Data Structure and Operations: Day-12 (Tower of Hanoi Solver, Traveling Salesman Problem, Job Sequencing Problem):

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 wipro.com.assignment10;
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 only one disk, move it from source to destination
        if (n == 1) {
            System.out.println("Move disk 1 from " + source + " to " + destination);
            return;
        }

        // Move the top 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) {
        int n = 3;  // Number of disks
        solveHanoi(n, 'A', 'B', 'C');  // A, B, and C are names of the pegs
    }
}
```

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.

```
package wipro.com.assignment10;

public class TravelingSalesman {
    private static final int INF = Integer.MAX_VALUE / 2;

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

        // Initialize dp array with infinity
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < (1 << n); j++) {
                dp[i][j] = INF;
            }
        }

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

        // Iterate over all subsets of cities
        for (int mask = 1; mask < (1 << n); mask++) {
            for (int u = 0; u < n; u++) {
                if ((mask & (1 << u)) != 0) {  // if city 'u' is in the subset
represented by 'mask'
                    for (int v = 0; v < n; v++) {
                        if ((mask & (1 << v)) == 0) {  // if city 'v' is not in the
subset represented by 'mask'
                            dp[v][mask | (1 << v)] = Math.min(dp[v][mask | (1 << v)],
dp[u][mask] + graph[u][v]);
                        }
                    }
                }
            }
        }

        // Find the minimum cost to return to the starting city
        int minCost = INF;
        for (int i = 1; i < n; i++) {
            minCost = Math.min(minCost, dp[i][(1 << n) - 1] + graph[i][0]);
        }
```

```
        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}
        };

        int minCost = FindMinCost(graph);
        System.out.println("The minimum cost to visit all cities and return to the
starting city is: " + minCost);
    }
}
```

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<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 wipro.com.assignment10;
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
class Job {
    int Id;
    int Deadline;
    int Profit;

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

    // toString method for easy printing
    @Override
    public String toString() {
        return "Job[Id=" + Id + ", Deadline=" + Deadline + ", Profit=" + Profit +
"]";
    }
}
```

```java
public class JobSequencingProblem {
    // Function to find the maximum profit sequence of jobs
    public static List<Job> JobSequencing(List<Job> jobs) {
        // Sort jobs by decreasing order of profit
        Collections.sort(jobs, (a, b) -> b.Profit - a.Profit);

        int n = jobs.size();
        // Find the maximum deadline
        int maxDeadline = 0;
        for (Job job : jobs) {
            if (job.Deadline > maxDeadline) {
                maxDeadline = job.Deadline;
            }
        }

        // Initialize a result array to store the result sequence
        Job[] result = new Job[maxDeadline];
        boolean[] slot = new boolean[maxDeadline];

        // Iterate through all given jobs
        for (Job job : jobs) {
            // Find a free slot for this job (we start from the last possible slot)
            for (int j = Math.min(maxDeadline - 1, job.Deadline - 1); j >= 0; j--) {
                // Free slot found
                if (!slot[j]) {
                    slot[j] = true;
                    result[j] = job;
                    break;
                }
            }
        }

        // Prepare the list of jobs that are included in the result
        List<Job> jobSequence = new ArrayList<>();
        for (Job job : result) {
            if (job != null) {
                jobSequence.add(job);
            }
        }

        return jobSequence;
    }

    public static void main(String[] args) {
        List<Job> jobs = Arrays.asList(
            new Job(1, 2, 100),
            new Job(2, 1, 19),
            new Job(3, 2, 27),
            new Job(4, 1, 25),
            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]
```