

# Analysis of Recursion, designing of Recursion

**Recursion** is a process in which a method calls itself again and again. We use recursion to solve bigger problem by dividing it into smaller similar sub-problems and then call them recursively.

Syn:

```
returntype datatype meth(Para_list)
```

```
{
```

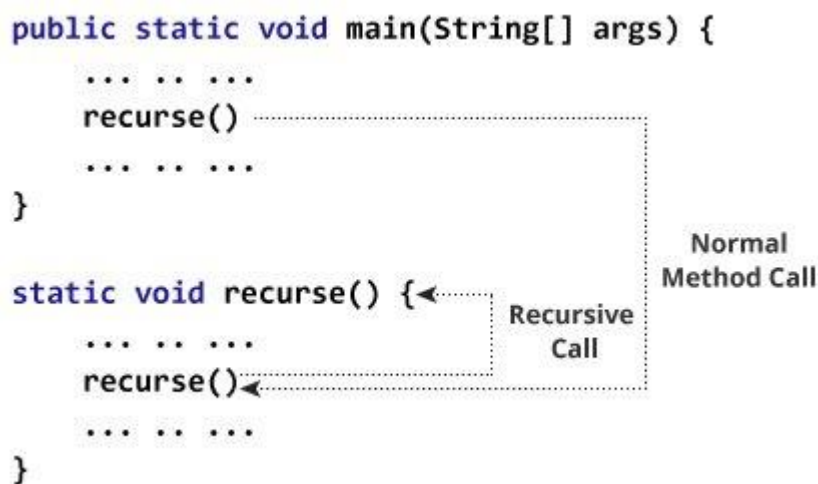
```
Stmts;
```

```
meth(Para_list);
```

```
}
```

## Ex: How Recursion works?

```
public static void main(String[] args) {  
    ... ..  
    recurse() .....  
    ... ..  
}  
  
static void recurse() {  
    ... ..  
    recurse().....  
    ... ..  
}
```



Working of Java

Recursion

In the above example, we have called the `recurse()` method from inside the `main` method. (normal method call). And, inside the `recurse()` method, we are again calling the same `recurse` method. This is a recursive call. In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.

Generally speaking, recursion is the concept of *well-defined self-reference*. It is the determination of a succession of elements by operating on one or more preceding elements according to a rule or a formula involving a finite number of steps.

In computer science, recursion is a programming technique using methods or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first.

A recursive method that calls itself. Notice that there's a case when the method does not call itself recursively, otherwise, the method will keep calling itself and will never stop to return a value.

Thus, a recursive method usually has a certain structure:

- (1) a **base case**, which does not call the method itself; and
- (2) a **recursive step**, which calls the method itself and moves closer to the base case.

**Important: Every recursion must have at least one base case, at which the recursion does not recur** (i.e., does not refer to itself).



More examples of recursion:

- Russian Matryoshka dolls. Each doll is made of solid wood or is hollow and contains another Matryoshka doll inside it.
- Modern OS defines file system directories in a recursive way. A file system consists of a top-level directory, and the contents of this directory consists of files and other directories.
- Much of the syntax in modern programming languages is defined in a recursive way. For example, an argument list consists of either (1) an argument or (2) an argument list followed by a comma and an argument.

## Types of recursion with example

Recursive methods can be classified on the basis of

**a)** Based on methods call itself –

Direct / Indirect

**b)** Based on pending operation at each recursive call –

Tail Recursive/ Head Recursive

c) Based on the structure of the method calling pattern –

Linear /Binary/ Tree

a) Based on methods call itself – Direct /  
Indirect  
Direct Recursion

- If a method calls itself from within itself is known as Direct Recursion.
- When the method invokes itself it is direct.

```
1 Static int methc(int num) {  
2     if (num == 0)  
3         return 1;  
4     else  
5         return methc(num - 1);  
6 }
```

Here, method 'methc' is calling itself. This is an example of direct recursion.

Indirect Recursion

- In Indirect Recursion, one method call one another mutually in a circular manner.
- For example, if a method 'meth1()' , calls method 'meth2()', which calls method 'meth2()' which again leads to 'meth1()' being invoked is called indirect recursion.

```
6 class Recursion {  
7     static void meth1(int n)  
8     {  
9         if (n > 0) {  
10             System.out.println("%d ", n);  
11             meth2(n - 1);  
12         }  
13     }  
14 }
```

```

12    }
13}
14
15Static void meth2(int n)
16{
17    if (n > 1) {
18        System.out.println("%d ", n);
19        meth1(n - 2);
20    }
21}
22
23public static void main(String[] args) {
24    meth1(10);
25    return 0;
26}

```

b) Based on pending operation at each recursive call – Tail Recursive/ Head Recursive  
Tail / Bottom Recursion

- Tail Recursion is an example of Direct Recursion, If a recursive method is calling itself and that recursive call is the last statement in the method then it's known as Tail Recursion.
- We can also say that if no operations are pending when the recursive method returns to its caller.
- It is an efficient method as compared to others, as the stack space required is less and even compute overhead will get reduced.
- After the call, the recursive method performs nothing.

```

1class Recursion {
2
3static void meth(int n)
4{
5    if (n > 0) {
6        System.out.println("%d ", n);

```

```

7      meth(n - 1); // Last statement in the method
8  }
9}
10
11public static void main(String[] args) {
12    meth(5);
13    return 0;
14}

```

**Time Complexity For Tail Recursion:**  $O(n)$

**Space Complexity For Tail Recursion:**  $O(n)$

### Head/Top Recursion

- Top Recursion is an example of Direct Recursion, If a recursive method is calling itself and that recursive call is the first statement in the method then it's known as Head Recursion.
- Nothing will perform before calling of a recursive method.

```

1class Recursion {
2    Static void meth(int n)
3    {
4        if (n > 0) {
5            meth(n - 1); // First statement in the method
6            System.out.println("%d ", n);
7        }
8    }
9    Public static void main(String args[])
10   {
11       meth(5);
12       return 0;
13   }
14}

```

**Time Complexity For Head Recursion:**  $O(n)$

**Space Complexity For Head Recursion:**  $O(n)$

- c) Based on the structure of the method calling pattern –
1. Linear /
  2. Binary/

### 3.Tree(multiple)

## Linear Recursion

- A linear recursive method is a method that only makes a single call to itself each time the method runs.
- Our Factorial recursive method is a suitable example of linear recursion as it only involves multiplying the returned values and no further calls to the method.

Below is an example of Linear Recursion

#### Example 1: Factorial Calculation

We know that the factorial of  $n$

( $n \geq 0$ ) is calculated by  $n! = n * (n-1) * (n-2) * \dots * 2 * 1$ .

Note that the product of  $(n-1) * (n-2) * \dots * 2 * 1$  is exactly  $(n-1)!$ .

Thus we can write the expression as  $n! = n * (n-1)!$ , which is the recursive expression of the factorial calculation.

```
class Factorial {  
  
    static int factorial( int n ) {  
        if (n != 0) // termination condition  
            return n * factorial(n-1); // recursive call  
        else  
            return 1;  
    }  
  
    public static void main(String[] args) {  
        int number = 4, result;  
        result = factorial(number);  
        System.out.println(number + " factorial = " + result);  
    }  
}
```

**Output:**

```
4 factorial = 24
```

In the above example, we have a method named `factorial()`.

The `factorial()` is called from the `main()` method. with the `number` variable passed as an argument.

Here, notice the statement,

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

The `factorial()` method is calling itself. Initially, the value of `n` is 4

inside `factorial()`. During the next recursive call, 3 is passed to

the `factorial()` method. This process continues until `n` is equal to 0.

When `n` is equal to 0, the `if` statement returns false hence 1 is returned.

Finally, the accumulated result is passed to the `main()` method.

## The time-complexity of recursive factorial(Linear Recursion) would be:

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

So,

The time complexity for one recursive call would be:

$T(n) = T(n-1) + 3$  (3 is for As we have to do three constant operations like multiplication, subtraction and checking the value of `n` in each recursive call)

=  $T(n-2) + 6$  (Second recursive call)

=  $T(n-3) + 9$  (Third recursive call)

.

.  
.
.

= T(n-k) + 3k

till, k = n

Then,

= T(n-n) + 3n

= T(0) + 3n

= 1 + 3n

To represent in Big-Oh notation,

T(N) is directly proportional to n,

Therefore, The time complexity of recursive factorial is O(n). As there is no extra space taken during the recursive calls,the space complexity is O(n).

Tree recursion

- In Tree recursion, Instead of a single method call there are two recursive calls for each non-base case.
• Methods with two recursive calls are referred to as binary recursive methods.
• In Tree recursion, there are pending operations that involve another recursive call to the method.

Below is an example of Tree Recursion

1 class TreeorBinaryRecursion
2 {
3 static void meth(int n)
4 {
5 if (n > 0) {
6 System.out.println(n);



```

7      meth(n - 1); //calling first time
8      meth(n - 2); //calling second time
9  }
10}
11 public static void main(String args[])
12{
13     meth(10);
14     //return 0;
15}
16}
17 Fibonacci series is also a example of tree/binary recursion
18
19 int fibonacci(int i){
20     if(i==0) return 0;
21     else if(i==1) return 1;
22     else return (fibonacci(i-1)+fibonacci(i-2));
23}
24

```

## Time Complexity of Fibonacci series (Binary/ Tree Recursion)

The time complexity of the iterative code is linear, as the loop runs from 2 to n, i.e. it runs in  $O(n)$  time

Calculating the time complexity of the recursive approach is not so straightforward, so we are going to dive in

```

fib(n):
    if n <= 1
        return 1
    return fib(n - 1) + fib(n - 2)

```

for  $n > 1$ :

$T(n) = T(n-1) + T(n-2) + 4$  (1 comparison, 2 subtractions, 1 addition)

```

T(n) = T(n-1) + T(n-2) + c
      = 2T(n-1) + c      //from the approximation T(n-1) ~ T(n-2)

```

```

= 2 * (2T(n-2) + c) + c
= 4T(n-2) + 3c
= 8T(n-3) + 7c
= 2^k * T(n - k) + (2^k - 1)*c
Let's find the value of k for
which: n - k = 0
k = n
T(n) = 2^n * T(0) + (2^n - 1)*c
= 2^n * (1 + c) - c i.e. T(n) ~ 2^n

```

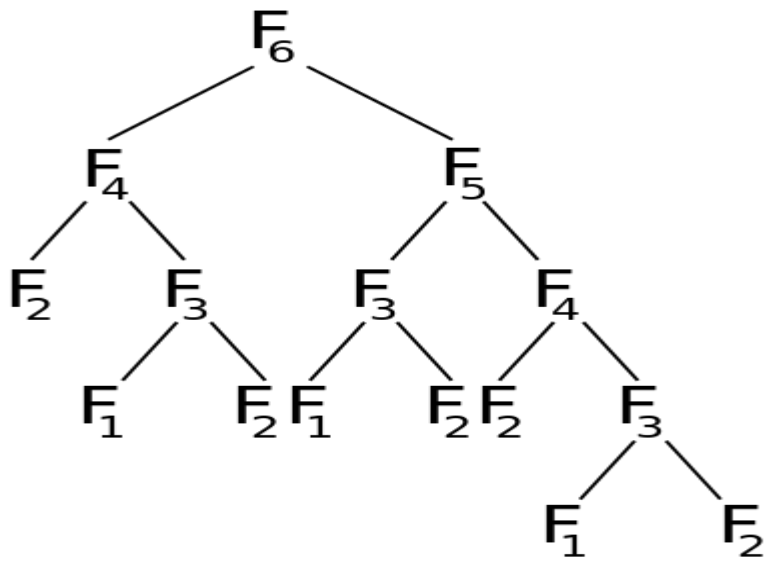
Hence the time taken by recursive Fibonacci is  $O(2^n)$  or exponential.

### Space Complexity:

For the iterative approach, the amount of space required is the same for fib(6) and fib(100), i.e. as N changes the space/memory used remains the same. Hence its space complexity is  $O(1)$  or constant.

For Fibonacci recursive implementation or any recursive algorithm, the space required is proportional to the maximum depth of the recursion tree, because, that is the maximum number of elements that can be present in the implicit function call stack.

Below is a diagrammatic representation of the Fibonacci recursion tree for fib(6):



Recursion tree for Fibonacci of 6

As you can see the maximum depth is proportional to the  $N$ , hence the space complexity of Fibonacci recursive is  $O(N)$ .

**Time Complexity For Tree Recursion:**  $O(2^n)$

**Space Complexity For Tree Recursion:**  $O(n)$