



Recursion

[Introduction](#)

[Types of Recursions](#)

[Tail Recursion](#)

[Head Recursion](#)

[Tree Recursion](#)

[Indirect Recursion](#)

[Exercises](#)

[Sum of N Numbers](#)

[Factorial](#)

Introduction

The fact here is that the function call to itself.

In this case, generally, the iteration is created by a condition, for example, in the iterative form we use the While loop.

Here we can see the iterative form.

```
public void CalculateIterative(int n)
{
    while (n > 0)
    {
        int k = n * n;
        Console.WriteLine(k);
        n--;
    }
}
```

Next an example of recursive for.

```
public void CalculateRecursive(int n)
{
    if (n > 0)
    {
        int k = n * n;
        Console.WriteLine(k);
        CalculateRecursive(n - 1);
    }
}
```

Now, the time complexity of the Recursive Function is calculated using the n time execution. For example, consider the last function.

```
function calculate(n) <- T(n)
  if n > 0 then      <- 1
    k := n ^ 2      <- 1
    print(k)        <- 1
    calculate(n-1)   <- T(n-1)
```

First, we need to calculate the $T(n)$ case, following the next.

$$\begin{aligned}
 T(n) &= 1 + 1 + 1 + T(n-1) \\
 T(n) &= T(n-1) + 3, \quad n > 0 \\
 T(n-1) &= T(n-2) + 3 \\
 T(n-2) &= T(n-3) + 3 \\
 &\vdots \\
 T(1) &= T(0) + 3 \\
 T(0) &= 1
 \end{aligned}$$

Now We can replace the cases.

$$\begin{aligned}
 T(n) &= T(n-1) + 3 \\
 T(n) &= (T(n-2) + 3) + 3 \\
 T(n) &= ((T(n-3) + 3) + 3) + 3 \\
 T(n) &= (T(0) + 3 + 3 + \dots + 3)
 \end{aligned}$$

So, thanks to Substitution we can find the Time Complex.

$$T(n) = 1 + 3n \rightarrow O(n) = n$$

```
function someFunction(n):
  ...
```

Suppose that the function algorithm cost in n is $T(n) = T(n-1) + n$ and, $T(1) = 1$.

Now we calculate using the next steps.

$$\begin{aligned}
 T(n) &= T(n-1) + n \\
 T(n-1) &= T(n-2) + (n-1) \\
 T(n-2) &= T(n-3) + (n-2)
 \end{aligned}$$

Now making substitution we have the next equation.

$$\begin{aligned}
 T(n) &= T(n-1) + n \\
 T(n) &= T(n-2) + (n-1) + n \\
 T(n) &= T(n-3) + (n-2) + (n-1) + n
 \end{aligned}$$

$$\begin{aligned}
 &\vdots \\
 T(n) &= T(n-k) + (n-k-1) + \dots + (n-2) + (n-1) + n
 \end{aligned}$$

With the last equation, we need to analyze the base case, i.e.
 $n - k = 1 \rightarrow k = n - 1$

$$\begin{aligned}
 T(n) &= T(n - (n-1)) + (n - (n-1) - 1) + \dots + (n-1) + n \\
 T(n) &= T(1) + 0 + \dots + (n-2) + (n-1) + n \\
 T(n) &= 1 + \dots + (n-2) + (n-1) + n
 \end{aligned}$$

$$\begin{aligned}
 T(n) &= \sum_{k=1}^n k = \frac{n(n+1)}{2} \\
 T(n) &= \frac{n^2 + n}{2} \rightarrow O(n) = n^2
 \end{aligned}$$

Types of Recursions

Tail Recursion

This is the commonest recursion type, here we improve the logic before executing the next call of the recursive function.

```
public void TailRecursion(int n)
{
    if (n > 0)
    {
        int k = n * n;
        Console.WriteLine(k);
        TailRecursion(n - 1);
    }
}
```

```
Tail Recursion
25
16
9
4
1
```

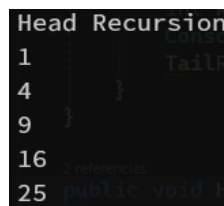
Head Recursion

Here first we execute the next call of the recursive function, after, we improve the logic, in this case, the logic will be executed when all the calling recursions end.

```
public void HeadRecursion(int n)
{
    if (n > 0)
    {
        HeadRecursion(n - 1);
        int k = n * n;
        Console.WriteLine(k);
    }
}
```

In this case, the calling stack will be.

Cal(4)	← Call Back			
- end	Cal(3)	← Call Back		
	- k = 16 - "16"	Cal(2)	← Call Back	
		- k = 9 - "9"	Cal(1)	← Call Back
			- k = 4 - "4"	Cal(0)
				- k = 1 - "1"



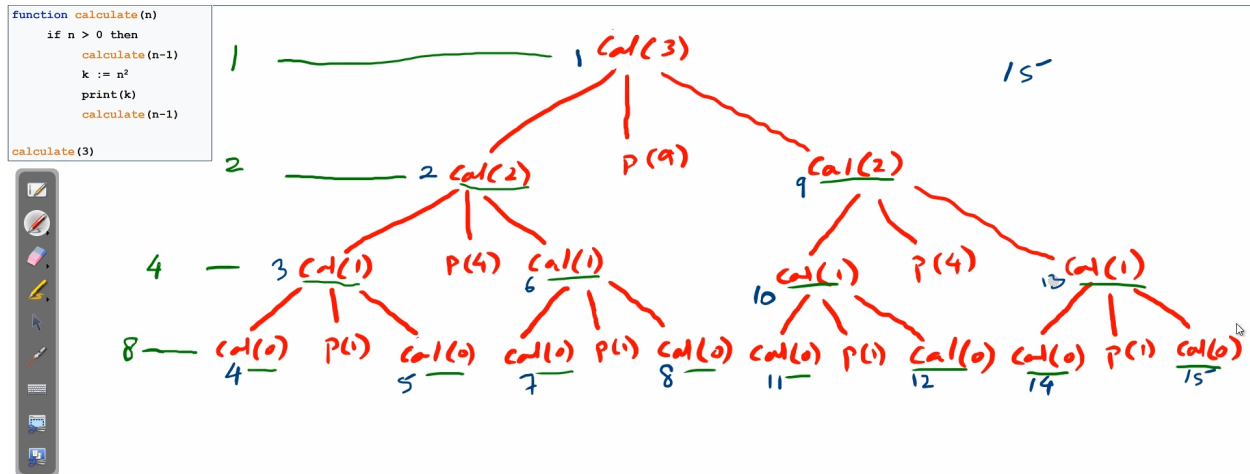
Note: This is like an accumulative output call.

Tree Recursion

In the tree recursion, we have more than one recursion call.

```
public void TreeRecursion(int n)
{
    if (n > 0)
    {
        TreeRecursion(n - 1);
        int k = n * n;
        Console.WriteLine(k);
        TreeRecursion(n - 1);
    }
}
```

In this case, the calling tree will be.



Now, the complexity in this case we are going to count the number of recursions, in this case, is 15 but is 15 for $n=3$, now suppose that $n=k$, now, we need to analyze each level in the tree, i.e. we have the following equation.

$$T(3) = 1 + 2 + 4 + 8$$

$$T(3) = 2^0 + 2^1 + 2^2 + 2^3$$

$$T(3) = \sum_{k=0}^3 2^k$$

$$T(n) = \sum_{k=0}^n 2^k \rightarrow O(n) = 2^n$$

This is the output for $n=3$.

```
Tree Recursion
1
4
1
ic void Run()
9
ursion typeRecur
4 WriteLine("Tail
ursion.TailRecur
1 WriteLine("Head
```

Indirect Recursion

This recursion is based on calling another function that calls the function from.

```
public void IndirectRecursionA(int n)
{
    if (n > 0)
    {
        int k = n * n;
        Console.WriteLine(k);
        IndirectRecursionB(n - 1);
    }
}

public void IndirectRecursionB(int n)
{
    if (n > 0)
    {
        int k = n * n;
        Console.WriteLine(k);
    }
}
```

```

    IndirectRecursionA(n - 1);
}
}

```

Exercises

Sum of N Numbers

Create an algorithm that sums the n terms, where n is a Natural Number.

First, we need to consider that the sum of the first n numbers can be written as follow.

$$\begin{aligned}
 sum(n) &= n + sum(n - 1) \\
 sum(n - 1) &= (n - 1) + sum(n - 2) \\
 &\vdots \\
 sum(1) &= 1
 \end{aligned}$$

Here the base case is $sum(1) = 1$ and the n case $sum(n) = n + sum(n - 1)$.

Using recursion we can improve the next algorithm.

```

public int SumOfN(int n)
{
    if (n == 1) return 1;
    return n + SumOfN(n - 1);
}

```

Now we are going to analyze the algorithm based on the next conditions.

$$T(n) = 1 + 1 + T(n - 1) \quad T(1) = 1$$

We are going to use substitution to get the algorithm cost.

$$\begin{aligned}
 T(n) &= 2 + T(n - 1) \\
 T(n - 1) &= 2 + T(n - 2) \\
 T(n - 2) &= 2 + T(n - 3) \\
 &\vdots \\
 T(1) &= 1
 \end{aligned}$$

Next, using the substitution we have the next equation.

$$\begin{aligned}
 T(n) &= 2 + T(n - 1) \\
 T(n) &= 2 + (2 + T(n - 2)) \\
 T(n) &= 2 + (2 + (2 + T(n - 3))) \\
 &\vdots \\
 T(n) &= 2 + 2 + 2 + \dots + 2 + 1 \\
 T(n) &= 2n + 1 \rightarrow O(n) = n
 \end{aligned}$$

Factorial

The factorial of n supplies the next formula.

$$fac(n) = n * (n - 1) * (n - 2) * (n - 3) \dots 1$$

First, we are going to consider the factorial with the next conditions.

$$fac(n) = n * fac(n - 1) \quad fac(1) = 1$$

With the base condition and the n condition, we can create the next algorithm.

```
public int Factorial(int n)
{
    if (n == 1) return 1;
    return n * Factorial(n - 1);
}
```

Now we are going to analyze the complexity of this algorithm.

$$T(n) = 1 + 1 + T(n - 1) \quad T(1) = 1$$

We are going to use substitution to calculate the complexity.

$$\begin{aligned} T(n) &= 2 + T(n - 1) \\ T(n) &= 2 + (2 + T(n - 2)) \\ T(n) &= 2 + (2 + (2 + T(n - 3))) \\ &\vdots \\ T(n) &= 2 + 2 + 2 + \dots + 2 + 1 \\ T(n) &= 2n + 1 \longrightarrow O(n) = n \end{aligned}$$