Student Name: ABHISHEK KUMAR **UID:24MCC20006**
Branch: MCA-CCD **Section/Group:1-A**
Semester: 1st **Date of Performance:30-10-24**
Subject Name: Design and Analysis of Algorithms Lab **Subject Code: 24CAP-612**

Case Study: Dynamic Programming and Its Applications

1. Introduction

Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them down into overlapping subproblems. It provides efficient solutions by storing intermediate results to avoid redundant calculations. Problems like the **Knapsack Problem** and **Longest Common Subsequence (LCS)** are classic examples of DP. These algorithms find applications in resource allocation, sequence alignment, and many real-world domains such as finance and bioinformatics.

In this case study, we will explore two problems solvable using dynamic programming and analyze the role of **memorization**, performance improvements, and comparisons with naive approaches.

1.Aim:

The objective of this study is to understand:

- How dynamic programming breaks down complex problems into smaller subproblems.
- The importance of memorization in reducing time complexity.
- A comparison between brute-force solutions and dynamic programming**.**
- Real-world applications of the algorithms.

2.Task to be Done:

1. Implement two problems solvable by dynamic programming:

- **Knapsack Problem** (0/1 version)
- **Longest Common Subsequence (LCS)**

2. Analyze the following for each:

- Time complexity improvements through **memorization** or **tabulation**.
- Comparison with brute-force or naive recursive methods.
- Implement both problems in **Python** and observe the outputs.

3.Algorithm/ Flowchart:
Algorithm: 0/1 Knapsack Problem

- **Problem**: Given a set of items with weights and values, determine the maximum value that can be obtained with a fixed capacity.
- **Approach**: Use a 2D table to store intermediate results, where dp[i][j] represents the maximum value using the first i items with capacity j.

Knapsack Algorithm (DP Approach):

1. Create a 2D array dp of size (n+1) × (capacity+1).
2. Initialize dp[0][j] = 0 for all j (no items, no value).
3. For each item i and capacity j:

- If weight[i-1] <= j:

dp[i][j] = max(value[i-1] + dp[i-1][j-weight[i-1]], dp[i-1][j])

- Else:

dp[i][j] = dp[i-1][j]

4. Return dp[n][capacity].

Flowchart: Knapsack Problem

1. Start
2. Initialize 2D table dp.
3. Loop through all items and capacities.
4. Store the maximum value in dp[i][j].
5. End

Algorithm: Longest Common Subsequence (LCS)

- **Problem**: Find the length of the longest subsequence that two sequences share in common.
- **Approach**: Use a 2D table dp where dp[i][j] stores the length of the LCS for the first i characters of string A and j characters of string B.

LCS Algorithm (DP Approach):

1. Create a 2D table dp of size (m+1) × (n+1) (where m and n are the lengths of the two strings).
2. Initialize dp[i][0] and dp[0][j] to 0.
3. For each i and j:
- If A[i-1] == B[j-1]:

dp[i][j] = dp[i-1][j-1] + 1

- Else:

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

4. Return dp[m][n].

Flowchart: LCS

1. Start
2. Initialize a 2D table dp.
3. Loop through characters of both strings.
4. Update dp[i][j] based on matching characters.
5. End

4.Code for experiment/Output:

- **Knapsack Problem**

```
def knapsack(weights, values, capacity):
    n = len(values)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

# Example usage
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5
print("Maximum value in Knapsack =", knapsack(weights, values, capacity))
```

- **Output:**

```
Maximum value in Knapsack = 7
```

- **Longest Common Subsequence**

```python
def lcs(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage
str1 = "AGGTAB"
str2 = "GXTXAYB"
print("Length of LCS =", lcs(str1, str2))
```

- **Output:**

```
Length of LCS = 4
```

6.Conclusion:

Dynamic programming offers an efficient way to solve problems with overlapping subproblems and optimal substructure. The 0/1 Knapsack Problem and LCS illustrate how storing intermediate results can significantly reduce redundant computations. Compared to naive recursive solutions, which have exponential time complexity, DP provides polynomial-time solutions by memorization or tabulation. These algorithms are widely applicable in real-world scenarios such as resource allocation, bioinformatics, and text comparison.

Time And Space Complexity:

- **Knapsack Problem:**
  - Time Complexity: O(n×W)O(n \times W)O(n×W) where nnn is the number of items and WWW is the knapsack capacity.

- Space Complexity: O(n×W)O(n \times W)O(n×W) for the dp array.

- **Longest Common Subsequence:**
  - Time Complexity: O(m×n)O(m \times n)O(m×n) where mmm and nnn are the lengths of the two sequences.

  - Space Complexity: O(m×n)O(m \times n)O(m×n) for the dp array.

Dynamic programming not only simplifies the complexity of solving problems but also provides a structured and efficient way to tackle optimization challenges across various fields.

7.Learning Outcome:

1. Understanding of Dynamic Programming: Learners will gain a clear understanding of the dynamic programming approach and its benefits in solving complex problems.
2. Implementation Skills: Learners will be able to implement dynamic programming algorithms for specific problems like the Knapsack Problem and LCS.
3. Comparison of Approaches: Learners will be able to compare and contrast dynamic programming with brute-force and recursive methods in terms of efficiency and practicality.
4. Application Awareness: Learners will become aware of the practical applications of dynamic programming in various domains, enhancing their problem-solving toolkit.

This case study provides a comprehensive overview of dynamic programming, emphasizing its significance and application in solving real-world problems.

Teacher's Signature