Unit 1

Algorithm Complexity

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
// Suppose we have to calculate the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum holds the sum of the n numbers
return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

- Space complexity: An algorithm's space complexity is the amount of space required to solve a problem and produce an output.
- Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

- To store program instructions
- To store constant values
- To store variable values
- To track the function calls, jumping statements, etc.

- Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space.
- The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

- So, Space complexity = Auxiliary space + Input size.

An algorithm's space complexity quantifies how much space or memory it takes to run as a function of the length of the input while an algorithm's time complexity measures how long it takes an algorithm to run as a function of the length of the input.

That's why it's so important to learn about the time and space complexity analysis of various algorithms.

1. O(1)

Where an algorithm's execution time is not based on the input size n, it is said to have constant time complexity with order O (1).

Whatever be the input size **n**, the runtime doesn't change. Here's an example:

```
• #include <stdio.h>
•  int main()
• {
•     printf("Hello World");
•     return 0;
• }
• Output
• Hello World
```

- In the above code “Hello World” is printed only once on the screen.
- So, the time complexity is constant: O(1) i.e. every time a constant amount of time is required to execute code, no matter which operating system or which machine configurations you are using.

2. O(n)

When the running time of an algorithm increases linearly with the length of the input, it is assumed to have linear time complexity, i.e. when a function checks all of the values in an input data set (or needs to iterate once through every value in the input), it is said to have a Time complexity of order O (n). Consider the following scenario:

```
• #include <stdio.h>
• void main()
• {
•         int i, n = 8;
•         for (i = 1; i <= n; i++) {
•                 printf("Hello World !!!\n");
•         }
• }
```

- Output
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!
- Hello World !!!

- In the above code “Hello World !!!” is printed only n times on the screen, as the value of n can change.
- So, the time complexity is linear: O(n) i.e. every time, a linear amount of time is required to execute code.

What does it mean to state best-case, worst-case and average time complexity of algorithms?

- Let's take the example of searching for an item sequentially within a list of unsorted items.
- If we're lucky, the item may occur at the start of the list. If we're unlucky, it may be the last item in the list.
- The former is called *best-case complexity* and the latter is called *worst-case complexity*.

- If the searched item is always the first one, then complexity is O(1).
- if it's always the last one, then complexity is O(n).
- We can also calculate the *average complexity*, which will turn out to be O(n).
- The term "complexity" normally refers to worst-case complexity.

Mathematically, different notations are defined

- Worst-case or upper bound: Big-O: $O(n)$
- Best-case or lower bound: Big-Omega: $\Omega(n)$
- Average-case: Big-Theta: $\Theta(n)$

- 1. Big O Notation
- Big-O notation represents the upper bound of the running time of an algorithm.
- Therefore, it gives the worst-case complexity of an algorithm.

Omega Notation

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best-case complexity of an algorithm.
- The execution time serves as a lower bound on the algorithm’s time complexity. It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Theta Notation

- Theta notation encloses the function from above and below.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
- The execution time serves as both a lower and upper bound on the algorithm’s time complexity.
- It exists as both, the most, and least boundaries for a given input value.