**WEEK – 1 DATA STRUCTURES AND ALGORITHMS**

**EXERCISE 7: FINANCIAL FORECASTING**

**SCENARIO:**

You are developing a financial forecasting tool that predicts future values based on past data.

**RECURSION:**

Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem. It continues until a base case is reached, which terminates the recursive calls.

It typically includes:

**A \*base case** – which ends the recursion

**A \*recursive case** – where the function calls itself with modified input

**Example (Factorial):**

java

int factorial(int n) {

if (n == 0) return 1

return n \* factorial(n - 1);

}

**HOW RECURSION WORKS:**

Recursion helps break complex problems into simpler sub-problems.

It’s ideal when:

* A problem has repetitive structure
* Each step builds on previous results
* The depth is manageable (not extremely large)
* Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. It simplifies complex problems by breaking them down into simpler subproblems, often with a clear base case to end the calls. This approach is especially useful in tasks like factorial, Fibonacci, and tree traversal.

**2) SETUP:**

The aim to calculate the future value (FV) of an investment using a growth rate over a number of years.

Formula (Compound Interest-style):

FV = P \* (1 + r)^n

Where:

P = Principal (initial amount)

r = Annual growth rate (e.g., 0.05 for 5%)

n = Number of years

**Recursive Version of Formula:**

FV(n) = FV(n - 1) + (1 + r)

FV(0) = P(Base Case)}

This lends itself naturally to recursion, where we reduce the number of years step by step until we reach 0.

**3) IMPLEMENTATION:**

java

public class FinancialForecast {

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

if (years == 0) {

return principal;

}

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

}

public static void main(String[] args) {

double principal = 10000;

double rate = 0.05

int years = 5;

double result = futureValue(principal, rate, years);

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

}

**Output:**

Future Value after 5 years: 12762.82

How It Works:

Year 5 = FV(4) × 1.05

Year 4 = FV(3) × 1.05

Year 0 = Principal (base case)

**4) ANALYSIS:**

**TIME COMPLEXITY:**

Each recursive call reduces years by 1 until it hits 0.

So, the total number of recursive calls is:

* + - Time Complexity = O(n), where n is the number of years
    - Space Complexity = O(n), due to the call stack (each recursive call is stored in memory)

**OPTIMIZING THE RECURSIVE SOLUTION:**

**CASE 1:** **USE ITERATION INSTEAD OF RECURSION**

An **iterative approach** performs the same task without recursion and is more memory-efficient.

java

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

double result = principal;

for (int i = 1; i <= years; i++) {

result \*= (1 + rate);

}

return result;

}

**Time Complexity**: O(n)

**Space Complexity**: O(1)

**Case 2: Use Exponentiation**

If you directly compute the power:

java

FV = P \* Math.pow((1 + r), n);

This uses **built-in optimized exponentiation** and avoids recursion altogether.

**Time Complexity**: Often O(log n) internally

**Use Case:** Best for high-performance applications