# Time & Space Complexity

## What is Time Complexity?

Time complexity measures how efficient an algorithm is as the input size increases.

It’s not the same as the actual time taken to run a program.

`Time Complexity != Execution Time`

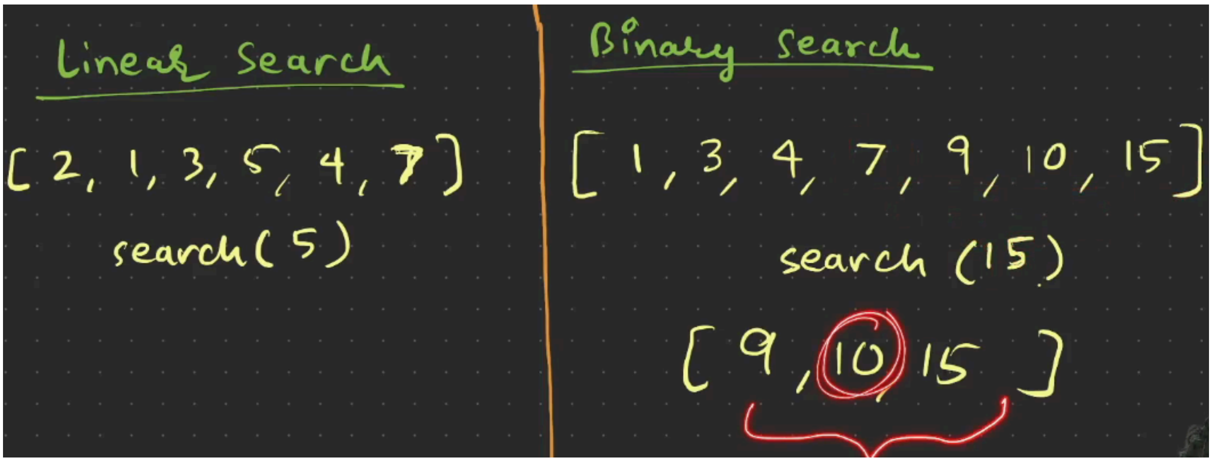
**Time Complexity ≠ Time Taken**

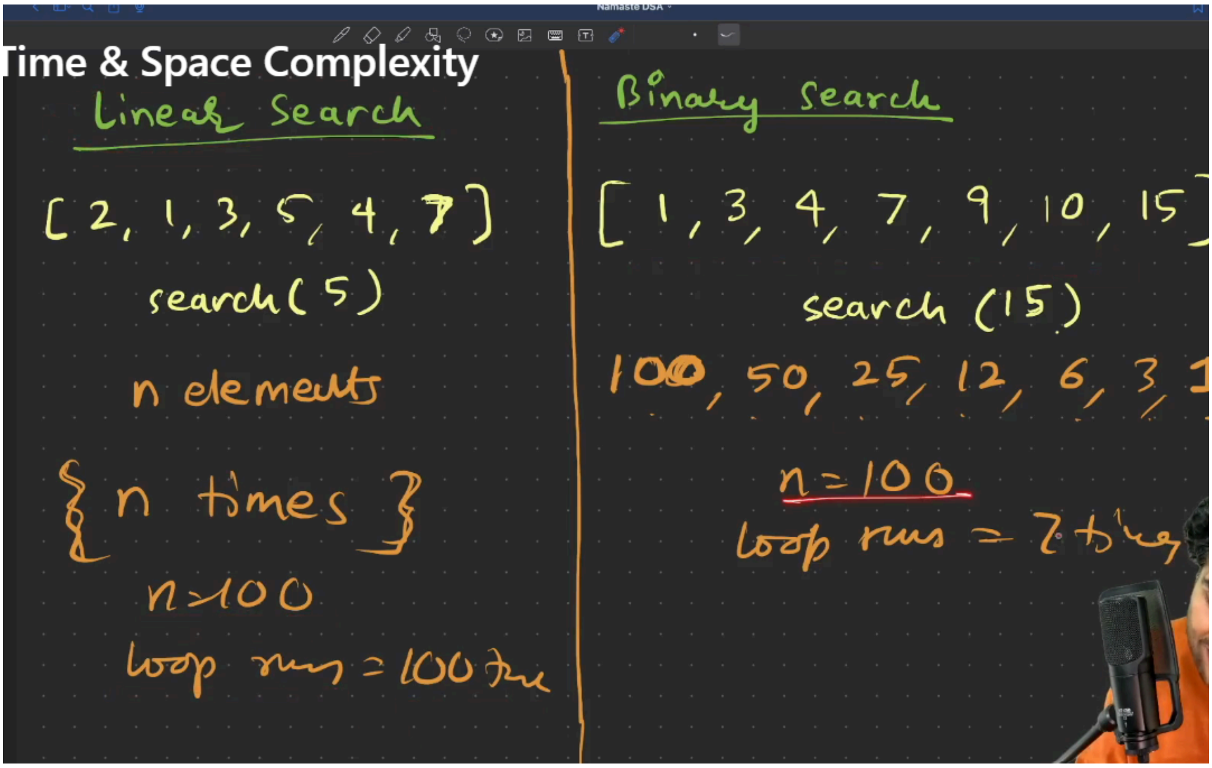
time taken of some code depend on where it will run, what are CPU Configuration

**Time Complexity** = Speed Efficiency ⇒ When Input Size Grows

### Examples

## Linear vs Binary Search

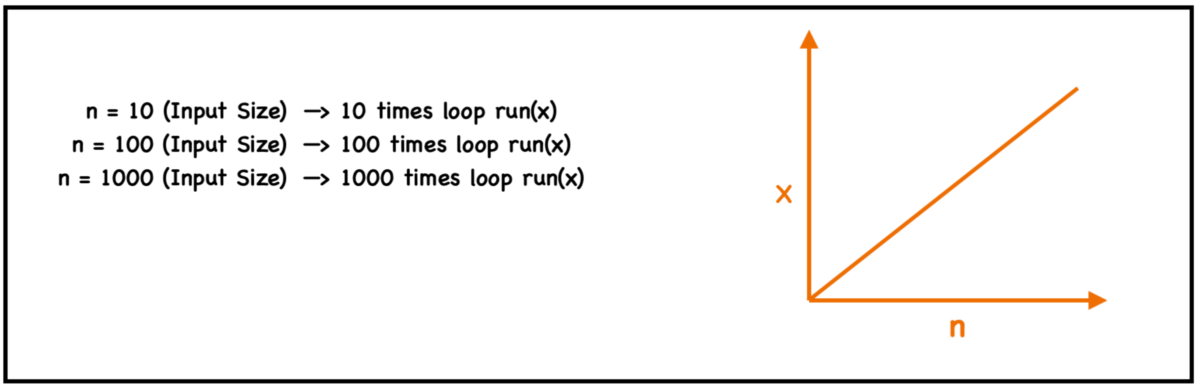





- when input size grows (increases) → how my Algorithm behave that is known as  Efficient or not

## Linear Search

- **Best Case:** Element at 1st index → 1 operation
- **Average Case:** Element at n/2 index → n/2 operations
- **Worst Case:** Element not found → n operations
- **Time Complexity:** O(n)
- **Requirement:** Can work on unsorted arrays
- in linear search if arry elements are n elements ⇒ n times itration happen

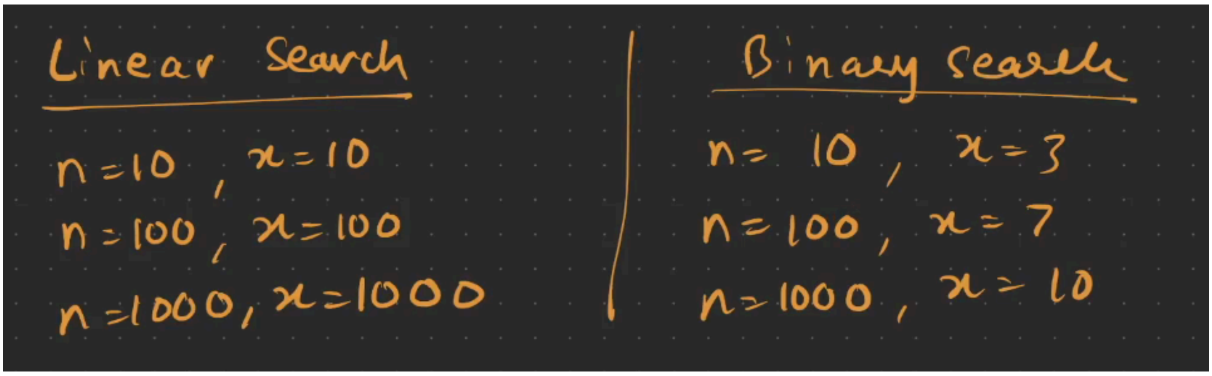


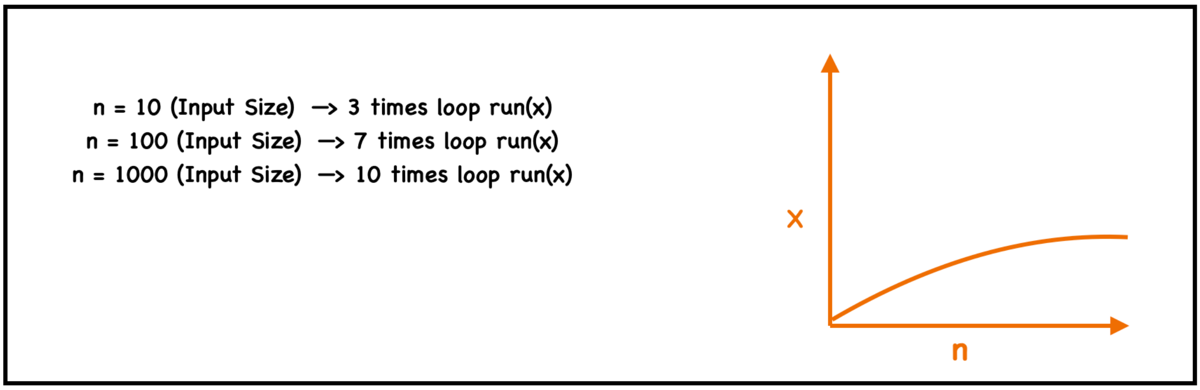
## Binary Search

- **Best Case:** Middle element matched → 1 operation
- **Average Case:** $\log_2(n)$ operations
- **Worst Case:** $\log_2(n)$ operations
- **Time Complexity:** O(log n)
- **Requirement:** Only works on sorted arrays

```
So i have n/2*1/2*1/2*1/2 .................X
Till it became 1
 n/2^x = 1
 n = 2^x
log2n = x

my loop and iternation run → log2n(Log of2^n) Times

in linear = n times
in binary  = (Log2N)

Binary is Speed Efficient

"LOG OF N BASE 2 === log2(n) === log2n"

log2(100) = 6.4 = ~7
log2(1000) = 9.94 =~ 10
```

- every time my loop runs and if i start with n element then its goes like → n→ n/2 →n/4→n/8






When we use `Linear Search` for an input size of 100, it runs 100 times, whereas `Binary Search` takes only 7 steps. This shows that Binary Search is more efficient. As the input size (n) increases, the way an algorithm behaves helps us understand how efficient it is. Also, the graph helps us understand that Binary Search is more efficient.

T.S. → 29

# Big O Notation

O(logn) > O(n)

It is nothing; just a symbol used to represent the worst-case complexity.

# Code Examples of Time Complexity

`O(1)`

```
// Accessing 5th index element
int value = arr[5];
```

The time complexity is O(1) because we directly access the 5th index without any iteration.

`O(n)`

```
for(int i = 0; i < n; i++) {
    // do something
}
```

`O(log n)`

```
// e.g., Binary Search
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while(low <= high) {
        int mid = (low + high) / 2;
        if(arr[mid] == key) return mid;
        else if(arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

`O(n^2)` – Nested Loop

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        // do something
    }
}
```

`O(n log n)`

```
for(int i = 0; i < n; i++) {
    int temp = n;
    while(temp > 1) {
        temp = temp / 2;
        // do something
    }
}
```

`O(n^3)` – Triple Nested Loops

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < n; k++) {
            // do something
        }
```

```
    }
}
```

O(2^n)

```
// Recursive Fibonacci
int fib(int n) {
    if(n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

O(n!)

```
// Permutation generator
void permute(string s, int l, int r) {
    if(l == r) {
        cout << s << endl;
    } else {
        for(int i = l; i <= r; i++) {
            swap(s[l], s[i]);
            permute(s, l + 1, r);
            swap(s[l], s[i]); // backtrack
        }
    }
}
```

## Time Complexity Priorities

- O(1) – Constant time

```
array size(n) = 1    → operation = 1
array size(n) = 10   → operation = 1
array size(n) = 100  → operation = 1
array size(n) = 1000 → operation = 1
```

```
find a number in 5th index
console.log(arr[5])
```

- `O(log n)` – e.g., Binary Search

```


n,n/2,n/4,,,,x = 1
x = (logn)
O(logn)
```

---

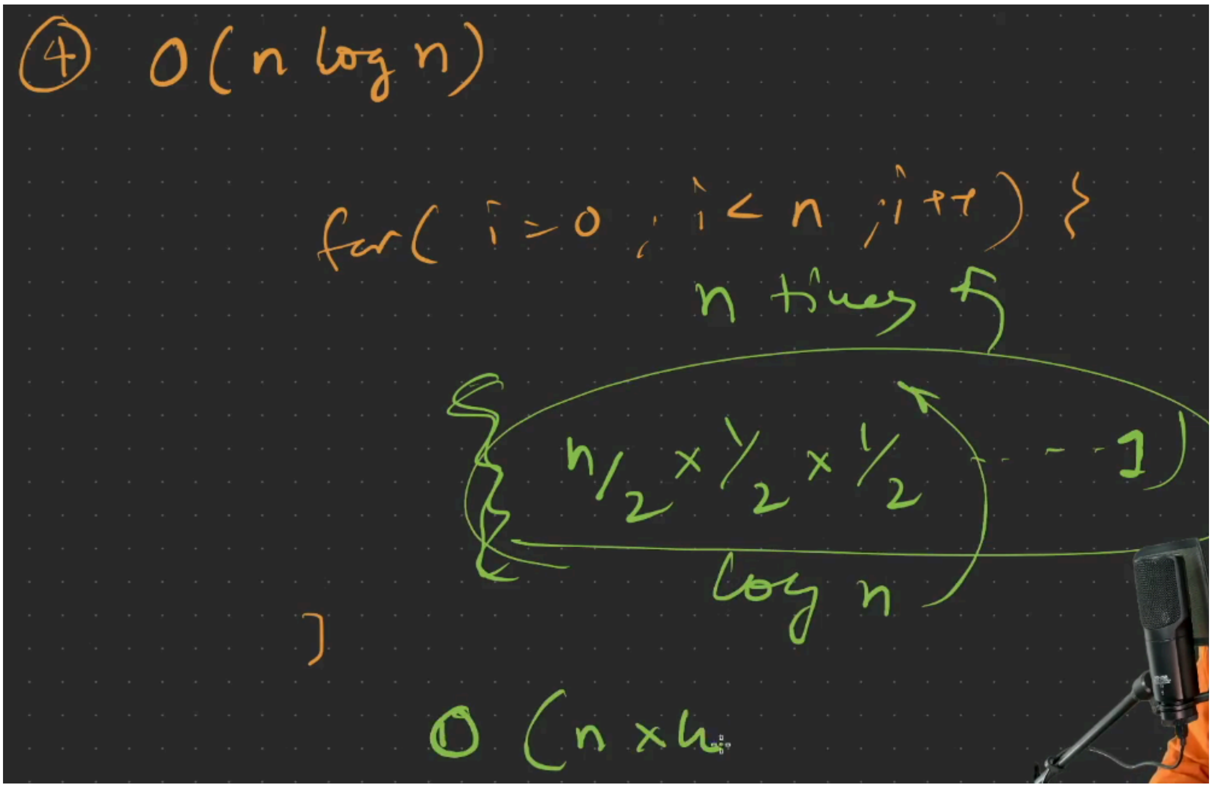
- `O(n)` – e.g., Linear Search

```
for(i=0; i<n;i++)
x = n
n = operation
```

---

- `O(n log n)` – e.g., Merge Sort

```
n*logn = in one loop there will be operation which we do n,n/2,n/4 ⟶ lo
gn
first loop = n;
second loop = logn ⇒ so 0(nlogn)
```




---

- O(n^2) – e.g., Nested Loops

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

      }
}

so n*n = n^2
```

- O(n^3) – e.g., Triple Nested Loops

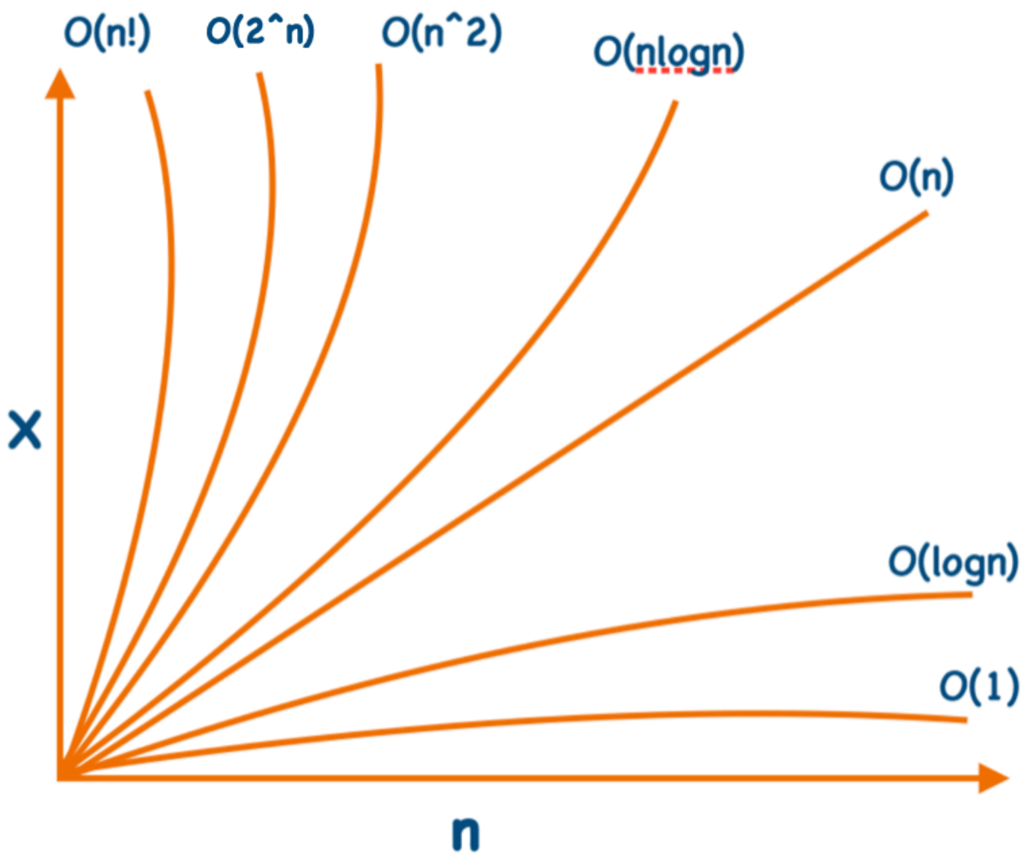
```
3 nested loop
```

- O(2^n) – Recursion (e.g., Fibonacci)

```
[2,2] → some algo which do 4 operation
[3,3] → some algo which do 9 operation
[4,4] → some algo which do 16 operation
[5,5] → some algo which do 25 operation
```

- `O(n!)` – e.g., Brute-force permutations

```
very rare
```




```
O(1) > O(logn) O(n) > O(nlogn) > O(n^2) > O(2^n) > O(n!)
```

## What is Space Complexity?

Space complexity refers to how much extra memory an algorithm uses.

### Examples:

- Access 5th element: `O(1)`
- Find max with variable: `O(1)`
- New array: `O(n)`
- 2D Matrix: `O(n^2)`

input array of size ⇒ n ⇒ O(n)

any related  1,2,3,4,5,6,7 ⇒ o(1) when variable is countable ⇒ constant space complexity ⇒ O(1)

newArray[n]  ⇒ O(n)

if you are creating 2d array ⇒ O(n^2)

```
//?1
for(i=0; i<n; i++){}
for(j=0; j<n; j++){}

n+n ⇒ time ⇒ O(2n) ⇒ when you came this type of complexity ⇒ goesa an
d tell ⇒ O(n)


//?2
for(i=0; i<n; i++){
   for(j=0; j<n; j++){

   }
}
O(n^2)

//?3
for(i=0; i<n; i++){}
for(j=0; j<n; j++){}
for(k=0; k<n; k++){}

O(3n) ⇒ but its O(n) Why ?

number of input size will be millions then that 3,10,20 ignore ⇒ O(n)

//?4

for(i=0; i<n; i++){
   for(i=0; i<n; i++){
```

```
            //logic
    }
}
for(j=0; j<n; j++){
          //logic
}

O(n^2 + n) ⇒O(n^2)



//?
O(n^3+n+n^2) ⇒ 0(n^3)

//?
O(n^2 +2n) ⇒ O(n^2)

//?
O(n^2+logn+2n+n) ⇒O(n^2)
```