Big O Notation

- Big O notation is a way to how measure how fast or slow a program is when it work with data.

- It tells us how much time or memory a program will take as we give it more and more things to do.

- It's like saying, "How does this task grow when there are more toys to pick up?

Think about this

- If you have 1 toy to pick-up, it's super fast.
- If you have 10 toys, it takes a bit longer.
- If you have 100 toys, it takes even longer.

Big O is a way to describe this growth.

Code 1 code 2

Decide krte hai konsa better hai

• faster

• memory

Suppose code 1 => 5 sec lagaye aur code 2 ne 15 sec

- Time complexity depend krti hai no of operation pr

=> Why we don't measure time complexity with time?

remember fast and slow computer

~~Same computer~~

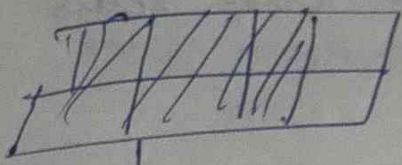
Same code

Diff speed due to fast and slow.

- Space complexity

Code 1

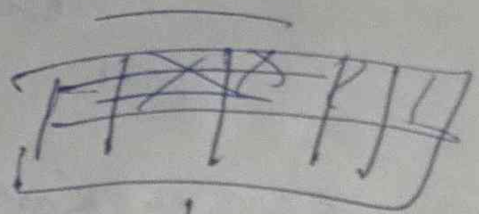
5 sec



jyada space

Code 2

15-sec



kam space

→ No of operations

Sum of n numbers

function addupto(n):

return n * (n + 1) / 2

1 op 2 op 3 op

3 operation ho raha hai ye hi no of operation hai

→ Notations

↳ Ω (omega) → Best case

↳ Θ (theta) → average case

↳ O (oh) → worst case

Find numbers

1 2 3 4 5 6 7 8 9 10 11

best case average case worst case

Big o notation ko hum $O(f(n))$ se denote karte hai

Type

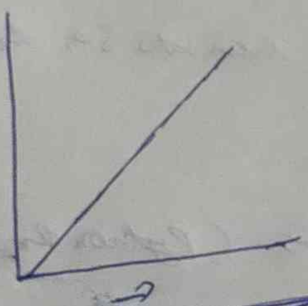
- $O(n)$: linear time

 · It's like picking up toys one by one. If there are 10 toys, you pick up 10.

```
def LinearLoop(n):
    for i in range (0, n)
        print(i)

linearLoop(11)
```

no. of operation



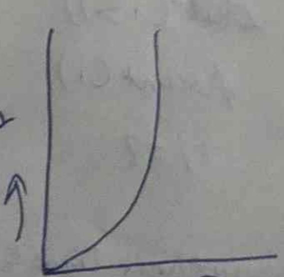
$\vec{n}$

⇒ Jitne n utne hi operation

- $O(n^2)$ ⇒ Quadratic time
  → is like doing a job where you have to do everything twice, or check everything which makes the job take much longer as the number thing grows

```
def quadraticLoop(n):
    for i in range(n):
        for j in range(n):
            ~~print (~~i) print(i, j)

quadraticLoop(10)
```



operation

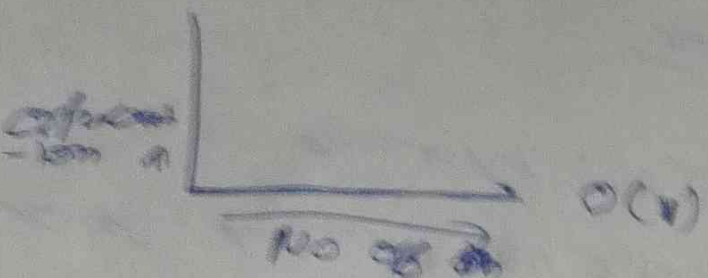
no of $\vec{n}$

operation (2x) ho gay

O(1) = Constant time

```
def example(n):
    return n+n

print(example(2))
```



=> It's like looking at just one toy, no matter how many toys are there - Super fast!

Example: Checking if a light is on or off

4) $O(\log n)$: Logarithmic Time

- It's like dividing your toys into groups and only looking at one group at a time.

=> If there 100 toys, you split them into 50, then 25, then 12 ... much faster

Example Binary search (Python example here)

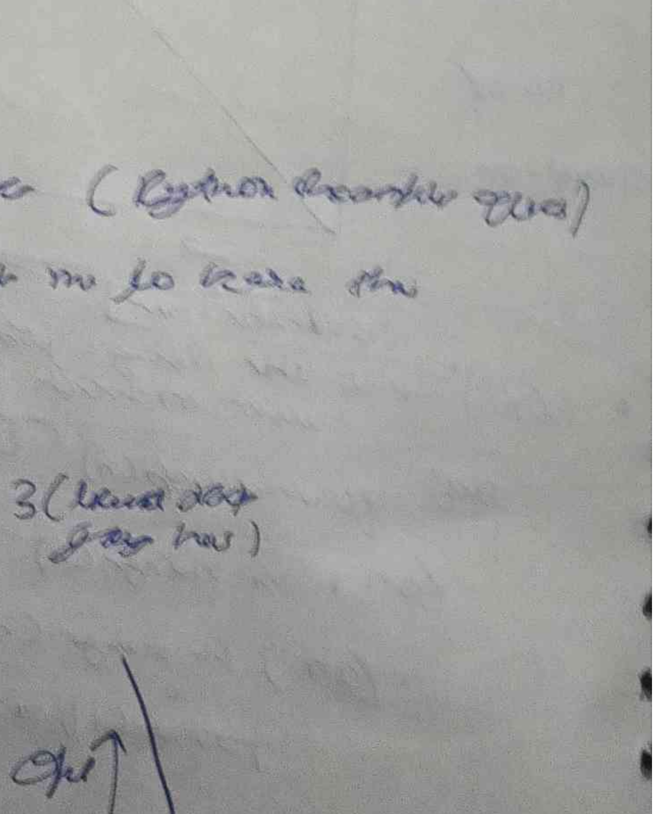
↳ US and me to learn this

```
=> def logarithmic_time_example(n):
    i = n
    while(i > 1):
        print(i)
        i //= 2
```

$2^3 = 8 \Rightarrow \log_2 8 = 3$ (how deep you go)

logarithmic_time_example(8)

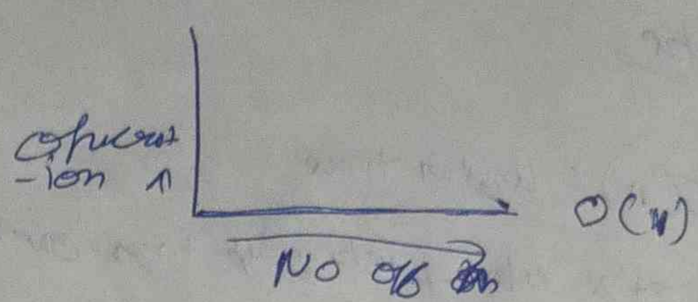
output: 8
4
2



O(1) : Constant time

```
def example(n):
    return n+n

print(example(2))
```



→ It's like Looking at just one toy, no matter how many toys are there. Super fast!

Example: Checking if a light is on or off

4). O(log n): Logarithmic Time

- It's like dividing your toys into groups and only looking at one group at a time.

→ If there're 100 toys, you split them into 50, then 25, then 12... much faster.

Example Binary tree (Python Example files)

↳ VS code me jo kara tha

```
→ def logarithmic_time_example(n):
    i = n
    while(i>1):
        print(i)
        i //= 2
```

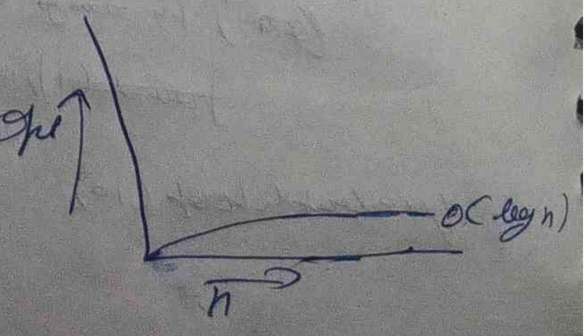
$2^3 = 8$

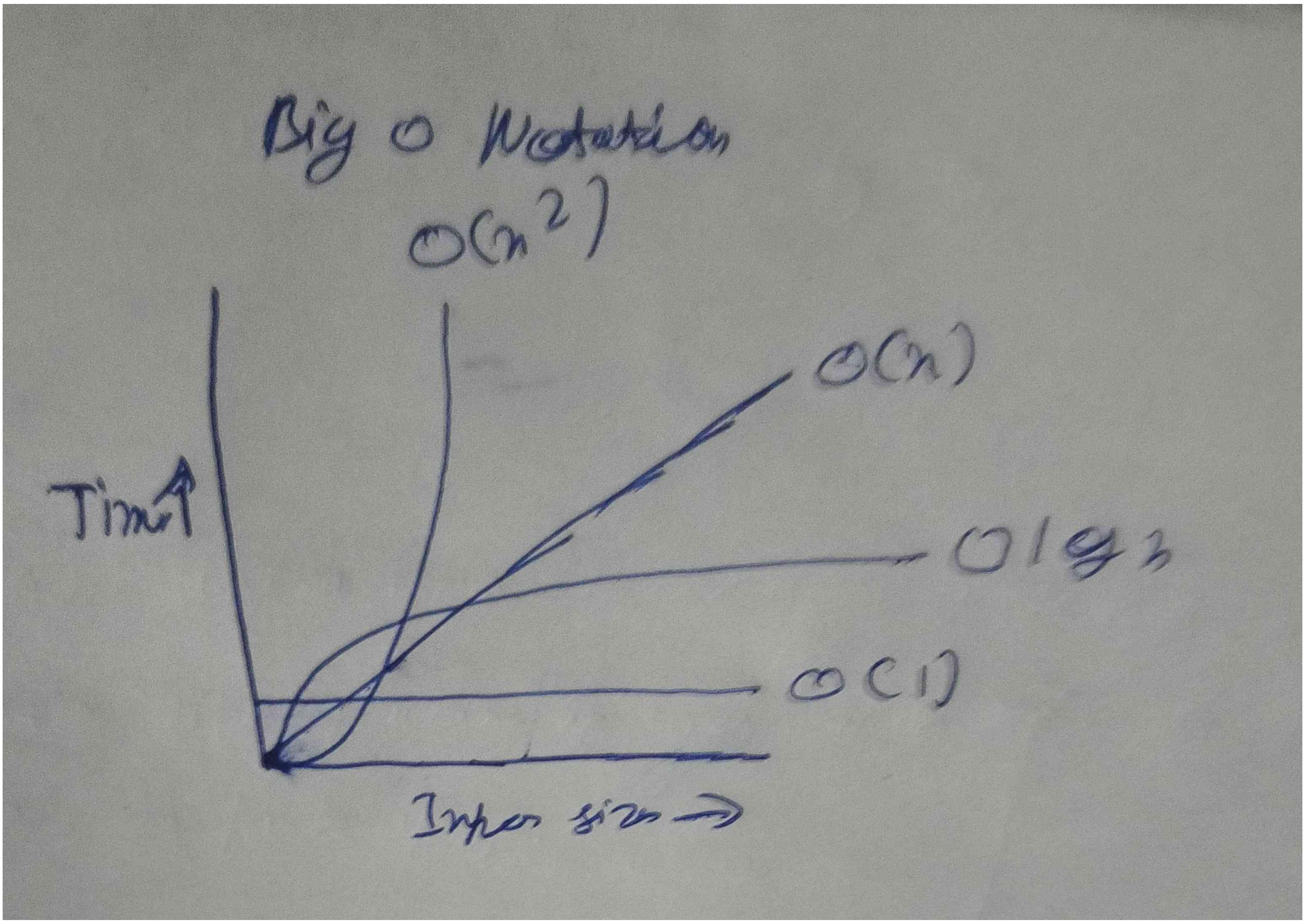
$\Downarrow$

$> \log_2 8 = 3$ ([illegible] gaya hai)

logarithmic_time_example(8)

output: 8
4
2




Big O Notation
O(n^2)
O(n)
O log n
O(1)
Time
Input size