Exercise 7: Financial Forecasting

**Step 1: Understand Recursive Algorithms**

Recursion

Recursion is a programming technique where a method calls itself to solve a smaller version of the original problem. It continues to break down the problem into simpler subproblems until a base condition is met, at which point the recursive calls begin to resolve.

**Structure of a Recursive Method**

A recursive method has two main components:

1. **Base Case** – The condition that stops the recursion.
2. **Recursive Case** – The part where the method calls itself with a smaller input.

**How Recursion Simplifies Certain Problems**

Recursion simplifies problems that can be broken into smaller, similar subproblems. It is especially useful in cases where the logic is naturally repetitive.

**Examples include:**

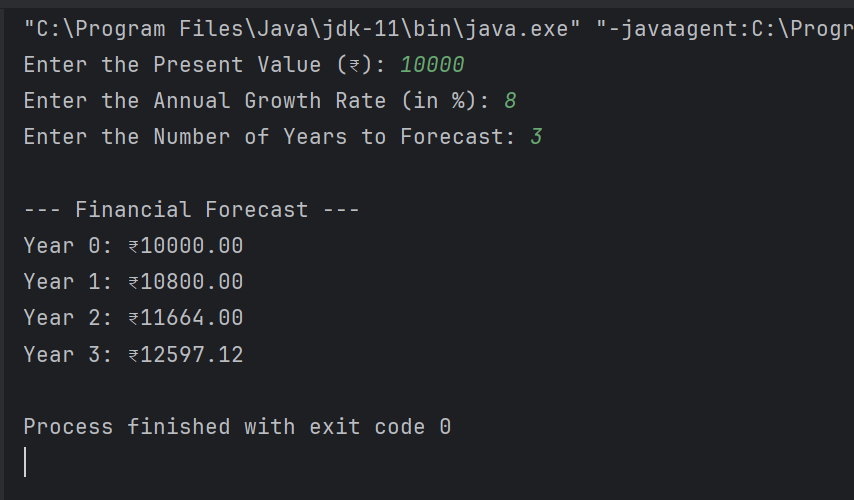
* **Mathematical problems** like factorial or Fibonacci series.
* **Tree or graph traversals**, where recursion mirrors the structure.
* **Divide-and-conquer algorithms** such as merge sort.

By handling one small part of the problem at a time, recursion often results in cleaner and more intuitive code.

**Step 2: Setup and Implementation**

import java.util.\*;  
public class FinancialForecast {  
 public static double calculateFutureValue(double presentValue, double growthRate, int years) {  
 if (years == 0) {  
 return presentValue;  
 } else {  
 return *calculateFutureValue*(presentValue, growthRate, years - 1) \* (1 + growthRate);  
 }  
 }  
  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.print("Enter the Present Value (₹): ");  
 double presentValue = sc.nextDouble();  
 System.*out*.print("Enter the Annual Growth Rate (in %): ");  
 double rate = sc.nextDouble();  
 double growthRate = rate / 100;  
 System.*out*.print("Enter the Number of Years to Forecast: ");  
 int years = sc.nextInt();  
 System.*out*.println("\n--- Financial Forecast ---");  
 for (int i = 0; i <= years; i++) {  
 double value = *calculateFutureValue*(presentValue, growthRate, i);  
 System.*out*.printf("Year %d: ₹%.2f%n", i, value);  
 }  
 }  
}

**OUTPUT:**

****

**Step-3: Analysis**

**Time Complexity of the Recursive Algorithm**

The recursive method calculateFutureValue(presentValue, growthRate, years) performs one recursive call per year until it reaches the base case. Thus, for n years, the function performs n recursive calls.

* **Time Complexity:** O(n)
* **Explanation:** Each call performs a constant-time multiplication and one recursive call. As the recursion depth increases linearly with the number of years, the time complexity is linear.

**Space Complexity**

Each recursive call consumes stack memory, so the space complexity is also linear.

* **Space Complexity:** O(n)
* **Explanation:** For n years, n function calls are pushed onto the call stack before they begin to resolve.

**Optimizing the Recursive Solution**

While the recursive approach is simple and intuitive for small values of n, it may lead to stack overflow for large values due to deep recursion. To Optimize:

1. **Use Iteration Instead of Recursion**  
   An iterative approach eliminates the use of the call stack, making it more memory -efficient. It computes the future value in a simple loop with constant space usage.
   * **Time Complexity:** O(n)
   * **Space Complexity:** O(1)
2. **Memoization**

Memoization is effective when subproblems overlap, such as in the Fibonacci series. In this case, each year's value depends only on the immediately preceding year, so memoization provides no significant benefit.

The recursive algorithm is sufficient for short- to medium-term forecasting where n is small. For larger time horizons or performance-critical applications, an iterative approach is recommended to reduce memory usage and avoid stack overflow.