**Exercise 7: Financial Forecasting**

**Understanding Recursion**

Recursion is a method of programming in which a function processes a problem by calling itself with simpler or smaller inputs. Through this, it breaks down difficult tasks into smaller, simpler components.

**Key Elements of Recursion:**

**Base Case:**

The base case is the halting condition of the recursion. It ensures that the function does not keep calling itself infinitely by giving a straightforward, simple response once the problem becomes sufficiently small.

**Recursive Case:**

This is the section where the function breaks down the problem into smaller instances of the same problem and applies itself with these smaller arguments.

For instance, in computing a factorial, the recursive step is defined as:

factorial(n) = n \* factorial(n - 1).

**How Recursion Simplifies Problem Solving**

**Divide and Conquer:**

Recursion divides a large problem into simpler subproblems, allowing each one to be solved more easily. Each recursive call handles a smaller piece of the initial problem.

**More Elegant Solutions:**

Recursive code is generally shorter and simpler to read than iterative solutions, particularly for problems naturally suited to a recursive solution such as factorials or tree traversals.

**A Natural Approach for Some Problems:**

Some problems naturally have a recursive nature to them. The Fibonacci sequence, tree data structures (where a node can be considered a smaller tree), and traditional puzzles such as the Tower of Hanoi are just a few examples**.**

**Time and Space Complexity of the Recursive Method**

The computeProjection method employs recursion with memoization to determine future investment value.

**Time Complexity:**

Every unique year count has one recursive call, performing constant work. Memoization prevents any redundant calculations. So, time complexity is O(n), where n is the number of years.

**Space Complexity:**

The recursion stack can be n calls deep, and the memo cache contains n items. So, space complexity is also O(n).

**Optimizations to Minimize Computation**

**Iterative Approach:**

Avoiding call stack overhead by replacing recursion with a loop saves space to O(1), without changing O(n) time complexity.

**Skip Memoization When Not Needed:**

For simple yearly growth, caching results offers minimal value and can be avoided.

**Direct Formula:**

Using the compound interest formula:

finalAmount = principal \* (1 + rate)years

calculates the outcome in constant time O(1), giving the most optimal solution where accuracy permits.