Logarithms

How to think about them, especially in programming interviews and algorithm design

What logarithm even *means*

Here's what a logarithm is asking:

"What power must we raise this base to, in order to get this answer?"

So if we say:

$$\log_{10} 100$$

The 10 is called the *base* (makes sense—it's on the bottom). Think of the 100 as the "answer." It's what we're taking the log *of*. So this expression would be pronounced "log base 10 of 100."

And all it means is, "What power do we need to raise this base (10) to, to get this answer (100)?"

$$10^x = 100$$

What x gets us our result of 100? The answer is 2:

$$10^2 = 100$$

So we can say:

$$\log_{10} 100 = 2$$

> The "answer" part could be surrounded by parentheses, or not. So we can say $\log_{10}(100)$ or $\log_{10} 100$. Either one's fine.

What logarithms are *used for*

The main thing we use logarithms for is **solving for x when x is in an exponent**.

So if we wanted to solve this:

$$10^x = 100$$

We need to bring the x down from the exponent somehow. And logarithms give us a trick for doing that.

We take the $\log_{10}$ of both sides (we can do this—the two sides of the equation are still equal):

$$\log_{10} 10^x = \log_{10} 100$$

Now the left-hand side is asking, "what power must we raise 10 to in order to get 10^x?" The answer, of course, is x. So we can simplify that whole left side to just "x":

$$x = \log_{10} 100$$

We've pulled the x down from the exponent!

Now we just have to evaluate the right side. What power do we have to raise 10 to do get 100? The answer is still 2.

$$x = 2$$

That's how we use logarithms to pull a variable down from an exponent.

Logarithm rules

These are helpful if you're trying to do some algebra stuff with logs.

Simplification: $\log_b(b^x) = x$ . . . *Useful for bringing a variable down from an exponent.*

Multiplication: $\log_b(x * y) = \log_b(x) + \log_b(y)$

Division: $\log_b(x/y) = \log_b(x) - \log_b(y)$

Powers: $\log_b(x^y) = y * \log_b(x)$

Change of base: $\log_b(x) = \frac{\log_c(x)}{\log_c(b)}$ . . . *Useful for changing the base of a logarithm from b to c.*

Where logs come up in algorithms and interviews

"How many times must we double 1 before we get to n" is a question we often ask ourselves in computer science. Or, equivalently, **"How many times must we divide n in half in order to get back down to 1?"**

> Can you see how those are the same question? We're just going in different directions! From n to 1 by dividing by 2, or from 1 to n by multiplying by 2. Either way, it's the same *number of times* that we have to do it.

The answer to both of these questions is $\log_2 n$.

It's okay if it's not obvious yet why that's true. We'll derive it with some examples.

Logarithms in binary search

This comes up in the time cost of **binary search**, which is an algorithm for finding a target number in a *sorted* list. The process goes like this:

1. **Start with the middle number: is it bigger or smaller than our target number?** Since the list is sorted, this tells us if the target would be in the *left* half or the *right* half of our list.
2. **We've effectively divided the problem in half**. We can "rule out" the whole half of the list that we know doesn't contain the target number.
3. **Repeat the same approach (of starting in the middle) on the new half-size problem**. Then do it again and again, until we either find the number or "rule out" the whole set.

So the question is, "how many times must we divide our original list size (n) in half until we get down to 1?"

$$n * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \ldots = 1$$

How many $\frac{1}{2}$'s are there? We don't know yet, but we can call that number x and solve for it:

$$n * \left(\frac{1}{2}\right)^x = 1$$

$$n * \frac{1^x}{2^x} = 1$$

$$n * \frac{1}{2^x} = 1$$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

Now to get the x out of that exponent! We'll use the same trick. Take the $\log_2$ of both sides...

$$\log_2 n = \log_2 2^x$$

The right hand side asks, "what power must we raise 2 to, to get 2^x?" Well, that's just x.

$$\log_2 n = x$$

So there it is. The total time cost of binary search is $O(\log_2 n)$.