Recursion is a programming teqnique that allows us to express functions that would be difficult to implement in an imperative way.

Recursion is a simple concept to explain but hard to master.

A Recursive function is simply a function that calls itself within itself.

It partially calculates the result in one iteration and then returns the partial result to the caller which combines the partial result with other parts and then returns its own result.

This process repeats until the final result is calculated from the partial results

It consists of the following parts:-

- **Base Case** to end the recursion. If omitted the recursion becomes **infinite** and doesn’t stop.
- **The General Case**: the part of the function where it **calls itself**

A famous example is the Fibonacci sequence defined as follows :-

$$fib(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n \geq 2 \end{cases}$$

With

-fib(2) = fib(1) + fib(0) = 2

-fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1)= (1+1)+1=3

-fib(4) = fib(3) + fib(2) = (fib(2) + fib(1))+3=(3+1)+3=7

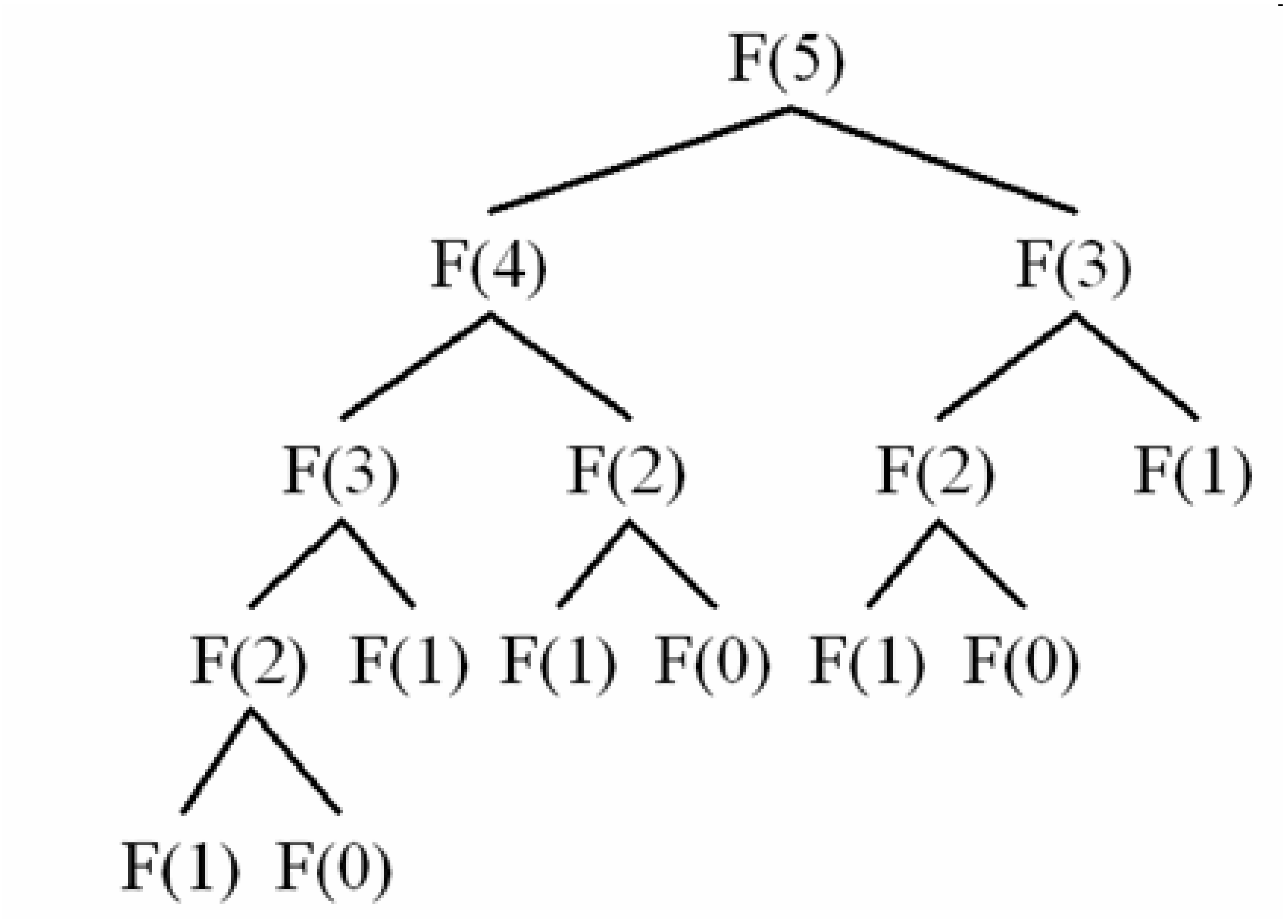
We can easily write a C++ function to implement the Fibonacci sequence easily:-

- We start with the **base cases** when *n == 0* and when *n == 1.*
- Then we handle the **general case** when *n >= 2.*

```
int fib(int n){
    //base case (essintial to end the recursion)
    if (n == 0 || n == 1)
        return 1;
    //general case
    //the function calls itself and calculates a
    //partial result and combines it with the other
    //part the it returns it's own result
    return fib(n-1) + fib(n-2);
}
```

We start by writing down the **base case** then move on to define the **general case** that includes the function calling itself with **different parameters.**

if we call the function with n=5; fib(5), then we will get the following trace:-



At each stage the function **calls itself twice** and gets two results and then it **adds** them to get the full results then it returns that result to the **caller.**

We can also turn a function that is written without recursion (imperative function) into a recursive function by turning loops into recursion.

For example:-

```
//return the sum of the array
int sum(int arr[], int SIZE){
    int result = 0;
    for(int i = 0; i < SIZE; i++)
        result += arr[i];
    return result;
}

//recursive implementation
int sum(int arr[], int SIZE, int i = 0){
    //when i goes out of range we end the recursion
    //and return zero which doesn't affect the sum
    if (i >= SIZE)
        return 0;
    //we add arr[i] to sum(i+1..n) and return
    return arr[i] + sum(arr, SIZE, i + 1);
}
```

We can turn loops into recursion with minimal effort.

Lets now try to guess what is the output if we are given the a recursive function and it’s initial parameters.

For the following function:-

```
int func(int x, int y){
    if (x == 0)
        return y;
    else
        return func(x - 1, x + y);
}
```

```
What is the output for func(5, 2) ?

To answer such question we must calculate the
values of the parameters at each stage

First iter: x=5, y=2
Second iter: x=4, y=2+5
Third iter: x=3, y=2+5+4
Forth iter: x=2, y=2+5+4+3
Fifth iter: x=1, y=2+5+4+3+2
Final iter: x=0, y=2+5+4+3+2+1, return y

And the result is y=2+5+4+3+2+1 = 17

To answer questions like these we need to trace
the function’s execution at each iteration and
calculate each partial result until we get to
the base case.
```