



Intro to Algorithms: Crash Course Computer Science #13

Crash Course: Computer Science

<https://youtube.com/watch?v=rL8X2mINHPM>

<https://nerdfighteria.info/v/rL8X2mINHPM>

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

Over the past two episodes, we got our first taste of programming in a high-level language, like Python or Java. We talked about different types of programming language statements – like assignments, ifs, and loops – as well as putting statements into functions that perform a computation, like calculating an exponent.

Importantly, the function we wrote to calculate exponents is only one possible solution. There are other ways to write this function – using different statements in different orders – that achieve exactly the same numerical result. The difference between them is the algorithm, that is the specific steps used to complete the computation.

Some algorithms are better than others even if they produce equal results. Generally, the fewer steps it takes to compute, the better it is, though sometimes we care about other factors, like how much memory it uses. The term algorithm comes from Persian polymath Muḥammad ibn Mūsā al-Khwarīzmi who was one of the fathers of algebra more than a millennium ago.

The crafting of efficient algorithms – a problem that existed long before modern computers – led to a whole science surrounding computation, which evolved into the modern discipline of... you guessed it! Computer Science!

INTRO

One of the most storied algorithmic problems in all of computer science is sorting... as in sorting names or sorting numbers.

Computers sort all the time. Looking for the cheapest airfare, arranging your email by most recently sent, or scrolling your contacts by last name -- those all require sorting. You might think "sorting isn't so tough... how many algorithms can there possibly be?" The answer is: a lot.

Computer Scientists have spent decades inventing algorithms for sorting, with cool names like Bubble Sort and Spaghetti Sort. Let's try sorting! Imagine we have a set of airfare prices to Indianapolis.

We'll talk about how data like this is represented in memory next week, but for now, a series of items like this is called an array. Let's take a look at these numbers to help see how we might sort this programmatically. We'll start with a simple algorithm.

First, let's scan down the array to find the smallest number. Starting at the top with 307. It's the only number we've seen, so it's also the smallest.

The next is 239, that's smaller than 307, so it becomes our new smallest number. Next is 214, our new smallest number. 250 is not, neither is 384, 299, 223 or 312. So we've finished scanning all numbers, and 214 is the smallest.

To put this into ascending order, we swap 214 with the number in the top location. Great. We sorted one number!

Now we repeat the same procedure, but instead of starting at the top, we can start one spot below. First we see 239, which we save as our new smallest number. Scanning the rest of the array, we find 223 is the next smallest, so we swap this with the number in the second spot.

Now we repeat again, starting from the third number down. This time, we swap 239 with 307. This process continues until we get to the very last number, and voila, the array is sorted and you're

ready to book that flight to Indianapolis!

The process we just walked through is one way – or one algorithm – for sorting an array. It's called Selection Sort -- and it's pretty basic. Here's the pseudo-code.

This function can be used to sort 8, 80, or 80 million numbers - and once you've written the function, you can use it over and over again. With this sort algorithm, we loop through each position in the array, from top to bottom, and then for each of those positions, we have to loop through the array to find the smallest number to swap. You can see this in the code, where one FOR loop is nested inside of another FOR loop.

This means, very roughly, that if we want to sort N items, we have to loop N times, inside of which, we loop N times, for a grand total of roughly N times N loops... Or N squared. This relationship of input size to the number of steps the algorithm takes to run characterizes the complexity of the Selection Sort algorithm.

It gives you an approximation of how fast, or slow, an algorithm is going to be. Computer Scientists write this order of growth in something known as – no joke – "big O notation". N squared is not particularly efficient.

Our example array had $n = 8$ items, and 8 squared is 64. If we increase the size of our array from 8 items to 80, the running time is now 80 squared, which is 6,400. So although our array only grew by 10 times - from 8 to 80 – the running time increased by 100 times – from 64 to 6,400!

This effect magnifies as the array gets larger. That's a big problem for a company like Google, which has to sort arrays with millions or billions of entries. So, you might ask, as a burgeoning computer scientist, is there a more efficient sorting algorithm?

Let's go back to our old, unsorted array and try a different algorithm, merge sort. The first thing merge sort does is check if the size of the array is greater than 1. If it is, it splits the array into two halves.

Since our array is size 8, it gets split into two arrays of size 4. These are still bigger than size 1, so they get split again, into arrays of size 2, and finally they split into 8 arrays with 1 item in each. Now we are ready to merge, which is how "merge sort" gets its name.

Starting with the first two arrays, we read the first – and only – value in them, in this case, 307 and 239. 239 is smaller, so we take that value first. The only number left is 307, so we put that value second. We've successfully merged two arrays.

We now repeat this process for the remaining pairs, putting them each in sorted order. Then the merge process repeats. Again, we take the first two arrays, and we compare the first numbers in them.

This time it's 239 and 214. 214 is lowest, so we take that number first. Now we look again at the first two numbers in both arrays: 239 and 250. 239 is lower, so we take that number next. Now we look at the next two numbers: 307 and 250. 250 is lower, so we take that.

Finally, we're left with just 307, so that gets added last. In every case, we start with two arrays, each individually sorted, and merge them into a larger sorted array. We repeat the exact same merging process for the two remaining arrays of size two.

Now we have two sorted arrays of size 4. Just as before, we merge, comparing the first two numbers in each array, and taking the lowest. We repeat this until all the numbers are merged, and then our array is fully sorted again!



Intro to Algorithms: Crash Course Computer Science #13

Crash Course: Computer Science

<https://youtube.com/watch?v=rL8X2mINHPM>

<https://nerdfighteria.info/v/rL8X2mINHPM>

The bad news is: no matter how many times we sort these, you're still going to have to pay \$214 to get to Indianapolis. Anyway, the "Big O" computational complexity of merge sort is $N \times \log N$. The N comes from the number of times we need to compare and merge items, which is directly proportional to the number of items in the array.

The $\log N$ comes from the number of merge steps. In our example, we broke our array of 8 items into 4, then 2, and finally 1. That's 3 splits.

Splitting in half repeatedly like this has a logarithmic relationship with the number of items - trust me! $\log_2 8$ equals 3 splits. If we double the size of our array to 16 - that's twice as many items to sort - it only increases the number of split steps by 1 since $\log_2 16$ equals 4.

Even if we increase the size of the array more than a thousand times, from 8 items to 8000 items, the number of split steps stays pretty low. $\log_2 8000$ is roughly 13. That's more, but not much more than 3 -- about four times larger -- and yet we're sorting a lot more numbers.

For this reason, merge sort is much more efficient than selection sort. And now I can put my ceramic cat collection in name order MUCH faster! There are literally dozens of sorting algorithms we could review, but instead, I want to move on to my other favorite category of classic algorithmic problems: graph search!

A graph is a network of nodes connected by lines. You can think of it like a map, with cities and roads connecting them. Routes between these cities take different amounts of time.

We can label each line with what is called a cost or weight. In this case, it's weeks of travel. Now let's say we want to find the fastest route for an army at Highgarden to reach the castle at Winterfell.

The simplest approach would just be to try every single path exhaustively and calculate the total cost of each. That's a brute force approach. We could have used a brute force approach in sorting, by systematically trying every permutation of the array to check if it's sorted.

This would have an N factorial complexity - that is the number of nodes, times one less, times one less than that, and so on until 1. Which is way worse than even N^2 . But, we can be way more clever!

The classic algorithmic solution to this graph problem was invented by one of the greatest minds in computer science practice and theory, Edsger Dijkstra, so it's appropriately named Dijkstra's algorithm. We start in Highgarden with a cost of 0, which we mark inside the node. For now, we mark all other cities with question marks - we don't know the cost of getting to them yet.

Dijkstra's algorithm always starts with the node with lowest cost. In this case, it only knows about one node, Highgarden, so it starts there. It follows all paths from that node to all connecting nodes that are one step away, and records the cost to get to each of them.

That completes one round of the algorithm. We haven't encountered Winterfell yet, so we loop and run Dijkstra's algorithm again. With Highgarden already checked, the next lowest cost node is King's Landing.

Just as before, we follow every unvisited line to any connecting cities. The line to The Trident has a cost of 5. However, we want to keep a running cost from Highgarden, so the total cost of getting to The Trident is 8 plus 5, which is 13 weeks.

Now we follow the off-road path to Riverrun, which has a high cost of 25, for a total of 33. But we can see inside of Riverrun that we've already found a path with a lower cost of just 10. So we disregard our new path, and stick with the previous, better path.

We've now explored every line from King's Landing and didn't find Winterfell, so we move on. The next lowest cost node is Riverrun, at 10 weeks. First we check the path to The Trident, which has a total cost of 10 plus 2, or 12.

That's slightly better than the previous path we found, which had a cost of 13, so we update the path and cost to The Trident. There is also a line from Riverrun to Pyke with a cost of 3. 10 plus 3 is 13, which beats the previous cost of 14, and so we update Pyke's path and cost as well. That's all paths from Riverrun checked... so... you guessed it, Dijkstra's algorithm loops again.

The node with the next lowest cost is The Trident and the only line from The Trident that we haven't checked is a path to Winterfell! It has a cost of 10, plus we need to add in the cost of 12 it takes to get to The Trident, for a grand total cost of 22. We check our last path, from Pyke to Winterfell, which sums to 31.

Now we know the lowest total cost, and also the fastest route for the army to get there, which avoids King's Landing! Dijkstra's original algorithm, conceived in 1956, had a complexity of the number of nodes in the graph squared. And squared, as we already discussed, is never great, because it means the algorithm can't scale to big problems - like the entire road map of the United States.

Fortunately, Dijkstra's algorithm was improved a few years later to take the number of nodes in the graph, times the log of the number of nodes, PLUS the number of lines. Although this looks more complicated, it's actually quite a bit faster. Plugging in our example graph, with 6 cities and 9 lines, proves it.

Our algorithm drops from 36 loops to around 14. As with sorting, there are innumerable graph search algorithms, with different pros and cons. Every time you use a service like Google Maps to find directions, an algorithm much like Dijkstra's is running on servers to figure out the best route for you.

Algorithms are everywhere and the modern world would not be possible without them. We touched only the very tip of the algorithmic iceberg in this episode, but a central part of being a computer scientist is leveraging existing algorithms and writing new ones when needed, and I hope this little taste has intrigued you to SEARCH further. I'll see you next week.