**1.Understand Recursive Algorithms:**

**Explain the Concept of Recursion and and how it can simplify certain problems:**

**Definition:**  
Recursion is a programming technique where a function **calls itself** to solve smaller instances of a problem until it reaches a stopping point known as the **base case**.  
It allows problems to be broken down into simpler sub-problems, making the solution easier to express logically.

**Structure of a Recursive Function**

A recursive function must have:

1. **Base Case** – A condition that ends the recursion.
2. **Recursive Case** – The function calls itself with smaller input.

**General Form:**

returnType function(parameters) {

if (base case condition)

return result;

else

return function(smaller problem);

}

**Simple Example – Factorial**

Factorial of n:  
n! = n × (n - 1)!  
With base case: 0! = 1

int factorial(int n) {

if (n == 0) return 1;

return n \* factorial(n - 1);

}

**How Recursion Simplifies Problems**

1. **Breaks Down Problems**
   * Complex tasks are split into smaller tasks automatically.
   * Easier to express problems like tree traversal, search, and divide-and-conquer.
2. **Cleaner Code**
   * Removes the need for manual loops in problems like permutations, combinations, etc.
3. **Natural Fit for Certain Problems**
   * Examples: Fibonacci, directory traversal, mathematical formulas, DFS in graphs.
4. **Easier to Map Real-World Problems**
   * Recursive thinking reflects natural problem-solving (e.g., finding a path in a maze).

**Real-World Analogy**

Think of a stack of books.  
To reach the bottom book, you remove one by one (recursively), and put them back after reading.  
Each "remove" is a recursive call, and "put back" is returning from the call.

**Use Cases of Recursion**

* Mathematical computations: factorial, Fibonacci, GCD
* Tree/graph algorithms: DFS, postorder traversal
* Searching and sorting: binary search, merge sort
* Parsing: HTML/XML parsing
* Problem-solving: backtracking, puzzle solvers

**How Recursion Can Be Simplified**

Let’s take **Fibonacci** as an example:

int fib(int n) {

if (n <= 1) return n;

return fib(n - 1) + fib(n - 2);

}

**4.Analysis:**

**Discuss the time complexity of your recursive algorithm:**

**Time Complexity of the Recursive Algorithm**

double forecast(int year, double value, double rate) {

if (year == 0) return value;

return forecast(year - 1, value, rate) \* (1 + rate);

}

Then the **time complexity** is:

* **O(n)** — because the function calls itself once per year until it reaches year 0.

There are no repeated calculations, so it’s efficient for small input.  
However, **in other recursive problems** like Fibonacci, the same values may be recomputed multiple times, leading to:

* **O(2ⁿ)** — exponential time in the worst case.

**Explain how to optimize the recursive solution to avoid excessive computation:**

To avoid excessive computation:

**1. Memoization (Top-Down DP):**

* Store results of previous recursive calls in a map or array.
* Avoids redundant calculations.
* Reduces time from **O(2ⁿ)** to **O(n)**.

**2. Dynamic Programming (Bottom-Up DP):**

* Use a loop to build the solution from base to final result.
* No recursion or call stack.
* Time and space: **O(n)**

**3. Tail Recursion (if supported):**

* A special kind of recursion where the recursive call is the last statement.
* Some languages optimize this to avoid extra stack frames.