

## ★ 1) What happens when we analyze algorithms?

When we analyze an algorithm, we describe its running time using **mathematical functions** such as:

$$f(n) = n^2 + 6n + 5$$

Here, **n = size of input.**

As n becomes very large:

- $n^2$  grows extremely fast
- $6n + 5$  grow much more slowly
- So the **extra terms become insignificant** compared to  $n^2$

**Example:**

Let n = **1000**

- $n^2 = 1,000,000$
- $6n = 6,000$  (very small in comparison)
- $5 = 5$  (tiny)

So:

$$n^2 + 6n + 5 \approx n^2$$

This process of focusing only on the most important term is called:

## Asymptotic Analysis

“Asymptotic” means:

How the algorithm behaves when **n becomes extremely large.**

---

## ★ 2) Why do we ignore constants?

For big inputs:

- $+5 \rightarrow$  makes almost no difference
- $+6n \rightarrow$  grows slowly
- $n^2 \rightarrow$  dominates everything

So we keep only the term that matters the most — the **dominant term**.

### Examples:

$$n^2 + 6n + 5 \rightarrow O(n^2)$$

$$10n + 9999 \rightarrow O(n)$$

$$5n^3 + 2n^2 \rightarrow O(n^3)$$

We ignore:

- Extra constants ( $+5, +9999$ )
- Smaller terms ( $6n, 2n^2$ )

- Constant multipliers (like the 5 in  $5n^3$ )

Because for very large n, **they don't affect the overall growth.**

## ★ POINT 3: Best Case, Worst Case, Average Case (Simplified Explanation)

### Real Life Example: Searching for a Book in a Library Shelf

Imagine you have **100 books** on a shelf.  
You want to find "**Harry Potter**".

---

#### BEST CASE — (Luckiest Situation)

**Harry Potter** is the **FIRST** book!

What happens:

- You check book 1 → It is Harry Potter ✓
- Only **1 check** needed

Time complexity → **O(1)**  
(Constant time — super fast)

---

#### WORST CASE — (Unluckiest Situation)

Two possibilities:

1. Harry Potter is the **LAST** book

2. Harry Potter is **NOT** in the shelf

What happens:

- You check book 1 → Not found
- You check book 2 → Not found
- ...
- You check all 100 books

Total checks = **100**

Time complexity → **O(n)**  
(Linear time — slow)

---



## AVERAGE CASE — (Normal Everyday Situation)

**Harry Potter is somewhere in the MIDDLE.**

What happens:

- On average, you find it around the 50th position
- About **n/2 checks**

Time complexity → **O(n)**

We ignore the **1/2** in Big-O.

---



## CODE VERSION: Linear Search

```
def linear_search(arr, target):  
    for i in range(len(arr)):  
        if arr[i] == target:  
            return i  
    return -1
```

---

## BEST CASE (Target at beginning)

Example: Search **10** in [10, 20, 30, 40, 50]

- Check index 0 → Found
- Loop ran **1 time**

Time → **O(1)**

---

## WORST CASE (Target at end or not present)

Example: Search **50**

- Check indices 0 → 10 **✗**
- Check index 1 → 20 **✗**
- Check index 2 → 30 **✗**
- Check index 3 → 40 **✗**
- Check index 4 → 50 **✓**

Loop ran **5 times** → **O(n)**

Or search for **99** (not in array):

- Checked all 5 elements → Not found
  - Loop ran **5 times** → **O(n)**
- 

## AVERAGE CASE (Target in middle)

Example: Search 30

- Check index 0 → 10 ✗
- Check index 1 → 20 ✗
- Check index 2 → 30 ✓
- Loop ran **3 times**  $\approx n/2$

Time → **O(n)**

( $n/2$  becomes  $n$  in Big-O)

---

## 🎯 WHY DO WE IGNORE THE “1/2” IN AVERAGE CASE?

$$\text{Average} \approx (n + 1) / 2$$

For large  $n$ :

- $(n + 1)/2 \approx n/2$
- Big-O ignores constants →  $n/2$  becomes  $n$

So:

$$O(n/2) = O(n)$$

Both grow linearly.

---

$n = 1000$ :

Best case: 1 operation

Worst case: 1000 operations

Average case: ~500 operations

All are  $O(n)$  because:

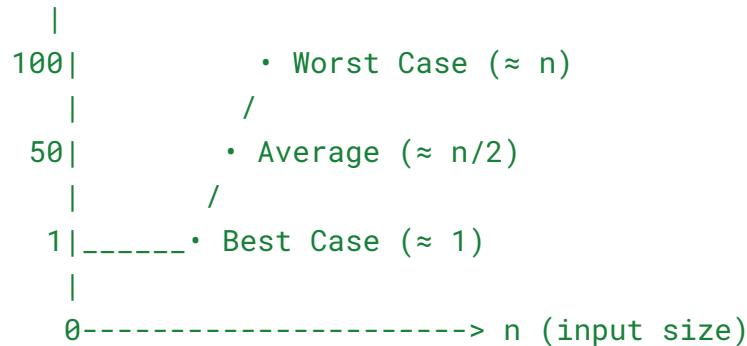
- 1, 500, 1000 all grow LINEARLY with  $n$

- The "shape" of growth is the same line



## Visual Growth Comparison (small ASCII)

### Operations



### Interpretation:

- Best-case is a flat constant line at 1.
- Average and worst both scale linearly with  $n$  (same slope), so both are  $O(n)$ .



## Summary Table

Case	Meaning	Example (book position)	Checks Needed	Complexity
Best	Easiest input (lucky)	First book	1	$O(1)$
Average	Typical/random input	Middle	$\sim n/2$	$O(n)$
Worst	Hardest input / not present	Last or not in shelf	$n$	$O(n)$

**Key point:** We usually use **Worst Case (Big-O)** to guarantee an upper bound on running time.



## Quick Notes to Avoid Confusion

- $O(1)$  does **not** mean "1 millisecond"; it means **constant number of operations** regardless of  $n$ .
  - $O(n)$  means operations grow **linearly** with  $n$ . Doubling  $n$  roughly doubles operations.
  - Average  $n/2$  looks smaller, but Big-O removes constants → still  $O(n)$ .
- 

## Final Takeaway (one-liner)

For linear search:

- Best =  $O(1)$  (target at front)
  - Average =  $O(n)$  (target somewhere in middle)
  - Worst =  $O(n)$  (target at end or not present)  
And when comparing algorithms, we **usually rely on worst-case (Big-O)** for guarantees.
- 

## 4 . Why Worst-Case Matters? — SINGLE EXAMPLE

### Example: Hospital Emergency System

#### Code Scenario

```
def find_available_doctor():
    for doctor in all_doctors:
        if doctor.is_available:
            return doctor
    return None # WORST CASE: All doctors busy
```

---



#### The Situation

## ✓ Normal Day

- Only 2–3 doctors busy
- You find an available doctor quickly
- Loop stops early → **Best Case O(1)** or **Average Case O(n/2)**

## ✗ Emergency Day

- Major accident → **ALL doctors busy at the same time**
  - System must check **every single doctor**
  - This becomes the **Worst Case O(n)**
  - AND this is the scenario you *must* prepare for
- 



## The Guarantee (Why worst-case matters)

A hospital must guarantee:

"Even when EVERY doctor is busy, our system will respond within **2 seconds** and show 'No doctors available' instead of crashing."

This guarantee is **based on worst-case time complexity**, not average case.

---



## What Happens if You Ignore Worst Case?

If worst-case isn't planned:

- ✗ System may get stuck looping forever
- ✗ App may freeze or crash
- ✗ Patients won't see correct status

-  No one knows all doctors are busy
-  Critical delay in emergencies

Lives could literally depend on that delay.

---

## What Happens WITH Worst-Case Planning?

If worst-case is planned:

-  System gracefully handles “all doctors busy”
-  Returns **None** safely
-  Quickly shows the message “**No doctors available**”
-  System stays stable under maximum load
-  Guarantees response time even in emergency situations

This is why engineers analyze the **upper bound** (Big-O worst case).



## Bottom Line (One Sentence)

**When lives depend on a system, you must design for the worst possible scenario — not the average one.**

---

# ⭐ 5) Asymptotic Notations — Explained Simply

Let's understand the three main notations using a **car speed analogy** 

When we evaluate an algorithm, we think about:

- Worst case
- Best case
- Exact/typical case

That's where Big-O, Big-Ω, and Big-Θ come in.

---

## 1 Big-O (Upper Bound) — WORST CASE

**Meaning:** “How slow can my algorithm get at maximum?”

Think of this as:

👉 Maximum speed limit your car can reach on a road.

**Code Example (Linear Search):**

```
def linear_search(arr, target):  
    for item in arr:          # WORST CASE:  
        if item == target:    # Target is last, or never found  
            return True         # Must check ALL elements  
    return False
```

**Big-O = O(n)**

Because in the worst case, we check every element.

→ Like saying:  
“This car will NEVER go faster than 120 km/hr.”

Big-O gives you a **guarantee on the slowest performance**.

---

## 2 Big-Ω (Omega) — BEST CASE

**Meaning:** “How fast can my algorithm be at minimum?”

Think of this as:

👉 **Minimum speed** (yes, even 0 km/hr when parked!)

### Code Example (Linear Search):

```
def linear_search(arr, target):
    for item in arr:          # BEST CASE:
        if item == target:    # Target is FIRST element
            return True        # Found immediately
    return False
```

### Big- $\Omega$ = $\Omega(1)$

Because in the best case, only **one check** is enough.

→ Like saying:

“This car CAN go as slow as 0 km/hr.”

Big- $\Omega$  shows the **potential best performance**.

---

## 3 Big- $\Theta$ (Theta) — TIGHT BOUND

**Meaning:** “How long does it take exactly, regardless of situation?”

Think of this as:

👉 **Cruise control fixed speed** — always constant.

### Code Example (Sum of Array):

```
def sum_array(arr):
    total = 0
    for number in arr:      # MUST visit every element
        total += number     # No shortcuts or early exit
    return total
```

### Big- $\Theta$ = $\Theta(n)$

Because it **always** loops through all n elements.

→ Like saying:

"**This car always drives at exactly 60 km/hr.**"

Big- $\Theta$  applies when **best = average = worst**.

---

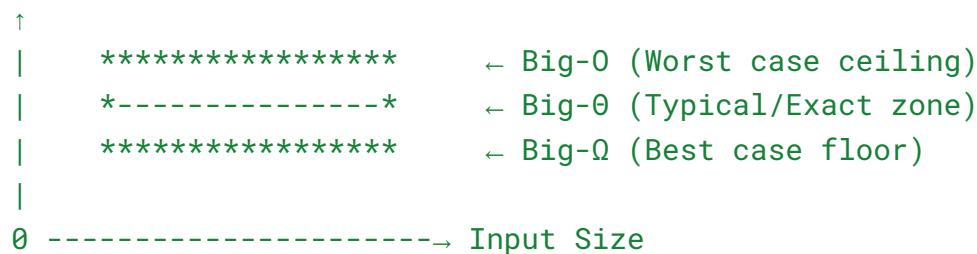
## 🎯 QUICK SUMMARY TABLE

Notation	Meaning	Example	Real Life Analogy
Big-O	Worst case	Linear search — $O(n)$	Maximum speed limit
Big- $\Omega$	Best case	Linear search — $\Omega(1)$	Minimum possible speed
Big- $\Theta$	Exact bound	Sum array — $\Theta(n)$	Cruise control fixed speed

---

## 📊 VISUAL REPRESENTATION (Simple View)

Performance Time



## FINAL REMEMBER POINTS

- **Big-O** = Your **promise** (max time) ⚡
- **Big- $\Omega$**  = Your **potential** (best time) 🚀

- **Big-Θ = Your certainty** (exact behavior) ✓
- 



## 6) Tiny Graphs to Understand Growth



graphs help you visualize how different time complexities grow as input size increases.

---

### Constant — O(1)

Always takes the same time, no matter how large input becomes.



### Logarithmic — O(log n)

Grows very slowly, almost flat. Common in binary search.



## Linear — $O(n)$

Grows directly in proportion to input size.

```
|  
|     /  
|     /  
|     /  
| - /
```

---

## Linearithmic — $O(n \log n)$

Slightly more than linear; common in efficient sorting algorithms.

```
|  
|     /  
|     /  
|     /  
| - /
```

(Almost linear but bends upward)

---

## Quadratic — $O(n^2)$

Growth becomes much faster; nested loops often cause this.

|

|

|

|

|

| /

---

## Cubic — $O(n^3)$

Even steeper growth; triple nested loops.

|        /

|        /

|       /

|\_ /

---

## Exponential — $O(2^n)$

Very steep, grows explosively. Often seen in brute-force recursive problems.

|

|       /

|       /

| / | /

---

| / | /

| / | /

| / | /

---



## 7) Summary Table (VERY IMPORTANT)



Case	Meaning	Expressed As	Used For
Big-O	Upper bound	Worst case	Guarantees maximum time
Big-Ω	Lower bound	Best case	Minimum possible time
Big-Θ	Tight bound	Average/exact	When best = worst

---

## 8) One-Line Summary of Everything

"Ignore constants, focus on growth rate: Worst = Big-O, Best = Omega, Exact = Theta."

# What is Big O Notation?

**Big O = Worst-Case Performance Guarantee**

It answers one question:

"How slow can my algorithm get in the worst possible scenario?"

---



## Mathematical Meaning of Big O

To describe an algorithm, we compare:

- $F(n)$  → Actual running time
- $G(n)$  → Simplified function showing growth pattern

Big O says:

$$F(n) \leq C \times G(n) \quad \text{for all } n \geq n_0$$

Where:

- $C$  = some constant multiplier
- $n_0$  = point from where inequality always holds

If such  $C$  and  $n_0$  exist  $\rightarrow F(n)$  is  $O(G(n))$ .

---



## Example 1 — Proving Big O

We want to prove:

$$F(n) = 5n + 4 \text{ is } O(n)$$

### ✓ Step 1: Write inequality

$$5n + 4 \leq C \times n \quad \text{for all } n \geq n_0$$

### ✓ Step 2: Find valid values for C and $n_0$

Try  $C = 6$ :

$$5n + 4 \leq 6n$$

$$4 \leq n$$

$$n \geq 4$$

Works → so  $C = 6, n_0 = 4$

Try  $C = 7$ :

$$5n + 4 \leq 7n$$

$$4 \leq 2n$$

$$n \geq 2$$

Also works → so  $C = 7, n_0 = 2$

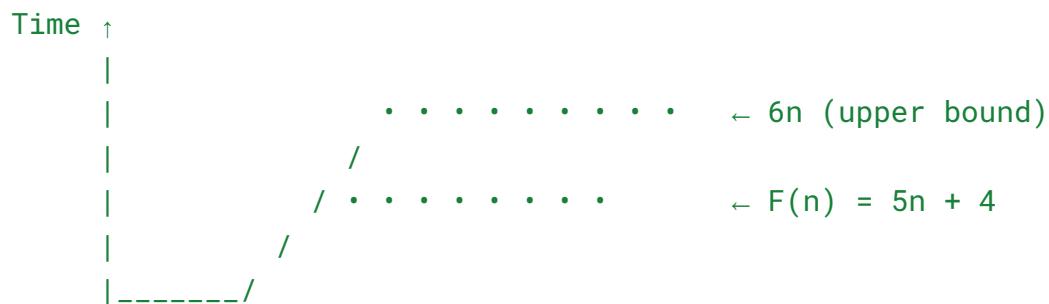
### ✓ Step 3: Conclusion

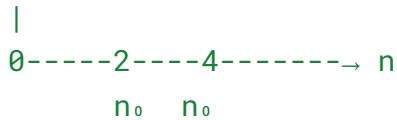
Since we found constants that satisfy the definition:

$$5n + 4 = O(n)$$



## Visual Understanding





As  $n$  grows large,  $F(n)$  stays **below** the line  $C \times n$ .

---



## More Examples (Highest-Term Rule)

### PIZZA PARTY ANALOGY — Big-O Notation

Imagine you're organizing a huge party and ordering food.

---

### Cost for 1 Party (PDF-Safe Table)

Item	Cost
<hr/>	
1 Giant Pizza	₹500
4 Coke Bottles	₹200
2 Garlic Bread	₹150
1 Dessert	₹100
<hr/>	
Total	₹950

---

Now Suppose Party Size = 100×

Item	Quantity	Cost
Giant Pizzas	100	₹50,000
Cokes	400	₹20,000
Garlic Bread	200	₹15,000
Desserts	100	₹10,000
<b>Total</b>		<b>₹95,000</b>

**Pizza alone = ₹50,000**

**All other items combined = ₹45,000**

**Pizza dominates the total cost.**

**So when planning for a huge party:**

**“Budget around ₹50,000 for pizzas.”**

**Everything else becomes relatively small.**

---

## Connecting This to Big-O Notation

**Consider the function:**

$$F(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$$

**As n becomes large, one term grows the fastest:**

**Dominant Term =  $5n^4$**

**All smaller terms become insignificant.**

---

## When $n = 10$

Term	Value
------	-------

---

$5n^4$	50,000
--------	--------

$3n^3$	3,000
--------	-------

$2n^2$	200
--------	-----

$4n$	40
------	----

1	1
---	---

---

Total	53,241
-------	--------

Dominant term = 94% of total

---

## When $n = 100$

Term	Value
------	-------

---

$5n^4$	500,000,000
--------	-------------

$3n^3$	3,000,000
--------	-----------

$2n^2$	20,000
--------	--------

$4n$	400
------	-----

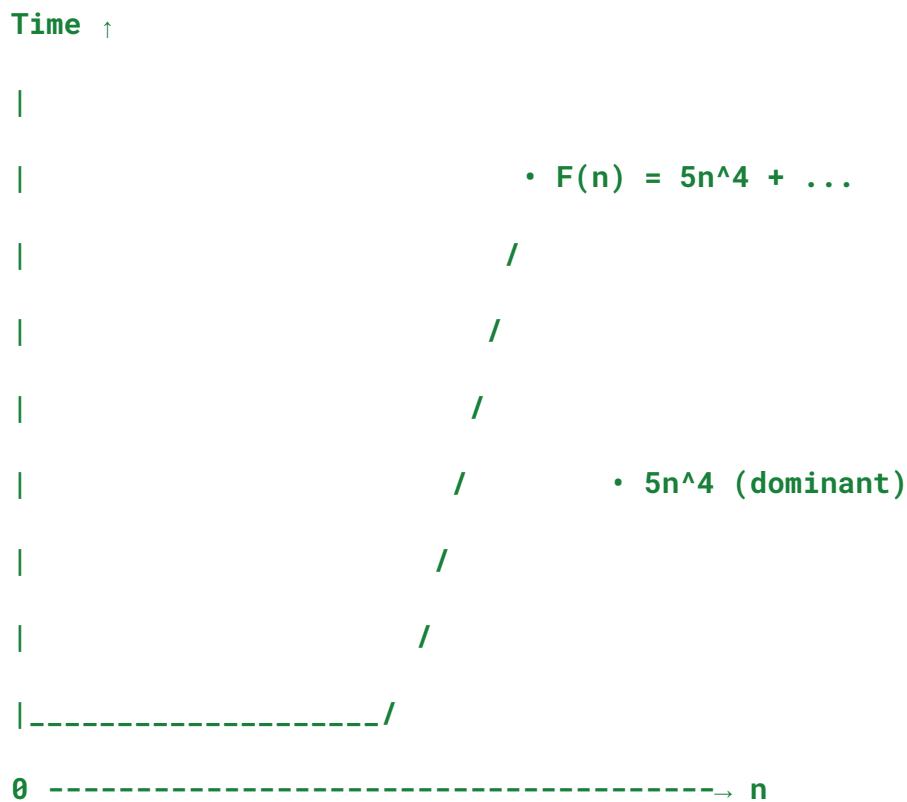
1	1
---	---

-----  
**Total      503,020,401**

Dominant term = 99.4% of total

---

## Graph Intuition (Text Version)



As  $n$  increases, smaller terms flatten out and become irrelevant.

---

## Rule of Big-O

**For large values of n:**

The fastest-growing term decides the Big-O.

**Growth speeds:**

$n$  → slow

$n^2$  → faster

$n^3$  → fast

$n^4$  → very fast

$2^n$  → extremely fast

So we keep only the dominant term.

---

## Quick Reference Table

Function	Dominant Term	Big-O
-----		
$5n^4 + 3n^3 + \dots$	$n^4$	$O(n^4)$
$20n^3 + 10n \log n + 5$	$n^3$	$O(n^3)$
$2^n + n^{100} + 100$	$2^n$	$O(2^n)$

---

## Final Analogy

$n^4$  = Elephant

$n^3$  = Large Dog

$n^2$  = Cat

$n$  = Mouse

1 = Ant

When they stand together,  
you only notice the elephant.

That's why Big-O keeps only the dominant term.

---

## Key Points to Remember

- Big O gives a **worst-case upper bound**
  - Ignore **constants** and **lower terms**
  - Only the **fastest-growing term** matters
  - We want a **simple but tight** upper bound  
(Example:  $n^2$  is tighter than  $n^3$ )
- 



## Why Big O is Useful

When we say:

$$5n + 4 = O(n)$$

We mean:

In the worst case, the algorithm grows linearly with input size.

We are **not** claiming it performs exactly  $5n + 4$  operations.

---



## Quick Practice

Try to identify the Big O:

Function	Big O
$3n^2 + 8n + 10$	$O(n^2)$
$100 \log n + 50$	$O(\log n)$
$n^3 + 2^n$	$O(2^n)$ (because exponential dominates)

---



## What is Big Omega ( $\Omega$ ) Notation?

**Big Omega = Best-Case Performance Guarantee**

It answers the question:

**"How fast can my algorithm possibly run in the best situation?"**

If you get extremely lucky, what is the minimum time it will take?

---



## Mathematical Definition of Big $\Omega$

We compare two functions:

- $F(n) \rightarrow$  Actual runtime
- $G(n) \rightarrow$  Simplified minimum growth function

Big Omega is defined as:

$$F(n) \geq C \times G(n) \quad \text{for all } n \geq n_0$$

Where:

- $C = \text{constant}$
- $n_0 = \text{starting point where inequality always holds}$

If we can find such constants  $\rightarrow F(n)$  is  $\Omega(G(n))$ .

---



## Example — Proving Big $\Omega$

Given:

$$F(n) = 5n + 4$$

We want to prove:

$$F(n) = \Omega(n)$$

---

### ✓ Step 1: Write inequality

$$5n + 4 \geq C \times n \quad \text{for all } n \geq n_0$$

---

### ✓ Step 2: Choose constants $C$ and $n_0$

Try  $C = 1$ :

$$\begin{aligned} 5n + 4 &\geq 1 \times n \\ 5n + 4 &\geq n \\ 4n + 4 &\geq 0 \end{aligned}$$

This is always true when  $n \geq 1$ .

So our constants:

- $C = 1$
  - $n_0 = 1$
- 

### ✓ Step 3: Conclusion

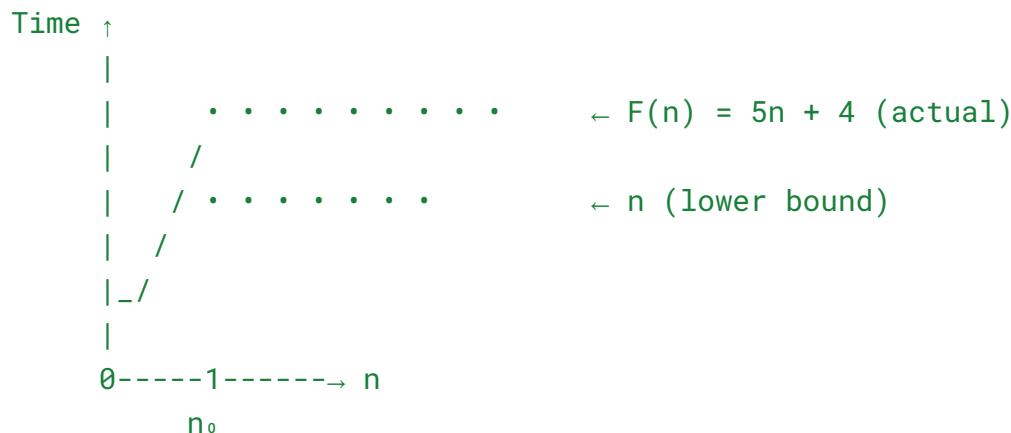
We found valid constants, so:

$$5n + 4 = \Omega(n)$$

---



## Visual Understanding



Meaning:

Our algorithm can *never* run faster than linear time.

---



## More Examples

**Big Theta = Tight Bound (both sides)**

It answers:

"Does this algorithm grow exactly like this function (not faster, not slower)?"

We must prove:

$$C_1 \times G(n) \leq F(n) \leq C_2 \times G(n) \quad \text{for all } n \geq n_0$$

We need **both lower and upper bound** → sandwich method.

---



## Example 1: $F(n) = 3n^2 + 8n + 10 = \Theta(n^2)$

We want to show:

$$C_1 \cdot n^2 \leq 3n^2 + 8n + 10 \leq C_2 \cdot n^2$$

---



### Step 1: Lower Bound (Find $C_1$ )

Check if the function is **at least** some constant  $\times n^2$ :

$$3n^2 + 8n + 10 \geq 3n^2$$

Because:

- $8n \geq 0$  for all  $n \geq 1$
- $10 \geq 0$

So:

- $C_1 = 3$
  - Works for all  $n \geq 1$
-

## Step 2: Upper Bound (Find $C_2$ )

We need a constant  $C_2$  such that:

$$3n^2 + 8n + 10 \leq C_2 n^2$$

Try  $C_2 = 4$ :

$$3n^2 + 8n + 10 \leq 4n^2$$

$$8n + 10 \leq n^2$$

$$n^2 - 8n - 10 \geq 0$$

Solve inequality:

The expression becomes positive when  $n \geq 10$

Check:

- $n = 10 \rightarrow 100 - 80 - 10 = 10 \geq 0 \checkmark$
- $n = 20 \rightarrow 400 - 160 - 10 = 230 \geq 0 \checkmark$

So inequality works.

Thus:

- $C_2 = 4$
  - $n_0 = 10$
- 



## Final Conclusion for Example 1

For  $n \geq 10$ :

$$3n^2 \leq 3n^2 + 8n + 10 \leq 4n^2$$

So:

✓  $F(n) = \Theta(n^2)$

---

## 🔍 Example 2: $F(n) = 7n \log n + 2n = \Theta(n \log n)$

Goal:

$$C_1 n \log n \leq 7n \log n + 2n \leq C_2 n \log n$$

---

### ✓ Step 1: Lower Bound ( $C_1$ )

Observe:

$$7n \log n + 2n \geq 7n \log n$$

Because  $2n \geq 0$

So simplest:

$$\rightarrow C_1 = 7$$

But we can even use **6** (a smaller constant), because:

Check:

$$7n \log n + 2n \geq 6n \log n$$

$$n \log n + 2n \geq 0$$

This is true for all  $n \geq 2$ .

Thus:

$$\rightarrow C_1 = 6$$

$$\rightarrow n_0 = 2$$

Either  $C_1=7$  or  $6$  works.

---

## Step 2: Upper Bound ( $C_2$ )

We need:

$$7n \log n + 2n \leq C_2 n \log n$$

Divide both sides by  $n$  ( $n > 0$ ):

$$7 \log n + 2 \leq C_2 \log n$$

Try  $C_2 = 8$ :

$$7 \log n + 2 \leq 8 \log n$$

$$2 \leq \log n$$

Solve:

$$\log n \geq 2$$

$$n \geq 4$$

So for  $n \geq 4$ , inequality holds.

Thus:

→  $C_2 = 8$

→  $n_0 = 4$

---

## 🎉 Final Conclusion for Example 2

For  $n \geq 4$ :

$$6n \log n \leq 7n \log n + 2n \leq 8n \log n$$

So:

✓  $F(n) = \Theta(n \log n)$

---

## 🎯 Final Summary

Function $F(n)$	$\Theta$ Bound	Why?
$3n^2 + 8n + 10$	$\Theta(n^2)$	$n^2$ is dominant term
$7n \log n + 2n$	$\Theta(n \log n)$	$n \log n$ grows faster than $n$

---

## Big Theta ( $\Theta$ ) Notation

Big Theta tells us the **EXACT growth rate** of an algorithm.

Not too fast.

Not too slow.

**Just right.**

---



## The Sandwich Rule (Formal Definition)

We want to “sandwich”  $F(n)$  between two functions:

Lower Bound:  $C_1 \times g(n)$

Actual:  $F(n)$

Upper Bound:  $C_2 \times g(n)$

The rule:

$$C_1 \times g(n) \leq F(n) \leq C_2 \times g(n) \quad \text{for all } n \geq n_0$$

If we can find such constants, then:

$$F(n) = \Theta(g(n))$$

---



## Example: $F(n) = 5n + 4$

Goal: Prove that:

$$5n + 4 = \Theta(n)$$

---

### Step 1: Find Upper Bound ( $C_2$ )

We check:

$$5n + 4 \leq C_2 \times n$$

Choose:

$$C_2 = 6$$

Check:

$$\begin{aligned} 5n + 4 &\leq 6n \\ 4 &\leq n \quad (\text{true for } n \geq 4) \end{aligned}$$

So upper bound holds.

---

## Step 2: Find Lower Bound ( $C_1$ )

We check:

$$5n + 4 \geq C_1 \times n$$

Try:

$$C_1 = 1$$

Check:

$$\begin{aligned} 5n + 4 &\geq n \\ 4n + 4 &\geq 0 \quad (\text{true for all } n \geq 1) \end{aligned}$$

Lower bound holds.

---

## Step 3: Final Sandwich

For all  $n \geq 4$ :

$$1n \leq 5n + 4 \leq 6n$$

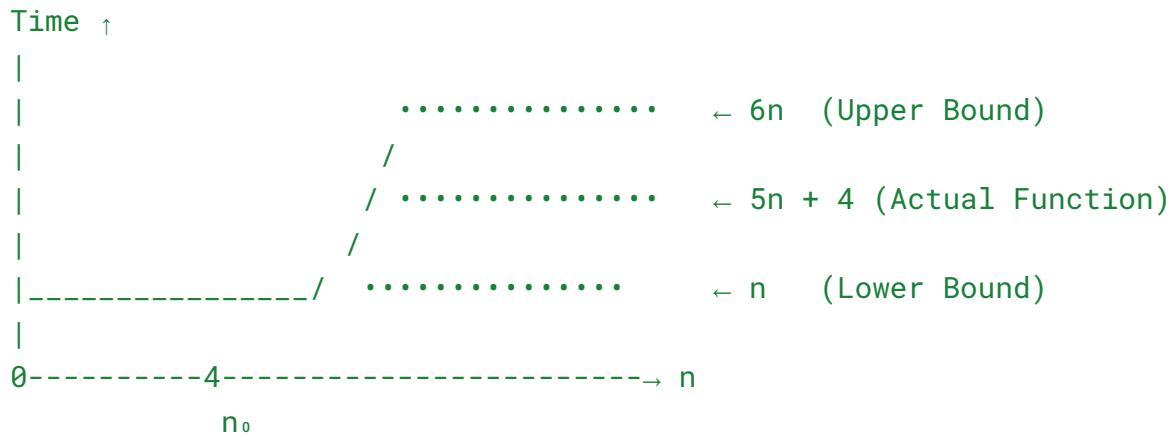
Proven:

$$5n + 4 = \Theta(n)$$

---



## Visual Intuition (Text Graph)



$F(n)$  stays between  $n$  and  $6n$  — this is  $\Theta(n)$ .

---



## Meaning of $\Theta(n)$

When we say:

$$F(n) = \Theta(n)$$

We are saying:

**The algorithm grows exactly linearly —  
not faster (like  $n^2$ ) and not slower (like  $\sqrt{n}$ ).**

---

# Difference Summary

Big-O = Won't grow faster than  $g(n)$   
Big- $\Omega$  = Won't grow slower than  $g(n)$   
Big- $\Theta$  = Grows exactly like  $g(n)$

---



## When Do We Use Big Theta?

Only when **best case and worst case are the same order**.

---

### Example 1: Sum of Array — ALWAYS $\Theta(n)$

```
def sum_array(arr):
    total = 0
    for num in arr:      # Always runs exactly n times
        total += num
    return total
```

No early exit → Best = Worst =  $n \rightarrow \Theta(n)$

---

### Example 2: Linear Search — NOT $\Theta(n)$

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i      # Can exit early
    return -1
```

Best case = 1

Worst case =  $n$

Different orders → NOT  $\Theta(n)$

You only say:

Worst case:  $O(n)$

Best case:  $\Omega(1)$

---

## Quick Guide

If an algorithm ALWAYS takes  $n$  steps  $\rightarrow \Theta(n)$

If sometimes 1 step, sometimes  $n$  steps  $\rightarrow$  NOT  $\Theta(n)$

If upper and lower bounds match  $\rightarrow \Theta(g(n))$

# Summary

## ★ Practical Significance of Asymptotic Notations

(Clean, Structured, PDF-Friendly Version)

---

## 1 Big-O Notation — WORST CASE (Most Important in Real Life)

### Meaning:

“Maximum time the algorithm can take.”

### Why we care:

Systems must survive **peak load**, **emergencies**, and **no-luck scenarios**.

### Example: Hospital System

```
def find_available_doctor(doctors):
    for doctor in doctors:      # WORST CASE: check ALL doctors
```

```
if doctor.is_available:  
    return doctor  
return None           # All doctors busy
```

Worst-case scenario:

- All doctors busy
- Loop checks every single doctor
- System must still respond safely

**We always plan for this case.**

---

## 2 Big-Omega ( $\Omega$ ) — BEST CASE

**Meaning:**

“Minimum time the algorithm can take.”

**Why it’s rarely used:**

We cannot build systems based on **luck**.

**Example: Same Hospital System**

```
def find_available_doctor(doctors):  
    if doctors[0].is_available:  # Best case: first doctor free  
        return doctors[0]       #  $\Omega(1)$ 
```

This is not reliable, so we do **not** use  $\Omega$  for planning.

---

## 3 Big-Theta ( $\Theta$ ) — EXACT Growth Rate

**Meaning:**

“Time **ALWAYS** grows at this rate.”

(Best case = Worst case)

## Example

```
def calculate_total_salary(employees):  
    total = 0  
    for emp in employees:      # ALWAYS process every employee  
        total += emp.salary  
    return total
```

This is always linear →  $\Theta(n)$ .

---

## Linear Search — All Three Notations Clearly

```
def linear_search(arr, target):  
    for i in range(len(arr)):  
        if arr[i] == target:  
            return i  
    return -1
```

Case	Scenario	Time	Notation
Best	Target at first position	1	$\Omega(1)$
Worst	Target at last position or absent	n	$O(n)$
Average	Target somewhere in middle	n/2	$\Theta(n)$

Why  $\Theta(n/2)$  becomes  $\Theta(n)$ ?

Because constants are ignored in asymptotic notation.

---



## Order of Growth (Fastest → Slowest)

(Plain text list for perfect PDF formatting)

$O(1)$	Constant	Example: Array access
$O(\log n)$	Logarithmic	Example: Binary search
$O(n)$	Linear	Example: Loop through list
$O(n \log n)$	Linearithmic	Example: Merge sort
$O(n^2)$	Quadratic	Example: Nested loops
$O(2^n)$	Exponential	Example: All subsets

---



## Key Practical Insights

### ✓ Always use Big-O (worst case) for planning

Systems must not break during heavy load.

### ✓ Use Big-Theta when performance is predictable

(Example: Summing an array)

### ✓ Big-Omega is rarely useful

Too optimistic for real systems.

---

## Why Big-O is the KING in Industry

```
# Thinking like a business
if we_use_best_case_planning:      # Dangerous!
    system_might_crash()
else:
    plan_using_worst_case()        # Safe!
    system_survives_peak_load()
```

### Real Example: Netflix

- Best case: few viewers → easy

- Worst case: millions join at same time → servers must be ready

Netflix plans for the worst case.

---

## **BOTTOM LINE**

Hope for the best ( $\Omega$ )  
But **ALWAYS** plan for the worst ( $O$ )

In interviews and industry — Big-O is your best friend.

---

## **Space Complexity — Explained Simply**

Space Complexity = How much **MEMORY** your algorithm uses.

It tells us **how much extra space** (RAM) an algorithm needs *while running*.

---

## **Memory Basics — How Much Space Do Data Types Use?**

Data Type	Typical Size	Example
boolean	1 byte	True / False
byte	1 byte	Small integers
char	2 bytes	'A', 'b'
int	4 bytes	10, -500
float	4 bytes	3.14

long/double 8 bytes      Larger  
numbers

---



## Memory Required for Arrays

Array Type	Memory Formula	Example (n = 100)
------------	----------------	-------------------

char[]	$2 \times n$ bytes	$2 \times 100 = 200$ bytes
--------	--------------------	-------------------------------

int[]	$4 \times n$ bytes	$4 \times 100 = 400$ bytes
-------	--------------------	-------------------------------

double[]	$8 \times n$ bytes	$8 \times 100 = 800$ bytes
----------	--------------------	-------------------------------

int[][]	$4 \times n \times m$ bytes	$4 \times 10 \times 10 = 400$ bytes
---------	-----------------------------	--

Arrays always take space proportional to their **size**.

---



## Example 1: Simple Sum Algorithm

```
def sum_numbers(n):      # n is double (8 bytes)
    total = 0            # int (4 bytes)
    i = 0                # int (4 bytes)
    while i <= n:
        total += i
        i += 1
    return total
```

### Memory Calculation

```
total (int) = 4 bytes
i (int)     = 4 bytes
n (double)  = 8 bytes
```

TOTAL = 16 bytes

This memory is **constant**, no matter the input size.

### Space Complexity = O(1) (Constant Space)

---

## Example 2: Processing a 2D Matrix

```
def process_matrix(matrix, n, m): # n, m are int (4 bytes each)
    total = 0                      # int (4 bytes)

    for i in range(n):              # i = 4 bytes
        for j in range(m):          # j = 4 bytes
            total += matrix[i][j]

    return total
```

### Memory Breakdown

total (int)	= 4 bytes
n, m (int)	= 4 + 4 = 8 bytes
i (int)	= 4 bytes
j (int)	= 4 bytes

### BUT the array is the BIG part

Assuming `matrix` is `double[][]`:

`matrix = 8 × n × m bytes`

### Total Space =

Fixed part = 16 bytes

Variable part =  $8 \times n \times m$  bytes



## Space Complexity = $O(n \times m)$

(Depends on matrix size)

---



# Key Points About Space Complexity

## 1 Fixed vs Variable Space

### ✓ Fixed Space:

Variables like `total`, `i`, `j` →  $O(1)$

### ✓ Variable Space:

Arrays →  $O(n)$ ,  $O(n^2)$ , etc.

---

## 2 Big-O Space Complexity Examples

### $O(1)$ — Constant Space

```
def example1(n):
    a = 1
    b = 2
    return a + b
```

### $O(n)$ — Linear Space

```
def example2(n):
    arr = [0] * n      # takes n × 4 bytes
    return arr
```

### $O(n^2)$ — Quadratic Space

```
def example3(n):
    matrix = [[0] * n for _ in range(n)]  # n² entries
    return matrix
```

---

## 3 Why Space Complexity Matters

- **Mobile apps:** limited RAM
  - **Servers:** millions of users → memory adds up
  - **Big data:** entire dataset won't fit in memory
  - **AI systems:** huge models need optimized memory usage
- 



## Quick Rules to Remember

- Single variables →  $O(1)$
  - Array of size  $n$  →  $O(n)$
  - 2D array  $n \times m$  →  $O(n \times m)$
  - Recursion →  $O(\text{depth})$  stack space
- 



## Real-World Example: Instagram Photo Processing

```
def process_photo(photo):  
    pixels = load_photo(photo)      # O(width * height)  
    filters = apply_filters(pixels) # extra temporary buffers  
    return compressed_photo
```

Millions of large photos → memory optimization becomes critical.

---



## Bottom Line

**Time Complexity = How FAST an algorithm runs**

**Space Complexity = How MUCH MEMORY it needs**

Both are essential for writing efficient algorithms. ⚡