Time Complexity

1 Concept of Time Complexity

Time complexity is a concept in computer science that deals with the efficiency of algorithms. When we write a computer program, we often want to know how long it will take to run. However, measuring the exact time in seconds is not reliable because different computers have different speeds.

Instead of seconds, time complexity measures the **number of operations** an algorithm performs relative to the size of the input. We denote the input size as n. For example, if we are sorting a list of 100 numbers, $n = 100$.

In simple terms: Time Complexity helps us predict how the running time increases as the input size (n) gets larger.

2 Types of Time Complexity

When analyzing an algorithm, the running time can vary depending on the arrangement of the data. We usually look at three scenarios:

2.1 Best Case Analysis

This describes the scenario where the algorithm runs the fastest. It occurs when the input is already in an ideal format. For example, if we are searching for a number in a list and we find it in the very first position, that is the best case. It is often denoted by the Greek letter Omega (Ω).

2.2 Average Case Analysis

This represents the expected running time for a random input. Calculating the average case requires a probability analysis of all possible inputs. It gives a realistic expectation of performance in typical day-to-day use.

2.3 Worst Case Analysis

This describes the scenario where the algorithm takes the longest possible time to finish. It occurs when the input is in the most difficult format (e.g., trying to sort a list that is exactly in reverse order).

Note: In computer science, we focus mostly on the **Worst Case** (denoted by Big O notation, $\mathcal{O}$) because it gives us a guarantee. We know the algorithm will never take longer than this limit.

3 Common Time Complexity Classes

We group algorithms into ”classes” based on how fast they grow. Here they are listed from fastest to slowest:

3.1 Constant Time – $\mathcal{O}(1)$

The running time does not change, no matter how much data you have.

- Example: Accessing the 5th element of an array. It takes the same time whether the array has 10 items or 1,000,000 items.

3.2 Logarithmic Time – $\mathcal{O}(\log n)$

The time grows very slowly. Usually, this happens when the problem size is cut in half at every step.

- Example: Finding a page in a phone book by constantly splitting the book in half (Binary Search).

3.3 Linear Time – $\mathcal{O}(n)$

The time grows directly in proportion to the input. If you double the data, you double the time.

- Example: Reading every page of a book one by one.

3.4 Linearithmic Time – $\mathcal{O}(n \log n)$

This is slightly slower than linear time but much faster than quadratic time. Most efficient sorting algorithms fall into this category.

- Example: Merge Sort or Quick Sort (average case).

3.5 Quadratic Time – $\mathcal{O}(n^2)$

The time grows proportionally to the square of the input. This is common in algorithms with nested loops.

- Example: Comparing every person in a room with every other person.

3.6 Cubic Time – $\mathcal{O}(n^3)$

The time grows proportionally to the cube of the input. This is often seen in algorithms involving 3D coordinates or naive matrix multiplication.

3.7 Exponential Time – $\mathcal{O}(2^n)$

The time doubles with each new addition to the input data. These algorithms become very slow very quickly and are usually impractical for large n.

- Example: Cracking a password by trying every possible combination of characters.

3.8 Factorial Time – $\mathcal{O}(n!)$

The slowest common class. It grows incredibly fast.

- Example: Finding the shortest route that visits every city in a country (The Traveling Salesman Problem).

4 Time Complexity of Iterative Constructs

To calculate time complexity, we often look at the loops in the code.

4.1 Single Loops

A loop that runs from 1 to n performs the operation n times.

- Complexity: $\mathcal{O}(n)$.

4.2 Nested Loops

If you have a loop inside another loop, and both run n times, you multiply the complexities.

- Calculation: $n \times n = n^2$.
- Complexity: $\mathcal{O}(n^2)$.

4.3 Multiple Independent Loops

If one loop runs n times, finishes, and then a second loop runs n times, you add the complexities.

- Calculation: $n + n = 2n$. In Big O, we drop constants like 2.
- Complexity: $\mathcal{O}(n)$.

4.4 Conditional Statements

If you have an 'if-else' statement, we usually assume the worst-case scenario. We calculate the complexity of the block of code inside the 'if' and the block inside the 'else', and take the larger one.

5 Time Complexity of Recursive Algorithms

5.1 Basic Concept of Recursive Cost

Recursive functions call themselves. To find their complexity, we need to know:

1. How many times the function calls itself (branching).
2. How the input size reduces (n becomes $n-1$ or $n/2$).

5.2 Analysis of Simple Recursive Functions

- **Linear Recursion:** If a function calls itself once with input $n-1$, it runs n times. Complexity: $\mathcal{O}(n)$.
- **Divide and Conquer:** If a function breaks the problem in half and does linear work to combine them (like Merge Sort), the complexity is typically $\mathcal{O}(n \log n)$.

6 Analysis of Standard Algorithms

Here is the time complexity breakdown for common algorithms studied in MSc courses:

6.1 Linear Search

We check every element until we find the target.

- Worst Case: $\mathcal{O}(n)$ (Target is at the end or not present).

6.2 Binary Search

We split the sorted array in half each time.

- Worst Case: $\mathcal{O}(\log n)$.

6.3 Bubble Sort

Swaps adjacent elements repeatedly. Requires nested loops.

- Worst Case: $\mathcal{O}(n^2)$.

6.4 Selection Sort

Finds the minimum element and puts it at the beginning.

- Worst Case: $\mathcal{O}(n^2)$.

6.5 Insertion Sort

Takes an element and inserts it into the correct position in a sorted sub-list.

- Worst Case: $\mathcal{O}(n^2)$.
- Best Case: $\mathcal{O}(n)$ (if already sorted).

6.6 Merge Sort

Divides the list into halves, sorts them, and merges them.

- Worst Case: $\mathcal{O}(n \log n)$.

6.7 Quick Sort

Picks a pivot and partitions the array.

- Average Case: $\mathcal{O}(n \log n)$.
- Worst Case: $\mathcal{O}(n^2)$ (if the pivot is chosen poorly, e.g., typically already sorted data).

7 Comparison of Algorithms Based on Time Complexity

The following table summarizes the time complexities:

Algorithm	Best Case	Average Case	Worst Case
Linear Search	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binary Search	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$

8 Practical Considerations in Time Complexity

While Big O notation is the standard theoretical tool, in real-world programming, other factors matter:

1. **Constants matter for small n:** An algorithm with $100n$ operations is theoretically $\mathcal{O}(n)$, but for small inputs, an $\mathcal{O}(n^2)$ algorithm with a small constant (like $1 \cdot n^2$) might actually be faster.
2. **Memory Access:** Algorithms that access memory sequentially (like iterating an array) are faster than those that jump around (like linked lists) due to CPU caching, even if the Big O is the same.
3. **Code Readability:** Sometimes a slightly slower $\mathcal{O}(n^2)$ algorithm is preferred over a complex $\mathcal{O}(n \log n)$ one because it is easier to write, debug, and maintain.

9 Summary of Key Concepts

- Time complexity measures how run-time scales with input size n.
- We usually focus on the Worst Case ($\mathcal{O}$) to ensure reliability.
- $\mathcal{O}(1)$ is the best; $\mathcal{O}(n!)$ is the worst.
- Nested loops usually result in quadratic time $\mathcal{O}(n^2)$.
- Divide and conquer strategies often yield logarithmic $\mathcal{O}(\log n)$ or linearithmic $\mathcal{O}(n \log n)$ time.

10 Practice Problems and Exercises

Problem 1: What is the time complexity of the following code?

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

Answer: $\mathcal{O}(n)$. The loops are separate, not nested.

Problem 2: If an algorithm takes 10 seconds for $n = 1000$ and 40 seconds for $n = 2000$, what is its likely time complexity? *Answer: $\mathcal{O}(n^2)$. Doubling the input ($1000 \to 2000$) quadrupled the time ($10 \to 40$).*