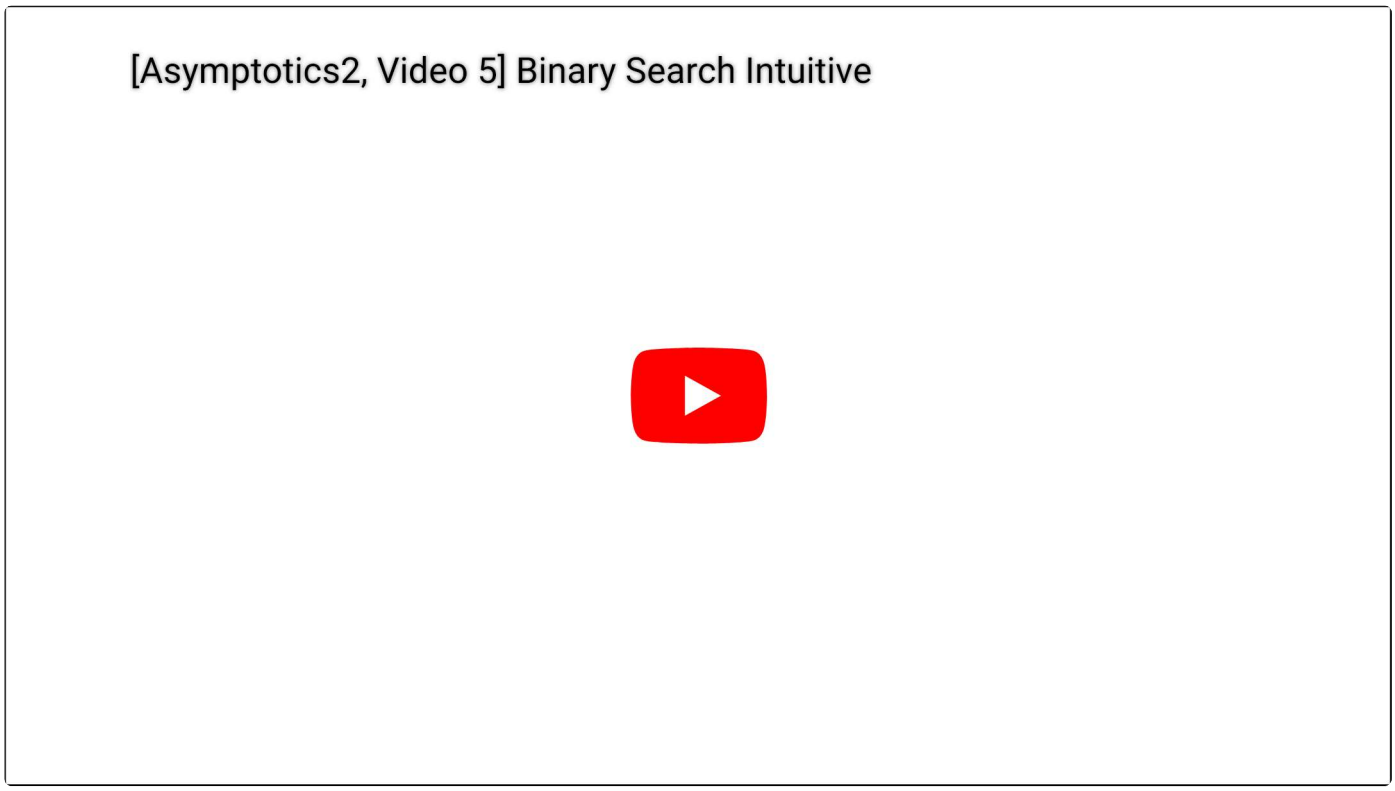
15.3 Binary Search

hi-lo!

Binary Search Introduction



Getting Familiar with Binary Search

Binary search is a nice way of searching a list for a particular item. It requires the list to be in sorted order and uses that fact to find an element quickly.

To do a binary search, we start in the middle of the list, and check if that's our desired element. If not, we ask: is this element bigger or smaller than our element?

If it's bigger, then we know we only have to look at the half of the list with smaller elements. If it's too small, then we only look at the half with bigger elements. In this way, we can cut in half the number of options we have left at each step, until we find our target element.

What's the worst possible case? When the element we want isn't in the list at all. Then we will make comparisons until we've eliminated all regions of the list, and there are no more

bigger or smaller halves left.

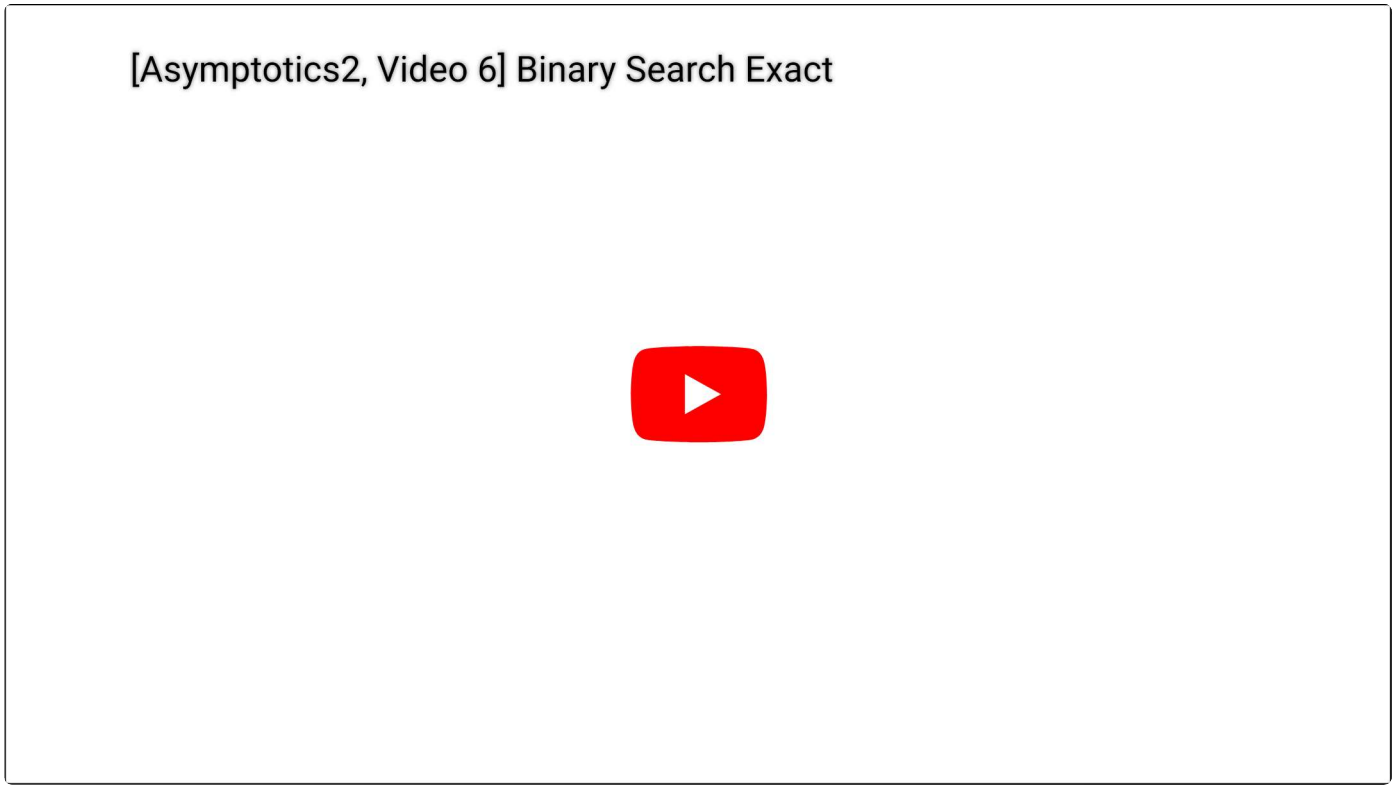
For an animation of binary search, see these slides.

What's the intuitive runtime of binary search? Take a minute and use the tools you know to consider this.

We start with N options, then $N/2$, then $N/4$ ... until we have just 1. Each time, we cut the array in half, so in the end we must perform a total of $log_2(N)$operations. Each of the $log_2(N)$ operations, eg. finding the middle element and comparing with it, takes constant time. So the overall runtime then is order $log_2(N)$.

It's important to note, however that each step doesn't cut it *exactly* in half. If the array is of even length, and there is no 'middle', we have to take either a smaller or a larger portion. But this is a good intuitive approach.

We'll do a precise way next.



A more precise analysis of Binary Search runtime

To precisely calculate the runtime of binary search, we'll count the number of operations, just as we've done previously.

First, we define our cost model: let's use the number of recursive binary search calls. Since the number of operations inside each call is constant, the number of calls will be the only thing varying based on the size of the input, so it's a good cost model.

Like we've seen before, let's do some example counts for specific N. As an exercise, try to fill this table in before continuing:

N	1	2	3	4
Count				

Alright, here's the result:

N	1	2	3	4
Count	1	2	2	3

These seems to support our intuition above of $log_2(N)$. We can see that the count seems to increase by one only when N hits a power of 2.

...but we can be even more precise: $C(N) = \lfloor log_2(N) \rfloor + 1$ (These L-shaped bars are the "floor" function, which is the result of the expression rounded down to the nearest integer.)

A couple properties worth knowing (see below for proofs):

- $\lfloor f(N) \rfloor = \Theta(f(N))$
- $\lceil f(N) \rceil = \Theta(f(N))$
- $log_p(N) = \Theta(log_q(N))$

The last one essentially states that for logarithmic runtimes, the base of the logarithm doesn't matter at all, because they are all equivalent in terms of Big-O (this can be seen by applying the logarithm change of base). Applying these simplifications, we see that $\Theta(\lfloor log_2(N) \rfloor = \Theta(log(N))$ just as we expected from our intuition.

Example Proof: Prove $\lfloor f(N) \rfloor = \Theta(f(N))$

Solution:

We start with the following inequality:

$$f(N) - \frac{1}{2} < f(N) \leq \lfloor f(N) + \frac{1}{2} \rfloor \leq f(N) + \frac{1}{2}$$

Simplifying $f(N) - \frac{1}{2}$ and $f(N) + \frac{1}{2}$ according to our big theta rules by dropping the constants, we see that they are of order $f(N)$. Therefore $\lfloor f(N) + \frac{1}{2} \rfloor$ is bounded by two expressions of order $f(N)$, and is therefore also $\Theta(f(N))$.

Exercise:

- Prove $\lceil f(N) \rceil = \Theta(f(N))$
- Prove $log_p(N) = \Theta(log_q(N))$

One cool fact to wrap up with: Log time is super good! It's almost as fast as constant time, and way better than linear time. This is why we like binary search, rather than stepping one by one through our list and looking for the right thing.

To show this concretely:

N	$log_2(N)$	Typical runtime (nanoseconds)
100	6.6	1
100,000	16.6	2.5
100,000,000	26.5	4
100,000,000,000	36.5	5.5
100,000,000,000,000	46.5	7

The input size increase by a factor of 1 trillion but the runtime only inceased by a factor of about 7! This shows how remarkable $log(N)$ runtime is especially for large size inputs.

Previous
15.2 Recursion

Next
15.4 Mergesort

Last updated 1 year ago