**Analysis of FibonacciTail.java**

This document explains and analyzes the functionality and time complexity of FibTailRecursion(), a tail recursive function that computes the Fibonacci sequence found in **FibonacciTail.java** and how despite it shares a similar time complexity and functionality to LinearFib(), it is sightly more efficient due to its tail recursion nature.

**FibTailRecursion()**

Let us first analyze how does FibTailRecursion()compute the Fibonacci sequence. In a similar manner to LinearFib(), it keeps track of two consecutive Fibonacci numbers, recursively calls itself while decrementing n by 1 every call, sums up both consecutive numbers and returns that result to the next recursive call. However, it differs from the its counterpart by returning a tail recursive function of itself and it passes the sum of the consecutive numbers to the next recursive call through the parameters of the tail recursive function. This can be seen in the following snippet of code: return fibTailRecursion(n-1, curr, prev + curr);. Along with that, the function must always be called with some fixed parameters where prev = 0 and curr = 1 as these are the starting numbers of the Fibonacci sequence. An example of this code snippet would be the following: double prev = 0; double curr = 1; double result = fibTailRecursion(i, prev, curr);. The function’s base cases also differs as it requires two base cases: if (n==0) { return prev; } and if (n==1) { return curr; }. Because of it’s recursive call, it also iterates once for every n inside the Fibonacci sequence, therefore FibTailRecursion() can iterate a maximum amount of n times. A pseudocode example of a tail recursive Fibonacci function can be found below:

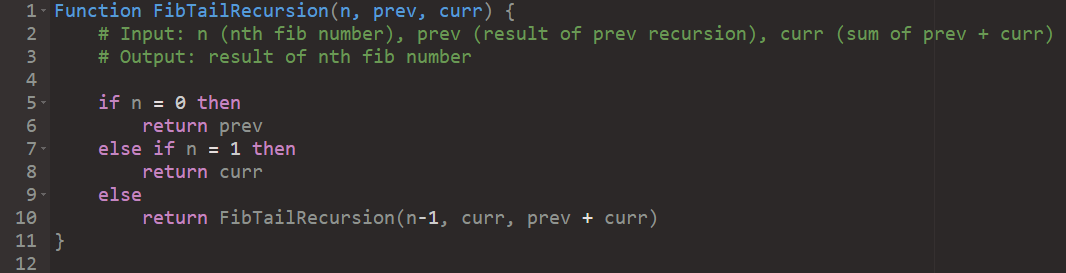


Figure : Pseudocode of FibTailRecursion()

We can also calculate its time complexity in a similar manner to LinearFib(). Assuming that the time taken for checking the if statements from the base cases and the addition operator inside the parameters takes a constant time, c, we can assume that FibTailRecursion() will iterate a maximum of times due to both if statements which is linear growth. We can verify this:

Let us analyze this problem with T(n) as a function of time.

By definition, FibTailRecursion()recursively calls itself using just FibTailRecursion(n-1). If FibTailRecursion()takes some time T(n) then we assume that FibTailRecursion(n-1) takes some time T(n-1). So, we have:

…

We notice that for every where k is any positive integer, we can replace it back in the previous . Substituting these values, we get:

)

…

*where T(0) is O(1)*

If we once again consider the worst-case scenario, the constants do not affect the final time complexity of , so we can omit them, thus leaving us with a time complexity of n. Therefore, using the big-Oh Notation, FibTailRecursion()is **O(n).**

If we plot the results into a graph, we can confirm the time complexity of our function as it follows a linear pattern of O(n): when n = 5 -> time = 400 ns, when n = 15, time -> 600 ns, when n = 25, time -> 1100 ns. During computation, however, the function follows this trend up until n = 65. Above n = 70, there are a lot more noticeable edge cases due to external factors out of our control. Below you will find a graph of such results:

Since the function is tail recursive, it means that every recursive call is immediately popped from the stack, thus saving some resources. The edge cases where n > 70 may be influenced by caching which explains the extremely low runtimes despite theoretically taking longer due to a larger n. As such, we can consider these results as correct. Overall, however, the graph respects a linear growth thus affirming our conclusion that FibTailRecursion()is **O(n).**

**FibTailRecursion() vs LinearFib():**

Using the results from **out.txt,** while LinearFib() provided some quick runtimes, we can see that it is relatively slower to FibTailRecursion()overall with FibTailRecursion() being at least 2x faster for every nth Fibonacci number making it more effective, and this is despite both shar the exact same time complexity of **O(n).** This can be explained by the advantages of having a tail recursive function as it saves time and resources by not assigning any values or operators to any variables in memory. Furthermore, as previously mentioned, the tail recursive calls get immediately popped from the stack once executed thus overall being more efficient at resource management.

We can compare the performances of both functions through the following graph:

From the graphs, we can confirm that FibTailRecursion()is overall more performant than LinearFib()due to its more effective use of resources. As such, we can conclude that the most effective recursion function out of the three is FibTailRecursion()with a time complexity of **O(n).**