Algorithms

Asymptotics notation

Notes

Akshay Rajput

1. Algorithm

What is algorithm?

Google says :- "Algorithm is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer."

An algorithm is a procedure composed of a sequence of well-defined steps, specified either in a natural language (a recipe can be regarded as an algorithm), or in appropriate code or pseudo-code.

2. Asymptotic notation

Asymptotic notations are methods/languages using which we can define the run- ning time of the algorithm based on input size. Let us first see how asymptotic notation identifies the behaviour of an algorithm as input size changes. Let us imagine the algorithm as a function f and n as the input size, f(n) will be the running time of the algorithm. Using this you can make is graph with y-axis as running time(f(n)) and x axis as input size(n). Plot some points on the graph for some input. Let us see some of the algorithmic run time in ascending order the fist one being the fastest running (or slowest growing) with input size. Logarithmic Function $\rightarrow \log_2 n$

Linear function $\rightarrow an+b$

QuadraticF unction $\rightarrow an^2 + bn + c$

Polynomial Function $\rightarrow a_1 n^z + ... + a_2 n^2 + a_3 n + a_4$, where a and z are some constant.

These are some basic function growth classifications used in various notations. The list starts at the slowest growing function (logarithmic, fastest execution time) and goes on to the fastest growing (exponential, slowest execution time). Notice that as 'n', or the input, increases in each of those functions, the result clearly increases much quicker in quadratic, polynomial, and exponential, com- pared to logarithmic and linear.

Notice the notation about to be discussed, you should use the simplest terms. This means you need to discard constants and lower order term as they don't weigh much in compared to higher order terms as the input size increases.

So above functions will become:-

Logarithmic $\rightarrow \log_2 n$

Linear $\rightarrow n$

Quadratic $\rightarrow n^2$

Polynomial $\rightarrow n^z$, where z is some constant Exponential $\rightarrow a^n$, where a is some constant

3. Big-O

Big-O commonly wrote as O, is an asymptotic notation for the worst case or the ceiling of growth for a given function. That means the function's complexity will not cross the growth of the asymptotic notation in any case.
Say f(n) is your algorithm run time, and g(n) is an arbitrary time complexity you are trying to relate to your algorithm. Then we can say f(n) is O(g(n)), if for some real constants c (c > 0) and n_0, f(n) <= c g(n) for every input size n (n >= n_0).

Example 1
f(n) = 3 log n + 100
g(n) = log n

Is f(n) = O(g(n))?

Is 3 log n + 100 = O(log n)?

3log n + 100 <= c * log n

Is there some pair of constants c, n_0 that satisfies this for all n >= n_0?

Lets take c = 200

3log n + 100 <= 200 * log n ∀ n > 2 (undefined at n = 1)
Yes! The definition of Big-O has been met therefore f(n) is O(g(n)).

Example 2
f(n) = $3*n^2$
g(n) = n

Is f(n) = O(g(n))? Is $3 * n^2$ = O(n)? Let's look at the definition of Big-O.

$$3 * n^2 <= c * n$$

Is there some pair of constants c, n_0 that satisfies this for all n >= n0? No, there isn't. f(n) is NOT O(g(n)).

4. Big-Omega

Big-Omega, commonly written as Ω, is an Asymptotic Notation for the best case, or a floor growth rate for a given function. It provides us with an asymptotic lower bound for the growth rate of run-time of an algorithm. f(n) is Ω(g(n)), if for some real constants c (c > 0) and n_0 ($n_0 > 0$), f(n) is >= c g(n) for every input size n (n >= n_0).

Note
The asymptotic growth rates provided by big-O and big-omega notation may or may not be asymptotically tight. Thus we use small-o and small-omega notation to denote bounds that are not asymptotically tight.

5. Small-o

Small-o, commonly written as o, is an Asymptotic Notation to denote the upper bound (that is not asymptotically tight) on the growth rate of run time of an algorithm.
f(n) is o(g(n)), if for all real constants c (c > 0) and some n_0 ($n_0 > 0$), f(n) is < c g(n) for every input size n (n > n_0).

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) <= g(n) holds for some constant c > 0, but in f(n) = o(g(n)), the bound f(n) < c g(n) holds for all constants c > 0. Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity;

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

6. Small-omega

Small-omega, commonly written as ω, is an Asymptotic Notation to denote the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

f(n) is ω(g(n)), if for all real constants c (c > 0) and n_0 ($n_0 > 0$), f(n) is > c g(n) for every input size n (n > n_0). The definitions of Ω-notation and ω-notation are similar. The main difference is that in f(n) = Ω(g(n)), the bound f(n) >= g(n) holds for some constant c > 0, but in f(n) = ω(g(n)), the bound f(n) > c g(n) holds for all constants c > 0.

Intuitively, in ω-notation, the function f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

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

7. Theta

Theta, commonly written as Θ, is an Asymptotic Notation to denote the asymp- totically tight bound on the growth rate of runtime of an algorithm.
f(n) is Θ(g(n)), if for some real constants c1, c2 and n_0 (c1 > 0, c2 > 0, n_0 > 0), c1*g(n)< f(n) < c2*g(n) for every input size n (n >= n_0).

∴ f(n) is Θ(g(n)) implies f(n) is O(g(n)) as well as f(n) is Ω(g(n)).
Big-O is the primary notation use for general algorithm time complexity.

8. Comparing functions

Many of the relational properties of real numbers apply to asymptotic compar- isons as well. For the following, assume that f(n) and g(n) are asymptotically positive. Transitivity:

- f(n) = Θ(g(n)) and g(n) = Θ(h(n)) then f(n) = Θ(h(n)).
- f(n) = O(g(n)) and g(n) = O(h(n)) then f(n) = O(h(n)).
- f(n) = Ω(g(n)) and g(n) = Ω(h(n)) then f(n) = Ω(h(n)).
- f(n) = o(g(n)) and g(n) = o(h(n)) then f(n) = o(h(n)).
- f(n) = ω(g(n)) and g(n) = ω(h(n)) then f(n) = ω(h(n)).

Reflexivity

- f(n) = O(f(n))
- f(n) = Θ(f(n))
- f(n) = Ω(f(n))

Symmetry

- f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

NOTE

- f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
- f(n) = o(g(n)) if and only if g(n) = ω(f(n))

Sometimes we may not be able to compare two functions using asymptotic notations for example we cannot compare n and n^{1+sinx}, as in n^{1+sinx} value of 1+sinx oscilates between 0 to 2 taking all values in between.