
🧠 TIME & SPACE COMPLEXITY – COMPLETE NOTES

🌱 1. What is Time Complexity?

Definition:
Time Complexity is the amount of time a program or algorithm takes to complete as a function of the input size **n**.

Human analogy:
In daily life, if baking bread takes you 30 minutes, you measure time based on the size of the job. Similarly, when a computer solves a problem, we measure how much time it takes based on the number of operations it performs.

⚙️ 2. Why Time Complexity Matters

- If a human takes 1 hour and a machine takes 5 hours — that's a useless machine.
- So, we analyze **how efficient** our code or algorithm is — not in seconds (hardware-dependent), but in **number of operations** (machine-independent).

🧩 3. Input Size n

- Represents how many elements the algorithm will handle.
  Example: In an array, n = number of elements.
  It could be 10, 10,000, or 10 million — we generalize it as **n**.

🕒 4. Measuring Time Complexity

Two main approaches:

Approach	Description
Analytical (Procedure-based)	Understand the steps of your logic, count operations manually.
Code-based	Look at loops and recursive calls in your program to infer the complexity.

In practice, we use both.

🔁 5. Basic Loop Patterns and Their Time Complexities

Case 1: Single Loop

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

```
  // do something
}
```

→ **Order(n)**
Every element is processed once.

In Python:

```
for i in range(n):
    pass
```

Also **O(n)**.
The interpreter still goes through each element once.

Case 2: Nested Loop

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

Each element triggers another full pass through all elements.
→ **Order(n^2)**

In Python:

```
for i in range(n):
    for j in range(n):
        pass
```

Still **O(n^2)** — Python just hides the type declarations, not the cost.

Case 3: Reducing Iterations (Triangular Pattern)

When you process the rest of the list for each element:

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

Number of comparisons:
$(n-1) + (n-2) + (n-3) + \ldots + 1 = (n^2 - n)/2$

→ **Order(n^2)** again (the degree of the polynomial decides the order).

Python equivalent:

```
for i in range(n):
    for j in range(i+1, n):
        pass
```

Case 4: Dividing by 2 each time (Logarithmic)

```
for(i = n; i > 1; i = i / 2) {
  // process
}
```

Here, the number of iterations reduces by half each time:
n → n/2 → n/4 → n/8 → ... → 1

Number of steps = **$\log_2(n)$**
→ **Order(log n)**

Python equivalent:

```
i = n
while i > 1:
    i = i // 2
```

Also **O(log n)** — very common in binary search and tree traversal.

📚 Common Time Complexities and Their Growth

Complexity	Example	Behavior
O(1)	Accessing an array element	Constant
O(log n)	Binary search	Very fast
O(n)	Linear search	Grows linearly
O(n log n)	Merge sort, quick sort (avg)	Efficient
O(n^2)	Nested loops, bubble sort	Slower
O(2^n)	Recursion like Fibonacci	Explodes quickly

Complexity	Example	Behavior
O(n!)	Permutations, brute force	Nightmare mode

🧮 6. Understanding Through Data Structures

a) Array

- Access: O(1)
- Traversal (visit every element): O(n)
- Nested traversal (compare each with each): $O(n^2)$

b) Linked List

- Similar to array in time complexity:
  - Traversal: O(n)
  - Search: O(n)
  - Space: O(n) + links (still O(n))

In Python, a list behaves more like a dynamic array — **O(1)** access but costly insertions in the middle (O(n)).

c) Matrix

- Dimensions = n × n → total n^2 elements.
- Full traversal = $O(n^2)$
- If you nest another loop (3 total), that's $O(n^3)$.

```
for i in range(n):
  for j in range(n):
    for k in range(n):
      pass  # O(n³)
```

d) Array of Linked Lists

- Suppose an array of size m, and each element is a list with n elements.
  Total processing: m + n → **Order(m + n)**
  Often simplified to **O(n)** if m and n are of similar scale.

e) Binary Tree

- If traversing only along height: O(log n)
- If processing all nodes: O(n)

In Python:

```
def inorder(node):
```

```
  if node:
    inorder(node.left)
    inorder(node.right)
```

Each node visited once → **O(n)**

📈 7. Space Complexity

Definition:
The amount of memory an algorithm needs to run, as a function of input size **n**.

We care about:

- Memory for variables
- Auxiliary data structures
- Function call stacks (recursion)

Examples:

Structure	Space Used	Space Complexity
Array of n elements	n	O(n)
Linked List	n nodes + n links ≈ 2n	O(n)
Matrix (n×n)	n^2	$O(n^2)$
Array of linked lists	m + n	O(m + n)
Binary tree with n nodes	n	O(n)

In Python, everything (even integers) is an object on the heap, so there's always a small constant overhead — but asymptotically, it behaves the same.

⚖️ 8. Time vs Space Trade-off

- Sometimes you use **more memory to save time** (like precomputed hash maps).
- Other times, you use **more time to save memory** (like streaming large files line by line).

Python often chooses convenience over micro-optimization, but as a developer, you must still know **which algorithms are efficient** for large data.

🔍 9. Key Takeaways

1. **Order(n)** means the time grows linearly with the number of inputs.
2. **Order(n^2)** means nested operations — be cautious.
3. **Order(log n)** means divide-and-conquer style efficiency.
4. Always measure growth, not seconds — hardware changes, math doesn't.
5. **Python code** may look simple, but behind the scenes it obeys these same mathematical rules.

Code Pattern	Time Complexity	Space Complexity
Simple for loop	O(n)	O(1)
Nested loops	$O(n^2)$	O(1)
Divide by 2 each iteration	O(log n)	O(1)
Full matrix traversal	$O(n^2)$	$O(n^2)$
Recursive tree traversal	O(n)	O(h) (stack height)
Binary search	O(log n)	O(1)
Merge sort	O(n log n)	O(n)