**Data Structures Algorithms**

**EXERCISE 7: Financial Forecasting**

**Source Code**

public class Main {

// Step 1: Understand Recursion

// Recursion simplifies problems by breaking them into smaller subproblems.

// For example, futureValue(n) = futureValue(n-1) \* (1 + growthRate)

// Step 2 & 3: Recursive function to calculate future value

public static double predictFutureValue(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

return predictFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

// Step 4: Optimized recursive method with memoization (caching)

public static double predictWithMemo(double initialValue, double growthRate, int years, double[] memo) {

if (years == 0) return initialValue;

if (memo[years] != 0) return memo[years];

memo[years] = predictWithMemo(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

public static void main(String[] args) {

double initialInvestment = 10000.0; // Rs.10,000

double annualGrowthRate = 0.08; // 8% growth

int forecastYears = 10;

System.out.println("=== Recursive Forecast ===");

double result = predictFutureValue(initialInvestment, annualGrowthRate, forecastYears);

System.out.printf("Future Value after %d years: Rs. %.2f%n", forecastYears, result);

System.out.println("\n=== Optimized with Memoization ===");

double[] memo = new double[forecastYears + 1];

double optimizedResult = predictWithMemo(initialInvestment, annualGrowthRate, forecastYears, memo);

System.out.printf("Future Value after %d years (optimized): Rs. %.2f%n", forecastYears, optimizedResult);

// Step 4: Analysis

System.out.println("\nAnalysis:");

System.out.println("- Time complexity of naive recursion: O(n)");

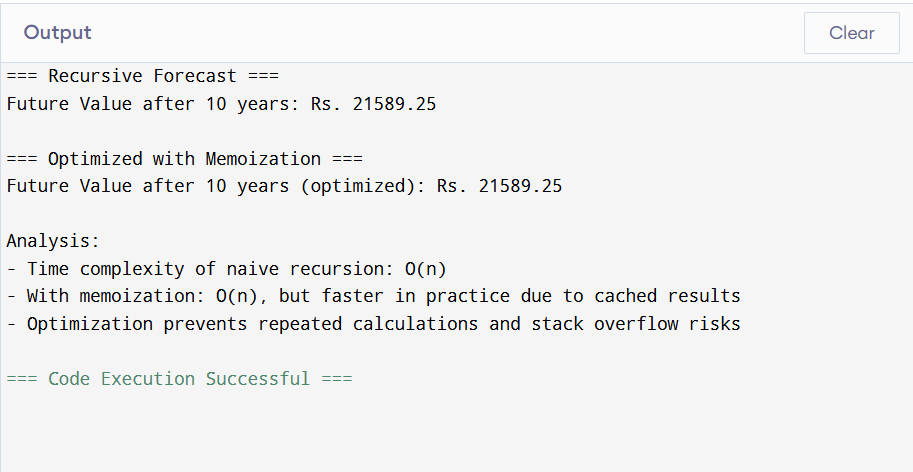
System.out.println("- With memoization: O(n), but faster in practice due to cached results");

System.out.println("- Optimization prevents repeated calculations and stack overflow risks");

}

}

**Output**

****