

Name: Swarnali Ghosh

Java Pre-Skilling Training Session

Assignment -5.6 (Task1 to Task3)

Module-5 (DAY 13 &14)

Mail-id: swarnalighosh666@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.

SOLUTION:

→Java program that solves the Tower of Hanoi puzzle for n disks.

```
package com.wipro.swarnali;

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) {
```

```

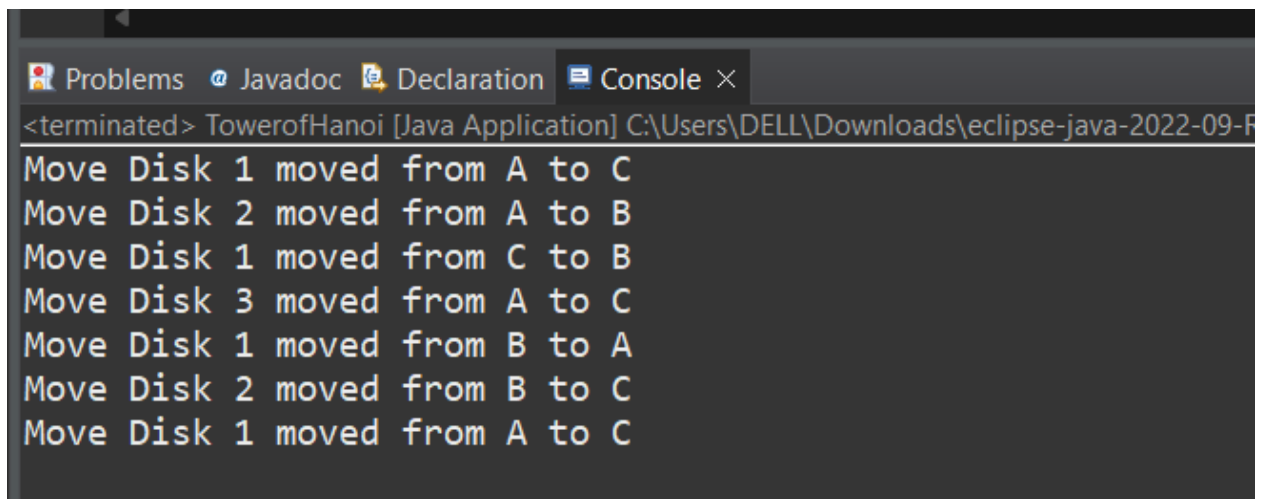
        if(n==1) {
            System.out.println("Move Disk 1 moved from " + rodFrom
+" to " + rodTo);
            return;
        }

        hanoi(n-1,rodFrom, rodTo, rodMiddle);

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

```

Output: -



The screenshot shows the Eclipse IDE's Console window. The title bar indicates the application is 'TowerofHanoi [Java Application]' running from 'C:\Users\DELL\Downloads\eclipse-java-2022-09-F'. The console output displays the sequence of moves for solving the Tower of Hanoi puzzle with 3 disks:

```

Move Disk 1 moved from A to C
Move Disk 2 moved from A to B
Move Disk 1 moved from C to B
Move Disk 3 moved from A to C
Move Disk 1 moved from B to A
Move Disk 2 moved from B to C
Move Disk 1 moved from A to C

```

The output shows the sequence of moves required to solve the Tower of Hanoi puzzle with 3 disks, following the rules of the game.

Task-2

Travelling 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.

SOLUTION:

```
package com.wipro.swarnali;

import java.util.Arrays;

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

        // Initialize the dp array with maximum values
        for (int[] row : dp) {
            Arrays.fill(row, Integer.MAX_VALUE / 2);
        }

        // Initialize starting city
        dp[1][0] = 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) {
                    for (int j = 0; j < n; j++) {
                        if ((mask & (1 << j)) != 0) {
                            dp[mask][i] = Math.min(dp[mask][i],
                                dp[mask ^ (1 << i)][j] + graph[j][i]);
                        }
                    }
                }
            }
        }
    }
}
```

```

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

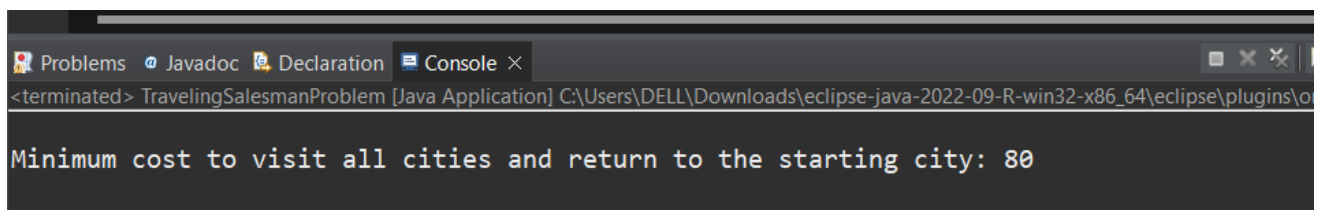
        return minCost;
    }

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

        System.out.println("\nMinimum cost to visit all cities and
return to the starting city: " + findMinCost(graph));
    }
}

```

Output: -



The screenshot shows the Eclipse IDE's console window. The title bar includes 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output displays the result of the program execution: 'Minimum cost to visit all cities and return to the starting city: 80'.

```

<terminated> TravelingSalesmanProblem [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x86_64\eclipse\plugins\o
Minimum cost to visit all cities and return to the starting city: 80

```

This implementation finds the minimum cost to visit all cities and return to the starting city using dynamic programming. The findMinCost function takes a 2D array graph representing the cost to travel between cities. It returns the minimum cost to visit all cities.

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.

SOLUTION:

Here's a Java implementation of the Job Sequencing Problem using the greedy method:

```
package com.wipro.swarnali;

import java.util.*;

class Job {
    int id, deadline, profit;

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

public class JobSequencingProblem {
    public static List<Job> jobSequencing(List<Job> jobs) {
        // Sort the jobs based on their profits in descending order
        jobs.sort((a, b) -> b.profit - a.profit);

        int maxDeadline = 0;
        for (Job job : jobs) {
            maxDeadline = Math.max(maxDeadline, job.deadline);
        }
    }
}
```

```

    }

    // Array to track whether a deadline slot is occupied or not
    boolean[] slots = new boolean[maxDeadline + 1];
    List<Job> result = new ArrayList<>();

    // Iterate through each job
    for (Job job : jobs) {
        // Find a slot before the deadline
        for (int i = job.deadline; i > 0; i--) {
            if (!slots[i]) {
                // If the slot is empty, assign the job to this
slot
                slots[i] = true;
                result.add(job);
                break;
            }
        }
    }

    return result;
}

public static void main(String[] args) {
    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));

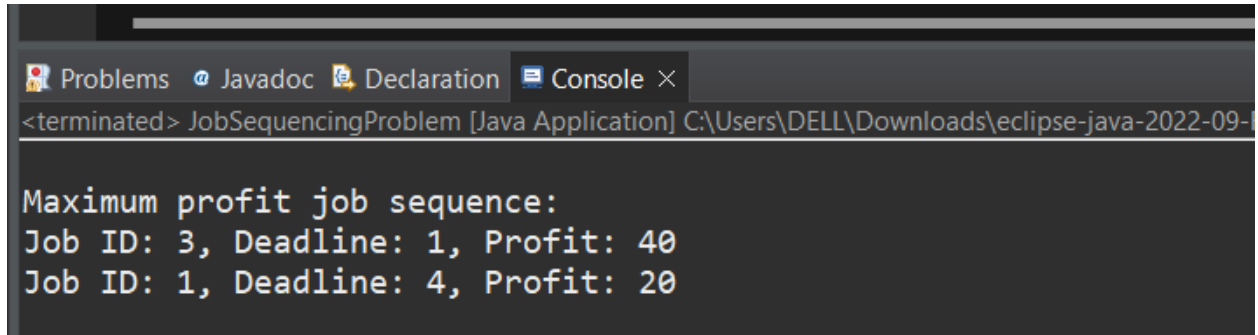
    List<Job> result = jobSequencing(jobs);

    System.out.println("\nMaximum profit job sequence:");

    for (Job job : result) {
        System.out.println("Job ID: " + job.id + ", Deadline: " +
job.deadline + ", Profit: " + job.profit);
    }
}
}

```

Output: -

A screenshot of the Eclipse IDE's console window. The window has a dark background and a light-colored title bar. The title bar contains the text "Problems Javadoc Declaration Console X". Below the title bar, the console output is displayed in a monospaced font. The output shows the text "<terminated> JobSequencingProblem [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-1" followed by a newline. The next line is "Maximum profit job sequence:". The following two lines are "Job ID: 3, Deadline: 1, Profit: 40" and "Job ID: 1, Deadline: 4, Profit: 20".

```
<terminated> JobSequencingProblem [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-1
Maximum profit job sequence:
Job ID: 3, Deadline: 1, Profit: 40
Job ID: 1, Deadline: 4, Profit: 20
```

In this implementation, the `Job` class is defined with properties `id`, `deadline`, and `profit`. The `jobSequencing` function takes a list of jobs and returns the maximum profit sequence of jobs that can be done before the deadlines.

It sorts the jobs based on their profits in descending order and then iterates through each job, assigning it to the latest available slot before its deadline.