Exercise 7: Financial Forecasting

Scenario:

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

Steps:

1. **Understand Recursive Algorithms:**
   - Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   - Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   - Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   - Discuss the time complexity of your recursive algorithm.
   - Explain how to optimize the recursive solution to avoid excessive computation.

→ 1. Understand Recursive Algorithms

What is Recursion?

Recursion is a technique where a method calls itself to solve a problem by breaking it into smaller sub-problems.

Benefits of Recursion:

- Simplifies complex problems (e.g., factorial, Fibonacci, tree traversal).
- Reduces code size for problems that follow a repetitive or divide-and-conquer approach.

Drawbacks:

- Can lead to **stack overflow** if not managed well.
- **Inefficient** if overlapping subproblems aren't cached (memoization).

2. Setup: Method to Predict Future Value

We’ll calculate the **future value** recursively using the formula:

FV = PV × (1 + r)^n

Where:

- FV = future value
- PV = present value
- r = growth rate per period
- n = number of periods

3. Implementation (Java)

Recursive Method (Basic Version)

```
public class FinancialForecast {

    // Recursive method to calculate future value

    public static double futureValue(double presentValue, double rate, int periods) {

        if (periods == 0) {

            return presentValue;

        }

        return (1 + rate) * futureValue(presentValue, rate, periods - 1);

    }

    public static void main(String[] args) {

        double pv = 1000.0;   // Present Value

        double rate = 0.05;   // 5% annual growth rate

        int n = 5;            // Forecast for 5 years

        double fv = futureValue(pv, rate, n);

        System.out.printf("Future Value after %d years: %.2f\n", n, fv);

    }

}
```

4. Analysis

◆ **Time Complexity**

- **Recursive Calls:** O(n) — one call for each period
- **Space Complexity:** O(n) — due to recursive call stack

◆ **Optimization Techniques**

Tail Recursion (where applicable)

- Not useful here since the multiplication occurs after the recursive call.

Memoization (not needed here)

- There's no overlapping subproblem, so memoization doesn’t help in this simple use case.

Convert to Iterative (if performance critical)

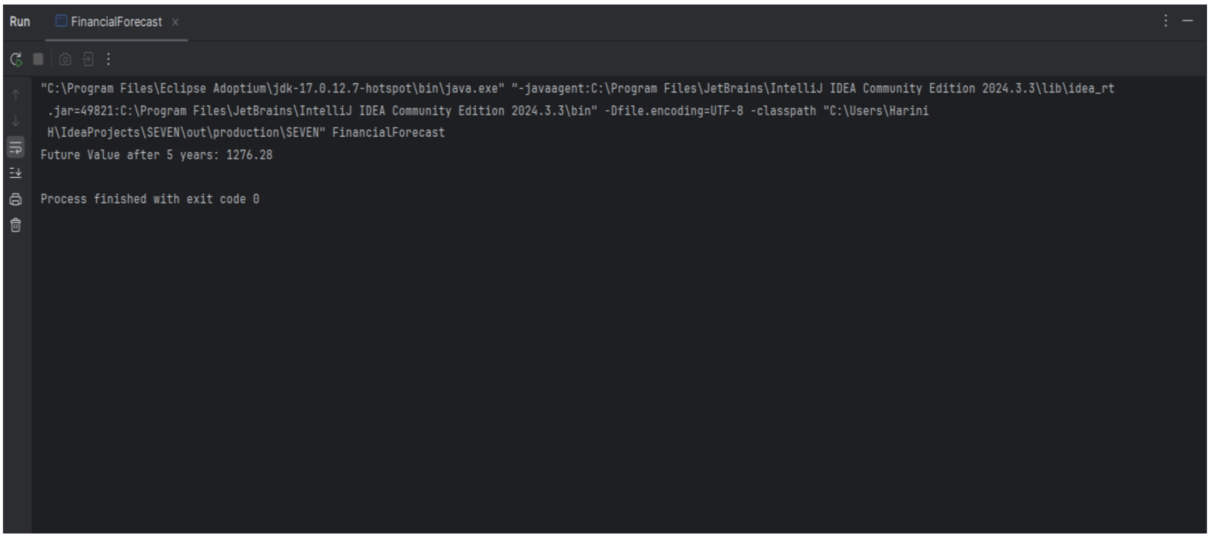
```
public static double futureValueIterative(double presentValue, double rate, int periods) {

    double result = presentValue;
    for (int i = 0; i < periods; i++) {
        result *= (1 + rate);
    }
    return result;
}
```

- Time Complexity: O(n)
- Space Complexity: O(1) — more memory-efficient

Summary

Aspect	**Recursive Method**	**Iterative Method**
Code Simplicity	✅ Elegant & clean	More lines of code
Stack Usage	❌ Uses stack (O(n))	✅ Constant memory
Performance	Good for small n	✅ Better for large n

OUTPUT:



```
"C:\Program Files\Eclipse Adoptium\jdk-17.0.12.7-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\lib\idea_rt
 .jar=49821:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.3\bin" -Dfile.encoding=UTF-8 -classpath "C:\Users\Harini
 H\IdeaProjects\SEVEN\out\production\SEVEN" FinancialForecast
Future Value after 5 years: 1276.28

Process finished with exit code 0
```