**Exercise 7: Financial Forecasting**

**Scenario**

We are developing a financial forecasting tool that predicts the future value of an investment based on past data using a recursive approach. The primary input from the user is the initial value (e.g., an investment amount) and the expected annual growth rate. The output is the projected value for a specified future year.

**Understanding Recursive Algorithms**

**Recursion** is a technique where a method calls itself to solve smaller instances of a problem until a base condition is met. In forecasting, recursion simplifies the repeated calculation of value updates over each year by expressing the problem in terms of the previous year’s result.

Value(year) = Value(year - 1) × (1 + growthRate / 100)

This recursion continues until the target year equals the current year (base case), where the initial value is returned.

**Setup**

A Java class FinancialForecast is created with the following:

* A growthRate field to store annual percentage growth.
* Validation logic for user input.
* A recursive method recursiveForecast(targetYear, initialValue) to compute the forecast.
* The current year is dynamically retrieved using LocalDateTime.now().getYear().

**Implementation**

The key method in the class is:

public static double recursiveForecast(int targetYear, double initialValue, double growthRate) {

if (targetYear <= CURRENT\_YEAR) {

return initialValue;

}

return recursiveForecast(targetYear - 1, initialValue, growthRate) \* (1 + growthRate / 100);

}

This method recursively reduces the target year until the base year is reached and applies compound growth in reverse (bottom-up) during recursion unwinding.

Validation is added for:

* Negative or unrealistically low growth rates.
* Negative initial investment.
* Target year before the current year.

**Analysis**

**Time Complexity**  
The recursive function has:

* **Time Complexity:** O(n) where n = targetYear - CURRENT\_YEAR.  
  Each recursive call represents one year’s growth calculation.
* **Space Complexity:** Also O(n) due to call stack usage for each recursive call.

**Optimization Discussion**  
Since there are **no overlapping subproblems**, this problem is not a good fit for memoization or dynamic programming.  
However, recursion helps in writing clean and readable code. For performance in real-world applications, this can be refactored into an **iterative or closed-form solution**, like:

futureValue = initialValue \* Math.pow((1 + growthRate / 100), years);

But since the goal is to practice recursive thinking, the implemented approach serves well.

**Conclusion**

This exercise demonstrates how recursion can model financial growth in a clean and expressive way. While it may not be the most optimal in terms of raw performance, it aligns with the problem’s requirement to explore recursive problem-solving and reinforces core programming principles effectively.