# Time Complexity:

Time complexity is a function that tells the relationship about how the time varies with the variation in input size.

NOTE: Amount of time taken is a function of the length of input.

Time Complexity != Time Taken

Explanation:

The above 2 graphs represent the input size and time relationship. For same set of data, both graphs show the linear relationship between time and input size. Now, obviously new machine with better hardware will acjieve the same task faster as compared to the old one. i.e. time taken (new machine) < time taken (old machine).

From the graph ѳ2 < ѳ1. However in both case there is linear relationship between the time and the input size. So the graph is time complexity which is completely different from time taken.

# Running time depends on:

1. Single vs multi processor
2. Read/Write speed to memory
3. 32 bit vs 64 bit
4. Input

We only care about input size while analysing the time complexity of a program.

Input size gives us the rate of growth of time.

# Why do we care so much about time complexity?

The above time vs size graph shows 3 curves: A – Linear, B – Logarithmic and C – Constant. For a big input size, linear time complexity graph shows that it takes more time then comes logarithmic and at last we have constant time complexity. The results for small input size may tell us a different story but we do only care for large input sizes.

# What do we consider when thinking about complexity?

1. Always look for worst case complexity.
2. Always look at complexity for large data.

# Rules for calculating Big O Notation:

1. If an algorithm performs a certain sequence of steps **f(N)** times for a mathematical function **f**, it takes **O(f(N))** steps.
2. If an algorithm performs an operation that takes **f(N)** steps and then performs another operation that takes **g(N)** steps for function f and g, the algorithm’s total performance is is **O(g(N) + f(N))**.
3. If an algorithm takes **O(g(N) + f(N))**steps and the function **f(N)** is bigger than **g(N)**, algorithm’s performance can be simplified to **O(f(N))**.
4. If an algorithm performs an operation that takes **f(N)** steps, and for every step performs another operation that takes **g(N)** steps, algorithm’s total performance is **O(f(N)×g(N))**.

Visit https[://medium.com/@stalonadsl948/big-o-notation-for-dummies-999a7e3fa1ec](https://medium.com/@stalonadsl948/big-o-notation-for-dummies-999a7e3fa1ec)

# Asymptotic Analysis and Asymptotic Notations

Mathematically, asymptotic analysis means analyzing the limiting behavior. In computer science, specifically in algorithm design, asymptotic analysis checks the efficiency of the algorithm with the variation in input sizes. Analysis of algorithm tells about the computational complexities like amount of time, memory taken and other resources to execute the algorithm.

Limiting behavior of a function gives us the properties of that function for a large argument. Using this concept we analyze how our algorithm works out for a very large input sizes.

Asymptotic notations are the standard notations on the basis of which we can discuss the efficiency of the algorithm. There are 3 types of asymptotic notations which are as follows:

1. Big Oh notation O( )
2. Big Omega notation Ω( )
3. Big Theta notation Θ( )

Let’s us discuss all of these one by one

1. Big Oh notation O( )

Big O notation is a mathematical notation that describes the limiting behavior (upper bound) of a function when the argument tends to a particular value or infinity.

If the complexity of a task is O(n3) (read as big o of n cube), it says that it is the upper bound and no matter what it will never exceed O(n3). The same task may be performed in O(n2) or O(n2logn) but it won’t exceed O(n3).

Mathematical treatment

Let f and g are 2 functions in n. Then,

f(n) = O(g(n)) IFF 0 ≤ f(n) ≤ c.g(n) such that there exist constants c,n0 > 0 ∀ n ≥ n0.

NOTE: f(n) cannot grow faster than g(n).

1. Big Omega notation Ω( )

Big Omega notation is just the reverse of the big O notation. This notation gives us the lower bound of the function when the input size grows up to a very large number or infinity. It describes the best case scenario of the algorithm.

Mathematical treatment

Let f and g are 2 functions in n. Then,

f(n) = Ω(g(n)) IFF 0 ≤ c.g(n) ≤ f(n) such that there exist constants c,n0 > 0 ∀ n ≥ n0.

1. Big Theta notation Θ( )

Big Theta notation gives us the average case scenario. It gives us the tight bound for the function.The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation

Mathematical treatment

Let f and g are 2 functions in n. Then,

f(n) = Θ(g(n)) IFF c1.g(n) ≤ f(n) ≤ c1.g(n) such that there exist constants c1,c2,n0 > 0 ∀ n ≥ n0.

