

Computational Algorithms - 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.

Code:

```
package towerofhanoi;
import java.util.Scanner;

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);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Number of Disks :");
        int n = sc.nextInt();
        sc.close();
        solveHanoi(n, 'A', 'B', 'C');
    }
}
```

Output:

```
Console x
<terminated> TowerOfHanoi [Java Application] C
Enter Number of Disks :
4
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
Move disk 4 from A to C
Move disk 1 from B to C
Move disk 2 from B to A
Move disk 1 from C to A
Move disk 3 from B to C
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B 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.

Code:

```
package travelingsalesman;

import java.util.Arrays;

public class TravelingSalesman {
    public static int FindMinCost(int[][] graph) {
        int n = graph.length;
        int[][] dp = new int[n][n];

        for(int[] row : dp) {
            Arrays.fill(row, Integer.MAX_VALUE);
        }
    }
}
```

```

for(int i=1;i<n;i++) {
    dp[i][1] = graph[0][i];
}

for(int visited=2;visited<n;visited++) {
    for(int last=0;last<n;last++) {
        for(int prev=0;prev<n;prev++) {
            if(dp[prev][visited - 1] != Integer.MAX_VALUE && last != prev) {
                dp[last][visited] = Math.min(dp[last][visited], dp[prev][visited-1]+graph[prev][last]);
            }
        }
    }
}

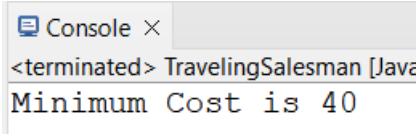
int minCost = Integer.MAX_VALUE;
for(int i=1;i<n;i++) {
    minCost = Math.min(minCost, dp[i][n-1]+graph[i][0]);
}

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}};
    System.out.println("Minimum Cost is " + FindMinCost(graph));
}
}

```

Output:



The screenshot shows a console window titled "Console" with a close button. The output text is:

<terminated> TravelingSalesman [Java

Minimum Cost is 40

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.

Job.java Code:

```
package jobsequencing;
```

```
public class Job {
```

```
    int Id;
```

```
    int Deadline;
```

```
    int Profit;
```

```
    Job(int id, int deadline, int profit) {
```

```
        this.Id = id;
```

```
        this.Deadline = deadline;
```

```
        this.Profit = profit;
```

```
    }
```

```
    @Override
```

```
    public String toString() {
```

```
        return "Job Id: " + Id + ", Deadline: " + Deadline + ", Profit: " + Profit;
```

```
    }
```

```
}
```

JobSequencing.java Code:

```
package jobsequencing;
```

```
import java.util.*;
```

```
public class JobSequencing {
```

```
    static List<Job> jobSequencing(List<Job> jobs) {
```

```
        jobs.sort((a, b) -> b.Profit - a.Profit);
```

```

int maxDeadline = 0;
for (Job job : jobs) {
    if (job.Deadline > maxDeadline) {
        maxDeadline = job.Deadline;
    }
}
Job[] result = new Job[maxDeadline];
boolean[] slot = new boolean[maxDeadline];

for (Job job : jobs) {
    for (int j = Math.min(maxDeadline - 1, job.Deadline - 1); j >= 0; j--) {
        if (!slot[j]) {
            slot[j] = true;
            result[j] = job;
            break;
        }
    }
}
List<Job> finalJobs = new ArrayList<>();
for (Job job : result) {
    if (job != null) {
        finalJobs.add(job);
    }
}
return finalJobs;
}

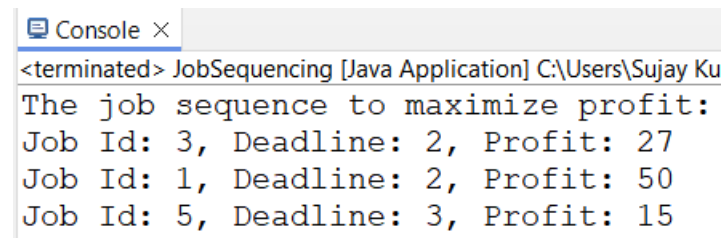
public static void main(String[] args) {
    List<Job> jobs = Arrays.asList(new Job(1, 2, 50), new Job(2, 1, 19), new Job(3, 2,
    27), new Job(4, 1, 25),
    new Job(5, 3, 15));

    List<Job> jobSequence = jobSequencing(jobs);

```

```
System.out.println("The job sequence to maximize profit:");  
for (Job job : jobSequence) {  
    System.out.println(job);  
}  
}  
}
```

Output:



The screenshot shows a console window titled "Console x" with the following output:

```
<terminated> JobSequencing [Java Application] C:\Users\Sujay Ku  
The job sequence to maximize profit:  
Job Id: 3, Deadline: 2, Profit: 27  
Job Id: 1, Deadline: 2, Profit: 50  
Job Id: 5, Deadline: 3, Profit: 15
```