Recursion

1. Introduction

Recursion is a powerful technique in programming where a **function calls itself** either **directly** or **indirectly** to solve a problem.

- Recursion breaks down a **large problem** into **smaller subproblems** of the same type.
- It continues until it reaches a condition where the problem becomes simple enough to solve directly.

💡 Example in real life:

- A set of **mirrors facing each other** (image repeats inside itself).
- **Nested folders** on a computer (folder inside folder).

2. Advantages of Recursion

1. **Simplifies Complex Problems**
    - Some problems are naturally recursive (like tree traversal, Tower of Hanoi, Fibonacci).
    - Writing recursive solutions makes the logic easier to understand.
2. **Reduces Code Size**
    - Recursive solutions require fewer lines of code compared to iterative solutions.
    - Example: Factorial in recursion is just a few lines, while loops may take more steps.
3. **Improves Readability**

- Recursive code often looks cleaner and more elegant.
- It expresses the solution in a mathematical way (e.g., `fact(n) = n * fact(n-1)`).

4. **Useful in Divide and Conquer Algorithms**
   - Algorithms like **Merge Sort, Quick Sort, Binary Search** are easier to implement using recursion.
5. **Natural Fit for Hierarchical Data**
   - Recursion is very effective when working with **tree and graph structures** (like file systems, XML/JSON parsing).
6. **Mathematical Problem Solving**
   - Problems like factorial, Fibonacci, GCD, permutations, and combinations are naturally defined recursively.

3. How Recursion Works

A recursive function always has **two parts**:

1. **Base Case (Stopping Condition)**
   - The simplest case of the problem that can be solved directly.
   - Prevents infinite recursion.
2. **Recursive Case**
   - The part where the function calls itself with a smaller input, moving closer to the base case.

Example: Factorial (n!)

Factorial of n = `n * (n-1) * (n-2) * ... * 1`

Recursive Definition:

- **Base Case**: `fact(0) = 1`
- **Recursive Case**: `fact(n) = n * fact(n-1)`

```
int factorial(int n) {
    if (n == 0)  // Base case
        return 1;
    else
        return n * factorial(n-1); // Recursive case
}
```

4. Types of Recursion

Recursion can be classified into different types based on **how and when** the recursive call is made. Let's go through each one:

4.1 Direct Recursion

A function is said to be directly recursive when it **calls itself directly.**

- This is the simplest form of recursion.
- The function directly calls itself within its own body.
- Base case is required to prevent infinite calls.

```
#include <iostream>
using namespace std;

void fun(int n) {
    if (n > 0) {  // base case
```

```
        cout << n << " ";
        fun(n - 1); // direct recursive call
    }
}

int main() {
    fun(5);
    return 0;
}
```

Output: 5 4 3 2 1

4.2 Indirect Recursion

A function is said to be indirectly recursive when it **calls another function**, and that function calls the first one.

- The recursion happens in a **chain of function calls**.
- At least two functions are involved.

```
#include <iostream>
using namespace std;

void B(int n); // function prototype

void A(int n) {
    if (n > 0) {
        cout << n << " ";
        B(n - 1);
    }
}
```

```
void B(int n) {
    if (n > 0) {
        cout << n << " ";
        A(n / 2);
    }
}

int main() {
    A(10);
    return 0;
}
```

Output: 10 9 4 3 1

4.3 Tail Recursion

If the **last statement of a function** is a recursive call, it is called tail recursion.

- No computation is left after the recursive call.
- Easy to convert into iteration.
- Memory efficient in some cases.

```
#include <iostream>
using namespace std;

void tailRec(int n) {
    if (n == 0) return;
    cout << n << " ";   // work first
    tailRec(n - 1);     // recursive call at the end
}

int main() {
```

```
    tailRec(5);
    return 0;
}
```

Output: 5 4 3 2 1

4.4 Head Recursion

If the **recursive call happens before any computation**, it is called head recursion.

- Work is done **after returning** from recursive call.
- Reverse order output compared to tail recursion.

```
#include <iostream>
using namespace std;

void headRec(int n) {
    if (n == 0) return;
    headRec(n - 1);   // recursive call first
    cout << n << " "; // work later
}

int main() {
    headRec(5);
    return 0;
}
```

Output: 1 2 3 4 5

4.5 Tree Recursion

A recursive function that calls itself **more than once** is called tree recursion.

- Execution forms a tree structure of calls.
- Very common in problems like **Fibonacci, Tower of Hanoi, Tree Traversals.**
- Less efficient because many calls repeat.

```
#include <iostream>
using namespace std;

void treeRec(int n) {
    if (n > 0) {
        cout << n << " ";
        treeRec(n - 1); // first recursive call
        treeRec(n - 1); // second recursive call
    }
}

int main() {
    treeRec(3);
    return 0;
}
```

Output: 3 2 1 1 2 1 1

4.6 Nested Recursion

In nested recursion, the function's argument is **itself a recursive call**.

- Function calls itself with a recursive function call as a parameter.
- Example: **McCarthy 91 function** is a famous nested recursive function.

```
#include <iostream>
using namespace std;

int nestedRec(int n) {
    if (n > 100)
        return n - 10;
    return nestedRec(nestedRec(n + 11)); // recursive call inside
argument
}

int main() {
    cout << nestedRec(95);
    return 0;
}
```

Output: 91