Recursion: function calling itself.

- → Merge Sort & Quick Sort
- → Trees | BST
- → Heap | Segment Tree
- → Dynamic Programming
- → Graphs.

→ $Sum(N) = \underbrace{1+2+3+4+\cdots+(N-1)}_{Sum(N-1)}+N$

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

→ Recursion: Solve the bigger problem using the sol^ of smaller sub problems.

 # 3 Steps to write Recursive code:

1. Assumption | Trust
2. Main logic:
   Solve the bigger problem using the smaller subproblem.
3. Base | Exit Condition
   ⇒ When you want to stop executing the main logic.

```
int sum(N) {
  1. Assume: Sum(N) returns the sum of numbers from 1 to N.
  2. Sum(N) → Sum(N-1) + N
  3
```

Sum(N) = Sum(N-1) + N

N = 5

15

Sum(5) { //N = 5

10 { Sum(4) + 5

3

10

Sum(4) { //N = 4

6 { Sum(3) + 4 ;

3

6

Sum(3) { //N = 3

3 { Sum(2) + 3

3

3

Sum(2) { //N = 2

1 { Sum(1) + 2 ;

3

1

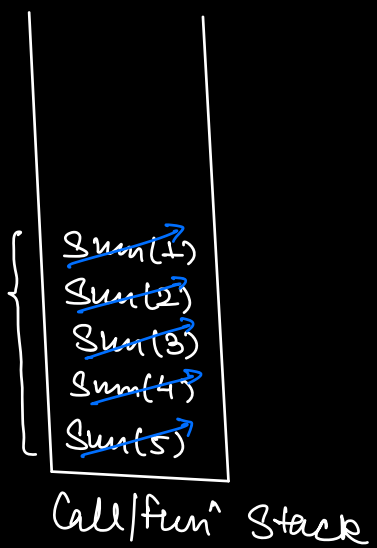
Sum(1) {

//N = 1

if(N == 1)

return 1;

3



Call/fun^ Stack

Sum(5)
↓
Sum(4)
↓
Sum(3)
↓
Sum(2)
↓
Sum(1)

TC of a recursive fun^ =
(No. of fun^ calls) * (TC of 1 fun^ call)
N O(1)

TC: O(N)

Recurrence Relation

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

1 → Addition

⇒ Substitution Method

$$T(N) = T(N-1) + 1$$
$$= [T(N-2) + 1] + 1$$
$$= T(N-2) + 2$$
$$= \{T(N-3) + 1\} + 2$$
$$= T(N-3) + 3$$
$$\vdots$$

After K steps

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

$$N-K = 0$$
$$\Rightarrow K = N$$

$$T(N) = T(0) + N \Rightarrow N+1$$

(T(0) → 1)

TC: O(N)

⇒ Fibonacci Series.

Golden Ratio

N:	1	2	3	4	5	6	7	8	9	....
fib:	1	1	2	3	5	8	13	21	34	

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

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

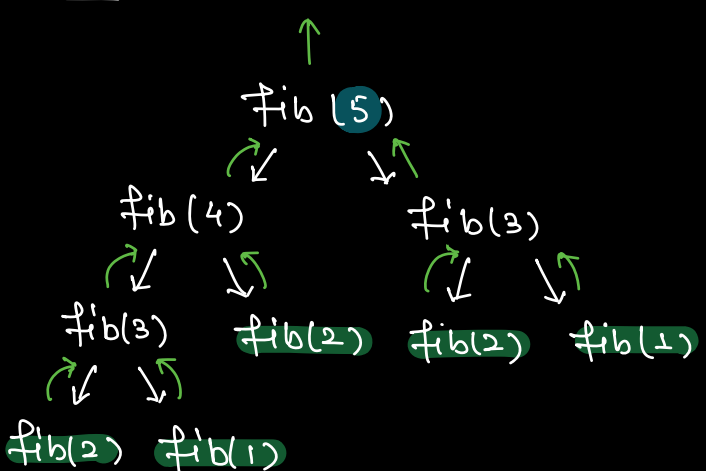
Exit Cond^

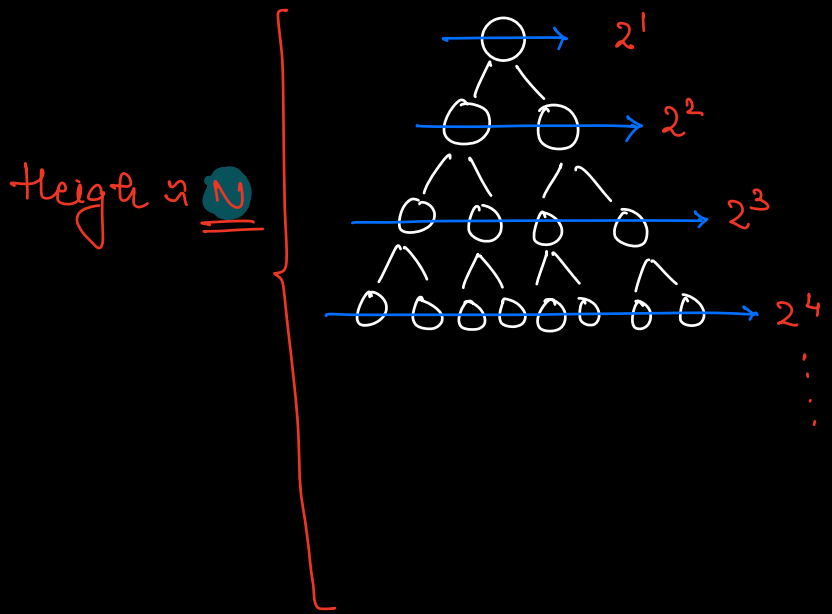
N = 1

fib(1) → fib(0) + fib(-1)
×

fib(2) → fib(1) + fib(0)
×

Recursion Tree





TC : No. of fun^ calls × TC of each fun^ call.

No. of fun^ calls $= 2^1 + 2^2 + 2^3 + \ldots\ldots 2^N$

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

$$= 2^{N+1} - 1$$

TC : $(2^{N+1} - 1) \times O(1)$

: $\boxed{O(2^N)}$ (Exponential TC)

→ N = 10 → ≈ 1000

SC : Height of the recursion Tree → O(N)

Recurrence Relation

HW

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

upper bound

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

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

$2T(N-1)$

$2T(n-2)+1$

$2T(n-3)+1$

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

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

$$= 4T(N-2) + 3$$

$$= 4(2T(N-3)+1) + 3$$

$$= 8T(N-3) + 7$$

$$= 8(2T(N-4) + 1) + 7$$

$$= 16T(N-4) + 15$$

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

$N-K = 0 \Rightarrow K = N$

$$= 2^N \cdot T(0) + 2^N - 1$$

$T(0) \rightarrow 1$

$$T(N) = 2 \cdot 2^N - 1$$

TC: $O(2^N)$

HW: Implement `power(a, n)` fun^

Q. Towers of Hanoi

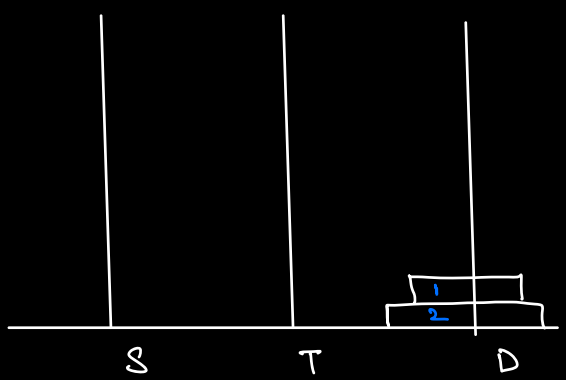
→ 3 towers `src`, `dest`, `temp`

→ There are N disks placed on src

→ Move all the disks from `src` to `dest` using `temp`

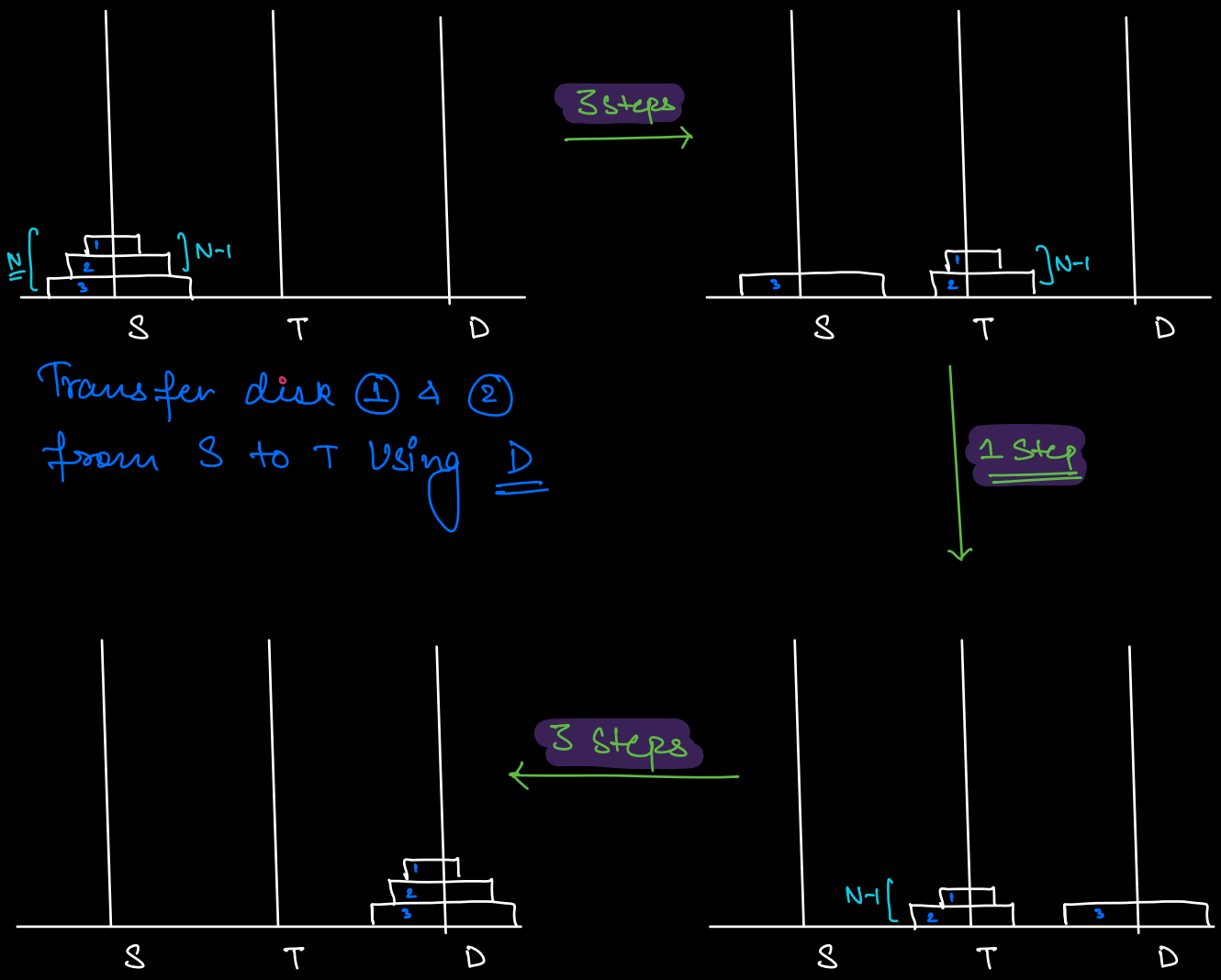
→ We can move only 1 disk at a time

→ We can't place a larger disk on a smaller disk.



1) S → T
2) S → D
3) T → D

} 3 steps



Transfer disk ① & ② from S to T using D

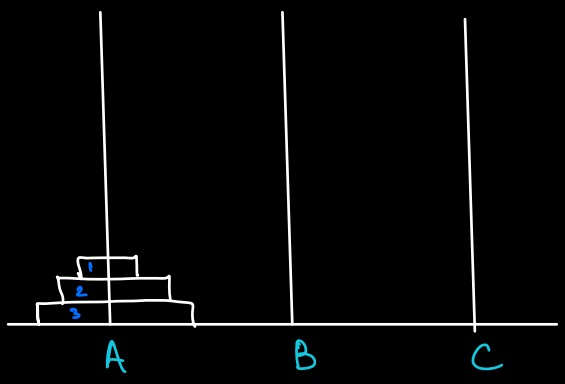
Transfer 2 disks from (T) to (D) using (S) tower. (Subproblem)

# of steps = 7

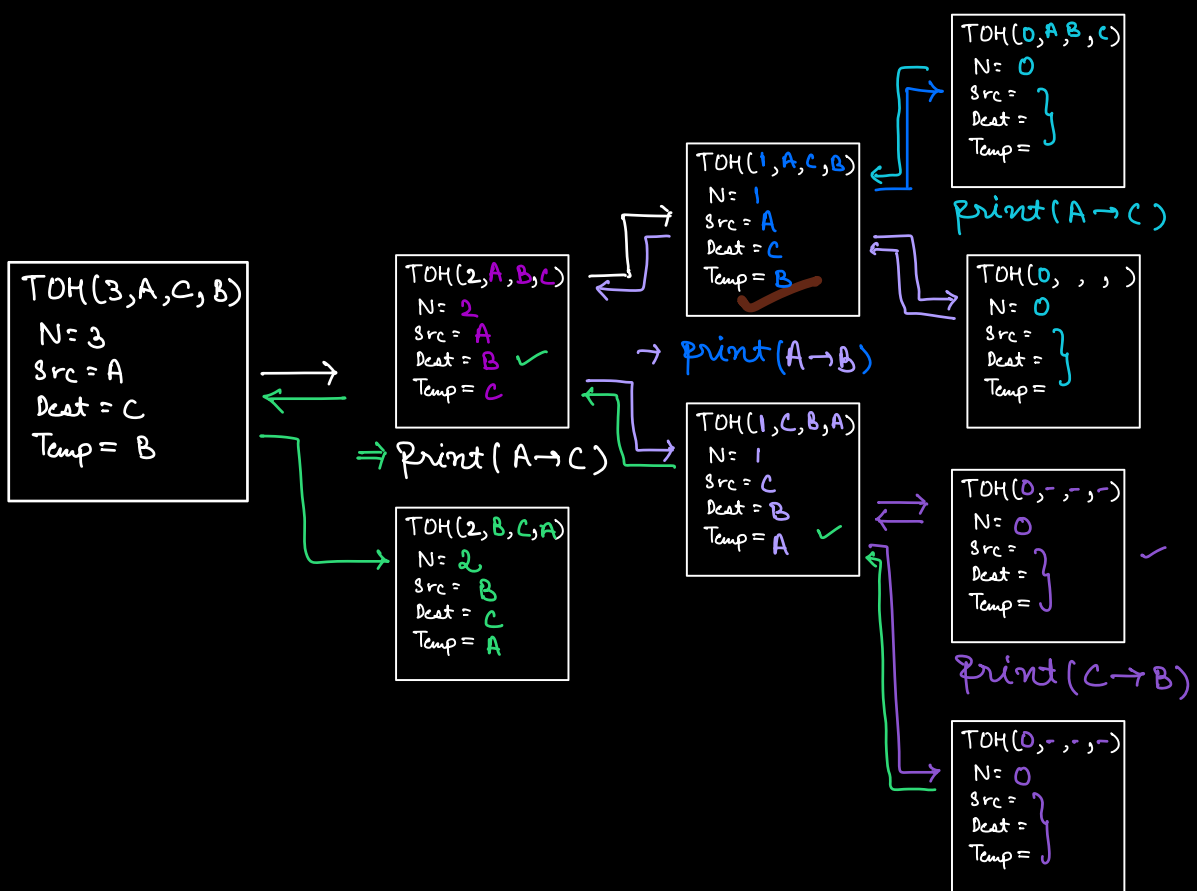
TOH (N, src, dest, temp) ⇒ Print the steps to shift N disks from tower src to dest using temp.

1) S → D
2) S → T
3) D → T
4) S → D
5) T → S
6) T → D
7) S → D

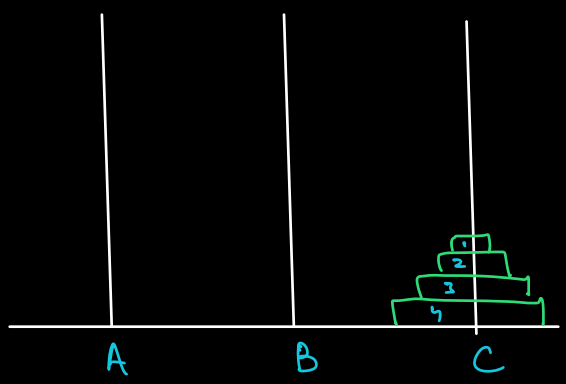
```
void TOH(N, src, dest, temp) {
    if (N == 0)
        return;
    // Assumption: TOH(x, A, B, C) transfers (n)
    // disks from A to B using C as a helper
    // tower.
    TOH(N-1, src, temp, dest);
    print(Src ⟶ Dest);
    TOH(N-1, temp, dest, src);
}
```



Transfer 3 disks from A to C using tower B.



A → C
A → B
C → B
A → C

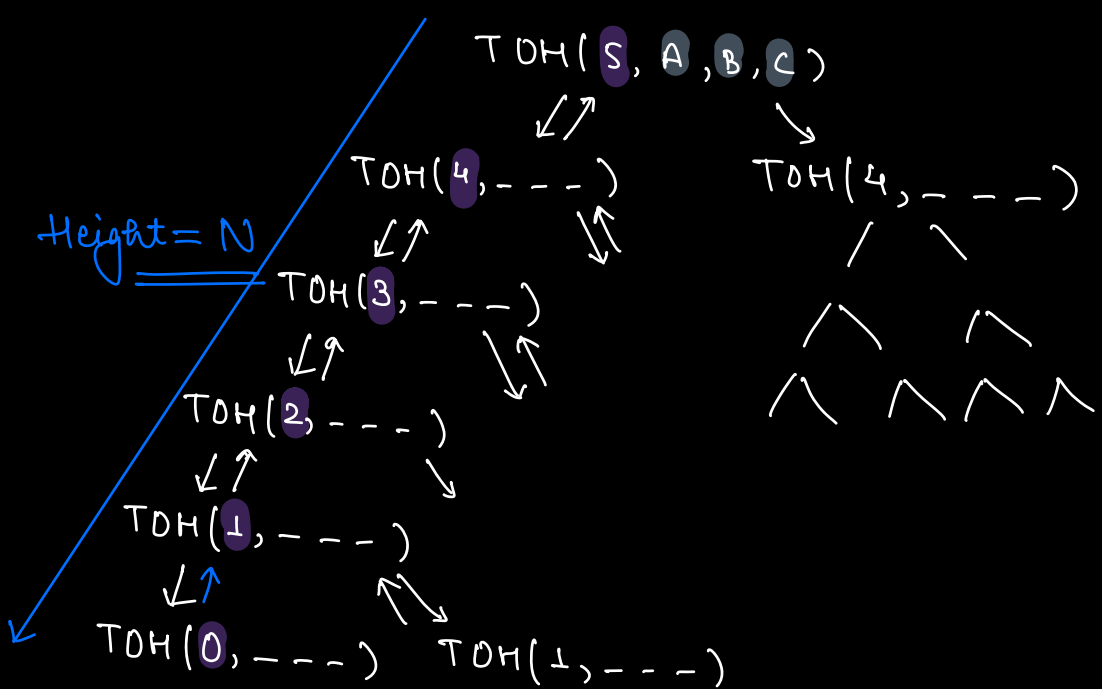


→ 7 steps
(Transfer 3 disks from A to B using C)

→ 1 step
(Transfer disk from A to C)

→ 7 steps
(Transfer 3 disks from B to C using A)

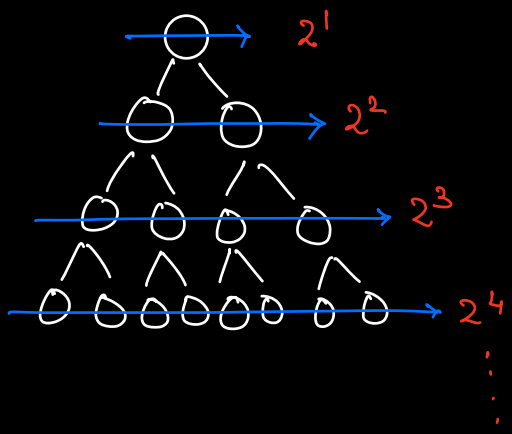
TC:



TC: # of recursive calls * TC of each func call.

: $2^N * 1$

: $O(2^N)$



Space Complexity

SC : O(N)

→ 1sec ≈ 10^8 iterations.