Tail Recursion

What is tail recursion?

A recursive function call is tail recursive when recursive call is the last thing executed by the function.

A recursive function call is said to be tail recursive if there is nothing to do after the function returns except return its value.

```c
 /* non tail recursive example of the
factorial function in C*/

  #include<stdio.h>
  void main(){
       int c;
       c=factorial(5);
       printf("%d\n",c);
  }

  factorial(n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
  }
```

```
// A tail recursive function to calculate factorial

  #include<stdio.h>
  void main(){
        int c;
        c=factorial(5);
        printf("%d\n",c);
  }

        fact(n, result) {
    if (n == 0) return result;
    return fact(n - 1, n * result);
  }

  factorial(n) {
    return fact(n, 1);
  }
```

Recursive procedures call themselves to work towards a solution to a problem. In simple implementations this balloons the stack as the nesting gets deeper and deeper, reaches the solution, then returns through all of the stack frames. This waste is a common complaint about recursive programming in general.

Why do we care?

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

```
/* tail-recursion of factorial1 can be equivalently
 defined in terms of goto: */

  #include<stdio.h>
  void main(){
        int c;
        c=factorial(5,1);
        printf("%d\n",c);
  }
    factorial(n, accumulator) {
  beginning:
    if (n == 0) return accumulator;
    else {
      accumulator *= n;
      n -= 1;
      goto beginning;
    }
  }
```

```
/* the goto version, we can derive a version
 that uses C's built-in control structures:*/

  #include<stdio.h>
  void main(){
        int c;
        c=factorial1(5,1);
        printf("%d\n",c);
  }

  factorial1(n, accumulator) {
    while (n != 0) {
      accumulator *= n;
      n -= 1;
    }
    return accumulator;
  }
```

```
/* the goto version, we can derive a version
 that uses C's built-in control structures:*/

  #include<stdio.h>
  void main(){
        int c;
        c=factorial1(5,1);
        printf("%d\n",c);
  }

  factorial1(n, accumulator) {
    while (n != 0) {
      accumulator *= n;
      n -= 1;
    }
    return accumulator;
  }
```

Advantages of tail recursion:

In traditional recursion, you perform your recursive calls first, and then you take the return value of the recursive call and calculate the result. So you have to wait till you return from every recursive call for result.

In tail recursion, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step.

In this way once you are ready to perform your next recursive step, you don't need the current stack frame any more. This way your program is optimised a lot.

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use.

Tail call

A tail call is the last call executed in a function; if a tail call leads to the parent function being invoked again, then the function is tail recursive.

The tail recursive function does not use the stack; the function calculates the factorial as it goes along! The last recursive call performs a simple calculation based on the input arguments and returns.
Thus, there is no need to push frames on to the stack because the new function already has everything it needs. As such stack overflows would not occur with tail call optimized functions.

Tail call

The biggest advantage of using tail calls is that they allow you to do extensive operations without exceeding the call stack. This makes it possible to do a lot of work in constant space without running into out of memory exceptions; this happens because the frame for the currently executing function is re-used by the newly-executed function call.

It is possible to write the same function using tail call recursion and this will allow you to do it for arbitrarily large values. The space optimization is from *O(n)* to *O(1)* since every new function call reuses existing frames rather than pushing stuff on the stack.