**1. Recursive Algorithms**

**Description:**  
• **Recursion** is a method where a function calls itself to solve a smaller instance of the same problem.  
• Commonly used in problems with **repetitive patterns**, such as Fibonacci series, tree traversal, etc.  
• Useful for breaking down complex problems into simpler subproblems.

**Example Use Case (Fibonacci for Forecasting):**  
Let’s say we are forecasting future financial values based on a Fibonacci-like pattern:  
F(n) = F(n-1) + F(n-2)  
where F(n) is the forecasted value at year n.

**Recursive Algorithm:**

1. Define a base case (e.g., F(0) and F(1)).
2. For F(n), return F(n-1) + F(n-2).
3. Function calls itself until base case is reached.

**4. Analysis**

**Time Complexity of Recursive Algorithm**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Naive Recursion | O(2ⁿ) |
| Optimized (Memoization) | O(n) |
| Optimized (Dynamic Programming) | O(n) |

Naive recursion recalculates the same values multiple times, leading to exponential time.

**How to Optimize Recursive Solutions**

| **Technique** | **Description** |
| --- | --- |
| **Memoization** | Store previously computed results in a cache (dictionary/array) to avoid redundant calls. |
| **Dynamic Programming (Bottom-Up)** | Build the solution from base cases up, storing results iteratively. |
| **Tail Recursion** | Convert recursion to iteration where supported, to reduce stack space. |
| **Recursive Limits** | Be cautious with recursion depth in large inputs to avoid stack overflow. |

**When to Use Recursion**

| **Situation** | **Recommended Approach** |
| --- | --- |
| Problem has overlapping subproblems | Recursion with memoization |
| Forecast pattern is Fibonacci-like | Recursive with optimization |
| Dataset is small and tree-structured | Pure recursion is fine |
| Performance is critical | Use dynamic programming |