Lab 7 Report

The non-tail recursive method of computing the nth number in the fibonacci sequence makes two recursive calls: fib(n-1) and fib(n-2) for each step. These calls are not the last operation performed; the last operation is the addition that happens after they return. Each recursive call adds a new element to the call stack, which makes the stack usage exponential aka O(n^2) memory complexity and potentially leading to stack overflow. The time complexity is also exponential O(n^2) because the function computes stuff multiple times. For example, fib(4) will compute fib(3) and fib(2), and fib(3) will compute fib(2) and fib(1). Here, fib(2) gets computed twice. This is where tail recursion would be better. The tail recursive method of computing the nth number in the fibonacci sequence makes only one recursive call: fibTail ~a:b ~b:(a + b) (n - 1). The recursive call is the last operation performed in the function which makes it tail recursive. For example, fibTail(4, a=0, b=1), here each recursive call updates a and b before making the next call so there are no other operations being performed after. Since the recursive call is the last operation, the compiler optimizes the recursion into a loop, which ensures constant stack space, making it O(1) memory complexity. The time complexity is O(n) because each step is a single recursive call and unlike fib() that is redundant like explained before, fibTail() stores the two most recent fibonacci numbers in a and b at each step giving it linear time complexity.

The reason tail recursion is the preferred for implementing functions like factorials is because tail recursion prevents stack overflow by using constant stack space, does not make redundant computations, and enables compiler optimizations for better performance.