Recursion

- Recursion is a programming technique in which a function solves a problem by calling itself, either directly or indirectly, with a smaller input each time until a stopping/base condition is reached.
- Why recursion works
    - Breaks a large problem into smaller subproblems
    - Each call handles a simpler case
    - Eventually reaches a base case, stopping further calls
- Two mandatory parts
    - Base case
        - Prevents infinite calls
        - Defines the simplest form of the problem
    - Recursive case
        - Function calls itself with modified arguments
        - Moves toward the base case
- Example

```
int factorial(int n) {
    if (n == 0)           // base case
        return 1;
    return n * factorial(n - 1);    // recursive case
}
```

Types of Recursion

1. Direct Recursion

- A function is said to be directly recursive if it calls itself directly inside its body.

```
void fun() {
    fun();
}
```

- Characteristics
    - Simple to understand
    - Easy to implement
    - Commonly used in problems like factorial, Fibonacci, binary search
- Example

```
void print(int n) {
    if (n == 0)
```

```
        return;
    printf("%d ", n);
    print(n - 1);
}
```

2. Indirect Recursion

- Indirect recursion occurs when two or more functions call each other in a cyclic manner.

```
void fun1() {
    fun2();
}

void fun2() {
    fun1();
}
```

- Characteristics
  - More complex than direct recursion
  - Used when problem logic is naturally split across functions
  - Often seen in state-based systems
- Applications
  - Menu-driven programs
  - Traffic light controllers
  - Compilers and parsers
  - Multiplayer games

Types of Direct Recursion

1. Tail Recursion

- A recursive function is tail recursive if the recursive call is the last statement executed by the function.

```
void fun(int n) {
    if (n == 0) return;
    fun(n - 1);    // tail recursion
}
```

- Characteristics
  - No operation is performed after the recursive call
  - Can be optimized into a loop by compilers
  - Uses less stack memory

2. Non-Tail Recursion

- A recursive function is non-tail recursive if there are operations after the recursive call.

```
int fact(int n) {
    if (n == 0)
        return 1;
    return n * fact(n - 1);    // non-tail
}
```

- Characteristics
  - Cannot be optimized easily
  - Uses more stack space
  - Most common form of recursion

3. Linear Recursion

- A function is said to have linear recursion if it makes only one recursive call per function invocation.

```
void fun(int n) {
    if (n == 0) return;
    fun(n - 1);
}
```

- Characteristics
  - Simple call chain
  - Stack depth grows linearly with input size
  - Used in counting, searching, summing problems

4. Multiple (Tree) Recursion

- A function is multiple recursive if it makes more than one recursive call.

```
int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

- Characteristics
  - Forms a tree-like structure of calls
  - High time complexity
  - Large number of repeated calls
- Applications
  - Fibonacci series
  - Game trees
  - Divide-and-conquer algorithms

5. Nested Recursion

- A recursive function is nested recursive if the function calls itself inside its own argument.

```
int fun(int n) {
    if (n > 100)
        return n - 10;
    return fun(fun(n + 11));
}
```

- Characteristics
  - Difficult to understand and trace
  - Rare in practical programming
  - Mostly used for academic purposes

Advantages of Tail Recursion

- Memory Efficient
  - Since nothing is left to do after the recursive call, the compiler/interpreter can reuse the same stack frame instead of creating a new one for each call. This is called tail call optimization (TCO).
- Avoids Stack Overflow
  - Large input sizes are safer because stack frames aren't accumulating like in non-tail recursion.
- Can Be Converted to Iteration
  - Tail recursion is essentially like a `while` or `for` loop. Many compilers automatically optimize it to iterative code.
- Faster Execution
  - Reusing stack frames reduces overhead, making tail recursion faster than non-tail recursion.

Non-Tail Recursion Drawbacks

- Each recursive call creates a new stack frame.
- Stack frames accumulate until the base case is reached.
- Risk of stack overflow for deep recursion.
- Slightly slower due to extra stack operations.