Time Complexity of Algorithms

For any defined problem, there can be N number of solution. This is true in general. If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions. And I am the one who has to decide which solution is the best based on the circumstances.

Similarly for any problem which must be solved using a program, there can be infinite number of solutions. Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number `n` is `n*n`):

One solution to this problem can be, running a loop for `n` times, starting with the number `n` and adding `n` to it, every time.

```
/*

    we have to calculate the square of n

*/

for i=1 to n

    do n = n + n

// when the loop ends n will hold its square

return n
```

Copy

Or, we can simply use a mathematical operator `*` to find the square.

```
/*

    we have to calculate the square of n

*/

return n*n
```

Copy

In the above two simple algorithms, you saw how a single problem can have many solutions. While the first solution required a loop which will execute for `n` number of times, the second solution used a mathematical operator `*` to return the result in one line. So which one is the better approach, of course the second one.

What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run `n` number of times, so the time complexity will be `n` atleast and as the value of `n` will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of `n`, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)

{

    statement;

}
```

Copy

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)

{

    for(j=0; j < N;j++)

    {

    statement;

    }

}
```

Copy

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)

{

    mid = (low + high) / 2;

    if (target < list[mid])

        high = mid - 1;

    else if (target > list[mid])

        low = mid + 1;

    else break;

}
```
Copy

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)

{

    int pivot = partition(list, left, right);

    quicksort(list, left, pivot - 1);

    quicksort(list, pivot + 1, right);

}
```
Copy

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE: In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.

4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Suppose you've calculated that an algorithm takes f(n) operations, where,

```
f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Copy

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

Now that we have learned the Time Complexity of Algorithms, you should also learn about Space Complexity of Algorithms and its importance.