"Great things never come from comfort zones".

SCALER

Recursion - 1

TABLE OF CONTENTS

Notes

Recursion

→ Function calling itself

→ Break a problem into subproblems and solve the problem based on the solutions to subproblems.

Find the sum of first n natural nos., given n.

$$sum(n) = \underbrace{1 + 2 + 3 + 4 + \dots + n-1} + n$$

$$sum(n) = sum(n-1) + n$$

⇓

smaller instance of same problem (subproblem)

Write a recursive code :-

→ Assumptions : Decide what the function does for the given problem.

→ Main logic : Express the problem based on subproblems (Solve)

→ Base Case : The inputs for which we need to stop the recursion.

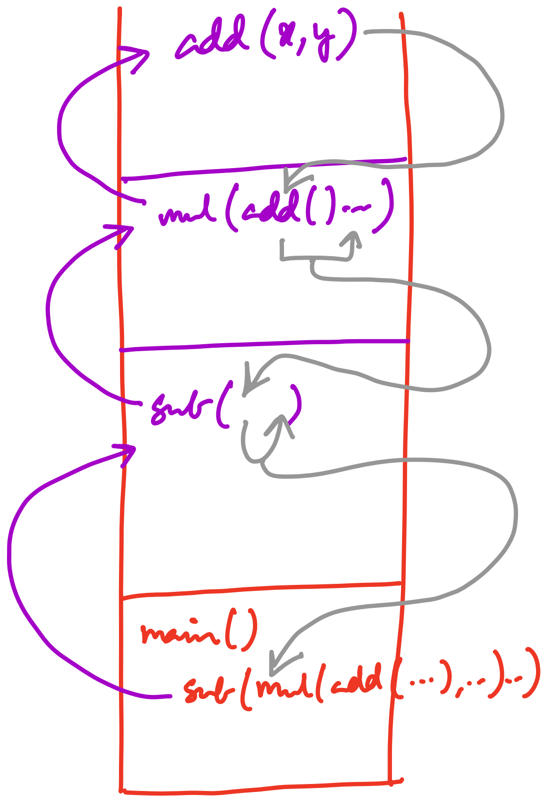
Why Recursion?

→ Pre-requisite of Backtracking , Trees , D.P , Graphs

→ Sorting algo's [Merge Sort , Quick Sort]

Function Call Tracing

```
main() {
    ...
    print(sub(mul(add(x,y), 30),75));
    ...
}
sub(x, y){
    ret x-y
}
mul(x,y){
    ret x*y
}
add(x,y){
    ret x+y
}
```



Function Call Stack

SCALER

Recursion

Sum of first N natural no’s

SCALER

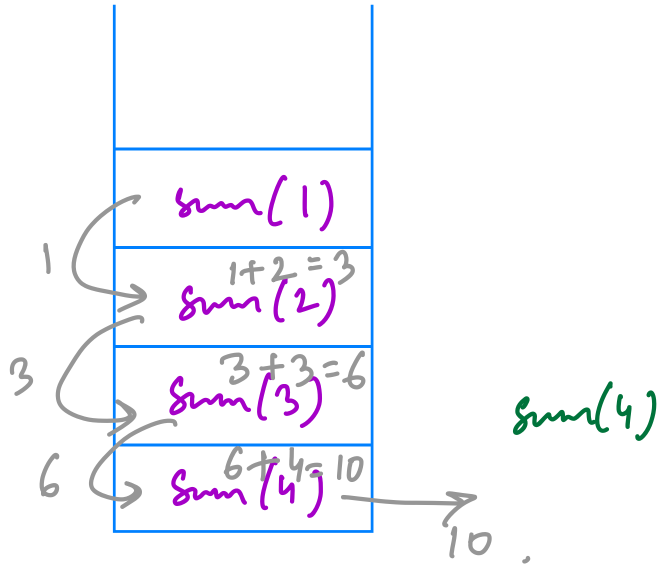
Function Call Tracing

Code-block

```
int add ( int x, int y ){
      return x + y ;
}
int mul ( int n, int y ){
      return x * y ;
}
int sub ( int x, int y ){
      return x - y ;
}
void print ( int x ){
      print (x) ;
}
```

Find the sum of first n natural nos.

```
int sum (int N){
    if(n==1)
        return 1;
    return sum(n-1)+n;
}
```



$O(n)$ T.C.
$O(n)$ S.C.

N Factorial

$5! = 1*2*3*4*5 = 120$

$0! = 1$

fact(n)

Assumption

- Take an integer n (>=0) as parameter
- calculate and return n!

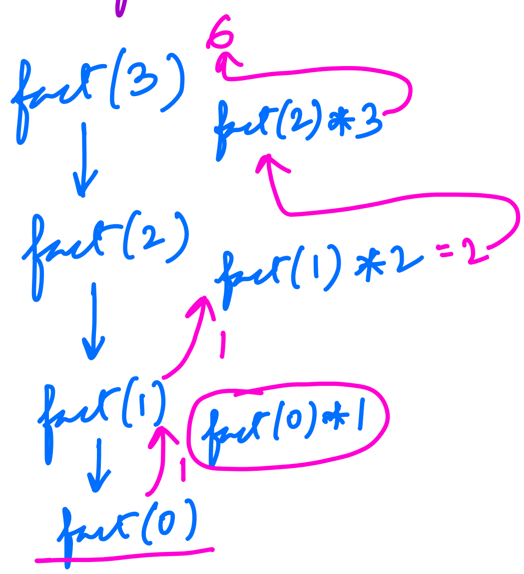
Main logic

fact(n) = fact(n-1) * n

Base case

Stopping condition in recursion.

```
int fact(n){
    if(n==0)
        return 1;
    return fact(n-1)*n;
}
```



dry-run

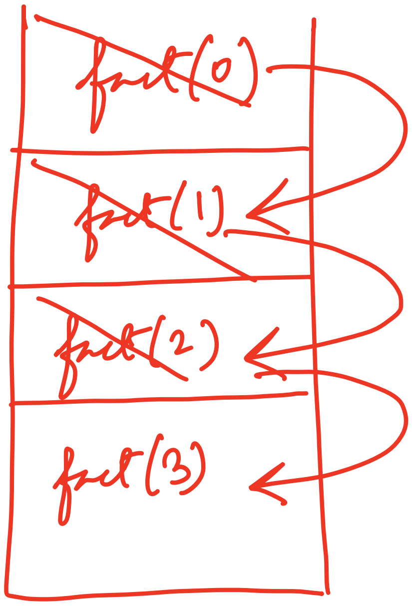
fact(3)

int fact(~~n~~ 3) {
1. if(n==0)
return 1;
2. return $\boxed{\text{fact(n-1)*n}}$; = 2*3 = 6
}

int fact(~~n~~ 2) {
1. if(n==0)
return 1;
2. return $\boxed{\text{fact(n-1)*n;}}$ = 1*2 = 2
}

int fact(~~n~~ 1) {
1. if(n==0)
return 1;
2. return $\boxed{\text{fact(n-1)*n;}}$ = 1
}

int fact(~~n~~ 0) { → Base Case
$\boxed{\text{1. if(n==0) return 1;}}$
2. return fact(n-1)*n;
}



Nth Fibonacci

N =	0	1	2	3	4	5	6	7	8	9	10
F.b →	0 ,	1 ,	1 ,	2 ,	3 ,	5 ,	8 ,	13 ,	21 ,	34 ,	55

Given value of N. Write a recursive function of find N^{th} fibonacci number.

$$fib_n = fib_{n-1} + fib_{n-2}$$

Assumption

fib(n) takes an integer n (>=0) as input and returns n^{th} fibonacci number

Main logic

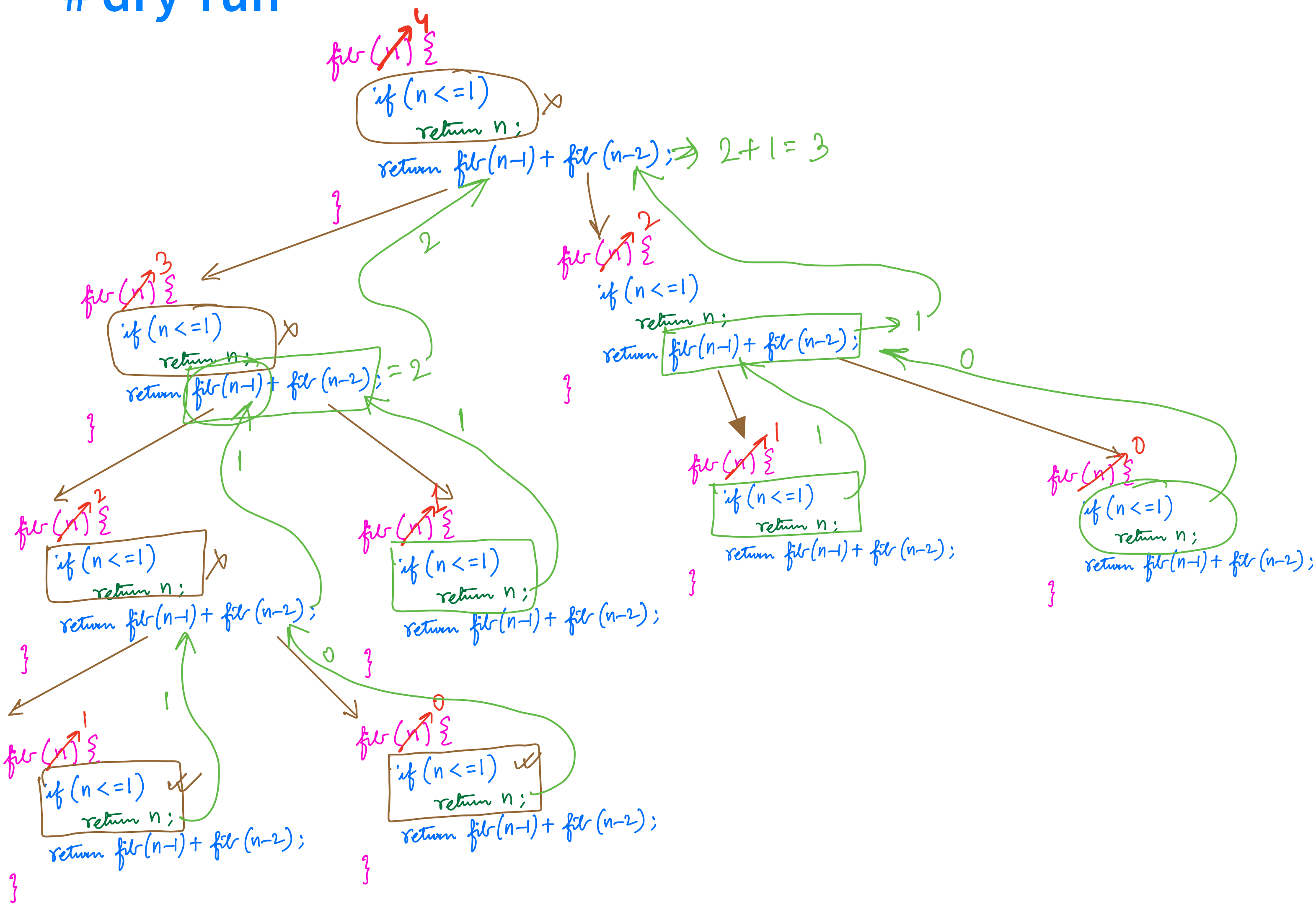
fib(n) = fib(n-1) + fib(n-2)

Base case

$$\left.\begin{array}{l} n=0 \Rightarrow \text{ret } 0 \\ n=1 \Rightarrow \text{ret } 1 \end{array}\right\} \quad n<=1 \Rightarrow \text{ret } n.$$

```
fib(n) {
    if (n<=1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

dry-run



Time Complexity = O(No. of function calls * Time per function call)

```
int fact(n){
    if(n==0)
        return 1;
    return fact(n-1)*n;
}
```

O(1)

fact(n)
↓
fact(n-1)
↓
fact(n-2)
↓
⋮
fact(0)

n+1 fn. calls
(n recursive calls)

$O(1) * (n+1)$
$= O(n)$ T.C.

fact(0)
⋮
fact(n-2)
fact(n-1)
fact(n)

Fn call stack

n+1 stack frame

S.C. → $O(1) * (n+1)$
$= O(n)$.

Time Complexity of Recursion - Using Recurrence relation

```
int fact (int N){
    if (N==0) {return)}
    return fact(N-1)*N;
}
```

T(N) → time taken to calculate factorial of N.
time taken to calculate factorial of N-1 => ?

SCALER

T.C Fibonacci

```
int fib(int N){
    if(N ≤ 1) {return N}
    return fib(N-1) + fib(N-2);
}
```

Recurrence relation →

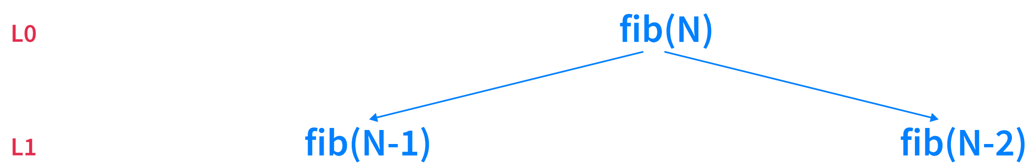
Another definition of Time Complexity

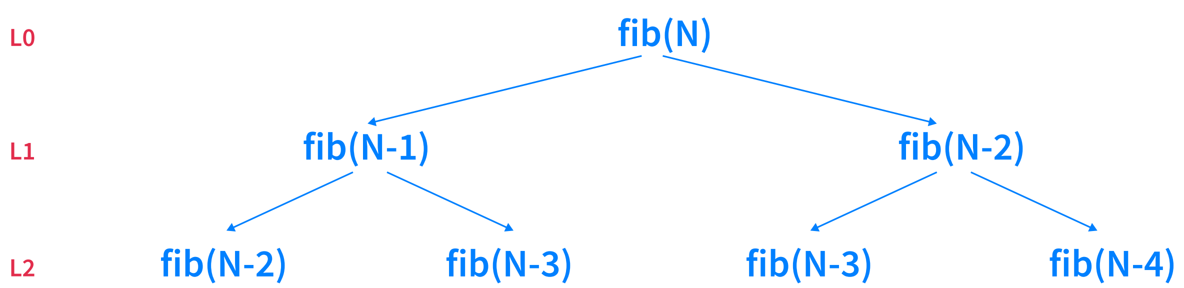
```
int fact (int N){
    if (N==0) {return     )}
    return fact(N-1)*N;
}
```

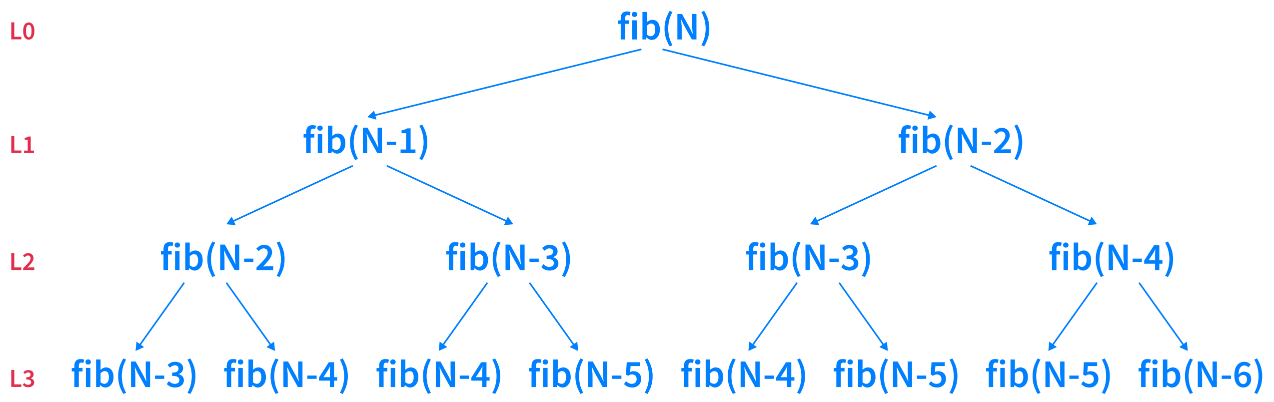
```
int fib(int N){
    if(N ≤ 1) {return N}
    return fib(N-1) + fib(N-2);
}
```

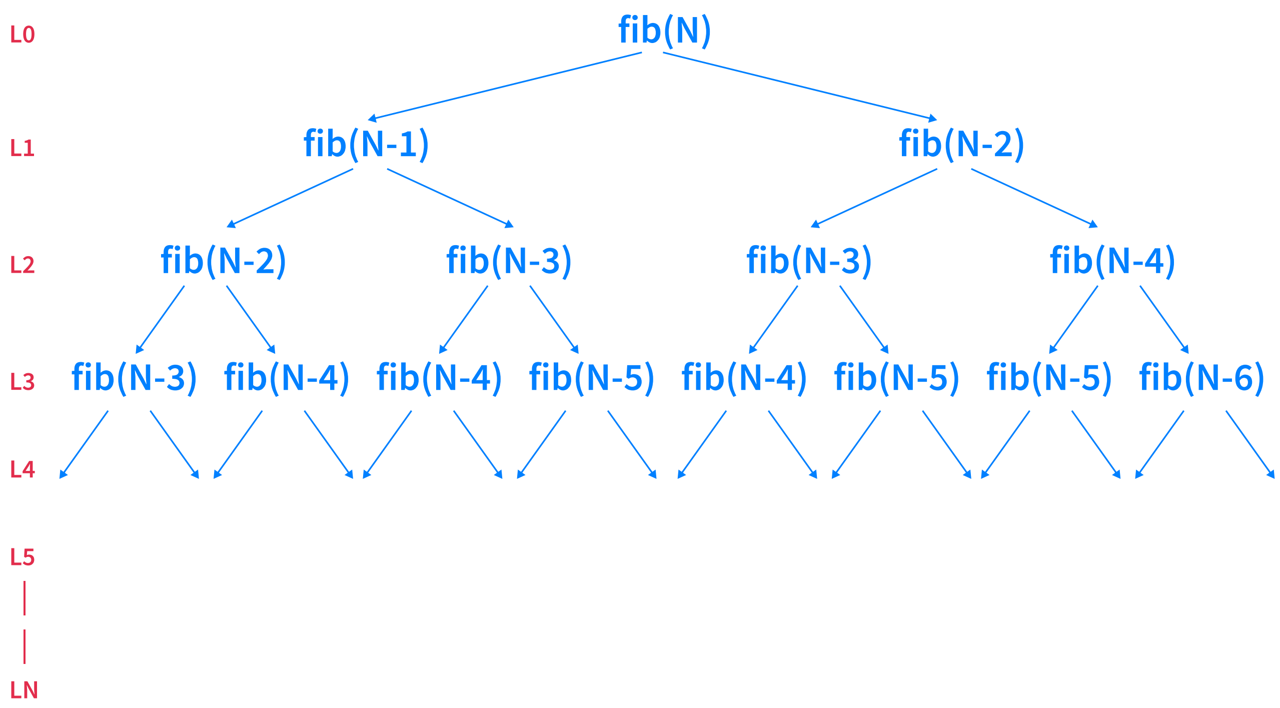
O(1) T.C. per
fn. call.

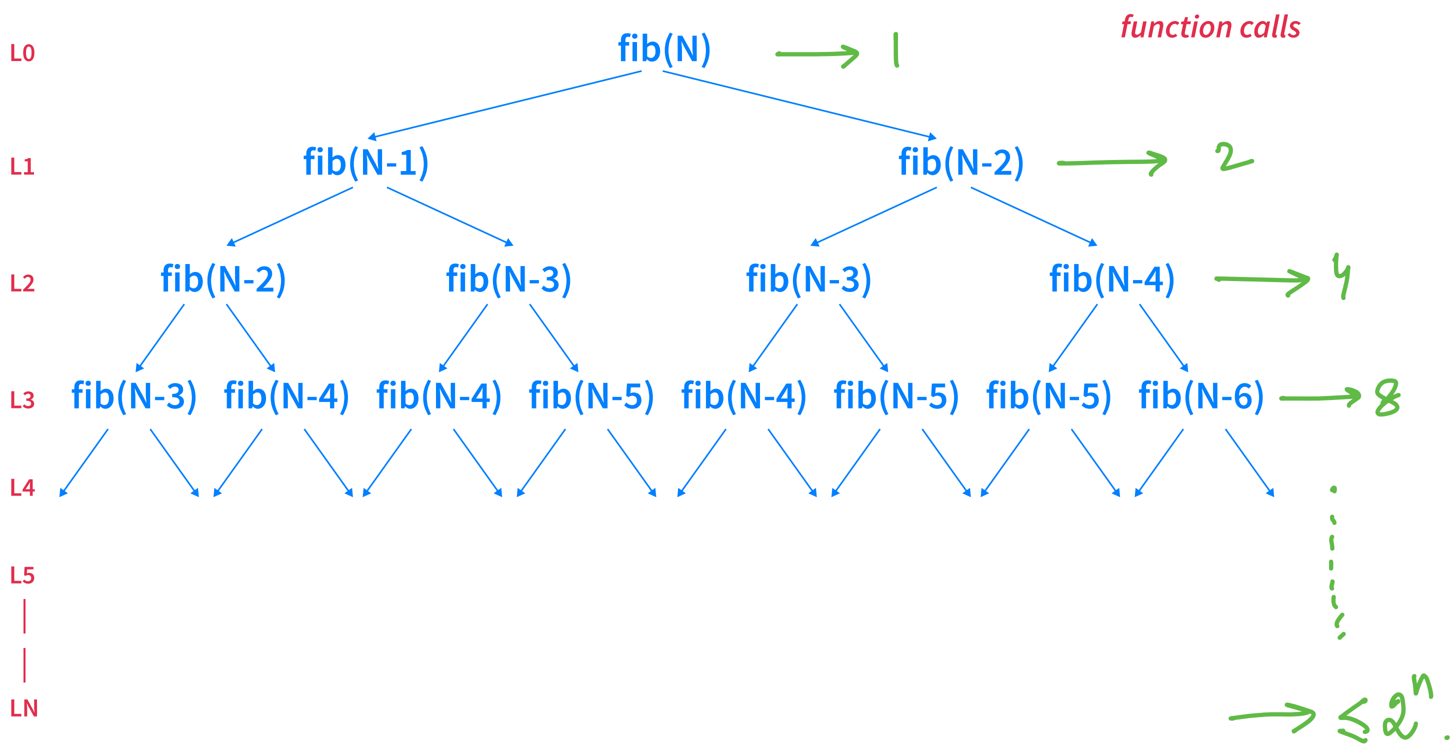

L0
fib(N)


L0
fib(N)
L1
fib(N-1)
fib(N-2)


L0
fib(N)
L1
fib(N-1)
fib(N-2)
L2
fib(N-2)
fib(N-3)
fib(N-3)
fib(N-4)


L0
fib(N)
L1
fib(N-1)
fib(N-2)
L2
fib(N-2)
fib(N-3)
fib(N-3)
fib(N-4)
L3
fib(N-3)
fib(N-4)
fib(N-4)
fib(N-5)
fib(N-4)
fib(N-5)
fib(N-5)
fib(N-6)


L0
fib(N)
L1
fib(N-1)
fib(N-2)
L2
fib(N-2)
fib(N-3)
fib(N-3)
fib(N-4)
L3
fib(N-3)
fib(N-4)
fib(N-4)
fib(N-5)
fib(N-4)
fib(N-5)
fib(N-5)
fib(N-6)
L4
L5
LN



total function calls : $2^0 + 2^1 + 2^2 + \dots + 2^n$

$$= 1 * \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

$$= O(2^n).$$

$$T.C \rightarrow O(1) * O(2^n) = O(2^n).$$

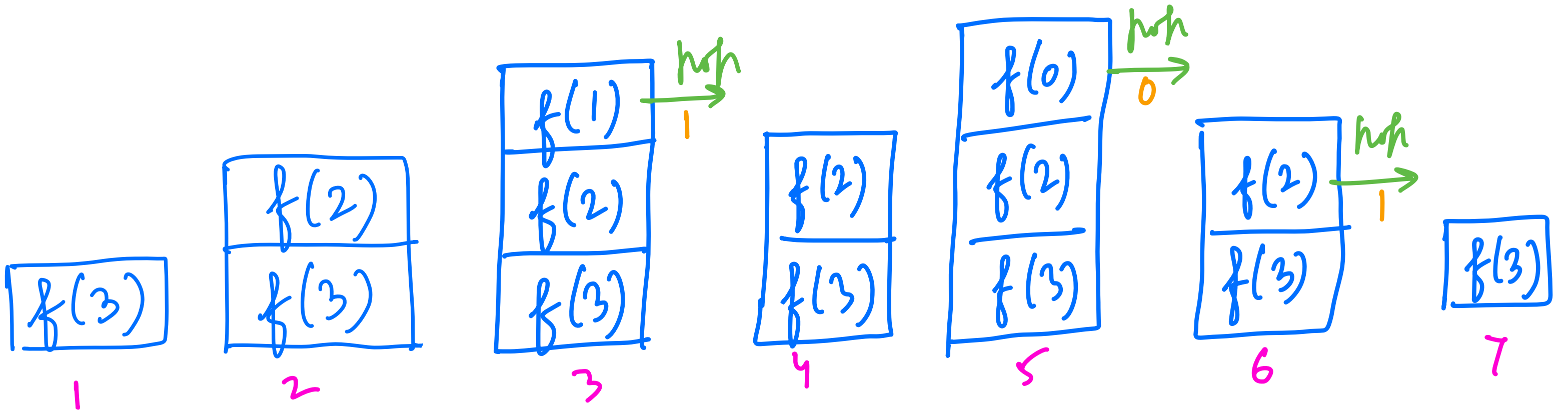
Space Complexity

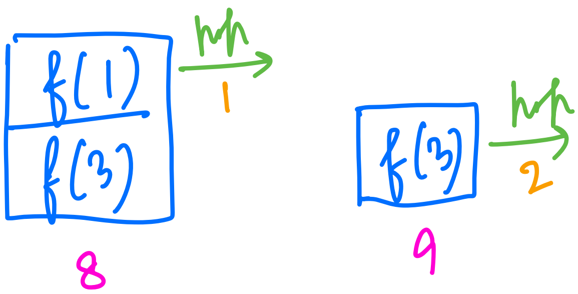
S.C. = Height of the Recursion Tree (Depth) * Space for each function call

$= (n+1) * O(1) = O(n).$

fib(3)

```
fib(n) {
    if (n<=1)
        return n;
    return fib(n-1) + fib(n-2);
}
```





S.C. → Max. depth/size of rec. call stack * Size of each frame

$= n * O(1) = O(n).$

[Break till 10:55 PM]

Q) Given n, print all nos. from 1 to n in increasing order. (n>0)

inc(5) → $\boxed{1 \quad 2 \quad 3 \quad 4}$ 5

inc(4)

↓

1. inc(4)
2. print(5)

Assumption :- inc(n) will print all nos. from 1 to n in inc. order.

Main logic :-

inc(n) → inc(n−1) → print all nos. 1 to n−1

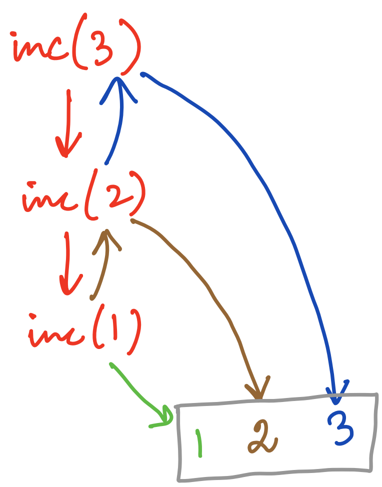
→ print(n)

Base case :-

n=1 → print(1).

```
void inc(n){
    if(n==1){
        print(n)
        return
    }
    inc(n-1)
    print(n)
}
```

$\boxed{\begin{array}{c} O(n)\ T.C \\ O(n)\ S.C. \end{array}}$



H/W Given n, print all nos. from 1 to n in decreasing order. (n>0)

(n to 1)