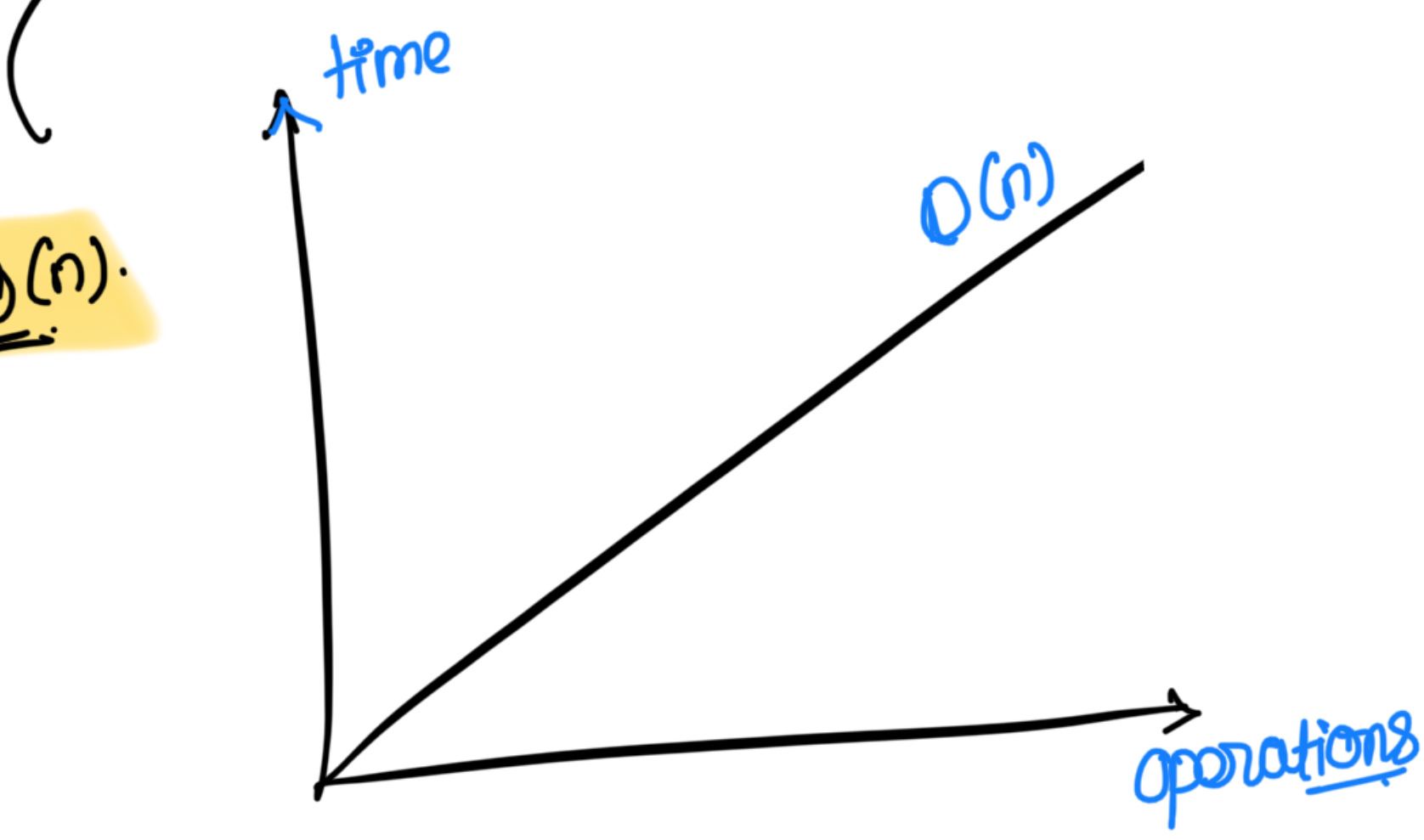
DSA

⇒ **Big O**

proportional.

① Big O(n).



Big O is always a straight line, that is the time & operations are proportional, so the more the operations, larger time it takes

Eg:

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

if 10 → 10 operations
20 → 20 ops.

O(n)

② Big O : Drop Constants.

Diff ways to simplify Big O, one of those is Drop Constants.

(eg):

```
def equation(n):
    for i in range(n):
        print(n)
    for j in range(n):
        print(n)
```

So every n values it runs n + n times.

= 2n

Drop constant is nothing but droping the constant from the

Big O

Big O

instead $2n$ ⇒ we write as n

it can be $3n$ or $100n$ or $50n$ → we drop Constant and take it as n

$O(n)$

③. $O(n^2)$

(eg):

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

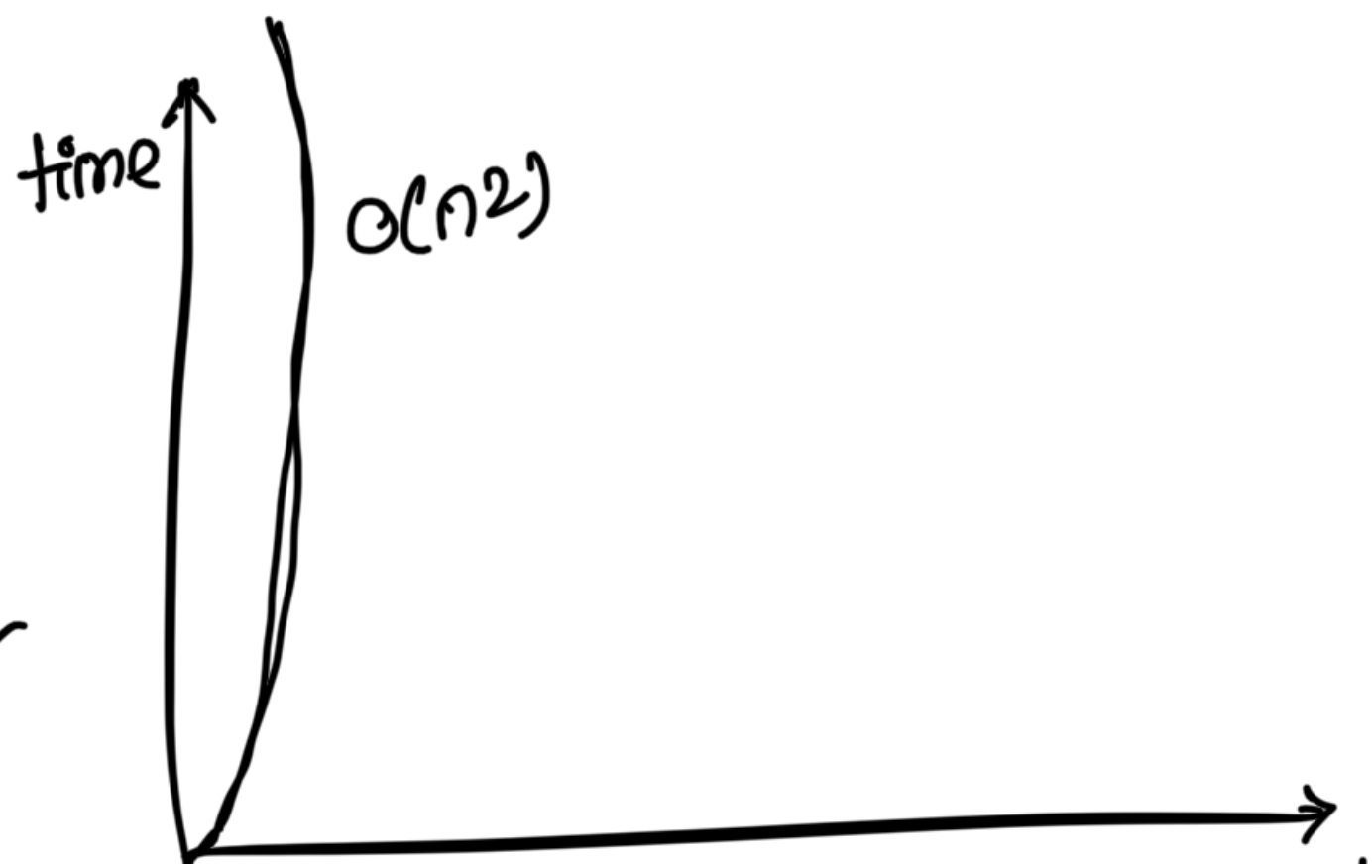
now for n
→ i run for 10 times
→ j run for 10 time for

every i

$O(n^2) \leftarrow n \times n \leftarrow 10 \times 10 = 100$

loop inside a loop will run for $O(n^2)$

graph looks like



the graph is

so steap,

operation.

means this is not efficient with time complexity

ie: it takes the most time to compile & run.

④ - Big O : Drop Non dominants

Another simplification method for Big O is Drop Non dominants.

(eg):

```
def print_nums(n):
```

2 run for

```
for i in range(n):
    for j in range(n):
        print (i,j)
```
$\Big\}\ O(n^2)$

```
for k in range(n):
    print(k)
```
$\Big\}\ O(n)$.

this become $O(n^2) + O(n)$

$O(n^2+n)$

↙ dominant

↳ less dominant / non dominant

(eg):

if n becomes 1k ⇒ but $O(n)$ will be

1k.

$O(n^2)$ will be

1 million ⇒ Comparing 1mil >>> 1k

n^2 ↙ dominant

n ↓ non dominant

So we drop non dominant $O(n^2+n)$

$O(n^2)$

this whole process is called dropping the non dominant

① $O(c)$ → constant time

(b). O(1) ⇒ Constant time.

(eg): def add_items (n):
return n+n

here whatever the value of n is,
↙
no. of. operations it takes is 1
↙
O(1)

if

return n+n+n ⇒ operation 3
↙
so O(3)

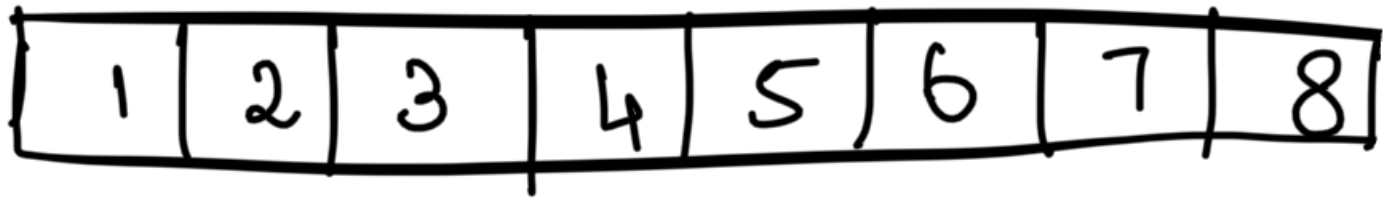
here 3 operation
Remains constant
even the n increases
so
↘ O(1)

even if the n increases, if the number of operations remained constant it is Big O(1).

note: O(1) is the most efficient Big O.

6. Big O (log n)

Ex (eg): we have a sorted list



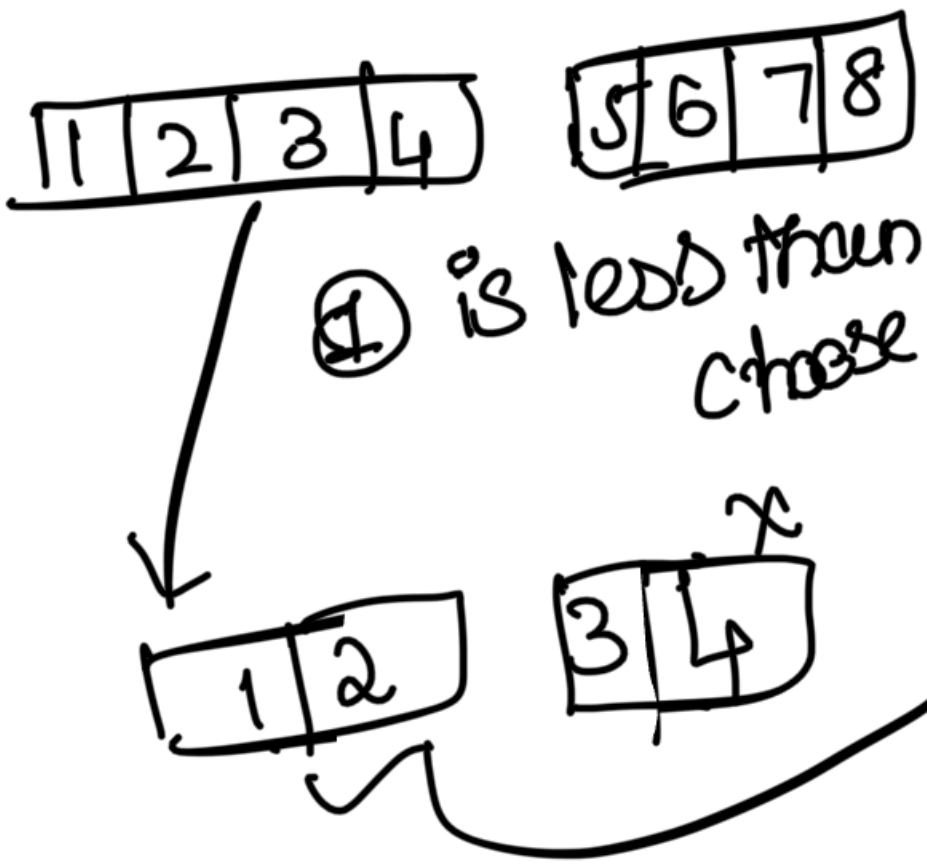
lets say we want find 1 in the above list.

instead iterating through,

we cut the list into half and see val is

in which list, and leave the other.

now in our case



it took us 3 operations to find the value

ie

$2^3 = \text{len}(arr)$

or

$\log_2 \text{len}\, arr = 3$

log is even more faster ✓

this is O(1)

efficient when the value gets larger.

this is nothing but how many times to cut to get to the one item (not first item but to one item)

→ ie 2|8 ⇒ 3
2|4
2

allocate is 2|n

(eg): lets take more than billion:

1,073,741,824 ⇒ 2|1,073,741,824

(31).

It takes just 31 sec for a billion

which is not even 1%.

(if O(n) it takes billion opperation &
for $O(n^2)$ its more than that.

note: O(log n) is nearly flat as O(1), and very efficient.

7. Big O(n log n)

↳ used in some Sorting Algorithms. in

(merge sort and quick sort)

⇓

note: if we gonna sort data like string,

this is the most efficient sort we can make.

8. Big O : Different terms for inputs.

↳ asked in interviews.

Code:

eg):

```
def print_val(n):
    for val in range(n):
        print(n)
    for val2 in range(n):
        print(n)
```

here we have only one parameter

n

so we can drop constants and can make it from O(2) to O(n)

eg):

```
def print_lines(a,b):
    for i in range(a):
        print(i)
    for i in range(b):
        print(i)
```

here we can't say $O(a) \neq O(b)$

coz a value could be larger or smaller than b

print (j)

it can be written as

$O(a+b)$ ⟸ $O(a) + O(b)$

eg):

```
def print_lines(a,b):
    for i in range(a):
        for j in range(b):
            print(i,j)
```

here it is ⇒ $O(a*b)$

9. Big O : lists.

lets take a list

$l1 = [11, 3, 23, 7]$
0 3

$l1.append(66) \Rightarrow l_1 = [11, 3, 23, 7, 66]$
0 4

↳ here it does only one operation.

$l_1.pop \Rightarrow l_1 = [11, 3, 23, 7]$

↳ again all we did is one operation.

there is no
... so ... (i) ... only one

re-indexing [illegible]

these are O(1) as it takes only operation

if we do:

l1.pop(0) or l1.insert(0, 11)

these will shift all the index and this will be O(n) as all the elements gets shifted.

so removing & adding starting of a list O(n)

and if we add

& remove the items ⇒ we think its O(n/2)

in center

But Big O measures only

worst case, so its

Big O (n)

note:

⇒ if we iterate through a list its O(n)

⇒ if we get value with index its O(1) ⇒ as its

directly goes to index not value by value

$O(n^2)$ ⇒ Loop within a loop

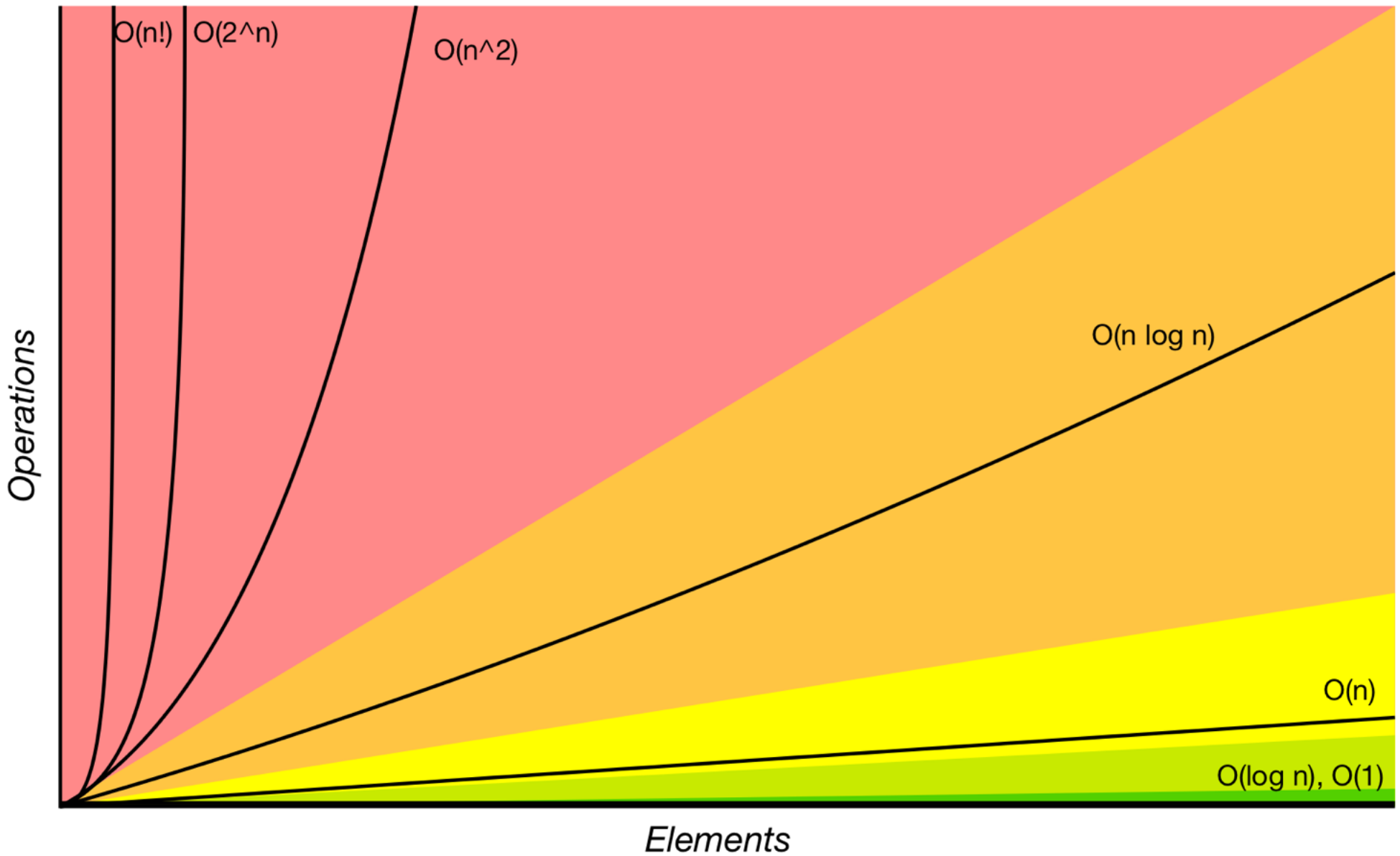
$O(n)$ ⇒ proportional

$O(\log n)$ ⇒ divide and conquer.

$O(1)$ ⇒ Constant

Big-O Complexity Chart

Horrible | Bad | Fair | Good | Excellent



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)

Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)