Exercise – 7 Financial Forecasting

(Step-1) What is Recursion?

Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem.

Every recursive function has:

1. Base Case – the stopping condition that ends the recursion.
2. Recursive Case – where the function calls itself with a smaller/simpler input.
3. Real-Life Example:
   Imagine standing in a line of people. To count the number of people, you ask the person in front of you,
   "How many people are in front of you?" and they ask the same question to the person in front of them — until the first person says,
   "No one is in front of me." That's the base case, and the count is built up backward.

Why is Recursion Useful?

Recursion is powerful because it breaks down complex problems into smaller subproblems, especially when the problem has a repetitive or nested structure.

It simplifies code for problems like:

- Factorials
- Fibonacci sequence
- Tree/graph traversal
- Financial forecasting (e.g., compounding over time)
- Backtracking algorithms (like maze solving, permutations)

Example code:-

```
public int factorial(int n) {

   if (n == 0) return 1;

   return n * factorial(n - 1);   }
```

(Step – 2) Create a method to calculate the future value using a recursive approach.

We will create a method to recursively calculate future value (FV) using the formula:

$FV = PV \times (1 + r)^n$

Where:

- PV = Present Value
- r = growth rate (decimal)
- n = number of years (or time periods)

Code:-

```
public static double futureValue(double presentValue, double rate, int years) {
      // Base case: 0 years means value stays the same
      if (years == 0) {
         return presentValue;
      }
      // Recursive case: calculate for one less year
      return (1 + rate) * futureValue(presentValue, rate, years - 1);
   }
```

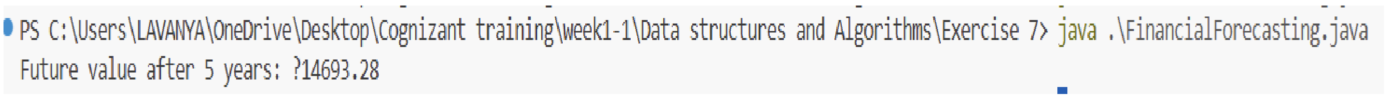
(Step-3) Implement a recursive algorithm to predict future values based on past growth rates.

```
public class FinancialForecasting {
   public static double futureValue(double presentValue, double rate, int years) {
      if (years == 0) {
         return presentValue;
      }
      return (1 + rate) * futureValue(presentValue, rate, years - 1);
   }
```

```
    public static void main(String[] args) {
        double presentValue = 10000;  // ₹10,000 initial investment
        double annualGrowthRate = 0.08;  // 8% growth
        int years = 5;

        double forecastedValue = futureValue(presentValue, annualGrowthRate,
years);
        System.out.printf("Future value after %d years: ₹%.2f%n", years,
forecastedValue);
    }
}
```

Output:-



```
PS C:\Users\LAVANYA\OneDrive\Desktop\Cognizant training\week1-1\Data structures and Algorithms\Exercise 7> java .\FinancialForecasting.java
Future value after 5 years: ?14693.28
```

(Step-4) Time Complexity of the Recursive Algorithm

The recursive algorithm for financial forecasting calculates the future value by multiplying the present value with the growth rate for each year. It does this by calling itself once for each year, reducing the year count by one in every call, until it reaches zero.

- Since the function is called once for each of the n years, the time complexity is O(n), where n is the number of years.
- Each call does a single multiplication and passes the result to the next call, so there are no repeated or nested calls, keeping it linear.

How to Optimize the Recursive Solution

While recursion simplifies the logic, it can become **inefficient or risky** for large values of n due to high memory usage or stack overflow errors. Here are ways to optimize it:

1. **Convert to Iterative Approach:**
   - Replace recursion with a loop.
   - This avoids the function call stack entirely.
   - Time complexity remains **O(n)** but **space complexity improves to O(1)**.
   - This is the most memory-efficient version.
2. **Use Direct Mathematical Formula:**
   - Apply the compound growth formula directly using a built-in power function.
   - This approach computes the result in **constant time**.
   - **Time complexity becomes O(1)** and **space complexity is also O(1)**.
   - This is the **most efficient** method and is preferred in real-world forecasting tools.
3. **Memoization (only for overlapping subproblems):**
   - If the same year’s result is used multiple times (e.g., in more complex models like trees), caching results can save redundant work.
   - Memoization reduces **time complexity** significantly in such cases but is not needed in simple linear forecasting.