# Data Structures & Algorithms

## MODULE 1: GROWTH OF FUNCTIONS

### TOPIC: Complexity Analysis of Recursive Functions

# Recursive Algorithm

A recursive algorithm is an algorithm that solves a problem by breaking it down into smaller instances of the same problem, and then solving those smaller instances.

| Direct Recursion | Indirect Recursion |
|---|---|
| Occurs when an algorithm calls itself | Occurs when a function calls another function |
| **Example:**<br>Fact (n):<br>    if n==0:<br>        return 1<br>    else:<br>        return n*Fact(n-1) | Example:<br>A(n):<br>  if n % 3 == 0:<br>    return true<br>  else:<br>    return B(n - 1)<br><br>B(n):<br>  if n % 3 == 1:<br>    return true<br>  else:<br>    return A(n - 1)<br><br>A(5) |

# Properties of Recursive Function

**A recursive** function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

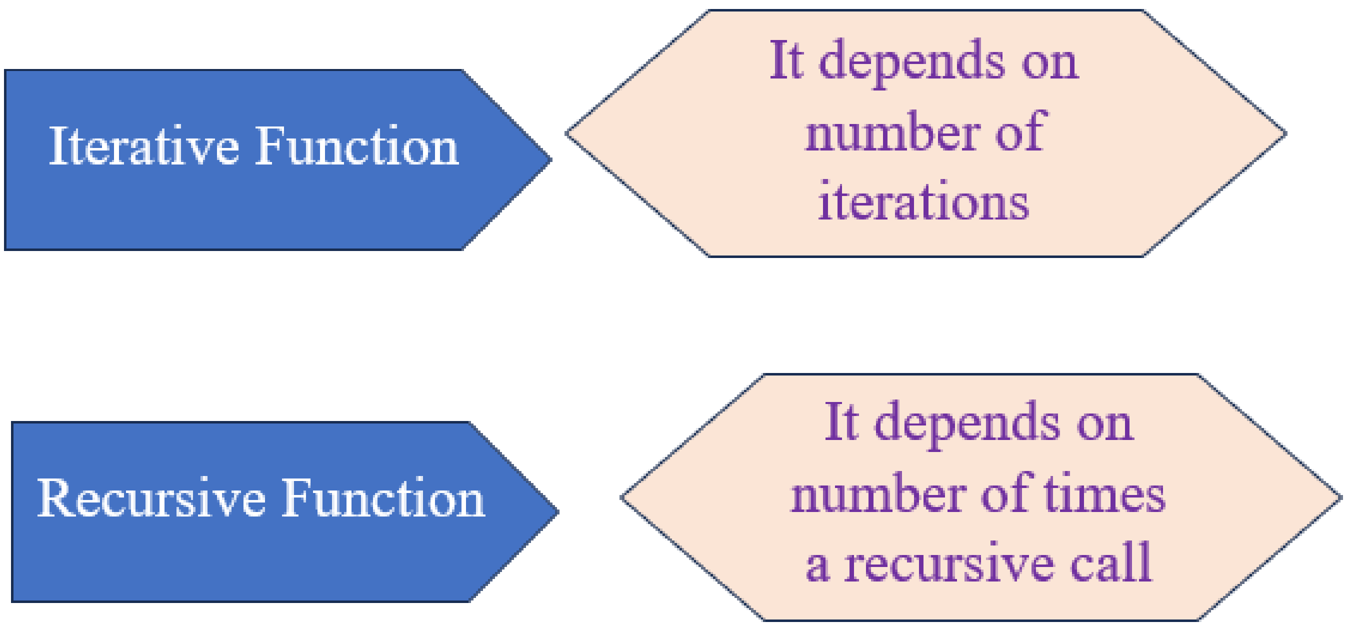
Base Criteria

Recursive Case

```
Fact (n):
    if n==0:
        return 1
    else:
        return n*Fact(n-1)
```

# Time Complexity

**Time complexity** is the amount of time it takes to run an algorithm.



# Space Complexity

**Iterative Function**

Hardly requires any extra space

**Example**

```
Sum = 0;
For i = 1 to n
    Sum = Sum + i;
return Sum
```

No extra space required

**Recursive Function**

Requires more space

**Example**

```
Fact (n):
    If n==0:
        return 1
    Else:
        return n*Fact(n-1)
```

| n=5 |
| --- |
| Fact (1) |
| Fact (2) |
| Fact (3) |
| Fact (4) |
| Fact (5) |

# Finding Complexity using TFC

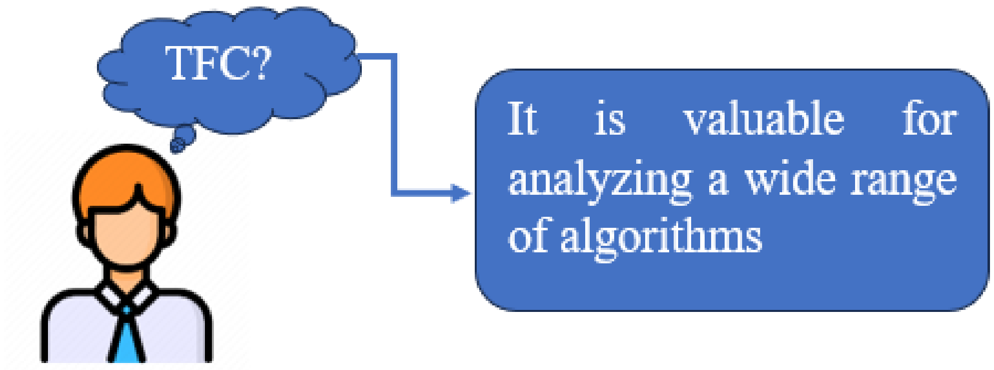
We can find complexity of iteration algorithm using frequency count.

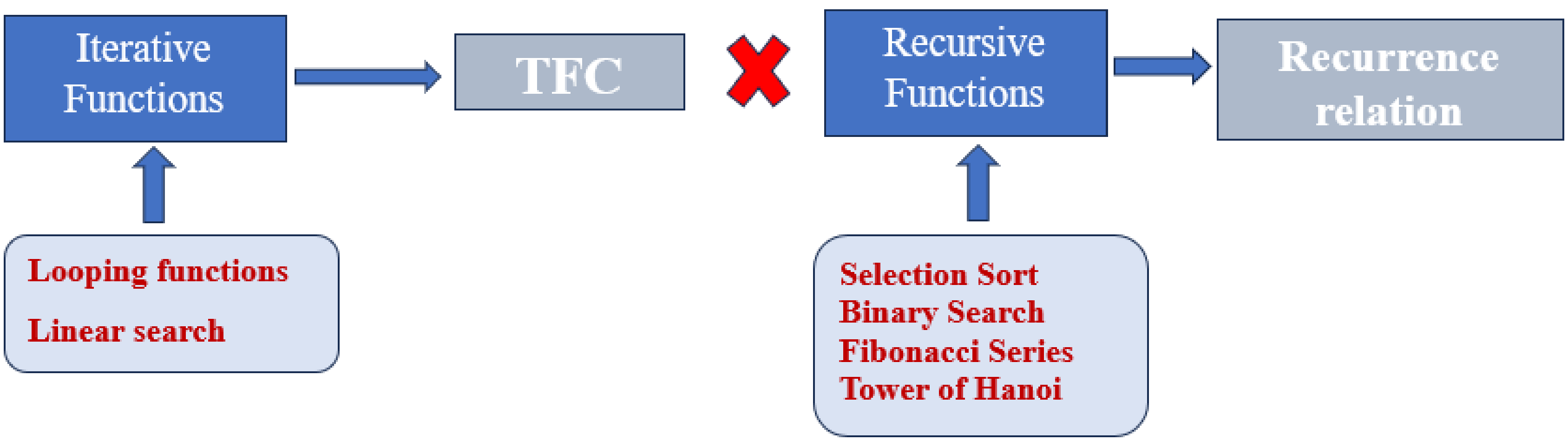
**For example:**

| | |
|---|---|
| for(i=1;i<n;i++) | n times |
| { | |
|     for(j=1 ; j<n ;j++) | (n-1) n times |
|     { | |
|         Statement; | $n^2$ times |
|     } | |
| } | |

Then the complexity of algorithm is $O(n^2)$.

# Why Recurrence Relation?



# What is a Recurrence Relation?

These recursive formulas are referred to as recurrence relations which are solved by repeated substitutions method.

A recurrence relation, T(n), is a recursive function of an integer variable n.

Formula:

$$T(n) = \begin{cases} 1 & if\ n = 0 \\ 1 + T(n-1) & for\ n > 0 \end{cases}$$

The portion of the definition that does not contain T is called the base case of the recurrence relation.

The portion that contains T is called the recurrent or recursive case

# Example 1: Factorial of Number

```
Fact (n)
{
    if n==0
        return 1
    else
        return n * Fact(n-1)
}
```

$$T(n) = T(n-1) + 3 \quad \text{if } n>0$$
$$T(0) = 1 \quad \text{if } n==0$$

$$T(n) = T(n-1) + 3$$
$$= T(n-2) + 6$$
$$= T(n-3) + 9$$
$$= T(n-k) + 3k$$

Simplify the eq

Consider constant as k

$$n-k=0 \Rightarrow k=n$$

$$\Rightarrow T(n) = T(n-n) + 3k$$
$$= T(0) + 3n$$
$$= 1 + 3n$$

$$T(n) \propto n \Rightarrow O(n)$$

The time complexity is directly proportional to n, then we can take Big Oh **O(n)**

# Example 2: Fibonacci Sequence

Fibonacci Sequence 0, 1, 1, 2, 3, 5, 8.. ..

```
Fib (n)
{
    if n <= 1
        return n
    else
        return Fib(n-1) + Fib(n-2)
}
```

$$T(n) = T(n-1) + T(n-2) + 4$$

$$T(0) = T(1) = 1$$

$$T(n-1) \approx T(n-2)$$

Time taken to calculate n-2 is lesser than n-2

$$T(n) = 2T(n-2) + c$$

$$= 2\{2T(n-4) + c\} + c$$

Simplify the Eq

$$= 4T(n-4) + 3c$$

$$= 8T(n-6) + 7c$$

$$= 16T(n-8) + 15c$$

$$T(n) = 2^k T(n-2k) + (2^k - 1)c$$

Find the expression from above

$$n - 2k = 0 \Rightarrow k = n/2$$

$$T(n) = 2^{n/2} T(n - 2(n/2)) + (2^{n/2} - 1)c$$

$$= 2^{n/2} T(0) + (2^{n/2} - 1)c$$

$$= 2^{n/2} + 2^{n/2} c - c$$

$$= 2^{n/2}(1 + c) - c$$

$$T(n) \propto 2^{n/2} \text{ (Lower bound)}$$

Lower Bound

# Example 2: Fibonacci Sequence

```
Fib(n)
{
  if(n<=1)
      return n
  else
      return Fib(n-1) + Fib(n-2)
}
```

$$T(n) = T(n-1) + T(n-2) + 4$$

$$T(0) = T(1) = 1$$

$$T(n-2) \approx T(n-1)$$

Time taken to calculate n-1 is greater than n-2

$$T(n) = 2T(n-1) + c$$
$$= 4T(n-2) + 3c$$
$$= 8T(n-3) + 7c$$

Simplify the Eq

$$= 2^k T(n-k) + (2^k - 1)c$$

Find the expression from above

$$n - k = 0 \Rightarrow k = n$$

$$T(n) = 2^k T(0) + (2^n - 1)c$$

$$T(n) = (1+c)2^n - c$$

$\therefore T(n) \propto 2^n$ (Upper Bound) → Upper Bound

From Previous: $T(n) \propto 2^{n/2}$ (Lower Bound) → Lower Bound

Time complexity **O($2^n$)** -> Fib (recursive) → Exponential Time Algorithm

**O(n)** -> Fib (Iterative) → Linear Time Algorithm

# Problem to Solve

1. Test (int n) {

```
        if (n>0){
                print (n);
                Test(n-1)
        }
    }
```

2. Test (int n)

```
    {
        if (n>0){
                for (i=0; i<n; i++){
                        print(n)
                    }
                Test(n-1)
        }
}
```

# Problem to Solve

3. Test (int n)

```
{
    if (n>0){
        for (i=1; i<n; i=i*2){
            print(i)
        }
        Test(n-1)
    }
}
```

# Problem to Solve

$$T(n)=T(n/2)+O(1)$$

4. Binary_search(arr, target, low, high):

```
if low > high:
    return -1
mid = (low + high) // 2
if arr[mid] == target:
    return mid
elif arr[mid] > target:
    return Binary_search(arr, target, low, mid - 1)
else:
    return Binary_search(arr, target, mid + 1, high)
```

# Problem to Solve

$$T(n)=2T(n-1)+O(1)$$

5. Tower_of_hanoi(n, source, target, auxiliary):

```
if n == 1:
    print("Move disk 1 from {source} to {target}")
else:
    Tower_of_hanoi(n-1, source, auxiliary, target)
    print("Move disk {n} from {source} to {target}")
    Tower_of_hanoi(n-1, auxiliary, target, source)
```

# Thank You