# Understanding the Logarithmic Time Complexity of Binary Search

August 11, 2023

For simplicity, let's consider a list of length $N$ where $N$ is a power of 2 (e.g., 2, 4, 8, 16,...). When you perform a binary search, you start with $N$ elements, then reduce it to $\frac{N}{2}$, then $\frac{N}{4}$, and so on until you're down to 1 element.

Here's the breakdown:

- After 1st comparison: $\frac{N}{2}$ elements remain.
- After 2nd comparison: $\frac{N}{4}$ elements remain.
- After 3rd comparison: $\frac{N}{8}$ elements remain.
- ...
- After k-th comparison: $\frac{N}{2^k}$ elements remain.

When the size becomes 1 (i.e., $\frac{N}{2^k} = 1$), we have essentially located the desired element (or know it's absent). Now, if you solve for $k$ in the above equation, you get:

$N = 2^k$

Taking the base-2 logarithm on both sides, you get:

$\log_2 N = k$

So, $k$, the number of operations or comparisons in this case, is proportional to $\log N$. That's why we say the time complexity of the binary search is $O(\log N)$.

For other algorithms or processes where the problem size gets divided by some constant factor other than 2 (let's say 3 or 10 or any other number), the idea remains similar. We would then talk in terms of $\log_3 N$ or $\log_{10} N$, etc., but in computational complexity, the base of the logarithm is often considered less relevant (due to the logarithm properties) and we just use $\log N$ to represent logarithmic complexity.