Extracted from:

A Common-Sense Guide to Data Structures and Algorithms

Level Up Your Core Programming Skills

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

A Common-Sense Guide to Data Structures and Algorithms

Level Up Your Core Programming Skills

Jay Wengrow

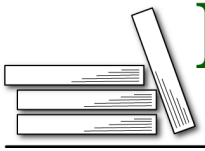
edited by Brian MacDonald

A Common-Sense Guide to Data Structures and Algorithms

Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Brian MacDonald
Copy Editor: Nicole Abramowtiz
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-244-2
Book version: P2.0—July 2018

We've seen in the preceding chapters that the number of steps that an algorithm takes is the primary factor in determining its efficiency.

However, we can't simply label one algorithm a "22-step algorithm" and another a "400-step algorithm." This is because the number of steps that an algorithm takes cannot be pinned down to a single number. Let's take linear search, for example. The number of steps that linear search takes varies, as it takes as many steps as there are cells in the array. If the array contains twenty-two elements, linear search takes twenty-two steps. If the array has 400 elements, however, linear search takes 400 steps.

The more accurate way to quantify efficiency of linear search is to say that linear search takes *N steps* for *N elements in the array*. Of course, that's a pretty wordy way of expressing this concept.

In order to help ease communication regarding time complexity, computer scientists have borrowed a concept from the world of mathematics to describe a concise and consistent language around the efficiency of data structures and algorithms. Known as *Big O Notation*, this formalized expression around these concepts allows us to easily categorize the efficiency of a given algorithm and convey it to others.

Once you understand Big O Notation, you'll have the tools to analyze every algorithm going forward in a consistent and concise way—and it's the way that the pros use.

While Big O Notation comes from the math world, we're going to leave out all the mathematical jargon and explain it as it relates to computer science. Additionally, we're going to begin by explaining Big O Notation in very simple terms, and continue to refine it as we proceed through this and the next three chapters. It's not a difficult concept, but we'll make it even easier by explaining it in chunks over multiple chapters.

Big O: Count the Steps

Instead of focusing on units of time, Big O achieves consistency by focusing only on the *number of steps* that an algorithm takes.

In *Why Data Structures Matter*, we discovered that reading from an array takes just one step, no matter how large the array is. The way to express this in Big O Notation is:

O(1)

Many pronounce this verbally as "Big Oh of 1." Others call it "Order of 1." My personal preference is "Oh of 1." While there is no standardized way to *pronounce* Big O Notation, there is only one way to *write* it.

O(1) simply means that the algorithm takes the same number of steps no matter how much data there is. In this case, reading from an array always takes just one step no matter how much data the array contains. On an old computer, that step may have taken twenty minutes, and on today's hardware it may take just a nanosecond. But in both cases, the algorithm takes just a single step.

Other operations that fall under the category of O(1) are the insertion and deletion of a value at the end of an array. As we've seen, each of these operations takes just one step for arrays of any size, so we'd describe their efficiency as O(1).

Let's examine how Big O Notation would describe the efficiency of linear search. Recall that linear search is the process of searching an array for a particular value by checking each cell, one at a time. 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 a maximum of N steps.

The appropriate way to express this in Big O Notation is:

O(N)

I pronounce this as "Oh of N."

O(N) is the "Big O" way of saying that for N elements inside an array, the algorithm would take N steps to complete. It's that simple.

Constant Time vs. Linear Time

Now that we've encountered O(N), we can begin to see that Big O Notation does more than simply describe the number of steps that an algorithm takes, such as a hard number such as 22 or 400. Rather, it describes how many steps an algorithm takes *based on the number of data elements that the algorithm is acting upon*. Another way of saying this is that Big O answers the following question: *how does the number of steps change as the data increases?*

An algorithm that is O(N) will take as many steps as there are elements of data. So when an array increases in size by one element, an O(N) algorithm will increase by one step. An algorithm that is O(1) will take the same number of steps no matter how large the array gets.

So Where's the Math?

As I mentioned, 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.

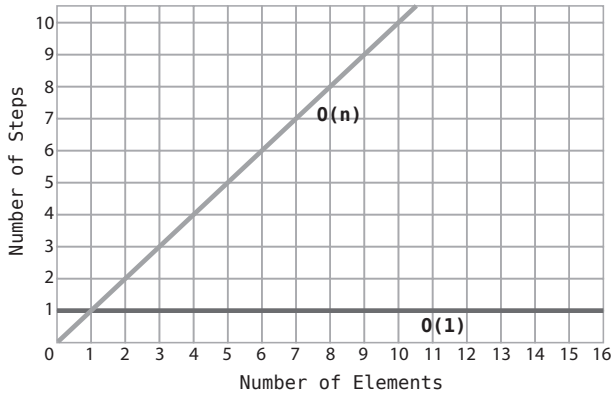
If you want to dig further into the math behind Big O, check out *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (MIT Press, 2009) for a full mathematical explanation. Justin Abrahms provides a pretty good definition in his article: https://justin.abrah.ms/computer-science/understanding-big-o-formal-definition.html. Also, the Wikipedia article on Big O (https://en.wikipedia.org/wiki/Big_O_notation) takes a fairly heavy mathematical approach.

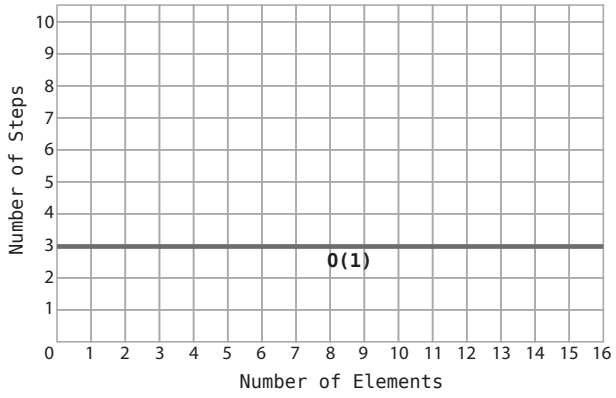
Look at how these two types of algorithms are plotted on a graph on page 8.

You'll see 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. For the record, O(N) is also known as *linear time*.

Contrast this with O(1), which is a perfect horizontal line, since the number of steps in the algorithm remains constant no matter how much data there is. Because of this, O(1) is also referred to as *constant time*.

As Big O is primarily concerned about how an algorithm performs across varying amounts of data, an important point emerges: an algorithm can be described as O(1) even if it takes more than one step. Let's say that a particular algorithm always takes *three* steps, rather than one—but it always takes these three steps no matter how much data there is. On a graph, such an algorithm would look like this:



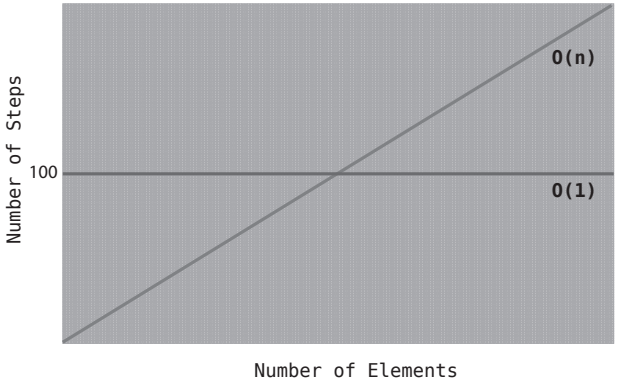


Because the number of steps remains constant no matter how much data there is, this would also be considered constant time and be described by Big O Notation as O(1). Even though the algorithm technically takes three steps rather than one step, Big O Notation considers that trivial. O(1) is the way to describe *any* algorithm that doesn't change its number of steps even when the data increases.

If a three-step algorithm is considered O(1) as long as it remains constant, it follows that even a constant 100-step algorithm would be expressed as O(1) as well. While a 100-step algorithm is less efficient than a one-step algorithm, the fact that it is O(1) still makes it *more efficient* than any O(N) algorithm.

Why is this?

See the following graph:



As the graph depicts, for an array of fewer than one hundred elements, O(N) algorithm takes fewer steps than the O(1) 100-step algorithm. At exactly one hundred elements, the two algorithms take the same number of steps (100). But here's the key point: for *all arrays greater than one hundred*, the O(N) algorithm takes more steps.

Because there will always be *some* amount of data in 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).

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

Same Algorithm, Different Scenarios

As we 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 where 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. Technically, this 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 *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.

The reason for this is that this “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.