All you need to know about “Big O Notation” to crack your next coding interview

Paul Rail Follow
Jun 12, 2018 · 5 min read

As part of my software development education(I enrolled in a program called Microverse), I needed to build up skills in various areas to become fully prepared for my first software position. And any software education program worth their salt will include a fair portion of the curriculum geared towards getting ready for the infamous coding interview.

So to this end, at the start of every single day, **I work on solving algorithms, as this is a major part (and to many *the most difficult part*) of most coding interviews.**

One thing that I’ve come across while working on computer science algorithms is something called the **“Big O Notation”**.

It is a pretty abstract and very esoteric concept that the vast majority of people will never hear about, or care about. BUT it is known as being a **common coding interview question**, and therefore it’s one of the things I have spent some time learning all about.

What you need to know

Here is what I have absorbed in order to prepare myself

To set the scene for “Big O,” we first need to acknowledge that **software is, of course, heavily based on data**. Huge mountains of data. And making use of that data is what coding is for. In order for a program to make use of data, it often needs to start by sorting that data into a logical order. Whether that is alphabetically, chronologically, by size, by date, and so on.

Sorting happens CONSTANTLY, and actually **represents a huge portion of all computer and internet activity.** I’ve heard programmers state that “Quick Sort is pretty much what runs all of the internet”.

What do they mean by that? Well sorting data is its own entire subsection within the study of computer science, and there are many well-defined algorithms for sorting. There is **Quick Sort, Bubble Sort, Selection Sort, Merge Sort, Heap Sort** and many more. Each with different approaches to get to the same or similar results.

Source: https://yourbasic.org/algorithms

But which one is best if they (almost) all return the same result?

Best usually means which is fastest. This is where “Big O” came into play.

Big O notation, sometimes also called “asymptotic analysis”, primarily **looks at how many operations a sorting algorithm takes to completely sort a very large collection of data.** This is a measure of efficiency and is how you can directly compare one algorithm to another.

When building a simple app with only a few pieces of data to work through, this sort of analysis is unnecessary. But when working with very large amounts of data, like a social media site or a large e-commerce site with many customers and products, **small differences between algorithms can be significant.**

Big O notation ranks an algorithms' efficiency

It does this with regard to "**O**" and "**n**", (example: *"O(log n)")*, where

- **O** refers to the order of the function, or its growth rate, and
- **n** is the length of the array to be sorted.

Let's work through an example. If an algorithm has the number of operations required formula of:

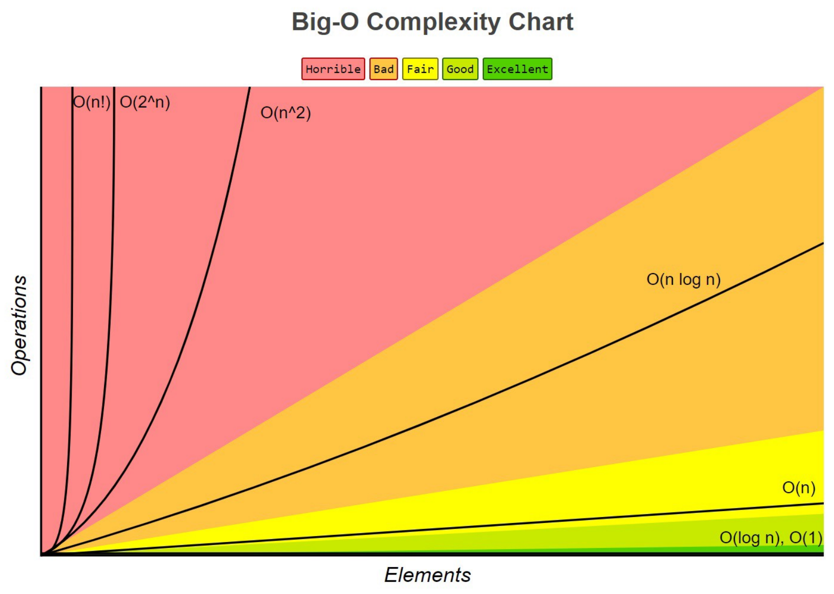
***f*(n) = 6n^4 - 2n^3 + 5**

As "**n**" approaches infinity (for very large sets of data), of the three terms present, **6n^4** is the only one that matters. So the lesser terms, **2n^3** and **5**, are actually just omitted because they are insignificant. Same goes for the "**6**" in **6n^4**, actually.

Therefore, this function would have an order growth rate, or a "big O" rating, of O(n^4) .

When looking at many of the most commonly used sorting algorithms, the rating of **O(n log n)** in general is the best that can be achieved. Algorithms that run at this rating include Quick Sort, Heap Sort, and

Merge Sort. **Quick Sort** is the standard and is used as the default in almost all software languages.



Source: http://bigocheatsheet.com/

It is important to note that **there is no single algorithm that is fastest in all cases**, as data can be input into a program in all manners of states. And the approaches of each algorithm will have a best case and worst case scenario where they perform at their best or worst.

While Quick Sort is the standard, it also competes with Merge Sort and Heap Sort, which are other O(n log n) rated sorting algorithms. There are scenarios where those are used instead.

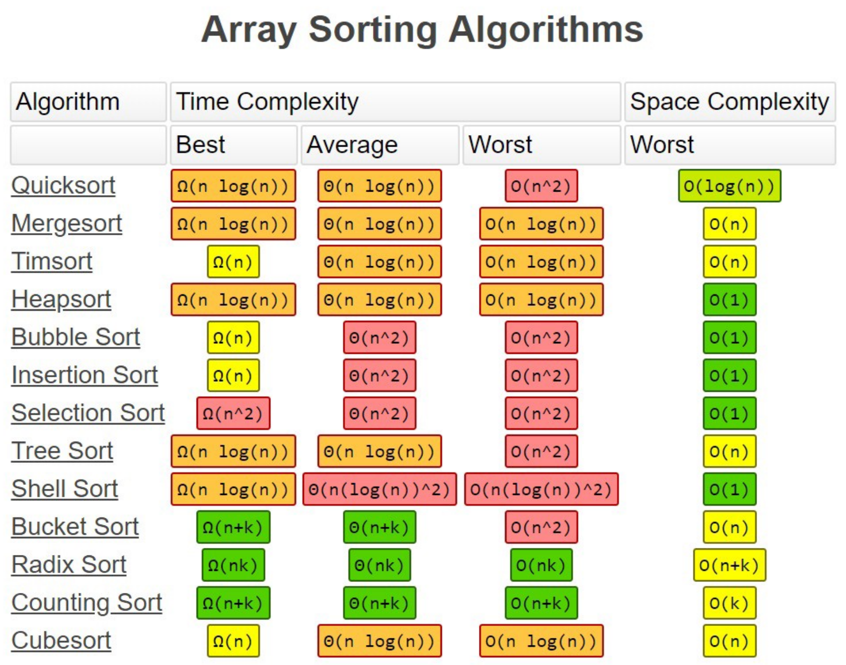
The most direct competitor of Quick Sort is **Heap Sort.** Heap Sort’s running time is also O(n log n), but Heap Sort’s average running time is usually considered slower than in-place Quick Sort.

Merge Sort is a **stable sort**, which means it preserves the input order of equal elements in the output, unlike standard in-place Quick Sort and Heap Sort.

Bubble / Insertion / Selection Sort run at O(n^2), which in terms of number of operations **can take significantly longer** than those listed above rated at O(n log n) when dealing with really big data. But there can be scenarios where the others are faster depending on the data.

There are also times where something that is very simple, such as Counting Sort, is great, because it is much faster to write up and much easier to visualize and comprehend.

Sometime you not only need to consider the time demands of an algorithm, but also the data-space demands as well (or perhaps even more so). Some algorithms also operate with a smaller storage footprint.



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Source: http://bigocheatsheet.com/

Why do you need to know all this?

So after all that, if you always just resort to using a language's built-in sorting algorithm, (which is based on Quick Sort), then why care about sorting algorithms and "Big O"? Why would companies ask you about it in an interview?

The answer is that studying Big O notation makes you grasp the very important concept of efficiency in your code. So when you do work with huge data sets, you will have a good sense of where major slowdowns are likely to cause bottlenecks, and where more attention should be paid to get the largest improvements. This is also called sensitivity analysis, and is an important part of solving problems and writing great software.

So if you are trying to prepare for your first interview, or perhaps you struggled in your last one, increasing your knowledge on concepts like Big O Notation and other computer science topics will help give you a leg up. You’ll be better equipped to demonstrate your potential and impress to land that position.