Day 13_14 Assignment Core Java
Digvijay Thakare(digvijaythakare2017@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.

Code-

```java
package com.epwipro.day13_14;

import java.util.Scanner;

public class TowerOfHanoi {

    // Recursive function to solve the Tower of Hanoi puzzle
    public static void solveHanoi(int n, char source, char auxiliary, char destination) {
        // Base case: if there's only one disk, move it from source to destination
        if (n == 1) {
            System.out.println("Move disk 1 from " + source + " to " + destination);
            return;
        }

        // Move n-1 disks from source to auxiliary using destination as a temporary peg
        solveHanoi(n - 1, source, destination, auxiliary);

        // Move the nth disk from source to destination
        System.out.println("Move disk " + n + " from " + source + " to " + destination);

        // Move the n-1 disks from auxiliary to destination using source as a temporary peg
        solveHanoi(n - 1, auxiliary, source, destination);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of disks: ");
        int n = scanner.nextInt();
        solveHanoi(n, 'A', 'B', 'C'); // A, B, and C are names of the pegs
        scanner.close();
    }
}
```

Output-

```
Enter the number of disks: 5
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
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 5 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
Move disk 3 from B to A
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 4 from B to C
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.

Code-

```java
package com.epwipro.day13_14;

import java.util.Arrays;

public class TravelingSalesman {

    // Function to find the minimum cost to visit all cities and return to the starting city
    public static int FindMinCost(int[][] graph) {
        int n = graph.length;
        int VISITED_ALL = (1 << n) - 1;
        int[][] dp = new int[n][(1 << n)];

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

        // Start the TSP from the first city (index 0)
        return tsp(graph, 0, 1, dp, VISITED_ALL);
    }
    private static int tsp(int[][] graph, int pos, int mask, int[][] dp, int VISITED_ALL) {
        int n = graph.length;

        // Base case: if all cities have been visited, return to the starting city
        if (mask == VISITED_ALL) {
            return graph[pos][0];
        }

        // If the result is already computed, return it
        if (dp[pos][mask] != Integer.MAX_VALUE) {
            return dp[pos][mask];
        }

        // Try to go to every other city to find the minimum cost
        for (int city = 0; city < n; city++) {
            // Check if the city has been visited
            if ((mask & (1 << city)) == 0) {
                // Calculate the cost to visit the next city and update dp array
                int newCost = graph[pos][city] + tsp(graph, city, mask | (1 << city), dp,
VISITED_ALL);
                dp[pos][mask] = Math.min(dp[pos][mask], newCost);
            }
        }

        return dp[pos][mask];
    }

    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("The minimum cost to visit all cities and return to the starting city
is: " + FindMinCost(graph));
    }
}
```

Output-

```
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 JobSequencing(List 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

Code-

```java
package com.epwipro.day13_14;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Job {
    int Id;
    int Deadline;
    int Profit;

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

    @Override
    public String toString() {
        return "Job Id: " + Id + ", Deadline: " + Deadline + ", Profit: " + Profit;
    }
}

public class JobSequencingProblem {

    public static List<Job> JobSequencing(List<Job> jobs) {
        // Sort the jobs by profit in descending order
        Collections.sort(jobs, (a, b) -> b.Profit - a.Profit);

        // Find the maximum deadline to create the schedule array
        int maxDeadline = 0;
        for (Job job : jobs) {
            if (job.Deadline > maxDeadline) {
                maxDeadline = job.Deadline;
            }
        }

        // Create a slot array to keep track of free time slots
        Job[] result = new Job[maxDeadline];
        boolean[] slot = new boolean[maxDeadline];

        // Iterate through the sorted jobs and assign them to the latest possible slot
        for (Job job : jobs) {
            // Find a free slot for this job (starting from the last possible slot)
            for (int j = job.Deadline - 1; j >= 0; j--) {
                if (!slot[j]) {
                    slot[j] = true;
                    result[j] = job;
                    break;
                }
            }
        }

        // Collect the jobs that were scheduled
        List<Job> scheduledJobs = new ArrayList<>();
        for (Job job : result) {
            if (job != null) {
                scheduledJobs.add(job);
            }
        }

        return scheduledJobs;
    }

    public static void main(String[] args) {
        List<Job> jobs = new ArrayList<>();
        jobs.add(new Job(1, 2, 100));
        jobs.add(new Job(2, 1, 19));
        jobs.add(new Job(3, 2, 27));
        jobs.add(new Job(4, 1, 25));
        jobs.add(new Job(5, 3, 15));

        List<Job> jobSequence = JobSequencing(jobs);
        System.out.println("The maximum profit sequence of jobs is:");
        for (Job job : jobSequence) {
            System.out.println(job);
        }
    }
}
```

Output-

```
The maximum profit sequence of jobs is:
Job Id: 3, Deadline: 2, Profit: 27
Job Id: 1, Deadline: 2, Profit: 100
Job Id: 5, Deadline: 3, Profit: 15
```