# Understanding Recursion and Financial Forecasting

## Concept of Recursion

Recursion is a programming strategy where a function solves a problem by calling itself directly or indirectly. This process continues until it reaches a well-defined stopping condition known as the base case. Recursion is often used to solve problems that have a repetitive or nested structure.

## Components of Recursion

Base Case:

This is the stopping condition. It prevents infinite recursion and typically handles the simplest scenario directly.

Recursive Case:

This defines how the function reduces the original problem into smaller instances and calls itself.

## Example: Factorial Calculation

The factorial of a number n (written as n!) is defined as:  
 n! = n × (n-1) × (n-2) × ... × 1  
 Base case: 0! = 1 or 1! = 1  
Recursive Java implementation:

public class Factorial {  
 public static int factorial(int n) {  
 if (n == 0 || n == 1) return 1;  
 return n \* factorial(n - 1);  
 }  
 public static void main(String[] args) {  
 int number = 5;  
 System.out.println("Factorial of " + number + " is " + factorial(number));  
 }  
}

## How Recursion Simplifies Problems

* Divide and Conquer: It divides problems into smaller sub-problems until they become manageable.
* Readable Logic: Recursion leads to elegant and often shorter solutions for complex problems.
* Natural Fit: Ideal for problems like tree traversals, backtracking, and problems with self-similarity.
* Stage-wise Solving: Each recursive call handles one level or phase of the problem.

## Drawbacks and Considerations

* Stack Overflow: Too many recursive calls may exceed the stack limit.
* Performance: Recursion without optimization can repeat computations unnecessarily.
* Debugging Complexity: Recursive call chains may be harder to trace and debug.
* Memory Usage: Each call adds overhead to the stack.

## Time and Space Complexity of Recursive Algorithm

In the “calculateFutureValueMemoized” function, recursion is optimized using memoization:

Time Complexity:

With memoization: O(n), where n is the number of years. Each value is calculated once and stored.

Space Complexity:

- Stack space: O(n), due to the recursive call depth.  
- Memoization space: O(n), one entry per year.

## Optimization Techniques for Recursive Solutions

**Iterative Method:**

public static double calculateFutureValueIterative(double principal, double rate, int years) {  
 for (int i = 0; i < years; i++) {  
 principal \*= (1 + rate);  
 }  
 return principal;  
}

**Formula-Based Method:**

public static double calculateFutureValue(double principal, double rate, int years) {  
 return principal \* Math.pow(1 + rate, years);  
}