Time And Space Complexity

Time Complexity: It is used to measure efficiency of an algorithm in terms of speed, as the input grows.

T.C ↓ → speed ↑

Time complexity ≠ Time taken to run a piece of code → because, it depends on various factors like machine, language etc---

Speed & Efficiency ⟶ when input size grows

Linear Search

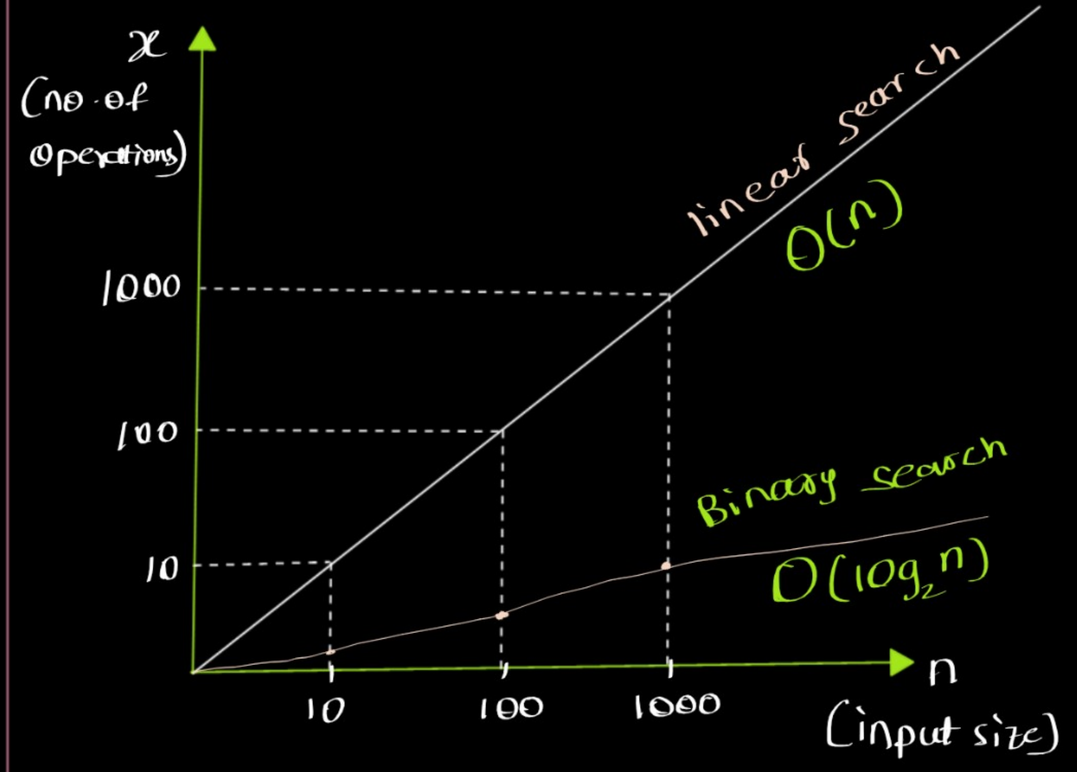
[5, 2, 1, 9, 12, 15]

n (no of elements) = 6

x (no of iterations) = 6

⇒ To search an element in an array using linear search, in worst case scenario it will take n iterations.

So, if n = 1000 then no of iterations x is also 1000.



Binary Search (only for sorted arrays)

[10, 20, 30, 40, 50, 60, 70]

n = 7

x (no of iterations) = 2 iteration

7/2 → 3
3/2 → 1

⇒ We will divide array by half and keep on doing it until only one element left.

$n/2 \times 1/2 \times 1/2 \cdots x = 1$

$n/2^x = 1$

$n = 2^x$

applying log on both sides

$\log_2 n = x \log_2 2$

$\therefore x = \log_2 n \quad (\because \log_2 2 = 1)$

So, no of iterations in binary search is $\log_2 n$ times.

if n = 1000 → $x = \log_2(1000) = 10$

x = 10

Binary search >>> Linear Search

Big-O notation: It is just a symbol which represents worst case

Linear search: [5, 2, 3 1, 2] → Best case → Search (5) → x = 1

→ Worst case → Search (100) → x = 5

x = n times

Binary search : [1, 2, 3, 4, 5] → Best case → Search (3) → $x=1$

→ worst case → Search (100) → 5/2, 1 } $x=2$

$$x = \log_2 n$$

linear search → $O(n)$

Binary Search → $O(\log_2 n)$

Efficiency $O(\log_2 n) > O(n)$

⇒ $O(n^2)$

```
for(i=0; i<n; i++)   → n times
{
  for(j=0; j<n; j++) → n times
  {
  }
}
```

n^2

⇒ $O(n \log n)$

```
for(i=0; i<n; i++)   → n times
{
    n/2, n/4, n/8 ... 1   → log n times
}
```

$n \log n$

⇒ $O(n^3)$ → three nested loops

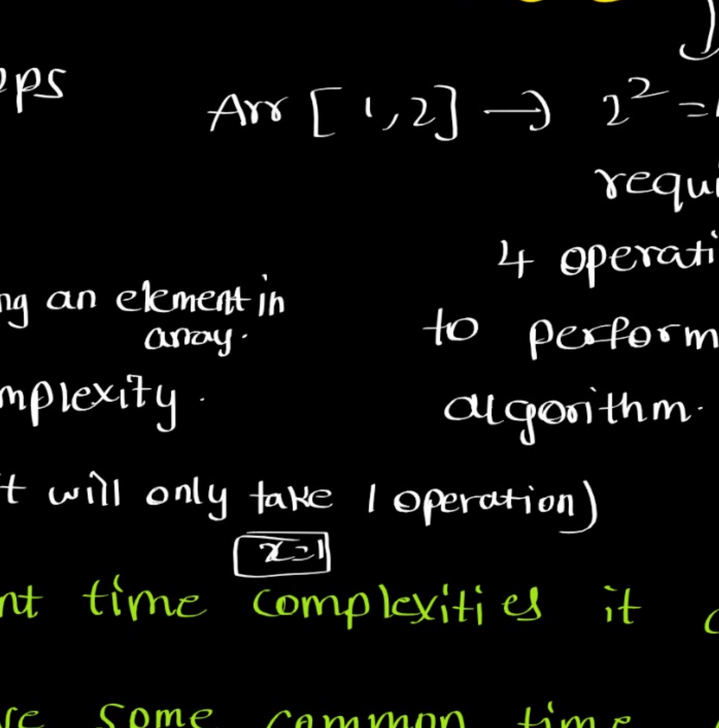
⇒ $O(2^n)$ → Arr [1, 2] → $2^2 = 4$ requires 4 operations to perform some algorithm.

⇒ $O(n!)$ → Arr [1, 2, 3, 4, 5] → $n!$ → $5! = 120$ requires 120 operations to perform some Algorithm.

⇒ $O(1)$ → Ex: accessing an element in array. Constant time complexity. (whatever the input it will only take 1 operation) [x=1]

⇒ Can have different time complexities it can be equations as well. But generally we have some common time complexities.

$O(1)$ $O(\log n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$



Efficiency:

$O(1) > O(\log n) > O(n)$

$> O(n \log n) > O(n^2)$

$> O(2^n) > O(n!)$

Space Complexity:

① 5th element in arr (arr)
{
return arr[4];
}
→ time → O(1)
→ Space → O(1) → we didn't use any extra space.

② findmax (arr) {
let max = arr[0]; → Space → O(1)
for(i=1; i<n; i++) { → time → O(n)
if(arr[i] > max) max = arr[i];
}
return max;
}

③ doubleArray(Arr)
{
new array = size(n) → Space → O(n)
for(i to n){ → time → O(n)
newarray[i] = arr[i] x 2;
}

⇒ for all variables it is constant space complexity → O(1)
⇒ for an array of size n space complexity → O(n)
⇒ for a 2d array then → O(n²)

[1, 2, 3, 4, 5] (n) →

1	2	3	4	5
2	4	6	8	10
4	8	12	16	20
8	16	24	32	40
16	32	48	64	80

n x n

for(0 to n) → n
{ for(0 to n) → n } → n²
{
}
}
nested loop
O(n²)

Ex! n = 1million
O((million)²)

for(0 to n) → n
for(0 to n) → n
for(0 to n) → n
} 3n but O(n)
individual loop

⇒ When Sample size becomes large then constant is negligible so, O(n)

Ex! n = 1million
O(3million) → ∴ negligible
O(million).

Ex:

```
for(0 to n) ──→ n  }
{                  } n²
  for(0 to n) ──→ n }
  {
  }
}
for(0 to n) ──→ n
{
}
```

$O(n^2+n)$

⇓

$O(n^2)$

$\Rightarrow O(n^3+n+2^2) \longrightarrow \boxed{O(n^3)}$

$\Rightarrow O(n^2+3n) \longrightarrow \boxed{O(n^2)}$

$\Rightarrow O(n^2+\log n+n+5) \longrightarrow \boxed{O(n^2)}$

Ultimately time complexity boils down to greater value. When input size becomes larger smaller values become negligible so, we can ignore smaller values.