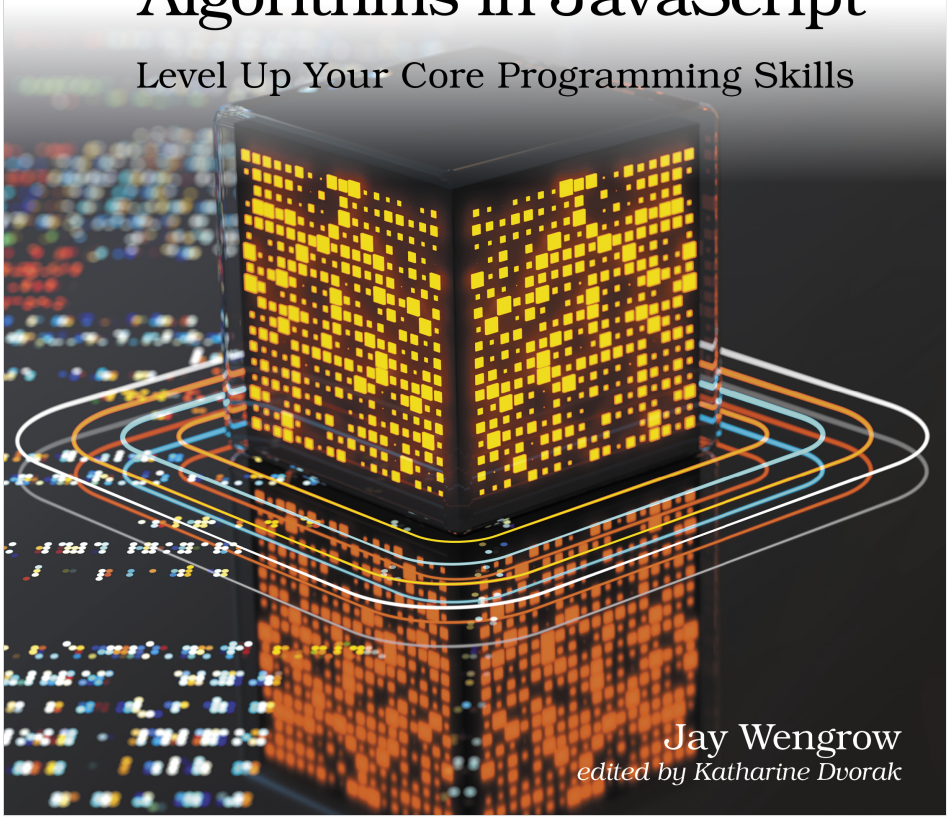
The Pragmatic Programmers

Volume 1

A Common-Sense Guide to

Data Structures and Algorithms in JavaScript

Level Up Your Core Programming Skills



Jay Wengrow

edited by Katharine Dvorak

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Big O: How Many Steps Relative to N Elements?

Big O achieves consistency by focusing on the number of steps an algorithm takes, but in a specific way. Let's start off by applying Big O to the algorithm of linear search.

In a worst-case scenario, linear search will take as many steps as there are elements in the array. As we've previously phrased it: for N elements in the array, linear search can take up to N steps. The appropriate way to express this in Big O notation is:

O(N)

Some pronounce this as "Big Oh of N." Others call it "Order of N." My personal preference, however, is "Oh of N."

Here's what the notation means. It expresses the answer to what we'll call the *key question*. The key question is this: *if there are N data elements, how many steps will the algorithm take?* Go ahead and read that sentence again. Then, emblazon it on your forehead, as this is the definition of Big O notation that we'll be using throughout the rest of this book.

The answer to the key question lies within the *parentheses* of our Big O expression. O(N) says that the answer to the key question is that *the algorithm will take N steps*.

Let's quickly review the thought process for expressing time complexity with Big O notation, again using the example of linear search. First we ask the key question: if there are N data elements in an array, how many steps will linear search take? Because the answer to this question is that linear search will take N steps, we express this as O(N). For the record, an algorithm that is O(N) is also known as having *linear time*.

Let's contrast this with how Big O would express the efficiency of *reading* from a standard array. As you learned in Chapter 1, Why Data Structures Matter, on page ?, reading from an array takes just one step, no matter how large the array is. To figure out how to express this in Big O terms, we're going to again ask the key question: if there are N data elements, how many steps will reading from an array take? The answer is that reading takes just one step. So we express this as O(1), which I pronounce "Oh of 1."

O(1) is interesting since although our key question revolves around N ("If there are N data elements, how many steps will the algorithm take?"), the answer has nothing to do with N. And that's actually the whole point: *no*

matter how many elements an array has, reading from the array *always* takes one step.

And this is why O(1) is considered the "fastest" kind of algorithm. Even as the data increases, an O(1) algorithm doesn't take any additional steps. The algorithm always takes a constant number of steps no matter what N is. In fact, an O(1) algorithm can also be referred to as having *constant time*.

So, Where's the Math?

As I mentioned earlier in this book, I'm taking an easy-to-understand approach to the topic of Big O. That's not the only way to do it; if you were to take a traditional college course on algorithms, you'd probably be introduced to Big O from a mathematical perspective. Big O is originally a concept from mathematics, and therefore, it's often described in mathematical terms. For example, one way of describing Big O is that it describes the upper bound of the growth rate of a function, or that if a function g(x) grows no faster than a function f(x), then g is said to be a member of O(f). Depending on your mathematics background, that either makes sense or doesn't help very much. I've written this book so that you don't need as much math to understand the concept. If you want to dig further into the math behind Big O, check out *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (MIT Press, 2009) for a full mathematical explanation. Justin Abrahms also provides a pretty good definition in his article: https://justin.abrah.ms/computer-science/understanding-big-o-formal-definition.html.

The Soul of Big O

Now that we've encountered O(N) and O(1), we begin to see that Big O notation does more than simply describe the number of steps an algorithm takes, such as with a hard number like 22 or 400. Rather, it's an answer to that key question on your forehead: if there are N data elements, how many steps will the algorithm take?

While that key question is indeed the strict definition of Big O, there's actually more to Big O than meets the eye.

Let's say we have an algorithm that always takes three steps no matter how much data there is. That is, for N elements, the algorithm always takes three steps. How would you express that in terms of Big O?

Based on everything you've learned up to this point, you'd probably say that it's O(3).

However, it's actually O(1). And that's because of the next layer of understanding Big O, which I will reveal now.

While Big O is an expression of the number of an algorithm's steps relative to N data elements, that alone misses the deeper *why* behind Big O, what I dub the "soul of Big O."

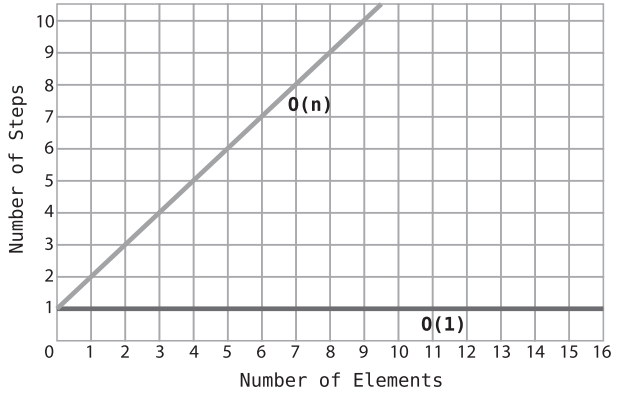
The soul of Big O is what Big O is truly concerned about: how will an algorithm's performance *change as the data increases*?

This is the soul of Big O. Big O doesn't want to simply tell you how many steps an algorithm takes. It wants to tell you the story of how the number of steps increases as the data *changes*.

Viewed with this lens, we don't care very much whether an algorithm is O(1) or O(3). Because both algorithms are the type that aren't affected by increased data, as their number of steps remains constant, they're essentially the same kind of algorithm. They're both algorithms whose steps remain constant irrespective of the data, and we don't care to make a distinction between the two.

An algorithm that is O(N), on the other hand, is a different type of algorithm. It's an algorithm whose performance *is* affected as we increase the data. More specifically, it's the kind of algorithm whose steps increase in direct proportion to the data as the data increases. This is the story O(N) tells. It tells you about the proportional relationship between the data and the algorithm's efficiency. It describes exactly how the number of steps increases as the data increases.

Look at how these two types of algorithms are plotted on a graph:



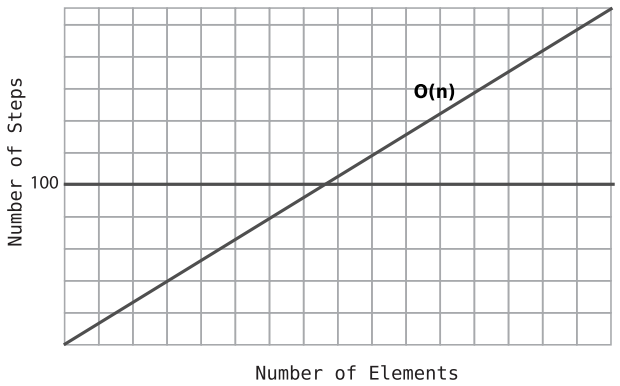
Notice that O(N) makes a perfect diagonal line. This is because for every additional piece of data, the algorithm takes one additional step. Accordingly, the more data, the more steps the algorithm will take.

Contrast this with O(1), which is a perfect horizontal line. No matter how much data there is, the number of steps remains constant.

Deeper into the Soul of Big O

To see why the soul of Big O is so important, let's go one level deeper. Say we had an algorithm of constant time that always took 100 steps no matter how much data there was. Would you consider that to be more or less performant than an algorithm that is O(N)?

Take a look at the following graph:



As the graph depicts, for a data set that is fewer than 100 elements, an O(N) algorithm takes fewer steps than the O(1) 100-step algorithm. At exactly 100 elements, the lines cross, meaning the two algorithms take the same number of steps, namely 100. But here's the key point: for *all arrays greater than 100*, the O(N) algorithm takes more steps.

Because there will always be *some* amount of data at which the tides turn, and O(N) takes more steps from that point until infinity, O(N) is considered to be, on the whole, less efficient than O(1) no matter how many steps the O(1) algorithm actually takes.

The same is true even for an O(1) algorithm that always takes one million steps. As the data increases, there will inevitably reach a point at which O(N) becomes less efficient than the O(1) algorithm and will remain so up toward an infinite amount of data.

Same Algorithm, Different Scenarios

As you learned in the previous chapters, linear search isn't *always* O(N). It's true that if the item we're looking for is in the final cell of the array, it will take N steps to find it. But when the item we're searching for is found in the *first* cell of the array, linear search will find the item in just one step. So this case of linear search would be described as O(1). If we were to describe the efficiency of linear search in its totality, we'd say that linear search is O(1) in a *best-case* scenario and O(N) in a *worst-case* scenario.

While Big O effectively describes both the best- and worst-case scenarios of a given algorithm, Big O notation generally refers to the *worst-case scenario* unless specified otherwise. This is why most references will describe linear search as being O(N) even though it *can* be O(1) in a best-case scenario.

This is because a "pessimistic" approach can be a useful tool: knowing exactly how inefficient an algorithm can get in a worst-case scenario prepares us for the worst and may have a strong impact on our choices.