Fibonacci Sequence in Log(n) time complexity

The first time I heard of the Fibonacci in Log(n) time complexity was when I was trying to solve one Kata (coding challenge) from codewars. The Millionth Fibonacci Kata can be accessed here.

Even thought my code was working – apparently – perfectly fine, the code was getting timed-out. When I went to the comments sections, one guy said that this Kata was unable to be solved in O(n) time complexity.

And that's when my research began.

First things first, I had to figure it out from where it came from, in order to be able to understand.
And I got to be honest, I don't understand the whole idea yet, but some things I cleared out and, with those little informations, I was able to code a solution that took me 7 seconds to solve a solution with n = 1000000. With the same entry, the code in O(n) took me 21.5 seconds to be solved.

No more talks let's go for some math here.

If you can calculate a function within a symmetrical callback recursive function, you can divide the problem in n times, so this will lead to a Logarithmic Solution.
If it does not make sense to you, let's take an easy example:

Imagine that you must implement a function that calculates x^n.

This function is inherently recursive, because:

For n even you know that:

$$x^n = x^{n/2}.x^{n/2}$$

Similarly for odd n :

$$x^n = x^{n/2}.x^{n/2}.x$$

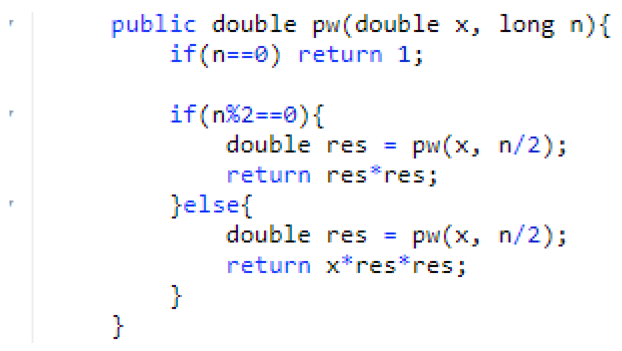
And our base-cases are:

$x^0 = 1$ and,

$x^1 = x$.

Coding this would be something like that:

Code in C#:



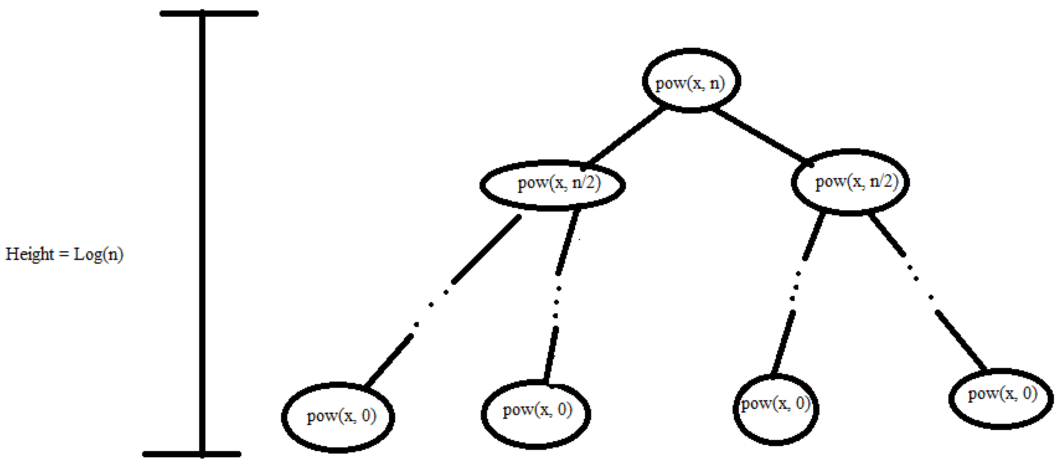
```
public double pw(double x, long n){
    if(n==0) return 1;

    if(n%2==0){
        double res = pw(x, n/2);
        return res*res;
    }else{
        double res = pw(x, n/2);
        return x*res*res;
    }
}
```

In this code that I implemented, I did not used base case where $n = 1$ simply because you don't need it. (Try to figure out why!).

As you can see, in each recursive call, the problem is divided by two.

This will lead to a Logarithmic **time** complexity.
Space complexity will be Logarithmic too because the max size of the heap will be the height of the Recursive tree.

You can visualize here:



Now back to our Fibonacci problem.

Somehow someone placed the first three Fibonacci numbers into a matrix.

That's the point where I said before I could not understand. But from now on, it's kinda easy to understand the algorithm.

Let's call this matrix as this base matrix as BS:

$$BS = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Which means

$$BS = \begin{pmatrix} f(2) & f(1) \\ f(1) & f(0) \end{pmatrix}$$

Since our Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8… etc, its everything ok.

Now let’s consider a generic matrix for Fibonacci FN as:

$$FN = \begin{pmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{pmatrix}$$

Whenever we multiply those two matrices (Let’s call it X matrix), we have this result:

$$X = \begin{pmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$X = \begin{pmatrix} f(n) + f(n-1) & f(n) \\ f(n-1) + f(n-2) & f(n-1) \end{pmatrix}$$

But we know that:

$$f(n+1) = f(n) + f(n-1)$$

$$f(n) = f(n-1) + f(n-2)$$

Now if we replace those values:

$$X = \begin{pmatrix} f(n+1) & f(n) \\ f(n) & f(n-1) \end{pmatrix}$$

Can you see a pattern here?

The X matrix is the following matrix of FN. Each cell in X is the next Fibonacci number in FN.

So more generally we can find f(n) in the position 0, 0 in the matrix FN:

$$f(n) = FN[0][0] = \begin{pmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{pmatrix}$$

Using the deduction, we made from the X matrix, we can also show that

$$FN = \begin{pmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{pmatrix} = \begin{pmatrix} f(n-1) & f(n-2) \\ f(n-2) & f(n-3) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} f(n-2) & f(n-2) \\ f(n-3) & f(n-4) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Etc…

And more generically, that matrix can be re-written as:

$$f(n) = FN[0][0] = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$$

From now on the problem is quite easy!

$$FN = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = BS^{n-1}$$

And using the same logic from exponential algorithm, we can divide every step in a half:

For (n – 1) even:

$$FN = BS^{n-1} = (BS)^{\frac{n-1}{2}} (BS)^{\frac{n-1}{2}}$$

And for (n – 1) odd:

$$FN = BS^{n-1} = (BS)^{\frac{n-1}{2}} (BS)^{\frac{n-1}{2}} (BS)$$

The code you can find in here.

PS.: My intention with this document and this repository in general is to study and archive **HOW I GOT INTO THE SOLUTION**. It's not my intention to prove – mathematically speaking – anything. All I did here is to store all the way I went through to get to the solution. Not only for this problem, but for every problem here in this repository. Hope this can help you somehow.

God bless you,
Gilcemir Filho.