# Exercise 7: Financial Forecasting

### 

### **Understand Recursive Algorithms**

**Concept of Recursion:**

* **Definition:** Recursion is a technique where a function calls itself to solve a problem. The problem is broken down into smaller instances of the same problem.
* **Base Case:** The simplest instance of the problem that can be solved directly without further recursion.
* **Recursive Case:** The part of the function where the recursion happens, solving a smaller instance of the problem and combining the results to form a solution to the original problem.

**Advantages of Recursion:**

* **Simplifies Complex Problems:** Certain problems are naturally recursive, such as calculating factorials, Fibonacci numbers, or navigating tree structures.
* **Code Clarity:** Recursive solutions can be more intuitive and easier to understand for problems that involve repeated sub-problems.

### **Step 2: Setup**

**Method to Calculate Future Value using a Recursive Approach:** To forecast future values, we can use a simple model where the future value depends on the current value and the growth rate. Let's assume the growth rate is compounded annually.

**Formula:** FutureValue(n)=CurrentValue×(1+GrowthRate)n\text{FutureValue}(n) = \text{CurrentValue} \times (1 + \text{GrowthRate})^nFutureValue(n)=CurrentValue×(1+GrowthRate)n

Where nnn is the number of years into the future.

### **Step 3: Implementation**

**Recursive Method to Predict Future Values:**

public class FinancialForecasting {

// Method to calculate future value recursively

public static double calculateFutureValue(double currentValue, double growthRate, int years) {

// Base case: if years is 0, the future value is the current value

if (years == 0) {

return currentValue;

}

// Recursive case: calculate future value by compounding the growth rate

return calculateFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double currentValue = 1000; // Example current value

double growthRate = 0.05; // Example growth rate (5%)

int years = 10; // Number of years to forecast

double futureValue = calculateFutureValue(currentValue, growthRate, years);

System.out.printf("The future value after %d years is: %.2f\n", years, futureValue);

}

}

### **Step 4: Analysis**

**Time Complexity:**

* **Recursive Time Complexity:** The recursive method provided has a time complexity of O(n)O(n)O(n), where nnn is the number of years. This is because each recursive call processes one year and the function is called recursively nnn times.

**Optimization and Avoiding Excessive Computation:**

* **Memoization:** To avoid recalculating the same values multiple times, we can use memoization. This technique involves storing the results of previous computations and reusing them, which can significantly reduce the number of recursive calls.

**Optimized Approach with Memoization:**

import java.util.HashMap;

import java.util.Map;

public class FinancialForecasting {

private static Map<Integer, Double> memo = new HashMap<>();

// Method to calculate future value using memoization

public static double calculateFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

if (memo.containsKey(years)) {

return memo.get(years);

}

double futureValue = calculateFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

memo.put(years, futureValue);

return futureValue;

}

public static void main(String[] args) {

double currentValue = 1000; // Example current value

double growthRate = 0.05; // Example growth rate (5%)

int years = 10; // Number of years to forecast

double futureValue = calculateFutureValue(currentValue, growthRate, years);

System.out.printf("The future value after %d years is: %.2f\n", years, futureValue);

}

}

**Explanation of Optimization:**

* **Memoization:** By storing the results of previous computations in a HashMap, we avoid redundant calculations, thus reducing the overall time complexity. This approach effectively turns the time complexity from O(n)O(n)O(n) to O(n)O(n)O(n) in practice but with much faster execution due to reduced recursive calls.

**Conclusion:** Recursion can be a powerful tool for problems that have a recursive structure. However, it can lead to excessive computation if not optimized. Memoization is a key technique to enhance performance by storing and reusing results, especially in recursive algorithms where overlapping sub-problems are common.