### **Exercise 7: Financial Forecasting**

#### Step 1: Understand Recursive Algorithms

**Concept of Recursion:** Recursion is a method of solving problems where the solution depends on solutions to smaller instances of the same problem. In a recursive algorithm, a function calls itself with modified parameters to perform a task. Recursion simplifies complex problems by breaking them down into more manageable sub-problems.

**Benefits of Recursion:**

1. Simplifies code for problems that have a natural recursive structure.
2. Makes code more readable and easier to understand.
3. Often used in divide-and-conquer algorithms, dynamic programming, and backtracking.

**Drawbacks of Recursion:**

1. Can lead to high memory usage due to the call stack.
2. Risk of stack overflow if the recursion depth is too large.
3. Often less efficient than iterative solutions due to repeated calculations.

#### Step 2: Setup

We will create a method to calculate the future value using a recursive approach in Java. The method will predict the future value based on past growth rates. We'll assume a simple scenario where the value grows by a fixed percentage each period.

**Step 3: Implementation**

public class FinancialForecasting {

// Method to calculate future value using recursion

public static double calculateFutureValue(double presentValue, double growthRate, int periods) {

// Base case: if no more periods, return the present value

if (periods == 0) {

return presentValue;

}

// Recursive case: calculate future value for the next period

return calculateFutureValue(presentValue \* (1 + growthRate), growthRate, periods - 1);

}

public static void main(String[] args) {

double presentValue = 1000; // Initial value

double growthRate = 0.05; // 5% growth rate

int periods = 10; // Number of periods

double futureValue = calculateFutureValue(presentValue, growthRate, periods);

System.out.println("Future Value: " + futureValue);

}

}

#### **Step 4: Analysis**

**Time Complexity:** The time complexity of the recursive algorithm is O(n), where n is the number of periods. This is because the function calls itself once for each period, leading to n recursive calls.

**Optimizing the Recursive Solution:** To avoid excessive computation, we can use memoization, a technique where we store the results of expensive function calls and reuse them when the same inputs occur again. However, in this simple example, memoization is not necessary because each calculation is independent of previous ones.

For more complex scenarios, like Fibonacci sequence calculation, memoization can be highly effective.

**Example of Memoization in Recursive Fibonacci Calculation:**

**import java.util.HashMap;**

**public class Fibonacci {**

**private static HashMap<Integer, Long> memo = new HashMap<>();**

**public static long fib(int n) {**

**// Check if result is already computed**

**if (memo.containsKey(n)) {**

**return memo.get(n);**

**}**

**// Base cases**

**if (n <= 1) {**

**return n;**

**}**

**// Recursive case**

**long result = fib(n - 1) + fib(n - 2);**

**memo.put(n, result); // Store result in memo**

**return result;**

**}**

**public static void main(String[] args) {**

**int n = 50; // Example input**

**System.out.println("Fibonacci number " + n + " is " + fib(n));**

**}**

**}**

**Conclusion:** Using recursion for financial forecasting can simplify the implementation but be mindful of the potential for stack overflow and high memory usage with deep recursion. For problems with overlapping subproblems, optimize using memoization to improve efficiency.