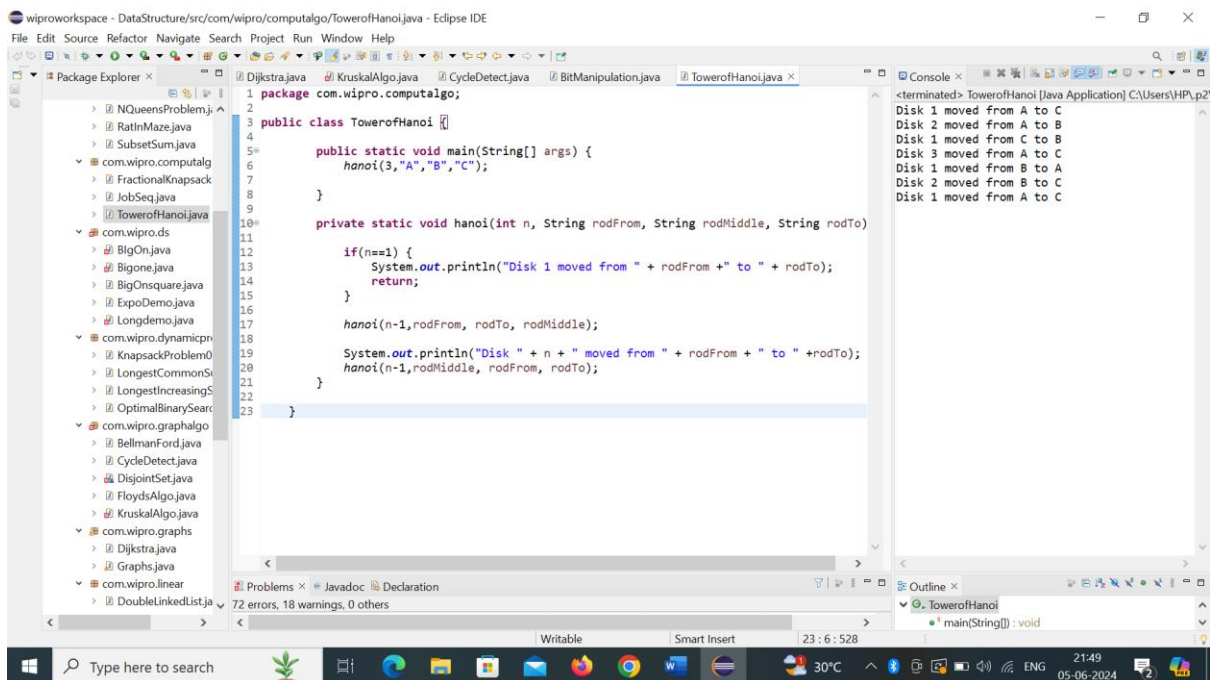


Day 13 and 14

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.



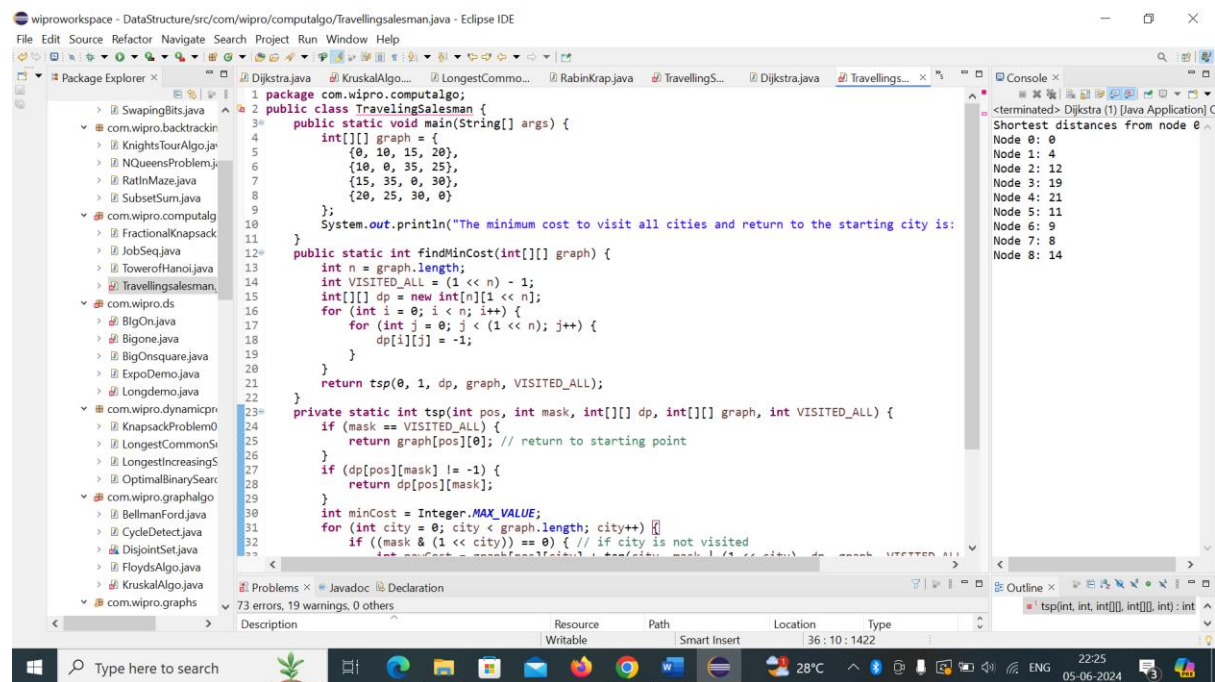
```
1 package com.wipro.computalgo;
2
3 public class TowerofHanoi {
4
5     public static void main(String[] args) {
6         hanoi(3, "A", "B", "C");
7     }
8
9     private static void hanoi(int n, String rodFrom, String rodMiddle, String rodTo)
10    {
11        if(n==1) {
12            System.out.println("Disk 1 moved from " + rodFrom + " to " + rodTo);
13            return;
14        }
15        hanoi(n-1, rodFrom, rodTo, rodMiddle);
16        System.out.println("Disk " + n + " moved from " + rodFrom + " to " + rodTo);
17        hanoi(n-1, rodMiddle, rodFrom, rodTo);
18    }
19 }
20
21
22
23 }
```

Console Output:

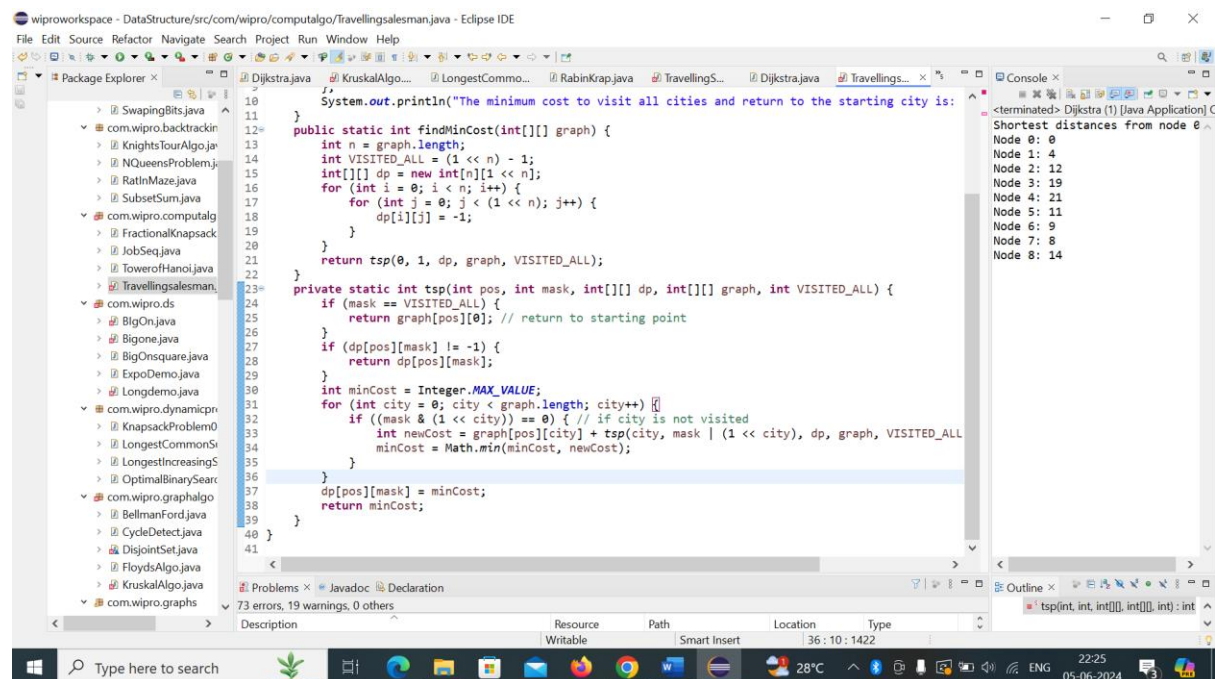
```
<terminated> TowerofHanoi [Java Application] C:\Users\HP\p2
Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C
```

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.



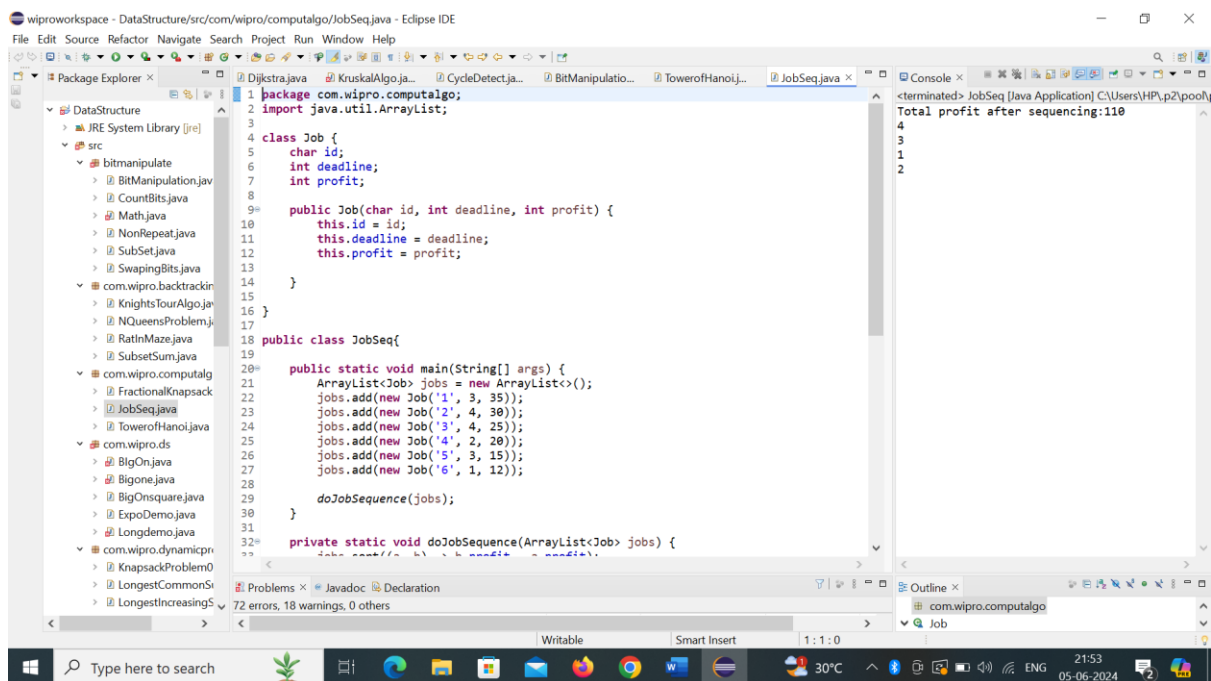
```
1 package com.wipro.computalgo;
2 public class TravelingSalesman {
3     public static void main(String[] args) {
4         int[][] graph = {
5             {0, 10, 15, 20},
6             {10, 0, 35, 25},
7             {15, 35, 0, 30},
8             {20, 25, 30, 0}
9         };
10        System.out.println("The minimum cost to visit all cities and return to the starting city is:");
11    }
12    public static int findMinCost(int[][] graph) {
13        int n = graph.length;
14        int VISITED_ALL = (1 << n) - 1;
15        int[][] dp = new int[n][1 << n];
16        for (int i = 0; i < n; i++) {
17            for (int j = 0; j < (1 << n); j++) {
18                dp[i][j] = -1;
19            }
20        }
21        return tsp(0, 1, dp, graph, VISITED_ALL);
22    }
23    private static int tsp(int pos, int mask, int[][] dp, int[][] graph, int VISITED_ALL) {
24        if (mask == VISITED_ALL) {
25            return graph[pos][0]; // return to starting point
26        }
27        if (dp[pos][mask] != -1) {
28            return dp[pos][mask];
29        }
30        int minCost = Integer.MAX_VALUE;
31        for (int city = 0; city < graph.length; city++) {
32            if ((mask & (1 << city)) == 0) { // if city is not visited
33                int newCost = graph[pos][city] + tsp(city, mask | (1 << city), dp, graph, VISITED_ALL);
34                minCost = Math.min(minCost, newCost);
35            }
36        }
37        dp[pos][mask] = minCost;
38        return minCost;
39    }
40 }
41 }
```



```
10        System.out.println("The minimum cost to visit all cities and return to the starting city is:");
11    }
12    public static int findMinCost(int[][] graph) {
13        int n = graph.length;
14        int VISITED_ALL = (1 << n) - 1;
15        int[][] dp = new int[n][1 << n];
16        for (int i = 0; i < n; i++) {
17            for (int j = 0; j < (1 << n); j++) {
18                dp[i][j] = -1;
19            }
20        }
21        return tsp(0, 1, dp, graph, VISITED_ALL);
22    }
23    private static int tsp(int pos, int mask, int[][] dp, int[][] graph, int VISITED_ALL) {
24        if (mask == VISITED_ALL) {
25            return graph[pos][0]; // return to starting point
26        }
27        if (dp[pos][mask] != -1) {
28            return dp[pos][mask];
29        }
30        int minCost = Integer.MAX_VALUE;
31        for (int city = 0; city < graph.length; city++) {
32            if ((mask & (1 << city)) == 0) { // if city is not visited
33                int newCost = graph[pos][city] + tsp(city, mask | (1 << city), dp, graph, VISITED_ALL);
34                minCost = Math.min(minCost, newCost);
35            }
36        }
37        dp[pos][mask] = minCost;
38        return minCost;
39    }
40 }
41 }
```

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.



```
1 package com.wipro.computalgo;
2 import java.util.ArrayList;
3
4 class Job {
5     char id;
6     int deadline;
7     int profit;
8
9     public Job(char id, int deadline, int profit) {
10         this.id = id;
11         this.deadline = deadline;
12         this.profit = profit;
13     }
14 }
15
16 public class JobSeq{
17
18     public static void main(String[] args) {
19         ArrayList<Job> jobs = new ArrayList<>();
20         jobs.add(new Job('1', 3, 35));
21         jobs.add(new Job('2', 4, 30));
22         jobs.add(new Job('3', 4, 25));
23         jobs.add(new Job('4', 2, 20));
24         jobs.add(new Job('5', 3, 15));
25         jobs.add(new Job('6', 1, 12));
26
27         doJobSequence(jobs);
28     }
29
30     private static void doJobSequence(ArrayList<Job> jobs) {
31         // ... (greedy algorithm implementation) ...
32     }
33 }
```

Console Output:

```
<terminated> JobSeq [Java Application] C:\Users\HP\p2\pool\
Total profit after sequencing:110
4
3
1
2
```

