Running Time of Binary Search

We know that linear search on an array of nn elements might have to make as many as nn guesses. You probably already have an intuitive idea that binary search makes fewer guesses than linear search. You even might have perceived that the difference between the worst-case number of guesses for linear search and binary search becomes more striking as the array length increases. Let's see how to analyze the maximum number of guesses that binary search makes.

The key idea is that when binary search makes an incorrect guess, the portion of the array that contains reasonable guesses is reduced by at least half. If the reasonable portion had 32 elements, then an incorrect guess cuts it down to have at most 16. Binary search halves the size of the reasonable portion upon every incorrect guess.

If we start with an array of length 8, then incorrect guesses reduce the size of the reasonable portion to 4, then 2, and then 1. Once the reasonable portion contains just one element, no further guesses occur; the guess for the 1-element portion is either correct or incorrect, and we're done. So with an array of length 8, binary search needs at most four guesses.

What do you think would happen with an array of 16 elements? If you said that the first guess would eliminate at least 8 elements, so that at most 8 remain, you're getting the picture. So with 16 elements, we need at most five guesses.

By now, you're probably seeing the pattern. Every time we double the size of the array, we need at most one more guess. Suppose we need at most **m** guesses for an array of length **n**. Then, for an array of length **2n**, the first guess cuts the reasonable portion of the array down to size **n**, and at most **m** guesses finish up, giving us a total of at most **m+1** guesses.

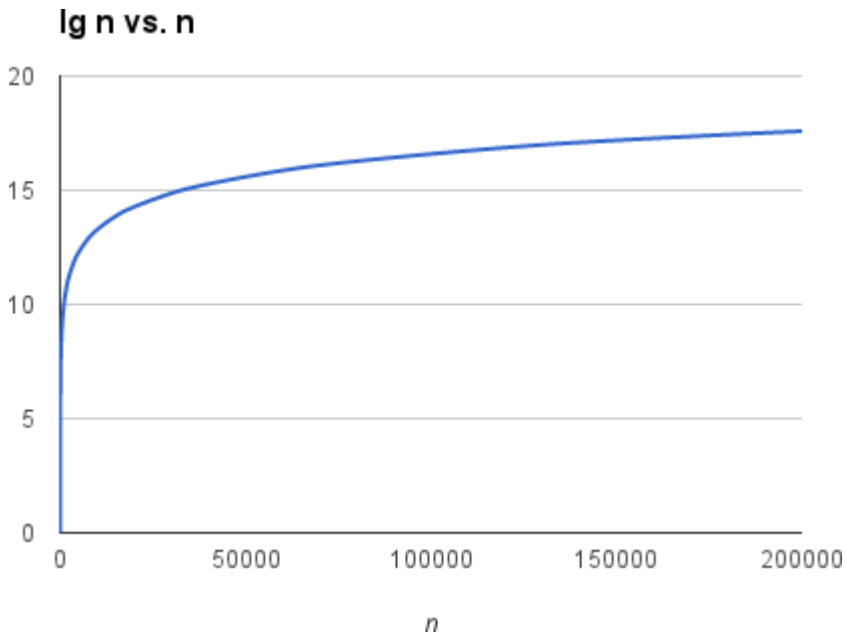
Let's look at the general case of an array of length **n**. We can express the

Let's look at the general case of an array of length **n**. We can express the number of guesses, in the worst case, as "the number of times we can repeatedly halve, starting at **n**, until we get the value 1, plus one." But that's inconvenient to write out. Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at **n**, until we get the value 1: the ***base-2 logarithm*** of **n**. We write it as **lg n**. (You can learn more about logarithms here.)

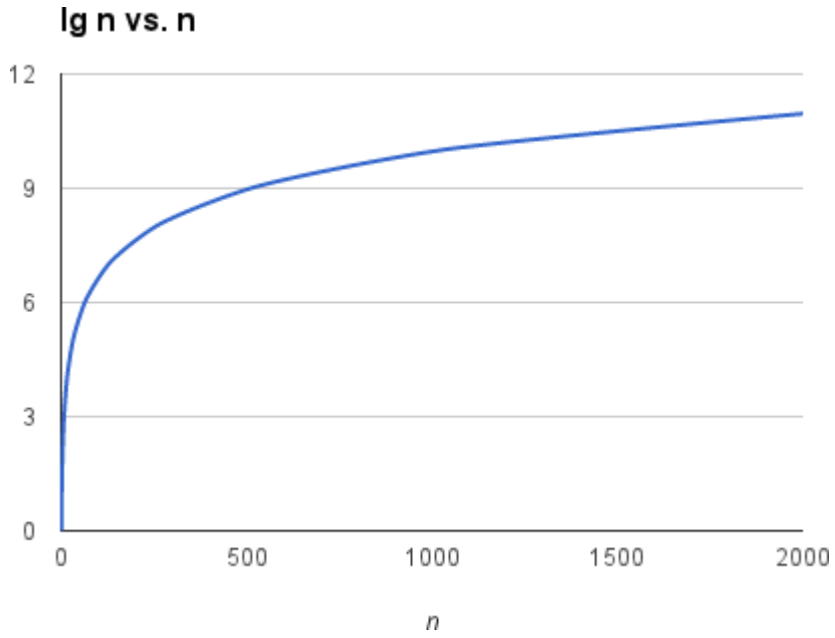
Here's a table showing the base-2 logarithms of various values of ***n***:

n	lg n
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21

We can view this same table as a chart:

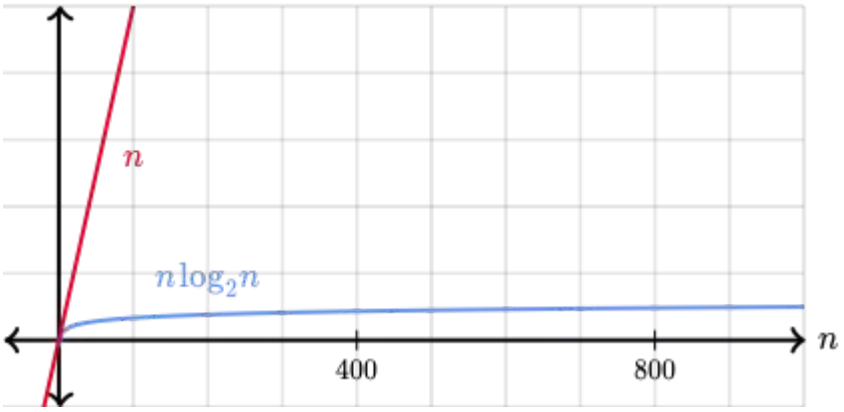


Zooming in on smaller values of n:



The logarithm function grows very slowly. Logarithms are the inverse of exponentials, which grow very rapidly, so that if **lgn=x**, then **n** = 2^{x}. For example, because **lg 128** = 7, we know that 2^{7} = **128**.

When **n** is not a power of 2, we can just go up to the next higher power of 2. For an array whose length is 1000, the next higher power of 2 is 1024, which equals 2^{10} . Therefore, for a 1000-element array, binary search would require at most 11 (10 + 1) guesses. For the Tycho-2 star catalog with 2,539,913 stars, the next higher power of 2 is 2^{22} (which is 4,194,304), and we would need at most 23 guesses. Much better than linear search! Compare them below:



In the next tutorial, we'll see how computer scientists characterize the running times of linear search and binary search, using a notation that distills the most important part of the running time and discards the less important parts.