NAGELLA PRAJITHRAJ

Day 13 and 14:

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.

Java implementation of the Tower of Hanoi puzzle using recursion:

```
public class TowerOfHanoi {

    public static void main(String[] args) {
        int n = 3;
        towerOfHanoi(n, 'A', 'B', 'C');
    }

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

Explanation:

Class and main Method:

- TowerOfHanoi class is defined with a main method where the Tower of Hanoi puzzle is solved for n = 3 disks.

Recursive Method towerOfHanoi:

- This method takes four parameters:
- n: Number of disks to be moved.
- source: The peg from which the disks are initially stacked.
- auxiliary: An auxiliary peg used for moving disks.
- destination: The peg onto which the disks have to be moved.

Base Case:

When n == 1, the method prints the move to transfer the disk directly from source to destination.

Recursive Case:

- For n > 1, the method recursively calls itself:
- Moves n-1 disks from source to auxiliary, using destination as the auxiliary peg.
- Moves the n-th disk from source to destination.
- Moves the n-1 disks from auxiliary to destination, using source as the auxiliary peg.

Output:

```
Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C
```

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.

Given a graph represented by a 2D array graph where graph[i][j] represents the cost to travel from city i to city j, we need to find the minimum cost to visit all cities exactly once and return to the starting city.

Approach:

Dynamic Programming Definition:

- Let's define dp[mask][i] where mask is a bitmask representing the set of cities visited so far, and i is the current city.
- dp[mask][i] will represent the minimum cost to visit all cities in the set mask and end at city i.

Initialization:

Start with the initial city as the starting point. Initialize dp[1 << start][start] = 0 where start is the index of the starting city.

Transition:

For each set of visited cities (mask), and for each city i that is not yet visited: **dp[mask**

| (1 << i)][i] = min(dp[mask | (1 << i)][i], dp[mask][j] + graph[j][i])

Here, j is any city already visited (j is in mask).

Final Calculation:

The answer will be min(dp[(1 << n) - 1][j] + graph[j][start]) where n is the number of cities, and start is the starting city.

Implementation:

We need to iterate over all possible subsets of cities and compute the minimum cost using the above transitions.

```
public class MinimumCostTravelAllCities {

   public static int findMinCost(int[][] graph) {
int n = graph.length;

       // dp[mask][i] will store the minimum cost to visit all cities in 'mask' ending at city 'i'
int[][] dp = new int[1 << n][n];

       // Initialize dp array with a large value (infinity)
for (int[] row : dp) {
          Arrays.fill(row, Integer.MAX_VALUE / 2);
       }

       // Start from city 0
int start = 0;

       // Base case: Starting from city 0, cost is 0
dp[1 << start][start] = 0;

       // Iterate over all subsets of cities          for
(int mask = 1; mask < (1 << n); mask++) {
          for (int i = 0; i < n; i++) {
             if ((mask & (1 << i)) != 0) { // i is in the current subset mask
                for (int j = 0; j < n; j++) {
                    if (j != i && (mask & (1 << j)) != 0) { // j is in the current subset mask
dp[mask][i] = Math.min(dp[mask][i], dp[mask ^ (1 << i)][j] + graph[j][i]);
                    }
                  }
                }
              }
            }

            // Find the minimum cost to visit all cities and return to starting city
int minCost = Integer.MAX_VALUE;
            int fullMask = (1 << n) - 1;
for (int j = 0; j < n; j++) {
if (j != start) {
                minCost = Math.min(minCost, dp[fullMask][j] + graph[j][start]);
              }
            }

            return minCost;
          }

          public static void main(String[] args) {
            int[][] graph = {
              {0, 10, 15, 20},
              {5, 0, 9, 10},
              {6, 13, 0, 12},
              {8, 8, 9, 0}
            };

            int minCost = findMinCost(graph);
```

```
        System.out.println("Minimum cost to visit all cities and return to starting city: " +
minCost); // Output: 35

    }

}
```

Output Explanation:

```
{0, 10, 15, 20},

{5, 0, 9, 10},

{6, 13, 0, 12},

{8, 8, 9, 0}

Minimum cost to visit all cities and return to starting city: 42
```

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.

```
import java.util.ArrayList; import

java.util.Collections; import

java.util.Comparator;

import java.util.List;


// Job class to store job details class

Job {

    int id;

    int deadline;

int profit;
```

```
   Job(int id, int deadline, int profit) {
       this.id = id;
       this.deadline = deadline;
this.profit = profit;
  }
}

public class JobSequencing {

   // Function to find the maximum profit job
sequence    public List<Job> jobSequencing(List<Job>
jobs) {       // Sort jobs by profit in descending order
       Collections.sort(jobs, new Comparator<Job>() {
           @Override           public int
compare(Job job1, Job job2) {              return
job2.profit - job1.profit;
           }
       });

       // Find the maximum deadline to determine the size of the job sequence array
int maxDeadline = jobs.stream().mapToInt(job -> job.deadline).max().orElse(0);
int[] sequence = new int[maxDeadline]; // This will store the sequence of jobs

       // Initialize the sequence array        for (int i =
0; i < maxDeadline; i++) {           sequence[i] = -1;
// -1 means slot is empty
       }
```

```
        // Fill the sequence array
int totalProfit = 0;        for
(Job job : jobs) {
            // Find a slot for this job, from its deadline to 0
for (int j = job.deadline - 1; j >= 0; j--) {                if
(sequence[j] == -1) {

                    sequence[j] = job.id;
totalProfit += job.profit;
                    break;
                }
            }
        }

        // Create the list of jobs in the sequence
List<Job> result = new ArrayList<>();
for (int i = 0; i < maxDeadline; i++) {            if
(sequence[i] != -1) {
                Job job = jobs.stream().filter(j -> j.id == sequence[i]).findFirst().orElse(null);
                if (job != null) {
result.add(job);
                }
            }
        }

        return result;
    }

    public static void main(String[] args) {
// Example usage:
    List<Job> jobs = new ArrayList<>();
jobs.add(new Job(1, 4, 20));        jobs.add(new
Job(2, 1, 10));        jobs.add(new Job(3, 1, 40));
jobs.add(new Job(4, 1, 30));
JobSequencing solution = new
JobSequencing();

    List<Job> result = solution.jobSequencing(jobs);


    // Output the jobs in the sequence
System.out.println("Job Sequence:");
    for (Job job : result) {
      System.out.println("Job Id: " + job.id + ", Deadline: " + job.deadline + ", Profit: " +
job.profit);
    }
  }
}
```

Explanation:

- **Job Class:** Represents a job with properties **id, deadline, and profit.**
- **JobSequencing Class:**
- jobSequencing Method:
- Sorts the jobs by profit in descending order using Collections.sort.
- Initializes an array sequence to keep track of the jobs scheduled.
- Iterates through the sorted jobs and finds a slot in the sequence array for each job, from its deadline to 0.
- Calculates the total profit and creates a list result of jobs in the sequence.

Main Method:

- Example usage of the jobSequencing method with a list of jobs.
- Prints out the jobs in the sequence.
- Example:

For the input jobs:

Job 1: Id=1, Deadline=4, Profit=20

Job 2: Id=2, Deadline=1, Profit=10

Job 3: Id=3, Deadline=1, Profit=40 Job

4: Id=4, Deadline=1, Profit=30 The

output will be:

Job Sequence:

Job Id: 3, Deadline: 1, Profit: 40

Job Id: 1, Deadline: 4, Profit: 20