Big O Notation

Big O Notation is a programming tool used to find out the time complexity of algorithms. It allows programmers to group algorithms based on how their run time or space requirements vary as the input size varies. It is used to give the upper-bound runtime or worst-case complexity of an algorithm. It analyzes and classifies algorithms depending on their run time or space requirement. It calculates the time and amount of memory needed to execute each code of statement in an algorithm for an input value.

Big O Notation is a mathematical notation named after the term “order of the function”, meaning growth of functions. It is also called Landau’s Symbol. Big O Notation in data structure is used to express how complex an algorithm is using algebraic terms. It belongs to the asymptotic notations group. Asymptotic notations describe an algorithm run time when the input tends towards a specific or limiting value. It helps analyze the algorithm performance change in order of input size

For people who work with big data, Big O Notation is especially useful in analyzing algorithms. The tool helps programmers calculate the scalability of an algorithm or count how many steps it must execute to give output based on data the program works on. If you are looking to fine-tune your code to increase efficiency, the Big O Notation in data structure can be very effective.

To understand what Big O Notation is,we can use this example,

$O(n^2)$ - It is pronounced as “Big O squared”

Where n = input size

The function $g(n) = n^2$ “

Inside the “O()” gives us an idea of how complex the algorithm is with respect to the input size.

Formal Definition of Big O notation

Once upon a time, there was an Indian king who wanted to reward a wise man for his excellence. The wise man asked for nothing but some wheat that would fill up a chessboard.

But here were his rules: in the first tile he wanted 1 grain of wheat, then 2 on the second tile, then 4 on the next one…each tile on the chess board needed to be filled by double the amount of grains as the previous one. The naïve king agreed without hesitation, thinking it would be a trivial demand to fulfill, until he actually went on and tried it…

Wheat and Chess Board, Image from Wikipedia

So how many grains of wheat does the king owe the wise man? We know that a chess board has 8 squares by 8 squares, which totals 64 tiles. So the last tile should have a total of $\mathbf{2^{63}}$ grains of wheat. If you do a calculation online, **for the entire chessboard,** you will end up getting $\mathbf{1.8446744*10^{19}}$ – that is about 18 followed by 18 zeroes.
Assuming that each grain of wheat weighs 0.01 grams, that gives us 184,467,440,737 tons of wheat. And 184 billion tons is quite a lot, isn't it?

The numbers grow quite fast later for exponential growth. The same logic goes for computer algorithms. If the required efforts to accomplish a task grow exponentially with respect to the input size, it can end up becoming enormously large.

As we will see in a moment, the growth of 2^n is much faster than n^2. Now, with n = 64, the square of 64 is 4096. If you add that number to 2^{64}, it will be lost outside the significant digits.

This is why, when we look at the growth rate, we only care about the dominant terms. And since we want to analyze the growth with respect to the input size, the coefficients which only multiply the number rather than growing with the input size do not contain useful information.

Time Complexity

Time complexity refers to how much time it takes for the program to complete the task. Instead of measuring the actual time required to execute each statement in the code, Time Complexity considers how many times each statement executes. The 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. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on. The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve a given problem is called the ***time complexity*** of the algorithm. Time complexity is a very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed. Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it takes for each statement to complete. As a result, it is highly dependent on the size of the processed data. It also aids in defining an algorithm's effectiveness and evaluating its performance

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.

If 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.

In conclusion, an algorithm that 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.

Knowing the time complexity of an algorithm with a given input data size can help you plan your resources, process, and provide the results efficiently and effectively hence making you an effective programmer.

Space Complexity

The space complexity is related to how much memory the program will use, and therefore is also an important factor to analyze. The space complexity works similarly to time complexity. It refers to the space the program takes to complete the task.

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

However, people frequently confuse space complexity with auxiliary space. Auxiliary space is simply extra or temporary space, and it is not the same as space complexity. To put it another way,

Auxiliary space + space use by input values = Space Complexity

The best algorithm/program should have a low level of space complexity. The less space required, the faster it executes.

Calculating and analyzing space complexity is important because in real-world applications developers are bounded/limited to acquire the memory in the devices. The calculation of the space complexity also helps the developer to know about the worst case of that algorithm

so as to improve it to perform in the worst case also. Whenever we say that our algorithm is sufficient then it means that the algorithm is solving the problem in less amount of time while taking the least amount of space.

Let's take an example:

```
#Sum Of N Natural Number
int sum(int n)
{
 int i,sum=0;
 for(i=n;i>=1;i--)
 sum=sum+i
 return sum;
}
```

So in the above example input value is 'n' that is constant which will take the space of O(1).

Now what about auxiliary space, so it is also O(1) because 'i' and 'sum' are also constants.

Hence total space complexity is O(1).

Why do we need to calculate the space complexity?

As discussed above space complexity is an important parameter that must be calculated to analyze any algorithm/problem and check its efficiency.

Nowadays all systems come up with a large memory so this space is not considered for them but to make our algorithm more efficient so that it can run in less amount of space we have to analyze the space complexity.

Developers of real-world applications are constrained by the memory space of the systems they chose to run on. This is where space complexity comes into play, as we never want to run a function or process that consumes more space than the system has available at any given time.