**RECURSION**

# Introduction

- Recursion is a technique where a function calls itself to solve a problem by breaking it down into smaller, similar sub-problems.

## Basic Structure of a Recursive Function

- Base Case: The stopping condition to end the recursion and prevent infinite loops.

*if (condition) return value;*

- Recursive Case: The part where the function calls itself.

*return recursiveFunction(newData);*

## Key Concepts

- Call Stack: Each function call is stored on the stack until it reaches the base case and starts returning values.

- Returning Values: Ensure the function returns values from the recursive case all the way back to the initial call.

## Rules for Writing Recursive Functions

- Identify the Base Case: Define the condition where recursion stops.

- Identify the Recursive Case: Define how the function progresses closer to the base case.

- Ensure Progression: Update parameters to move closer to the base case in each call.

# Common Issues with Recursion

- Stack Overflow: Occurs when the base case is missing or unreachable, causing infinite recursive calls.

- Memory Consumption: Each recursive call adds to the call stack, which uses memory. Ensure recursion is efficient.

# Recursive vs. Iterative

## Theorem

Anything that can be implemented recursively can also be implemented iteratively.

Recursion and iteration are interchangeable, but the choice depends on the situation.

## Pros of Recursion

- Readable and DRY (Don’t Repeat Yourself): Recursive solutions are often more concise and maintainable, especially for problems like tree traversal.

- Simpler for Certain Problems: Recursion can simplify code when dealing with unknown depths or inherently recursive structures (e.g., trees, graphs).

- Elegant for Traversals: Ideal for scenarios like searching or traversing tree-like data structures.

## Cons of Recursion

- Memory Overhead: Every recursive call adds to the call stack, increasing memory usage.

- Stack Overflow Risk: Too many recursive calls can lead to stack overflow errors.

- Harder to Understand: For some, recursion can be less intuitive, particularly in complex cases.

- Performance: Iterative solutions are generally more efficient as they avoid the overhead of repeated function calls.

## When to Use Recursion

- Tree and Graph Traversal: Recursion simplifies traversing or searching trees/graphs (e.g., DFS).

- Divide and Conquer: Use recursion when a problem can be divided into smaller, similar subproblems (e.g., Merge Sort, Quick Sort).

- Identical Subproblems: When the problem involves solving smaller versions of the same problem (e.g., Fibonacci, Factorial).

- Combining Subproblem Solutions: If solving smaller problems and combining them solves the overall problem (e.g., tree traversal).

## Key Indicators for Recursion:

- Problem can be split into smaller subproblems.

- Subproblems have the same structure.

- Solutions to subproblems combine to solve the main problem.