#### GREEDY ALGORITHM

**A greedy algorithm is a problem-solving approach that builds up a solution piece by piece, always choosing the option that seems the best at that moment (locally optimal choice). The hope is that by choosing a local optimum at each step, the overall solution will also be globally optimal.**

Greedy algorithms do not always guarantee an optimal solution but work well for problems that have optimal substructure and follow the greedy choice property.

**Example:**

One classic example of a greedy algorithm is the Coin Change Problem:

Problem: Given a set of coin denominations (e.g., 1, 5, 10, 25) and an amount, find the minimum number of coins needed to make that amount.

Greedy Approach:

* Always choose the largest denomination that does not exceed the remaining amount.

For example, to make an amount of 63 using denominations of 25, 10, 5, and 1:

* Start by choosing the largest coin (25): 63 - 25 = 38.
* Choose another 25: 38 - 25 = 13.
* Now, choose 10: 13 - 10 = 3.
* Finally, choose three 1 coins: 3 - 1 - 1 - 1 = 0.

So, the minimum number of coins is 6: (25 + 25 + 10 + 1 + 1 + 1).

**Time Complexity:**

The time complexity of a greedy algorithm depends on the specific problem and how the greedy choices are made. In general, it can vary from O(n)O(n)O(n) (linear) to O(nlog⁡n)O(n \log n)O(nlogn) depending on how the greedy choice is determined.

* For the coin change problem described above, if the list of denominations is already sorted, the time complexity is O(n)O(n)O(n), where nnn is the number of coins needed.

**Space Complexity:**

Greedy algorithms typically have a low space complexity because they make decisions step-by-step and usually don't require maintaining large data structures. In most cases, the space complexity is O(1)O(1)O(1), i.e., constant space.

#### ****DYNAMIC PROGRAMMING****

**Dynamic Programming (DP)** is a method for solving problems by breaking them down into overlapping subproblems, solving each just once, and storing their solutions for reuse.

**Types of DP**:

1. **Memoization (Top-Down)**: Recursive approach where you store results in a table to avoid recomputing.
2. **Tabulation (Bottom-Up)**: Iterative approach where you build up the solution from the smallest subproblems.

**Common DP Algorithms**:

1. **0/1 Knapsack Problem**: Solves the knapsack problem by storing solutions to subproblems where different items and capacities are considered.
2. **Longest Common Subsequence (LCS)**: Finds the longest subsequence common to two strings.
3. **Fibonacci Series**: Uses DP to compute Fibonacci numbers without redundant calculations

**TIME COMPLEXITY**

* **Bottom-up Approach (Tabulation):** O(n)O(n)O(n) because each Fibonacci number from 0 to nnn is computed once.
* **Top-down Approach (Memoization):** O(n)O(n)O(n) for the same reason, but with slightly different recursion mechanics.

**Space Complexity:**

* **Bottom-up Approach (Tabulation):** O(n)O(n)O(n) because we store all Fibonacci numbers up to nnn.
* **Top-down Approach (Memoization):** O(n)O(n)O(n) for storing the results in the memoization table. However, it also has an additional overhead of O(n)O(n)O(n) for the recursion call stack.