**1. Understanding Recursive Algorithms:**

**Concept of Recursion**

Recursion is a programming technique where a function calls itself to solve a problem. A recursive function typically has two parts:

1. **Base Case**: The condition under which the recursion ends. It provides the simplest, smallest instance of the problem, which can be solved directly without further recursion.
2. **Recursive Case**: The part of the function that breaks down the problem into smaller, more manageable sub-problems and then calls itself to solve these sub-problems.

Recursion can simplify problems that can be divided into similar sub-problems, making the solution more intuitive and concise. However, it can also lead to excessive computation and stack overflow if not implemented carefully.

**4. Analysis:**

**Time Complexity**

The time complexity of this recursive algorithm is O(n), where n is the number of periods. This is because the function calls itself n times, decrementing the periods each time until it reaches 0.

**Optimization: Avoiding Excessive Computation**

One of the main issues with simple recursion is that it can lead to excessive recomputation. In this case, each recursive call computes the future value from scratch, even if it has been calculated previously. To optimize this, we can use **memoization** or **dynamic programming** to store previously computed results and reuse them.

**Benefits of Optimization**

* **Reduced Time Complexity**: With memoization, each unique computation is done only once, potentially reducing the time complexity to O(n) from what could otherwise involve a lot of redundant calculations.
* **Improved Efficiency**: By storing intermediate results, the algorithm reduces the number of recursive calls, thus lowering the computational overhead.

Memoization is particularly useful in problems with overlapping sub-problems and optimal substructure, which is common in recursive algorithms used for forecasting, dynamic programming, and similar computational problems.