Principal of Mathematical Induction (PMI)

Q => Proove that Sum of first n natural numbers, is equal to $(n \times (n+1))/2$

$\rightarrow \frac{(K+1)(K+2)}{2}$

1) Base Case → It is the Smallest input value for which we already know the ans.

↳ $n = 1$

2) Assumption → let's assume formulae work correctly for $n = K$

3) Selfwork → Using the fact that formula works for $n = k$, we will try to prove that formula works for $n = k+1$ also

assume → prove

$K \rightarrow K+1$

$1^{st} \rightarrow 2^{nd} \rightarrow 3 \rightarrow 4^{th} \rightarrow 5^{th}$

Base Case

assumption → for $n = K$

i.e. Sum of first K natural no. is equal to $\frac{K \times (K+1)}{2}$

Selfwork → Proove for $n = K+1$

Sum of first $K+1$ natural no ??

$$\underbrace{1 + 2 + 3 + 4 \ldots\ldots\ldots\ldots + (K-1) + (K)}_{\text{Sum of first } K \text{ natural no.}} + (K+1)$$

$$= \frac{K \times (K+1)}{2} + (K+1) \Rightarrow (K+1)\left[\frac{K}{2} + 1\right] \rightarrow \frac{(K+1)(K+2)}{2}$$

H.P

$$\underbrace{1 + 2 + 3}_{6} + 4$$

Recursion

A child couldn't sleep, so her mother told a story about a little frog,

who couldn't sleep, so the frog's mother told a story about a little bear,

who couldn't sleep, so the bear's mother told a story about a little weasel

...who fell asleep.

...and the little bear fell asleep;

...and the little frog fell asleep;

...and the child fell asleep.

Recursion

→ It is a programming + math concept.

→ What is Recursion ??

Recursion is a technique using which we solve bigger problems by calculating ans of smaller subproblems. We generally denote the bigger problem as a func^n^, & some arguments, & then call the same func^n^ with diff. arguments denoting smaller subproblems. So we get the ans of smaller subproblem & build the ans for bigger problem.

Recursion is funcⁿ calling itself.

We try to solve bigger problems using ans of smaller subproblems

Ex Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1$

$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$

$1! = 1$

$n! \rightarrow n \times (n-1) \times (n-2) \times (n-3) \ldots\ldots\ldots 2 \times 1$

Qn Given a value n, (int) calculate n! recursively

Let's say f is a function which takes a value n as an argument & can calculate n!

Base Case

if (n==1) → 1

$$f(n) = \underbrace{n \times}_{\text{self work}} \underbrace{f(n-1)}_{\text{assume}}$$

f(n) → Calc → n!

implement

self work

assume

f(n-1) works correctly

$5 \times 4!$

$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}$

$4! = 4 \times 3 \times 2 \times 1$

$3! = 3 \times 2 \times 1$

$(K+1)!$??

Base Case → for n==1 we already know that f(n) will be 1.

Assumption assume func^n works correctly for some value K.

$f(K) \rightarrow$ ✓✓

Selfwork $n = (K+1)$ → $(K+1) \times f(K)$

$$(K+1)! \Rightarrow (K+1) \times \underbrace{(k) \times (k-1) \cdots\cdots 3 \times 2 \times 1}_{K!}$$

$K!$ → we already know

$$(K+1)! \Rightarrow (K+1) \times (K)!$$

$n! = n \times (n-1)!$

$$f(n) = \boxed{n \times f(n-1)}$$

$n!$

$f(5) \rightarrow 5 \times f(4)$ 24

$\hookrightarrow 4 \times f(3)$ (6)

$\hookrightarrow 3 \times f(2)$ (2)

$\hookrightarrow 2 \times 1 \; \boxed{f(1)}$

Base Case

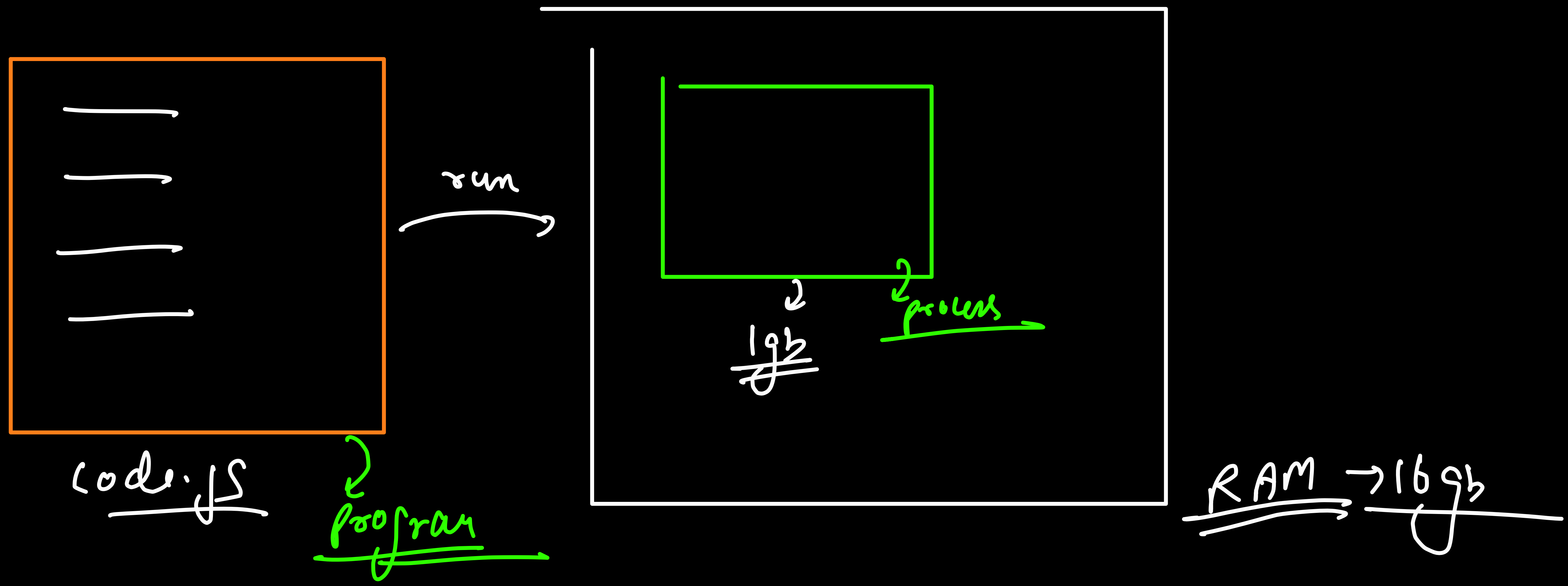
120

$f(n)$

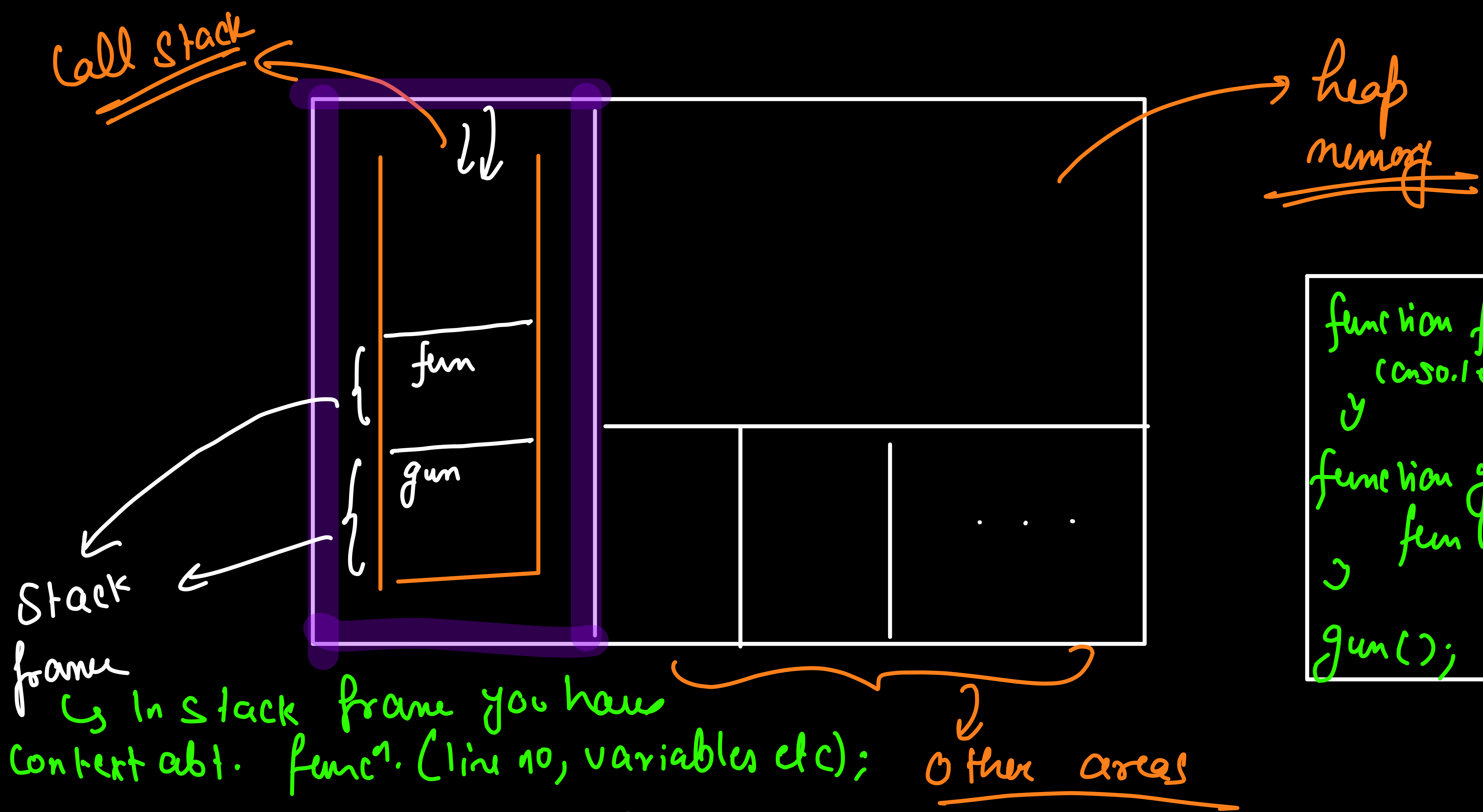
$\downarrow$

$n!$

The magic comes with func^n



Program in a running state is process.

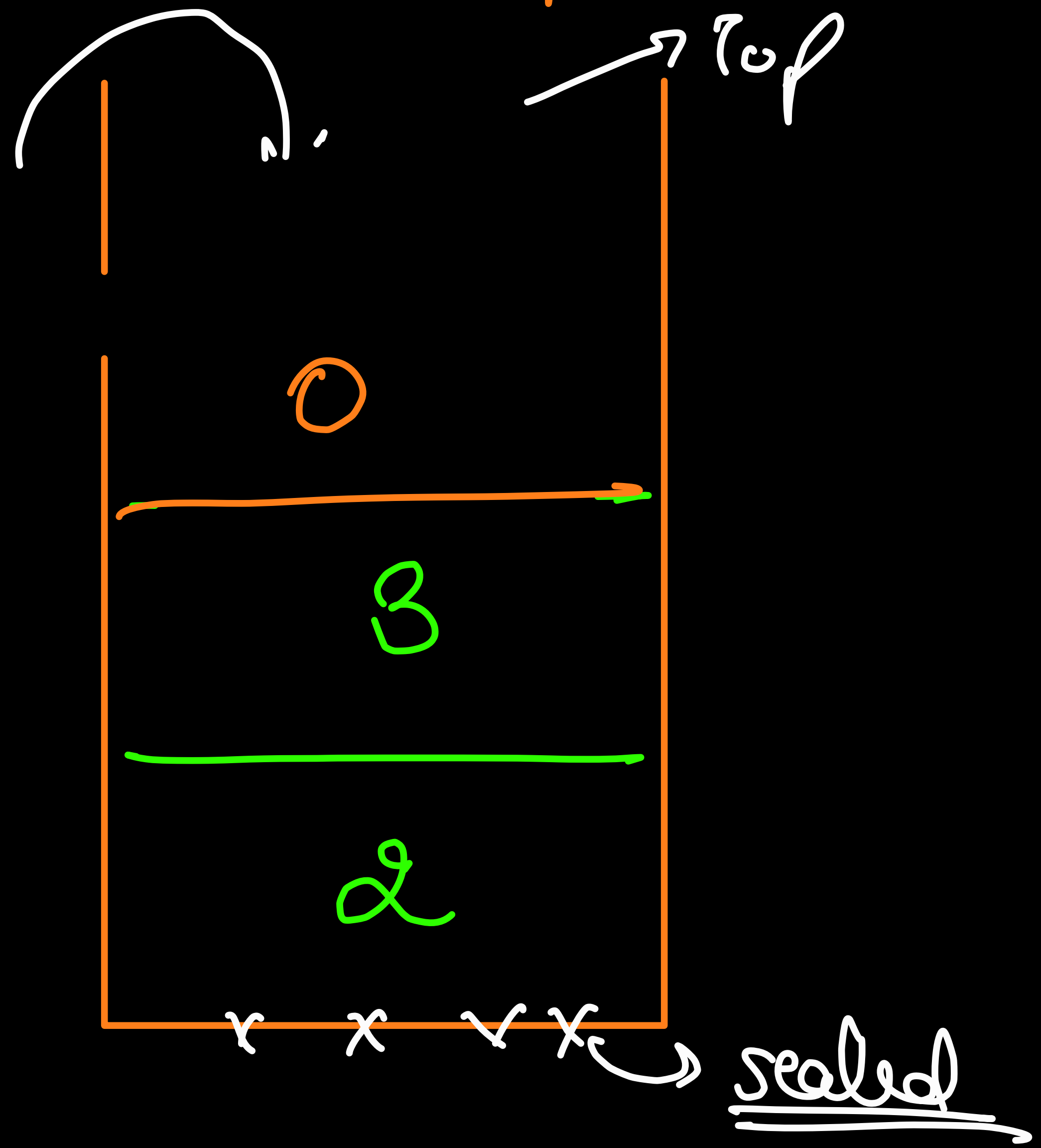



```
function fun(){
    console.log(abc);
}
function gun(){
    fun();
}
gun();
```

↳ In stack frame you have context abt. func^n. (line no, variables etc);

whenever we call a function from anywhere in the code, it adds a new entry in the call stack called as stack frame

Stacks → linear data structure (mental model to store data in different fashion) in which we can add / remove / get data from the top only.

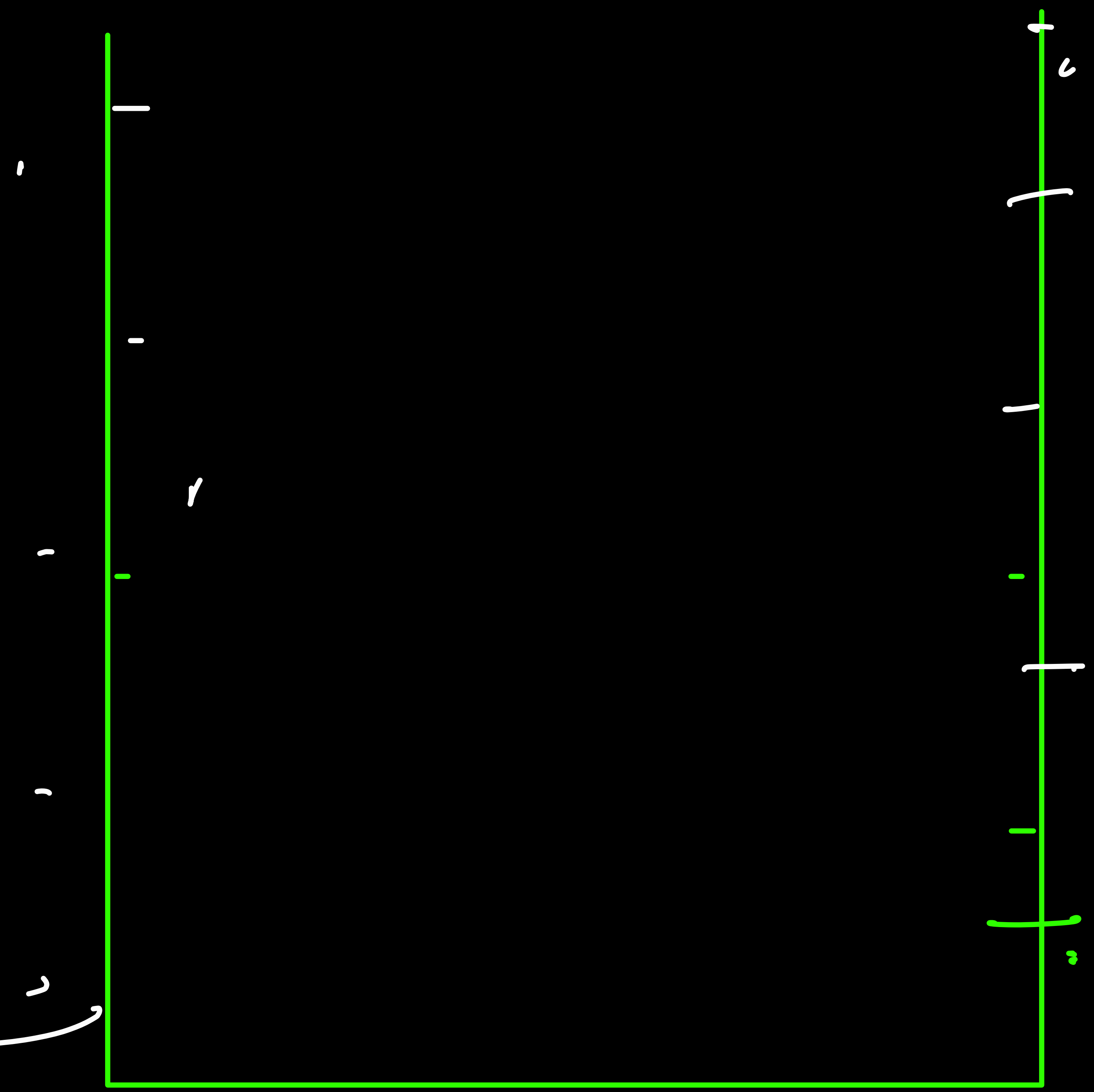


```
function f(n) {
    // base case
    if(n == 1) {
        return 1;
    }
    return n * f(n-1);
}

console.log(f(5));
```

5x 24

120



Q: Given a value n (+ve integer), calculate the nth fibonacci, recursively.

Ex → n = 5

ans → 5

Ex → n = 6

ans → 8

starter

2^{nth}

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 .....

0th, 1st, 2nd, 3rd, 4th, 5th, 6th, 7th, 8th, 9th

Let's say we have a function f, that takes an argument n, & calculates nth fib.

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

nth fib

implement

assume we get this value correctly from f.

self work

if (n==0 || n==1)
return n;

Base Case

if (n==1) return 1;
if (n==0) return 0;

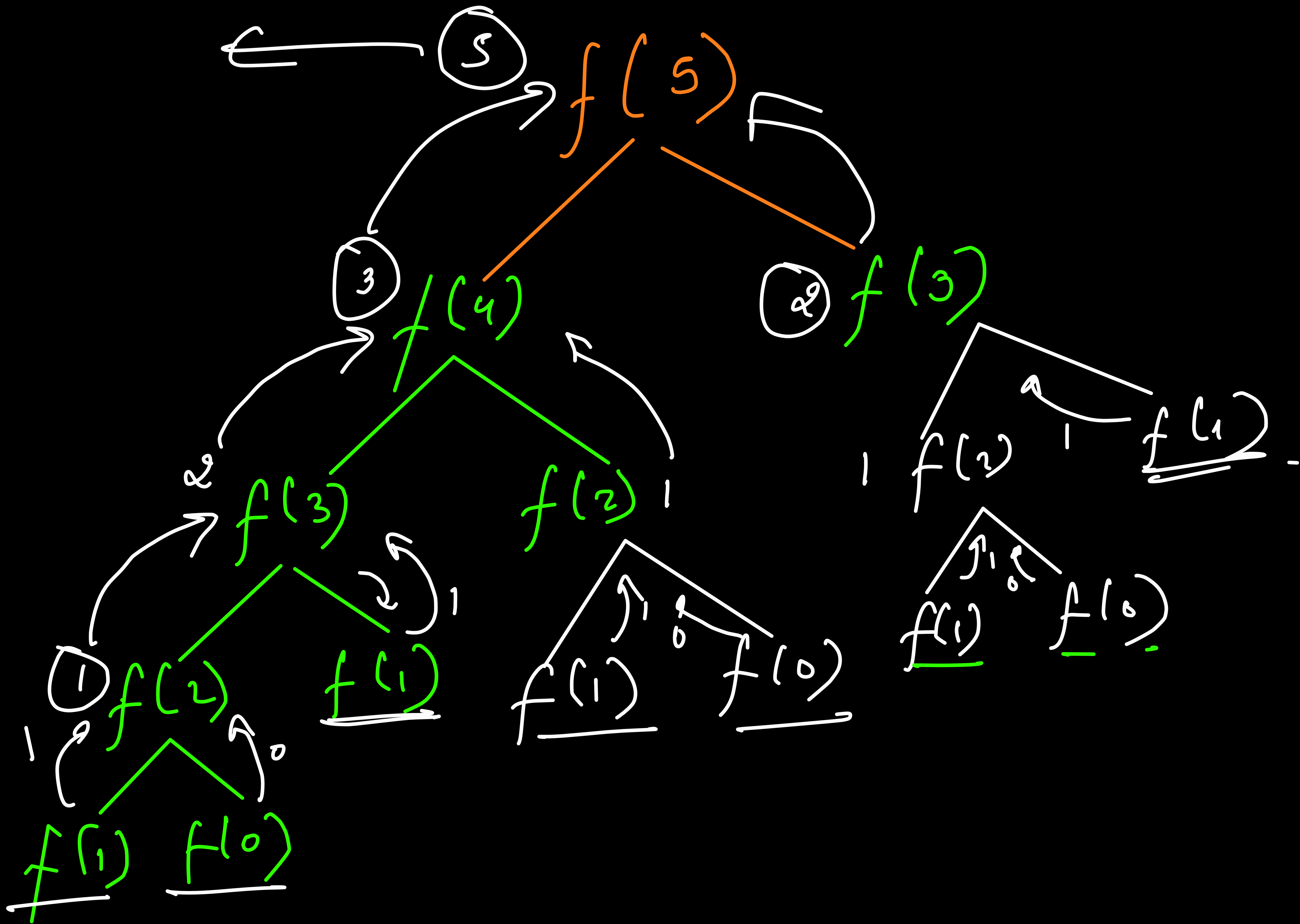
> if (n==0 || n==1)
> return n;

assumption, → Let's assume that the function works correctly for f(n-1) and f(n-2)

$f(n-1) \rightarrow (n-1)^{th}$ fib

$f(n-2) \rightarrow (n-2)^{th}$ fib

Selfwork → add f(n-1) & f(n-2)


5
f(5)
3
f(4)
2
f(3)
f(3)
2
f(2)
1
f(2)
1
f(2)
f(1)
f(1)
f(0)
f(1)
f(0)
f(1)
f(0)
f(1)
f(0)

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

console.log(f(4));
```