1. Greedy algorithm

A **greedy algorithm** builds up a solution piece by piece, choosing the locally optimal choice at each step, with the hope of finding the global optimum.

**EXAMPLE**

A common example of a greedy algorithm is Activity Selection:

Problem: Given a set of activities with their start and finish times, select the maximum number of activities that don't overlap.

**Greedy Approach:**

* Sort the activities by their finish times.
* Select the first activity that finishes the earliest.
* For each subsequent activity, select it if its start time is after the finish time of the previously selected activity.

Example activities with (start, finish) times:  
Activity 1: (1, 3)  
Activity 2: (2, 5)  
Activity 3: (4, 6)  
Activity 4: (6, 8)  
Activity 5: (5, 9)  
Activity 6: (8, 10)

**Solution:**

* Sort by finish time: Activity 1 (1, 3), Activity 3 (4, 6), Activity 4 (6, 8), Activity 6 (8, 10), Activity 2 (2, 5), Activity 5 (5, 9).
* Select Activity 1 (finish = 3), Activity 3 (start = 4), Activity 4 (start = 6), Activity 6 (start = 8).

Thus, the maximum number of non-overlapping activities is 4 (Activities 1, 3, 4, and 6).

#### ****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:**

* The time complexity for filling the DP table is **O(n⋅W)O(n \cdot W)O(n⋅W)** where:
  + nnn is the number of items.
  + WWW is the maximum weight capacity of the knapsack.

This is because we compute the value for every combination of items and capacities.

**Space Complexity:**

* The space complexity is **O(n⋅W)O(n \cdot W)O(n⋅W)** since we need a 2D table of size n×Wn \times Wn×W to store the results of subproblems.