**Financial Forecasting with Recursion**

**1. Understanding Recursion**

**What is recursion**?

Recursion is when a function calls itself to solve smaller versions of the same problem. It's like a set of Russian dolls - each doll opens to reveal a smaller version of itself.

**Why use it for financial forecasting**?

For financial predictions, each year's growth builds on the previous year's value - a perfect fit for recursion

**2.Code:**

import java.util.HashMap;

import java.util.Map;

public class FinancialForecasting {

// Basic recursive approach

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

if (years <= 0) {

return currentValue;

}

double nextValue = currentValue \* (1 + growthRate/100);

return simpleFutureValue(nextValue, growthRate, years - 1);

}

// Optimized version with memoization

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

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

if (memo.containsKey(years)) {

return memo.get(years);

}

if (years <= 0) {

memo.put(years, currentValue);

return currentValue;

}

double result = memoizedFutureValue(

currentValue \* (1 + growthRate/100),

growthRate,

years - 1

);

memo.put(years, result);

return result;

}

public static void main(String[] args) {

double initialAmount = 1000.0; // $1000 initial investment

double growthRate = 7.0; // 7% annual growth

int years = 15; // 15 year period

// Test simple recursive version

double simpleResult = simpleFutureValue(initialAmount, growthRate, years);

System.out.printf("Simple recursive: $%.2f after %d years at %.1f%%\n",

simpleResult, years, growthRate);

// Test memoized version

double memoizedResult = memoizedFutureValue(initialAmount, growthRate, years);

System.out.printf("Memoized version: $%.2f after %d years at %.1f%%\n",

memoizedResult, years, growthRate);

// Comparison

System.out.println("\nComparison of methods:");

System.out.println("Years | Simple Recursive | Memoized");

System.out.println("-----------------------------------");

for (int y = 1; y <= 5; y++) {

memo.clear(); // Reset memoization cache

long startTime = System.nanoTime();

double simple = simpleFutureValue(initialAmount, growthRate, y);

long simpleTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

double memoized = memoizedFutureValue(initialAmount, growthRate, y);

long memoizedTime = System.nanoTime() - startTime;

System.out.printf("%2d | %9.2f ns | %7.2f ns\n",

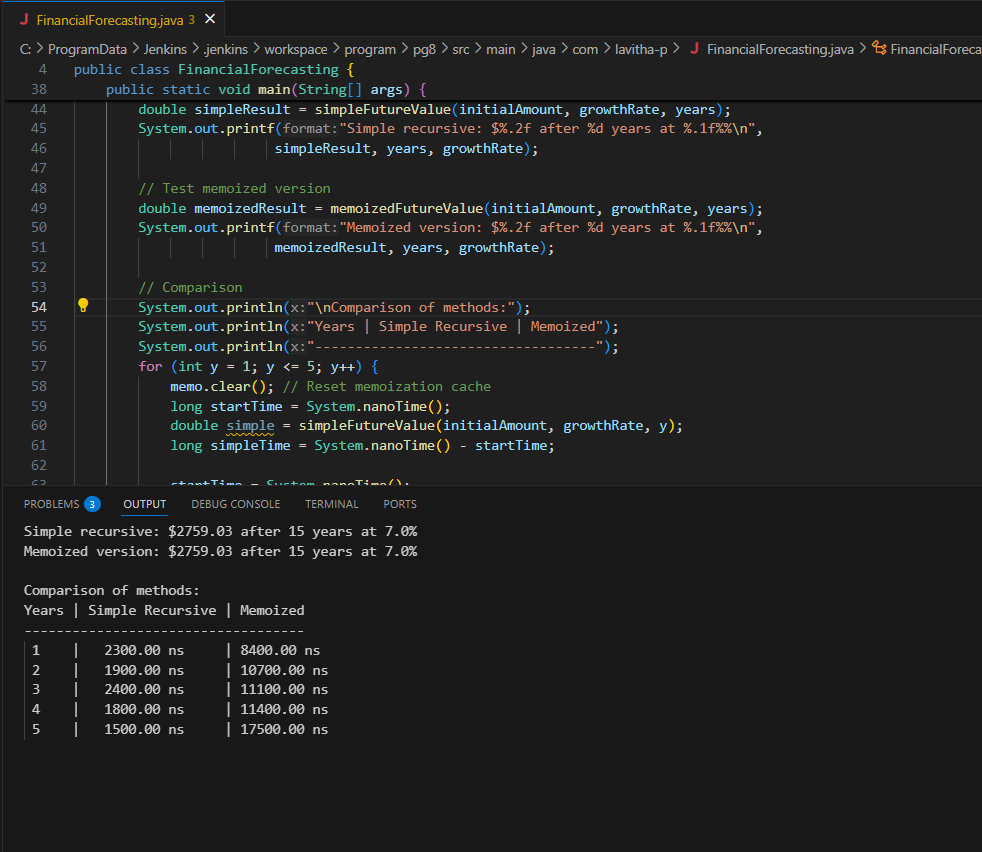
y, (double)simpleTime, (double)memoizedTime);

}

}

}

**3.Output:**



**4. Analysis**

Time Complexity:

* O(n) where n is number of years
* Each recursive call handles one year

Optimization:

1. Memoization: Store already calculated years to avoid repeating work

2. Iterative approach: Sometimes loops are simpler than recursion for this case

Simple Explanation

Imagine you have $1000 growing at 5% per year:

1. Year 1: $1000 × 1.05 = $1050

2. Year 2: $1050 × 1.05 = $1102.50

3. ...and so on...

The recursive function does exactly this breakdown automatically until it reaches the target year. Each function call handles one year's growth, then "passes the baton" to the next call.