Time Complexity & Big O notation

This morning I wanted to eat some pizzas; so I asked my brother to get me some from Dominos (3km far)

He got me the pizza and I was happy only to realize it was too less for 29 friends who come to my house for a surprise visit!

My brother can get 2 pizzas for me on his bike but pizza for 29 friends is to huge of an input for him which he cannot handle

What is Time Complexity?
Time Complexity is the study of efficiency of algorithms

Time Complexity = How time taken to execute an algorithm grows with the size of the input

Consider two developers who created a algorithm to sort n numbers. Shubham & Rohan did this independently

when ran for input size n, following results where recorded.

no of elements (n)	Shubham's Algo	Rohan's Algo
10 elements	90ms	122 ms
70 elements	110ms	124 ms
110 elements	180ms	131 ms
1000 elements	2s	800ms

Quick Quiz :- who's Algorithm is Better ?

⇒ If we See for Smaller no of elements shubham's Algo is better and for more elements Rohan's Algo was Effective but overall Rohan's Algo was Effective

Time Complexity : Sending GTA V to a friend

let us Say you have a friend living 5 kms away from your place you want to send him a game

final Scams are over and you want him to get this 60GB file from you. How will you send it to him?

Note (That both of you are using (JIO) 4G with 1Gb/day data limit)

The best way to send him the game is by delivering it to his home.

Copy the game to a Hard disk and send it!

Will yo do the Same thing for Sending a game like minesweeper which is Kbs of Size?

No because you can send it via internet

(*) As the file size grows, time taken by Online Sending increases linearly $\longrightarrow O(n^1)$

(*) As the file size grows time taken by physical sending remains constant $O(n^0)$ or $O(1)$

Calculating Order in terms of input size

In order to calculate the order, most impactful term containing n is taken into account ↳ size of input

Let us assume that formula of an algorithm in terms of input size n looks like this:

Algo 1 $\rightarrow \underbrace{K_1 n^2}_{\text{Highest order term}} + \underbrace{K_2 n + 36}_{\text{Can ignore lower order terms}} \Rightarrow O(n^2)$

Algo 2 $\rightarrow K_1 K_2^2 + K_3 K_2 + 8$

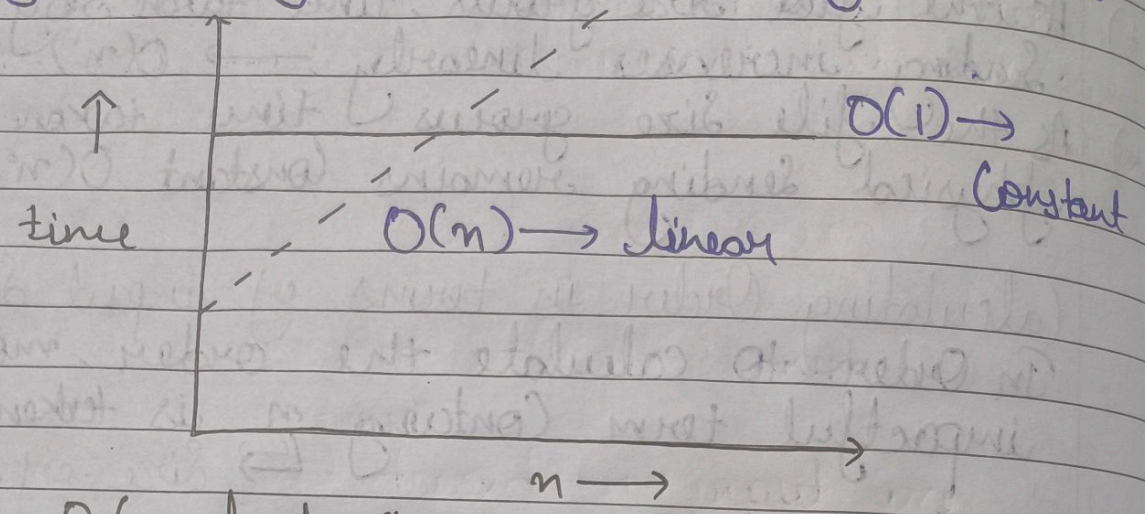
$\Downarrow$

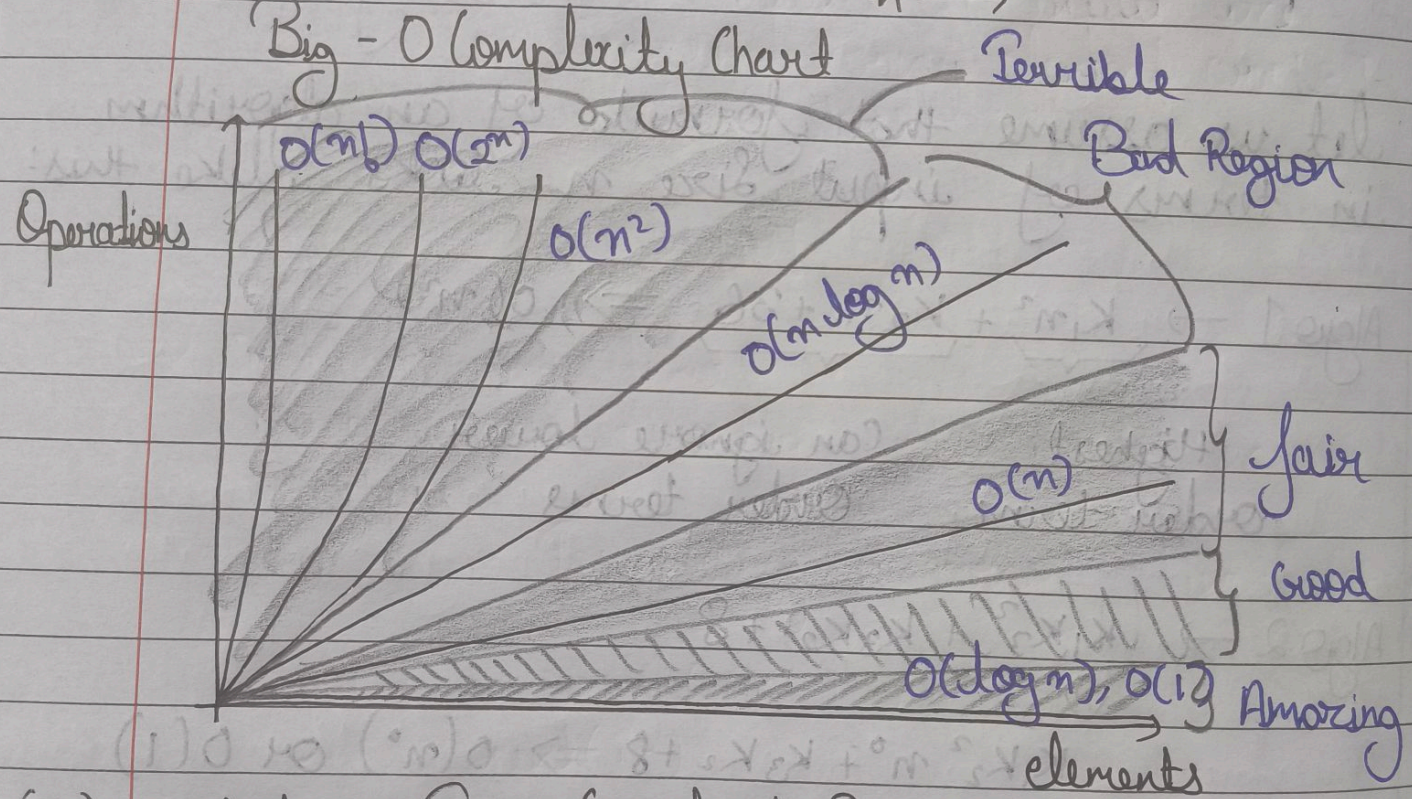
$K_1 K_2^2 n^0 + K_3 K_2 + 8 \Rightarrow O(n^0)$ or $O(1)$

Note that these are the formulas for time taken by them

Visualising Big O

If we were to plot O(1) and O(n) on a graph they will look something like this





(*) What is Time Complexity?

Time Complexity describes how the running Time of an algorithm grows as the input Size increases

It is Not the actual time in milliseconds
It is the rate of growth of time with respect to input Size (n)

Why we need it
- Different machines → different speeds
- Different languages → different Execution times
- Same algorithm → behaves differently for small vs large input

So we measure growth not seconds

(*) Input Size (n)
n ⇒ Size of input
Example
Array of length 100 → (n=100)
String of 50 character → (n=50)
Matrix 10×10 → (n=100)

(*) Big-O Notation
Big-O gives an upper bound on time growth
It answers
: what is the worst-case growth when input becomes very large?

we ignore :- Constants (×2, ×10) - lower-order terms
Because growth dominates Everything

(*) Best, worst & Average Case
Best Case
faster possible input
worst Case (Most important)
slowest possible input

Average Case
Expected time for random input

Example: linear Search

Best Case → O(1) (element at first index)
worst Case → O(n) (element at last index / not present)
Average Case → O(n)

(*) DSA mostly cares about worst case

(*) Common Time Complexities (with Meaning)

⇒ O(1) – Constant time
Time does not depend on input size
Example :- Accessing array element arr[5] - push/pop from stack

x = arr[10]

⇒ O(n) – linear time
Time grows directly with input size
Example :-
Traversing an array - linear Search

```
for i in range(n):
    print(i)
```

⇒ O(n²) – Quadratic time
usually nested loops
Example :-
Bubble Sort - Comparing all pairs

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

⇒ O(logn) – Logarithmic Time
Input size reduces by half each step
Example :-
Binary Search

```
while low <= high
    mid = (low + high) // 2
```

very fast growth

⇒ O(n logn)
Combination of linear + log
Examples :-
Merge Sort - Quick Sort (average case)

⇒ $O(2^n)$ - Exponential Time
Every input doubles the work
Example :-
Recursive Fibonacci

```
def fib(n)
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

⇒ O(n!) - factorial time
All permutations
Example :-
Traveling Salesman (brute force)

(*) Rules to Calculate time Complexity

Rule 1:- Ignore Constants

$O(2n) \rightarrow O(n)$

$O(100) \rightarrow O(1)$

Rule 2:- Drops lower-Order terms

$O(n^2+n+1) \rightarrow O(n^2)$

Rule 3: loops

- One loop $\rightarrow O(n)$
- Nested loop $\rightarrow O(n^2)$

Rule 4: Consecutive loops (Add)

```
for i in range(n): pass
for j in range(n): pass
```

$\rightarrow O(n+n) \rightarrow O(n)$

Rule 5: Nested loops (Multiply)

```
for i in range(n):
    for j in range(n):
        pass
```

$\rightarrow O(n*n) \rightarrow O(n^2)$

Rule 6: Halving input

```
n = n//2
```
→ O(log n)

*) Recursion & Time Complexity

Linear Recursion

```
def f(n)
    if n==0: return
    f(n-1)
```

→ O(n)

Binary Recursion

```
def f(n)
    if n==0: return
    f(n-1)
    f(n-1)
```

→ $O(2^n)$

*) Space vs Time (Brief Intro)

Time Complexity → Execution time
Space Complexity → Extra memory used

Examples:- Iterative Solution → less space
Recursive Solution → uses call stack

(Detailed notes later)

(*) Real-world Analogy Recap

- Email small file → O(n)
- Sending hard disk → O(1)

Growth matters, not actual speed.

(*) Interview Mindset
when asked Complexity
(1) Identify input size (n)
(2) count loops
(3) check nesting
(4) check recursion
(5) Give worst case

(*) Final Truth
Time Complexity is not math. It is pattern recognition
Once you understand growth, Big-O becomes obvious