**Assignment 8**

**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Title:**

Assignment Based on Dynamic Programming to Implement Travelling Salesman Problem (Determine Time and space complexity)

**Theory:**

## Objective

To implement the Travelling Salesman Problem using dynamic programming and analyze its time and space complexity. The goal is to find the shortest possible route visiting every city exactly once and returning to the city of origin.

## Theory and Explanation

The Travelling Salesman Problem (TSP) is a classic optimization problem where a salesman must travel to nnn cities, each exactly once, minimizing total travel cost (distance/time). The problem is NP-hard, and naive brute-force searches all permutations ((n−1)!(n-1)!(n−1)! possibilities), which is computationally infeasible for large nnn.

Dynamic Programming provides an efficient (though still exponential) solution by solving subproblems for subsets of cities and storing intermediate results to avoid redundant computations. The approach uses bitmasking to represent subsets and a recursive relation to determine the minimal tour cost.

For a subset SSS of cities including the starting city (say city 0), and a city j∈Sj \in Sj∈S, define C(S,j)C(S, j)C(S,j) as the shortest path length starting at city 0, visiting all cities in SSS exactly once, and ending at city jjj.

The recursive formula is:

C(S,j)=min⁡i∈S,i≠j[C(S−{j},i)+dist(i,j)]C(S, j) = \min\_{i \in S, i \neq j} [C(S - \{j\}, i) + dist(i, j)]C(S,j)=i∈S,i=jmin[C(S−{j},i)+dist(i,j)]

The base case: C({0},0)=0C(\{0\}, 0) = 0C({0},0)=0.

Finally, the minimal tour cost is:

min⁡j≠0[C(AllCities,j)+dist(j,0)]\min\_{j \neq 0} [C(\text{AllCities}, j) + dist(j, 0)]j=0min[C(AllCities,j)+dist(j,0)]

## Key Points

* Use bitmask to represent subsets of cities visited.
* Solve subproblems bottom-up or via memoization (top-down).
* Memoize shortest cost for visiting subset SSS ending at city jjj.
* Combine results to build toward the full solution covering all cities.
* Total subproblems: O(n×2n)O(n \times 2^n)O(n×2n), each takes O(n)O(n)O(n) time.

## Pseudocode

text

function tsp(dist, mask, pos, n, memo):

if mask == (1 << n) - 1:

return dist[pos][0]

if memo[pos][mask] is not -1:

return memo[pos][mask]

ans = INFINITY

for city in 0 to n-1:

if city not visited in mask:

ans = min(ans, dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo))

memo[pos][mask] = ans

return ans

main:

n = number of cities

dist = n x n matrix of distances

memo = 2D array of size n x (1<<n) initialized with -1

result = tsp(dist, 1, 0, n, memo)

print result

## Example Output

Input:

text

Enter number of cities:

4

Enter distance matrix:

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

Output:

text

Minimum cost of travelling all cities: 80

(This corresponds to the route 0 -> 1 -> 3 -> 2 -> 0 with total distance 80.)

**Code:**

import java.util.Scanner;

public class TravellingSalesman {

    static int tsp(int[][] dist, int mask, int pos, int n, int[][] memo) {

        if (mask == (1 << n) - 1) {

            return dist[pos][0];

        }

        if (memo[pos][mask] != -1) {

            return memo[pos][mask];

        }

        int ans = Integer.MAX\_VALUE;

        for (int city = 0; city < n; city++) {

            if ((mask & (1 << city)) == 0) {

                int newAns = dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo);

                ans = Math.min(ans, newAns);

            }

        }

        memo[pos][mask] = ans;

        return ans;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of cities:");

        int n = sc.nextInt();

        int[][] dist = new int[n][n];

        System.out.println("Enter distance matrix:");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                dist[i][j] = sc.nextInt();

            }

        }

        int[][] memo = new int[n][1 << n];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < (1 << n); j++) {

                memo[i][j] = -1;

            }

        }

        int result = tsp(dist, 1, 0, n, memo);

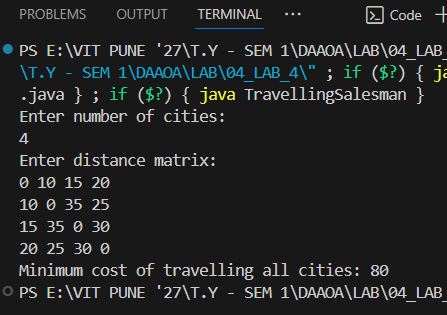
        System.out.println("Minimum cost of travelling all cities: " + result);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity

* Number of subproblems: n×2^n (position × visited subset)
* For each subproblem, we check up to n cities.
* Overall time complexity: **O(n^2×2^n)**

## Space Complexity

* Memoization table size: n×2^n
* Extra space for recursion stack: O(n)
* Overall space complexity: **O(n×2^n)**

## Pseudocode with Complexity Comments

text

FUNCTION tsp(dist, mask, pos, n, memo)

IF mask == (1 << n) - 1 // Check if all cities visited; Time: +1

RETURN dist[pos][0] // Return distance back to start; Time: +1

IF memo[pos][mask] != -1 // Memoization check; Time: +1

RETURN memo[pos][mask]

ans ← INFINITY // Time: +1

FOR city FROM 0 TO n-1 // Time: +n (check all possible next cities)

IF (mask & (1 << city)) == 0 // If city not visited; Time: +1 per city

newAns ← dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo) // Recursion; Time: exponential by subproblem count

ans ← MIN(ans, newAns) // Time: +1 (compare and assign)

memo[pos][mask] ← ans // Store computed answer; Time: +1

RETURN ans // Return minimum cost found; Time: +1

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of cities:" // Time: +1

INPUT n // Time: +1

DECLARE 2D array dist[n][n] // Space: +n^2

PRINT "Enter distance matrix:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

FOR j FROM 0 TO n-1 // Time: +n per i; Total: \*n^2

INPUT dist[i][j] // Time: +1 per input

DECLARE 2D array memo[n][1 << n] // Space: +n \* 2^n

FOR i FROM 0 TO n-1 // Time: +n

FOR j FROM 0 TO (1 << n) - 1 // Time: +2^n per i; Total: \*n\*2^n

memo[i][j] ← -1 // Time: +1 per initialization

result ← tsp(dist, 1, 0, n, memo) // Recursive call; Time: O(n^2 \* 2^n)

PRINT "Minimum cost of travelling all cities: " + result // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

## Complexity Explanation

* **Time Complexity:** There are n×2n possible state combinations of (current city, visited subset). Each state recursively calls up to n cities, so total time complexity is O(n^2 2^n)
* **Space Complexity:** The memoization table stores results for each (city, mask) state: O(n 2^n)space.
* Input distance matrix uses O(n^2) space, but this is dominated by memo space.
* The recursive recursion stack depth at most O(n), which is less significant compared to memo.

**Conclusion**

In this lab exercise, we learned how to Travelling Salesman problem using Dynamic Programming.