* Recursive code complexity analysis :-

→ Analysis will be based on the no. of instructions executed w.r.t the input.

* Problem 01 :-

```
test.js > ...
1   function foo(n) {
2       if (n == 1) return 1;
3       return n * foo(n - 1);
4   }
```

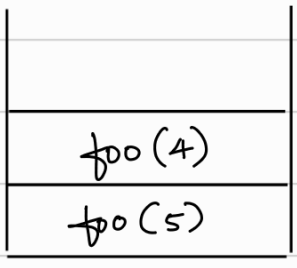
Assume foo(5), so line 2 will be constant.

Line 3 has 2 operations :-

(i) Product → c (ii) Function call → c

→ But this does not mean that the entire operation is constant.

→ Whenever we call foo(5), the function call goes to the call stack.



→ Whatever time foo(4) takes will be extra apart from the function call.

∴ Total instructions = no. of instructions in one function call × Total no. of function calls

→ This might not work for certain relations like divide & conquer relations.

→ We can use this formula when the instructions are same in every function call.

foo(5) + foo(4) + foo(3) + foo(2) + foo(1)

↓ ↓ ↓ ↓ ↓

c c c c c

∴ Total instructions = c × n

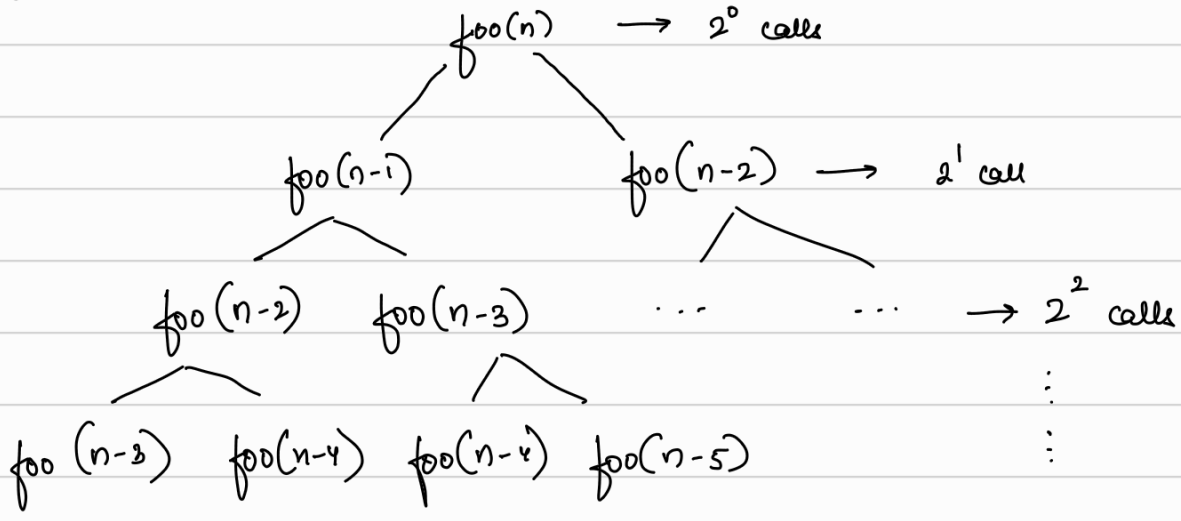
≈ n

∴ TC = O(n)

* Problem 2 :-

```js
function foo(n) {
    if (n == 0 || n == 1) return n;
    return foo(n - 1) + foo(n - 2);
}
```

→ In one function call, we have constant number of operations.



$\therefore$ Total calls $\approx 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{n-1}$

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

$$= 2^n - 1$$

$\approx 2^n$ function calls.

For every function call we are doing c operations.

$\therefore$ Total instructions $= 2^n \times c$

$$\boxed{TC = 2^n}$$

* Problem 3 :-

```js
function foo(n) {
    if (n == 0) return;

    for (let i = 1; i <= n; i++) {
        // Some operations
    }

    foo(n - 1);
}
```

Here the function does not have constant number of operations because of the for loop. which is dependent on n. So we cannot use the previous formula.

foo(5) + foo(4) + foo(3) + foo(2) + foo(1)

↓ ↓ ↓ ↓ ↓

C+5+C C+4+C C+3+C C+2+C C+1+C

$\therefore 5+2C + 4+2C + 3+2C + 2+2C + 1+2C$

Replacing with n

$\therefore n+2C + (n-1)+2C + (n-2)+2C \dots + 1+2C$

Ignoring 2C as its constant

$\therefore n + (n-1) + (n-2) + \dots + 1$ → Sum of natural numbers

$= \frac{n(n+1)}{2}$

$\therefore TC = O(n^2)$

* Problem 4 :-

```
test.js > ...
1   function foo(arr, n) {
2       // assume arr.length = k
3
4       if (n == 0) return;
5
6       for (let i = 1; i <= arr.length; i++) {
7           // Some operations
8       }
9
10      foo(arr, n - 1);
11  }
```

Here, the array length = k is constant everytime. In every function call, we execute a loop of O(k) & rest are constant operations. We have n function calls.

$\therefore TC = O(n \times k)$

* Problem 5 :-

```
test.js > ...
1   function foo(arr, n) {
2       if (n <= 1) return 1;
3       return foo(n - 1) + foo(n - 1);
4   }
```

$\therefore TC = O(2^n)$