**1. Time Complexity**

**✅ What is it?**

**Time complexity** describes **how the runtime of an algorithm grows** as the size of the input increases.

It tells you **how fast** your algorithm is.

**📈 Common Time Complexities (From Fastest to Slowest):**

| **Complexity** | **Description** | **Example** |
| --- | --- | --- |
| O(1) | Constant time – doesn't depend on input size | Accessing element in array by index |
| O(log n) | Logarithmic – input gets divided each step | Binary search |
| O(n) | Linear – grows directly with input size | Looping over an array |
| O(n log n) | Log-linear – divide and conquer with merge/sort | Merge Sort, Quick Sort |
| O(n²) | Quadratic – nested loops | Brute force for Two Sum |
| O(2ⁿ) | Exponential – very slow | Recursive Fibonacci |
| O(n!) | Factorial – worst cases | Solving permutations |

**🔍 Example: Two Sum**

python

CopyEdit

# Brute Force

for i in range(n):

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

# total = nums[i] + nums[j]

Here we have two nested loops → for each i, j runs → O(n²) time complexity.

**🧠 How to Analyze:**

* Look at loops: each loop = multiply time
* Recursive calls: count how they branch
* Ignore constants and low-order terms in Big O
  + E.g. O(2n + 3) → becomes O(n)

**📦 2. Space Complexity**

**✅ What is it?**

**Space complexity** measures **how much memory** (RAM) your algorithm uses with respect to input size.

It includes:

* Variables
* Data structures (lists, sets, dicts)
* Call stack (for recursion)

**🔍 Common Examples:**

| **Code** | **Space Complexity** |
| --- | --- |
| a = 5 | O(1) – one variable |
| arr = [1, 2, 3] | O(n) – size depends on input |
| Recursive calls | Stack size = O(n) |

**🔍 Example: Two Sum with HashMap**

num\_map = {} # dictionary

for num in nums:

num\_map[num] = i

* You are storing each element → Space = O(n)

**🧠 Real-Life Analogy:**

**Time Complexity:**

Like the **amount of time it takes** to find a book in a library:

* If it's sorted by genre → faster (O(log n))
* If it's thrown everywhere → slower (O(n) or O(n²))

**Space Complexity:**

Like the **amount of shelf space or paper** you need while doing a task.

**✅ Summary Table:**

| **Complexity Type** | **Measures...** | **Goal** |
| --- | --- | --- |
| Time Complexity | Execution time | Make algorithm faster |
| Space Complexity | Memory consumption | Use less memory |

Visual diagrams

**📊 1. Time Complexity Visual Chart**

**📈 Growth of Common Time Complexities (as input n increases)**

^

| O(2^n)

| O(n!)

| \*

| \*

| \*

| \* O(n^2)

| \*

| \* O(n log n)

| \*

| \* O(n)

|\* O(log n)

|\_\_

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_>

Input size (n)

**🔍 Interpretation:**

* **O(1):** Constant time (flat line)
* **O(log n):** Grows slowly (binary search)
* **O(n):** Linear growth (loop once over array)
* **O(n log n):** Sorting algorithms
* **O(n²):** Nested loops (like brute-force Two Sum)
* **O(2ⁿ)/O(n!):** Recursive algorithms (combinations, permutations)

**🧠 2. Space Complexity Examples with Diagrams**

def sum\_nums(nums):

total = 0

for num in nums:

total += num

return total

**Memory Usage:**

* Only total variable

✅ Constant space → O(1)

**📦 Example 2: O(n) Space – Storing Elements**

def reverse(nums):

return nums[::-1]

**Memory Usage:**

* Original list nums
* New reversed list

[ 1, 2, 3, 4 ] → [ 4, 3, 2, 1 ]

↑ ↑

input output

✅ Space grows with input size → O(n)

**📦 Example 3: O(n) Time and Space – HashMap (Two Sum)**

num\_map = {}

for i, num in enumerate(nums):

complement = target - num

if complement in num\_map:

return [num\_map[complement], i]

num\_map[num] = i

**Diagram:**

nums = [2, 7, 11, 15], target = 9

Step 1:

num\_map = { 2: 0 }

Step 2:

complement = 9 - 7 = 2

→ Found in map ✅ → return [0, 1]

* HashMap grows with input → O(n) space
* Looping once → O(n) time

**🎓 Recap Table with Visual Icons**

| **Complexity** | **Graph Shape** | **Example** | **Notes** |
| --- | --- | --- | --- |
| O(1) | ▬▬▬▬▬▬▬▬ | arr[0] | Constant, fast always |
| O(log n) | slowly rising curve ↗️ | Binary Search | Divide & conquer |
| O(n) | straight line 📈 | Single loop | Linear growth |
| O(n log n) | curve between 📈 and ⬆️ | Merge Sort | Fast sorting |
| O(n²) | steep curve ⬆️ | Nested loops | Brute force comparison |
| O(2ⁿ) | exponential 🚀 | Recursive subsets | Very slow |

**🔚 Summary**

* **Visual growth curves** help predict performance on large input sizes.
* **Flat = fast**, **steep = slow**.
* Always aim for **O(n)** or better if possible.
* Understand both **how long** (time) and **how much memory** (space) an algorithm uses.