**Exercise 7: Financial Forecasting**

**Recursion**

**Recursion** is when a method calls itself to solve smaller instances of the same problem.  
It simplifies problems like factorial, Fibonacci series, etc., by breaking them down into smaller steps.

**Project Setup**

Recursive method to forecast future values based on:

* **initial amount**
* **growth rate** (percentage)
* **number of years**

**Formula:**  
futureValue(n) = futureValue(n-1) \* (1 + rate)

**FinancialForecast.java**

package javapro;

import java.util.\*;

public class FinancialForecast {

public static double futureValue(double initial, double rate, int years) {

if (years == 0) {

return initial;

} else {

return *futureValue*(initial, rate, years - 1) \* (1 + rate);

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.***in***);

System.***out***.print("Enter the initial amount: ");

double initial = sc.nextDouble();

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

double ratePercent = sc.nextDouble();

double rate = ratePercent / 100;

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

int years = sc.nextInt();

double result = *futureValue*(initial, rate, years);

System.***out***.printf("Future value after %d years: %.2f\n", years, result);

sc.close();

}

}

**OUTPUT:**

Enter the initial amount: 5000

Enter annual growth rate (in %): 10

Enter number of years: 5

Future value after 5 years: 8052.55

**Time Complexity & Optimization**

**🔹 Time Complexity:**

* **Recursive Time Complexity:** O(n) – because it calls itself n times.
* **Space Complexity:** O(n) – due to recursive call stack.

**Optimization**

**1. Convert to Iterative Approach (Best for performance)**

Recursion builds up a stack of calls, which is slow and risky for large inputs.  
Iterative version avoids this by using a loop:

public static double futureValueIterative(double initial, double rate, int years) {

double result = initial;

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

result \*= (1 + rate);

}

return result;

}

* **Time Complexity:** O(n)
* **Space Complexity:** O(1) → Much better than recursion's O(n)

**2. Use Memoization (store previous results)**

Store computed results in a map/array to avoid recalculating.

import java.util.HashMap;

public class FinancialForecastMemo {

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

public static double futureValue(double initial, double rate, int years) {

if (years == 0) return initial;

if (memo.containsKey(years)) return memo.get(years);

double result = futureValue(initial, rate, years - 1) \* (1 + rate);

memo.put(years, result);

return result;

}

}

* **Time Complexity:** O(n)
* **Space Complexity:** O(n) due to memoization and recursion stack