Experiment **#12**		Student ID	
Date		Student Name	[@KLWKS_BOT THANOS]

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

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

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

Pre-Requisites:

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

Pre-Lab:

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example-1:

Input: n = 2 **Output**: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

Example 2:

Input: n = 3 **Output**: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

- **Algorithm/Program**:

```
#include <stdio.h>

int climbStairs(int n) {
    if (n <= 2) {
        return n;
    }
    int prev = 1, curr = 2, next;
    for (int i = 3; i <= n; i++) {
```

```
  next = prev + curr;
      prev = curr;

      curr = next;
   }
   return curr;
}

int main() {
   int n1 = 2, n2 = 3;
   printf("Ways to climb %d steps: %d\n", n1, climbStairs(n1));
   printf("Ways to climb %d steps: %d\n", n2, climbStairs(n2));
   return 0;
}
```

- **Data and Results:**

Data:

Two inputs: $n = 2$ and $n = 3$.

Result:

For $n = 2$: 2 ways; for $n = 3$: 3 ways.

- **Inference Analysis**:

Analysis:

Number of ways follows Fibonacci sequence for given n.

Inferences:

Climbing stairs is solved by summing previous two steps.

In-Lab:

Write a program to compute the n^{th} Fibonacci number using Dynamic Programming (DP).

Input Format:

A single integer n, where $n \geq 0$

Output Format:

A single integer representing the n^{th} Fibonacci number.

Constraints:

$0 \leq n \leq 10^5$

Example:

Input: 10 **Output**: 55

- **Procedure/Algorithm:**

```
#include <stdio.h>

long long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    long long fib[n + 1];
    fib[0] = 0;
    fib[1] = 1;

    for (int i = 2; i <= n; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }

    return fib[n];
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%lld\n", fibonacci(n));
    return 0;
}
```

- **Data and Results:**

Data

Input: Integer n, constraints: $0 \leq n \leq 10^5$.

Result

Fibonacci number for given n computed using dynamic programming.

- **Inference Analysis**:

Analysis

Algorithm uses $O(n)$ time and $O(n)$ space complexity.

Inferences

Efficient computation with memory optimization possible for larger n.

Post-Lab:

Given a set of coin denominations and a target amount, find the minimum number of coins required to make up that amount. Assume an unlimited supply of each coin denomination.

Input:

Coin denominations: {1, 2, 5}

Target amount: 11

Output:

Minimum number of coins needed to make 11: 3

- **Procedure/Algorithm:**

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

int minCoins(int coins[], int n, int target) {
    int dp[target + 1];

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

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

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

int main() {
    int coins[] = {1, 2, 5};
```

```
int n = sizeof(coins) / sizeof(coins[0]);
    int target = 11;

    int result = minCoins(coins, n, target);
    if (result != -1) {
        printf("Minimum number of coins needed to make %d: %d\n", target, result);
    } else {
        printf("It is not possible to make %d with the given denominations.\n", target);
    }

    return 0;
}
```

- **Data and Results:**

Data

Coin denominations: {1, 2, 5}, Target amount: 11, Output: 3.

Result

Minimum number of coins needed to make 11: 3.

- **Analysis and Inferences:**

Analysis

Dynamic programming finds minimum coins by iterating through denominations and amounts.

Inferences

3 coins are required to form the target amount 11 efficiently.

- **Sample VIVA-VOCE Questions:**

1. What fundamental principle forms the basis of Dynamic Programming?

- The principle of optimality, where an optimal solution to a problem can be constructed from optimal solutions to its subproblems.

2. Name two key characteristics of problems well-suited for Dynamic Programming solutions.

- Overlapping subproblems: Subproblems are solved multiple times.
- Optimal substructure: The optimal solution can be formed from optimal solutions of subproblems.

3. Can you list any two classical problems that are often solved using Dynamic Programming?

- Fibonacci sequence computation.
- Knapsack problem.

4. How does Dynamic Programming differ from greedy algorithms in problem-solving?

- Dynamic programming solves problems by breaking them into subproblems, solving them once, and storing results. Greedy algorithms make local choices without revisiting subproblems.

5. Mention any advantage of using Dynamic Programming over brute force methods.

- Dynamic programming avoids recomputation of overlapping subproblems, reducing time complexity significantly.

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