

# Recursion

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

Recursion	Iteration
The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
A conditional statement is included in the body of the function to force the function to return without recursion call being executed	The iteration statement is repeatedly executed until a certain condition is reached.
Slow in execution.	Fast in execution

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

## Explanation

return 5 \* factorial(4) = 120  
└─ return 4 \* factorial(3) = 24  
    └─ return 3 \* factorial(2) = 6  
        └─ return 2 \* factorial(1) = 2  
            └─ return 1 \* factorial(0) = 1

## Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
}
else
{
    // Statements;
    recursive call;
}
```

## Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
    int n,f;
    printf("Enter the value of n?");
    scanf("%d",&n);
    f = fibonacci(n);
    printf("%d",f);
}
int fibonacci (int n)
{
    if (n==0)
    {
        return 0;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

## 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 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.

### **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 calling function and 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.

### **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 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.

### **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, 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, Iterative Tower of Hanoi.