Recursive Function

A **recursive function** is a function that calls itself in order to solve a problem. Recursive functions break down complex problems into simpler, smaller sub-problems that are easier to solve, often following a pattern where the function repeatedly calls itself with modified parameters until a base condition is met.

Key Components of a Recursive Function:

1. **Base Case**: The condition under which the function stops calling itself. This is essential to prevent infinite recursion.
2. **Recursive Case**: The part of the function where it calls itself, with modified arguments, working towards the base case.

How Recursive Functions Work

In each recursive call:

- The function processes a part of the problem.
- It reduces the problem's complexity by modifying the arguments.
- This continues until the base case is reached, at which point the function begins to return results back up the call stack.

When to Use Recursion

- **Problems with Repetitive Substructure**: Problems like factorial, Fibonacci series, and tree traversals are naturally recursive.
- **Dividing and Conquering**: Recursive algorithms work well with tasks like quicksort, mergesort, and binary search.
- **Tree or Graph Data Structures**: Recursive functions are effective for traversing trees, graphs, and nested data structures.

Pros and Cons of Recursion

- **Pros**:
  - Makes code simpler and more readable for tasks with naturally recursive structures.
  - Often results in shorter, more elegant code.
- **Cons**:
  - Higher memory usage due to function call stack.
  - Risk of **stack overflow** if the recursion depth is too large or the base case is missing.

Recursion is a powerful tool for breaking down complex problems, but it should be used carefully to ensure efficiency and prevent errors.

Problem: Fibonacci Sequence

The **Fibonacci sequence** is a series of numbers in which each number (after the first two) is the sum of the two preceding ones:

- Fibonacci sequence: `0, 1, 1, 2, 3, 5, 8, 13, ...`

Each number in the sequence can be defined as:

- **Base cases**: F(0)=0F(0) = 0F(0)=0 and F(1)=1F(1) = 1F(1)=1
- **Recursive formula**: F(n)=F(n−1)+F(n−2)F(n) = F(n - 1) + F(n - 2)F(n)=F(n−1)+F(n−2)

Task

Write a recursive function in C# to find the nnn-th Fibonacci number

Solution Using Recursion

1. **Define the Recursive Function**:
    - The function will have a base case for F(0)F(0)F(0) and F(1)F(1)F(1).
    - For all other values of nnn, the function will call itself with the values n−1n - 1n−1 and n−2n - 2n−2.
2. **Recursive Steps**:
    - For each call of the function, it breaks the problem down into calculating `F(n - 1)` and `F(n - 2)`.
    - It adds up these results to get the Fibonacci number at `n`.
    - The process repeats until it reaches the base cases, then the function starts returning results up the call stack.

```
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);
}
// Usage
Console.WriteLine(Fibonacci(6)); // Output: 8
```

Explanation of the Fibonacci Function

1. **Base Cases**:
   - If `n` is `0`, the function returns `0`.
   - If `n` is `1`, the function returns `1`.
2. **Recursive Case**:
   - For any other `n`, it calls `Fibonacci(n - 1)` and `Fibonacci(n - 2)` and adds their results.
3. **Recursive Calls Breakdown** for `Fibonacci(6)`:
   - `Fibonacci(6)` calls `Fibonacci(5) + Fibonacci(4)`.
   - `Fibonacci(5)` calls `Fibonacci(4) + Fibonacci(3)`.
   - This process continues, breaking down the problem until the base cases are reached.

4. **Recursive Call Stack**:

- Recursive calls are placed on the call stack until reaching a base case.
- Once base cases are hit, the results propagate back up, solving each part of the problem.

Visual Representation of the Recursive Calls

For `Fibonacci(4)`, the call tree looks like this:

```
Fibonacci(4)
├── Fibonacci(3)
│   ├── Fibonacci(2)
│   │   ├── Fibonacci(1) -> 1
│   │   └── Fibonacci(0) -> 0
│   └── Fibonacci(1) -> 1
└── Fibonacci(2)
      ├── Fibonacci(1) -> 1
      └── Fibonacci(0) -> 0
```

The results are summed as the function returns up the stack:

- `Fibonacci(2) = 1 + 0 = 1`
- `Fibonacci(3) = 1 + 1 = 2`
- `Fibonacci(4) = 2 + 1 = 3`

Advantages and Disadvantages of Using Recursion for Fibonacci

- **Advantages**:
  - Recursive solutions closely resemble the mathematical definition, making the code simple and clear.
- **Disadvantages**:
  - This implementation has **exponential time complexity** due to repeated calls (e.g., `Fibonacci(3)` is calculated multiple times).
  - **Stack overflow risk**: For large `n`, it can lead to excessive memory use and potential stack overflow.