# Exercise 7: Financial Forecasting

## 1. Understand Recursive Algorithms

**Concept of Recursion:**

- Definition: Recursion is a technique where a function solves a problem by calling itself with smaller instances of the same problem. It's particularly useful for problems that can be divided into similar subproblems.

**- Base Case and Recursive Case:**

- Base Case: The simplest instance of the problem that can be solved directly without further recursion. It provides a stopping condition for the recursion.

- Recursive Case: The part of the function where it calls itself with a modified argument, working towards the base case.

**- Advantages:**

- Simplicity: Recursive solutions can be more intuitive and easier to understand, especially for problems involving hierarchical structures or repeated patterns.

- Reduction of Code: Often results in more concise code compared to iterative solutions.

**- Drawbacks:**

- Stack Overflow: Deep recursion can lead to stack overflow errors if the recursion depth is too large.

- Performance Issues: Recursive solutions can be less efficient due to repeated calculations unless optimized.

## 2. Setup

**Objective:** Create a method to calculate the future value of an investment using recursion.

**Assumptions:**

- `initialValue`: The initial amount of money.

- `growthRate`: The rate of growth per period (e.g., 5% would be 0.05).

- `periods`: The number of future periods to forecast.

## 3. Implementation

**Recursive Algorithm to Predict Future Values:**

**calculateFutureValue:** Recursively calculates the future value of an investment given an initial value, growth rate, and number of periods. Each call multiplies the initial value by (1 + growthRate) and decrements the period count until periods reach zero (base case).

**Iterative Approach for Comparison:calculateFutureValueIterative:** Iteratively calculates the future value of an investment by applying the growth rate for a specified number of periods. It multiplies the initial value by (1 + growthRate) in a loop for each period.

## 4. Analysis

**Time Complexity:**

**- Recursive Algorithm:**

- Time Complexity: O(n), where `n` is the number of periods. Each recursive call reduces the number of periods by 1, leading to `n` recursive calls.

- Space Complexity: O(n) for the recursion stack space, as each call adds a new frame to the stack.

**- Iterative Approach:**

- Time Complexity: O(n), as it involves a single loop that iterates `n` times.

- Space Complexity: O(1), as it uses a constant amount of space regardless of the number of periods.

**Optimization:**

- Memoization: Can be used to store intermediate results and avoid redundant calculations, but it's not particularly necessary for this straightforward calculation unless the problem becomes more complex.

- Iterative Approach: Generally more practical for problems like this with a simple recurrence relation. It avoids the overhead of recursive function calls and is more efficient in terms of time and space complexity.

**Explanation:**

- Recursive Approach: While elegant and potentially easier to understand, it might not be the most efficient for this particular problem due to linear time complexity and the potential for stack overflow with deep recursion.

- Iterative Approach: More practical for problems with a straightforward recurrence relation. It avoids the overhead associated with recursion and is generally more efficient in terms of both time and space complexity.

By comparing both approaches, you can choose the method that best fits the needs of your application and the constraints of your environment.