Day 13 and 14: Manisha Assignment

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.

Explanation:

1. TowerOfHanoi Class:

- solveHanoi Method:

- Takes four parameters: `n` (number of disks), `source` (source rod), `auxiliary` (auxiliary rod), and `destination` (destination rod).

- If there is only one disk (`n == 1`), it prints the move from the source rod to the destination rod and returns.

- Recursively calls itself to move `n-1` disks from the source rod to the auxiliary rod using the destination rod as an auxiliary.

- Prints the move of the nth disk from the source rod to the destination rod.

- Recursively calls itself to move `n-1` disks from the auxiliary rod to the destination rod using the source rod as an auxiliary.

2. main Method:

- Defines the number of disks (`n`).

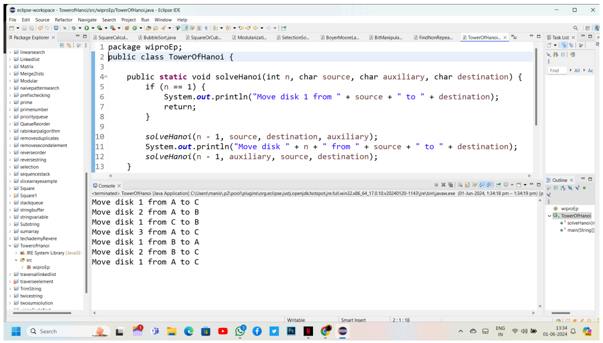
- Calls the `solveHanoi` method with `n` disks and the names of the rods ('A', 'B', and 'C').

- The program prints each move required to solve the Tower of Hanoi puzzle with `n` disks.

- Each move is in the format: "Move disk X from Y to Z", where `X` is the disk number, `Y` is the source rod, and `Z` is the destination rod.

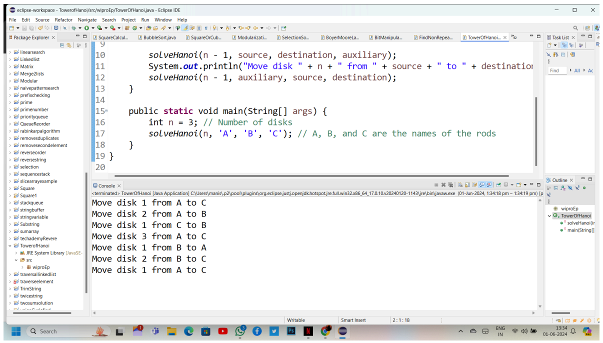
- The moves follow the rules of the game: only one disk can be moved at a time, and a disk can only be placed on an empty rod or on top of a larger disk.

Here Output as Follows:



```java
package wiproEp;
public class TowerOfHanoi {

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

        solveHanoi(n - 1, source, destination, auxiliary);
        System.out.println("Move disk " + n + " from " + source + " to " + destination);
        solveHanoi(n - 1, auxiliary, source, destination);
    }
```

```
<terminated> TowerOfHanoi [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 1:34:18 pm – 1:34:19 pm)
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
```



```java
        solveHanoi(n - 1, source, destination, auxiliary);
        System.out.println("Move disk " + n + " from " + source + " to " + destinatio
        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 the names of the rods
    }
}
```

```
<terminated> TowerOfHanoi [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 1:34:18 pm – 1:34:19 pm)
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.

Explanation:

1. FindMinCost Method:

- Takes a 2D array `graph` representing the cost to travel between cities.

- Initializes `n` as the number of cities and `VISITED_ALL` as the bitmask representing all cities visited.

- Creates a 2D array `dp` for memoization and initializes all values to `Integer.MAX_VALUE / 2` to avoid overflow.

- Calls the helper function `tsp` starting from city 0 with a bitmask representing the starting city visited.

2. tsp Method:

- Recursively calculates the minimum cost to visit all cities starting from `current` city and with the `mask` bitmask.

- If all cities have been visited (`mask == VISITED_ALL`), returns the cost to return to the starting city.

- If the value is already computed (`dp[current][mask]` is not the initial value), returns the stored value.

- Iterates through all cities, checking if the city has not been visited (`(mask & (1 << next)) == 0`).

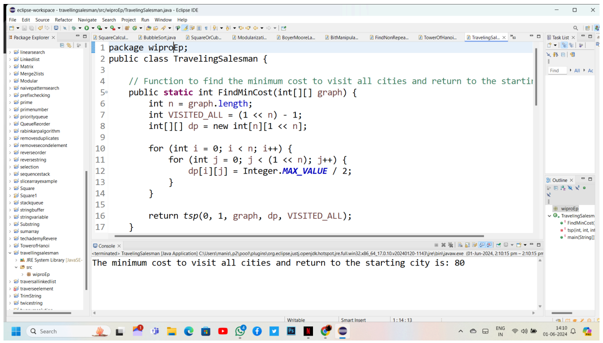
- Updates the `dp[current][mask]` with the minimum cost.

3. main Method:

- Defines the sample graph with 4 cities.

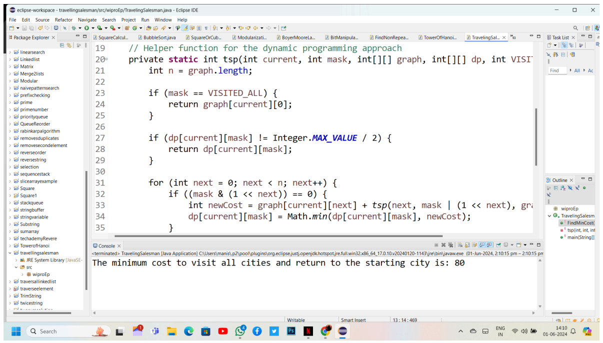
- Calls the `FindMinCost` method with the sample graph and prints the minimum cost.

Here output as Follows:



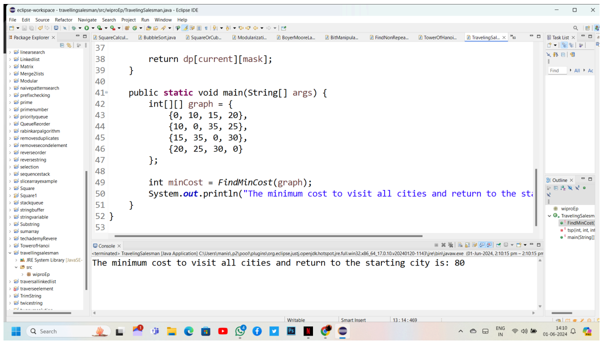
```
package wiproEp;
public class TravelingSalesman {

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

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < (1 << n); j++) {
                dp[i][j] = Integer.MAX_VALUE / 2;
            }
        }

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

```
<terminated> TravelingSalesman [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 2:10:15 pm – 2:10:15 pm
The minimum cost to visit all cities and return to the starting city is: 80
```



```
    // Helper function for the dynamic programming approach
    private static int tsp(int current, int mask, int[][] graph, int[][] dp, int VISIT
        int n = graph.length;

        if (mask == VISITED_ALL) {
            return graph[current][0];
        }

        if (dp[current][mask] != Integer.MAX_VALUE / 2) {
            return dp[current][mask];
        }

        for (int next = 0; next < n; next++) {
            if ((mask & (1 << next)) == 0) {
                int newCost = graph[current][next] + tsp(next, mask | (1 << next), gra
                dp[current][mask] = Math.min(dp[current][mask], newCost);
            }
```

```
<terminated> TravelingSalesman [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 2:10:15 pm – 2:10:15 pm
The minimum cost to visit all cities and return to the starting city is: 80
```



```

        return dp[current][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}
        };

        int minCost = FindMinCost(graph);
        System.out.println("The minimum cost to visit all cities and return to the sta
    }
}

```

```
<terminated> TravelingSalesman [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 2:10:15 pm – 2:10:15 pm
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.

Explanation:

1. Job Class:

- **Contains three properties: `Id`, `Deadline`, and `Profit`.**
- **Constructor initializes these properties.**

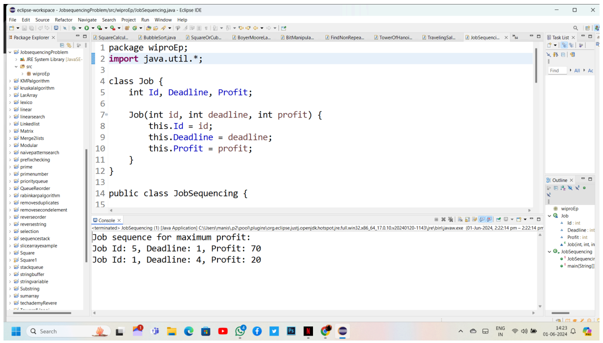
2. JobSequencing Method:

- **Takes a list of `Job` objects.**
- **Sorts the jobs in descending order of profit.**
- **Finds the maximum deadline to define the schedule array size.**
- **Initializes the schedule array to keep track of which jobs are scheduled at which time slots.**
- **Iterates through the jobs and schedules each job at the latest possible time slot before its deadline, ensuring no two jobs are scheduled at the same time.**
- **Returns the list of jobs in the sequence that yields the maximum profit.**

3. main Method:

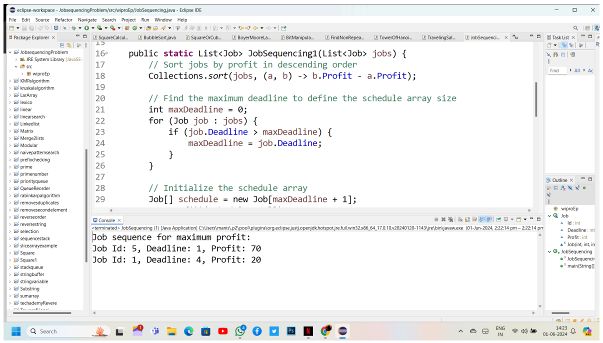
- **Creates a list of jobs with specific deadlines and profits.**
- **Calls the `JobSequencing` method with the list of jobs.**
- **Prints the job sequence for maximum profit.**

Here Output as Follows:



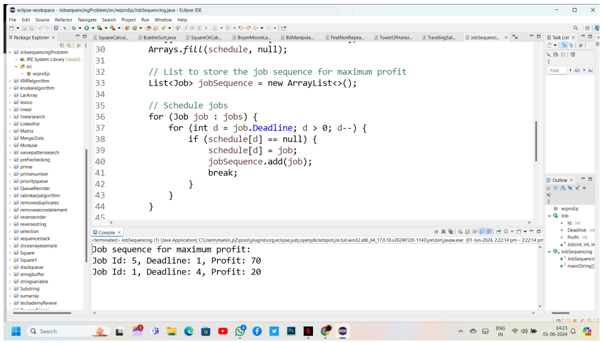
```
package wiproEp;
import java.util.*;

class Job {
    int Id, Deadline, Profit;

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

public class JobSequencing {

```

```
<terminated> JobSequencing (1) [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 2:22:14 pm – 2:22:14 pm
Job sequence for maximum profit:
Job Id: 5, Deadline: 1, Profit: 70
Job Id: 1, Deadline: 4, Profit: 20
```



```
    public static List<Job> JobSequencing1(List<Job> jobs) {
        // Sort jobs by profit in descending order
        Collections.sort(jobs, (a, b) -> b.Profit - a.Profit);

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

        // Initialize the schedule array
        Job[] schedule = new Job[maxDeadline + 1];
```

```
<terminated> JobSequencing (1) [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 2:22:14 pm – 2:22:14 pm
Job sequence for maximum profit:
Job Id: 5, Deadline: 1, Profit: 70
Job Id: 1, Deadline: 4, Profit: 20
```



```
        Arrays.fill(schedule, null);

        // List to store the job sequence for maximum profit
        List<Job> jobSequence = new ArrayList<>();

        // Schedule jobs
        for (Job job : jobs) {
            for (int d = job.Deadline; d > 0; d--) {
                if (schedule[d] == null) {
                    schedule[d] = job;
                    jobSequence.add(job);
                    break;
                }
            }
        }
```

```
<terminated> JobSequencing (1) [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (01-Jun-2024, 2:22:14 pm – 2:22:14 pm
Job sequence for maximum profit:
Job Id: 5, Deadline: 1, Profit: 70
Job Id: 1, Deadline: 4, Profit: 20
```