**Exercise 7: Financial Forecasting**

**Scenario:**

You are 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.

**Recursion**: A programming technique where a method calls itself to solve a problem. It typically involves:

* **Base Case**: The condition where recursion stops, providing a simple solution.
* **Recursive Case**: The method calls itself with modified arguments to solve the problem progressively.

**Advantages**:

* **Simplifies Complex Problems**: Ideal for problems with a natural recursive structure (e.g., tree traversals, factorial calculations).
* **Cleaner Code**: Reduces the need for complex loops, enhancing readability.

**Challenges**:

* **Stack Overflow**: Excessive recursion depth can consume stack space, leading to stack overflow errors.
* **Performance Overhead**: Multiple function calls and context switching can degrade performance.

1. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
2. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
3. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Time Complexity Analysis of Recursive Algorithm**:

* **Recursive Calls**: The function fibonacci(n) makes two recursive calls: fibonacci(n - 1) and fibonacci(n - 2). This branching leads to an exponential growth in the number of calls.
* **Number of Calls**: The number of calls grows exponentially because each call results in two more calls. The total number of calls is roughly proportional to the value of the Fibonacci number being computed, which is approximately 2^n.

**Overall Time Complexity**:

* The time complexity of this recursive Fibonacci algorithm is **O(2^n)**. This exponential growth makes it very inefficient for large values of n.

**Optimized Example: Using Memoization**

To improve efficiency, we can use memoization to store already computed Fibonacci numbers:

**Time Complexity Analysis with Memoization**:

* **Recursive Calls**: Each unique value of n is computed only once.
* **Work per Call**: Each call does a constant amount of work if the result is already cached.

**Overall Time Complexity**:

* With memoization, the time complexity is reduced to **O(n)**. This is because each Fibonacci number from 0 to n is computed once and then cached.

**Finally,**

**Non-Memoized Fibonacci**:

* **Time Complexity**: O(2^n)

**Memoized Fibonacci**:

* **Time Complexity**: O(n)