

## Problem 6: Fibonacci Sequence.

Array implementation:

```
int array[n+1];  
array[0] = 0;  
array[1] = 1;  
for (int i = 2; i <= n; i++) {  
    array[i] = array[i-1] + array[i-2];  
}  
return array[n];
```

$O$  - represents operations, which equate to clock cycles.

$O_B$  - operations before our loop.

$O_i$  = operations during our loop.

$O_A$  = operations after our loop.

Mathematical Analysis:

$$O_B + \sum_{i=2}^n O_i + O_A$$

$$= O_B + (n-1)O_i + O_A$$

$$= nO_i - O_i + O_B + O_A.$$

$f(n)$  = first order polynomial.

$$f(n) = O_i n + (-O_i + O_B + O_A).$$

$$= C'n + C,$$

where  $C' = O_i$

$$C = -O_i + O_B + O_A$$

$\therefore$  Big O of this function =  $O(n)$ .

↑  
Linear.



Recursive implementation:

if ( $n == 0$ ) return 0;

if ( $n == 1$ ) return 1;

return  $\text{fib}(n-1) + \text{fib}(n-2)$ ;

O - represent operations, which equate to clock cycles.

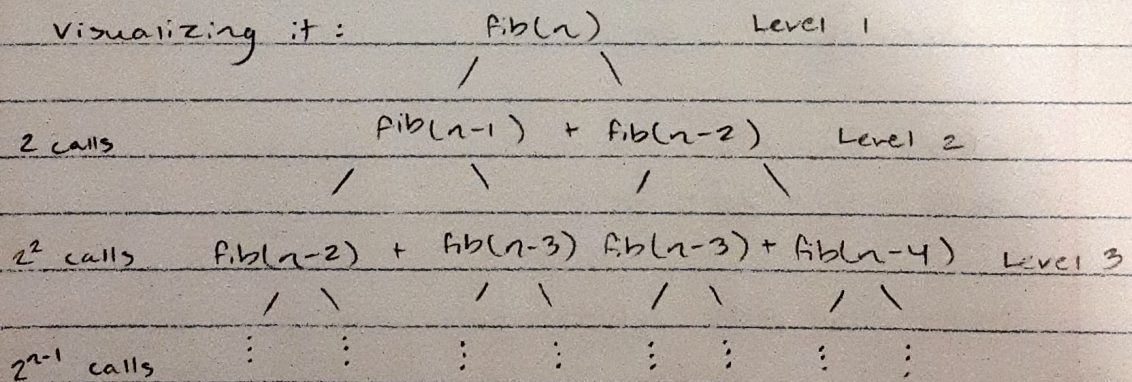
OB - Base condition operations.

(Only execute as base conditions.)

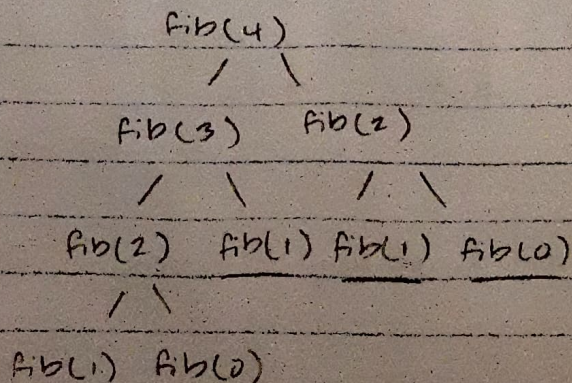
OR - Recursion operations.

Each recursion operation ORN can further be broken down into 2 more recursion operations:  $\text{OR}_{n-1}$  and  $\text{OR}_{n-2}$ .

$$\text{OR}_n = \text{OR}_{n-1} + \text{OR}_{n-2}.$$



⇒ For example: if  $n = 4$ , then:





Note : The recursive calls are not the same.

The 2nd recursive call ( $A(n-2)$ ) decrements by 2 each time. This results in the base conditions being reached faster. This means that the right side of our recursive tree will always be shorter than the left side.

⇒ With each level, the number of function calls increases exponentially. Following this logic, the last level should have  $2^{n-1}$  calls at most. This means that we have an order of  $2^{n-1}$ . However, we disregard any constants when trying to find Big O.

Then :  $A(n)$  = exponential function.

$$\begin{aligned} A(n) &= O_B + O_R = O_B + O(2^{n-1}) \\ &= O_B + O(2^n). \end{aligned}$$

∴ Big O of this function =  $O(2^n)$ .