Dynamic Programming

- dynamic programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and
- utilizing the fact that the optimal solution to the overall problem depends on the optimal solutions to its subproblems

Two Key Concepts:

1. **Optimal Substructure:**
    - A problem exhibits an optimal substructure if an optimal solution can be constructed from the optimal solutions of its subproblems.
    - This means that solving the smaller subproblems and combining their solutions yields an optimal solution to the entire problem.

    Example:

    - In the **Shortest Path Problem**, the shortest path between two nodes AA and BB can be constructed from the shortest paths between intermediate nodes along the way, thereby showing an optimal substructure.
2. **Overlapping Subproblems:**
    - A problem exhibits overlapping subproblems if the same subproblems are solved multiple times in the process of solving the main problem.
    - DP leverages this by storing solutions to subproblems to avoid redundant calculations.

    Example:

    - In the **Fibonacci Sequence Problem**, the nth Fibonacci number can be expressed as the sum of the (n-1)th and (n-2)th Fibonacci numbers, and these subproblems are repeatedly called when computing larger Fibonacci numbers.

Proving Optimal Substructure:

1. **Divide and Conquer Approach:**
    - Break the problem down into smaller subproblems.
    - Show how the optimal solution to these smaller subproblems can be combined to form the optimal solution to the entire problem.
2. **Recursive Solution:**
    - Define a recursive function that calculates the solution by combining results from smaller subproblems.
    - For example, in the **Longest Common Subsequence Problem**, the function $LCS(X,Y)L**CS(X,Y)$ can recursively call $LCS(X-1,Y)L**CS(X-1,Y)$ and $LCS(X,Y-1)L**CS(X,Y-1)$ to build the solution.
3. **Correctness Proof:**
    - Show that the recursive solution forms an optimal solution by verifying that the problem's global optimum is achieved by combining the local optima.

DP in Action:

1. **Top-Down Approach:**
    - Use recursion with memoization to store the results of subproblems, preventing repeated computations.
2. **Bottom-Up Approach:**
    - Build up the solution iteratively by solving the smallest subproblems first and storing their results in a table or array.
3. **Time Complexity:**
    - DP algorithms are often efficient with time complexities ranging from $O(n)$ to $O(n2)$ depending on the problem, due to the reuse of subproblem solutions.

Common Examples:

1. **Knapsack Problem:**
    - Has an optimal substructure where the maximum value obtainable for a given capacity is the maximum of including or excluding the current item, plus the values obtained from the remaining capacity.
2. **Edit Distance Problem:**
    - Computes the minimum number of edits required to transform one string into another. This can be built recursively by considering the costs associated with inserting, deleting, or substituting characters.