Time Complexity and Space Complexity

Time Complexity is a way to measure **how fast an algorithm runs as the input size grows**.

Why Time Complexity matters

- Helps compare **algorithms**
- Predicts **performance for large inputs**
- Crucial for **DSA interviews**

How it's expressed

- Using **Big-O notation** → focuses on the **worst-case** growth.
- A **mathematical notation**
- Used to **represent** time complexity in the **worst case**

Notation	Meaning	Case
Big-O (O)	Upper bound	Worst case
Big-Ω (Omega)	Lower bound	Best case
Big-Θ (Theta)	Tight bound	Average / exact

First, one simple idea 🧠

When we talk about **time complexity**, we ask:

"How does the running time change when input size n increases?"

There are **3 notations** to describe this behavior.

1️⃣ Big-O (O) — Worst Case

What it means

👉 **Maximum time** an algorithm can take
👉 **Upper limit** (guarantee)

"In the worst situation, the algorithm will not take more than this."

Example: Linear Search

```
for i in arr:
```

```
    if i == x:
        return True
```

Cases:

- Best case: x at first position → 1 step
- Worst case: x at last position or not present → n steps

✅ **Big-O = O(n)**
Because in the **worst case**, it checks all elements.

Why Big-O is important

- Inputs are unpredictable
- Gives **safe guarantee**
- Used in **interviews**

2️⃣ **Big-Ω (Ω) — Best Case**

What it means

👉 **Minimum time** an algorithm can take
👉 **Lower limit**

"In the best situation, it will take at least this much time."

Same example: Linear Search

- Best case: element found at first index

✅ **Big-Ω = Ω(1)**

⚠️ But:

- Best case may **never happen**
- So it's **not reliable alone**

3️⃣ **Big-Θ (Θ) — Tight / Exact Bound**

What it means

👉 Describes **exact growth**
👉 Best case and worst case grow **at same rate**

"No matter what, time grows like this."

Example: Printing array

```
for i in arr:
    print(i)
```

- Best case: n steps
- Worst case: n steps

✅ **Big-Θ = Θ(n)**

Example: Accessing every element once

Code

```
def find_max(arr):
    max_val = arr[0]
    for i in arr:
        if i > max_val:
            max_val = i
    return max_val
```

Step-by-step thinking 🧠

No matter what the input is:

- Already sorted
- Reverse sorted
- Random values

👉 The loop **must check all n elements** to find the maximum.

Cases analysis

Case	Steps
Best case	n comparisons
Worst case	n comparisons

✔️ Growth rate is **always linear**

✅ **Summary: 3 Time Complexity Notations**

🔹 **1. Big-O (O) — Worst Case**

- **Meaning:** Maximum time an algorithm can take
- **Why used:** Guarantees performance in the worst situation
- **Example:** Linear search → **O(n)**

🔹 **2. Big-Ω (Ω) — Best Case**

- **Meaning:** Minimum time an algorithm can take
- **Why used:** Shows best possible performance
- **Example:** Linear search → **Ω(1)**

🔹 **3. Big-Θ (Θ) — Tight / Exact Bound**

- **Meaning:** Actual growth rate
- **Why used:** Best and worst cases grow at the same rate
- **Example:** Printing array → **Θ(n)**

First: the real question

Why do we mostly use Big-O notation and not Ω (best case) or Θ (tight bound)?

1️⃣ **Because Big-O gives a GUARANTEE**

Big-O = worst case

👉 It tells us the **maximum time** an algorithm can take.

Think like this:

"No matter what input comes, the algorithm will NOT be slower than this."

This guarantee is very important in:

- Exams
- Interviews
- Real systems

2️⃣ **Best case (Ω) is not reliable**

Example: Linear Search

for i in arr:

```
if i == x:
    return True
```

- Best case Ω(1): element at first position
- But what if:
    - Element is at last?
    - Element is not present?

👉 Best case may **never happen**

So Ω gives **false confidence**.

3️⃣ Θ (tight bound) is hard to prove

To say Θ:

- You must prove **best and worst grow the same**
- Not easy for many algorithms

Example:

- Quick Sort
    - Best: O(n log n)
    - Worst: $O(n^2)$

👉 So Θ is **not always possible**

4️⃣ Inputs are unpredictable in real life

We don't control:

- User data
- Order of elements
- Test cases

So we assume:

"Worst input can come"

Big-O prepares us for that.

5️⃣ Easy comparison between algorithms

Big-O helps compare algorithms clearly.

Example:

- Algorithm A → O(n)
- Algorithm B → O(n^2)

Even if B is fast sometimes,
👉 **A is safer always**

1️⃣ "No matter what input comes, the algorithm will NOT be slower than this"

Let's break this sentence.

Meaning in simple words:

Big-O tells the maximum time limit of an algorithm.
The algorithm will never take MORE time than that limit.

Example: Linear Search

Array size = n = 5

[10, 20, 30, 40, 50]

Cases:

- Best case → element found at 1st position → **1 step**
- Middle case → element at 3rd position → **3 steps**
- Worst case → element at last or not found → **5 steps**

👉 **Maximum steps = 5 = n**

So we say:

Time complexity = O(n)

This means:

- It may take **1, 3, or 5 steps**
- But it will **never take more than n steps**

That is what:

"NOT be slower than this" means

2️⃣ Very simple analogy (IMPORTANT) 🚶

Going to college

- Best case → no traffic → 20 minutes
- Worst case → heavy traffic → **60 minutes**

If you ask:

“How much time does it take to reach college?”

What will you answer?

❌ “20 minutes” (best case)
☑ **“At most 60 minutes”** (worst case)

Because:

- Sometimes traffic comes
- You must be prepared

👉 That **60 minutes = Big-O**

3️⃣ What does “reliable” mean here?

Reliable = Safe + Trustworthy

It means:

- The value **will not lie**
- It **works for all inputs**
- It gives a **guarantee**

Example:

- Saying “at most 60 minutes” → reliable
- Saying “20 minutes” → risky (may be wrong)

4️⃣ Why best case is NOT reliable

Best case says:

“If everything is perfect…”

But in real life:

- Input may be bad
- Data may be unordered
- Worst input can come

So:

- Best case is **hope**
- Worst case is **guarantee**

5️⃣ One sentence you can remember 🔑

- Big-O tells the maximum time an algorithm can take, so we can trust it for any input.

Real Life EG: why Big O alone considering?

Step 1: What does "reliable" actually mean here?

Reliable means:

Something you can **depend on**, even when conditions are bad.

In algorithms:

- Input order can be **good or bad**
- Data can be **small or large**
- We must give an answer that **will not fail**

What does "will not fail" mean?

- It means the statement we give should be **true for ALL inputs**, not just for some lucky cases.

Step 2: College travel example (DETAILED) 🚶

Imagine this situation:

You have an **important exam at 9:00 AM**.

You ask your friend:

"How much time will it take to reach college?"

Case A: Friend says "20 minutes" ❌ (Best case)

Why this is **risky**:

- This assumes:
    - No traffic
    - Green signals
    - Empty roads

But what if:

- There is traffic?
- There is road work?
- There is an accident?

👉 You may reach **late**
👉 This answer **fails** when conditions change

So:

- This answer works **only sometimes**
- You **cannot trust** it always

Case B: Friend says "At most 60 minutes" ✅ (Worst case)

Why this is **reliable**:

- This assumes:
    - Heavy traffic
    - Red signals
    - Worst road conditions

Now:

- If traffic is less → you reach early (good)
- If traffic is heavy → you still reach on time

👉 In **all cases**, this answer is correct
👉 It **never lies**

That's why it is **reliable**

Why interviews & systems need reliability

Think about:

- Online exams
- Banking apps
- Traffic systems

They cannot say:

“It will work fast sometimes”

They need:

“It will never cross this time limit”

That’s why:

- Big-O (worst case) is used
- It gives a **guarantee**

Note: Reliable means the answer remains correct even in the worst situation.

In Time Complexity (TC),

👉 we do NOT measure real clock time (seconds ⏱️).

👉 we measure how many basic operations an algorithm performs as input size grows.

Why don’t we use actual seconds?

Because seconds depend on:

- machine speed 💻
- programming language
- compiler
- background processes

But Time Complexity must be machine-independent.

So we ask:

“If input size becomes very large, how does the number of operations grow?”

Time Complexity = growth rate of operations as input size increases, independent of machine speed.

Big-O order

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

👉 **For DSA, you mainly need up to $O(n^2)$** ☑

Note: Reaching **cubic time complexity ($O(n^3)$)** is bad because the runtime grows extremely fast with input size, making the solution impractical for DSA and real-world problems.

1️⃣ **O(1) — Constant Time**

Meaning

- Time **does not change** with input size
- Always takes the **same amount of time**

👉 The operation takes **the same time, No matter how big the input is**.

Whether input size is:

- 1
- 10
- 1,000
- 1,000,000

⛔ Time does **NOT increase**.

Example 1

arr = [10, 20, 30]

print(arr[1])

Explanation

- Doesn't matter if array size is 10 or 1,000,000
- Accessing arr[1] takes **one step**

☑ **Very fast & best**

Example 2: Dictionary / HashMap lookup

freq = {1: 3, 2: 5, 3: 2}

print(freq[2])

Why O(1)?

- Hash table directly finds the key
- No iteration needed

☑ **O(1)** (average case)

How to identify O(1) quickly 🔑

Ask this question:

"Does this operation run without looping or recursion?"

If **YES** → **O(1)**

---

2 O(n) — Linear Time

Meaning

- Time grows **directly** with input size

Example 1:

```
for x in arr:
    print(x)
```

Explanation

- 5 elements → 5 steps
- 100 elements → 100 steps

Simple rule:

- If input doubles → time doubles
- If input triples → time triples

☑ **Acceptable & common**

Example 2 : Finding an element in an array (Linear Search)

```
def search(arr, x):
    for i in arr:
        if i == x:
            return True
    return False
```

Why O(n)?

- In worst case, element is **last or not present**
- Loop checks **every element**

Array size	**Comparisons**
5	5
100	100
1000	1000

👉 Time grows **linearly**

☑ **O(n)**

Very simple real-life analogy 🚶

Imagine checking attendance:

- 10 students → 10 checks
- 100 students → 100 checks

👉 One-by-one checking = **O(n)**

How to identify O(n) quickly 🔑

Ask:

“Is there a single loop that runs once for each element?”

If **YES** → **O(n)**

3️⃣ O(log n) — Logarithmic Time

Meaning

- Input size **reduces by half** each step

Example: Binary Search

searching in a sorted array

Explanation

- For 16 elements → steps ≈ 4
- For 1024 elements → steps ≈ 10
- Growth is **very slow**

✅ Efficient

When n becomes very large, log n stays small. Yes, when n is very large, log n is still very small because the input is reduced by half at every step.

Core idea (most important)

In **O(log n)**, the problem size is **reduced by half in every step**.

That’s why:

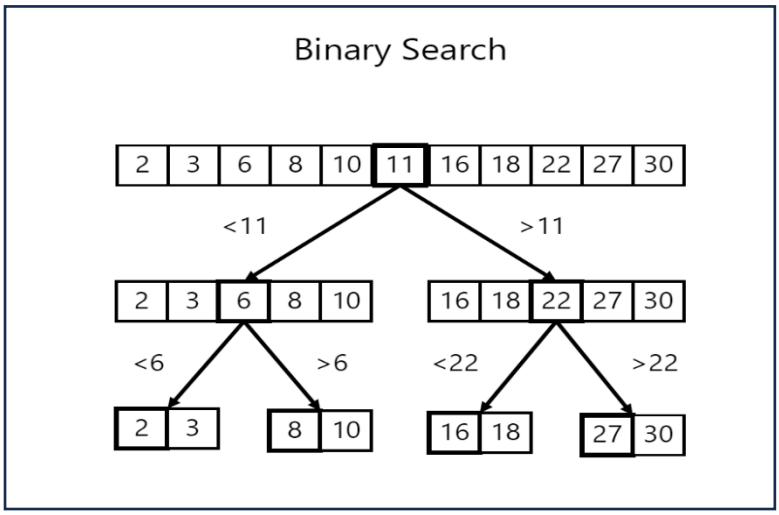
- n grows fast ❌
- log n grows **very slowly** ☑

Example 1 : Binary Search (classic)

Imagine a **sorted array**:

[1, 3, 5, 7, 9, 11, 13, 15]

n = 8



Steps:

1. Check middle → 7
2. Decide left or right half
3. Again check middle of that half

Each step:

8 → 4 → 2 → 1

👉 Number of steps ≈ **$\log_2(8) = 3$**

Example 2: Big n vs log n (THIS WILL CLICK 🤯)

n (input size)	**$\log_2(n)$ (steps)**
8	3
16	4
32	5
1,000	~10
1,000,000	~20
1,000,000,000	~30

👉 **n increased from 1 thousand to 1 billion**
👉 **log n increased only from 10 to 30**

That's why **log n is small even when n is huge**

Example 3: Guessing a number game 🎯

Think of this game:

- Number is between **1 and 100**
- Each time you guess, you are told **higher or lower**

Best strategy:

- Guess middle every time

Steps:

100 → 50 → 25 → 12 → 6 → 3 → 1

Only **7 guesses** needed! 👉 **O(log n)**

Example : Cutting a paper ✂️

Suppose you have a long paper:

- You fold it into half each time

Fold	Size
Start	n
1st fold	n/2
2nd fold	n/4
3rd fold	n/8

Each fold = **one step**

👉 Number of folds = **log n**

How to identify O(log n) quickly 🔑

Ask this question:

"Is the input size reduced (divided) in every step?"

If **YES** → **O(log n)**

🔴 O(n log n) — Linearithmic Time

Meaning

- Combines linear + logarithmic
- Seen in **efficient sorting**

Example

- Merge Sort
- Quick Sort (average case)

Explanation

- Divide array (log n)
- Process each level (n)

☑ Optimal for sorting

First, what does O(n log n) mean?

It means **two things are happening together**:

1. **log n** → the problem is **divided into smaller parts**

2. **n** → all elements are **processed at each level**

So:

Divide (log n times) + Work on all elements (n each time)

Best real example: Merge Sort

Suppose we have this array:

[8, 3, 5, 2, 7, 1, 6, 4]

n = 8

Step 1: Divide the array (this gives log n)

We keep dividing into halves:

Level 1: 8 elements

[8 3 5 2 | 7 1 6 4]

Level 2: 4 + 4

[8 3 | 5 2] [7 1 | 6 4]

Level 3: 2 + 2 + 2 + 2

[8|3] [5|2] [7|1] [6|4]

Level 4: single elements

How many levels?

8 → 4 → 2 → 1

That is:

$\log_2(8) = 3$ levels

👉 **This is the log n part**

Step 2: Merge (process all elements → n)

Now we **merge back**:

- At **each level**, we compare and merge **ALL elements**
- Total work at each level ≈ **n**

Example:

- Level 1 merge → 8 elements processed
- Level 2 merge → 8 elements processed
- Level 3 merge → 8 elements processed

👉 Work per level = **n**

Final calculation 🧠

- Number of levels = **log n**
- Work per level = **n**

So total time:

n × log n

☑ **Time Complexity = O(n log n)**

Very important table (THIS MAKES IT CLEAR)

For n = 8

Level	**Elements processed**
1	8
2	8
3	8

Total work:

8 + 8 + 8 = 24

Which is:

n × log n = 8 × 3

Compare with O(n) and O(n^2) (VERY IMPORTANT)

n	**O(n)**	**O(n log n)**	**O(n^2)**
100	100	~700	10,000
1,000	1,000	~10,000	1,000,000

👉 O(n log n) is **much faster than O(n^2)**
👉 That's why it's **optimal for sorting**

How to identify O(n log n) quickly 🔑

Ask:

1. Is the problem **divided into halves**? → log n
2. Are **all elements processed** at each step? → n

If **both YES → O(n log n)**

---

5 O(n²) — Quadratic Time

Meaning

- For every element, you loop through all elements again

Example 1:

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

Explanation

- 10 elements → 100 steps
- 100 elements → 10,000 steps

⚠ **Slow for large n**
✔ OK for small inputs

Example 2: Checking duplicates in an array

```
def has_duplicate(arr):
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:
                return True
    return False
```

Explanation

- First element compared with all others
- Second element compared with remaining
- Continues till end

For n elements:

- About n × n comparisons

☑ **O(n²)**

Example 3: Real-life analogy (VERY IMPORTANT) 🚶

Attendance checking

- Class has n students
- Teacher compares **each student with every other student**

If:

- 10 students → 100 checks
- 50 students → 2,500 checks

👉 Work = n × n
👉 **O(n²)**

How to identify O(n²) quickly 🔑

Ask:

"Is there a loop inside another loop that depends on input size?"

If **YES** → **O(n²)**

What does "ignore constants and lower-order terms" mean?

In **Time Complexity**, we care about:

How fast the work grows when input size n becomes very large.

Not the exact number of operations.

Step 1️⃣: Understand "growth" (VERY IMPORTANT)

Look at these two expressions:

- n^2
- $n^2 - n$
- $5n^2 + 100n + 1000$

As n becomes **very large** (say 1,000,000):

- n^2 → **huge**
- n → much smaller
- constants → almost nothing in comparison

👉 **n² dominates everything**

Step 2: Why we ignore lower-order terms

Example

$$n^2 - n$$

For small n:

- n = 5 → 25 − 5 = 20

For large n:

- n = 1,000 → 1,000,000 − 1,000
- n = 1,000,000 → 1,000,000,000,000 − 1,000,000

👉 That − n barely matters.

So we say:

$$O(n^2 - n) = O(n^2)$$

Step 3: Why we ignore constants

Example

$$5n^2$$

If:

- Computer A is 5× faster → time becomes n²
- Computer B is slower → time becomes 10n²

⚠ But the **growth trend is the same**.

So:

$$O(5n^2) = O(n^2)$$

Step 4: Real-life analogy 🚗 (VERY IMPORTANT)

Distance vs speed

- Car A: 60 km/h
- Car B: 120 km/h

Both:

- distance = **1000 km**

👉 Speed (constant) changes time,
👉 **distance (n)** decides how big the journey is.

Time Complexity measures **distance**, not speed.

Step 5: Golden Rule (Memorize This)

In Big-O notation, we keep only the fastest-growing term and drop constants and smaller terms.

Step 6: Common Examples

Expression	Big-O
n + 10	O(n)
3n + 100	O(n)
$n^2 + n$	$O(n^2)$
$n^2 + 100n + 50$	$O(n^2)$
log n + 5	O(log n)

Space Complexity

The amount of extra memory an algorithm needs to run, based on the input size.

It tells us **how much space (RAM)** an algorithm uses **as input grows**.

◇ **What space complexity includes**

1. **Input space** – memory used to store input (usually ignored)
2. **Auxiliary space** – extra memory used by the algorithm
   👉 **We mainly measure auxiliary space**

1 O(1) — Constant Space

Extra memory does **not grow** with input size.

👉 The algorithm uses a **fixed amount of extra memory**
👉 **Memory does NOT increase** when input size increases

Important:

- Input size can be **10 or 1,000,000**
- Extra memory used remains **the same**

Example 1: Sum of array elements

```
def sum_arr(arr):
    total = 0
    for x in arr:
        total += x
    return total
```

Extra memory used:

- total
- x

That's it ❗
No new array, no growing memory.

Even if:

- arr has 10 elements
- arr has 1 million elements

👉 Extra memory = **2 variables only**

Example 2: Swapping two numbers

```
a = 5
b = 10
a, b = b, a
```

Extra memory:

- No extra data structure
- Uses same variables

Real-life analogy 🤔

Imagine:

- You count students using **your fingers**

- Fingers count stays the same even if students increase

👉 That's **O(1) space**

2️⃣ O(n) — Linear Space [Extra memory grows **linearly**.]

👉 The **extra memory used increases in proportion to input size n**
👉 If input size doubles → memory usage also doubles

Example 1: Creating a new array or recursion depth = n.

```
def copy_array(arr):
    new_arr = []
    for x in arr:
        new_arr.append(x)
    return new_arr
```

Explanation

- If arr has 5 elements → new_arr stores 5 elements
- If arr has 1000 elements → new_arr stores 1000 elements

Extra memory used = size of new array = n

☑ **Space Complexity = O(n)**

Example 2️⃣: Storing frequencies (dictionary)

```
def count_freq(arr):
    freq = {}
    for x in arr:
        freq[x] = freq.get(x, 0) + 1
    return freq
```

Explanation

- In worst case, all elements are unique
- Dictionary stores n keys

Extra memory grows with input size

☑ **O(n) space**

4️⃣ O(n log n) — Rare in Space

Occurs in advanced algorithms using multiple levels of storage

👉 **Rare in DSA interviews**

Interviewers prefer solutions that are:

- **O(1)** space (best)
- **O(n)** space (acceptable)

Using **O(n log n)** extra memory is usually considered **inefficient**.

DSA focuses more on time optimization

When space increases too much:

- Memory limit can exceed
- Solution becomes impractical

So interviewers avoid asking problems that need **O(n log n) space**.

Where O(n log n) space MAY appear (just concept)

Example (conceptual, not interview-level)

- Recursive algorithms where:
    - Each level uses **O(n)** memory
    - Depth is **log n**

Total space:

n + n + n (log n times)

→ O(n log n)

But ⚠️:

- These are **advanced / theoretical**
- Not expected in normal DSA interviews

5️⃣ O(n²) — Quadratic Space

Using a 2D matrix.

Used only when the problem naturally requires it.

Example

dp = [[0]*n for _ in range(n)] (DP on **strings,pairs**)and Matrix problems

$O(n^2)$ space **is absolutely used in DSA**, mainly in dynamic programming ,grid problems and dense graphs.

Example: Longest Common Subsequence (LCS)

dp = [[0] * (m + 1) for _ in range(n + 1)]

Why $O(n^2)$?

- Rows → characters of string 1
- Columns → characters of string 2
- Each cell stores a result for a **pair**

👉 If strings are length n and n
👉 Space = n × n = $O(n^2)$

What you REALLY need for DSA interviews for Space Complexity

Focus mainly on:

- **O(1)** (best)
- **O(log n)** (recursion)
- **O(n)** (arrays, recursion stack)

Summary:

Complexity	Measures
Time Complexity	How fast the algorithm runs
Space Complexity	How much extra memory it uses

1. O(1) — Constant

- Hashing
- Direct access
- Math operations

2. O(log n) — Logarithmic

- Binary search
- Trees (BST height)
- Divide & conquer

3. O(n) — Linear

- Arrays

- Strings
- Two pointers
- Sliding window

4. O(n log n) — Linearithmic (VERY IMPORTANT) ⭐

- Merge sort
- Quick sort (average)
- Most optimal solutions

5. O(n^2) — Quadratic

- Nested loops
- Brute force
- DP (basic)

👉 Acceptable only when n is small (≤10^4 or less)

◇ Commonly used Time Complexities in DSA – O(1),O(log n), O(n), O(n logn), O(n^2)

◇ Commonly used Space Complexities in DSA - O(1),O(log n), O(n)

- Created by Rakshitha A
- GitHub: github.com/Rakshitha20045
- LinkedIn: linkedin.com/in/rakshitha-a-4320a3274