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:

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

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

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

f(n) = first order polynomial.

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

$$= c' n + c,$$

where $c' = O_i$

$c = -O_i + O_B + O_A$

∴ Big O of this function = $O(n)$.

Linear.

Recursive implementation:

```
if (n == 0) return 0;
if (n == 1) return 1;
return fib(n-1) + fib(n-2);
```

O - represent operations, which equate to clock cycles.

O_B - Base condition operations. (Only execute as base conditions.)

O_R - Recursion operations.

Each recursion operation O_{RN} can further be broken down into 2 more recursion operations: O_{RN-1} and O_{RN-2}.

$$O_{RN} = O_{RN-1} + O_{RN-2}.$$

Visualizing it:

	fib(n)	Level 1
2 calls	fib(n-1) + fib(n-2)	Level 2
2^2 calls	fib(n-2) + fib(n-3) fib(n-3) + fib(n-4)	Level 3
2^{n-1} calls	⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮	

⇒ For example: if n = 4, then:

fib(4)

fib(3) fib(2)

fib(2) fib(1) fib(1) fib(0)

fib(1) fib(0)

Note : The recursive calls are not the same. The 2nd recursive call (fib(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 : f(n) = exponential function.

$$f(n) = O_B + O_R = O_B + O(2^{n-1})$$
$$= O_B + O(2^n).$$

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