**Understanding Recursion in Programming**

**Recursion** is a programming approach where a function calls itself to solve smaller sub-problems of a larger task. This technique helps simplify complex problems by breaking them down into more manageable components.

| **Component** | **Description** | **Example (Fibonacci)** |
| --- | --- | --- |
| **Base Case** | A condition that terminates recursion and returns a direct result. | If n == 0 return 0, if n == 1 return 1. |
| **Recursive Case** | The part where the function breaks the problem into smaller calls. | fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) |

### **Example: Fibonacci Using Recursion**

The **Fibonacci sequence** is a series where each number is the sum of the two preceding ones:

Ex:

0, 1, 1, 2, 3, 5, 8, 13, ...

javaCode

public class Fibonacci {

public static int fibonacci(int n) {

// Base cases

if (n == 0) return 0;

if (n == 1) return 1;

// Recursive case

return fibonacci(n - 1) + fibonacci(n - 2);

}

public static void main(String[] args) {

int n = 6;

System.out.println("Fibonacci number at position " + n + " is " + fibonacci(n));

}

}

### **Why Use Recursion?**

1. **Simplifies Problem Solving**
   * Breaks a complex problem into easier sub-problems.
2. **Cleaner Code for Certain Problems**
   * Recursion provides elegant solutions for naturally recursive problems like tree traversal, backtracking, etc.
3. **Fits Naturally in Recursive Data Structures**
   * Useful in algorithms on trees, graphs, and nested structures.

| **Issue** | **Explanation** |
| --- | --- |
| **Performance Cost** | Recursive Fibonacci has O(2^n) time complexity without optimization. |
| **Stack Overflow** | Deep recursion may exceed stack memory. |
| **Harder to Debug** | Logic errors in recursive flows can be less obvious than in loops. |

### **Improving Recursive Fibonacci with Memoization**

javaCode

import java.util.HashMap;

import java.util.Map;

public class FibonacciMemoized {

private static Map<Integer, Integer> memo = new HashMap<>();

public static int fibonacci(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

if (memo.containsKey(n)) return memo.get(n);

int result = fibonacci(n - 1) + fibonacci(n - 2);

memo.put(n, result);

return result;

}

public static void main(String[] args) {

int n = 30;

System.out.println("Fibonacci number at position " + n + " is " + fibonacci(n));

}

}

### **Complexity Comparison**

| **Version** | **Time Complexity** | **Space Complexity** | **Pros** | **Cons** |
| --- | --- | --- | --- | --- |
| **Naive Recursive** | O(2^n) | O(n) | Simple and easy to write | Extremely slow for large n |
| **Memoized Recursive** | O(n) | O(n) | Fast and avoids repetition | Still uses recursion stack |
| **Iterative** | O(n) | O(1) | Most efficient | More complex code than basic |

### **Iterative Fibonacci Approach**

javaCode

public static int fibonacciIterative(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

int prev1 = 0, prev2 = 1, current = 0;

for (int i = 2; i <= n; i++) {

current = prev1 + prev2;

prev1 = prev2;

prev2 = current;

}

return current;

}

### **Conclusion**

Recursion is a powerful technique for solving problems with a repetitive, nested, or tree-like structure. While it may not always be the most efficient in raw performance, it often leads to cleaner, easier-to-understand code—especially when combined with optimizations like **memoization**.