Logarithms in algorithms

Overview

In algorithms, you may notice logarithms appear frequently such as $\log(n)$. Why do they appear and how do they work? These notes seek to explain logarithms and how to work with them.

First lets cover what they are. Logarithms are the inverse of exponents.

$$\log_b(n) = a \iff b^a = n$$

They evaluate the needed exponent for a given base and answer (or argument). For example, suppose we want to find out that 2 to the power of what gives us 8, we can use logs:

$$\log_2(8) = 3 \iff 2^3 = 8$$

Also note the following for logs:

$$\log(n) = \log_{10}(n) \tag{1}$$
$$\ln(n) = \log_e(n) \tag{2}$$

The first one just means that log without a base is implicitly 10, and the second one ln refers to the natural logarithm with base $e = 2.71828$ (Euler's number). Do not confuse ln as shorthand for log.

To convert from exponential to log form, log both sides with the same base the exponent is raised to as such:

$$2^3 = 8 \tag{1}$$
$$\log_2(2^3) = \log_2(8) \qquad \log_a(b^c) = c\log_a(b) \tag{2}$$
$$3\log_2(2) = \log_2(8) \qquad \log_a(a) = 1 \tag{3}$$
$$3 = \log_2(8) \tag{4}$$

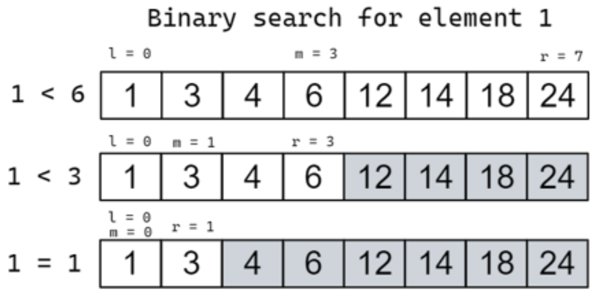
These are the full steps, but the idea is the log and the argument gets cancelled out. So you can pretty much go from step 2 to step 4. To convert back to exponential form, exponentiate both sides.

$$3 = \log_2(8) \tag{1}$$
$$2^3 = 2^{\log_2(8)} \qquad a^{log_a(n)} = n \tag{2}$$
$$2^3 = 8 \tag{3}$$

The log will cancel out the base and return the argument.

When do logarithms appear?

Generally, logs will appear in algorithms where there is either repeated division or multiplication in regards to the problem's input size. Most commonly $\log_2$ will but other logs such as $\log_3$ can appear too. Take for example binary search for an array of size 8:



We can see the total elements remaining for being selected is halved on each iteration: $8 \rightarrow 4 \rightarrow 2$. The exact number of operations binary search performs in the worst-case is $\lfloor \log_2(n) + 1 \rfloor$ but this example clearly shows how exponent come in the picture. If we just ignore the $+1$ for now, it took us 3 iterations (including the first step) to go from 8 to 2, which is what exactly a log will represent for us, so we can express this as $\log_2(8) = 3$.

Here is another example of a logarithmic algorithm:

Algorithm 1 Mystery(n)

```
$S = 0$
**while** $i = 1$ less than equal to $n$ **do**
    $S = S + 2$
    $i = i * 2$
**return** S
```

Lets make addition the basic operation addition and $n = 16$. Lets count how many times addition gets executed:

- $i = 1 \leq 16$ is executed
- $i = 2 \leq 16$ is executed
- $i = 4 \leq 16$ is executed
- $i = 8 \leq 16$ is executed
- $i = 16 \leq 16$ is executed
- $i = 32 \leq 16$ is not executed

Since i is doubling on each input, it performed 5 executions from iterations 1 to 16. In exponential form, we're asking $2^x = 16$ that is how many times can we double 2 until we reach 16? By representing this as a log we get $\log_2(16) = 4$ and we would also need to $+1$ to count for when $i = 1$ so the exact number of operations is $\lfloor \log_2(n) + 1 \rfloor$ which belongs in complexity class of $\Theta(\log n)$.

Notice for both the binary search and the loop, the binary search size went downwards from 8 and the loop went up from 2, regardless if we're going up or down we can represent it using logs.

If we modify this algorithm to instead multiply i by 3:

Algorithm 2 Mystery(n)

```
  $S = 0$
  **while** $i = 1$ less than equal to $n$ **do**
    $S = S + 2$
    $i = i * 3$
  **return** S
```

Then the number of operations this algorithm performs would then be $\lfloor \log_3(n) + 1 \rfloor$. In fact:

Algorithm 3 Mystery(n, k)

where k is an integer > 1

```
  $S = 0$
  **while** $i = 1$ less than equal to $n$ **do**
    $S = S + 2$
    $i = i * k$
  **return** S
```

Will perform $\lfloor \log_k(n) + 1 \rfloor$ operations, all of which belong to $\Theta(\log n)$.

Why we drop the base in asymptotic notations?

When expressing logarithmic functions using asymptotic notations, we can omit the base. This is simply because of the base of change rule for logs which is:

$$\log_a(n) \cdot \log_b(a) = \log_b(n) \tag{1}$$
$$\text{or}$$
$$\log_a(n) / \log_a(b) = \log_b(n) \tag{2}$$

In the first rule, $\log_b(a)$ and in the second rule, $\log_a(b)$ will both evaluate to constants, and constants do not affect a functions order of growth in asymptotic notations.

Logarithms in recursive functions

Recursive algorithms that divide the input size for each recursive call will exhibit logarithmic order of growth. Lets take a basic example:

$$C(n) = C(n/2) + 1 \text{ for } n > 1, \quad C(1) = 1$$

Solving this via backwards substitution:

$$\begin{aligned}
C(n) &= C(n/2) + 1 && && (1)\\
&= C(n/2) + 1 && \text{sub } C(n/2) = C(n/2^2) + 1 && (2)\\
&= [C(n/2^2) + 1] + 1 = C(n/2^2) + 1 + 1 && \text{sub } C(n/2^2) = C(n/2^3) + 1 && (3)\\
&= [C(n/2^3) + 1] + 1 + 1 = C(n/2^3) + 1 + 1 + 1 && && (4)\\
&= \dots && && (5)\\
&= C(n/2^i) + i && && (6)
\end{aligned}$$

At this point, we need $n/2^i$ to equal the initial of condition of 1. We know for fractions that if $x/x = 1$, so we need $2^i = n$. One way is to log both sides as we shown above and we get $i = \log_2(n)$. The other way is using the log law $a^{\log_a(n)} = n$, which means a number raised to a log power with the same base will cancel each other out and return the result. So we set $i = \log_2(n)$ and substitute this in:

$$= C(1) + \log_2(n) \tag{7}$$
$$= 1 + \log_2(n) \in \Theta(\log n) \tag{8}$$

Assuming the base case is $C(1)$, any division recurrence relation n/b, i will equal $\log_b(n)$ (note this does not mean the complexity class of the recurrence relation will always be $\Theta(\log n)$. This is clear to see since once we we establish the pattern to be $n/b^i = 1$, we will just solve for $b^i = n$. Just like the binary search above, n is being halved which is how logs come out of division recurrence relations.

One other quick mention is how the textbook we use *"Introduction to the Design & Analysis of Algorithms"* solves recurrence relations with division. Instead of raising the denominator to a power, such as $n/2^2$, $n/2^3$, it will refer to a theorem called the **smoothness rule** which claims under broad assumptions that the order of growth for $n = 2^k$ (assuming we're using 2 as the denominator in our recurrence) will hold for all values of n, not just for those that aren't powers of 2. In terms of the result, there is no difference but it may be easier to express more complex summations using this rule. Simply set $n = 2^k$ and use exponent rules to solve the recurrence relation (recall exponent laws that is $2^k/2 = 2^{k-1}$):

$$\begin{aligned}
C(n) &= C(2^{k-1}) + 1 && && (1)\\
&= C(2^{k-1}) + 1 && \text{sub } C(2^{k-1}) = C(2^{k-2}) + 1 && (2)\\
&= [C(2^{k-2}) + 1] + 1 = C(2^{k-2}) + 1 + 1 && \text{sub } C(2^{k-2}) = C(2^{k-3}) + 1 && (3)\\
&= [C(2^{k-3}) + 1] + 1 + 1 = C(2^{k-3}) + 1 + 1 + 1 && && (4)\\
&= \ldots && && (5)\\
&= C(2^{k-i}) + i && a^0 = 1, \text{ set } i = k && (6)\\
&= C(2^{k-k}) + k && && (7)\\
&= C(1) + k && && (8)
\end{aligned}$$

Recall we set $n = 2^k$, so we need to substitute this back. Apply the same log rules we just discussed, $k = \log_2(n)$ and we get:

$$C(n) = 1 + \log_2(n) \in \Theta(\log n) \tag{9}$$

Log laws

$b > 0$ but $b \neq 1$ and m, n, k are real numbers, m and n are positive:

$$\log_b(m \cdot n) = \log_b(m) + \log_b(n) \tag{1}$$
$$\log_b(m/n) = \log_b(m) - \log_b(n) \tag{2}$$
$$\log_b(m^a) = a \log_b(m) \tag{3}$$
$$\log_b(1) = 0 \tag{4}$$
$$\log_b(b^k) = k \text{ or } \log_b(b) = 1 \tag{5}$$
$$b^{\log_b(k)} = k \tag{6}$$

Change of base rule as mentioned above:

$$\log_a(n) \cdot \log_b(a) = \log_b(n) \tag{7}$$
$$\log_a(n) / \log_a(b) = \log_b(n) \tag{8}$$

For explanation of these laws, there are many good resources online so they won't be explained here.