Time and space Complexity.

Time Complexity

Definition: Time Complexity is **not the actual execution time** of a code snippet, which can vary depending on the machine (Windows, Mac) or server (online platforms). Instead, Time Complexity measures the **amount of time taken as a function of input size**. It measures the time taken **in terms of operations**. An operation is considered any statement in the code, such as an integer declaration or a loop iteration. It describes how the number of operations changes as the input size (represented by 'n') increases or decreases.

Why do we need to calculate Time Complexity?

Calculating time complexity is crucial because it helps us understand how efficient an algorithm is as the input size grows. Here’s why it matters:

- **Performance Prediction:** It helps us estimate how fast an algorithm runs for large inputs, preventing slow or impractical solutions.
- **Comparison of Algorithms:** When choosing between multiple approaches, knowing their time complexity guides us toward the best option.
- **Scalability:** A program that runs well on small inputs may become unusable for large data sets if it has poor time complexity.
- **Cost Efficiency:** In real-world applications, optimized algorithms save time and computing resources, making systems more efficient.

For example, a **linear search (O(n))** becomes impractical for large datasets when compared to a **binary search (O(log n))**, which is much faster. By analyzing time complexity, we make better choices to ensure our programs run smoothly.

•**Example: Linear Search:** In a Linear Search algorithm, which searches for a target element in an array, the number of operations is directly related to the size of the array, 'n'. If the array size is 1, it takes approximately 1 operation. If the size is 10, it takes approximately 10 operations. If the size is 100, it takes 100 operations, and if it's 100,000, it takes approximately 100,000 operations. This shows a behaviour whereas the input size increases, the number of operations increases proportionally.

•**Graph Representation:** This relationship can be plotted on a graph with input size 'n' on the x-axis and number of operations on the y-axis. For Linear Search, plotting points like (1,1), (10,10), (100,100), etc., results in a **straight line**. The equation for this line is y = x, or f(n) = n.

•**Big O Notation:** To represent Time Complexity, the **Big O notation (Big O)** is used. It is a **symbol or notation** that describes the Time Complexity as a function of n. For the linear relationship described above, the Time Complexity is written as **Big O of n (O(n))**. This indicates that as the input size increases, the number of operations grows linearly with n.

•**Worst Case Scenario:** When calculating Time Complexity, the **worst-case scenario** is usually considered. This is because, especially for large inputs like those in real-life applications like Facebook or Google, the worst-case performance matters most. An algorithm performing well in the worst case will also perform well for smaller values. Big O notation represents the **worst case scenario** or the **upper bound**. It means that in the worst case, an algorithm will perform at most 'n' operations (for O(n)), and its behaviour won't be worse than that. Best case scenarios can exist (e.g., finding the target at the first index in Linear Search), but the worst case is the upper bound.

•**Calculating from Functions:** Sometimes, the number of operations might result in a complex function, like $4n^2 + 3n + 5$. To determine the Time Complexity using Big O, **two steps** are followed:

1. **Ignore constants:** Mathematical numbers (like 4, 3, 5) are constants and are ignored. The function becomes $n^2 + n + 1$ (after ignoring constants and replacing constant terms with 1).
2. **Consider only the largest term:** For very large values of 'n', the term with the highest power of 'n' dominates the function's growth. In $n^2 + n + 1$, n^2 is the largest term for large 'n'. The smaller terms (n and 1) are ignored.

Therefore, the Time Complexity for $4n^2 + 3n + 5$ is **Big O of n^2 ($O(n^2)$)** 7.

Another example: $100 + 5n^2 + \sqrt{n}$. Ignoring constants gives $1 + n^2 + \sqrt{n}$. For large 'n', n^2 is the largest term (e.g., for n = 1 million, n^2 is 10^{12}, $\sqrt{n}$ is 10^3, and 1 is 1). So, the Time Complexity is **Big O of n^2 ($O(n^2)$)**.

•

Other Notations: Besides Big O (O), there are other notations: **Theta (Θ)** for **average case** Time Complexity and **Omega (Ω)** for **best case** Time Complexity (also called the lower bound). However, for interviews, coding tests, and competitive programming platforms, **Big O is almost exclusively used**. Mathematical formulas and theorems like Master's Theorem or calculus (limits, derivations) can be used to calculate these complexities, especially for semester exams in college, but are typically not required for placements.

Space Complexity:

Definition: Space Complexity is **not the actual space taken** by the program, but the **amount of space taken by an algorithm as a function of input size**. Space is occupied in two ways when code runs: space for the **input** itself (like an array or string) and **auxiliary space**.

Auxiliary Space: Space Complexity primarily focuses on this **extra space** taken by the algorithm, separate from the input space. For example, in Linear Search, the array is the input, and its space is not considered for Space Complexity; only any extra space used by variables or data structures within the algorithm is counted.

Example: Square Array: If a problem requires creating a new array (square array) of the same size 'n' as the input array to store squared elements, this new array occupies auxiliary space. As the input size 'n' increases, the auxiliary space for the new array also increases proportionally. The size of the auxiliary space is exactly 'n', so the Space Complexity is **Big O of n (O(n))**. Graphically, this also appears as a straight line.

Example: Sum of Array Elements: If a problem requires calculating the sum of elements in an array of size 'n', a single variable (like sum) might be used. The space occupied by this variable remains constant regardless of whether the input array size is 10, 100, or 1 million. The auxiliary space is constant, represented as Big O of k (a constant number of operations/space units). Ignoring the constant, the Space Complexity is **Big O of 1 (O(1))**, referred to as **Constant Space Complexity**. Graphically, this is a straight line parallel to the x-axis, showing that space remains constant as input size increases.

Time vs. Space Complexity: While both are important, **Time Complexity is often prioritised** in modern systems. This is because storing extra space is comparatively easier and cheaper than making an algorithm faster in terms of time. Users generally desire a faster and more efficient experience. The fundamental principles for calculating Time and Space Complexity are similar.

Common Time Complexities and Use Cases

Time Complexities are visualised on a graph, generally moving from best (closer to the y-axis) to worst (further from the y-axis).

O(1) - Constant Time:

◦Best possible Time Complexity.
◦Number of operations/space remains constant regardless of input size.
◦Occurs in algorithms with **no loops or recursion**, using simple formulas or constant number of operations.
◦Examples: Calculating sum of 1 to n using formula n*(n+1)/2, printing the first element of an array, printing the last element of a sorted array.
◦Operations on hash tables (like unordered maps/sets in C++ or hash maps in Java) such as adding, deleting, or finding elements are considered O(1) on average (amortised constant time), though theoretically they might be different.
◦Operations inside a loop body are generally assumed to be constant time.
◦Graph: A horizontal line parallel to the x-axis.

•O(n) - Linear Time:
◦The number of operations grows linearly with the input size 'n'.
◦A **good Time Complexity**.
◦Occurs in algorithms with **a single loop** that iterates through the input once.
◦Examples: Linear Search, calculating n factorial using a loop, Kadane's Algorithm, calculating sum of n numbers using a loop, Majority Element using Moore's Voting

Algorithm, calculating N-th Fibonacci number using dynamic programming/iteration (future topic).
◦The number of loop iterations is proportional to 'n', and the work inside the loop is constant.

•O(n^2) - Quadratic Time:
◦The number of operations grows as the square of the input size.
◦Considered a **bad Time Complexity**.
◦Occurs typically with **two nested loops** where each loop iterates up to 'n' times.
◦Examples: Unoptimized sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort. Pattern printing problems with nested loops.
◦Analysis of nested loops: If the outer loop runs n times and the inner loop runs n times for each outer loop iteration, the total operations are roughly $n * n = n^2$. Detailed analysis of Selection Sort nested loops shows the sum of operations is proportional to $n(n-1)/2$, which simplifies to O(n^2) after ignoring constants and lower terms.

•O(n^3) - Cubic Time:
◦The number of operations grows as the cube of the input size.
◦**Even worse** than O(n^2).
◦Occurs typically with **three nested loops**, each iterating up to 'n' times.
◦Example: Printing all possible subarrays of a given array.

•O(log n) - Logarithmic Time:
◦Considered a **very good Time Complexity**, close to O(1) .
◦Occurs in algorithms that **repeatedly divide the input size** by a constant factor (like 2) in each step.
◦Examples: Binary Search, operations on Binary Search Trees (BSTs).
◦Binary Search analysis: The search space is halved in each step (n -> n/2 -> n/4 -> ...). To find how many steps (x) it takes to reach a size of 1 (n / 2^x = 1), the equation solves to n = 2^x, which implies $x = \log_2 n$. So, the time complexity is O(log n).

•O(nlog n) - Linearithmic Time:
◦**Better** than O(n^2) but **worse** than O(n).
◦Occurs in **optimized sorting algorithms**.
◦Examples: Merge Sort, the average case of Quick Sort (worst case is n^2).
◦Also seen in Greedy Algorithms that involve sorting.
◦Graphically, O(n log n) lies between O(n) and O(n^2).

•O(2^n), O(3^n), etc. - Exponential Time:
◦The number of operations grows exponentially with the input size.
◦Considered **quite bad** Time Complexities.
◦Typically seen in **brute-force recursive solutions** without optimization.
◦Example: The brute-force recursive Fibonacci code. Graphically represented high on the worst-case side.

•O(n!) - Factorial Time:
◦The number of operations grows as the factorial of the input size.
◦**Even worse** than Exponential Time.
◦Less common but seen in certain brute-force recursive problems.
◦Examples: Brute-force N-Queens problem, Brute-force Knight's Tour, generating all possible permutations of a string. For a string of size n, there are n! permutations, and generating them takes O(n!) time.

Calculating Time and Space Complexity for Recursion
Recursion requires a slightly different approach for complexity analysis. Two primary methods are discussed for Time Complexity:
1.**Recurrence Relation:** Writing an equation that describes the time taken for a problem of size 'n' in terms of smaller subproblems. This equation is then solved mathematically.
2.**Recursion Tree:** Drawing a tree representing the recursive calls and calculating the total number of calls multiplied by the work done in each call. This is often preferred for practical cases as recurrence relations can be complex.

•Example: Factorial Recursion (O(n) Time, O(n) Space)
◦**Time Complexity (Recurrence Relation):** For a factorial function F(n), the recurrence relation is F(n) = F(n-1) + O(1) (where O(1) represents the constant work like multiplication and checking the base case). Solving this relation by substitution shows that F(n) is proportional to n, leading to **O(n)** Time Complexity.

◦**Time Complexity (Recursion Tree):** For Factorial(4), the calls are F(4) -> F(3) -> F(2) -> F(1) -> F(0). This forms a linear chain. For an input 'n', there are approximately n calls. Each call performs constant work (O(1)). Total Time Complexity = Total Calls * Work per Call = n * O(1) = **O(n)**.

◦**Space Complexity:** In recursion, the **Call Stack** uses memory to store data for each active function call. Even if no extra variables are defined, this stack space is counted. The space complexity is determined by the **depth of the recursion tree** or the **height of the call stack** multiplied by the memory occupied in each call. For Factorial(n), the recursion depth/call stack height is approximately n. Each call occupies constant memory. Space Complexity = Depth * Memory per Call = n * O(1) = **O(n)**.

•Example: Fibonacci Recursion (Brute Force) (O(2^n) Time, O(n) Space)

◦**Time Complexity (Recursion Tree):** For Fibonacci(4), the calls branch out: F(4) calls F(3) and F(2); F(3) calls F(2) and F(1), and so on. This forms a branching tree. The total number of calls at each level roughly doubles (2^0, 2^1, 2^2, 2^3, ...). Summing these calls down to level n-1 (2^0 + 2^1 + ... + 2^(n-1)) using a geometric progression formula result in approximately 2^n calls. Each call does constant work (O(1)). Total Time Complexity = Total Calls * Work per Call = O(2^n) * O(1) = **O(2^n)** . (The exact time complexity for Fibonacci is O(1.618^n), based on the golden ratio, but O(2^n) is the standard answer expected.

◦**Space Complexity:** The depth of the Fibonacci recursion tree for F(n) is approximately n (the longest path from the root to a base case). Each call occupies constant memory. Space Complexity = Depth * Memory per Call = n * O(1) = **O(n)**.

•**Example: Merge Sort Recursion (O(n log n) Time, O(n) Space)**

◦**Work Done in Each Call:** The Merge Sort recursive function calls itself on subproblems and then calls a separate `merge` function. The constant work in the recursive function itself is O(1). The `merge` function is crucial; it takes two sorted subarrays and merges them into one sorted array. Analysis of the `merge` function shows it uses several loops that iterate through the elements of the two subarrays and a temporary array. If the total size of the two subarrays being merged is 'n', the `merge` step takes **O(n) time**.

◦**Time Complexity (Recursion Tree):** Merge Sort divides the array in half repeatedly (n -> n/2 -> n/4 -> ...) until subarrays of size 1 are reached. The number of levels in this division process is $\log_2 n$. At each level of the recursion tree, the total work done by the `merge` step across all calls at that level is proportional to the total number of elements being processed at that level, which is always 'n'. Since there are log n levels and each level does O(n) work, the total Time Complexity is O(n) * log n = **O(n log n)**.

◦**Space Complexity:** The recursion call stack depth for Merge Sort is log n. However, the `merge` function itself uses a **temporary array/vector** of size proportional to the total elements being merged at that step. In the worst case, this temporary array is created for segments up to size n. Although the temporary space for sub-calls is released after the merge, the largest temporary space required at any point during the execution of the recursive calls is O(n). Therefore, the overall Space Complexity, considering both the recursive stack (O(log n)) and the temporary array (O(n)), is O(log n + n). Since 'n' is the larger term, the Space Complexity simplifies to **O(n)**.

Practical Usage of Time Complexity
Understanding Time Complexity is essential when solving problems on coding platforms like LeetCode or CodeForces.

•**Constraints:** Problems on these platforms provide **constraints** on the input size (n). These constraints help determine the acceptable Time Complexity for a solution.

•**Execution Limit:** Generally, coding platforms allow approximately **10^8 operations to run within 1 second**. If a code performs significantly more operations, it results in a **Time Limit Exceeded (TLE)** error.

•**Estimating Acceptable Complexity:** By examining the input constraint (maximum value of 'n') and the 10^8 operations limit, one can estimate the Time Complexity required to avoid TLE.
◦if $n \le 10^5$, an $O(n^2)$ solution $(10^5)^2 = 10^{10}$ operations) will likely get TLE as $10^{10} > 10^8$. An O(n log n) solution ($10^5 * \log(10^5) \approx 10^5 * 5 * \log_2(10) \approx 10^5 * 5 * 3.3 \approx 1.65 * 10^6$ operations) is acceptable as $1.65 * 10^6 < 10^8$. So for $n \le 10^5$, aiming for O(n log n) or better (O(n), O(log n), O(1)) is necessary.
◦If $n \ge 10^8$ (meaning n can be very large, e.g., 10^9), only O(log n) or O(1) solutions are typically acceptable. An O(n) solution would exceed the 10^8 limit.
◦If $n \le 10^8$, an O(n) solution is usually fine.

◦If $n \leq 10^6$, an $O(n \log n)$ solution should be acceptable. This might hint that a sorting-based approach could work.
◦If $n \leq 400$ (roughly $4*10^2$), an $O(n^2)$ solution might be acceptable ($400^2 \approx 1.6 * 10^5 < 10^8$).
◦If $n \leq 500$, an $O(n^3)$ solution might be acceptable ($500^3 = 125 * 10^6 = 1.25 * 10^8 \approx 10^8$).
◦If $n \leq 25$, an O(2^n) solution might be acceptable ($2^{25} \approx 3.3 * 10^7 < 10^8$). This small constraint on n for exponential complexity can hint at a brute-force recursive approach.
◦If $n \leq 12$, an O(n!) solution might be acceptable ($12! \approx 4.79 * 10^8 \approx 10^8$). This also hints at brute-force recursion.

•**Estimating Worst Case:** Constraints help estimate the worst-case performance of an algorithm without writing and running the full code. For instance, if O(n log n) is required based on constraints, there's no need to waste time thinking about $O(n^2)$ solutions.

In summary, Time and Space Complexity are fundamental for analysing algorithm efficiency. Big O is the standard notation for worst-case complexity in practical scenarios. Common complexities range from O(1) (best) to O(n!) (worst). Calculating complexity involves analysing loops and recursion, with recursion analysis often using recursion trees. Space complexity considers auxiliary space and, for recursion, the call stack. Practical application on coding platforms involves using input constraints to determine the maximum acceptable Time Complexity based on the 10^8 operations per second limit.