Bubble Sort

The original uploader was Nmnogueira at English Wikipedia., CC BY-SA 2.5 https://creativecommons.org/licenses/by-sa/2.5, via Wikimedia Commons

Let's do our first sort! The bubble sort is a typical first one to do because it matches the human mental model of sorting pretty well. The algorithm is pretty simple: compare two items in an array that are next to each other. If they're out of order (that is, the larger one comes first in the array) swap them. Then move forward one index, compare again, swap if needed, and continue to the next item in the array. Once we've reached the end of the array, if we've swapped anything in the previous run through, the array *could* still be out of order, so we have to pass through again. Once we run through the whole array with no swaps, the array is sorted!

Let's do a short example.

```
[1, 5, 4, 2, 3]

Are 1 and 5 out of order? No.
Are 5 and 4 out of order? Yes. Swap.

[1, 4, 5, 2, 3]

Are 5 and 2 out of order? Yes. Swap.

[1, 4, 2, 5, 3]

Are 5 and 3 out of order? Yes. Swap.

[1, 4, 2, 3, 5]

End of the array, did we swap anything? Yes. Loop again.
Are 1 and 4 out of order? No.
Are 4 and 2 out of order? Yes. Swap.

[1, 2, 4, 3, 5]

Are 4 and 3 out of order? Yes. Swap.

[1, 2, 3, 4, 5]

Are 4 and 5 out of order? No.
End of the array, did we swap anything? Yes. Loop again.
Are 1 and 2 out of order? No.
Are 2 and 3 out of order? No.
Are 3 and 4 out of order? No.
Are 4 and 5 out of order? No.
End of the array, did we swap anything? No. List is sorted.
```

That's it!

So, one more fun trick. Eventually, on your first runthrough, you will find the biggest number and will continue swapping with it until it reaches the last element in the array (like we did with 5 in our example.) This is why it's called bubble sort: the biggest elements bubble to the last spots. Thinking about this for a second, that means at the end of the iteration we're guaranteed to have the biggest element at the last spot. At the end of our first iteration above, 5 is the last element in the array. That means the last element in the array is definitely at its correct spot. That part of the array is sorted. So, on our second run through of the array, we technically don't need to ask `Are 4 and 5 out of order? No.` because know ahead of time that 5 is going to be bigger no matter what. So we can check one less element each iteration because we know at the end of each iteration we have one more guaranteed element to be in its correct. At the end of the second iteration we'll have 4 in its correct place, then 3, then 2, etc.

It's a little optimization (in our Big O it'd affect the coefficient so it'd wouldn't reflected in the Big O.) but it does make it a bit faster. It makes the sorting go faster as algorithm progresses.

Big O

So what is the computational complexity of this algorithm? Let's talk worst case, best case, and average case.

What is the absolute best, ideal case for this algorithm? A sorted list. It'd run through the list once, comparing each element to its neighbor once, not swap anything, and be done. In our list of five that would be four comparisons and no swaps. Best case here is O(n) since the best case is just one run through so it'd grow linearly.

What is the absolute worst case scenario? It'd be a backwards sorted list. Then it'd have to run all comparisons and always swap. Assuming we did the optimized version above, that'd be 4 comparisons and swaps on the first go-around, then 3, then 2, then 1, totalling 10 comparisons and swaps. This will grow exponentially as the list gets bigger. This would be $O(n^2)$.

Okay, so average case. You can think of average case as "what happens if we through a randomly sorted list at this". For our bubble sort, it'd be like the one we did above: some numbers in order, some not. What happens if we through 10 at it versus 1000. The amount of comparisons and swaps would grow exponentially. As such, we'd say it's $O(n^2)$ as well.

What about the spatial complexity? In our case, we're operating on the array itself and not creating anything else in memory, so the spatial complexity of this is O(1) since it'd never grow. Because we are operating on the array itself, we'd say this sort is **destructive**. What if you needed to keep the original array in addition to the sorted one? You couldn't use this sorting method or you'd have to pre-emptively make a copy. Other sorting algorithms are non-destructive.

Is this sorting algorithm **stable**? To be considered a stable sort, the sort must guarantee that if two things are *equal* that that they stay in that same order. For example, if we have an array of users that looks like this:
`[{state: "CO", name: "Sarah Drasner"}, {state: "CA", name: "Shirley Wu"}, {state: "CA`
and we're sorting by state, we'd have to guarantee that Shirley comes before Scott for the sort to be considered stable. Sometimes this is important to you, sometimes you don't care. So is bubble sort stable? Yes, it'll guarantee that equivalent items come back in the order they were in.