TIME & SPACE COMPLEXITY

📘 Time and Space Complexity – Quick Notes

🚀 What is Time Complexity?

Time complexity describes **how the runtime of an algorithm grows** as the input size increases. It is usually expressed using **Big O notation**.

Time Complexity	Description	Example
O(1)	Constant Time	Accessing an array element
O(log n)	Logarithmic Time	Binary search
O(n)	Linear Time	Looping through an array
O(n log n)	Linearithmic Time	Merge Sort, Quick Sort (avg)
$O(n^2)$	Quadratic Time	Nested loops (Bubble Sort)
$O(2^n)$	Exponential Time	Recursive Fibonacci
O(n!)	Factorial Time	Permutations/Combinations

⏱️ Worst, Best, and Average Case

- **Worst Case (Big O):** Maximum time taken
- **Best Case (Ω):** Minimum time taken

- **Average Case (Θ):** Average time taken over all inputs

🧠 What is Space Complexity?

Space complexity refers to the **amount of memory** used by an algorithm in terms of the size of input.

Space Complexity	Description
O(1)	Constant space (no extra memory)
O(n)	Linear space (e.g., storing inputs in array)
$O(n^2)$	For 2D arrays or matrix operations

🔄 Common Operations and Their Time Complexities

Operation	Time Complexity
Array access (indexing)	O(1)
Insert/delete at end (array)	O(1)
Insert/delete at start (array)	O(n)
Searching in array (unsorted)	O(n)
Searching in array (sorted)	O(log n)
Hash map get/put	O(1) avg / O(n) worst
Binary Search Tree (balanced)	O(log n)
BFS / DFS (graph traversal)	O(V + E)

📚 Tips to Improve Time & Space Complexity

- Use **hash maps** for faster lookup.
- Apply **divide and conquer** strategies like Merge Sort.
- Avoid nested loops if possible; use pre-processing (prefix sums, etc.).
- Use **in-place algorithms** to save space.

📎 Examples

```
python
CopyEdit
# O(n) Time, O(1) Space - Sum of Array
def array_sum(arr):
    total = 0
    for num in arr:
        total += num
    return total

# O(log n) Time - Binary Search
def binary_search(arr, target):
    low, high = 0, len(arr)-1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

📦 Tools for Complexity Analysis

- **Big-O Cheat Sheet:** https://www.bigocheatsheet.com
- Python: Use `time` and `memory_profiler` for profiling
- Visualizer: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

---

Tricks for Time Complexity

✅ Tricks to Identify Time Complexity

1. Count the Loops

- **Single loop running `n` times →** `O(n)`
- **Nested loops each running `n` times →** `O(n²)`
- **Nested loop with inner loop halving the input →** `O(n log n)`

```
python
CopyEdit
for i in range(n):          # O(n)
    for j in range(n):      # O(n)
        print(i, j)         # → O(n²)
```

---

2. Look for Dividing Input (Binary Search Pattern)

- If the input size is **halved each iteration**, the complexity is **`O(log n)`**.

```
python
CopyEdit
while low <= high:          # Halves each time → O(log n)
    mid = (low + high) // 2
```

---

3. Recursive Pattern

Use the **recurrence relation**:

- `T(n) = T(n/2) + O(1)` → `O(log n)`
- `T(n) = 2T(n/2) + O(n)` → `O(n log n)`
- `T(n) = T(n-1) + O(1)` → `O(n)`

Use the **Master Theorem** for recursive algorithms.

---

4. Hash Table/Set/List Tricks

- Hashing operations are often `O(1)` on average.
- Watch out for operations like `in` on lists → `O(n)`

```
python
CopyEdit
myset = set(arr)  # O(n) to build set
if x in myset:    # O(1) lookup
```

5. Check for Early Returns / Breaks

Sometimes loops return early:

```
python
CopyEdit
for i in range(n):   # Worst case O(n), best case O(1)
    if arr[i] == target:
        return i
```

6. Mathematical Series or Formulas

- Summing from 1 to n: `O(n)`
- Nested loops with increasing ranges:

```
python
CopyEdit
for i in range(n):
    for j in range(i, n):   # Triangular pattern → O(n²)
```

7. Amortized Complexity

- Example: Appending to a Python list is **O(1)** on average due to **dynamic resizing**.

🔍 Time Complexity Estimation Template

Pattern/Structure	Time Complexity
Single `for` loop	`O(n)`
Two nested `for` loops	`O(n²)`
`while` halving n	`O(log n)`
Divide & Conquer (e.g., Merge Sort)	`O(n log n)`
Recursive factorial	`O(n!)`
Using heap or priority queue	`O(log n)` per op

⚡ Quick Time Complexity Heuristics

Code Snippet Type	Likely Time Complexity
Loop over array once	$O(n)$
Nested loop over entire array	$O(n^2)$
Binary search	$O(\log n)$
Merge sort / Quick sort (avg)	$O(n \log n)$
Fibonacci (naive recursion)	$O(2^n)$
Hash map operations	$O(1)$ avg, $O(n)$ worst

🧠 Pro Tip

If you have a code snippet and are unsure, just follow these steps:

1. **Identify loops and recursion.**
2. **Check how the input size `n` changes (halved, constant, growing).**
3. **Use known patterns (search, sort, recursion tree).**
4. **Look for break conditions or early returns.**