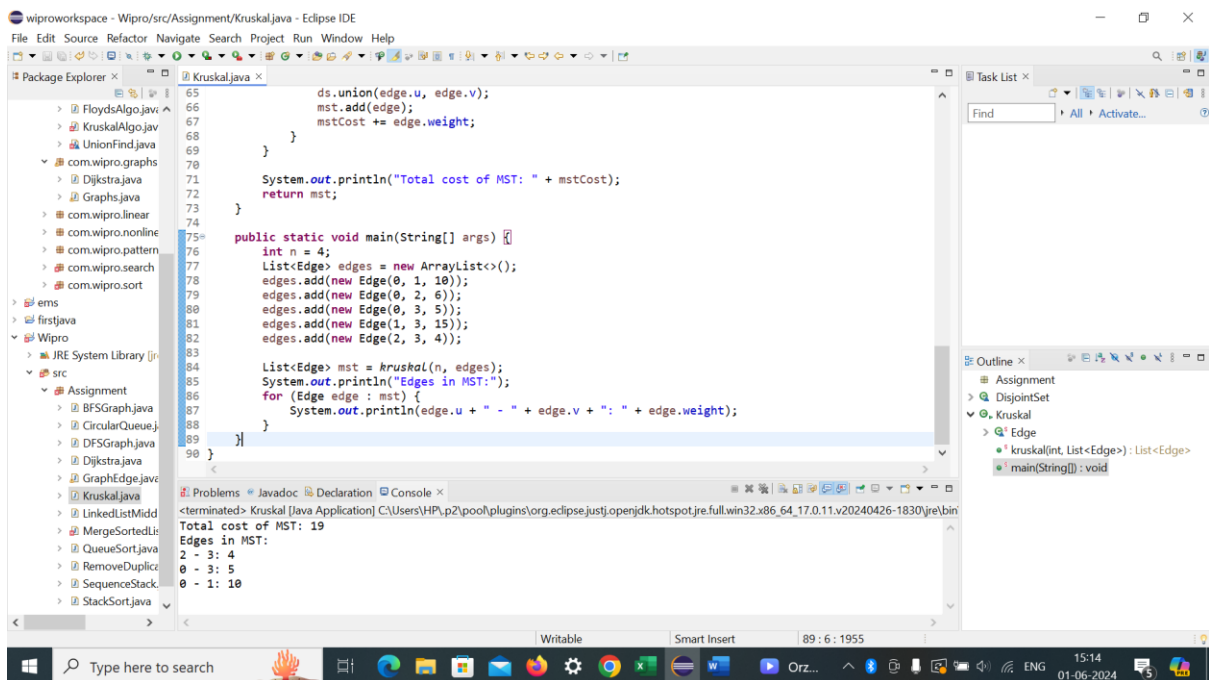
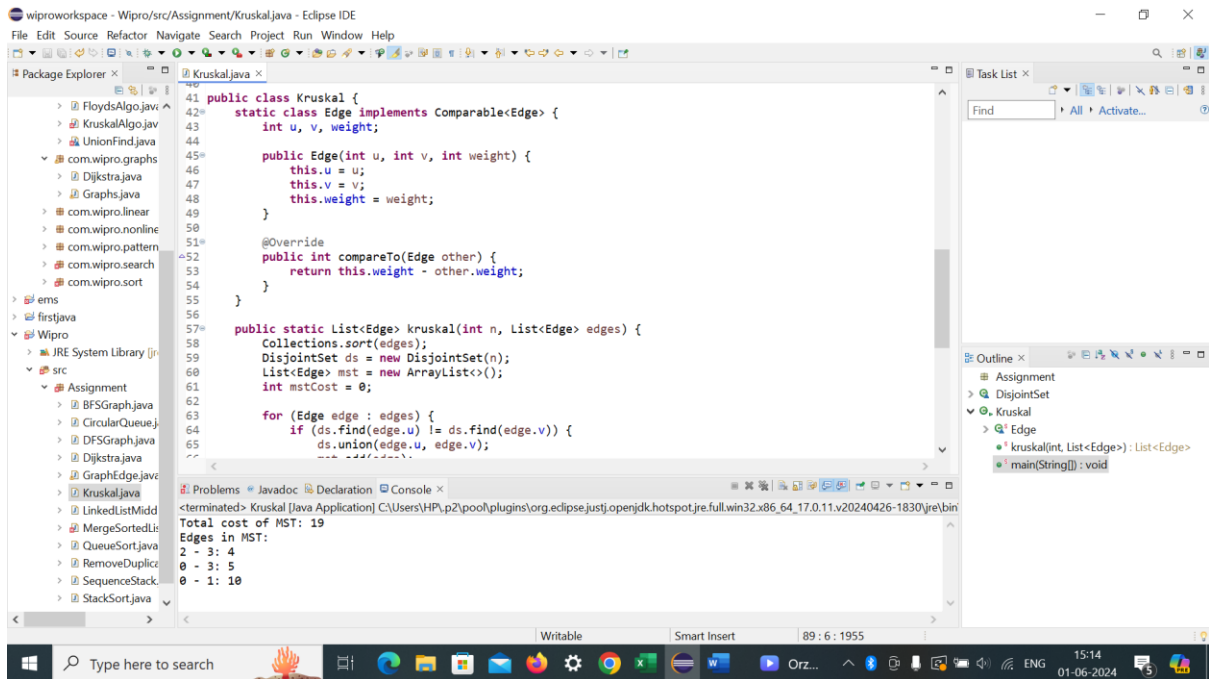
Problems Javadoc Declaration Console

<terminated> Kruskal [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64\_17.0.11.v20240426-1830\jre\bin

Total cost of MST: 19

Edges in MST:

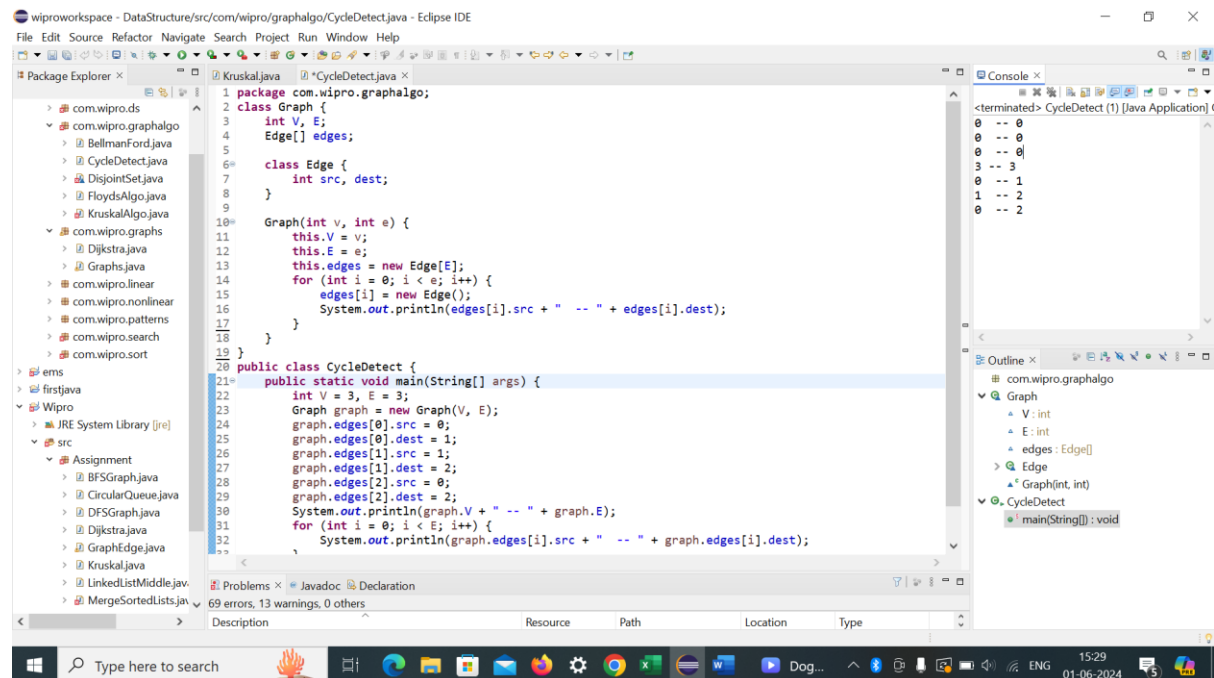
- 2 - 3: 4
- 0 - 3: 5
- 0 - 1: 10





### 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.



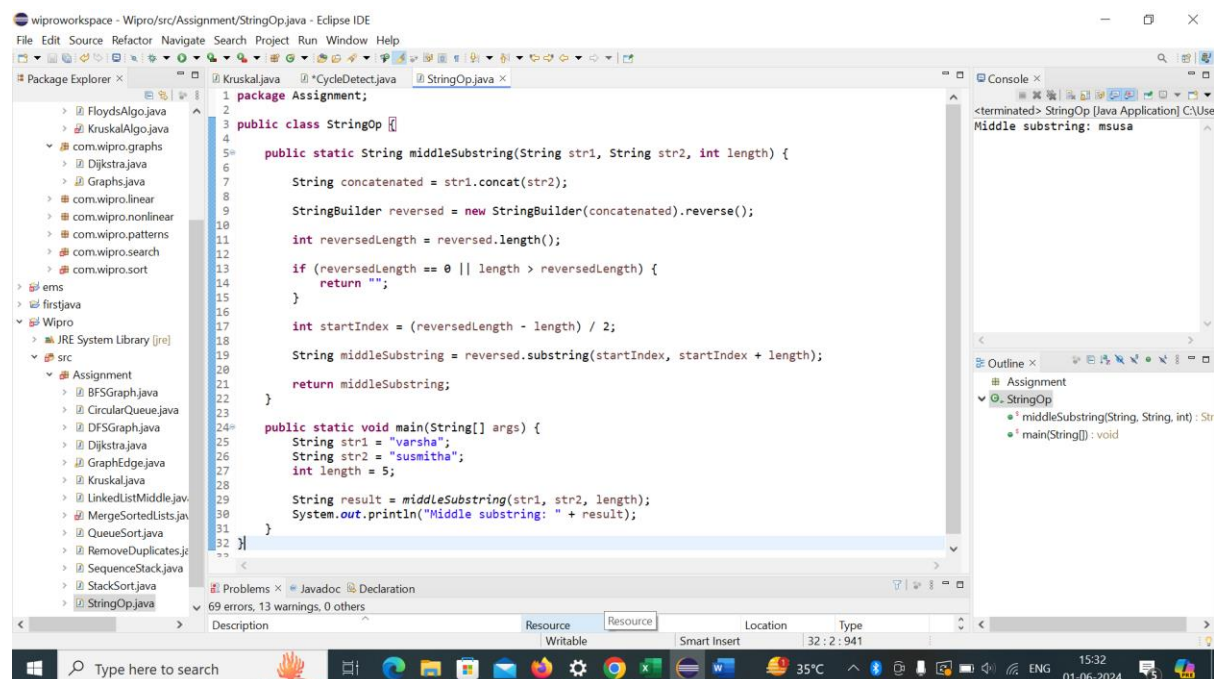
```
1 package com.wipro.graphalgo;
2 class Graph {
3     int V, E;
4     Edge[] edges;
5
6     class Edge {
7         int src, dest;
8     }
9
10    Graph(int v, int e) {
11        this.V = v;
12        this.E = e;
13        this.edges = new Edge[E];
14        for (int i = 0; i < e; i++) {
15            edges[i] = new Edge();
16            System.out.println(edges[i].src + " -- " + edges[i].dest);
17        }
18    }
19
20    public class CycleDetect {
21        public static void main(String[] args) {
22            int V = 3, E = 3;
23            Graph graph = new Graph(V, E);
24            graph.edges[0].src = 0;
25            graph.edges[0].dest = 1;
26            graph.edges[1].src = 1;
27            graph.edges[1].dest = 2;
28            graph.edges[2].src = 2;
29            graph.edges[2].dest = 0;
30            System.out.println(graph.V + " -- " + graph.E);
31            for (int i = 0; i < E; i++) {
32                System.out.println(graph.edges[i].src + " -- " + graph.edges[i].dest);
33            }
34        }
35    }
36}
```

Console Output:

```
<terminated> CycleDetect (1) [Java Application] (
0 -- 0
0 -- 0
0 -- 0
3 -- 3
0 -- 1
1 -- 2
0 -- 2
```

### 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.



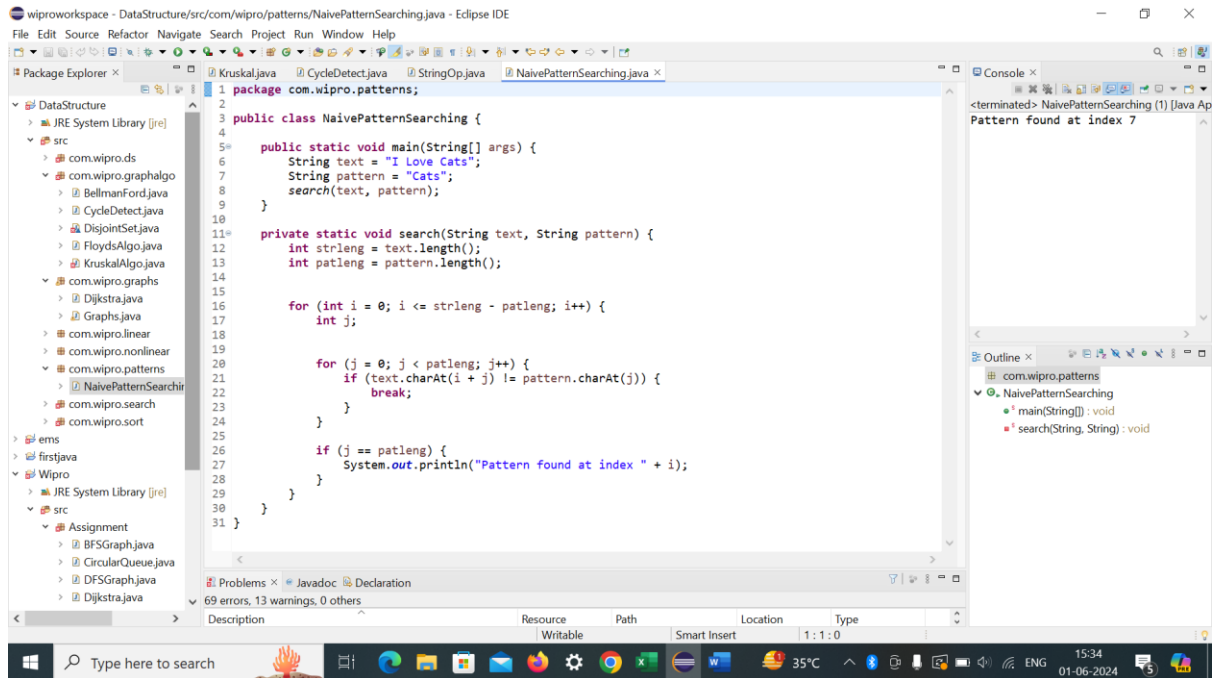
```
1 package Assignment;
2
3 public class StringOp {
4
5     public static String middleSubstring(String str1, String str2, int length) {
6         String concatenated = str1.concat(str2);
7         StringBuilder reversed = new StringBuilder(concatenated).reverse();
8         int reversedLength = reversed.length();
9
10        if (reversedLength == 0 || length > reversedLength) {
11            return "";
12        }
13
14        int startIndex = (reversedLength - length) / 2;
15        String middleSubstring = reversed.substring(startIndex, startIndex + length);
16        return middleSubstring;
17    }
18
19    public static void main(String[] args) {
20        String str1 = "varsha";
21        String str2 = "susmitha";
22        int length = 5;
23
24        String result = middleSubstring(str1, str2, length);
25        System.out.println("Middle substring: " + result);
26    }
27}
```

Console Output:

```
<terminated> StringOp [Java Application] C:\Use
Middle substring: msusa
```

## 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



```
1 package com.wipro.patterns;
2
3 public class NaivePatternSearching {
4
5     public static void main(String[] args) {
6         String text = "I Love Cats";
7         String pattern = "Cats";
8         search(text, pattern);
9     }
10
11     private static void search(String text, String pattern) {
12         int strleng = text.length();
13         int patleng = pattern.length();
14
15         for (int i = 0; i <= strleng - patleng; i++) {
16             int j;
17
18             for (j = 0; j < patleng; j++) {
19                 if (text.charAt(i + j) != pattern.charAt(j)) {
20                     break;
21                 }
22             }
23
24             if (j == patleng) {
25                 System.out.println("Pattern found at index " + i);
26             }
27         }
28     }
29 }
30
31 }
```

Console Output: <terminated> NaivePatternSearching (1) [Java Ap  
Pattern found at index 7

Outline:

- com.wipro.patterns
- NaivePatternSearching
  - main(String[]): void
  - search(String, String): void

Problems: 69 errors, 13 warnings, 0 others