**Exercise 7:Financial Forecasting**

**Recursion**

**Concept:**  
Recursion is a programming technique where a function calls itself to solve a problem by breaking it into smaller sub-problems. It involves two main parts:

* **Base Case**: The condition under which recursion stops.
* **Recursive Case**: The part of the function where the method calls itself with a reduced problem.

### ****Example: Factorial Using Recursion****

public class FactorialExample {

public static int factorial(int n) {

if (n == 0 || n == 1) return 1; // Base case

return n \* factorial(n - 1); // Recursive case

}

public static void main(String[] args) {

System.out.println("Factorial of 5 is: " + factorial(5));

}

}

Output:

Factorial of 5 is: 120

### ****Financial Forecasting Using Recursion****

**Scenario:**  
Predict the future value of an investment using past data and a fixed annual growth rate.

**Recursive Formula:**  
FutureValue(P, R, Y) = P × (1 + R) recursively applied for Y years.

### ****1. Recursive Method without Memoization****

public class FinancialForecast {

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

if (years == 0) return principal; // Base case

return futureValue(principal \* (1 + rate), rate, years - 1); // Recursive call

}

public static void main(String[] args) {

double result = futureValue(10000, 0.1, 3);

System.out.printf("Future Value: %.2f", result);

}

}

Output:

Future Value: 13310.00

### ****2. Recursive Method with Memoization (Optimized)****

import java.util.HashMap;

public class FinancialForecastMemo {

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

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

if (years == 0) return principal;

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

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

memo.put(years, result);

return result;

}

public static void main(String[] args) {

System.out.printf("Future Value: %.2f", futureValue(10000, 0.1, 3));

}

}

Output:

Future Value: 13310.00

**3. Iterative Method (Alternative)**

public class FinancialForecastIterative {

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

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

principal \*= (1 + rate);

}

return principal;

}

public static void main(String[] args) {

System.out.printf("Future Value: %.2f", futureValue(10000, 0.1, 3));

}

}

**4. Formula-Based Method (Most Efficient)**

public class FinancialForecastFormula {

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

return principal \* Math.pow(1 + rate, years);

}

public static void main(String[] args) {

System.out.printf("Future Value: %.2f", futureValue(10000, 0.1, 3));

}

}

### ****Time Complexity Comparison****

| **Method** | **Time Complexity** | **Space Complexity** | **Notes** |
| --- | --- | --- | --- |
| Recursion | O(n) | O(n) | Simple but can overflow stack |
| Memoized | O(n) | O(n) | Efficient, avoids redundant calls |
| Iterative | O(n) | O(1) | Efficient and stack-safe |
| Formula | O(1) | O(1) | Fastest if precision is acceptable |

### ****When to Use Each Method****

| **Approach** | **Use Case** |
| --- | --- |
| Recursion | Educational/demo purposes with small input |
| Memoized | Repeated computations with overlapping subproblems |
| Iterative | General-purpose and large input datasets |
| Formula | Financial modeling with large time spans |