**Exercise 7: Financial Forecasting**

**Scenario:**

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.

**SOLUTION**

**Step 1: Understand Recursive Algorithms**

**What is recursion?**

Recursion is a technique where a method calls itself to solve smaller versions of the same problem. It keeps reducing the size of the problem until it reaches a base case, at which point it stops calling itself.

Example from daily life:

Imagine stacking books:

* You place a book.
* Then place another on top.
* And another… until a condition is met (say, 5 books).  
  Then, you unstack one at a time.

That’s how recursive calls go deeper and then return one by one — like stacking and unstacking.

**Why recursion is useful for forecasting:**

Forecasting involves repeated patterns:

* If I have ₹10,000 today,
* And it grows 10% each year,
* Then next year it’s ₹11,000,
* Then ₹12,100, and so on.

This is repetitive and can be broken into smaller problems

**Step 2: Setup**

**Objective:**  
Create a method that calculates the **future value** of an amount using a **recursive approach**.

The future value of an investment or amount is typically calculated using the **compound interest formula**:

Future Value = Present Value × (1 + Growth Rate) ^ Years

Instead of using the power operator, we can use **recursion** to simulate the repeated multiplication year by year.

**Recursive Idea:**

We break the problem into smaller sub-problems using the following logic:

* If the number of years is 0, we return the **current value** (base case).
* Otherwise, we:
  + Multiply the currentValue by (1 + growthRate)
  + Recurse with years - 1

This pattern repeats until the number of years reduces to 0.

**Method Signature:**

public static double forecastValue(double currentValue, double growthRate, int years)

* **currentValue** – the present value (e.g., ₹10,000)
* **growthRate** – the annual growth rate in decimal (e.g., 8% = 0.08)
* **years** – number of years into the future to calculate

**Purpose of the Method:**

This method will form the **core logic** of the forecasting tool. It will be used to compute the future value of money or assets over time, assuming a consistent annual growth rate, using a **recursive algorithm**.

**Step 3: Implementation**

**Objective:**  
Implement the recursive algorithm to calculate the future value over a given number of years.

**Java Code:**

public class ForecastingTool {

public static double forecastValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

} else {

double nextYearValue = currentValue \* (1 + growthRate);

return forecastValue(nextYearValue, growthRate, years - 1);

}

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.08;

int years = 5;

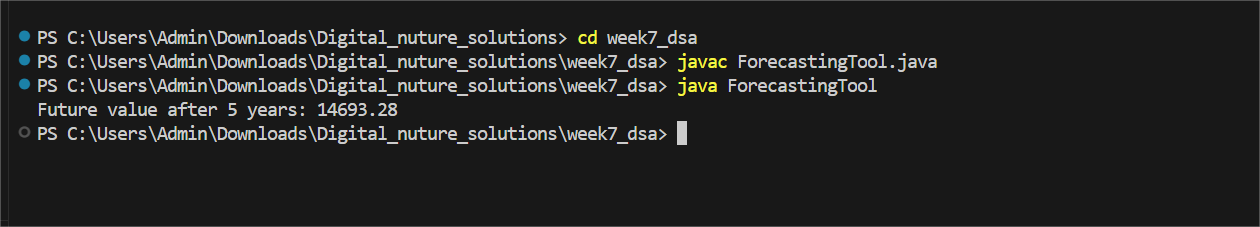
double futureValue = forecastValue(initialValue, growthRate, years);

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

}

}

**Sample Output**



**Explanation:**  
Starting from ₹10,000, the value is increased by 8% each year. The method calculates the value recursively for 5 years, returning the final projected value.

**Step 4: Analysis**

**🔹 Time Complexity:**

The method makes one recursive call for each year. So if you forecast for n years, the method calls itself n times.

* **Time Complexity:** O(n)  
  (where n is the number of years)

This is linear time, meaning the computation time increases proportionally with the number of years.

**🔹 Space Complexity:**

Each recursive call is stored in the **call stack** until it completes. So for n years, the call stack holds n frames at most.

* **Space Complexity:** O(n)

This can lead to **stack overflow** errors if n is very large.