DYNAMIC PROGRAMMING :

Dynamic programming (DP) is a method used to solve problems by breaking them down into smaller, overlapping subproblems, solving each subproblem only once, and storing the solutions. Instead of recalculating the same results repeatedly, DP uses a memory-based approach (called **memoization**) to reuse previously computed results, making the algorithm more efficient.

Types of Dynamic Programming Approaches:

1. **Top-Down (Memoization)**: Solve the problem recursively, and store the results of subproblems to avoid recomputation.

2. **Bottom-Up (Tabulation)**: Solve smaller subproblems first and use their results to build up to the solution of the larger problem

EXAMPLES OF DYNAMIC PROGRAMMING:

- Fibonacci Sequence
- Longest Common Subsequence (LCS)
- Edit Distance (Levenshtein Distance)
- Longest Increasing Subsequence (LIS)
- Matrix Chain Multiplication

1.Fibonacci Sequence :

- **Problem**: Compute the *nn*n-th Fibonacci number, where each number is the sum of the two preceding ones: F(n)=F(n−1)+F(n−2).

- **DP Approach**: Instead of recalculating the Fibonacci sequence recursively, store previously calculated Fibonacci numbers to avoid redundant computations.

- **Time Complexity**: O(n), since each Fibonacci number from 1 to n is computed once.
- **Space Complexity**: O(n) if we store all values in an array. This can be optimized to O(1) by only keeping the last two Fibonacci numbers and updating them in place.

2.Longest Common Subsequence (LCS):

- **Problem**: Find the longest subsequence that appears in the same order in two sequences (though not necessarily contiguous).

- **DP Approach**: Use a 2D table to store the length of the LCS for each pair of prefixes of the two sequences. The table is filled based on whether characters match or not.

- **Time Complexity**: O(mn), where m and n are the lengths of the two sequences. We check every pair of characters and compute the LCS length for each pair.
- **Space Complexity**: O(mn) for storing the table. This can sometimes be reduced to O(min(m,n)) by storing only the current and previous rows of the table.

3.Edit Distance :

- **Problem**: Find the minimum number of operations (insertions, deletions, substitutions) required to convert one string into another.

- **DP Approach**: Use a 2D table to store the edit distances between all prefixes of the two strings. Each cell in the table is filled based on the cost of performing an operation (insertion, deletion, substitution).

- **Time Complexity**: O(mn), where m and n are the lengths of the two strings. We compute the minimum operations for each character pair.

- **Space Complexity**: O(mn) for the 2D table, but with optimization, this can be reduced to O(min(m,n)), since we only need the current and previous rows

4.Longest Increasing Subsequence:

- **Problem**: Find the length of the longest subsequence in an array where the elements are in increasing order.

- **DP Approach**: Use an array to store the length of the LIS ending at each index. For each element, compare it with all previous elements and update the LIS length if the current element is larger.

- **Time Complexity**: O(n^2), where n is the length of the array. For each element, we compare it with all previous elements to update the LIS.
- **Space Complexity**: O(n), since we only need an array of size n to store the LIS length at each index. Optimized solutions using binary search can reduce the time complexity to O(nlogn).

5.Matrix Chain Multiplication:

- **Problem**: Given a sequence of matrices, determine the most efficient way to multiply them (minimizing the total number of scalar multiplications).

- **DP Approach**: Use a 2D table where each cell stores the minimum number of multiplications needed to multiply a specific subchain of matrices. Fill the table by considering different ways to split the chain.

- **Time Complexity**: O(n^3), where *nn*n is the number of matrices. For each pair of matrices, we try all possible ways to split them and compute the scalar multiplications.
- **Space Complexity**: O(n^2) for the 2D table used to store the minimum multiplications required for each subchain of matrices.