**Recursive Algorithm**

Recursion is a method of solving a problem where a function calls itself to solve smaller instances of the same problem. It is useful for problems that can be broken down into simpler sub-problems.

**Example**: Calculating factorial  
factorial(n) = n \* factorial(n - 1)

In financial forecasting, recursion can be used to compute future values by repeatedly applying a growth rate formula.

A recursive method that forecasts future financial value based on:

* initialValue: the starting value
* growthRate: the rate at which the value grows annually
* years: the number of years into the future

**Recursive Formula:**

futureValue(years) = futureValue(years - 1)\*(1+growthRate)

Base case: if years == 0, return initialValue

**Analysis**

**(i) Time Complexity**

* The time complexity is O(n) where n is the number of years.
* This is because the function makes one recursive call per year.

**(ii)Optimization Suggestions**

Although the current recursive function is efficient for small values of n, optimization is essential when:

* There are repeated subproblems, such as in recursive Fibonacci calculations.
* Deep recursion risks causing stack overflow in cases of large n.

**Optimization Strategies**

* **Memoization**  
  Store the results of previous recursive calls in a cache to avoid recomputing values.This is useful when the same inputs might be processed multiple times.
* **Iterative Approach**

The iterative approach calculates the future value using a simple loop, avoiding recursive calls and reducing the risk of stack overflow.

**Example code for Iterative Approach:**

public static double forecastIterative(double initialValue,double growthRate,int years) {

double value = initialValue;

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

value\*= (1+growthRate);

}

return value;

}

**Conclusion**

Recursion is a powerful tool to express problems like financial forecasting elegantly. However, for large-scale computations, recursive methods should be optimized or replaced with iterative versions to improve efficiency and prevent stack overflow.