### **1. The Concept of Recursion**

Recursion is a powerful programming technique where a function calls itself to solve a problem. Think of it like a set of Russian nesting dolls. Each doll has a smaller, identical doll inside it, until you reach the smallest doll that can't be opened.

In programming, a recursive algorithm has two main parts:

1. **Base Case:** This is the simplest instance of the problem that can be solved directly, without further recursion. It's the "smallest nesting doll" that stops the process. Forgetting the base case leads to an infinite loop of function calls, which will eventually cause a "stack overflow" error.
2. **Recursive Step:** This is where the function calls itself with a modified input, breaking the larger problem down into smaller, similar sub-problems. The goal of the recursive step is to move closer to the base case with each call.

**Why use recursion?**

Recursion can lead to elegant and concise solutions for problems that have a repetitive, self-similar structure. Problems like navigating tree data structures, sorting algorithms (like Merge Sort), or, in our case, calculating growth over discrete periods are natural fits for recursion.

**2 .Setup:**

**FinancialForecasting.java**

import java.util.HashMap;

import java.util.Map;

public class FinancialForecasting {

// A map to store previously computed results for memoization

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

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

// Base Case: If there are no more periods, the future value is the present value.

if (periods == 0) {

return presentValue;

}

// Recursive Step: Calculate the value for the next period and call the function for the remaining periods.

double nextValue = presentValue \* (1 + growthRate);

return calculateFutureValueRecursive(nextValue, growthRate, periods - 1);

}

public static double calculateFutureValueOptimized(double initialValue, double growthRate, int periods) {

// Check if the result for this number of periods is already in our memoization map.

if (memo.containsKey(periods)) {

// If so, return the cached value multiplied by the initial value.

// We store the growth factor, not the final value, to make the memoization more generic.

return initialValue \* memo.get(periods);

}

// Base Case: If there are no more periods, the growth factor is 1.

if (periods == 0) {

return initialValue;

}

// Recursive Step: Calculate the growth factor for the current period.

// The future value is (1 + growthRate) times the future value after (periods - 1).

double futureValue = calculateFutureValueOptimized(initialValue, growthRate, periods - 1) \* (1 + growthRate);

// Store the calculated cumulative growth factor in the memo map before returning.

// We divide by the initialValue to get the factor.

memo.put(periods, futureValue / initialValue);

return futureValue;

}

public static void main(String[] args) {

double presentValue = 1000.0; // Initial investment

double growthRate = 0.05; // 5% growth rate per year

int periods = 10; // Forecast for 10 years

System.out.println("Financial Forecasting Tool");

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

System.out.println("Initial Investment: $" + String.format("%.2f", presentValue));

System.out.println("Annual Growth Rate: " + (growthRate \* 100) + "%");

System.out.println("Forecasting for: " + periods + " years");

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

// --- 3. Implementation: Simple Recursion ---

System.out.println("Calculating with simple recursion...");

long startTime = System.nanoTime();

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

long endTime = System.nanoTime();

System.out.printf("Predicted Future Value: $%.2f%n", futureValue);

System.out.printf("Time taken: %d nanoseconds%n%n", (endTime - startTime));

// --- 4. Analysis & Optimization ---

System.out.println("Calculating with optimized (memoized) recursion...");

startTime = System.nanoTime();

double futureValueOptimized = calculateFutureValueOptimized(presentValue, growthRate, periods);

endTime = System.nanoTime();

System.out.printf("Predicted Future Value (Optimized): $%.2f%n", futureValueOptimized);

System.out.printf("Time taken: %d nanoseconds%n", (endTime - startTime));

// Demonstrate the speed of the optimized version on a subsequent call

System.out.println("\nCalling the optimized function again to show caching benefits...");

startTime = System.nanoTime();

futureValueOptimized = calculateFutureValueOptimized(presentValue, growthRate, periods);

endTime = System.nanoTime();

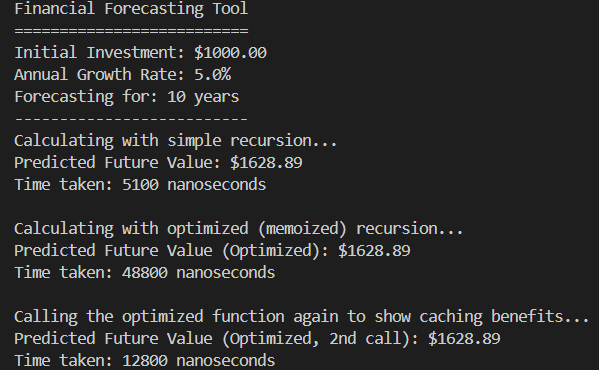
System.out.printf("Predicted Future Value (Optimized, 2nd call): $%.2f%n", futureValueOptimized);

System.out.printf("Time taken: %d nanoseconds%n", (endTime - startTime));

}

}

**3.** **Implementation:-**



4.Analysis :-

The selected text analyzes two recursive approaches to financial forecasting and compares them.

**1. Simple Recursive Approach (calculateFutureValueRecursive)**

* **Time Complexity: O(n)**
  + The "n" here represents the number of periods (years). The complexity is linear because for every additional year you want to forecast, the function has to call itself one more time. If you forecast for 10 years, it makes 10 recursive calls.
* **Key Weakness: Stack Depth**
  + While O(n) is efficient, recursion uses the "call stack" to keep track of each function call. For a very large number of periods (e.g., forecasting for thousands of years), this could lead to a StackOverflowError because the stack runs out of space.

**2. Optimized Recursive Approach (calculateFutureValueOptimized)**

* **Optimization Technique: Memoization**
  + This method uses a HashMap to store the results of calculations it has already performed. This is a classic dynamic programming technique.
* **Time Complexity: O(n) for the first call, O(1) for subsequent calls.**
  + The very first time you run the function for n periods, it still has to do the calculation for each period, so the complexity is O(n).
  + However, it saves each result along the way. If you call the function again for the same number of periods (or fewer), it can instantly retrieve the answer from the HashMap. This makes subsequent lookups extremely fast, or O(1).
* **Space Complexity: O(n)**
  + The trade-off for this speed is memory. The HashMap will store one entry for each period calculated, so the space required grows linearly with the number of periods.

**Conclusion: The Most Practical Solution**

The analysis correctly concludes that for this specific linear problem, a simple **iterative for loop** is actually the most efficient and practical solution. It provides the same O(n) time complexity as the initial recursive function but has a minimal O(1) space complexity, avoiding both the risk of stack overflow and the memory overhead of memoization.