1. Direct Recursion: A method directly calls itself.

```
public class Factorial {

    public static int factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int result = factorial(5);
        System.out.println(result);
    }
}
```

Examples:
- Factorial calculation
- Fibonacci series
- Tower of Hanoi

- used for problems that can be ***naturally divided into smaller subproblems***.

2. Indirect Recursion: A method calls another method, which eventually calls the original method.

```
public class IndirectRecursion {

    public void methodA() {
        // ... some code
        methodB();
    }

    public void methodB() {
        // ... some code
        methodA();
    }
}
```

Example:
- A function **A** calls function **B,** and function ***B*** calls function ***A***.

- it is less common but can be useful for solving problems tha***t involve cyclic dependencies***.

3. Tail Recursion: A recursive call is the last operation in a method. This can be optimized by the compiler to avoid stack overflow.

```
public class TailRecursion {

    public static int factorial(int n, int result) {
        if (n == 0) {
            return result;
        } else {
            return factorial(n - 1, result * n);
        }
    }

    public static void main(String[] args) {
        int result = factorial(5, 1);
```

Example:
- Reverse a string using tail recursion.
-

`System.out.println(result);`
`}`
`}`	

- Tail recursion can be ***optimized to improve performance***, but it's not always applicable

---

GFG

Types of **Recursions**:
Recursion are mainly of **two types** depending on whether **a function calls itself from within itself** or **more than one function call one another mutually**.

1. Direct Recursion
   a. Tail Recursion
   b. Head Recursion
   c. Tree Recursion
   d. Nested Recursion
2. Indirect Recursion

Direct Recursion

- A function calls itself directly.

Can be further categorized into four types:

- **Tail Recursion**: The recursive call is the last statement in the function. This is an efficient form of recursion that can be optimized by compilers.

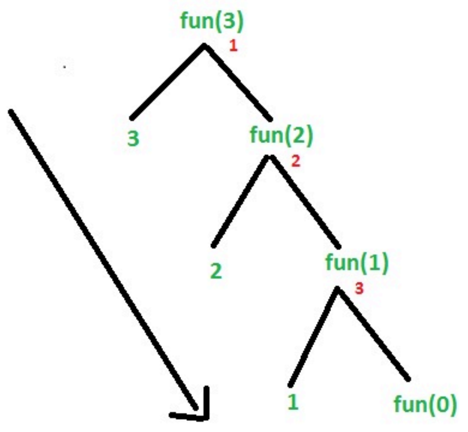
  Example: Printing numbers in descending order.

`static void fun(int n)`
`{`
`if (n > 0)`
`{`
`System.out.print(n + " ");`
 	
`// Last statement in the function`
`fun(n - 1);`
`}`
`}`
	int x = 3;
	fun(x);
	--- 	
	Output
	3 2 1
		Time Complexity For Tail Recursion : O(n)
		Space Complexity For Tail Recursion : O(n)

  - If do same thing using loop and change in Complexity TC & SC

`static void fun(int y)`
`{`
`while (y > 0) {`
`System.out.print(" "+ y);`
`y--;`
`}`
`}`
	int x = 3;
	fun(x);
	
	Output
	3 2 1
	Time Complexity: O(n)
	Space Complexity: O(1)

Tracing Tree Of Recursive Function



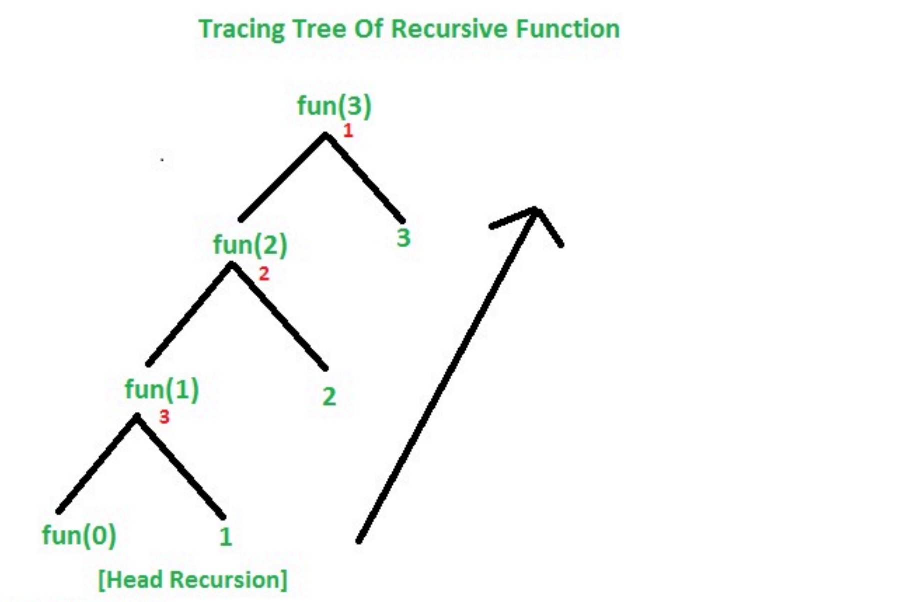
[Tail Recursion]

Output: 3 2 1
*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

- Head Recursion: The recursive call is the first statement in the function. Can be less efficient than tail recursion.

  Example: Printing numbers in ascending order.

static void fun(int n)
{
  if (n > 0) {
 // First statement in the function
     fun(n - 1);
 System.out.print(" "+ n);
  }
	int x = 3;
	  fun(x);
	
	Output
	 1 2 3
	Time Complexity O(n)
	Space Complexity: O(n)
		Using loop …
		static void fun(int n)
		{
		  int i = 1;
		  while (i <= n) {
		    System.out.print(" "+ i);
		    i++;
		  }
		}
			TC -O(n)
			SC -O(1)



Output: 1 2 3
*Digits in red showing that the order in which the calls are made and note that printing done at returning time. And it does nothing at calling time.

- Tree Recursion: A function calls itself multiple times within a single call.
  To understand **Tree Recursion** let's first understand **Linear Recursion.** If a recursive function calling itself for one time then it's known as **Linear Recursion**.

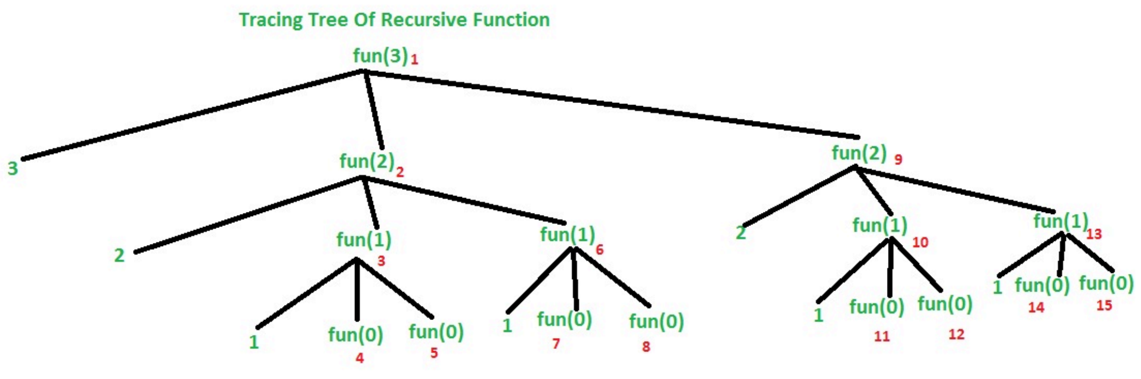
```
fun(n)
{
    // some code
    if(n>0)
    {
        fun(n-1); // Calling itself only once
    }
    // some code
}
```

  Example: Implementing a binary search tree.

```
// Recursive function
static void fun(int n)
{
    if (n > 0) {
        System.out.print(" "+ n);

        // Calling once
        fun(n - 1);

        // Calling twice
        fun(n - 1);
    }
}
```

```
fun(3);

Output
 3 2 1 1 2 1 1
```

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



[Tree Recursion]

Output: 3 2 1 1 2 1 1
***Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.**

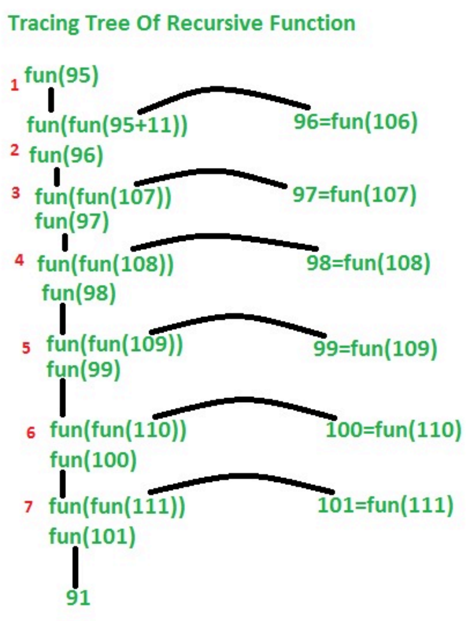
In this recursion, a recursive function will pass the parameter as a recursive call. That means "**recursion inside recursion"**.

- Nested Recursion: A recursive function calls itself as a parameter..

  Example: Calculating a complex mathematical formula.

```
static int fun(int n)
{
   if (n > 100)
      return n - 10;

   // A recursive function passing parameter
   // as a recursive call or recursion
   // inside the recursion
   return fun(fun(n + 11));
}
```

```
int r;
   r = fun(95);
   System.out.print("  "+ r);

Output
 91
```



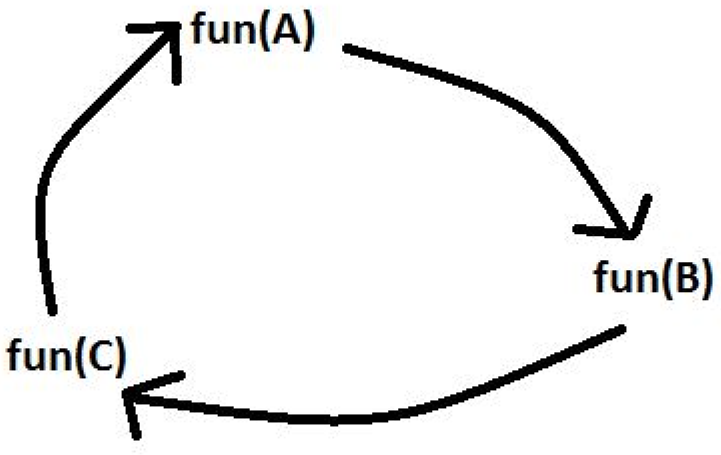
[Nested Recursion]

Output: 91
*Digits in red showing that the order in which the calls are made

---

Indirect Recursion

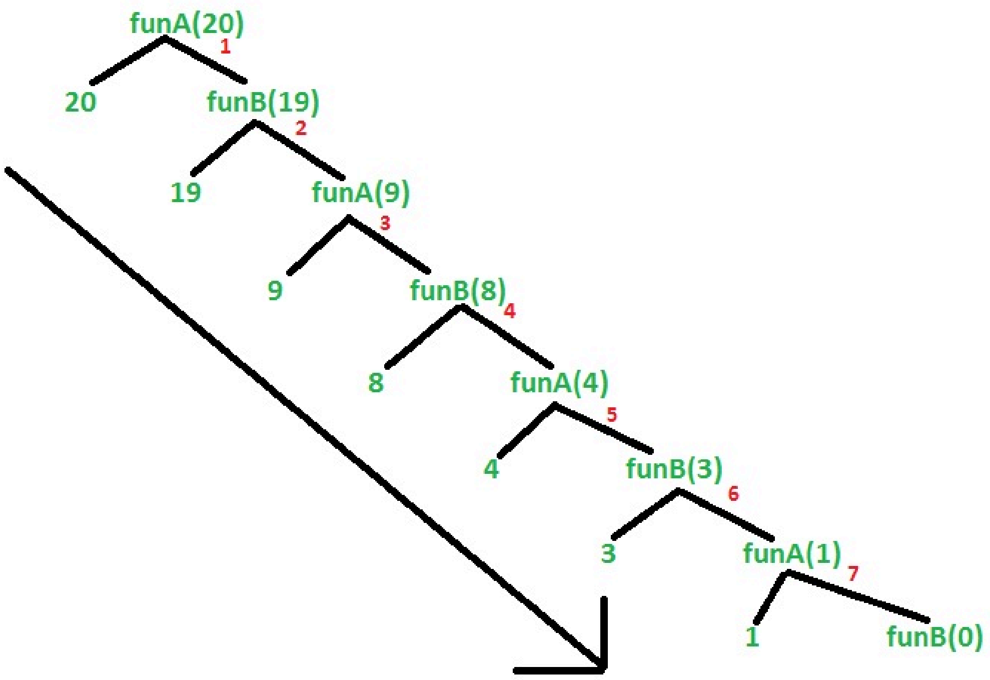
- Two or more ***functions call each other in a circular manner***.
- Can be ***less efficient than direct recursion*** and ***may lead to stack overflow errors*** if not carefully implemented.
- Example: Simulating a conversation between two agents.



static void funA(int n)
{
    if (n > 0) {
        System.out.print(" " +n);
 	
        // Fun(A) is calling fun(B)
        funB(n - 1);
    }
}
	Output
	20 19 9 8 4 3 1

```
static void funB(int n)
{
   if (n > 1) {
      System.out.print(" "  +n);

      // Fun(B) is calling fun(A)
      funA(n / 2);
   }
}
```

Tracing Tree Of Recursive Function



[Indirect recursion]

Output: 20 19 9 8 4 3 1

*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.