Logarithms

If $v = 10^k$, then $\log_{10}(v) = k$.

The name "log" is short for "logarithm". This operator was introduced by John Napier in his 1614 book titled *Mirifici logarithmorum canonis descriptio* (*A Description of the Wonderful Table of Logarithms*).

The logarithm operator is the inverse of exponentiation: 10^k is 10 raised to the power k, $\log(10^k)$ is k. The number 10 is called the *base* of the logarithm. Besides the notation $\log_{10}(v)$, one sees $\log_{10} v$ and, when the base is obvious from the context, $\log(v)$ and $\log v$. In Java, $\log_{10}(v)$ can be calculated using function Math.log10(v).

Another commonly used base is e, the mathematical constant 2.71828… whose value is the limit as n approaches infinity of $(1 + 1/n)^n$. In Java, use Math.E for e and Math.log(v) for $\log_e(v)$. Although e and *log e* are extremely important in mathematics, they are not used much in dealing with data structures, and we won't mention them again.

Log base 2, that is, $\log_2(...)$, arises when analyzing the time or space complexities of several algorithms. We discuss only what you need to know about $\log_2(...)$ to understand its use in analyzing these time and space complexities. From now on, we use the notation $\log v$ for $\log_2(v)$.

Processing the bits of a positive integer

Recall (look at JavaHyperText entry "binary number system") that the number 2^k for k a natural number is 1 followed by k 0's. For example, $2^5 = 32_{10} = 100000_2$. Therefore, any integer v in the range $2^{k-1} \leq v < 2^k$ requires exactly k bits. Thus, v requires *ceil*(log v) bits, where *ceil* is the *ceiling function*, which raises its argument, if necessary, to the next highest integer. (In Java, use function Math.ceil.)

> Suppose $v = 10$. Then $\log v = 3.321928094887362\ldots$ and *ceil*(log v) = 4.
> Suppose $v = 16$. Then $\log v = 4$ and *ceil*(log v) = 4.

Because of this, we see that v requires O(log v) bits when written in binary.

Algorithms that halve an integer

Binary search, sorting method *merge sort*, and an efficient exponentiation algorithm all work (roughly) by continually halving an integer. So, let us consider any algorithm that starts with $v = 2^k$ (with k a natural number 0, 1, 2, …) and at each step cuts v in half, stopping when $v = 1$. After one step, $v = 2^{k-1}$; after two steps, $v = 2^{k-2}$; and so on. Exactly k steps will be done. That's $\log v$ steps.

The algorithm can also be executed when v is not a power of 2 but lies in this range: $2^{k-1} < v < 2^k$. Halving will be done using Java **int** arithmetic, $v/2$. After one step, we have $2^{k-2} \leq v < 2^{k-1}$, after two steps, $2^{k-3} \leq v < 2^{k-2}$, and so on. Again, k steps will be executed.

From this, we infer that this halving algorithm executes exactly *ceil*(log v) steps, which is O(log v) steps. We will use this to help develop the time or space complexities of the aforementioned algorithms.

Algorithms that double an integer

Let $n = 2^p$, so $\log n = p$. Value p need not be an integer. Consider an algorithm that starts with **int** $k = 1 = 2^0$ and doubles it until $k \geq n$ for a given integer n. Thus, k takes on the values $2^0, 2^1, ..., 2^{ceil(p)}$. Variable k gets doubled *ceil*(p) = *ceil*(log n) times. That's O(log n) times.

An important identity

Here is an identity concerning logarithms:

$$\log_{10} x = (\log_2 x) / (\log_2 10) = (\log_2 x) / (3.321928094887362\ldots)$$

From this, we infer that $\log_{10} x$ is $O(\log_2 x)$ and $\log_2 x$ is $O(\log_{10} x)$.

The importance of logarithmic versus linear algorithms

Suppose we have two algorithms for searching an array of size n. One takes linear time, O(n), and the other takes logarithmic time, O(log n). Suppose $n = 32768 = 2^{15}$. The linear-time algorithm could take roughly 32768 steps, the logarithmic algorithm only 15. What a difference!