Problem Solving Assignment

Course Scheduling System: Priya is a member from the training department of a company who has to prepare a training plan that has different courses. Few of these courses need some prerequisite courses to be completed. The goal is to prepare a training plan such that all the courses are included in the correct order and the learning is on track.

Problem Identification:
Priya from the training department needs to prepare a training plan for various courses. Some of these courses have prerequisites that need to be completed before others can be taken. The main objective is to create a training plan that lists all the courses in the correct order, ensuring that all prerequisite requirements are met and the learning process is smooth and on track.

Problem Decomposition:

- Identify courses and their prerequisites.
- Represent Course dependencies

Pattern Recognition:

- Graph Representation: The problem can be represented using a directed graph where nodes represent courses and directed edges represent the prerequisite relationships.
- Topological Sorting

Abstraction:
Cycle Detection: Ensure the graph has no cycles, as a cycle would indicate a circular dependency, making it impossible to satisfy prerequisites.

To address Priya's task of preparing a training plan that includes courses in the correct order with respect to their prerequisites, I have used Topological Sorting problem in a Directed Acyclic Graph.

1. Courses and Prerequisites: Each course is a node in a graph.
2. Dependencies: If course u requires course v to be completed first, there is a directed edge from node u to node v ($u \rightarrow v$).

The objective is to order these courses such that for every directed edge ($u \rightarrow v$), course u comes before course v in the order.

Topological Sort

Topological sorting for a graph is a linear ordering of its vertices such that for every directed edge $u \rightarrow v$, vertex u comes before v in the ordering.

Steps:

- **Model the Problem as a Graph**:
  - Represent each course as a node.
  - Represent each prerequisite relationship as a directed edge.
- **Detect Cycles**:

  Since topological sort is only possible for Directed Acyclic Graphs (DAGs), we need to ensure there are no cycles in the graph. Cycle detection can be done using Depth-First Search (DFS).

Algorithm for Topological Sorting using DFS:

Here's a step-by-step algorithm for topological sorting using Depth First Search:

- Create a graph with **n** vertices and **m**-directed edges.
- Initialize a stack and a visited array of size **n**.
- For each unvisited vertex in the graph, do the following:
  - Call the DFS function with the vertex as the parameter.
  - In the DFS function, mark the vertex as visited and recursively call the DFS function for all unvisited neighbors of the vertex.
  - Once all the neighbors have been visited, push the vertex onto the stack.
- After all, vertices have been visited, pop elements from the stack and append them to the output list until the stack is empty.
- The resulting list is the topologically sorted order of the graph.

Code:

```
import java.util.*;



public class TopologicalSort {

    private int vertices; // Number of vertices

    private LinkedList<Integer>[] adjList; // Adjacency list

    private int[] inDegree; // Array to store in-degrees of vertices
```

```java
// Constructor

TopologicalSort(int v) {

    vertices = v;

    adjList = new LinkedList[v];

    inDegree = new int[v];

    for (int i = 0; i < v; ++i) {

        adjList[i] = new LinkedList<>();

        inDegree[i] = 0;

    }

}



// Function to add an edge into the graph

void addEdge(int v, int w) {

    adjList[v].add(w);

    inDegree[w]++;

}



// Function to print all topological sorts

void allTopologicalSorts() {
```

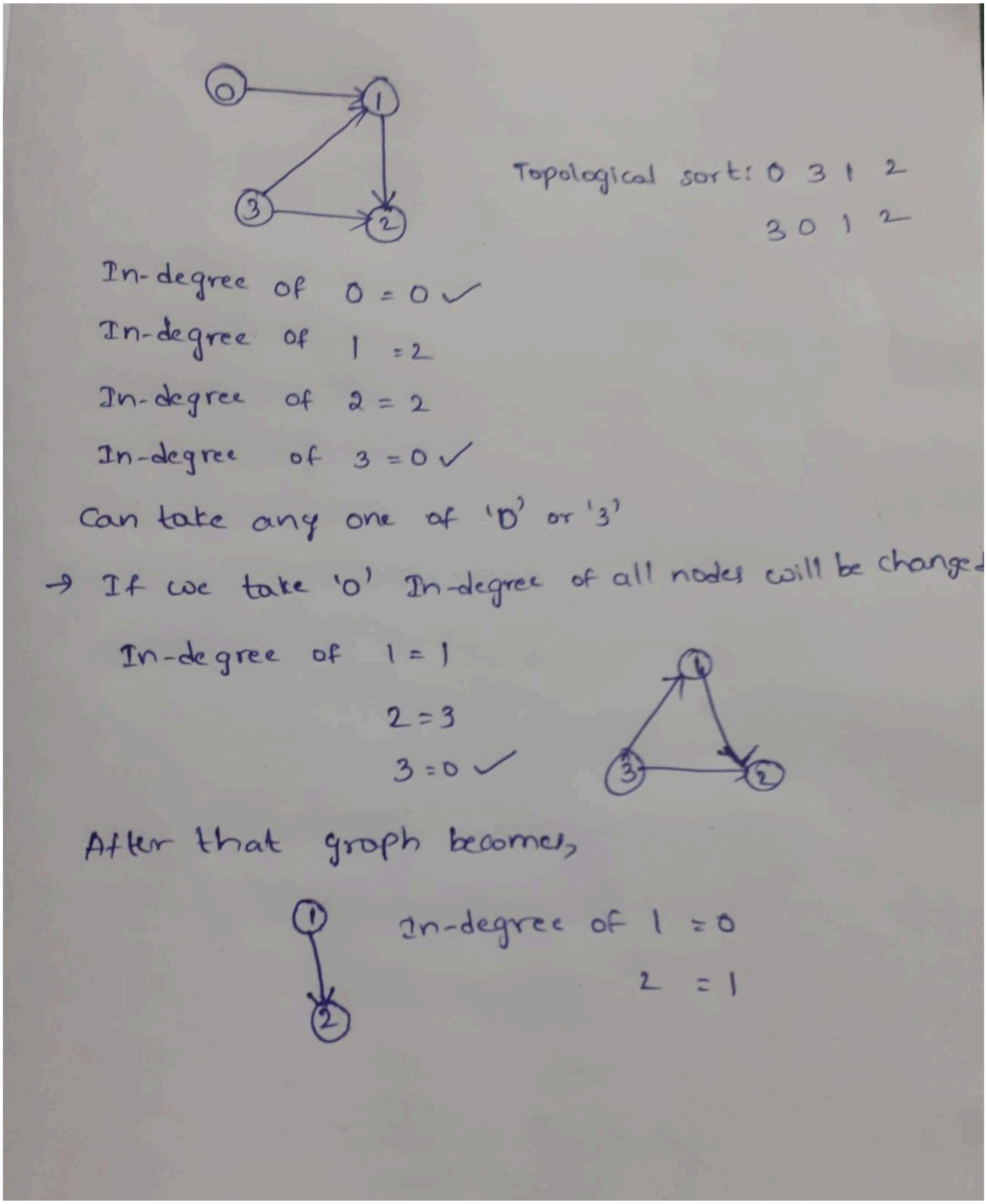
```
        boolean[] visited = new boolean[vertices];

        LinkedList<Integer> stack = new LinkedList<>();



        allTopologicalSortsUtil(visited, stack);

    }



    // Recursive utility function for topological sort

    private void allTopologicalSortsUtil(boolean[] visited,
LinkedList<Integer> stack) {

        boolean flag = false;



        for (int i = 0; i < vertices; i++) {

            if (!visited[i] && inDegree[i] == 0) {

                for (int adj : adjList[i]) {

                    inDegree[adj]--;

                }



                stack.add(i);

                visited[i] = true;

                allTopologicalSortsUtil(visited, stack);
```

```java
                visited[i] = false;

                stack.removeLast();

                for (int adj : adjList[i]) {

                    inDegree[adj]++;

                }



                flag = true;

            }

        }



        if (!flag) {

            stack.forEach(v -> System.out.print(v + " "));

            System.out.println();

        }

    }



    // Function to print all incoming edges of nodes

    void printIncomingEdges() {
```

```java
    for (int i = 0; i < vertices; i++) {

        System.out.print("Node " + i + ": ");

        for (int j = 0; j < vertices; j++) {

            if (adjList[j].contains(i)) {

                System.out.print(j + " ");

            }

        }

        System.out.println();

    }

}



public static void main(String args[]) {

    Scanner scanner = new Scanner(System.in);



    System.out.println("Enter the number of vertices:");

    int V = scanner.nextInt();



    System.out.println("Enter the number of edges:");

    int E = scanner.nextInt();
```

```java
        TopologicalSort graph = new TopologicalSort(V);


        System.out.println("Enter the edges (source destination):");
        for (int i = 0; i < E; i++) {
            int v = scanner.nextInt();
            int w = scanner.nextInt();
            graph.addEdge(v, w);
        }


        System.out.println("All Topological Sorts:");
        graph.allTopologicalSorts();


        System.out.println("Incoming Edges of Nodes:");
        graph.printIncomingEdges();


        scanner.close();
    }
}
```

Example:



Time Complexity: O(V+E)

Where, V represents vertices

E represents Edges

Space Complexity:O(V)