**Recursion**

* The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
* Two types of recursion – Linear and Divide and conquer
* We can add one or more base conditions that stop the recursion
* If the base case is not reached or not defined, then the ***StackOverflowError*** may arise.
* 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.
* ***Tail recursion*** is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically nothing is left to execute after the recursion call.
* The tail recursive functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler.
* Compilers usually execute recursive procedures by using a stack. This stack consists of all the pertinent information, including the parameter values, for each recursive call. When a procedure is called, its information is pushed onto a stack, and when the function terminates the information is popped out of the stack. Thus for the non-tail-recursive functions, the stack depth (maximum amount of stack space used at any time during compilation) is more.
* ***Non-tail or head recursion*** is defined as a recursive function in which the recursive call is the first statement that is executed by the function. It means there is no statement or operation before the recursive calls.

Questions

1. **Why is tail recursion optimization faster than normal recursion?**

* In non-tail recursive functions, after one recursive call is over, there are more statements to compute, and hence all the functions do not unfold as soon as the base case is hit.
* Every time when a recursive call is done (or any function call), a stack frame is added to the call stack. This keeps track of where you were at the time the function call was made so we can continue from where we left off. If this is done enough times, say through a recursive function to compute the 1 billionth Fibonacci number, you can get a stack overflow, which will typically terminate the process.
* Tail recursion works off the realization that some recursive calls don’t need to “continue from where they left off” once the recursive call returns. Specifically, when the recursive call is the last statement that would be executed in the current context. Tail recursion is an optimization that doesn’t bother to push a stack frame onto the call stack in these cases, which allows your recursive calls to go very deep in the call stack.
* If the last action of a method is a call to another method, instead of creating a new stack frame for the context of the new method (arguments, local variables, etc.), we can replace the current one. One of the drawbacks of normal recursive methods is that they heavily use the call stack. Tail-call optimization eliminates this problem and guarantees that the performance of a recursive algorithm is exactly as good as its iterative counterpart (yet potentially it is much more readable).