Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It can simplify problems that can be divided into similar subproblems. Here are key concepts:

- **Base Case**: The condition under which the recursion terminates. Without a base case, recursion would continue indefinitely.
- **Recursive Case**: The part of the function that includes the recursive call. This breaks the problem down into smaller instances.
- **Stack Usage**: Each recursive call adds a new layer to the call stack, which consumes memory. Proper management of recursion is crucial to avoid stack overflow.

Advantages of Recursion:

- Simplifies code for problems that have a natural recursive structure (e.g., tree traversals, factorial computation).
- Can make complex algorithms easier to implement and understand.

Disadvantages of Recursion:

- Can lead to excessive stack usage if not managed properly.
- May be less efficient than iterative solutions due to overhead from function calls.

Time Complexity Analysis

1. **Recursive Function Call**:
    - The computeFutureValue method is called recursively years times.
    - Each call performs a constant amount of work: calculating initialAmount * (1 + annualRate) and making a recursive call with years - 1.
2. **Base Case**:
    - The recursion terminates when years is 0, which happens after exactly years recursive calls.
3. **Overall Complexity**:
    - The time complexity is proportional to the number of recursive calls.
    - Since the method makes one recursive call per year, the time complexity is O(n)O(n)O(n), where n is the number of years.

To optimize a recursive solution and avoid excessive computation :

1. **Memoization**: Store and reuse results of expensive computations. Useful for problems with overlapping subproblems but not needed here.
2. **Tail Recursion Optimization**: Rearrange recursion so the recursive call is the last operation, potentially optimizing stack usage. This method is less critical for this specific problem.
3. **Iterative Approach**: Convert recursion into an iterative loop to avoid recursion overhead and stack issues. This is generally more efficient for the computeFutureValue problem.
4. **Avoid Redundant Computation**: Use dynamic programming for problems with overlapping subproblems to avoid redundant calculations.