Why is Time complexity Significant?

Let us first understand what defines an algorithm.

An Algorithm, in computer programming, is a finite sequence of well-defined instructions, typically executed in a computer, to solve a class of problems or to perform a common task. Based on the definition, there needs to be a sequence of defined instructions that have to be given to the computer to execute an algorithm/ perform a specific task. In this context, variation can occur the way how the instructions are defined. There can be any number of ways, a specific set of instructions can be defined to perform the same task. Also, with options available to choose any one of the available programming languages, the instructions can take any form of syntax along with the performance boundaries of the chosen programming language. We also indicated the algorithm to be performed in a computer, which leads to the next variation, in terms of the operating system, processor, hardware, etc. that are used, which can also influence the way an algorithm can be performed.

Now that we know different factors can influence the outcome of an algorithm being executed, it is wise to understand how efficiently such programs are used to perform a task. To gauge this, we require to evaluate both the Space and Time complexity of an algorithm.

By definition, the Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. While Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Now that we know why Time complexity is so significant, it is time to understand what is time complexity and how to evaluate it.

To elaborate, Time complexity measures the time taken to execute each statement of code in an algorithm. If a statement is set to execute

repeatedly then the number of times that statement gets executed is equal to N multiplied by the time required to run that function each time.

The first algorithm is defined to print the statement only once. The time taken to execute is shown as **0 nanoseconds**. While the second algorithm is defined to print the same statement but this time it is set to run the same statement in FOR loop 10 times. In the second algorithm, the time taken to execute both the line of code – FOR loop and print statement, is **2 milliseconds**. And, the time taken increases, as the N value increases, since the statement is going to get executed N times.

By now, you could have concluded that when an algorithm uses statements that get executed only once, will always require the same amount of time, and when the statement is in loop condition, the time required increases depending on the number of times the loop is set to run. And, when an algorithm has a combination of both single executed statements and LOOP statements or with nested LOOP statements, the time increases proportionately, based on the number of times each statement gets executed.

This leads us to ask the next question, about how to determine the relationship between the input and time, given a statement in an algorithm. To define this, we are going to see how each statement gets an order of notation to describe time complexity, which is called Big **O Notation**.

What are the Different Types of Time complexity Notation Used?

As we have seen, Time complexity is given by time as a function of the length of the input. And, there exists a relation between the input data size (n) and the number of operations performed (N) with respect to time. This relation is denoted as Order of growth in Time complexity and given notation O[n] where O is the order of growth and n is the length of the input. It is also called as **‘Big O Notation’**

Big O Notation expresses the run time of an algorithm in terms of how quickly it grows relative to the input ‘n’ by defining the N number of operations that are done on it. Thus, the time complexity of an algorithm is denoted by the combination of all O[n] assigned for each line of function.

There are different types of time complexities used, let’s see one by one:

1. Constant time – O (1)

2. Linear time – O (n)

3. Logarithmic time – O (log n)

4. Quadratic time – O (n^2)

5. Cubic time – O (n^3)

and many more complex notations like **Exponential time, Quasilinear time, factorial time, etc.** are used based on the type of functions defined.

Constant time – O (1)

An algorithm is said to have constant time with order O (1) when it is not dependent on the input size n. Irrespective of the input size n, the runtime will always be the same.

The above code shows that irrespective of the length of the array (n), the runtime to get the first element in an array of any length is the same. If the run time is considered as 1 unit of time, then it takes only 1 unit of time to run both the arrays, irrespective of length. Thus, the function comes under constant time with order O (1).

Linear time – O(n)

An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input. When the function involves checking all the values in input data, with this order O(n).

The above code shows that based on the length of the array (n), the run time will get linearly increased. If the run time is considered as 1 unit of time, then it takes only n times 1 unit of time to run the array. Thus, the function runs linearly with input size and this comes with order O(n).

Logarithmic time – O (log n)

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size. The number of operations gets reduced as the input size increases. Algorithms are found in binary trees or binary search functions. This involves the search of a given value in an array by splitting the array into two and starting searching in one split. This ensures the operation is not done on every element of the data.

Quadratic time – O (n^2)

An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n^2) with the length of the input. Generally, nested loops come under this order where one loop takes O(n) and if the function involves a loop within a loop, then it goes for O(n)*O(n) = O(n^2) order.

Similarly, if there are ‘m’ loops defined in the function, then the order is given by O (n ^ m), which are called **polynomial time complexity**functions.

Thus, the above illustration gives a fair idea of how each function gets the order notation based on the relation between run time against the

number of input data sizes and the number of operations performed on them.

E.g. 1

```
int main{
   cout<< “Hello”;
}
```

This will take very less time to execute i.e. X.

E.g. 2

```
int main{
   cout<< “Hello”;
   cout <<“Hello”;
}
```

This will take very less time to execute i.e. X*2.

In this example these are very small operations it takes very less time to execute and these are negligible.

Actual impact starts when there are huge number of operations in our program. These operations will depend on our input.

If we take input from as N, and we’ll use for loop in our program,

E.g. 1

```
int main{
   for (int i=0; i<n; i++){
      cout<< “Hello”;
   }
}
```

In this you can see we used coutt operation one times, but this operation will be repeated N times.
In any program time will be always = N.

So here we are finding relation between input size & running time (operation)

If we double our input how much time it takes to execute,
If we triple it then how much time it will take to execute.

In how many ways we can calculate time complexity
Linear - if we set N=4, running time will be 4,

Quadratic- if we increase input size 2 times, it will perform operations 4 times,

Cubic - if running time increase 2 times, operation increase 1 time,

It can be logn relation or Square root relation there can be multiple relations,

Our code will be efficient if there are minimal relations,

Actual calculating complexity
E.g. 1

```
int main() {
    int N;
    std::cin >> N;

    for (int i = 0; i < N; i++) {
        std::cout << "Hello";
    }

    return 0;
}
```

Here time complexity is nothing but running time or Numbers of operations performed in program,

Here if we increase N, number of operations will be increased, time complexity will be increased.

We calculate time complexity in three ways

1. Best Case - Ω (1)
2. Average Case - $\frac{o(n+1)}{2}$
3. Worst Case - O(n)

Best case - Ω (1): omega of 1

Numbers : (1, 2, 3, 4, 5)
Search for : 1
Best case is that which takes minimum time to execute code & perform operations.
In this example if we search for 1 it's available on 1st index and it will take the minimum amount of time (it will perform only 1 operation) to execute.

Average case - $\frac{o(n+1)}{2}$: theta of n

Numbers : (3, 2, 1, 4, 5)
Numbers : (3, 2, 4, 1, 5)
Search for : 1
Average case is that which takes more than minimum time but less than maximum time to execute code & perform operations.
We'll check all possible cases like in first example it performs 3 operation & in second it performs 4 operations, so we'll calculate all possible cases as (3+4)/n

Worst case - O(n): big O of n
Numbers : (3, 2, 5, 4, 1)
Search for : 1
Worst case is that which takes maximum time to execute code & perform operations. It should not take more time than N.
If we take 10 power 5, it will take 1,00,000 operations to execute.

Whenever we discuss time complexity in our interviews or in competitive coding we always discuss in worst case (big O) complexity, because here we want to find the maximum time our program can take to execute & perform operations. (e.g. In our platform how many users can work at a time)

Also we can calculate Best & Average case but we won't consider it, always worst case will be considered.

Some examples to calculate Time Complexity
e.g.1

```
int main() {
    int n;
    std::cin >> n;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            std::cout << "Hello";
        }
    }

    return 0;
}
```

In this scenario j will perform operations n times for each i which is outer loop, it means i will perform operations N times. j will also operate N times, so the results would be n*n or you call it as N square. Which will be maximum time to execute and operate our program.

Complexity will be : O(n square)

e.g.2

```
int main() {
    int n, m;
    std::cin >> n >> m;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            std::cout << "Hello";
        }
    }

    return 0;
}
```

In this scenario j will perform operations m times for each i which is outer loop, i will perform operations n times, so maximum operations we can perform are n*m,

Complexity will be : O(n*m)

e.g.3

```
int main() {
    int n, m;
    std::cin >> n >> m;

    for (int i = 0; i < n; i++) {
        std::cout << "Hello";
    }
    for (int i = 0; i < m; i++) {
        std::cout << "Hello";
    }
    return 0;
}
```

In this scenario i will perform operations n times & j will perform operations m times, so maximum operations we can perform are n+m,
Complexity will be : O(n+m)
In other scenario if n is n power 6 m is 3, here n is significant where m is negligible. Final input size will be depend on higher input size.
Complexity will be : O(n)

Complexity comparison: (number of operations)

Compare	O(n)	O(n^2)	O(n^3)
n = 1	1	1	1
n = 2	2	4	8
n = 3	3	9	27
n = 10^5	10^5	10^10	10^15

Time Complexities of all Searching Algorithms

Algorithm	**Time Complexity**			Space Complexity
	Best	**Average**	**Worst**	Worst
Linear Search	O(1)	O(N)	O(N)	O(N)
Binary Search	O(1)	O(log N)	O(log N)	O(1)

Time Complexities of all Sorting Algorithms

Algorithm	**Time Complexity**			Space Complexity
	Best	**Average**	**Worst**	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Bucket Sort	Ω(n +k)	θ(n +k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n + k)
Count Sort	Ω(n +k)	θ(n +k)	O(n +k)	O(k)
Shell Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(1)
Tim Sort	Ω(n)	θ(n	O(n	O(n)

		log(n))	log (n))	
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Cube Sort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)