Recursion - 1

↓

→ function calling itself

→ Problem is broken down to smaller sub-problems & using the solution of smaller sub-problems, big problem is solved

Q → find sum of first N natural numbers using recursion.

$sum(N) = 1 + 2 + 3 + \ldots + N = \boxed{\frac{N*(N+1)}{2}}$

$sum(5) = \underbrace{1+2+3+4}_{sum(4)} + 5$

$sum(5) = sum(4) + 5$

$\boxed{sum(N) = sum(N-1) + N}$

↳ $sum(N-2) + (N-1)$

⋮

Steps to write recursive code

1. Define the problem / function

$sum(N) = 1 + 2 + 3 + \ldots + N$

2. Define recursive relation

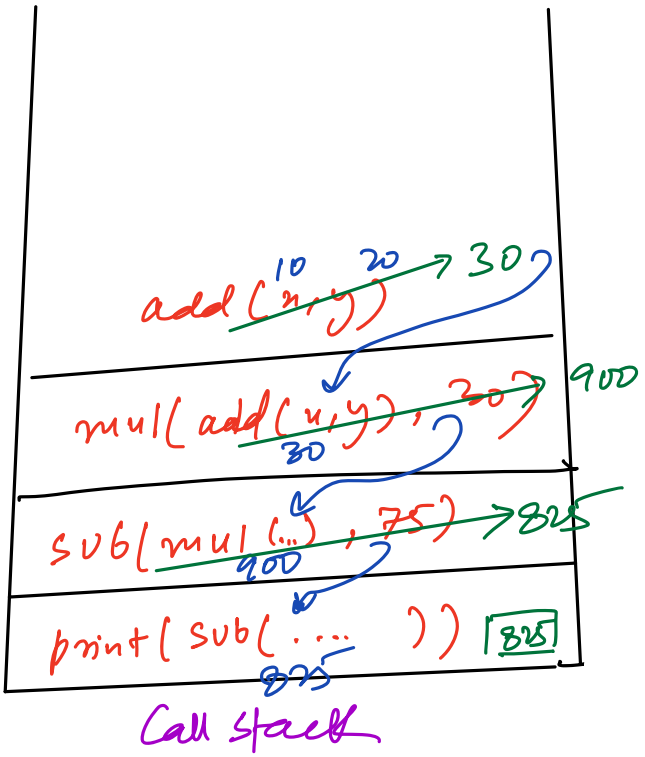
$sum(N) = sum(N-1) + N$

3. Define base case. // simplest version of the problem

$sum(1) = 1 \quad (N=1)$

function call tracing

```
int add( int x, int y ){
    return x+y
}
int mul( int x, int y ){
    return x*y
}
int sub( int x, int y ){
    return x-y
}
void main() {
    x = 10, y = 20
    print( sub( mul( add(x,y), 30 ), 75 ))
}
```



Call stack

Q→ Given a +ive integer N, find factorial of N.

N=3 fact(3) = 1 × 2 × 3 = 6

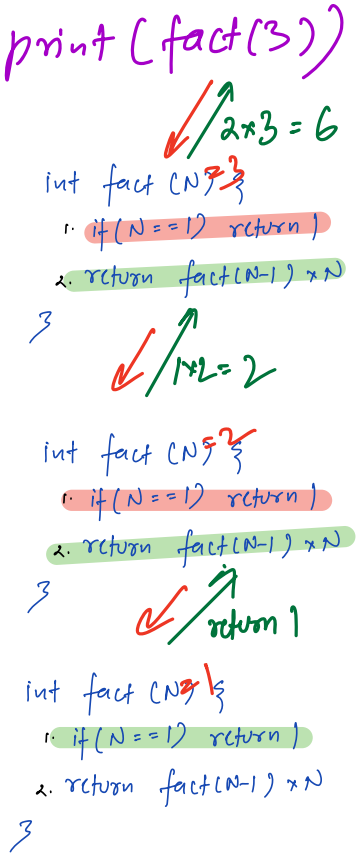
N=5 fact(5) = 1 × 2 × 3 × 4 × 5 = 120

fact(4)

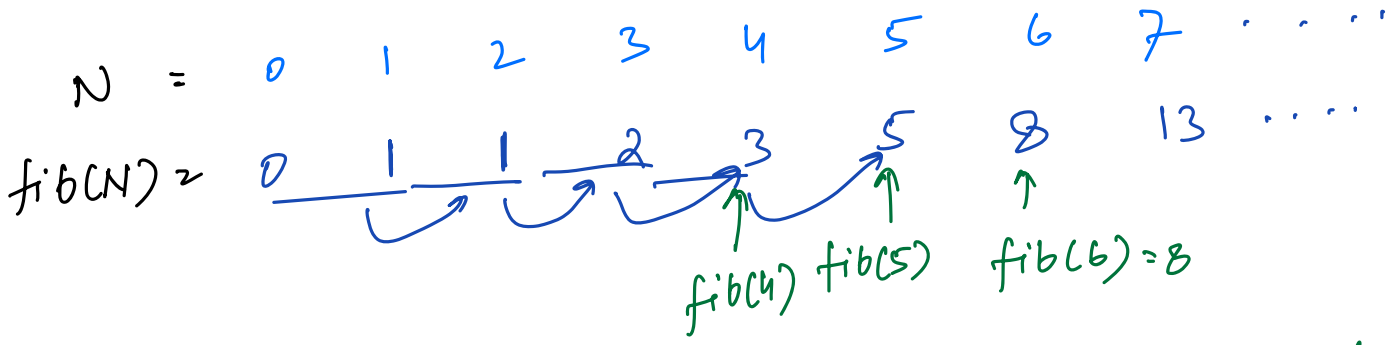
1. fact(N) = 1 × 2 × 3 × . . . . × N
2. fact(N) = fact(N-1) × N
3. fact(1) = 1

```
int fact (N) {
  1. if (N == 1) return 1
  2. return fact(N-1) * N
}
```

print (fact(3))



Q → Given a +ive number N, find N^{th} fibonacci number.



fib(6) = fib(5) + fib(4)

1. fib(N) → N^{th} fibonacci no.

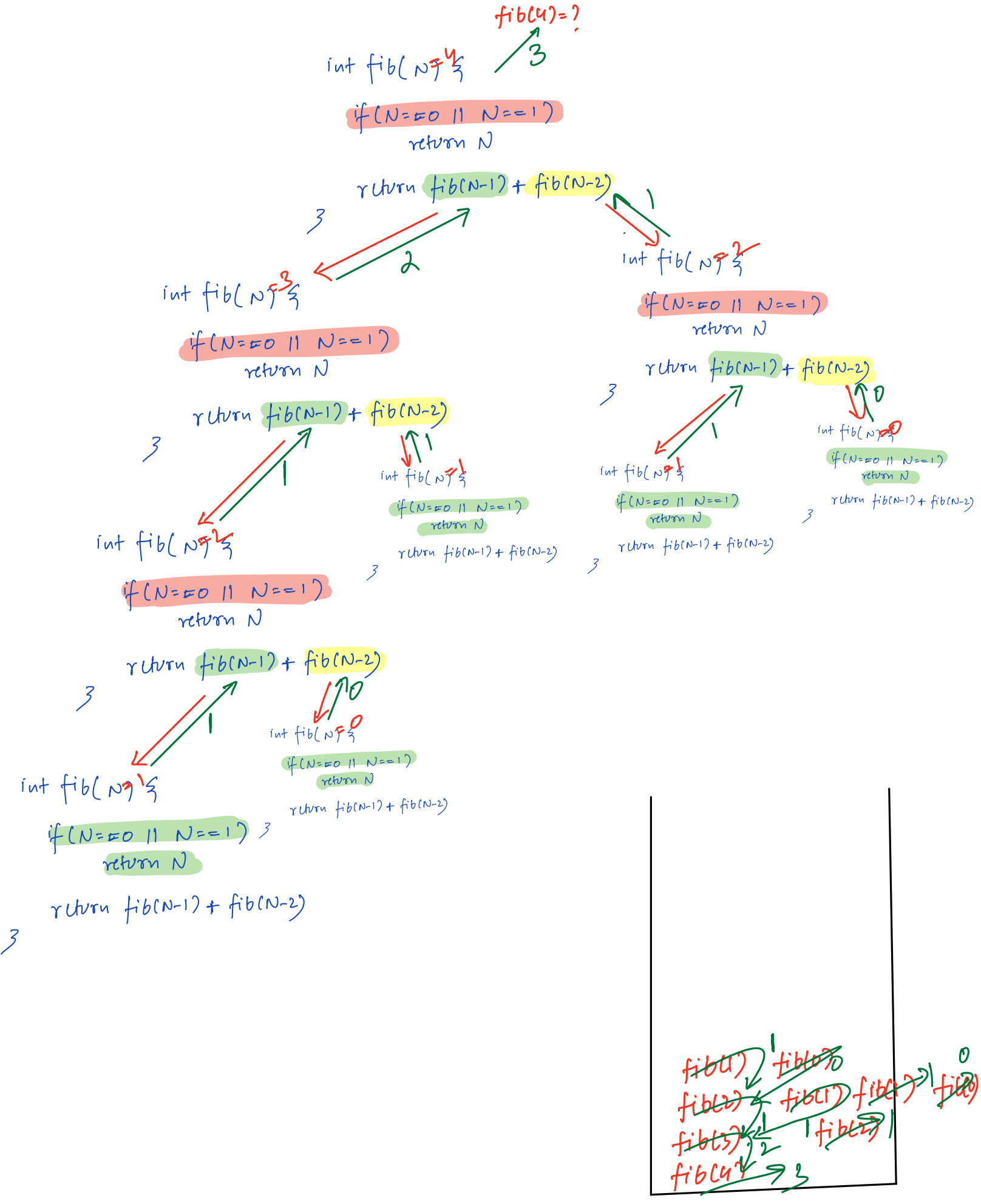
2. fib(N) = fib(N-1) + fib(N-2)

3. fib(0) = 0
   fib(1) = 1

fib(3) = fib(2) + fib(1)
↓
fib(1) + fib(0) (1, 0)
↓
fib(0) + fib(-1)

```
int fib(N) {
    if(N==0 || N==1)
        return N

    return fib(N-1) + fib(N-2)
}
```


fib(4)=?
int fib(N=4){
if(N==0 || N==1)
return N
return fib(N-1) + fib(N-2)
int fib(N=3){
int fib(N=2){
int fib(N=1){
int fib(N=0){
int fib(N=2){
int fib(N=1){
int fib(N=0){
int fib(N=1){

fib(4)
fib(3)
fib(2)
fib(1)
fib(0)
fib(2)
fib(1)
fib(0)
fib(1)
fib(0)
fib(3)
fib(4)
0
1
1
2
3

Q → Given 2 +ive numbers a & b, find a^b using recursion.

eg $a = 2$
$b = 3$

$2^3 = \underbrace{2 \times 2}_{2^2} \times 2 = 8$

1. pow(a,b) → a x a x a x . . . . . x a (b times)
2. pow(a,b) = pow(a, b-1) x a
3. pow(a,0) = 1 [b=0]

```
int pow(a, b) {
    if (b==0) return 1;
    return pow(a, b-1) * a
}
```

Alternate Soln →

$a^6 = a^3 \times a^3$

$a^{10} = a^5 \times a^5$

$a^7 = a^3 \times a^3 \times a$

$a^b = a^{b/2} \times a^{b/2}$ b → even

$a^b = a^{b/2} \times a^{b/2} \times a$ b → odd

1. $pow(a,b) = a^b$

2. $pow(a,b) =$
   - → $pow(a,b/2) * pow(a,b/2)$ [b = even]
   - → $pow(a,b/2) * pow(a,b/2) * a$ [b = odd]

3. $pow(a,0) = 1$

FAST exponentiation

```
int pow(a, b) {
  if (b == 0) return 1
  if (b % 2 == 0) {
    return pow(a,b/2) * pow(a,b/2)
  }
  else {
    return pow(a,b/2) * pow(a,b/2) * a
  }
}
```

```
int pow(a, b){
  if (b == 0) return 1
  x = pow(a,b/2)
  if (b % 2 == 0)
    return x * x
  else
    return x * x * a
}
```

pow(2,5) ↑ 4*4*2 = 32
- ↓ pow(2,2) ↑ 2*2 = 4
  - ↓ pow(2,1) ↑ 1*1*2 = 2
    - ↓ pow(2,0) ↑ 1

$2^5 = 32$

Time Complexity → O(# iterations)

→ O(#function calls × TC of function)

```
int fact(N) {
1.  if(N==1) return 1
2.  return fact(N-1) * N
}
```

O(N)

O(1) ⇒ O(N×1) = O(N)

$$T(N) = T(N-1) + 1$$
$$= [T(N-2) + 1] + 1$$
$$= (T(N-3) + 1) + 1 + 1$$
$$T(N) = T(N-3) + 3$$

$$T(N) = T(N-K) + K$$

if $N-K = 1$

$K = N-1$

$$= T(1) + N - 1$$
$$= 1 + N - 1$$
$$= N$$

$$\boxed{T(N) = O(N)}$$

```
int pow(a,n) {
    if (n==0) return 1;
    return pow(a,n-1) * a
}
```

$$T(N) = T(N-1) + 1 = O(N)$$

```
int pow(a,n) {
    if (n==0) return 1
    if (n%2==0) {
        return pow(a,n/2) * pow(a,n/2)
    }
    else {
        return pow(a,n/2) * pow(a,n/2) * a
    }
}
```

$$T(n) = T(n/2) + T(n/2) + 1$$

$$T(n) = 2T(n/2) + 1$$

$$= 2\left(2T(n/4) + 1\right) + 1$$

$$= 2^2 T(n/4) + 2 + 1$$

$= 2^2\left(2T(n/8)+1\right) + \frac{2+1}{3} = 2^2-1$

$= 2^3\,T\left(n/2^3\right) + \frac{4+2+1}{7} = 2^3-1$

$$T(n) = 2^K T\left(n/2^K\right) + 2^K - 1$$

$n/2^K = 1 \Rightarrow 2^K = n$

instead of T(0) take T(1) as base

$$T(n) = n\,T(1) + n - 1$$

$$= n + n - 1$$

$$\boxed{T(n) = O(N)}$$

```
int pow(a, n){
  if (n==0) return 1
  x = pow(a, n/2)
  if (n%2 == 0)
     return x * x
  else
     return x * x * a
```

$T(n) = T\left(n/2^1\right) + 1$

$= T\left(n/4_{2^2}\right) + 1 + 1$

$= T\left(n/8_{2^3}\right) + 1 + 2$

$T(n) = T\left(n/2^K\right) + K$

$T(1) = 1$

3

$n/2^k = 1 \Rightarrow 2^k = n$

$k = \log_2 n$

$T(n) = T(1) + \log n$

$T(n) = 1 + \log n$

$\boxed{T(n) = \log_2 n}$

```
int fib(N) {
    if(N==0 || N==1)
        return N
    return fib(N-1) + fib(N-2)
}
```

$T(N) = T(N-1) + T(N-2) + 1$

$\approx T(N-1) + T(N-1) + 1$

$\approx 2T(N-1) + 1$

$= 2(2T(N-2) + 1) + 1$

$= 2^2 T(N-2) + 3$

$T(N) = 2^K T(N-K) + 2^K - 1$

$T(0) = 1 \qquad N-K = 0 \Rightarrow K = N$

$$T(N) = 2^N T(1) + 2^N - 1$$

$$= 2 \cdot 2^N - 1$$

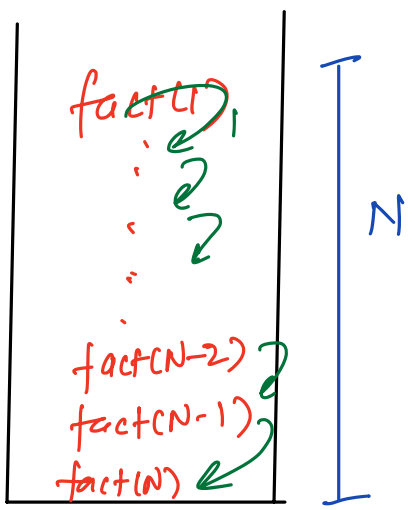
$$\boxed{T(N) = O(2^N)}$$

fact(6) ⇄ fact(5) ⇄ fact(4) ⇄ fact(3) ⇄ fact(2) ⇄ fact(1)

6 calls.

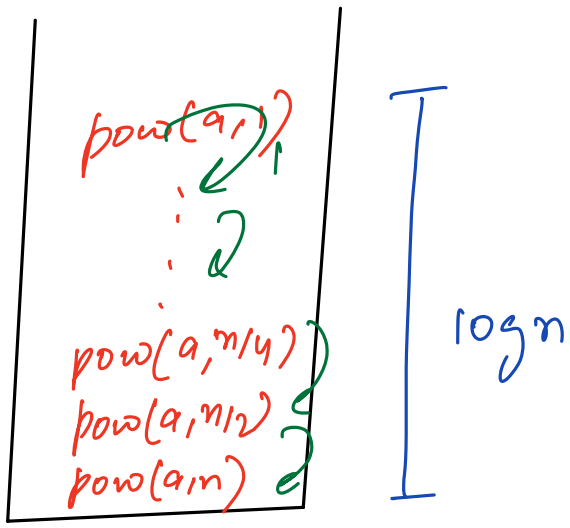
Space Complexity

```
int fact (N) {
  1. if (N == 1) return 1
  2. return fact(N-1) * N
}
```



$$SC = O(N)$$

```
int pow(a, n){
    if (n==0) return 1
    x = pow(a, n/2)
    if (n%2 == 0)
        return x * x
    else
        return x * x * a
}
```

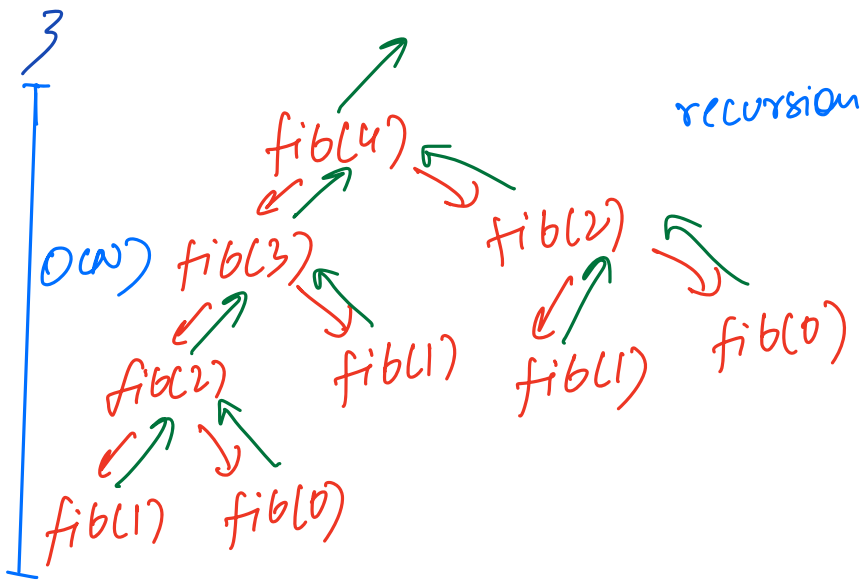


$n \to n/2 \to n/4 \to \cdots \to 1$

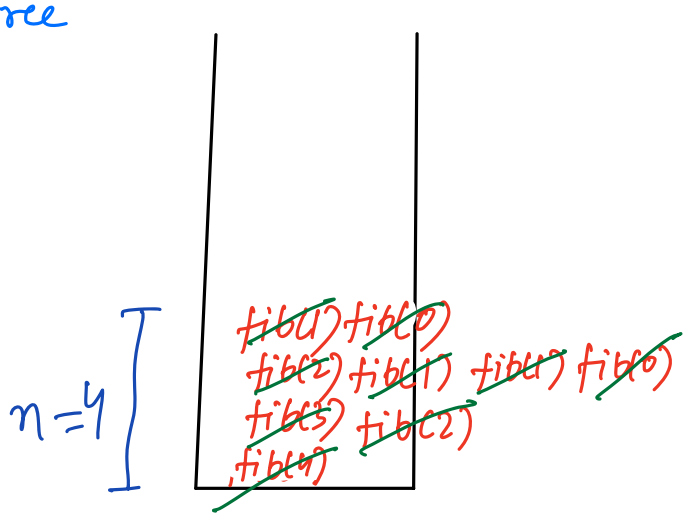
logn times

$\therefore C = O(\log_2 N)$

```
int fib(N) {
    if(N==0 || N==1)
        return N
    return fib(N-1) + fib(N-2)
}
```



recursion free



$SC = O(N)$