**WHAT IS RECURSION:**

**What is Recursion?**   
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are [Towers of Hanoi (TOH)](https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/), [Inorder/Preorder/Postorder Tree Traversals](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/), [DFS of Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

**Need of Recursion**

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number

**Properties of Recursion:**

* Performing the same operations multiple times with different inputs.
* In every step, we try smaller inputs to make the problem smaller.
* Base condition is needed to stop the recursion otherwise infinite loop will occur

**A Mathematical Interpretation**

Let us consider a problem that a programmer has to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

***approach(1) – Simply adding one by one***

***f(n) = 1 + 2 + 3 +……..+ n***

but there is another mathematical approach of representing this,

***approach(2) – Recursive adding***

***f(n) = 1 n=1***

***f(n) = n + f(n-1) n>1***

There is a simple difference between the approach (1) and approach(2) and that is in **approach(2)** the function “ **f( )** ” itself is being called inside the function, so this phenomenon is named recursion, and the function containing recursion is called recursive function, at the end, this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way

**How are recursive functions stored in memory?**

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

**What is the base condition in recursion?**   
In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

int fact(int n)  
{  
 if (n < = 1) // base case  
 return 1;  
 else   
 return n\*fact(n-1);   
}

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

**How a particular problem is solved using recursion?**   
The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know the factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0

**Why Stack Overflow error occurs in recursion?**   
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

int fact(int n)  
{  
 // wrong base case (it may cause  
 // stack overflow).  
 if (n == 100)   
 return 1;  
  
 else  
 return n\*fact(n-1);  
}

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

**What is the difference between direct and indirect recursion?**   
A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly. The difference between direct and indirect recursion has been illustrated in Table 1.

**// An example of direct recursion**  
void directRecFun()  
{  
 // Some code....  
  
 directRecFun();  
  
 // Some code...  
}

// An example of indirect recursion  
void indirectRecFun1()  
{  
 // Some code...  
  
 indirectRecFun2();  
  
 // Some code...  
}  
void indirectRecFun2()  
{  
 // Some code...  
  
 indirectRecFun1();  
  
 // Some code...  
}

**What is the difference between tailed and non-tailed recursion?**   
A recursive function is tail recursive when a recursive call is the last thing executed by the function. Please refer [tail recursion article](https://www.geeksforgeeks.org/tail-recursion/) for details.

**How memory is allocated to different function calls in recursion?**   
When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached

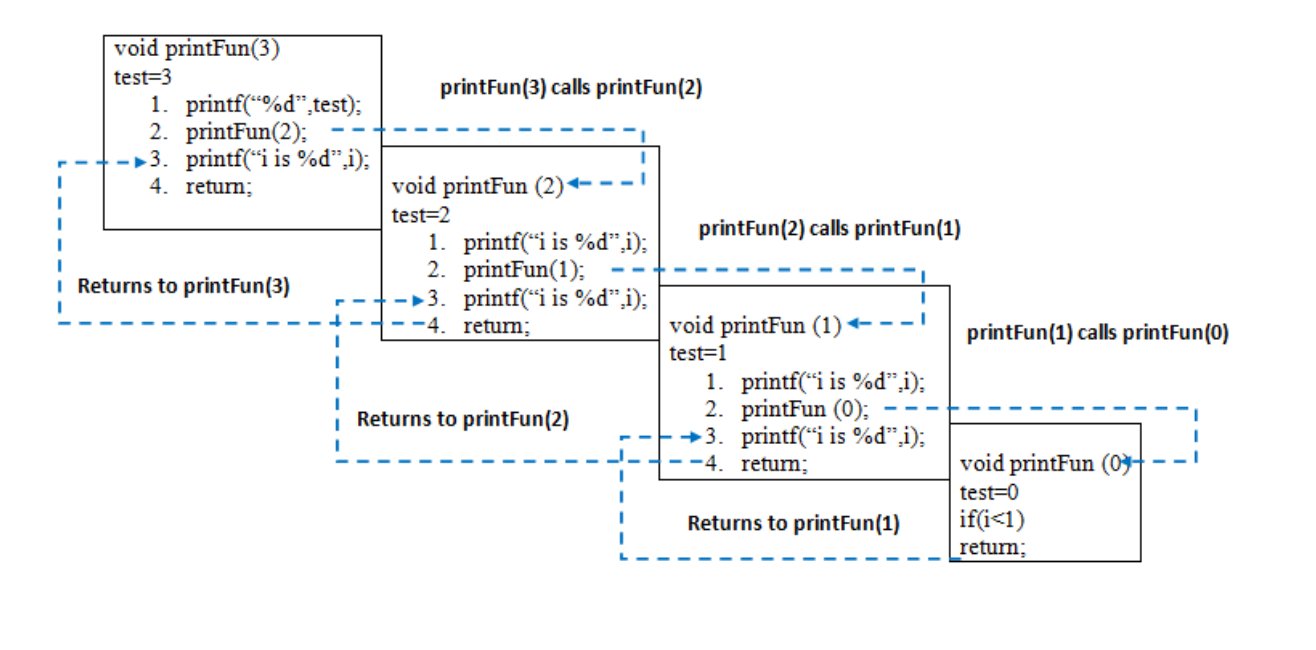
, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.  
Let us take the example of how recursion works by taking a simple function.

|  |
| --- |
| // A C++ program to demonstrate working of  // recursion  #include <bits/stdc++.h>  **using** **namespace** std;    **void** printFun(**int** test)  {  **if** (test < 1)  **return**;  **else** {  cout << test << " ";  printFun(test - 1); // statement 2  cout << test << " ";  **return**;  }  }    // Driver Code  **int** main()  {  **int** test = 3;  printFun(test);  } |

**Output :**

3 2 1 1 2 3

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed into the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. The remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



**Recursion VS Iteration**

|  |  |  |
| --- | --- | --- |
| **SR No.** | **Recursion** | **Iteration** |
| 1) | Terminates when the base case becomes true. | Terminates when the condition becomes false. |
| 2) | Used with functions. | Used with loops. |
| 3) | Every recursive call needs extra space in the stack memory. | Every iteration does not require any extra space. |
| 4) | Smaller code size. | Larger code size. |

Now, let’s discuss a few practical problems which can be solved by using recursion and understand its basic working. For basic understanding please read the following articles.   
[Basic understanding of Recursion.](https://www.geeksforgeeks.org/recursion/)Problem 1: Write a program and recurrence relation to find the Fibonacci series of n where n>2 .   
Mathematical Equation:

n if n == 0, n == 1;   
fib(n) = fib(n-1) + fib(n-2) otherwise;

*Recurrence Relation:*

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

**Recursive program:**

**Input:** n = 5   
Output:  
Fibonacci series of 5 numbers is : 0 1 1 2 3

**Implementation:**

* C++
* C
* Java
* Python3
* C#
* Javascript

|  |
| --- |
| // C++ code to implement Fibonacci series  #include <bits/stdc++.h>  **using** **namespace** std;    // Function for fibonacci    **int** fib(**int** n)  {  // Stop condition  **if** (n == 0)  **return** 0;    // Stop condition  **if** (n == 1 || n == 2)  **return** 1;    // Recursion function  **else**  **return** (fib(n - 1) + fib(n - 2));  }    // Driver Code  **int** main()  {  // Initialize variable n.  **int** n = 5;  cout<<"Fibonacci series of 5 numbers is: ";    // for loop to print the fibonacci series.  **for** (**int** i = 0; i < n; i++)  {  cout<<fib(i)<<" ";  }  **return** 0;  } |

**Output**Fibonacci series of 5 numbers is: 0 1 1 2 3

**Time Complexity:** O(2n)  
Auxiliary Space: O(n)

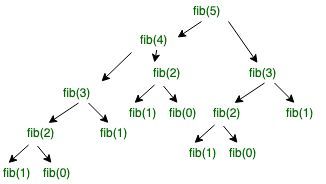
Here is the recursive tree for input 5 which shows a clear picture of how a big problem can be solved into smaller ones.   
fib(n) is a Fibonacci function. The time complexity of the given program can depend on the function call.

*fib(n) -> level CBT (UB) -> 2^n-1 nodes -> 2^n function call -> 2^n\*O(1) -> T(n) = O(2^n)*

For Best Case.

T(n) = θ(2^n\2)

**Working:**



**Problem 2:** Write a program and recurrence relation to find the Factorial of n where n>2 .   
Mathematical Equation:

1 if n == 0 or n == 1;   
f(n) = n\*f(n-1) if n> 1;

**Recurrence Relation:**

T(n) = 1 for n = 0  
T(n) = 1 + T(n-1) for n > 0

**Recursive Program:**   
Input: n = 5   
Output:   
factorial of 5 is: 120  
Implementation:

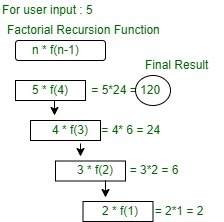
* C++
* C
* Java
* Python3
* C#
* Javascript

|  |
| --- |
| // C++ code to implement factorial  #include <bits/stdc++.h>  **using** **namespace** std;    // Factorial function  **int** f(**int** n)  {  // Stop condition  **if** (n == 0 || n == 1)  **return** 1;    // Recursive condition  **else**  **return** n \* f(n - 1);  }    // Driver code  **int** main()  {  **int** n = 5;  cout<<"factorial of "<<n<<" is: "<<f(n);  **return** 0;  } |

**Output**factorial of 5 is: 120

**Time complexity:** O(n)  
Auxiliary Space: O(n)

**Working:**



*Diagram of factorial Recursion function for user input 5.*

**What are the disadvantages of recursive programming over iterative programming?**   
Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than the iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

Moreover, due to the smaller length of code, the codes are difficult to understand and hence extra care has to be practiced while writing the code. The computer may run out of memory if the recursive calls are not properly checked.

**What are the advantages of recursive programming over iterative programming?**   
Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, [Tower of Hanoi](https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/), etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer [Inorder Tree Traversal without Recursion](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/), [Iterative Tower of Hanoi](https://www.geeksforgeeks.org/iterative-tower-of-hanoi/).

**Summary of Recursion:**

* There are two types of cases in recursion i.e. recursive case and a base case.
* The base case is used to terminate the recursive function when the case turns out to be true.
* Each recursive call makes a new copy of that method in the stack memory.
* Infinite recursion may lead to running out of stack memory.
* Examples of Recursive algorithms: Merge Sort, Quick Sort, Tower of Hanoi, Fibonacci Series, Factorial Problem, etc.

The Fibonacci sequence is, by definition, the integer sequence in which every number after the first two is the sum of the two preceding numbers. To simplify:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

It has many applications in mathematics.

Recursive functions are those functions which, basically, call themselves.

First of all, let’s think about what the code is going to look like. It’ll include:

· A recursive function F (F for Fibonacci): to compute the value of the next term.

Our function will take *n* as an input, which will refer to the *n*th term of the sequence that we want to be computed. So, F(4) should return the fourth term of the sequence.

Let’s plan it. The code should, regardless the language, look something like this:

Let’s plan it. The code should, regardless the language, look something like this:

[function F(n) if n = 0  
 return 0 if n = 1  
 return 1 else  
 return F(n-1) + F(n-2)](https://unsplash.com/?utm_source=medium&utm_medium=referral)

Note: the term 0 of the sequence will be considered to be 0, so the first term will be 1; the second, 1; the third, 2; and so on. You get it.

Let’s analyze the function for a moment. If it gets 0 as an input, it returns 0. If it gets 1, it returns 1. If it gets 2… Well, in that case it falls into the else statement, which will call the function again for terms 2–1 (1) and 2–2 (0). That will return 1 and 0, and the two results will be added, returning 1. Perfect.

Now you can see why recursive functions are a problem in some cases. Imagine you wanted the 100th term of the sequence. The function would call itself for the 99th and the 98th, which would themselves call the function again for the 98th and 97th, and 97th and 96th terms…and so on. It would be **really** slow.

But the good news is that it actually works!

So let’s start with the different languages. I won’t give too much detail (actually, no detail at all) to make your reading experience better. There isn’t too much to detail anyways.

Let’s jump into it:

### **Python**

[def F(n): if n == 0:  
 return 0 if n == 1:  
 return 1 else:  
 return F(n-1) + F(n-2)](https://unsplash.com/?utm_source=medium&utm_medium=referral)

ADVERTISEMENT

### **Swift**

[func F(\_ n: Int) -> Int { if n == 0 { return 0  
 } if n == 1 { return 1  
 } else { return F(n-1) + F(n-2)  
 }}](https://unsplash.com/?utm_source=medium&utm_medium=referral)

### **JavaScript**

[function F(n) { if(n == 0) { return 0;  
 } if(n == 1) { return 1;  
 } else { return F(n-1) + F(n-2);  
 }}](https://unsplash.com/?utm_source=medium&utm_medium=referral)

### **Java**

[public static int F(int n) { if(n == 0) { return 0;  
 } if(n == 1) { return 1;  
 } else { return F(n-1) + F(n-2);  
 }}](https://unsplash.com/?utm_source=medium&utm_medium=referral)

### **C++**

[int F(int n) { if(n == 0) { return 0;  
 } if(n == 1) { return 1;  
 } else { return F(n-1) + F(n-2);  
 }}](https://unsplash.com/?utm_source=medium&utm_medium=referral)

And that’s it. I chose these languages just based on popularity — or at least because these 5 are the most common ones