**What is Recursion?**

Recursion is a method in which a **function calls itself** to solve a problem. It is often used to break down complex problems into smaller subproblems of the same type.

**Key Characteristics of Recursion**

1. **Base Case** – The condition where recursion stops.
2. **Recursive Case** – The function calls itself with a modified argument.
3. **Function Stack** – Every recursive call is stored in a stack until the base case is reached.

**How Recursion Works? (Step-by-Step)**

Consider the **factorial function**:

|  |
| --- |
| int factorial(int n) {  if (n == 0 || n == 1) return 1; // Base Case  return n \* factorial(n - 1); // Recursive Case  } |

**Tracing factorial (4)**

|  |
| --- |
| factorial(4)  → 4 \* factorial(3)  → 3 \* factorial(2)  → 2 \* factorial(1)  → 1 (Base Case) |

Now, it **unwinds**:

|  |
| --- |
| factorial(1) = 1  factorial(2) = 2 \* 1 = 2  factorial(3) = 3 \* 2 = 6  factorial(4) = 4 \* 6 = 24 |

✅ **Final Output**: factorial (4) = 24

**Types of Recursions**

1️ **Direct Recursion** – A function calls itself directly.

|  |
| --- |
| void func() {  func(); } |

2️ **Indirect Recursion** – Function A calls Function B, and Function B calls Function A.

|  |
| --- |
| void A() { B(); }  void B() { A(); } |

3️ **Tail Recursion** – The recursive call is the last operation in the function.

|  |
| --- |
| void tailRecursion(int n) {  if (n == 0) return;  System.out.println(n);  tailRecursion(n - 1);  } |

4️ **Non-Tail Recursion** – The recursive call is **not** the last operation.

|  |
| --- |
| int sum (int n) {  if (n == 0) return 0;  return n + sum (n - 1);  } |

**Recursion vs. Iteration**

| **Feature** | **Recursion** | **Iteration** |
| --- | --- | --- |
| Memory Usage | High (stack frames) | Low |
| Speed | Slow (stack overhead) | Faster |
| Readability | Simple & clean | More complex |
| Use Case | Tree, Graph, Divide & Conquer problems | Loops, simple problems |

**Advantages of Recursion**

✅ Reduces code complexity.  
✅ Solves problems naturally suited to recursion (e.g., Tree, Graph).  
✅ Uses less boilerplate code compared to iteration.

**Disadvantages of Recursion**

❌ Higher memory usage due to function call stack.  
❌ Slower execution because of function calls.  
❌ Can cause **Stack Overflow** if the base case is missing.

**When to Use Recursion?**

✔️ Problems that can be divided into smaller **subproblems**.  
✔️ **Tree Traversal** (Preorder, In order, Post order).  
✔️ **Graph Traversal** (DFS).  
✔️ **Divide & Conquer** algorithms (Merge Sort, Quick Sort).  
✔️ **Backtracking** (Sudoku, N-Queens, Maze Solving).

**When NOT to Use Recursion?**

❌ Simple loops can do the job efficiently.  
❌ Memory is limited (large recursion depth).  
❌ Real-time applications needing fast performance.

**Stack Overflow Issue in Recursion**

If recursion goes too deep, it can cause a **StackOverflowError**.  
Example:

|  |
| --- |
| void infiniteRecursion() {  infiniteRecursion(); // No base case!  } |

✅ **Solution**: Always define a base case.

**Example Recursion Problems and Concepts**

1️. **Fibonacci Sequence** – Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)  
2️.**Binary Search** – O(log N)  
3️.**Tower of Hanoi** – Move disks between rods  
4️.**Backtracking Problems** – N-Queens, Maze Solver  
5️.**Sorting Algorithms** – Merge Sort, Quick Sort