# DATA STRUCTURES AND ALGORITHM- FINANCIAL FORECASTING

# 1. Recursive Algorithms Explained

Recursion solves problems by breaking them into smaller, identical subproblems until reaching a base case. For financial forecasting:

* - Base Case: Initial value (period 0)
* - Recursive Case: Future value at period t is derived from period t-1 and its growth rate:

FV(t) = FV(t-1) × (1 + growth\_rate[t])

# 2. Implementation

public class FinancialForecaster {  
  
 public static double calculateFutureValue(double initial, double[] growthRates, int periods) {  
 if (periods == 0) return initial;   
  
 double prevValue = calculateFutureValue(initial, growthRates, periods - 1);  
 return prevValue \* (1 + growthRates[periods - 1]);  
 }  
  
 public static double calculateFutureValueIterative(double initial, double[] growthRates) {  
 double current = initial;  
 for (double rate : growthRates) {  
 current \*= (1 + rate);  
 }  
 return current;  
 }  
  
 public static void main(String[] args) {  
 double initial = 1000.0;  
 double[] growthRates = {0.05, 0.06, 0.07};  
  
 double futureValue = calculateFutureValue(initial, growthRates, growthRates.length);  
 System.out.printf("Recursive result: $%.2f\n", futureValue);  
  
 double iterativeValue = calculateFutureValueIterative(initial, growthRates);  
 System.out.printf("Iterative result: $%.2f\n", iterativeValue);  
 }  
}

# 3. Time Complexity Analysis

| Approach | Time Complexity | Space Complexity |  
|-----------------|-----------------------|------------------------|  
| Recursive | O(n) | O(n) (stack) |  
| Iterative | O(n) | O(1) |

Key Observations:

* - Both approaches have O(n) time complexity
* - Recursive method uses O(n) stack space, risking StackOverflowError for large datasets
* - Iterative approach uses constant space (O(1))

# 4. Optimization Strategies

* Problem: Recursive approach is inefficient for large datasets due to:
* - Stack memory overhead
* - Function call overhead
* Solutions:

1.Iterative Replacement (Recommended):

public static double calculateFutureValueIterative(double initial, double[] growthRates) {  
 double current = initial;  
 for (double rate : growthRates) {  
 current \*= (1 + rate);  
 }  
 return current;  
}

* - Eliminates stack usage
* - Constant space complexity

2. Memoization (For demonstration):

public static double calculateFutureValueMemo(double initial, double[] growthRates, int period, Double[] memo) {  
 if (memo[period] != null) return memo[period];  
  
 if (period == 0) {  
 memo[0] = initial;  
 return initial;  
 }  
  
 double prev = calculateFutureValueMemo(initial, growthRates, period - 1, memo);  
 double current = prev \* (1 + growthRates[period - 1]);  
 memo[period] = current;  
 return current;  
}

* - Usage: Double[] memo = new Double[growthRates.length + 1];
* - Still uses recursion but avoids redundant calculations

# 5. Output

Recursive result: $1190.91  
Iterative result: $1190.91

Calculation Trace:

* Period 0: $1000.00 (Base)
* Period 1: $1000.00 × 1.05 = $1050.00
* Period 2: $1050.00 × 1.06 = $1113.00
* Period 3: $1113.00 × 1.07 = $1190.91

# 6. Recommendations

1. Use iterative approach for production systems:

* - Avoids stack overflow
* - Better memory efficiency
* - Handles large datasets

2. Use recursion only when:

* - Dataset size is guaranteed small
* - Code readability outweighs performance concerns

3. Always validate inputs:

if(growthRates ==null ||periods >growthRates.length) {

throw new IllegalArgumentException(“Invalid Growth Rates Array”);

}