**Algorithmic Big O(log N)**

We commonly see O(log N) in runtimes. Where does this come from?

Let's look at **binary search** as an example.

In binary search, we are looking for an example x in an N-element sorted array.

If x == middle, then we return.

If x < middle, then we search on the left side of the array.

If x > middle, then we search on the right side of the array.

We start off with an N-element array to search.

Then, after a single step, we're down to N/2 elements.

One more step, and we're down to N/4 elements.

Another step, and we’re down to N/8 elements.

… and so on until we either find the value or we're down to just one element.

The total runtime is then a matter of how many steps (dividing N by 2 each time) we can take until N becomes 1. In this case, N is the size of a sorted array.

N = 16

N = 8 /\* divide by 2 \*/

N = 4 /\* divide by 2 \*/

N = 2 /\* divide by 2 \*/

N = 1 /\* divide by 2 \*/

We could look at this in reverse (going from 1 to 16 instead of 16 to 1).

How many times we can multiply 1 by 2 until we get N?

N = 1

N = 2 /\* multiply by 2 \*/

N = 4 /\* multiply by 2 \*/

N = 8 /\* multiply by 2 \*/

N = 16 /\* multiply by 2 \*/

From this, we can conclude that 2k = N.

What is k in the expression 2k = N? This is exactly what log expresses.

**Formula**: log2N = k 🡪 2k = N

**Problem**: log216 = 4 🡪 24 = 16

This is the essential property of O(log N).

**When you see a problem where the number of elements in the problem space gets halved on each pass of the algorithm, then the runtime is most likely O(log N).**

When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a O(log N) runtime.

**Common Mistake**

* Note this common mistake when determining the runtime of an algorithm:
* The following code reverses an array. What is its runtime?

void reverse(int[] array) {

for (int i= 0; i < array.length / 2; i++) {

O(1)

}

}

* You may think this is O(log N) because you see the array being split in half. **This is wrong.**
* This algorithm actually runs in O(N) time. To be exact, the runtime is O(N/2) because we only iterate over half of the array.
* Because we only iterate over half of the array, the algorithm does not satisfy the fact that we must half the input array size over *every* pass of the loop, which this function does not do.
* The fact that it only goes through half of the array (in terms of iterations) does not impact the big O time, so the runtime is O(N).

**Binary Search Tree and O(log N)**

* This is the same reason why inserting or finding an element in a balanced binary search tree is O(log N).
* Each time we are at a node, we compare its left and right children. Whichever is smaller/larger, that the branch we travel down.
* Half the nodes are on each side, so we cut the problem space in half each time we traverse the tree.