**Analysis of BinaryFib.java vs LinearFib.java**

This document explains and analyzes the time complexity between the BinaryFib() and LinearFib()functions which are both found in their respective **BinaryFib.java** and **LinearFib.java** files and how although both functions produce the same results on any same nth Fibonacci number, the duration it took to compute those results are vastly different.

**BinaryFib()**

Let us first analyze how BinaryFib()works. For BinaryFib() to compute, it takes in the nth Fibonacci number as a parameter and recursively calls two more instances of the BinaryFib() function: one decrementing n by 1 and the other decrementing n by 2 which is found in the following code snippet return binaryFib(n - 1) + binaryFib(n - 2);. Once the function’s base case has been reached (n <= 1), it will return and sum all the previous recursion calls which will finally compute the result of the nth Fibonacci number. Although the function does produce correct results, this is a very ineffective way of computing the Fibonacci sequence as producing two more instances of BinaryFib() grows the computation time of the program exponentially by 2n as each recursive call will evoke two more instances of BinaryFib()and, there will be multiple of instances of BinaryFib()repeating itself.

Diagram

Description automatically generatedSuppose we want to calculate BinaryFib(5)where the nth Fibonacci number = 5. To compute, BinaryFib(5), the function will call BinaryFib(4)and BinaryFib(3). From there, BinaryFib(4)will call BinaryFib(3)and BinaryFib(2) while BinaryFib(3) will call BinaryFib(2) and BinaryFib(1). This will continue until BinaryFib(1) or BinaryFib(0)is reached, and it will return the sum of the previous recursions. A tree diagram visualizing this process is shown below:

Figure : Tree Diagram visualizing BinaryFib()

Analyzing the diagram, we make the following observation: the number of BinaryFib()calls is multiplied by two every time BinaryFib(n-1). Tracking n and the number of calls manually, we notice the following pattern:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | 0 | 1 | 2 | 3 | … | n |
| **# iterations (calls)** | 1 | 2 | 4 | 8 | … | 2n |
|  | 20 | 21 | 22 | 23 | … | 2n |

Assuming that for every n where each iteration and operation takes a constant amount of time, c, this means that as n grows larger and larger, we can hypothesize that the time complexity of the program runs times which is an exponential growth. We can verify this:

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

By definition, BinaryFib(n) recursively calls itself by using: BinaryFib(n-2) + BinaryFib(n-1). BinaryFib(n)takes some function T(n) time, so we can assume that BinaryFib(n-2)takes T(n-2) BinaryFib(n-1)takes T(n-1). So, we have:

*where T(1) and T(0) are base cases.*

We notice that T(n-2) is always one step behind T(n-1), thus we can assume that as T(n-2) is O(T(n-1)). Substituting these values, we get:

…

…

From those calculations, we conclude that in 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 2n. Therefore, using the big-Oh Notation, BinaryFib()is **O(2n).**

Based on our results of the **out.txt** file, the duration of time to compute the Fibonacci numbers for BinaryFib()accurately represents the function of **O(2n).** Looking at the **out.txt** file: when n = 5 -> time = 900 ns, when n = 10 -> time = 7200 ns, when n = 15 -> time = 123200 ns, thus we can easily see an exponential growth as n grows larger. Plotting our results into a graph helps us visualize this clearly:

From this graph, we can observe that as n grows into infinity, it becomes exponentially longer to compute the Fibonacci sequence. In fact, running the code for any n greater than 40 may take even longer than a full day, so those results were omitted from the **out.txt** file. As such, we can conclude that BinaryFib()is an ineffective method of computing the Fibonacci sequence as its time complexity is **O(n2).**

**LinearFib():**

While BinaryFib()proves to be tedious and ineffective, LinearFib()is a lot more efficient when computing the Fibonacci sequence, so let us analyze the functionality behind LinearFib(). Similarly, to BinaryFib(), it takes nth Fibonacci number as a parameter but unlike its counterpart, it computes the Fibonacci sequence by storing an array of two consecutive Fibonacci numbers and at each recursion call, it evokes LinearFib(n-1), sums up the two consecutive Fibonacci numbers and sets it at the first index of the array so that when LinearFib()reaches its base case (n <= 1), the function returns this additional information to the next level of recursion. This is found in the following code snippet: double[] answer = {temp[0] + temp[1], temp[0]}; return answer;. Doing so, it allows us to correctly calculate the Fibonacci sequence while only looping through the size n of the input only once per iteration. This means LinearFib() will iterate a maximum of n times making it a much more plausible choice than BinaryFib()’s 2n times.

Since we assume that every operation including array and if operations take a constant time, c, in each iteration and we know that LinearFib()will only loop through each iteration once so it will loop a maximum of n times, we can deduce that the program will run a max of times which is a linear growth. Once again, we can verify this:

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

By definition, LinearFib()recursively calls itself using just LinearFib(n-1). If LinearFib()takes some time T(n) then we can assume that LinearFib(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, LinearFib()is **O(n).**

If we verify our calculations with the results in **out.txt**, we can see that apart from a couple of edge cases, the trend of the time follows a linear pattern of O(n) as n gradually increases: when n = 5 -> time = 1000 ns, when n = 15, time -> 1600 ns, when n = 25, time -> 2500 ns, etc. Let us plot these results into a graph:

Since the computation time is extremely fast, we can track the runtime till completion which is n= 100. As such, we can verify from this graph that our results approximately follow a linear trend as n grows into infinity, thus asserting our conclusion that LinearFib()is **O(n).**

**BinaryFib() vs LinearFib()**

Now that we’ve determined the time complexities of both functions, let us compare the two to figure out which one is more effective. We know that the lower the time complexity is, the better the performance. Since LinearFib()is **O(n),** BinaryFib()is **O(2n)**, and **O(n)** > **O(n2),** we conclude that LinearFib()is the more effective program as n increases. What if we want to know how much more effective is LinearFib()compared to BinaryFib()? Plotting a graph that compares both results:

We can see that the difference between BinaryFib()and LinearFib()are incomparable. Setting it side by side, the duration of both functions is very similar while at the start where n < 30, however, we notice that at n >= 30, the two functions greatly distinguish in their runtimes as BinaryFib()take exponentially longer to run as it will start take a couple of seconds, minutes and even hours to run. In fact, this is the bottleneck that LinearFib()solves as since its time complexity is O(n), the time it takes to compute the Fibonacci sequence for n from 0 to 100 is relatively fast compared to its counterpart. Thus, for smaller nth Fibonacci numbers, whether using BinaryFib()or LinearFib(), does not make much of a difference, however for larger nth Fibonacci numbers, it is definitely recommended and ideal to compute using LinearFib().

In conclusion, we analyzed the functionality and time complexity of both BinaryFib()and LinearFib() functions. We determined that BinaryFib() is **O(n2)** and LinearFib() is **O(n)** which makes it ideal for larger Fibonacci sequences as BinaryFib() can take multiple hours to compute, thus making LinearFib()the more efficient algorithm between the two.