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 computalgo;



public class TowerofHanoi {


    public static void main(String[] args) {
        hanoi(3,"A","B","C");


    }


    private static void hanoi(int n, String rodFrom,
String rodMiddle, String rodTo) {


        if(n==1) {
            System.out.println("Disk 1 moved from " +
rodFrom +" to " + rodTo);
            return;
        }
        hanoi(n-1,rodFrom, rodTo, rodMiddle);

        System.out.println("Disk " + n + " moved from " + rodFrom + " to " +rodTo);
        hanoi(n-1,rodMiddle, rodFrom, rodTo);
    }

}
```

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.

```
package com.wipro.algo;

import java.util.Arrays;

public class TravelingSalesman {

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

```java
        int VISITED_ALL = (1 << n) - 1;
        int[][] dp = new int[n][1 << n];

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

        return tsp(0, 1, graph, dp, VISITED_ALL);
    }

    private static int tsp(int current, int visited,
int[][] graph, int[][] dp, int VISITED_ALL) {
        if (visited == VISITED_ALL) {
            return graph[current][0];
        }

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

        int minCost = Integer.MAX_VALUE;
        for (int city = 0; city < graph.length; city++)
{
            if ((visited & (1 << city)) == 0) {
                int newVisited = visited | (1 << city);
                int cost = graph[current][city] +
tsp(city, newVisited, graph, dp, VISITED_ALL);
                minCost = Math.min(minCost, cost);
            }
        }

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

    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));
    }
}
```

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 computalgo;

import java.util.ArrayList;
import java.util.Arrays;

class Job {
    char id;
    int deadline;
    int profit;

    public Job(char id, int deadline, int profit) {
        this.id = id;
        this.deadline = deadline;

        this.profit = profit;

    }

}


public class JobSequence {


    public static void main(String[] args) {

        ArrayList<Job> jobs = new ArrayList<>();

        jobs.add(new Job('1', 3, 35));

        jobs.add(new Job('2', 4, 30));

        jobs.add(new Job('3', 4, 25));

        jobs.add(new Job('4', 2, 20));

        jobs.add(new Job('5', 3, 15));

        jobs.add(new Job('6', 1, 12));


        doJobSequence(jobs);

    }


    private static void doJobSequence(ArrayList<Job> jobs) {

        jobs.sort((a, b) -> b.profit - a.profit);


        int maxDeadline = jobs.stream().mapToInt(job -> job.deadline).max().orElse(0);


        boolean[] filledSlots = new boolean[maxDeadline];
```

```
char[] result = new char[maxDeadline];

Arrays.fill(result, '-');

for (Job job : jobs) {
    for (int i = job.deadline - 1; i >= 0; i--) {
        if (!filledSlots[i]) {
            filledSlots[i] = true;
            result[i] = job.id;
            break;
        }
    }
}

int totalProfit = 0;
System.out.println("Job sequence:");
for (int i = 0; i < maxDeadline; i++) {
    if (result[i] != '-') {
        for (Job job : jobs) {
            if (job.id == result[i]) {
                totalProfit += job.profit;
                break;
            }
        }
    }
```

```
            System.out.print(result[i] + " ");

        }


        System.out.println("\nTotal profit after sequencing: " + totalProfit);

    }

}
```