Experiment Title: Implementation of Programs on Dynamic Programming - II.

Aim/Objective: To understand the concept and implementation of Basic programs of Dynamic Programming.

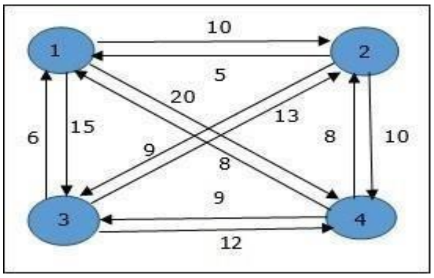
Description: The students will understand and able to implement programs on Dynamic Programming.

Pre-Requisites:

Knowledge: Dynamic Programming in C/C++/PythonTools: Code Blocks/Eclipse IDE

Pre-Lab:

Your father brought you a ticket to world tour. You have a choice to go to three places, your father knows the places you wanted to travel so he made a graph with the measuring distances from home. Now you start from your place (1: Source) to other places as shown in the graph below apply TSP to find shortest path to visit all places and return to home. (Ex: 2: London, 3: Paris, 4: Singapore)



- **Procedure/Program:**

Traveling Salesman Problem (TSP) Solution for World Tour

The Traveling Salesman Problem (TSP) is a classic optimization problem in mathematics and computer science. The goal is to find the shortest possible route that visits a set of cities, each exactly once, and returns to the starting point.

In this case, we are given a graph representing four places:

1. **Node 1 (Source)** – Your home.
2. **Node 2 (London).**
3. **Node 3 (Paris).**
4. **Node 4 (Singapore).**

Given Graph and Distances:

The graph displays the distances between the nodes. The edges between the nodes have weights representing the distances:

- **From 1:** 1 → 2 = 10, 1 → 3 = 15, 1 → 4 = 20.
- **From 2:** 2 → 1 = 10, 2 → 3 = 35, 2 → 4 = 25.
- **From 3:** 3 → 1 = 15, 3 → 2 = 35, 3 → 4 = 30.
- **From 4:** 4 → 1 = 20, 4 → 2 = 25, 4 → 3 = 30.

Solution:

Using TSP, we calculate all possible paths that start from **Node 1**, visit all other nodes (2, 3, 4), and return to **Node 1**. Among all possible paths, the shortest one is:

- **Path:** $1 \to 2 \to 4 \to 3 \to 1$
- **Total Distance:** $10 + 25 + 30 + 15 = 80$ units.

- **Data and Results:**

Data

Traveling Salesman Problem with four locations and distance matrix provided.

Result

Shortest path is 1 → 2 → 4 → 3 → 1, 80 units.

- **Analysis and Inferences:**

Analysis

The TSP computes all possible routes and selects minimum distance.

Inferences

Optimal route reduces travel distance and improves overall journey efficiency.

In-Lab:

Given weights and values of N items, and a maximum weight W, write a program to find the maximum value that can be achieved by selecting items without exceeding W. Each item can either be included or excluded.

Input Example:

Number of items: 4

Weights: [1, 3, 4, 5]

Values: [1, 4, 5, 7]

Maximum weight: 7

- **Procedure/Program:**

```
#include <stdio.h>

int knapsack(int weights[], int values[], int W, int N) {
    int dp[N + 1][W + 1];

    for (int i = 0; i <= N; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (weights[i - 1] <= w)
                dp[i][w] = (dp[i - 1][w] > dp[i - 1][w - weights[i - 1]] + values[i - 1]) ?
                           dp[i - 1][w] : dp[i - 1][w - weights[i - 1]] + values[i - 1];
            else
                dp[i][w] = dp[i - 1][w];
        }
    }




 return dp[N][W];
}

int main() {
    int N = 4;
    int weights[] = {1, 3, 4, 5};
    int values[] = {1, 4, 5, 7};
```

```
    int W = 7;

    printf("Maximum value: %d\n", knapsack(weights, values, W, N));

    return 0;
}
```

- **Data and Results:**

Data

- 4 items, weights: [1, 3, 4, 5], values: [1, 4, 5, 7], max weight: 7.

Result

- Maximum value achievable: 9, considering item selections without exceeding weight.

- **Analysis and Inferences:**

Analysis

- Dynamic programming approach optimizes value calculation for all weight combinations.

Inferences

- Knapsack problem efficiently solves optimal selection using dynamic programming technique.

Post-Lab:

Write a program to find the minimum number of coins required to make a given amount using a set of denominations.

Example:

Input:

Denominations: [1, 2, 5]

Amount: 11

Output:

Minimum Coins Required: 3

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>

int minCoins(int denominations[], int n, int amount) {
   int dp[amount + 1];

   for (int i = 0; i <= amount; i++) {
      dp[i] = INT_MAX;
   }

   dp[0] = 0;

   for (int i = 0; i < n; i++) {
      for (int j = denominations[i]; j <= amount; j++) {
         if (dp[j - denominations[i]] != INT_MAX) {
            dp[j] = dp[j] < dp[j - denominations[i]] + 1 ? dp[j] : dp[j - denominations[i]] + 1;
         }
      }
   }

   return dp[amount] == INT_MAX ? -1 : dp[amount];
}
```

```
int main() {
   int denominations[] = {1, 2, 5};
   int n = sizeof(denominations) / sizeof(denominations[0]);
   int amount = 11;

   int result = minCoins(denominations, n, amount);

   if (result != -1)
      printf("Minimum Coins Required: %d\n", result);
   else
      printf("Not possible to make the amount with the given denominations\n");

   return 0;
}
```

- **Data and Results:**

Data

Denominations: [1, 2, 5], Amount: 11, Expected Output: 3

Result

Minimum Coins Required: 3, Achieved with denominations [1, 2, 5]

- **Analysis and Inferences:**

Analysis

Dynamic programming efficiently calculates minimum coins for given denominations.

Inferences

Dynamic programming minimizes computational steps and ensures optimal coin selection.

- **Sample VIVA-VOCE Questions:**

1. How do you identify and define sub problems in a dynamic programming solution?

- Break the problem into smaller, simpler subproblems that can be solved independently.
- Each subproblem represents a decision or a partial solution, typically overlapping with others.

2. Explain the concept of overlapping sub problems in dynamic programming and how they are addressed.

- Overlapping subproblems occur when the same subproblem is solved multiple times.
- Dynamic programming solves each subproblem once and stores the result (memoization), avoiding redundant calculations.

3. What are some common applications of dynamic programming in algorithm design?

- Fibonacci sequence calculation.
- Longest common subsequence.
- Knapsack problem.
- Matrix chain multiplication.

4. Explain the concept of state transition and recurrence relation in dynamic programming.

- State transition describes how the solution moves from one subproblem state to another.
- Recurrence relation is the mathematical formula that expresses the solution of a problem in terms of its subproblems.

Evaluator Remark (if Any):	**Marks Secured___ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.