RECURSION

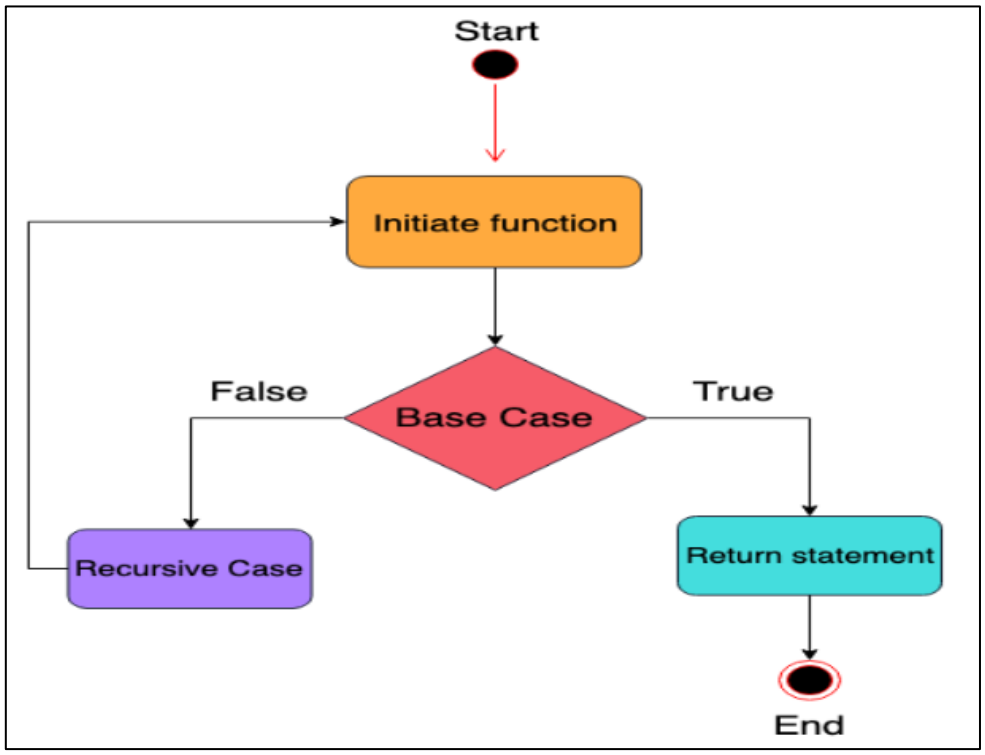
Recursion is a programming technique where a function calls itself in order to solve a problem. The function solves a smaller instance of the same problem in each recursive step until it reaches a base case (a condition that stops the recursion).

Components of a Recursive Function

A recursive function consists of 2 main parts:

1. **Base Case:** The base case is where all further calls to the same function stop, meaning that it does not make any subsequent recursive calls. It is an identifiable condition.
2. **Recursive Case/Call**: The recursive case is where the function calls itself repeatedly until it reaches the base case.

Code Flow of a Recursive Function



Memory Allocation in Recursion

When a function is called, its memory is allocated on a stack. Stacks in computing architectures are the regions of memory where data is added or removed in a last-in-first-out (LIFO) process. Each program has a reserved region of memory referred to as its stack. When a function executes, it adds its state data to the top of the stack. When the function exits, this data is removed from the stack.

IT'S TYPES

1. Direct Recursion
   When a function calls itself directly.

```c
void directRecursion(int n) {
    if (n <= 0)
        return;
    directRecursion(n - 1); // direct recursive call
}
```

2. Indirect Recursion
   When a function calls another function, which in turn calls the original function.

```c
void functionB(int n);

void functionA(int n) {
    if (n <= 0)
        return;
    functionB(n - 1); // calls functionB
}

void functionB(int n) {
    if (n <= 0)
        return;
    functionA(n - 1); // calls functionA
}
```

3. Tail Recursion
   A recursion where the recursive call is the last operation in the function.

```c
void tailRecursion(int n) {
    if (n <= 0)
        return;
    tailRecursion(n - 1); // no operations after the recursive call
}
```

4. Non-Tail Recursion
   In this type, some operations are performed after the recursive call.

```cpp
int nonTailRecursion(int n) {
    if (n <= 0)
        return 0;
    return n + nonTailRecursion(n - 1);
}
```

ISSUES

1. **Stack Overflow**
   Every recursive call consumes stack space. If recursion depth becomes too large, it can lead to a stack overflow error. This happens because each recursive call adds a new frame to the call stack, and the system's stack memory can be exhausted.
2. **Performance (Time Complexity)**
   Recursive functions can be less efficient if they involve repeated calculations. For example, tree recursion often results in exponential time complexity, causing performance degradation. Dynamic programming can be used to optimize this.
3. **Space Complexity**
   Recursive algorithms can be memory-intensive because of the space needed to store all the recursive calls in the call stack. Even a simple tail-recursive function can consume more memory than an equivalent iterative function.
4. **Difficulty in Debugging**
   Recursion can be more difficult to debug than iterative solutions because tracking the flow of execution through multiple recursive calls can be confusing.

Example

FIBONACCI FUNCTION

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1. Mathematically,
$F(n)=F(n-1)+F(n-2)$
with base cases $F(0)=0$ and $F(1)=1$.

CODE:

```cpp
#include <iostream>

using namespace std;


int fibonacci(int n) {

   // Base case: F(0) = 0, F(1) = 1
```

```
    if (n <= 1)
        return n;
    // Recursive case: F(n) = F(n-1) + F(n-2)
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    cout << "Enter the position in the Fibonacci sequence: ";
    cin >> n;
    cout << "Fibonacci number at position " << n << " is " << fibonacci(n) << endl;
    return 0;
}
```

FACTORIAL FUNCTION

The factorial of a number n is the product of all positive integers less than or equal to n. Mathematically,

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

CODE:

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

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

```
    return n * factorial(n - 1);
}

int main() {
    int number;
    cout << "Enter a number to find its factorial: ";
    cin >> number;
    cout << "Factorial of " << number << " is " << factorial(number) << endl;
    return 0;
}
```