hashtable / map

Doubt

- get() Practically O(1) for most cases (99%)

→ worst case O(N) but the prob of having worst is very less

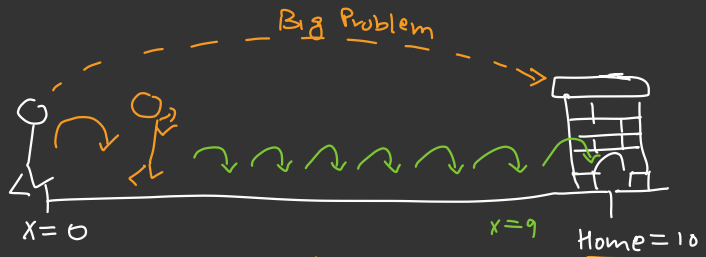
only understand once we cover how to build your hash-table

↳ Separate Chaining / Probing

Advanced DSA

Recursive Functions / Recursion - I

Recursion → Technique in CS / programming where we can define solution to **big problem (N)** in terms of smaller problems of the **same type**



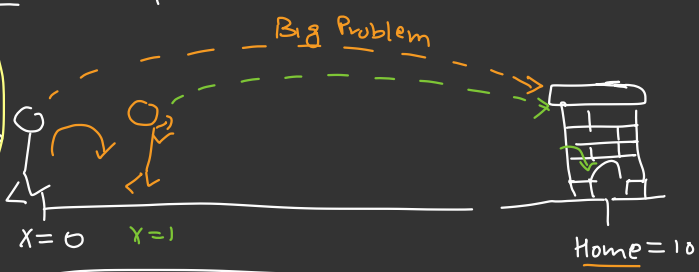

```
goToHome( x , Home ){
        src   dest

    for( , x < Home; ){        x = 9
        x = x + 1
    }
    Print ("Reached Home" + x)
}
```

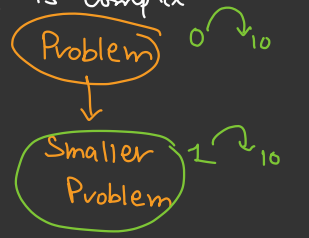
// goToHome(0, 10), function call

iterative code → Better than Recursion → Always takes less space

→ Recursion is helpful in problems where writing loops is complex

- MergeSort
- Quicksort
- ToH
- Backtracking
- Sudoku Solver
- :





0 → 10
↓
1 → 10
↓
2 → 10
↓
3 → 10
↓
4 → 0
⋮
8 → 10
↓
9 → 10
↓
10 → 10 ← Stop when problem is solved
↓
Void

```
→ goToHome ( X , Home ) {
    if ( x == Home ) {
        return ;
    }
    → goToHome ( X+1 , Home )
}
```

Base Case

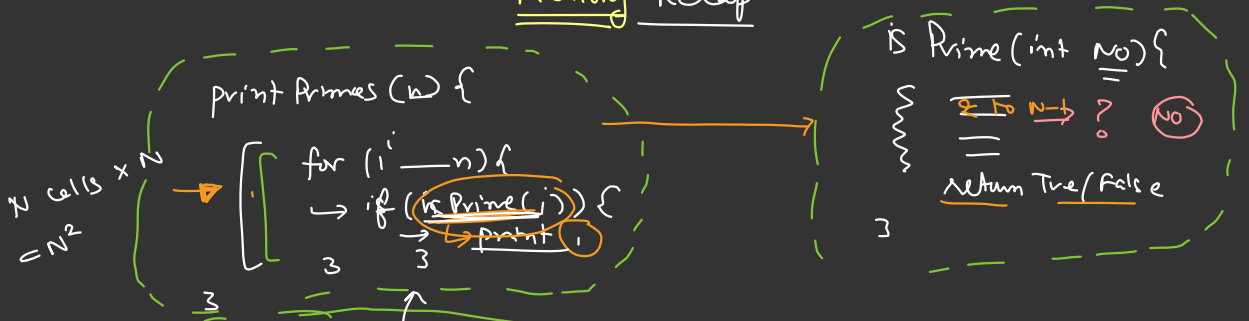
Recursive Case

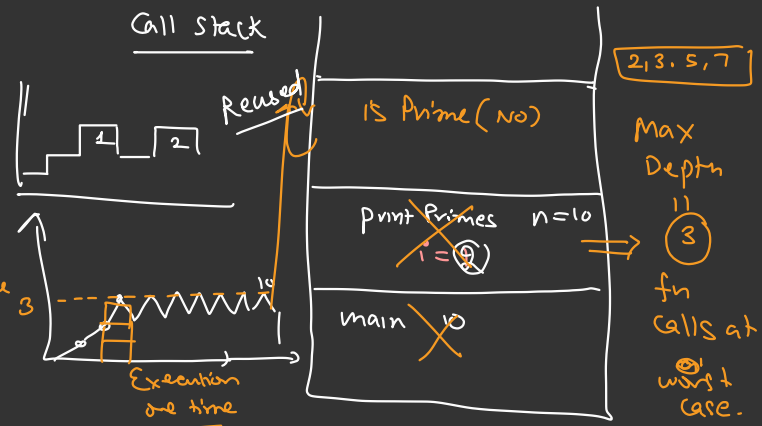
Recursive Relation

$$f(n) = f(n')$$

$n' < n$

Memory Recap





- ↑ Total fn calls → 1 Main + 1 Print Primes + ↑ Check Primes (10) (N)
  - = N fn calls
  - T will depend on no of calls
  - x work done in each call.

10 times

$= N * N$

$= O(N^2)$

- Space → $O(1)$

  Max Extra Space utilised during the execution of the program

  ↳ Space from call stack (stack memory) + any additional data structures which depend upon N.

  → Should be included!!

Q1 Recursive Code to print all Numbers from 1 to N.

N = 5 output - 1, 2, 3, 4, 5

```
printNumbers ( N ) {

}
```

Rules - to write Rec. Code

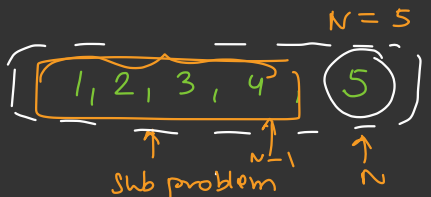
Base Case

1. Think about the smallest case

   [n == 1 → print 1, return

   OR

   ↳ [n == 0 → return



Assume Subsol is Known

① Assume the smaller subproblem can be solved

f(n) — many ways

f(n-1) f(n-2) --- f(n/2) f(n/3) — ans can computed "automatically"

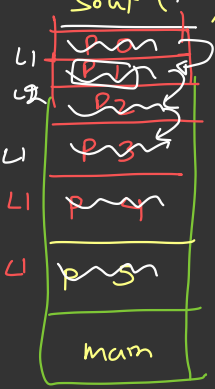
⇒ print Fn (N) →

if (N == 0) → return

L1 → print Fn (N-1) // Rec Call

L2 → Sout (N)

f(n)

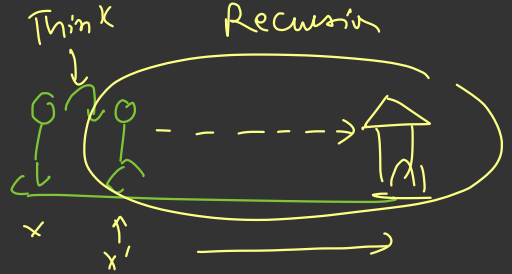


1, 2, 3, 4, 5

Print Fn(5)

③ Recursive Case

Using Assuming, express f(n) in terms of f(n') where n' < n.



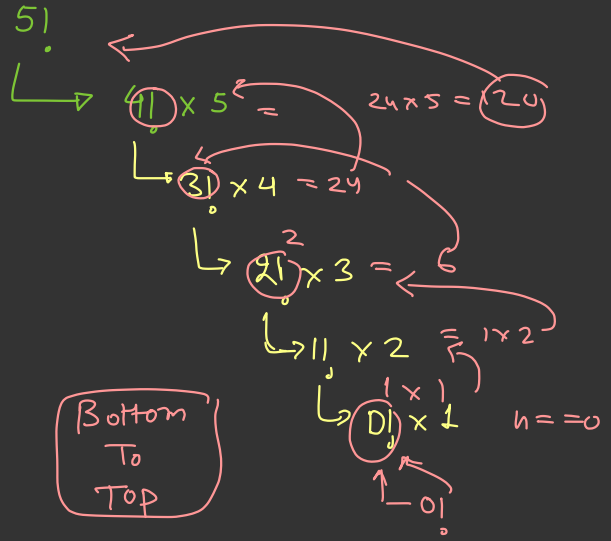
Factorial N →

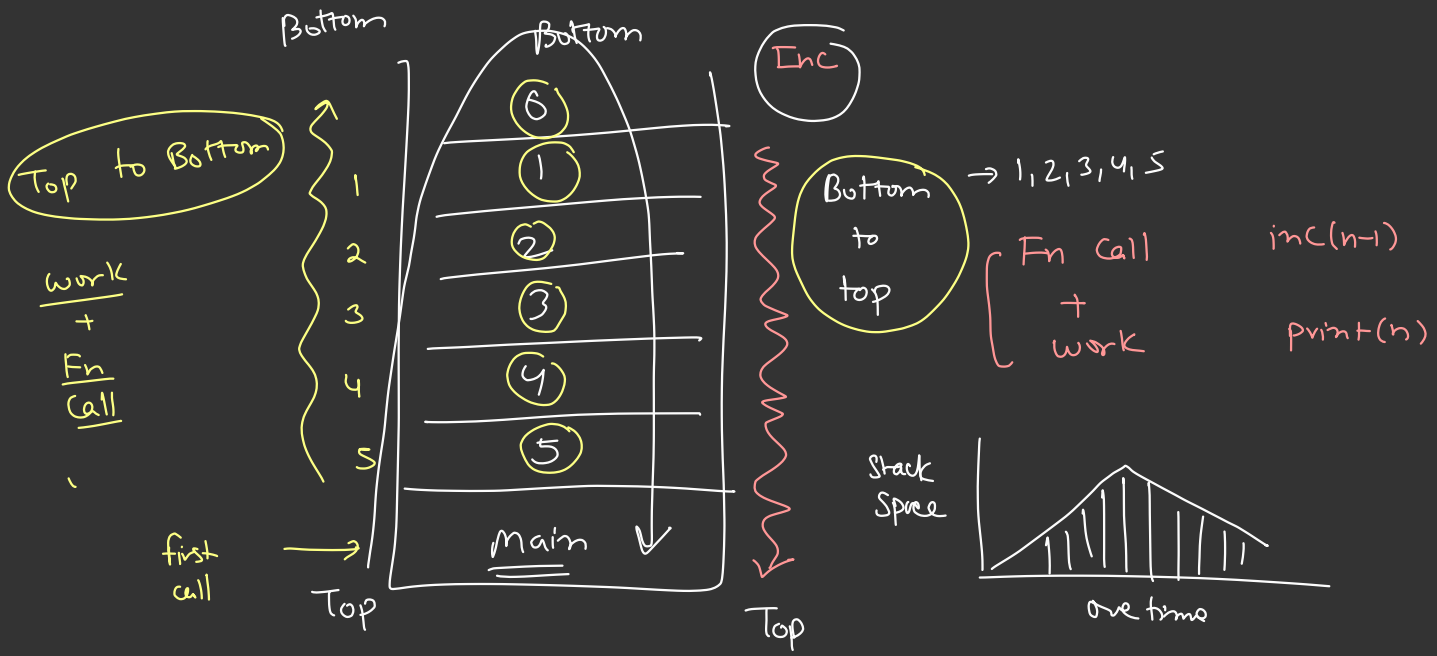
```
fact (N) {
    // Base case
    if (n==0) {
        return 1.
    }
    // Rec case
    return fact(N-1) * N
}
```

3

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

$N! = (N-1)! \times N$ ← Rec. Case




Bottom
Bottom
Inc
Top to Bottom
work
+
Fn
Call
1
2
3
4
5
6
1
2
3
4
5
Bottom
to
top
→ 1, 2, 3, 4, 5
Fn Call
+
work
inc(n-1)
print(n)
first
call
Top
main
Top
Stack
Space
one time

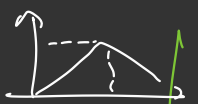
```
int fact (N) {
    // Base case
    if (n==0) {
        return 1;
    }
    // Rec case
    return fact(N-1) * N
}
```

Base case

3

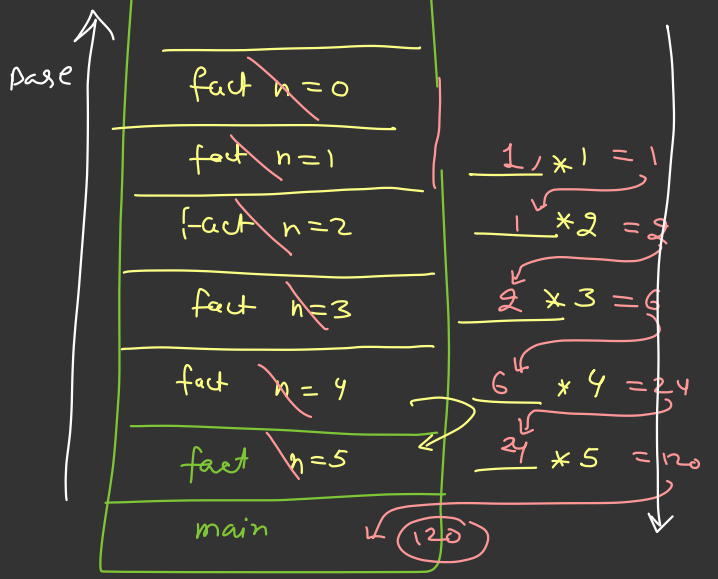
```
main() {
    fact(5)
}
```

3



Time → No of calls × Each call = O(N)

Space → Max space Utilised at any point = O(N)



Doubt

fun() { ... prints ... return }
No Rec Call

main() {

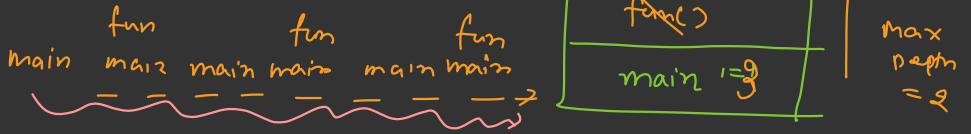
for (i=1 ——→ N) {

fun(), ⇒ K

}

}

10.45



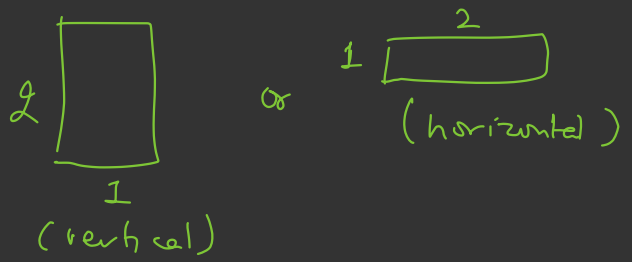
Space Complexity → O(1) → Max Depth of Call Stack

Time Complexity → O(N) → Total Calls done

Tiling Problem

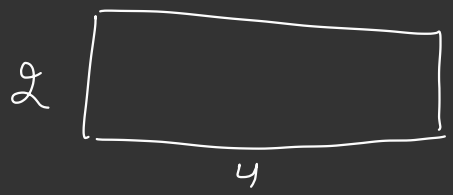
Given a Floor of Size $N \times 2$

Tiles →

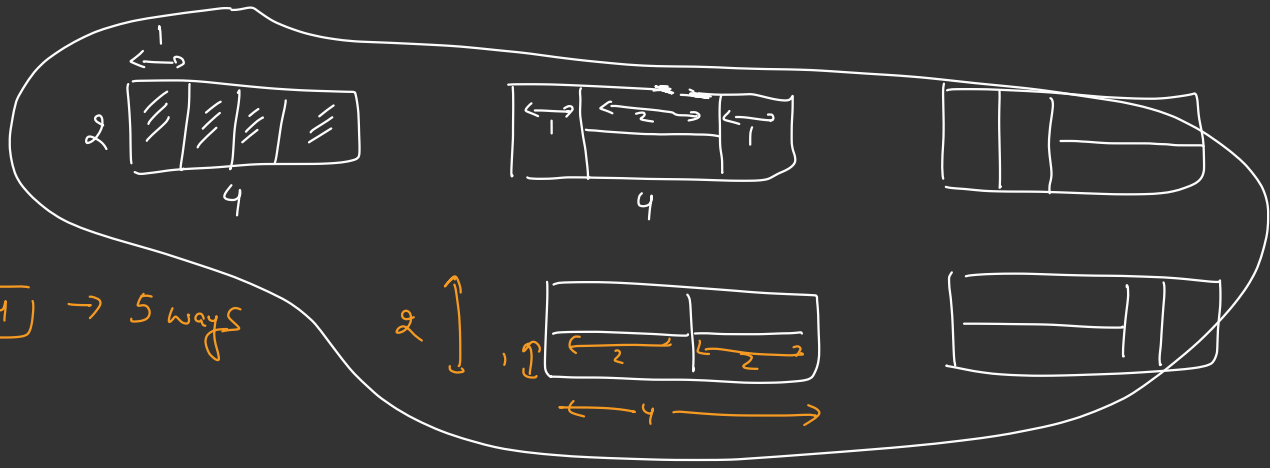


No of ways to design the floor.

$N = 4$

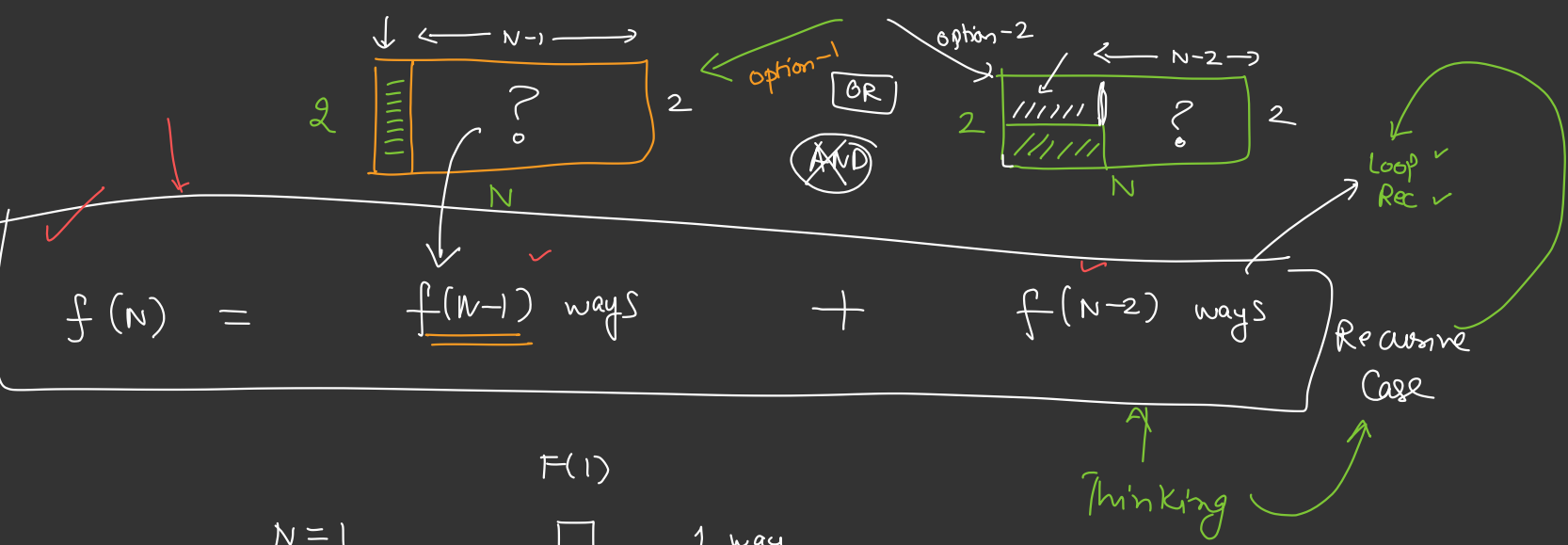


output



$f(N)$ = Do work for 1 Tile / Place 1 Tile + Recursion do work for Rest

↑ No of ways to fill a floor of len N.



F(1)

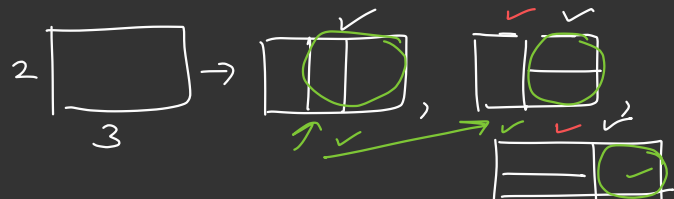
N = 1

1 way

N = 2

2 ways

N = 3



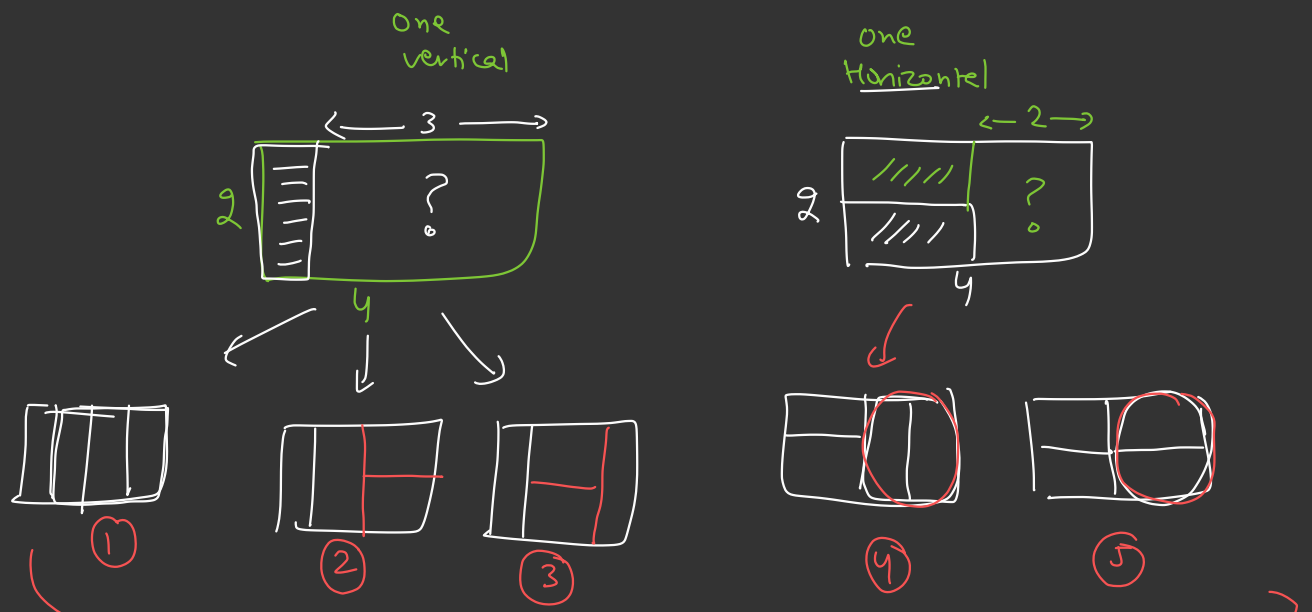
$f(3) = f(2) + f(1)$

$= 3$

$N = 4$

$f(4) = f(3) + f(2)$
$= 3 + 2$
$= 5$ ways

$f(4) = f(3) + f(2)$

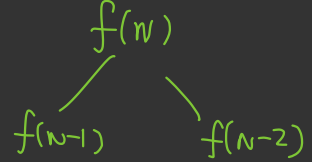


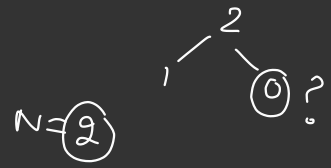
TODO // $f(5) = f(4) + f(3) = 5 + 3 = 8$ ways

Tiling Problem

Complicated

```
int f(n){
    if(N==1)
        return 1;
    if(N==2)
        return 2;
    return f(n-1) + f(n-2);
}
```





1, 2, 3, 5, 8, 13, ------

Base Case

Fibonacci Series

Base case → `if(n==0 or n==1){ return n;`

0, 1, 1, 2, 3, 5, 8, 13, ------

n=0 n=1 n=2 n=3 ... f(n) = f(n-1) + f(n-2)

[Loop is Better than Rec. Always.

Time → $O(2^N)$ very Bad if you Recursion

$O(N)$ good. —— Loop. // code → toDo