**Exercise 7: Financial Forecasting**

**1. Understanding Recursive Algorithms**

**Recursion**

Recursion is a programming technique where a function calls itself in order to solve smaller instances of the same problem. This approach is often used when a problem can be divided into similar subproblems, making it easier to solve through repeated self-invocation.

**How Recursion Works**

1. **Base Case:** The condition under which the recursion ends. This prevents infinite loops by specifying when the function should stop calling itself.
2. **Recursive Case:** The part of the function where it calls itself with a smaller or simpler subproblem.

**Advantages of Recursion**

* **Simplicity:** Often leads to shorter and more elegant solutions.
* **Natural Fit:** Ideal for problems that can be naturally divided into similar subproblems (e.g., factorial calculation, Fibonacci sequence).

**Disadvantages of Recursion**

* **Performance Overhead:** Recursive calls can lead to excessive memory usage and stack overflow for deep recursion.
* **Complexity:** May be harder to understand for some problems and can lead to inefficiency without proper base cases.

**2. Analysis**

**Time Complexity**

* **Recursive Algorithm:**
  + Time Complexity: O(n)
  + Reason: The recursive function is called once per period, leading to a linear growth in calls.
  + Space Complexity: O(n) due to the recursive call stack.
* **Iterative Algorithm (for comparison):**
  + Time Complexity: O(n)
  + Reason: The iterative loop also runs once per period.

**Optimization Techniques**

Recursive algorithms can lead to performance issues if not optimized properly. Here are some ways to optimize recursive solutions:

1. **Tail Recursion:**
   * Tail recursion is a type of recursion where the recursive call is the last operation in the function. Java can optimize tail-recursive functions, but it doesn’t do so automatically as some other languages do (like Scala). However, understanding tail recursion helps design more efficient algorithms.
   * In our example, the calculateFutureValueRecursive method already exhibits tail recursion because the recursive call is the last operation.
2. **Memoization:**
   * Store previously computed results to avoid redundant calculations.
   * Memoization is more suitable for problems like the Fibonacci sequence or other dynamic programming problems where overlapping subproblems exist.
3. **Iterative Approach:**
   * Converting recursion to iteration can avoid stack overflow issues for deep recursion.
   * In our example, the calculateFutureValueIterative method shows an iterative approach that provides the same result without recursion.