What is Time Complexity?

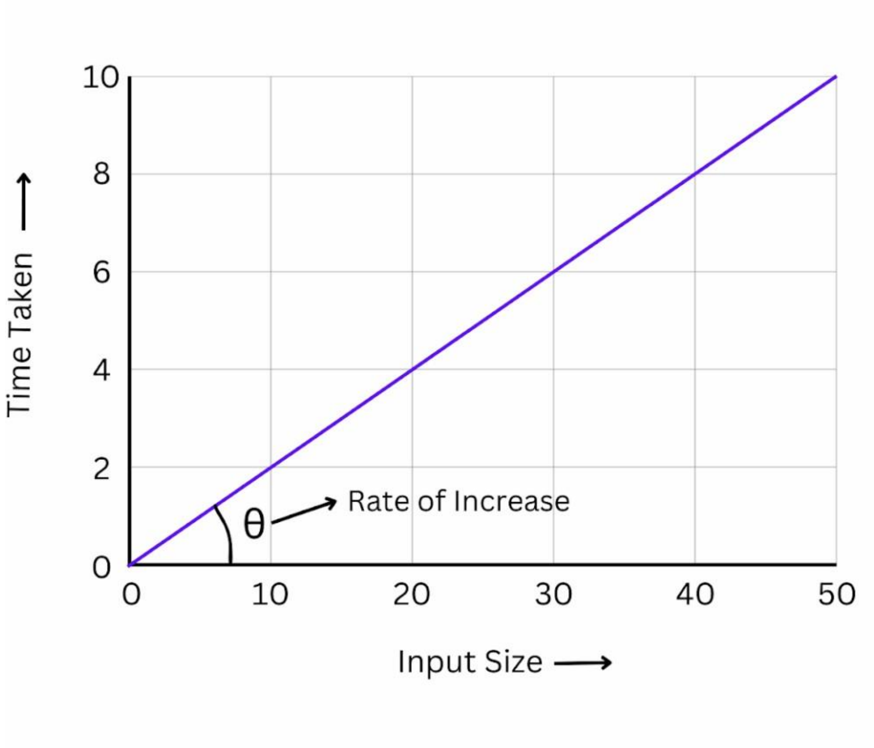
When solving a problem, there may be multiple approaches (codes) to achieve the desired result. The concept of time complexity helps us evaluate & compare these approaches.

In a way, allowing us to determine which one is more efficient. It's a crucial metric that often becomes a key focus during technical interviews. Mainly because it demonstrates the effectiveness of a solution.

At first glance, it might seem to represent the time a machine takes to execute a piece of code. However, this is a misconception. It's more about the code than the machine.

Time complexity measures how the time required to execute a code changes as the size of the input grows. It is independent of the machine used to execute the code & focuses solely on the algorithm's behavior with increasing input size. This makes it a universal metric to compare algorithms.

Let's understand this using the following diagram:



Representing the Time Complexity of Any Code

Let's understand this using the following example:

```
i = 1
while i <= 5:
    print("Code and Debug")
    i += 1
```

The time complexity of this code can be determined by analyzing the number of steps it takes to execute.

Step-by-Step Analysis

In terms of **Big O notation**, time complexity reflects the number of steps relative to the input size. Let's break down the steps for this code:

1. **Initialization:** The variable i is assigned an initial value of 1
2. **Comparison:** The condition i <= 5 is checked
3. **Execution:** The statement print("Code and Debug") is executed
4. **Increment:** The value of i is increased by 1 using i += 1

This flow repeats, and the condition i <= 5 is re-checked after each increment. The process continues until the condition becomes false, i.e., when i exceeds 5.

Let's analyze the steps further:

- The loop runs **5 times** since i starts at 1 and increments by 1 until it reaches 6.
- For each iteration, three core steps are executed:
    - Checking the condition
    - Executing the print statement
    - Incrementing i

Thus, the total number of steps can be calculated as 5 × 3 = 15. In terms of **Big O notation**, this is **O(15)**.

Generalizing for N

If we replace the constant 5 with a variable **N**, the loop will run **N** times. The number of steps would then become:
N×3=3N

When expressed in terms of Big O, we ignore constant factors. So the time complexity becomes **O(N)**.

Why Shouldn't We Use Machine Time to Evaluate Code?

The actual execution time of a code snippet can vary greatly depending on the machine it runs on. For instance:

- Running the same program on an older, low-performance computer like an older Windows PC. And on a high-end machine like the latest MacBook, will yield significantly different results.
- The high-performance machine will likely execute the code faster due to better hardware. While the older machine will take longer.

These differences in execution time arise from hardware configurations. Such as the processor speed, available memory & system architecture.

As a result, evaluating code solely based on the time it takes to execute on a specific machine is not a reliable method of comparison.

Why Manual Counting Isn't Feasible?

Manually counting the steps works only for simple cases like these. But becomes impractical for more complex code. Consider scenarios where:

- The loop runs millions or billions of times
- Nested loops or complex operations are involved within the loop

In such cases, manually counting the steps would be cumbersome and error-prone. Hence, a systematic approach to calculate time complexity is essential.

Rules for Calculating Time Complexity

To simplify the process, we follow these key rules:

1. **Worst-case Scenario:** Always analyze the worst-case scenario. To ensure the algorithm performs well under all conditions.
2. **Ignore Constant Factors:** Constants like **3** in **3N** are ignored. Because they don't significantly impact scalability as **N** grows.
3. **Focus on Growth Rate:** Lower-order terms or small constant values are omitted. As they have negligible impact on the algorithm's efficiency for large inputs.

Let’s See the Rules for Calculating Time Complexity with an Example

> *Always calculate time complexity in terms of worst case.*

Let’s discuss the rules individually with respect to the given code:

```
marks = int(input("Enter your marks = "))

if marks >= 90:
    print("A")
elif marks >= 80 and marks < 90:
    print("B")
elif marks >= 70 and marks < 80:
    print("C")
elif marks >= 60 and marks < 70:
    print("D")
elif marks >= 50 and marks < 60:
    print("E")
else:
    print("Fail")
```

Step-by-Step Explanation

1. **Initialization**: The code starts by taking an integer input for marks.
2. **Condition Checks**: The input value is then compared against several thresholds (>= 90, >= 80, etc.) to determine which grade to assign.
3. **Output**: Based on the condition that evaluates to True, the corresponding grade is printed & the remaining conditions are skipped.

Now let’s evaluate the code for the three scenarios:

1. Best Case

The **best case** occurs when the first condition evaluates to True. Meaning that the code executes the least number of steps. For instance:

- If marks = 95, the condition marks >= 90 is satisfied immediately

- Steps involved:
  1. Input is taken
  2. The condition marks >= 90 is checked
  3. The grade A is printed

In total, **3 steps** are executed. This represents the **minimum execution time** for this code.

2. Worst Case

The **worst case** occurs when none of the conditions in the if & elif blocks evaluate to True. So the code has to check every condition until it reaches the else block. For instance:

- If marks = 45, all the if and elif conditions are evaluated & found to be False, and the program executes the else block
- Steps involved:
  1. Input is taken
  2. The conditions marks >= 90, marks >= 80, marks >= 70, marks >= 60, and marks >= 50 are all checked
  3. The else block executes, and Fail is printed

In total, **7 steps** are executed. This represents the **maximum execution time** for this code.

3. Average Case

The **average case** lies somewhere between the best and worst cases. Depending on the distribution of marks. For example:

- If marks = 75, the conditions marks >= 90 and marks >= 80 are checked. Both evaluate to False & the third condition marks >= 70 and marks < 80 is satisfied.
- Steps involved:
  1. Input is taken
  2. The conditions marks >= 90 and marks >= 80 are checked
  3. The condition marks >= 70 is satisfied, and C is printed

In this case, **5 steps** are executed. The actual **average case** depends on the likelihood of each condition being met, which may vary based on real-world data.

Worst-Case Time Complexity

For a generalized input size **N**, the number of steps is determined by the number of conditions in the code. Since we always analyze the **worst-case scenario** for time complexity:

- The worst-case scenario involves evaluating all conditions sequentially, followed by executing the else block
- Assuming there are **k** conditions, the worst-case time complexity is **O(k)**, where **k = 6** in this example

Why Focus on the Worst Case?

1. **Ensures Robustness**: Evaluating the worst case ensures the algorithm handles the most demanding situations efficiently.
2. **Predicts Scalability**: It provides insights into how the code performs as the number of conditions (k) increases.
3. **Standard Practice**: In algorithm analysis, the worst-case scenario is used as a benchmark to compare different solutions.

Avoid Including the Constant Terms

Let's check into the rule of **ignoring constant terms** when calculating time complexity. Using an example to understand why this approach simplifies the analysis without losing accuracy.

Consider the time complexity expression:

$$T(N) = 5N^{10} + 8N^{3} + 2$$

Here, T(N) represents the total number of operations performed by the algorithm as a function of the input size **N**. Let's analyze what happens when **N** becomes very large.

Step-by-Step Analysis

Suppose **N = 10^6** (a very large number). Substituting this value into the equation, we get:

$$T(N) = 5(10^{60}) + 8(10^{18}) + 2$$

Breaking this down:

1. **$5(10^{60})$:** This is an extremely large term, dominating the overall value of T(N)
2. **$8(10^{18})$:** This term, while significant on its own, is much smaller compared to $5(10^{60})$
3. **2:** This constant term is negligible in comparison to the other terms

When **N** becomes large, the term $5(10^{60})$ overwhelmingly dictates the growth of **T(N)**. While **$8(10^{18})$** and **2** contribute very little to the overall value.

Why Ignore Constant Terms?

1. **Insignificant Contribution:** As **N** grows, the constant terms & smaller-order terms (like **$8N^3$**) become insignificant in comparison to the highest-order term **($5N^{10}$)**.
2. **Focus on Growth Rate:** Time complexity analysis is concerned with how the algorithm scales with input size. And not with the exact number of operations. Ignoring constants simplifies the expression while retaining key insights about the algorithm's behavior.
3. **Standardization:** In Big O notation, we represent only the dominant term without its coefficient. For **$T(N) = 5N^{10} + 8N^3 + 2$**, the dominant term is **N^{10}**, so the time complexity is written as: **$O(N^{10})$**.

Now that we have fully understood time complexity. Let's move on to the next important part.

What is Space Complexity?

Space complexity refers to the amount of memory required by a program during its execution. This includes all the memory used by variables, data structures & any additional memory allocated while solving the problem.

Since memory usage can vary depending on the machine. We represent space complexity using **Big O notation**, rather than standard units like MB or GB.

> **Formal Definition:**
>
> *Space complexity is the sum of two components:*
>
> 1. ***Input Space:** Memory required to store the input data*
> 2. ***Auxiliary Space:** Additional memory required to solve the problem. Including temporary variables, data structures, or function call stacks.*

Example

Consider the following code:

```
a = int(input())  # Input Space
b = int(input())  # Input Space
c = a + b         # Auxiliary Space
print(c)
```

Here:

- a and b are used to store the inputs, which account for the **input space**
- c is a temporary variable used to store the result, accounting for the **auxiliary space**

The total space complexity is O(3), as three variables are used. However, when written in Big O notation, constants are ignored. So, the space complexity simplifies to **O(1)** (constant space).

Using Arrays

If we use an array of size **N** in a program, the space complexity becomes **O(N)**, as **N** elements are stored in memory.

Good Coding Practice: Avoid Input Manipulation

If asked to solve a problem such as adding two numbers **a** and **b**. One way to reduce space complexity is by modifying the input directly, like this:

```
b = a + b
```

This approach eliminates the need for an additional variable c, reducing space usage. However, this is generally **not considered good practice**. Especially in coding interviews, for the given reasons:

1. **Input Integrity:** The original input values might be required for other purposes in the program or system.
2. **Company Standards:** Many companies reuse input data for multiple operations. Altering inputs may lead to unintended consequences.

> **Rule of Thumb:** *Avoid manipulating input data unless explicitly instructed to do so by the interviewer or problem statement. Always prioritize clarity & integrity of your code over reducing space complexity.*

Points to Remember for Competitive Programming

When participating in competitive programming or solving problems on platforms like Leetcode or GeeksforGeeks. Your code is executed on online servers. These servers generally handle about **10^8** operations per second.

Key Guidelines:

1. If the time limit is 2 seconds, your code should ideally perform no more than **2×10^8** operations
2. For a 5-second time limit, the threshold increases to **5×10^8** operations

To ensure your code executes within the time limit:

- Keep the time complexity around **O(10^8)** operations, ignoring constants & smaller terms