Analysis

Time Complexity of Recursive Algorithm

In a recursive financial forecasting algorithm, the future value is typically calculated using a recurrence relation based on past growth. For example:

Java code:

```
double forecast(int years, double currentValue, double rate) {

  if (years == 0) return currentValue;

  return forecast(years - 1, currentValue, rate) * (1 + rate);

}
```

In this example:

- Each call to forecast() calls itself once with a reduced year count.
- There are n total recursive calls for n years.

Time Complexity: O(n)
Explanation:
Each year requires one recursive call. The number of calls is directly proportional to the number of years being forecasted.

Optimization Techniques

Although the time complexity is linear (O(n)), recursion can still lead to inefficiency due to:

- **Stack usage**: Each recursive call adds to the call stack, increasing the risk of stack overflow for large inputs.
- **Redundant computation**: If recursion involves branching or recomputation of overlapping subproblems, it can be further optimized.

Optimizations:

1. **Convert to Iterative Approach:**
    - A loop-based implementation achieves the same result without recursive overhead:

Java code:

```
double forecastIterative(int years, double currentValue, double rate) {

  for (int i = 0; i < years; i++) {

    currentValue *= (1 + rate);
  }
  return currentValue;
}
```

- Time Complexity: O(n)
- Space Complexity: O(1)

2. **Memoization (if overlapping subproblems exist):**
    - Store intermediate results to avoid recalculating values in recursive branching cases.
    - Useful if the recurrence is more complex than a simple linear chain.
3. **Tail Recursion (if language supports optimization):**
    - Tail-recursive versions may be optimized by some compilers or interpreters to avoid additional stack usage.

Conclusion

Recursive algorithms offer elegant solutions for forecasting, especially in conceptually simple models. However, they must be optimized when dealing with large inputs. Converting recursion to iteration is often the most practical approach, offering both clarity and performance with no risk of stack overflow.