Exercise 7: Financial Forecasting

1. Understanding the Problem

In a financial forecasting tool, the goal is to predict future values such as income, expenses, or stock prices based on past trends. One effective way to build such a tool is by using **recursive algorithms**, which break down problems into smaller, repeatable components.

Recursion is a natural fit for problems where a future value depends on one or more previous values — a common case in financial models like the **Fibonacci sequence**, **compound interest projections**, or **trend-based forecasting**.

Why Is Recursion Important in Forecasting?

➤ Solves Problems with Repeating Patterns

When a forecast value is derived from previous values using a formula, recursion helps model this directly and clearly.

➤ Simplifies Code

Instead of writing lengthy loops, a recursive function calls itself with updated parameters, making the logic more readable and easier to maintain.

➤ Matches Real-World Financial Formulas

Many forecasting models are naturally recursive, where the next month's value depends on the previous month's result. Recursion fits these patterns directly.

2. Understanding Recursive Algorithms

A **recursive algorithm** is a function that calls itself to solve smaller instances of a problem until it reaches a base case that stops the recursion.

Example: Recursive Forecast Formula

forecast the nth month's revenue like this:
revenue(n) = revenue(n - 1) + growthRate

This means each month's revenue builds on the last. In Java-like pseudocode:

```
int forecast(int month) {
    if (month == 1) return baseRevenue;
    return forecast(month - 1) + growthRate;
}
```

This small function captures the logic in a concise and natural way.

---

3. Time Complexity Analysis

Recursive algorithms can vary widely in performance depending on how they are implemented.

➤ Simple Recursion (Without Caching)

If the function repeatedly recalculates values without saving results, it can be **very inefficient**.
For example, a naive Fibonacci-like forecast has exponential time complexity: **O(2^n)**.

➤ Tail Recursion or Memoization

If we save already computed results (called **memoization**), or rewrite the recursion to avoid redundant calls, we can bring the complexity down to **O(n)** — much more efficient.

```
int[] memo = new int[n+1];


int forecast(int month) {
   if (month == 1) return baseRevenue;
   if (memo[month] != 0) return memo[month];
   memo[month] = forecast(month - 1) + growthRate;
   return memo[month];
}
```

This avoids recalculating values and speeds up the algorithm significantly.

---

4. Optimizations and Best Practices

1. **Define a Clear Base Case**
   Every recursive algorithm must stop somewhere. Without a proper base case, the function will run forever and cause a stack overflow.
2. **Use Memoization for Repeated Subproblems**
   When forecasting involves overlapping sub-calculations, store results to avoid duplicate computation.

3. **Switch to Iteration for Large Datasets**

In some cases, recursion may not be suitable for very large forecasts due to stack limitations. Converting the logic to a **loop-based** approach can improve stability.

4. **Keep It Readable and Testable**

Wrap the recursive logic in a class like ForecastEngine and provide a clean method like getForecast(month) to make testing and reuse easier.