Recursion and stack

Some computer programming languages allow a module or **function to call itself**. This technique is known as 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**.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, **recursion may be applied to sorting, searching, and traversal problems.**

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.**

In the following example, recursion is used to calculate the factorial of a number.

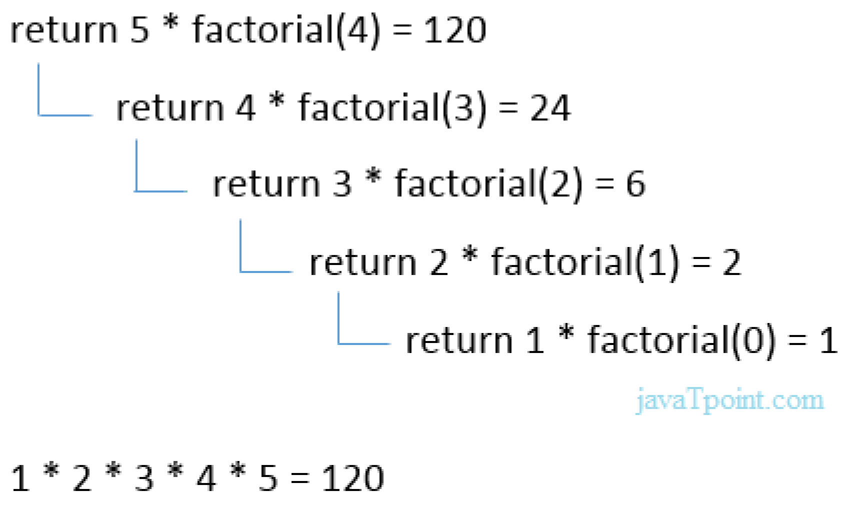
Example :

```
#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 == 1)
   {
      return 1;
   }
   else
   {
      return n*fact(n-1);
   }
}
```

Output

```
Enter the number whose factorial you want to calculate: 5
factorial = 120
```



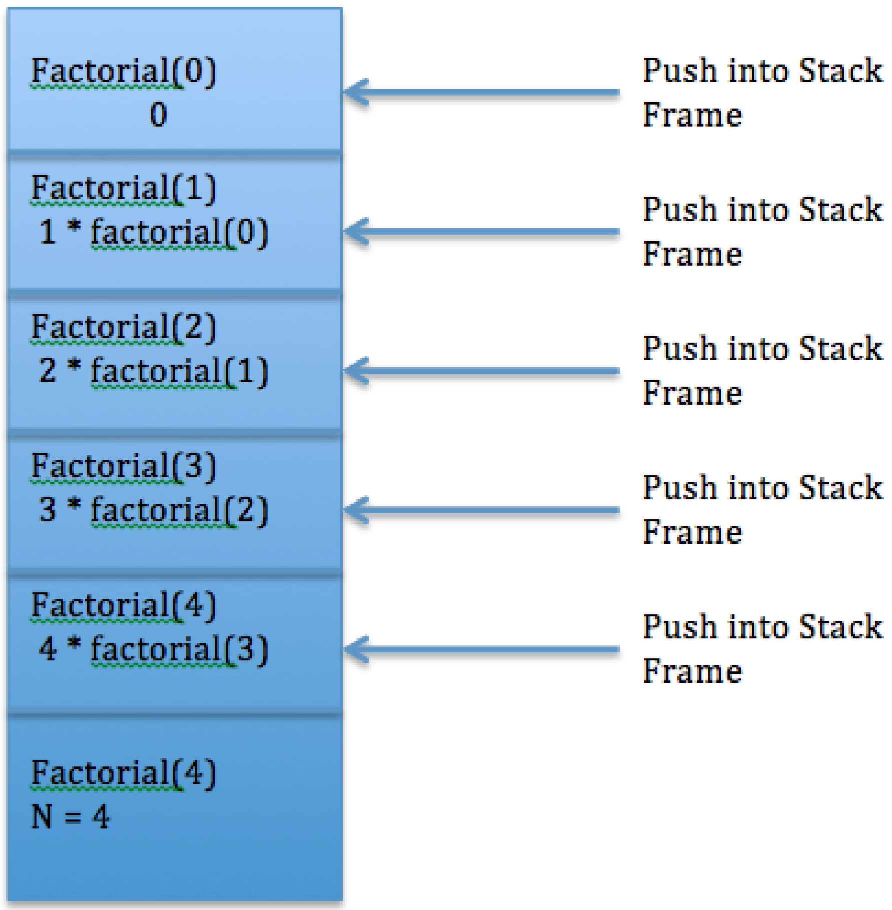
```
return 5 * factorial(4) = 120
   └── return 4 * factorial(3) = 24
          └── return 3 * factorial(2) = 6
                 └── return 2 * factorial(1) = 2
                        └── return 1 * factorial(0) = 1

1 * 2 * 3 * 4 * 5 = 120
```

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.

Let us consider the following example to understand the memory allocation of the recursive functions.

Implementation of recursion



Analysis of Recursion

- One may argue why to use recursion, as the same task can be done with iteration.
- The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

• Recursion vs Iteration

1) In recursion, function **call itself** until the base condition is reached.

On other hand iteration means **repetition of process** until the condition fails. For example – when you use loop (for,while etc.) in your programs.

2) Iterative approach involves four steps, initialization , condition, execution and updation.

In recursive function, only base condition (terminate condition) is specified.

3) Recursion keeps your code **short and simple** Whereas iterative approach makes your code longer.

4) Recursion is slower than iteration due to overhead of maintaining stack whereas iteration is faster.

5) Recursion takes more memory than iteration due to overhead of maintaining stack .

6) If recursion is not terminated (or base condition is not specified) than it creates stack overflow (where your system runs out of memory).

7) Any recursive problem can be solved iteratively . But you can't solve all problems using recursion.