Growth of Functions

By Deepakshi Choudhary

Introduction

Asymptotic notations are used to express the rate of growth of an algorithm's running time, so before discussing growth of functions, let’s understand the running time.

Running Time is represented in terms of input size n, If the input size is n (which is always positive), then the running time is some function f of n. i.e.

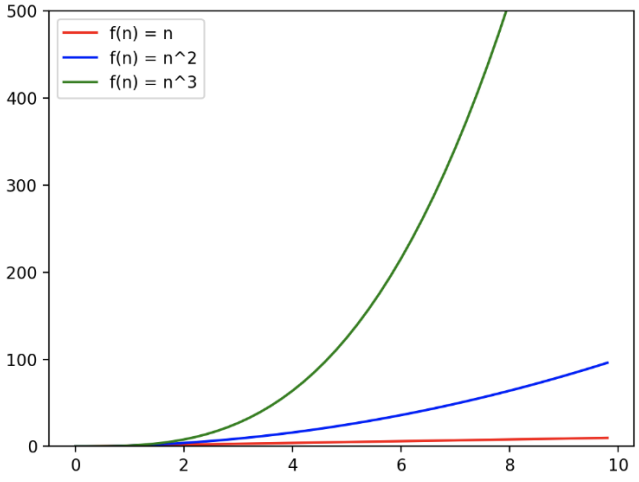
$$\text{Running Time} = f(n)$$

The functional value of f(n) gives the number of operations required to process the input with size n. So the running time would be the number of operations (instructions) required to carry out the given task.
Function f(n) is monotonically non-decreasing. That means, if the input size (means the number of instances in the input) increases, the running time also increases or remains constant. The running time is also called a time complexity.

Growth of Functions

As the running time is expressed in terms of input size (n),so the three possible running times are n,n^2,n^3. Among these three running times, which one is better? To find this out, we need to analyze the growth of the functions.

The easiest way of comparing different running times is to plot them and see the natures of the graph. The following figure shows the graphs of n, n^2 and n^3. (x-axis represents the size of the input and y-axis represents the number of operation required i.e. running time)



In the above figure, we can clearly see the function n^3 is growing faster than functions n and n^2. Therefore, running time n is better than running times n^2, n^3.

Another way of checking if a function f(n) grows faster or slower than another function g(n) is to divide f(n) by g(n) and take the limit n→∞ as follows

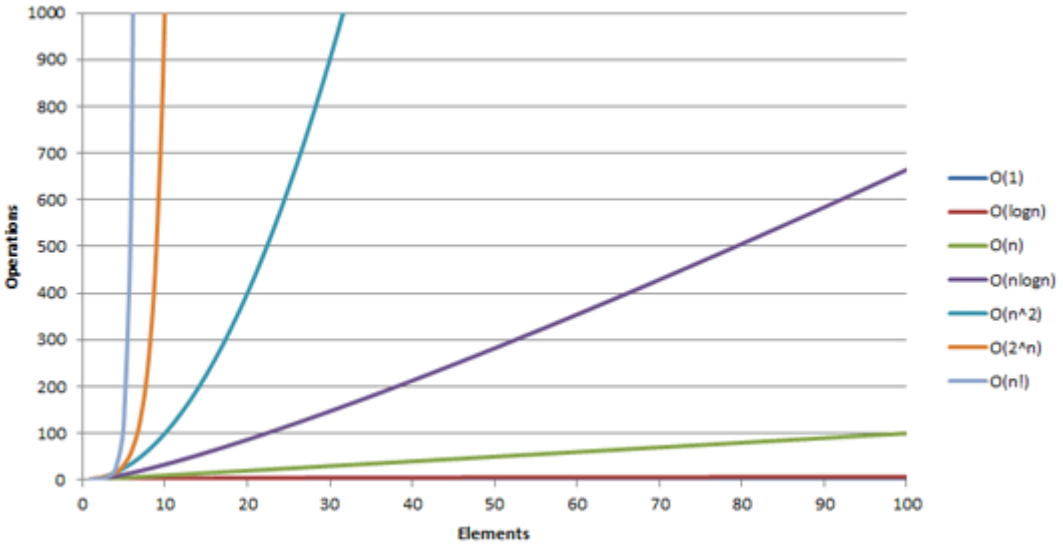
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

If the limit is 0, f(n) grows faster than g(n). If the limit is ∞, f(n) grows slower than g(n).The table below shows common running times in algorithm analysis. The entities in the table are presented from slower to quicker (best to worst) running times.

Running Time	**Examples**
Constant	1, 2, 100, 300, …
Logarithmic	logn , 5 logn, …
Linear	n, n+3 , 2n+3,…
n log n	n logn, 2n logn+n,….
Polynomial	Quadratic, Cubic, or higher order
Exponential	2^n, 3^n, 2^n+ n^4,…..
Factorial	n!, n!+n,…..

The figure below shows the graphical representations of these functions (running times).

1<log <n<n logn<n^2<n^3<2^n<n!



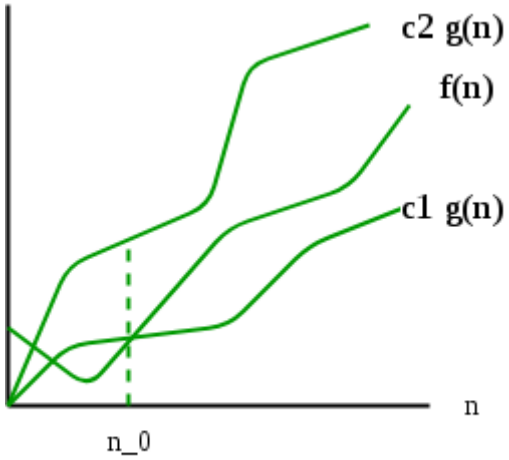
Graph showing the growth of common running times

Asymptotic Notations

Running time are expressed as n^2+3n+2 or 3n etc. These are called exact running time or exact complexity of an algorithm. We are rarely interested in the exact complexity of the algorithm so when we want to find the approximation in terms of upper, lower and tight bound. The O notation gives the upper bound to the exact complexity and denoted by O (Big-o), Θ gives the tight bound on exact complexity and Ω gives the lower bound on exact complexity. There are other two notations o (Little-o) and ω with slight variation in O and Ω. All these notations are described below.

1) **Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants.
   For a given function g(n), we denote Θ(g(n)) is following set of functions.
   Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that
   0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

   The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



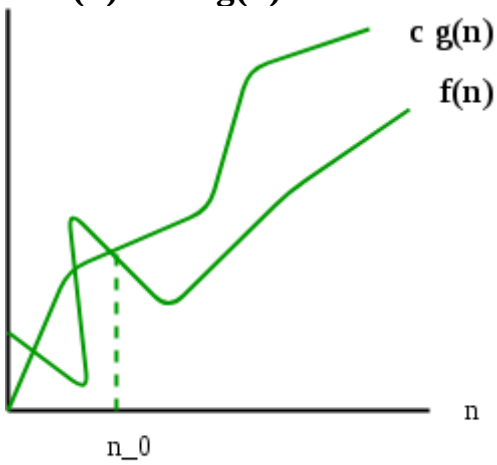
f(n) = theta(g(n))

2) **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. The Big O notation is useful when we only have an upper bound on the time complexity of an

algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0 such that

0 <= f(n) <= c*g(n) for all n >= n0}

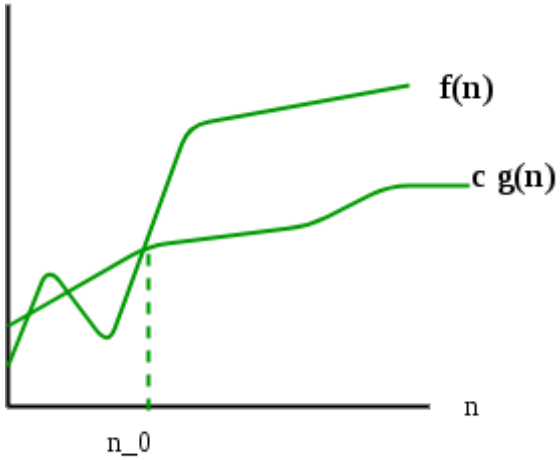


f(n) = O(g(n))

3) **<u>Ω Notation:</u>** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
Ω Notation can be useful when we have a lower bound on the time complexity of an algorithm. But the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.
For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that

0 <= c*g(n) <= f(n) for all n >= n0}.



f(n) = Omega(g(n))

4) **<u>o-notation:</u>** This notation is called Small Oh notation. We use o-notation to denote an upper bound that is not asymptotically tight.
Formally, f(n) is o(g(n)) if there exist constants c and n0 such that

f(n)<c*g(n) for all n<n0

The definitions of O-notation and o-notation are similar. The main difference is that in f(n)=O(g(n)), the bound f(n)≤cg(n) holds for some constant c>0, but in f(n)=o(g(n)), the bound f(n)<cg(n) holds for all constants c>0. Alternatively, f(n) is o(g(n)) if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

5) **ω-notation:** This notation is called Small Omega notation. We use ω-notation to denote an upper bound that is not asymptotically tight. Formally, f(n) is ω(g(n)) if there exist constants c and n0 such that

f(n)>c*g(n) for all n<n0

Alternatively, f(n) is ω(g(n)) if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

- All the analysis we do in the algorithms are only for a large input. It can be wrong when applied to the small input.
- When your program has a small number of input instances, do not worry about the complexity. Use the algorithm that is easier to code.