Day 23 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

Task 1: Tower of Hanoi Solver

Create a program that solves the Tower of Hanoi puzzle for n disks. The solution should use recursion to move disks between three pegs (source, auxiliary, and destination) according to the game's rules. The program should print out each move required to solve the puzzle.

```
package algorithms;

public class TowerOfHanoi {

      public static void solveHanoi(int n, char source, char 
auxiliary, char destination) {
                      if (n == 1) {
                           System.out.println("Move disk 1 from " + 
source + " to " + destination);
                           return;
                      }

                      solveHanoi(n - 1, source, destination, 
auxiliary);
                      System.out.println("Move disk " + n + " from " 
+ source + " to " + destination);
                      solveHanoi(n - 1, auxiliary, source, 
destination);
           }

      public static void main(String[] args) {
                int numDisks = 3;
                System.out.println("Total Number of Disks: " + 
numDisks);
                solveHanoi(numDisks, 'L', 'M', 'N');
      }
}
```

Output:

```
Total Number of Disks: 3
Move disk 1 from L to N
Move disk 2 from L to M
Move disk 1 from N to M
Move disk 3 from L to N
Move disk 1 from M to L
Move disk 2 from M to N
Move disk 1 from L to N
```

Task 2: Traveling Salesman Problem

Create a function int FindMinCost(int[,] graph) that takes a 2D array representing the graph where graph[i][j] is the cost to travel from city i to city j. The function should return the minimum cost to visit all cities and return to the starting city. Use dynamic programming for this solution.

```java
package algorithms;

import java.util.Arrays;

public class TravellingSalesmanProblem {

           private static final int INF = Integer.MAX_VALUE;

           public static int findMinCost(int[][] graph) {
                int n = graph.length;
                int[][] dp = new int[n][(1 << n)];

                for (int[] row : dp) {
                           Arrays.fill(row, -1);
                }
                return tsp(0, 1, graph, dp);
           }

           private static int tsp(int currentPos, int visited,
int[][] graph, int[][] dp)
{
                int n = graph.length;

                if (visited == (1 << n) - 1) {
                      return graph[currentPos][0] == 0 ? INF :
graph[currentPos][0];
                }

                if (dp[currentPos][visited] != -1) {
                           return dp[currentPos][visited];
                }

                int minCost = INF;

                for (int city = 0; city < n; city++) {
                           if ((visited & (1 << city)) == 0 &&
graph[currentPos][city] != 0)
{
                                      int newCost =
graph[currentPos][city] + tsp(city, visited | (1 << city), graph,
dp);
                           minCost = Math.min(minCost, newCost);
```

```
                    }
                }

                dp[currentPos][visited] = minCost;
                return dp[currentPos][visited];
            }

            public static void main(String[] args) {
            int[][] graph = { { 0, 10, 15, 20 }, { 10, 0, 35, 25 },
{ 15, 35, 0, 30},{ 20, 25, 30, 0 } };

                System.out.println("Given Graph:");
                for (int i = 0; i < graph.length; i++) {
                    System.out.println(Arrays.toString(graph[i]));
                }

                int result = findMinCost(graph);
                System.out.println("The Minimum Cost to visit all
Cities and return to " + "the Starting City is: " + result);
            }
}
```

Output:

```
Given Graph:
[0, 10, 15, 20]
[10, 0, 35, 25]
[15, 35, 0, 30]
[20, 25, 30, 0]
The Minimum Cost to visit all Cities and return to the Starting City
is: 80
```

Task 3: Job Sequencing Problem

Define a class Job with properties int Id, int Deadline, and int Profit. Then implement a function List<Job> JobSequencing(List<Job> jobs) that takes a list of jobs and returns the maximum profit sequence of jobs that can be done before the deadlines. Use the greedy method to solve this problem.

```
package algorithms;
import java.util.ArrayList;

import java.util.Collections;

import java.util.List;


public class Job {

    int Id;
    int Deadline;
    int Profit;

    public Job(int id, int deadline, int profit) {
            Id = id;
            Deadline = deadline;
            Profit = profit;
    }
}

public class JobSequencing {
    public static List<Job> scheduleJobs(List<Job> jobs) {
            Collections.sort(jobs, (job1, job2) -> job2.Profit -
job1.Profit); //
//Sort by decreasing profit

            List<Job> scheduledJobs = new ArrayList<>();
            int[] deadlines = new int[jobs.size() + 1];

            for (Job job : jobs) {
            // Find the latest slot available before deadline
                    int slot = job.Deadline;
                    while (slot > 0 && deadlines[slot] > 0) {
                            slot--;
                    }

                    if (slot > 0) {
                            deadlines[slot] = job.Id;
                            scheduledJobs.add(job);
                    }
            }
            return scheduledJobs;
    }
```

```java
    public static void main(String[] args) {
        List<Job> jobs = new ArrayList<>();
        jobs.add(new Job(1, 2, 50));
        jobs.add(new Job(2, 1, 100));
        jobs.add(new Job(3, 2, 30));
        jobs.add(new Job(4, 1, 20));

        List<Job> scheduledJobs = scheduleJobs(jobs);
        System.out.println("Scheduled Jobs: ");
        for (Job job : scheduledJobs) {
                System.out.println("Job Id: " + job.Id + ", Profit: " + job.Profit);
        }
    }
}
```

Output:

```
Scheduled Jobs:
Job Id: 2, Profit: 100
Job Id: 1, Profit: 50
```