Recursive Code Complexity Analysis

↳ We have to do the time complexity analysis, using the no. of instructions executed w.r.t the input factorial

```
function f0(n) {
    if(n == 1) return 1;
    return n*f0(n-1);
}
```

→ c

→ we are callig func^n

product → const

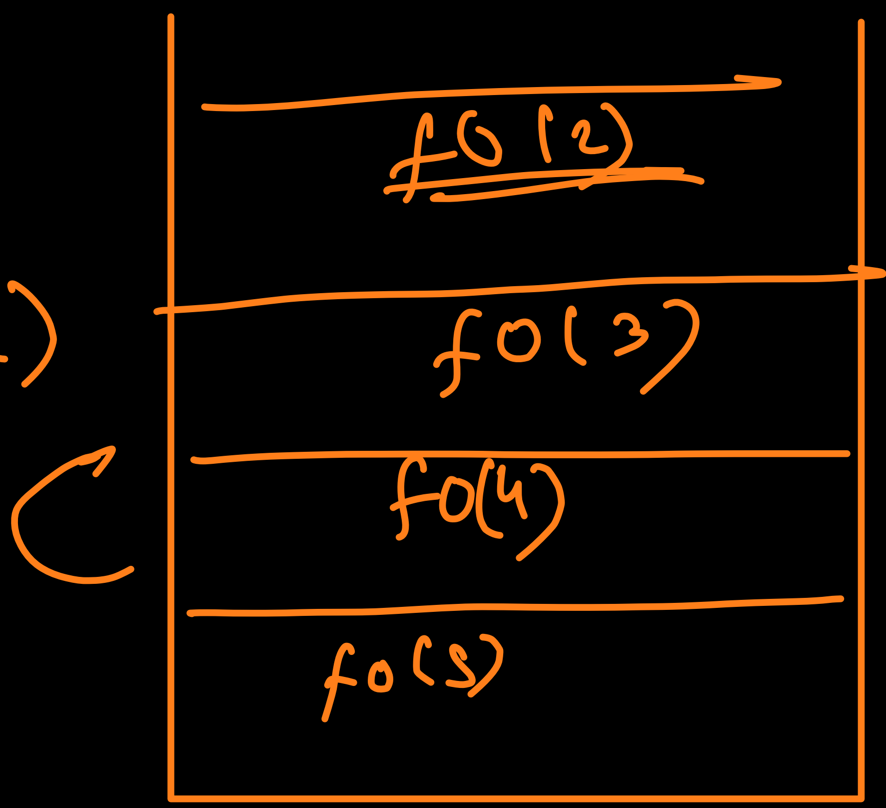
callig a func^n → const

How many total instructions were executed to calculate $n!$

$$f0(5) + f0(4) + f0(3) + f0(2)$$

$$\downarrow \quad \downarrow \quad \downarrow$$

$$C \quad C \quad C$$



```
function f0(n) {
    if(n == 1) return 1;
    return n*f0(n-1);
}
```

→

divide n conquer

Total instructions = no. of instructions in one func^n call × Total no. of func^n calls

$= c \times n$

$= nc$

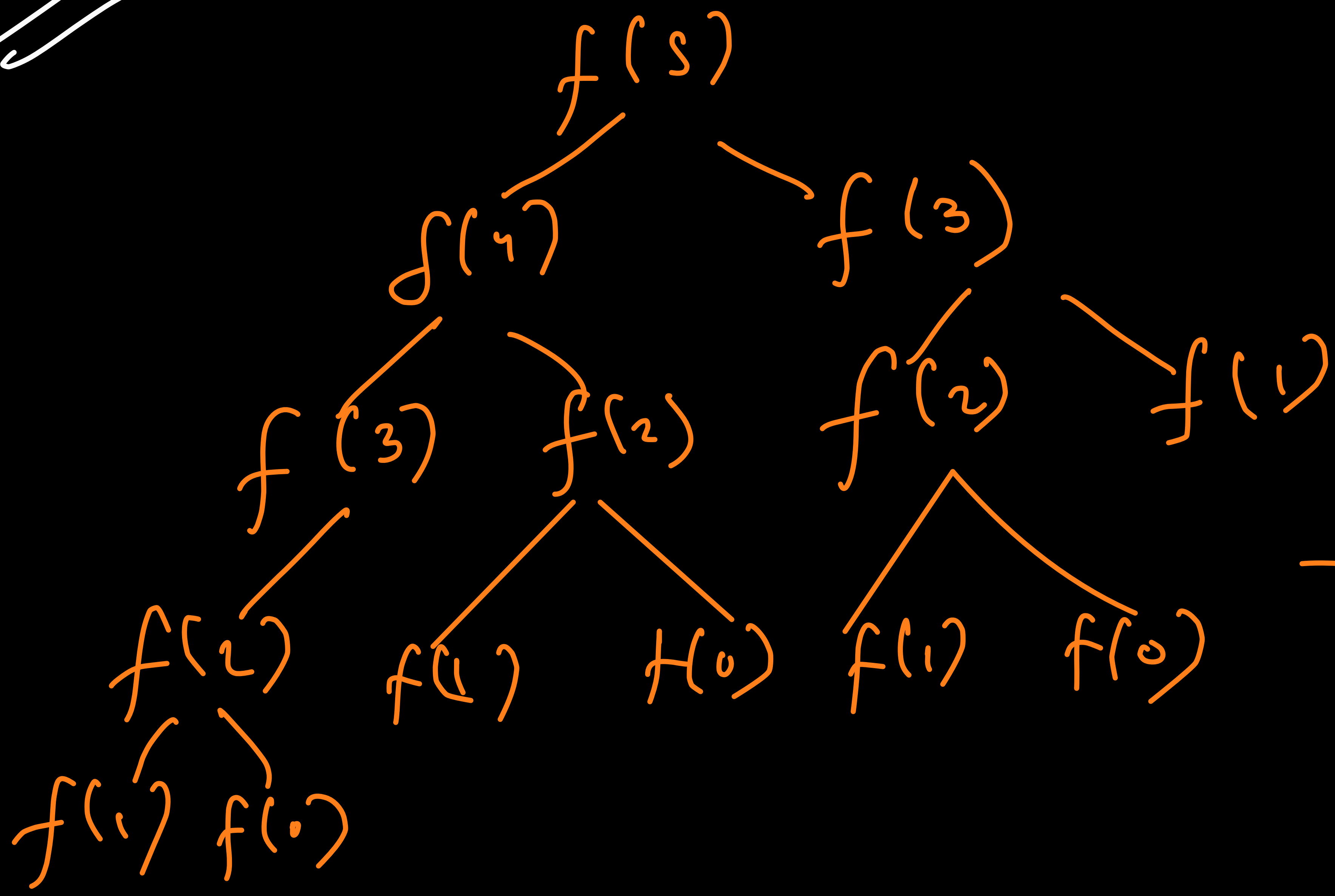
$\to O(n)$

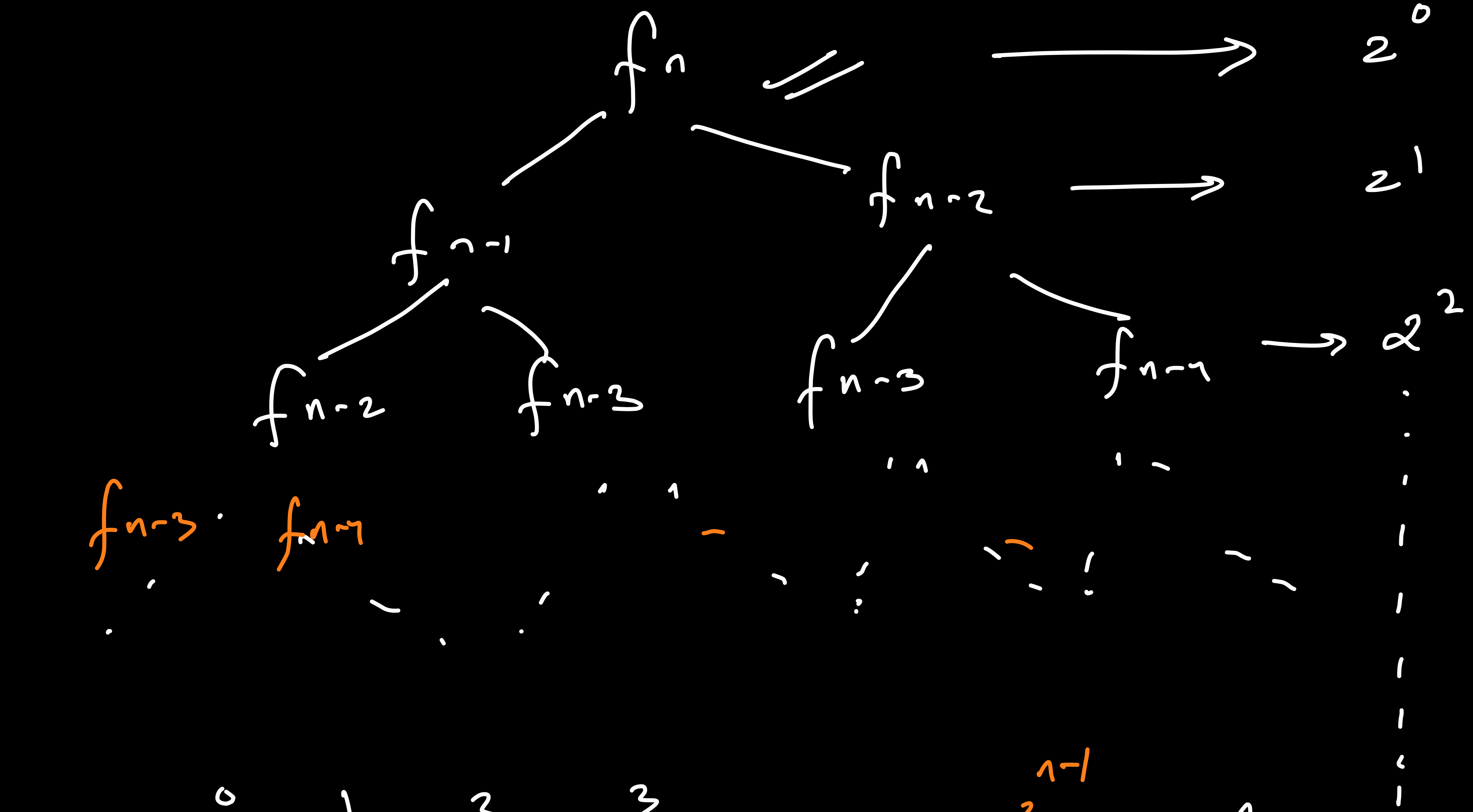
fib

```
function f1(n) {
    if(n == 1 || n == 0) return n;
    return f(n-1) + f(n-2);
}
```

→ In one func^n call we have const. ops.

→ $2^n \times c \rightarrow \underline{\underline{O(2^n)}}$





$$\approx \underbrace{2^0 + 2^1 + 2^2 + 2^3 \cdots\cdots 2^{n-1}}\quad \cancel{gP}$$

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

Complexity of iterative fib

$2 \rightarrow n \rightarrow n-2$

$\approx n$

for (i = 2 ; i <= n ; i++) $\rightarrow O(n)$

c = a + b

a = b

b = c

}

c

Note

$\hookrightarrow O(\log n)$

n^{th} fib

matrix expo

```
function f2(n) {
    if(n == 0) return; → c
    for(let i = 1; i <= n; i++) {
        // some op
    }
    f2(n-1);
}
```

$\rightarrow O(n^2)$

$f2(5) \rightarrow f2(4) \rightarrow f2(3)$ . . . . . . $f(1)$

$c + 5 + c$ $c + 4 + c$ $c + 3 + c$ $c + 1 + c$

$5 + 2c + 4 + 2c + 3 + 2c$ - - - - -

$n + \underbrace{2c} + (n-1) \underbrace{+ 2c} + (n-2) \underbrace{+ 2c}$. - - - - - - - - $1 + \underbrace{2c}$

$n + (n-1) + (n-2)$ . . . . $2 + 1 \rightarrow \frac{n(n+1)}{2} \rightarrow \frac{n^2}{\cancel{2}} + \underbrace{\frac{n}{2}}_{\times}$

$O(n^2)$

```
function f3(arr, n) {
    // assume arr.length → k
    if(n == 0)return;
    for(let i = 1; i ≤ arr.length; i++) {
        // some op
    }
    f3(arr, n-1);
}
```

→ $O(k)$

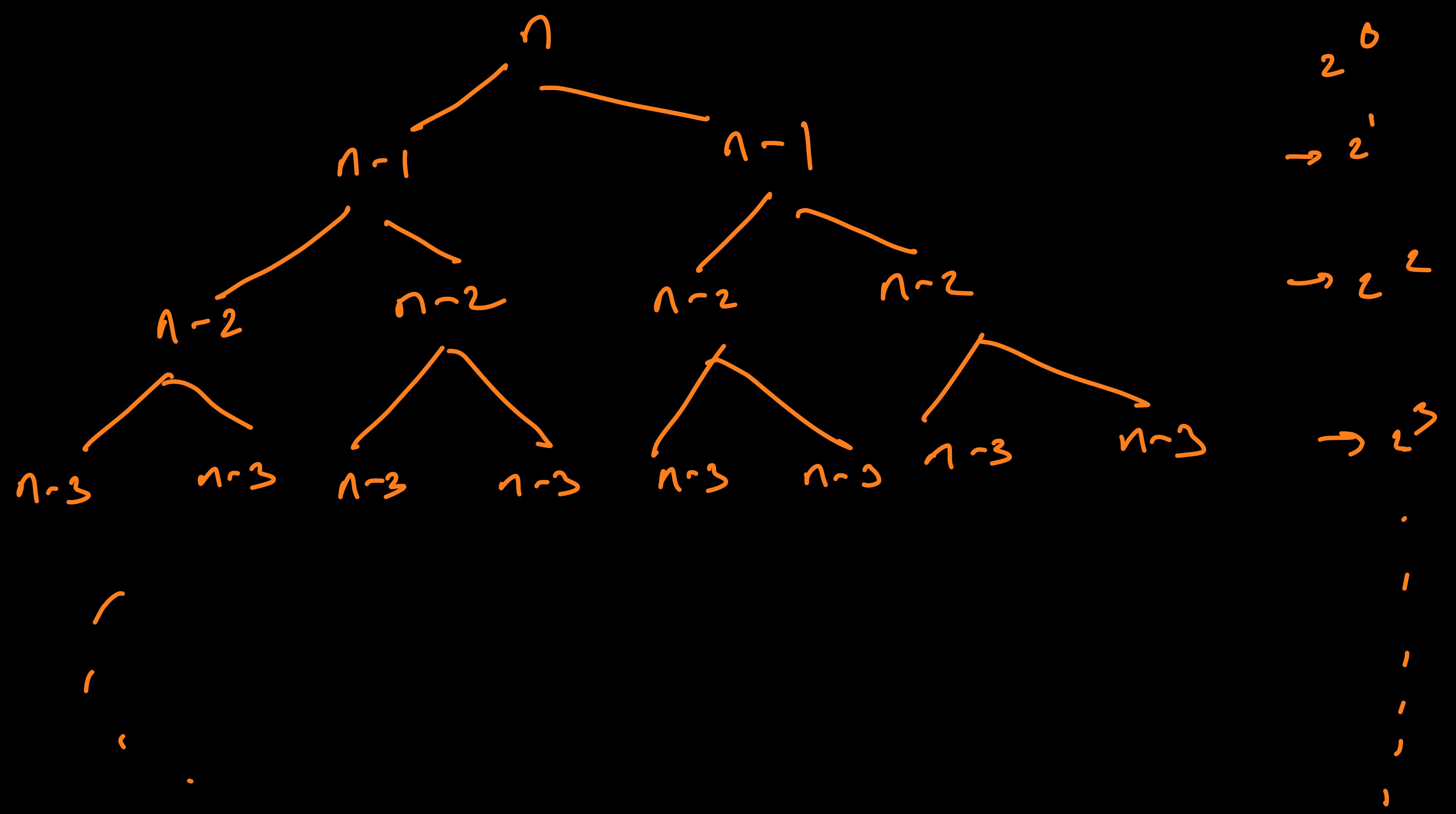
Every func^n has same no. of ops → $O(k)$

Total func^n calls → n

Total ops → $O(nk)$ → linear

```
function f4(n) {
    if(n ≤ 1) return 1;
    return f4(n-1) + f4(n-1);
}
```

$O(2^n)$



```
function fs(n) {
    if( n<=1) return 1;
    return 2*fs(n-1);
```

3

$$O(n)$$

fs(10)
↓
fs(9)
↓
8
↓
7
⋮
3
↓
$2*2$ 2
↓
1 1 2×1
↓
0

Space Complexity

↓

asymptotic analysis

Space taken w.r.t change in input

eg do

we donot consider Space of input array / data structure → output

function f(n) {

a = 10;

b = 20;

c = 30;

⋮

Array(n); → n

}

$$\underline{C}$$

→ $O(n)$

→ You're given 2 sorted arrays & we have to merge them to create a new sorted array.

leetcode

a, b

c → n+m

b → m

C → [ ]

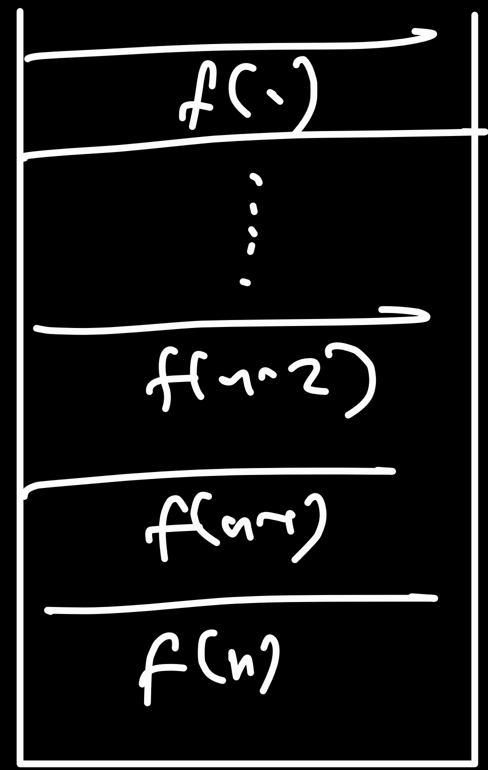
O(n+m)

what about space complexity of recursive codes ??

factorial

```
function f0(n) {
    if(n == 1) return 1;
    return n*f0(n-1);
}
```

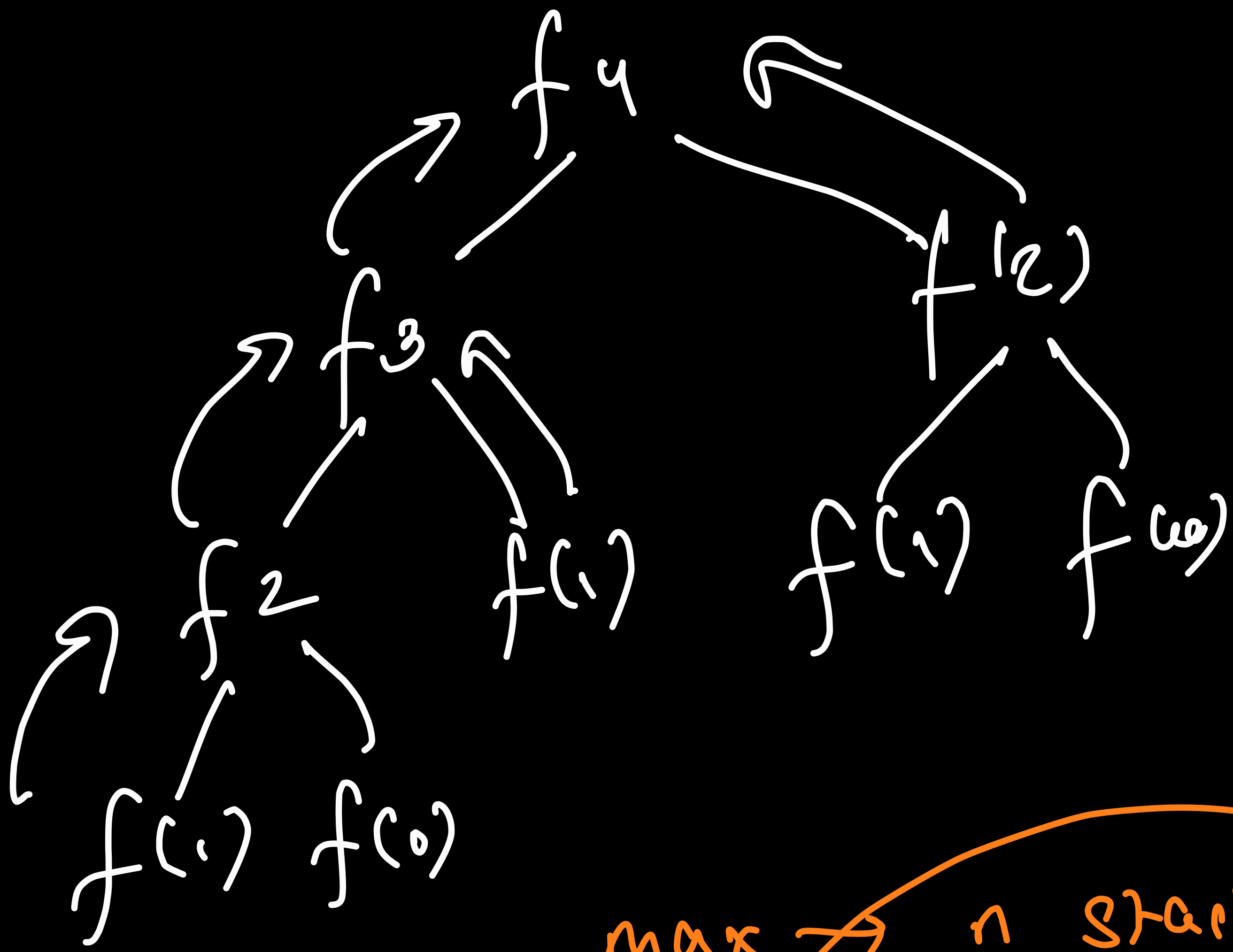
there is an extra space always involved with recursive codes

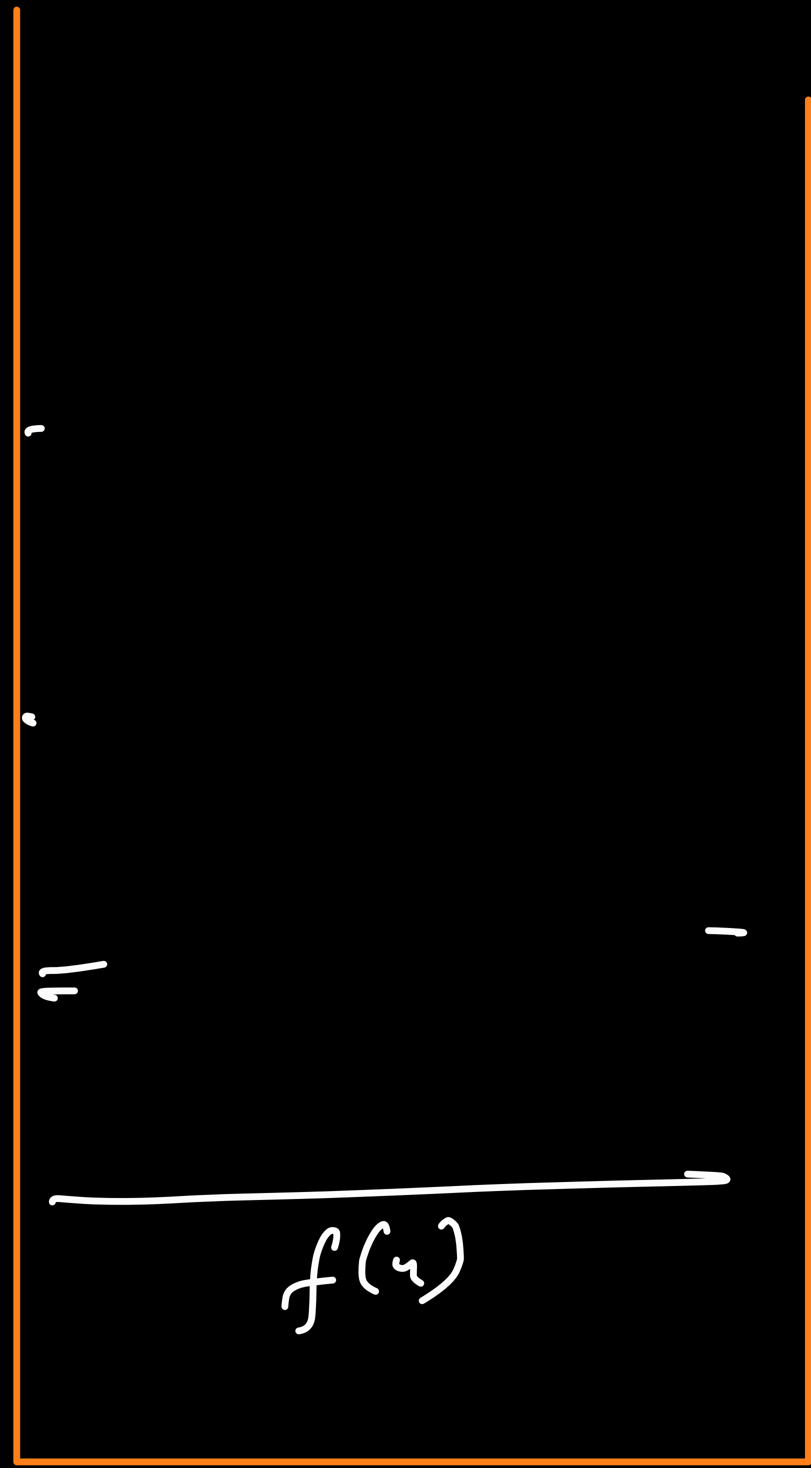


Space of call stack

O(n)

```
function f1(n) {
    if(n == 1 || n == 0) return n;
    return f(n-1) + f(n-2);
}
```

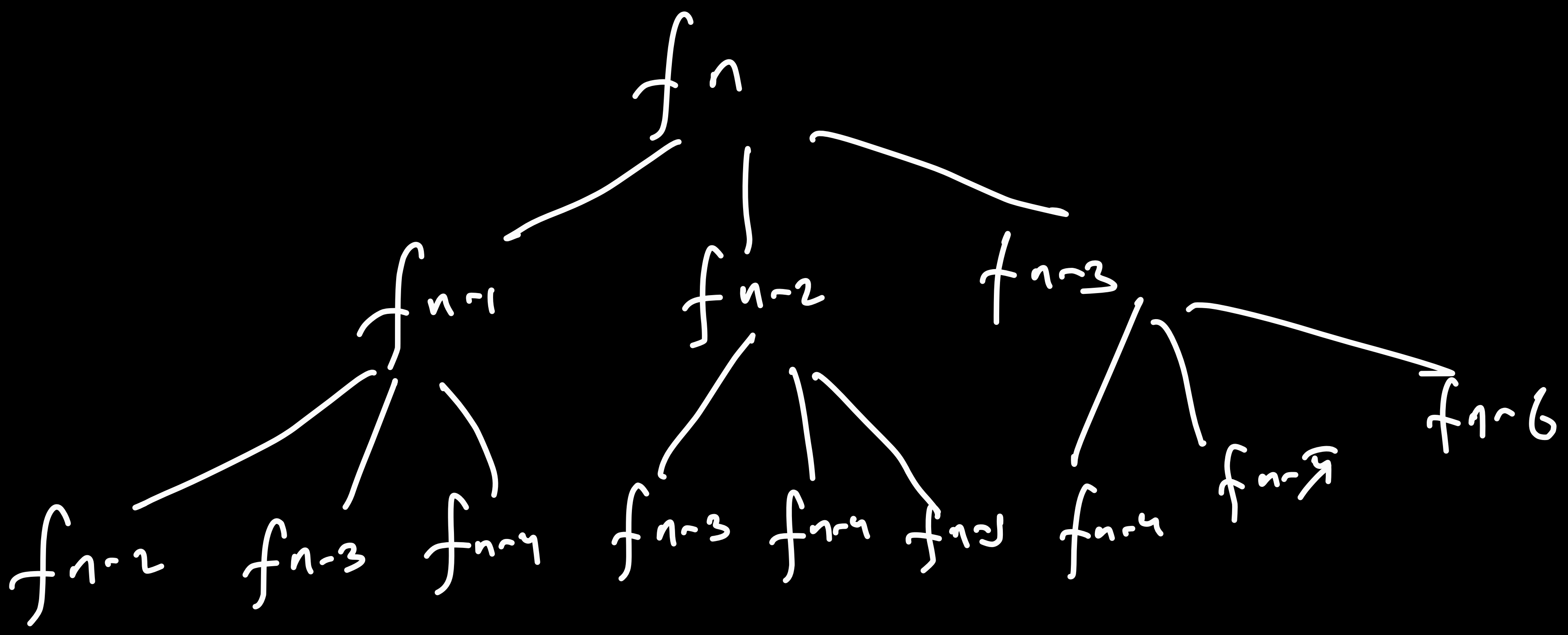




max → n stack frames → O(n)

```
function f(n) {
    if(n ⩽ 1) return 1;
    return f(n-1) + f(n-2) + f(n-3)
}
```

what is the time and space for this ??



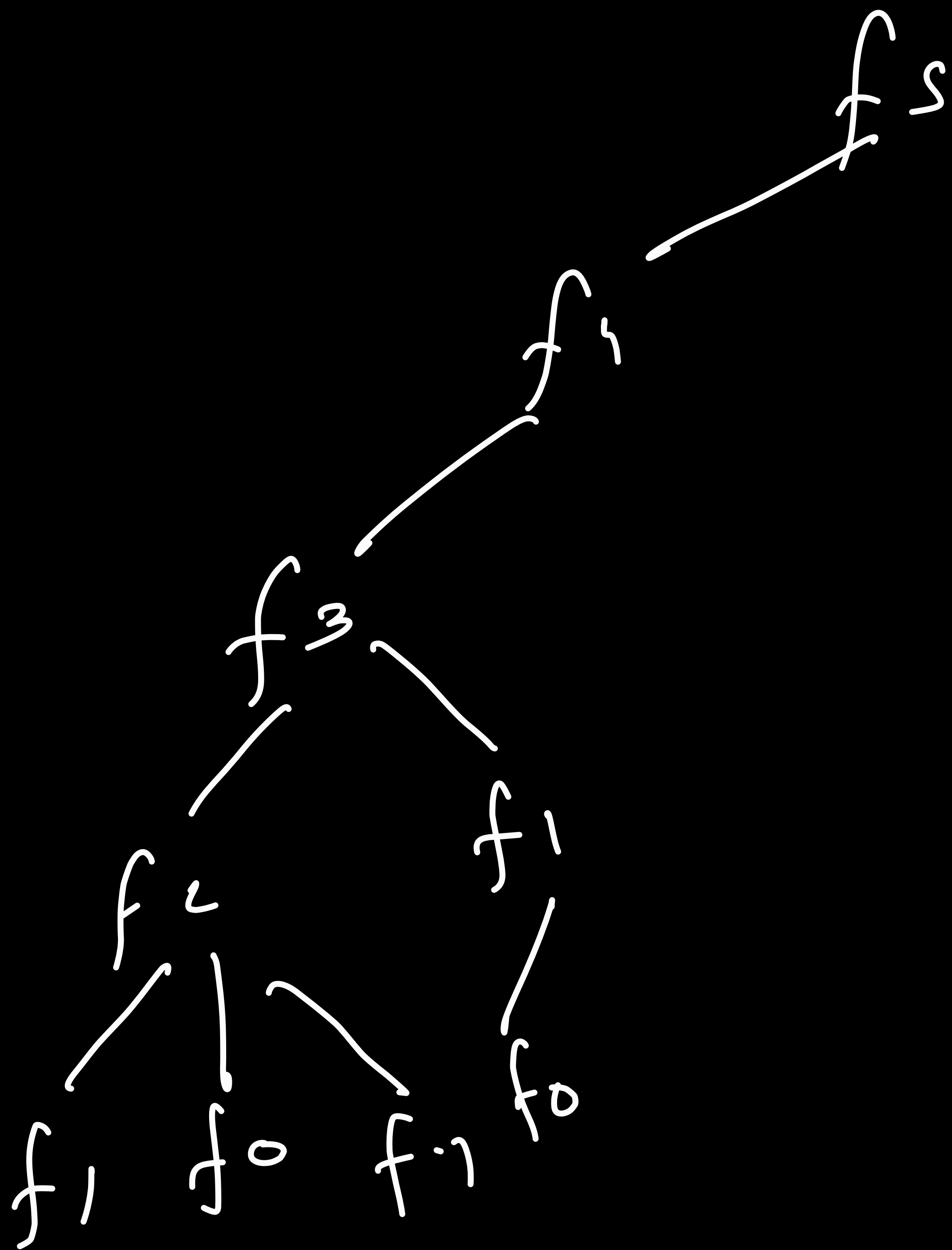
3^0

3^1

3^2

$\approx 3^n \longrightarrow O(3^n) \leftarrow$ time

$O(n) \leftarrow$ Space


f5
f4
f3
f2
f1
f1
f0
f-1
f0

Amortized Analysis

→ good for situation when an algo performs some good ops & some bad ops

→ It refers to determining the time-average running time for a sequence of operations.

It is diff from average asymptotic analysis, because here we do not make any assumption about data values, whereas in the asymptotic average analysis, we assume an overall avg performance.

→ amortized analysis is a good way for analysing complexity for those algo, which perform very good in most of the cases but extremely bad in some of cases

Arrays

[1]

push / append → python / JS / ruby → Dynamic arrays

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

C / C++ / Java → arrays are of fixed size

Vectors

Arraylists

So, arrays are always handled algorithmically for demonstrating dynamic nature.

Q. We have access to fixed size arrays & we have to create dynamic arrays out of it.

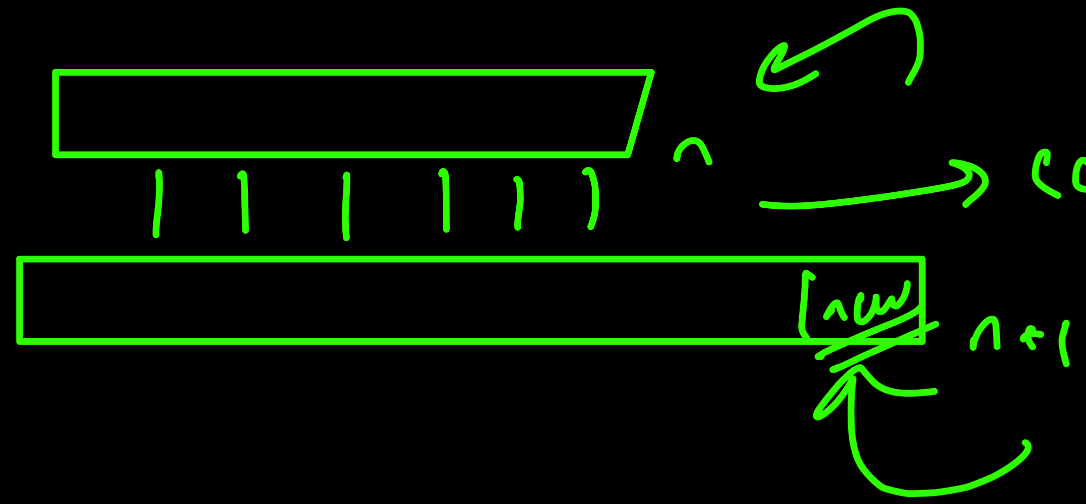
↪

2	1	6	3	8	5

2	1	6	3	8	16	6

Push (16)

Everytime inc the size of array by 1.



→ copy **n elements**

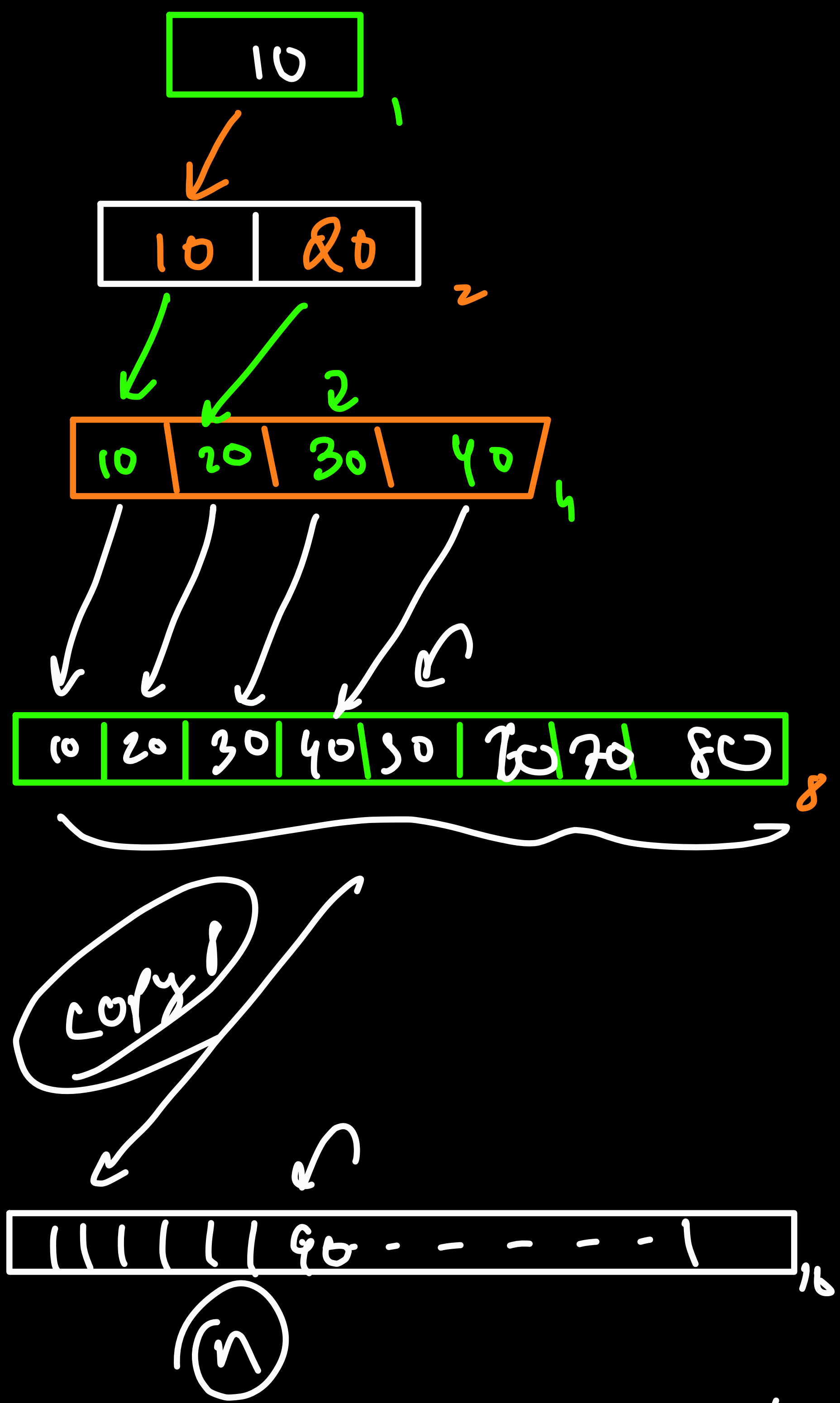
Push at the end of array → **O(n)**

X

Instead of inc the length of array by 1, how about we double it ??

cost

insert → 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110....



operations / ins	capacity	size	element
1	1	1	10
2 → 2^0+1	2	2	20
3 → 2^1+1	4	3	30
1	4	4	40
5 → 2^2+1	8	5	50
1	8	6	60
1	8	7	70
1	8	8	80
9 → 2^3+1	16	9	90
⋮	⋮	⋮	⋮

→ $n \approx 2^K$

average ios = $\dfrac{\text{Total instructions}}{\text{total insertions}}$

$2^k = n$
$k \approx \log n$

$$\frac{(1+2+3+1+5+1+1+1+9+1+1 \ldots\ldots)}{n}$$

$$\frac{(1+(2^0+1)+(2^1+1)+1+(2^2+1)+1+1+1+(2^3+1)+1+1 \ldots)}{n}$$

$$\frac{(\overbrace{1+1+1+1 \ldots\ldots +1}^{\rightarrow n \text{ occ. of ones}}) + (2^0+2^1+2^2+2^3 \ldots\ldots)}{n}$$

$$\frac{n + 1 \times (2^{\log_2 n} - 1)}{n} \Rightarrow \frac{n + n - 1}{n}$$

$$\rightarrow \frac{2n-1}{n} \rightarrow \text{constant}$$

This algo only works for push/pop from last.