TIME COMPLEXITY

What is Time Complexity?

Time complexity measures how the runtime of an algorithm grows with input size **n**. It helps analyze the efficiency of algorithms.

Introduction to Time Complexity

Time complexity measures how the runtime of an algorithm increases as the input size grows. It helps in understanding the efficiency of an algorithm.

Why Time Complexity?

- Helps compare different algorithms.
- Predicts performance before execution.
- Essential for competitive programming & real-world applications.

Types of Time Complexities

Different time complexities are classified based on their growth rate:

Constant Time - O(1)

The algorithm runs in the same amount of time, regardless of input size.

Example:

```
# Accessing an element in an array (Indexing is O(1))
def get_first_element(arr):
    return arr[0]
```

Logarithmic Time - O(log N)

The execution time increases logarithmically as input size grows.

Example:

- **Binary Search (Divide & Conquer)**

```
import math
n = 16
print(math.log2(n))  # Output: 4 (log₂16 = 4)
```

Linear Time - O(N)

The execution time increases proportionally to the input size.

Example:

```
# Traversing an array (O(N))
def print_elements(arr):
    for item in arr:
        print(item)
```

Linearithmic Time - O(N log N)

Used in efficient sorting algorithms.

Example:

- **Merge Sort, Quick Sort (Average Case)**

```
# Sorting an array (O(N log N))
arr = [5, 2, 9, 1]
arr.sort()
```

Quadratic Time - $O(N^2)$

Used in brute force algorithms.

Example:

- **Bubble Sort, Selection Sort, Insertion Sort**

```
# Nested loops (O(N²))
def pairwise_combinations(arr):
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            print(arr[i], arr[j])
```

Cubic Time - $O(N^3)$

Occurs when there are three nested loops.

Example:

```
# Triple nested loops (O(N³))
def cubic_function(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                print(i, j, k)
```

Exponential Time - O(2^N)

Grows exponentially, worst for large inputs.

Example:

- **Fibonacci using Recursion**

```
# Fibonacci (O(2^N))
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Factorial Time - O(N!)

Occurs in brute force problems like the Traveling Salesman Problem (TSP).

Example:

```
# Generating all permutations (O(N!))
from itertools import permutations
arr = [1, 2, 3]
print(list(permutations(arr)))
```

Best, Average, and Worst Case Complexity

Different cases arise based on input distribution:

Case	Definition	Example
Best Case (Ω)	Minimum time taken	Already sorted array for Bubble Sort (O(N))
Average Case (Θ)	Expected time taken	Random input distribution
Worst Case (O)	Maximum time taken	Reverse sorted array for Bubble Sort ($O(N^2)$)

Space Complexity & Auxiliary Space

Space Complexity kya hoti hai?

Space Complexity **ek algorithm ke execution ke dauraan use hone wali total memory** ko represent karti hai. Isme **input size (N)** ke sath **memory consumption ka relation** dekha jata hai.

Space Complexity ka Formula

Total Space Complexity = **Fixed Part (Constant Space) + Variable Part (Dynamic Space)**

- **Fixed Part:** Constants, Code Space, and Static Variables → **O(1)**
- **Variable Part:** Input Variables, Function Calls, Auxiliary Space → **Depends on N**

Auxiliary Space vs. Space Complexity

Space Complexity: Algorithm ke execution ke dauraan total memory usage.

Auxiliary Space: Extra space jo algorithm use karta hai (excluding input storage).

Concept	Definition	Example
Space Complexity	Algorithm ke total memory usage	Input + Extra Space
Auxiliary Space	Sirf extra memory (excluding input)	Recursion Stack, Temporary Arrays

Example: Merge Sort

- **Total Space Complexity:** O(N) (because it uses extra arrays for merging)
- **Auxiliary Space:** O(N) (because it requires additional memory apart from input storage)

Example: Quick Sort

- **Total Space Complexity:** O(N) (in worst case recursion stack)
- **Auxiliary Space:** O(log N) (average case recursion depth)

Space Complexity ki Different Cases

Constant Space - O(1)

Jab algorithm ka memory usage **input size se independent hota hai**.

Example: Variable assignment, swapping, loop counters.

```
def swap(a, b):
    temp = a   # O(1) space
    a = b
    b = temp
    return a, b
```

Linear Space - O(N)

Agar algorithm ko **N elements ke liye extra memory chahiye**.

Example: Creating a new array, storing visited elements.

```
def store_elements(n):
    arr = []   # O(N) space
    for i in range(n):
        arr.append(i)
    return arr
```

Logarithmic Space - O(log N)

Jab **recursive calls ka depth logarithmic ho** (like binary recursion).

Example: Binary Search, QuickSort.

```
def binary_search(arr, low, high, x):
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == x:
        return mid
    elif arr[mid] < x:
        return binary_search(arr, mid + 1, high, x)  # O(log N) recursion stack
    else:
        return binary_search(arr, low, mid - 1, x)
```

Quadratic & Higher Space - $O(N^2)$, $O(2^n)$

Agar algorithm ko input size ke square ya exponential memory chahiye.

Example: Matrix-based DP, Brute-force Recursion.

```
def all_subsets(arr):
    # Exponential space complexity (O(2^N))
    from itertools import combinations
    subsets = []
    for i in range(len(arr) + 1):
        subsets.extend(combinations(arr, i))
    return subsets
```

Space Complexity in Different Data Structures

Data Structure	Space Complexity
Array	O(N)
Linked List	O(N)
Stack (n elements)	O(N)
Queue (n elements)	O(N)
Hash Table	O(N)
Binary Tree (balanced)	O(N)
Binary Tree (skewed)	O(N)
Heap (n elements)	O(N)
Graph (Adj List)	O(V + E)

Space Complexity in Recursion & Iteration

Algorithm	Recursive Space Complexity	Iterative Space Complexity
Factorial (Recursive)	O(N) (Call Stack)	O(1)
Fibonacci (Recursive)	O(N) (Call Stack)	O(1)
Merge Sort	O(N) (Extra array)	O(N)
Quick Sort	O(log N) (Recursion depth)	O(1) (In-place)
DFS (Recursion)	O(H) (Tree height)	O(N) (Stack for explicit DFS)

Optimizing Space Complexity

🔷 **In-place Algorithms:** Extra memory use na kare. (Example: Quick Sort)

🔷 **Bit Manipulation:** Integers store karne ke liye bitwise operators ka use.

🔷 **Sliding Window & Two-Pointer Approach:** Space-efficient searching/sorting.

🔷 **Recursion to Iteration Conversion:** Stack Overflow avoid karne ke liye iterative approach prefer karein.

Types of Time Complexity

Complexity	Growth Rate	Example
O(1)	Constant Time	Accessing an element in an array
O(log n)	Logarithmic Time	Binary Search
O(n)	Linear Time	Traversing an array
O(n log n)	Linearithmic Time	Merge Sort, Heap Sort
O(n^2)	Quadratic Time	Bubble Sort, Selection Sort
O(2^n)	Exponential Time	Recursion in Fibonacci
O(n!)	Factorial Time	Traveling Salesman Problem

Good vs Bad Complexity

✅ **Preferred Complexities**	❌ **Avoid If Possible**
O(1), O(log n), O(n)	O(n^2), O(2^n), O(n!)
O(n log n) (sorting)	Brute-force solutions
Efficient recursive approaches	Inefficient nested loops

Competitive Programming (CP) Time Complexity Constraints

Input Size (n)	**Preferred Complexity**
n ≤ **12**	O(n!) or O(2^n) (Brute Force)
n ≤ **25**	O(2^n * n) (Bit mask DP)
n ≤ **1000**	O(n^2) (Quadratic Algorithms)

$n \leq 10^5$	O(n log n) (Sorting, Efficient DP)
$n \leq 10^6$	O(n), O(log n) (Linear or Logarithmic)
$n > 10^8$	O(1) (Constant Time Solution)

Time Complexity Cheat Sheet for Data Structures

Array

Operation	Best Case	Worst Case
Access	O(1)	O(1)
Search	O(1)	O(n)
Insert (End)	O(1)	O(1)
Insert (Middle)	O(1)	O(n)
Delete (End)	O(1)	O(1)
Delete (Middle)	O(1)	O(n)

Linked List

Operation	Best Case	Worst Case
Access	O(1)	O(n)
Search	O(1)	O(n)
Insert (Head)	O(1)	O(1)
Insert (Middle)	O(1)	O(n)
Delete (Head)	O(1)	O(1)
Delete (Middle)	O(1)	O(n)

Stack

Operation	Best Case	Worst Case
Push	O(1)	O(1)
Pop	O(1)	O(1)

Peek	O(1)	O(1)

Queue

Operation	Best Case	Worst Case
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)

Binary Search Tree (BST)

Operation	Best Case	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Hash Table (HashMap, HashSet)

Operation	Best Case	Worst Case
Insert	O(1)	O(n) (hash collisions)
Search	O(1)	O(n)
Delete	O(1)	O(n)

Heap (Min Heap / Max Heap)

Operation	Best Case	Worst Case
Insert	O(1)	O(log n)
Delete (Heapify)	O(1)	O(log n)
Get Min / Max	O(1)	O(1)

Graph Algorithms

Algorithm	Time Complexity
BFS / DFS	O(V + E)
Dijkstra's Algorithm	O((V + E) log V)

Floyd-Warshall	$O(V^3)$
Kruskal’s Algorithm	O(E log E)
Prim’s Algorithm	O(E + V log V)