Recursion

• What is Recursion?

Recursion is a programming pattern that is useful when a task can easily be split up into several similar, but simpler tasks. Or when a task can be simplified into an easy action plus a simpler variant of the same task.

Recursion can be defined as the process in which a function calls itself, either directly or indirectly. In simple terms, when a function calls itself, it is known as recursion. The function that is calling itself is called the recursive function.
A simple example –

```
void a() {
   cout << "Hello Hi Welcome!" << endl;
   a();
}
```

Recursion involves few important steps where the recursive methods play an important role. This is accomplished by breaking the whole problem into smaller problems. They then call individual copies of those smaller problems to solve them, simplifying the overall problem. The recursive call is the process of calling each copy of that smaller problem, one after the other, to solve it.(there may be multiple recursive calls).

But there must be a terminating condition in recursion, otherwise these calls may go endlessly leading to an infinite loop of recursive calls and call stack overflow.

- **When should we use Recursion?**

Sometimes we are unable to solve a problem iteratively due to its high complexity. But, if we break the problem into smaller versions of the problem, we may be able to find solutions for those smaller versions of the problem. And then, we may combine those solutions to arrive at our answer for the larger problem.

This exactly is the underlying concept behind recursion -- Recursive functions break any problem up into smaller sub-problems, the answer to which is either already known or can be found by using another algorithm. Hence, finally combining the results to build up the answer for the entire problem.

- **We can use recursion for the following reasons:**

Recursion can be useful when dealing with problems that have many possible branches and are too complex for iteration.
For example, imagine you are searching for a file on your laptop. You enter the root folder, within that you enter another folder, and so forth as you climb folders on folders until you locate your file. You can think of it as a tree with multiple branches. So, these kinds of problems are best solved using recursion. For these types of problems, an iterative approach may not be preferable.

A recursive solution is usually shorter than an iterative one.
Suppose we want to compute the factorial of any number in C++. Let's go with the iterative approach.

```
int factorial(int n) {
   int fact = 1;
   for(int i = 1; i <= n; i++) {
      fact *= i;
   }
   return fact;
}
```

If you go by the recursion approach, it is as simple as:

```
int factorial(int n) {
   return (n <= 1) ? 1 : (n * factorial(n - 1));
}
```

All recursive algorithms can [also] be implemented iteratively and vice versa. Also, recursions are best suited and easiest for Trees and graphs to do traversal.

- **Format of recursive function**

Whenever we are talking about recursive functions, we cannot deny the two most important parts of recursion.

1. A base case, in which the recursion can terminate and return the result immediately.
2. A recursive case, in which the function is supposed to call itself, to break the current problem down into smaller problems

Similarly, the input size gets smaller and smaller with each recursive call, and we get the solution to that smaller problem. Later, we combine those results to find the solution to the entire problem.

This is a pattern followed by every recursive function. Let's take the example of factorial function for this,

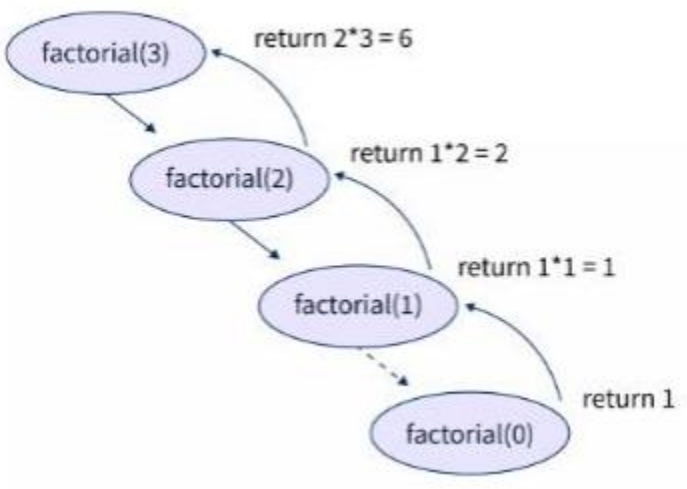
```
function factorial(n):
if (n <= 1) //Base Case
return 1;
else
return n * factorial(n-1); //Recursive Case
```

If n <= 1, then everything is trivial. It is called the base case of recursion, because it immediately returns the result: factorial(1) = 1.

Otherwise, we can represent factorial(n) as n factorial(n - 1) This is called a

recursive step we transform the task into a simpler action (multiplication by n) and a simpler call of the same task (factorial with lower n). Next steps simplify it further and further until n reaches 1.

The recursion tree for the above function if called with factorial(3) will look like:



• **Base condition in recursion**

The base case is the terminating case in recursion. This is where the recursive calls finally come to an end, or terminate preventing any endless (infinite) recursive calls. It is a problem whose answer we already know and which can be directly returned without any further recursive calls being made. If base case is not provided, the entire memory can be exhausted by the infinite recursive calls, resulting in the call stack overflow.

• **Advantages/Disadvantages of Recursion**

These are the advantages of recursion,

1. Usually code is simpler to write.
2. Extremely useful when a task can be simplified into an easy action plus a simpler variant of the same task.
3. To solve problems which are naturally recursive for example - the tower of Hanoi.
4. Recursion reduce the length of code.
5. Useful in solving data structures - tree related problems.

Below are few disadvantages of recursion:
1. Recursive functions are inefficient in terms of space and time complexity
2. They may require a lot of memory space to hold intermediate results on the system's stacks.
3. Recursion can sometimes be slower than iteration because in addition to the loop content, it has to deal with the recursive call stack frame. This will result in more code being run, which will make it slower.
4. The computer may run out of memory if the recursive calls are not properly checked(stack overflow), or the base case is not added.
5. Sometimes they are hard to analyze or it is difficult understand the code.

• **Applications of Recursion**

Recursive solutions are best suited to some problems. Below are a few examples --

Tree Traversals: InOrder, PreOrder PostOrder

Graph Traversals.

Towers of Hanoi.

Backtracking Algorithms.

Divide and Conquer Algorithms.

Dynamic Programming Problems.

Merge Sort, Quick Sort.

Binary Search.

Fibonacci Series, Factorial, etc