Logarithms and Big-O

Sam McCauley
Algorithms, Spring 2025

Logarithms

The definition of a logarithm is that if $b^x = y$, then $\log_b y = x$.

Let's recall some classic log rules:

Theorem 1. For any $a, b, x, y > 0$:

$$\log_b(xy) = \log_b x + \log_b y \qquad \log_b(x/y) = \log_b x - \log_b y$$

$$\log_a x = \frac{\log_b x}{\log_b a} \qquad \log_b(x^y) = y \log_b x$$

Sometimes in this course we may see some somewhat complicated expressions involving logarithms, especially when there are logarithms in exponents. My recommendation to simplify them is to use the above log rules to: (1) make all the logs base 2, and (2) put everything into the exponent with 2 as a base.

Let's see some an example of this idea in action.

Example. Let's simplify $n^{1/\log_2 n}$.

Using the above idea, let's make the equation into an exponent with 2 as the base. Substituting $n = 2^{\log n}$, we have that $n^{1/\log_2 n} = (2^{\log n})^{1/\log_2 n}$. From algebra, we know that $(a^b)^c = a^{bc}$, so $(2^{\log_2 n})^{1/\log_2 n} = 2^{(\log_2 n)/\log_2 n} = 2$.

Therefore, $n^{1/\log_2 n} = 2$.

Sometimes, the "wrong" variable is in the base of the number. The following theorem is a black box way to fix it, and its proof uses this same idea of putting everything into an exponent of 2.

Theorem 2. For any $a, b, c > 0$,

$$a^{\log_b c} = c^{\log_b a}.$$

Proof. Let's move everything into the base of 2.

$$a^{\log_b c} = (2^{\log_2 a})^{(\log_2 c)/\log_2 b} = 2^{\log_2 a \cdot \log_2 c/\log_2 b}.$$

Now, a similar sequence of steps get us to the right side of the equation from the theorem.

$$2^{\log_2 a \cdot \log_2 c/\log_2 b} = (2^{\log_2 c})^{\log_2 a/\log_2 b} = c^{\log_b a}.$$

□

Big O Notation

Big O is a way to talk about functions that is particularly useful for analyzing the running time of an algorithm. In short: when analyzing the running time, we care about how the algorithm behaves on large inputs—we do not care about lower-order terms, or multiplying by constants, or the behavior when n is small.

In a formal sense, we compare two *functions* using big-O notation. Oftentimes, $f(n)$ represents the number of steps an algorithm takes.

Definition 1. $f(n)$ is $O(g(n))$ if there exist constants c and n_0 such that:

$$\forall n > n_0, \;\; f(n) \leq c \cdot g(n)$$

You can use the following results without proof in this course.[1]

Theorem 3.

- If the function is a polynomial, it is big-O of the largest exponent.
  - **Example:** $.3n^5 + 1000n^2 + 2n = O(n^5)$
- Logs are big-O of any polynomial.
  - **Example:** $\log n = O(n^{.01})$
- Exponents are larger than any polynomial: a polynomial is big-O of any exponential func-

[1] You may have seen in the past that expressions like $f(n) = O(g(n))$ are strictly speaking incorrect: one ought to write $f(n) \in O(g(n))$. Writing $=$ is nonetheless standard practice.

tion.

  - **Example:** $n^{100} = O(2^n)$

- Any constant independent of n can be represented with $O(1)$
  - **Example:** $2000 = O(1)$ or $.01 = O(1)$.

Bear in mind when using big-O is that it's just an upper bound—even though we write $=$, you should think of it as $\leq$.

Example. The following statements are all true:

- $n = O(n)$
- $n = O(n \log n)$
- $n = O(n^2)$
- $100n = O(n^2)$
- $\log n = O(2^n)$

But all of these statements are *false*—for large n, the function on the left is larger, even ignoring constants:

- ✗ $n \log n = O(n)$
- ✗ $n^2 = O(n)$
- ✗ $n^2 = O(100n)$
- ✗ $2^n = O(\log n)$

One tricky part about big-O is that when we say we "do not care about constants," we mean *multiplying* by a constant, like $100n$ vs n. We do care about the difference between 2^n and (say) 6^n.

Example. We have that $2^n = O(6^n)$; however, $6^n \neq O(2^n)$.

Proof. We have that $2^n = O(6^n)$ because $2^n \leq 6^n$ for $n \geq 1$ (so we can take $c = 1$ and $n_0 = 1$ in the definition of big-O).

Now, we show that $6^n \neq O(2^n)$. Assume the contrary: there exist constants c and n_0 such that

for all $n > n_0$, $c2^n > 6^n$. Dividing both sides by 2^n, we have that $c > 6^n/2^n$, which we can rewrite as $c > 3^n$. But this is not true for any $n > \log_3 c$, so it cannot be true for all $n > n_0$. □

Big Ω and Big Θ

Students are often surprised to hear that big-O is merely an upper bound since that's not how they've used it in the past (and in all fairness, often not how I use it either).

When someone says "insertion sort takes $O(n^2)$ time," it's reasonable to take it as "this algorithm takes roughly n^2 time" rather than "this algorithm takes at most roughly n^2 time, but it could take less. Maybe it even takes $O(n)$ time. ¯_(ツ)_/¯" That said, the strict definition of big-O is this last interpretation.

Big-Ω and big-Θ notation address this gap.

First, big-Ω notation (this is a capital Greek letter omega; in latex you can make this symbol with `\Omega`) is essentially identical to big-O notation, but with $\geq$ rather than $\leq$. (The definitions are identical but the inequality is swapped.)

Definition 2. $f(n)$ is $O(g(n))$ if there exist constants c and n_0 such that:

$$\forall n > n_0, \;\; f(n) \geq c \cdot g(n)$$

This is used for lower bounds: one could say "any comparison-based sorting algorithm takes at least $\Omega(n \log n)$ time". It can also be used for things other than running time: one could say "any connected graph with n vertices has[2] $\Omega(n)$ edges".

Big-Θ notation (this is a capital Greek letter theta; in latex you can use `\Theta`) means that two functions are asymptotically the *same*. This is what people often mean when they say big-O.

Definition 3. $f(n)$ is $\Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

One could say "the worst-case running time of insertion sort is $\Theta(n^2)$", or "the number of possible ways to make a group of three out of n students is $\Theta(n^3)$".

[2] To be precise, it has $\geq n - 1$ edges; therefore one can say it has $\Omega(n)$ edges.