Big O Notation (to measure time complexity)

Order of time complexity (lower to higher)

lower

1. 1
   ↓
2. $\log(n)$
   ↓
3. $\sqrt{n}$
   ↓
4. n
   ↓
5. $n \log n$
   ↓
6. n^2
   ↓
7. 2^n
   ↓
8. $!n$

Higher

ex (I) for $n = 4$, $\log(n)$ or $\log_2 4 = 2$

$\log(n)$ half of 'n'

(II) Rules for finding time complexity

O(n)

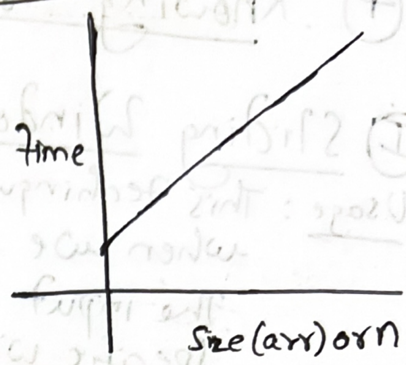
time = $a * n + b$

↓

★ 1. Keep fastest growing term

↓

time = $a * n$



★ 2. Drop constants. i.e. (drop 'a'. so we are left with 'n' only)

↓

time = $O(n)$

So, the Big (O) for time = $a * n + b$ is order of 'n'.

Other examples of O(n)

```
def get_squared_num(nums):
    squared_nums = [ ]
    for n in nums:
        squared_nums.append(n*n)
    return squared_nums
```

Big(O)
↓
$\boxed{O(n)}$

Console:

```
nums = [2, 5, 8, 9]
get_squared_nums(nums)
# return [4, 25, 64, 81]
```

Reason for O(n)

→ Because the time it takes for iterating this program is proportional to no. of computation it is doing.

→ let say the input array size is 4 it will do 4 iterations or if the array ... 4 million ... it will take 4 million

(II) O(1) Complexity

eg:

size(arr) → 100 → 0.22 milliseconds.

size(arr) → 1000 → 0.23 milliseconds.

Here, for size(arr) function the order of complexity is O(1) as when increase the size of input the time is almost the same.

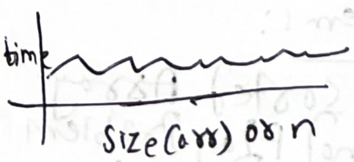
time = a

↓

1. Keep fastest growing term
2. Drop constants. a → 1

↓

time = O(1)



III $O(n^2)$

Ex:
```
nums = [3,6,2,4,3,6,8,9]
for i in range(len(nums)):
    for j in range(i+1, len(nums))
        if nums[i] == nums[j]:
            print(nums[i] + " is a duplicate")
            break.
```

$O(n^2)$
⇓ Reason

Here, we are running $\boxed{\text{two for loops}}$ & comparing nums to find duplicate.

linear eq^n of time of above f^n. is: $\boxed{time = a * n^2 + b} \rightarrow O(n^2)$
after applying 2 rules → $O(n^2)$

Note 2 Block problem

```
nums = [3,6,2,4,3,6,8,9]
for i in range(len(nums)):
    for j in range(i+1, len(nums)):
        if nums[i] == nums[j]:
            duplicate = nums[i]
            break
```

⇓ Block-I → n^2 i
duplicate = None

Block-II → n iter

```
for i in range(len(nums)):
    if num(i) == duplicate:
        print(i)
```

For Block I & II represent $time = an^2 + bn + c$
⇓ apply 2 rules
$O(n^2)$

remember (log me base 2 hota he)

Twice of log n

IV Measuring space complexity

4	9	15	21	34	57	68	91

search for 68

ways of finding 68.

```
1. for i in range(len(nums)):
       if nums[i] == 68:
           print(i)
```

$\boxed{O(n)}$ ⇐

⇓ It might be good for less no's but let say if we want to apply it for million nums might be very time consuming.

THERE IS A BETTER WAY → BINARY SEARCH

Note
⇒ Iteration $k = n/2^k$ (to get the arr size to 1)

$1 = n/2^k$ (1 for worst case scenario)

$n = 2^k$

$\log_2 n = \log_2 2^k = k \log_2 2$
⇓ k

no. of iteration
⇓ $k = \log(n) \rightarrow O(\log n)$

Hence, the Complexity of Binary search is $\boxed{O(\log n)}$

$k = O(\log n) \Rightarrow \log_2(8)$
$\rightarrow \log_2(2^3)$
$\rightarrow 3 \log_2 2$
$\rightarrow 3$ iterations

BINARY SEARCH
Steps → First find middle element & compare with 68. if is less you discard the left side array.

Iteration 1 = $n/2$ (Middle element)

4	9	15	21	34	57	68	91

discard

Iteration 2 = $(n/2)/2 = n/2^2$ (again find middle element)

34	57	68	91

discard

Iteration 3 = $(n/2^2)/2 = n/2^3$ i.e $8/8 = 1$

68	91

Conclusion: Using Binary Search we found ans in 3 Iterations instead of n which is 7 in above ex.

for k ⇓
Iteration k = $(n/2^k)$

example of $O(1)$

```
def find_first_pe(prices, eps, index):
    pe = prices[index]/eps[index]
    return pe
```

Reason of $O(1)$: pe function is a constant function as doesn't matter whatever index we pass the time execution will going to remain constant hence $O(1)$.