**Asymptotic Complexity**

Asymptotic complexity is related to the growth of a function, i.e. how a function behaves when the input data gets increasingly large.

For example, take the follow bit of code:

if (n<2) cout << n;  
else cout << n/2;

C++

We can say that each line of this code takes a certain time to execute. This time is the cost. For some code with multiple lines, the total time to execute all of the instructions is the overall cost.

In this particular case, we can see that regardless of the value of n, the bit of code will take the same amount of time to run, i.e. the time complexity is independent of the value of n. We say that this bit of code runs in constant time. In fact, we do not say that, we write it in notation format as . Each of these notations and their proper meanings will be explained in due course.

Here is another example:

for (int i = 0; i<n; i++) printf("%d", i);

C++

This is a loop. Here, the number of iterations increase if n increases, which in turn means the time complexity increases. Thus, we have a time complexity .

Now compare the two following pieces of code that each check if a number is a prime number or not.

Method 1:

for (i=2; i<n; i++) if (n%i == 0) break;

C++

This is a brute force approach. In the worst case, we would need to perform the loop n times. Thus, the worst-case time complexity is . The method has linear time complexity.

Method 2:

for (i=2; i<sqrt(n); i++) if (n%i == 0) break;

C++

This approach, in the worst case, requires iterations. Thus, the worst-case time complexity is .

From here we can see that the notation is used for the worst-case time complexity. It is called Big O notation.

**Big O Notation**

Big O Notation helps us describe how well some particular code performs. Consider the following function:

int sum (int n)  
{  
 int sum = 0;  
 for (int i=1; i<=n; i++)  
 sum += i;  
 return sum;  
}

C++

Now consider another function:

int sumOfArray(int n)  
{  
 return n\*(n+1)/2;  
}

C++

Intuitively, you know that the second function is likely to be better, but how do we prove this. Better, could mean several things. It could mean faster, less memory intensive or even more readable. For now, we will focus on which code is faster, formally, time complexity.

An easy way to check this would be to simply time the two functions. This doesn’t work for general use however, since different machines will run the functions over different times, depending on the hardware it has and how much pressure it is under. Time complexity doesn’t deal with actual time taken to run a function, but rather the behaviour of a function depending on input, i.e. whether it takes more time to run based on the size of the input. We find time complexity by looking at the number of operations.

The second code was doing three things, one addition, one multiplication and one division. Regardless of the size of the input, that function would still be doing just those three operations. We say that this function has a time complexity O(1). This just means that the time it takes to operate remains the same.

The first code had a for loop, and so as the value of n increased, it would have to run that loop more times. The number of operations was dependant on n. Thus, we say that the time complexity is O(n), meaning the time it takes to operate grows *linearly* with the value of n. Notice that there are other lines in this code and the exact time is actually going to be a function like an+b, but we are not concerned with the exact time, only with the behaviour. If we had two separate loops, we wouldn’t say the time complexity was O(2n), we would say that it was O(n), since again, the time it takes grows linearly.

Consider another example:

void someFunct()  
{  
 for (int i=0; i<n; i++)  
 for (int j=0; j<n; j++) *//some code;*}

C++

This function has a nested loop, so as the value of n increases, the number of operations increases quadratically. For each number, we have to run the outer loop, and inside that loop, we have to run the inner loop n times. So, we run n loops, n times. . Thus, time complexity is O(n2).

In general, what we learnt is that constants and smaller terms don’t matter. O(5n) is just O(n) since the 5 becomes irrelevant for very large values of n. O(5n2 + 8n) is just O(n2), since the 8n becomes irrelevant.

We have previously said that the Big O Notation gives us the upper bound or worst-case time complexity, i.e. the behaviour of an algorithm in the worst case. For example, for an algorithm with a time complexity O(n), in the worst case (where n is becoming very large), the time taken will increase linearly with n.

We also have Big Omega Notation and Big Theta Notation. Big Omega Notation, (n), gives us the lower bound, or best-case time complexity, i.e. the behavior of an algorithm in the best case. Big Theta Notation, (n), is said to give the asymptotic tight bound on the running time. It essentially says that once n gets large enough, it will remain between two bounds. This will be discussed later.