Other Common Asymptotic Notations and Why Big O Trumps Them

This lesson covers the various asymptotic notations for algorithms and why computer scientists prefer Big O instead of other notations.

We'll cover the following

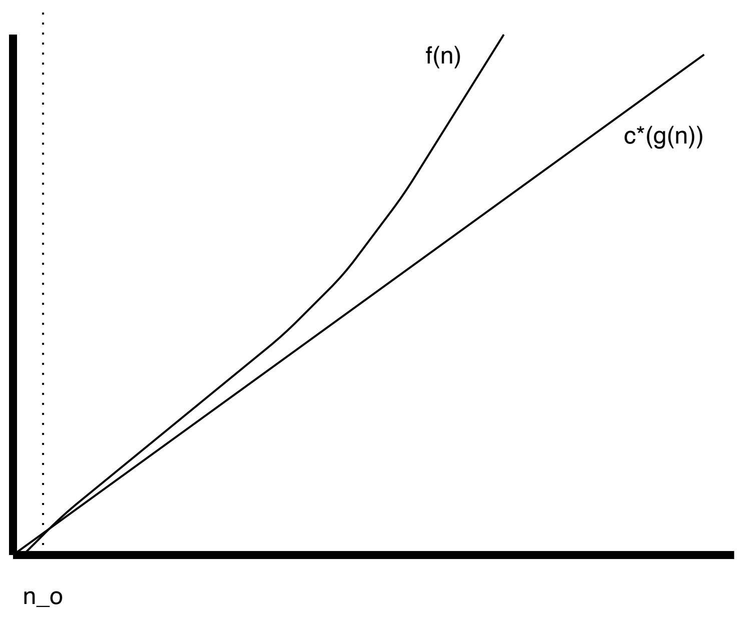
Big ‘Omega’ - $\Omega(.)$#

Mathematically, a function $f(n)$ is in $\Omega(g(n))$ if there exists a real constant $c > 0$ and there exists $n_o > 0$ such that $f(n) \geq cg(n)$ for $n \geq n_o$. In other words, for sufficiently large values of n, $f(n)$ will grow at least as fast as $g(n)$.

> It is a common misconception that Big O characterizes worst-case running time while Big Omega characterizes best-case running time of

> an algorithm. There is no one-to-one relationship between of
> cases and the asymptotic notations.

The following graph shows an example of functions $f(n)$ and $g(n)$ that have a Big Omega relationship.



Big Omega

Quick quiz on Big Omega!

1 $n^3 \in \Omega(1)$

A) True

B) False

Submit Answer

Question 1 of 3
0 attempted

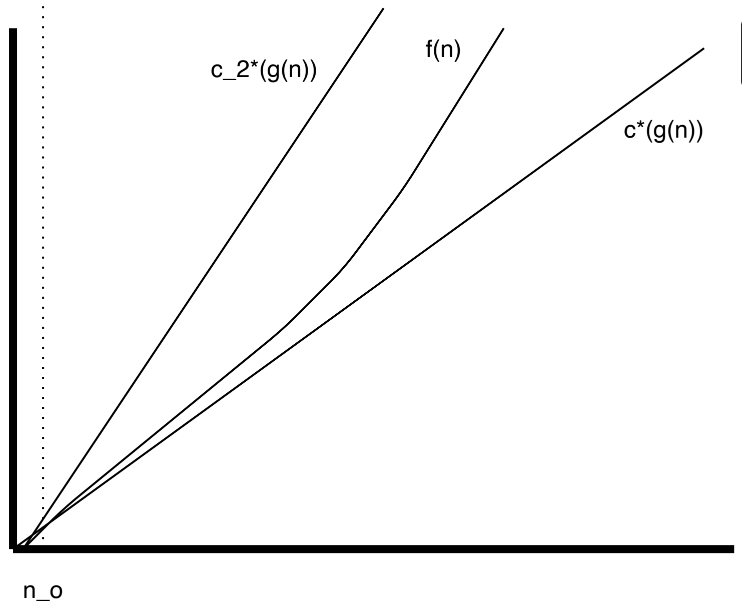
Reset Quiz

Did you know that if $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$? Try proving it!

Big 'Theta' - $\Theta(.)$#

if $f(n)$ is in $O(g(n))$ and $f(n)$ is also in $\Omega(g(n))$ then it is in $\Theta(g(n))$. Formally, $f(n)$ is in $\Theta(n)$ if there exist constants $c_1 > 0$, $c_2 > 0$ and an $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ and $n \geq n_o$. If $f(n)$ is $\Theta(g(n))$, then the two functions grow at the same rate, within constant factors.

So Big Theta is an 'asymptotically tight bound.' When a particular running time is $\Theta(g(n))$, the running time is at least $c_1 g(n)$ and at most $c_2 g(n)$, i.e., it is sandwiched between two functions. Here's how to think of $\Theta(n)$:



Quick quiz on big Θ!

1 $n^2 \in \Theta(n^3)$

A) True

B) False

Submit Answer

Question 1 of 3
0 attempted

Reset Quiz

Little ‘o’ - o(.)#

$f(n)$ is in $o(g(n))$ if for *any* constant $c > 0$ there is an $n_0 >$ h $f(n) < cg(n)$ (strictly *less*) for all $n \geq n_o$. For Big O, it is sufficient that one pair of c and n_0 values exist to satisfy the $f(n) \leq cg(n)$ inequality for all $n \geq n_0$. In case of little o, there must be a value of $n_0 > 0$ for *any* positive choice of c.

The little ‘o’ notation shows that there is a minimum n after which the inequality holds no matter how small you make c, as long as it is not negative or zero.

> **Did you know** $n \notin o(n)$ because n is not strictly less than itself!

Examples of little o:#

$n + 10 \in o(n^2)$

$n + 10 \in o(nlogn)$

$n + 10 \notin o(n)$

$n + 10 \notin o(n - 10)$

Little ‘omega’ - ω(.)#

A function $f(n)$ is in $\omega(g(n))$ if for *any* $c > 0$ there exists an $n_0 > 0$ such that $f(n) > cg(n)$ for $n \geq n_0$.

Examples of little ω:#

$$5n^2 \in \omega(n) \in \omega(\sqrt{n}) \notin \omega(n^2)$$

Why Big ‘O’ is preferred over other notations#

All of these notations have a variety of applications in mathematics. However, in algorithm analysis, we tend to focus on the worst case time and space complexity. It tends to be more useful to know that the **worst case** running time of a particular algorithm will grow **at most** as fast as a constant multiple of a certain function than to know that it will grow **at least** as fast as some other function. In other words, Big Omega is often not very useful for use with worst-case running time.

What about Big Theta, though? Indeed it would be great to have a tight bound on the worst-case running time of an algorithm. Imagine that there is a complex algorithm for which we haven’t yet found a tight bound on the worst-case running time. In such a case, we can’t use Big Theta, but we can still use a loose upper bound (Big O) until a tight bound is discovered. It is common to see Big O being used to characterize the worst-case running time even for simple algorithms where a Big Theta characterization is easily possible. That is not technically wrong, because Big Theta is a subset of Big O.

The little o and little omega notations require a strict level of inequality (< or >) and the ability to show that there is a valid n_0 for any valid of choice of c. This is not always easy to do.

For all the above reasons, it is most common to see Big O being used when talking about an algorithm’s time or space complexity.

> **Note:** Now, we will simplify the analysis by just counting t m f executions of each line of code, instead of the number of operations. For example, we will say `print(sum)` will take one primitive operation instead of two.

In the next lesson, we will discuss some useful formulas that will help you calculate the time complexity of algorithms.

Back
Introduction to Asymptotic Analysis a...

Next
Useful Formulas

Completed

Report an Issue