**Financial Forecasting**

**Recursion** is a fundamental concept in computer science where a function calls itself to solve smaller instances of the same problem. It is often used to break down complex problems into simpler sub-problems.

**Key Components of Recursion**

1. **Base Case:**
   * The condition under which the recursion stops.
   * It prevents the function from calling itself indefinitely.
   * Example: In calculating the factorial of a number, the base case is when the number is 1.
2. **Recursive Case:**
   * The part where the function calls itself with a smaller or simpler input.
   * It moves the problem towards the base case.
   * Example: In calculating the factorial of a number nnn, the recursive case is n×factorial(n−1)n \times \text{factorial}(n-1)n×factorial(n−1).

**Example: Factorial Calculation**

The factorial of a number nnn is the product of all positive integers less than or equal to nnn. It can be defined recursively as:

factorial(n)={1if n=0n×factorial(n−1)if n>0\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{factorial}(n-1) & \text{if } n > 0 \end{cases}factorial(n)={1n×factorial(n−1)​if n=0if n>0​

**Example:**

public class Factorial {

public static int factorial(int n) {

if (n == 0) {

return 1; // Base case

} else {

return n \* factorial(n - 1); // Recursive case

}

}

public static void main(String[] args) {

int number = 5;

System.out.println("Factorial of " + number + " is: " + factorial(number));

}

}

**Optimizing Recursion**

1. **Memoization:**
   * Storing the results of expensive function calls and reusing them when the same inputs occur again.
2. **Iterative Approach:**
   * Converting recursive solutions to iterative ones using loops to avoid the overhead of multiple function calls.

**Benefits:**

* Simplifies code for problems that can be divided into similar sub-problems (e.g., calculating factorials, Fibonacci numbers).
* Can make algorithms more intuitive and easier to understand.

**Drawbacks:**

* Recursive algorithms can lead to excessive function calls and increased memory usage if not handled properly.
* Sometimes leads to stack overflow issues for deep recursion.

**Analysis**

**Time Complexity:**

* The time complexity of the recursive algorithm is O(n)O(n)O(n), where nnn is the number of periods.
  + Each recursive call reduces the number of periods by 1, resulting in nnn calls.
  + The space complexity is also O(n)O(n)O(n) due to the call stack depth.

**Optimization:**

* **Memoization:** Store results of previously computed values to avoid redundant calculations and improve efficiency.
* **Iterative Approach:** For larger datasets or deeper recursion, consider converting the recursion to an iterative approach to prevent stack overflow and reduce memory usage.