**Module 2 - Data Structures and Algorithms**

**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms**

**What is Recursion?**

Recursion is a programming technique where a method calls itself to solve a smaller subproblem of the main problem. This continues until it reaches a base condition, where the recursion stops.

**Why Recursion?**

Recursion simplifies problems that can be broken down into repeated sub-steps — such as calculating powers, factorial, Fibonacci numbers, etc.

**In our case:**

We use recursion to calculate compound interest for future forecasting:

Formula:  
Future Value = Present Value × (1 + growth rate)^n  
Rather than directly applying the formula, we simulate it step-by-step using recursive calls.

**2.Setup**

We define a method:

static double predictFutureValue(double presentValue, double growthRate, int years)

Where:

* presentValue is the current amount.
* growthRate is the annual percentage increase (as decimal).
* years is how far we forecast into the future.

**3.Implementation**

**AIM:**

To Implement a recursive algorithm to predict future values based on past growth rates.

**CODE:**

import java.util.Scanner;

public class FinancialForecasting {

    static double predictFutureValue(double presentValue, double growthRate, int years) {

        if (years == 0) {

            return presentValue;

        }

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

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the current amount (in RS.): ");

        double currentAmount = sc.nextDouble();

        System.out.print("Enter annual growth rate (in %): ");

        double rate = sc.nextDouble();

        System.out.print("Enter number of years to forecast: ");

        int years = sc.nextInt();

        double growthRate = rate / 100;

        double futureValue = predictFutureValue(currentAmount, growthRate, years);

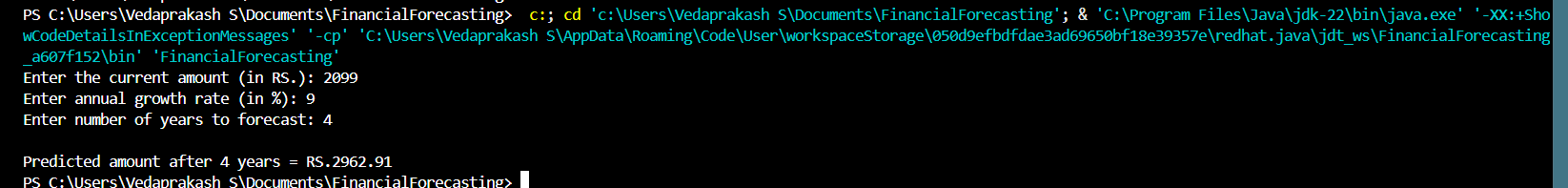
        System.out.printf("\nPredicted amount after %d years = RS.%.2f\n", years, futureValue);

        sc.close();

    }

}

**Output:**



**4. Analysis**

**Time Complexity:**

* Each recursive call reduces years by 1.
* So if years = n, there are n recursive calls.
* Time Complexity: O(n)

**Space Complexity:**

* Since each call is stored in the call stack, space complexity is also O(n)

**How to Optimize the Recursive Solution to Avoid Excessive Computation**

In recursive algorithms, especially ones like in financial forecasting, calling the same function multiple times with slightly changed inputs can consume a lot of memory and processor time. Every recursive call uses stack memory, and too many recursive calls can lead to stack overflow errors or unnecessary computations.

So, to optimize the recursive approach, we can use these methods:

**i). Replace Recursion with Iteration (Most Practical in Java)**

In Java, recursion isn't always the most efficient choice—especially for simple tasks like calculating future value over a period. We can use a loop instead of recursion.

**Example:**

static double predictFutureValue(double presentValue, double growthRate, int years) {

for (int i = 0; i < years; i++) {

presentValue = presentValue \* (1 + growthRate);

}

return presentValue;

}

**Why it's better:**

* Uses no function call stack
* Works fast even for very large input
* Cleaner and easier to understand

**ii). Use Tail Recursion (Only in Some Languages)**

Tail recursion means the recursive call is the last thing done by the function.

static double helper(double presentValue, double growthRate, int years) {

if (years == 0) return presentValue;

return helper(presentValue \* (1 + growthRate), growthRate, years - 1);

}

But in Java, the compiler does not support tail recursion optimization, so it won’t improve much. It's better to use loops instead.

**iii). Use Pre-built Power Function for Better Performance**

If we observe closely, compound interest follows this formula:

Future Value=Present Value×(1+Growth Rate)^Years

So we can directly use:

double futureValue = presentValue \* Math.pow((1 + growthRate), years);

**Why it’s efficient:**

* Internally optimized
* Avoids loops or recursion completely
* Fast for even very large numbers of years