**FINANCIAL FORECASTING**

**Concept of Recursion**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. This method breaks down a complex problem into simpler sub-problems until it reaches a base case that can be solved directly. A recursive function typically consists of two main parts:

**1. Base Case:** The condition under which the recursion stops. It provides a simple, non-recursive solution to a problem, preventing infinite recursion.

**2. Recursive Case:** The part of the function where it calls itself with modified parameters, progressively reducing the problem's size until the base case is reached.

**How Recursion Simplifies Problems**

**1. Divide and Conquer:** Recursion is effective in problems that can be divided into similar sub-problems. By solving each sub-problem recursively, complex problems can be tackled more easily. Classic examples include algorithms for sorting (like quicksort and mergesort) and searching.

**2. Simplifies Code:** Recursive solutions can often be more elegant and concise than their iterative counterparts. For example, problems involving tree traversals or factorial calculations are naturally expressed using recursion, leading to simpler and more readable code.

**3. Eases Problem Solving**: For problems with a naturally hierarchical or nested structure, recursion mirrors the problem's structure, making it easier to design and understand the solution. For instance, traversing a directory structure or solving puzzles like the Tower of Hanoi are more intuitive with recursion.

**4. State Management:** Recursion automatically manages state through the function call stack. Each recursive call maintains its own set of parameters, which simplifies the handling of state and intermediate results.

Example: Factorial Calculation

Calculating the factorial of a number (n!) is a classic example of recursion:

public int factorial(int n) {

if (n <= 1) {

return 1; // Base case

} else {

return n \* factorial(n - 1); // Recursive case

}

}

In this example, `factorial(n)` calls itself with `n - 1` until `n` is 1 or less, where it returns 1, the base case. The recursive calls build up the result as they return, ultimately computing `n!`.

**Limitations:**

- Stack Overflow: Deep recursion can lead to stack overflow if the recursion depth is too high, as each function call consumes stack space.

- Performance: Recursive solutions might be less efficient due to overhead from multiple function calls compared to iterative solutions.

**Time Complexity Analysis:**

**1. Without Optimization (Memoization):**

Without memoization, the algorithm would have to recompute the future value for each period multiple times, leading to exponential time complexity. Specifically, the time complexity would be O(2^n) in the worst case, as each recursive call generates two further calls until the base case is reached.

**2. With Memoization:**

Memoization significantly improves the efficiency by storing previously computed results. In this case, each unique value of `periods` is computed only once. The time complexity with memoization is reduced to O(n), where `n` is the number of periods. This is because each period's value is computed and stored in the `memoizedMap` once, and subsequent lookups are O(1).

**Space Complexity:**

- **Without Optimization:** The space complexity is O(n) due to the depth of the recursion stack.

**- With Memoization:** The space complexity remains O(n), as the memoization table (`memoizedMap`) also requires space proportional to the number of unique period values. Additionally, the recursion stack has a space complexity of O(n) for storing the function call stack.

**Optimizing Recursive Solutions**

**1. Memoization:**

The provided algorithm uses memoization to store previously computed results, which avoids redundant computations. This technique reduces the time complexity by ensuring that each unique sub-problem is solved only once. The `memoizedMap` stores results for each value of `periods`, allowing for O(1) retrieval of previously computed values.

**2. Iterative Approach:**

For some problems, converting a recursive solution to an iterative one can be beneficial. In this case, an iterative solution can compute the future value in O(n) time complexity without the overhead of recursive calls or additional space for memoization.