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

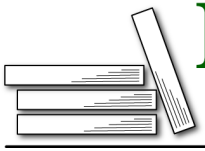
edited by Brian MacDonald

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

Bubble Sort

Before jumping into our practical problem, though, we need to first learn about a new category of algorithmic efficiency in the world of Big O. To demonstrate it, we'll get to use one of the classic algorithms of computer science lore.

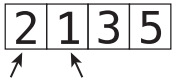
Sorting algorithms have been the subject of extensive research in computer science, and tens of such algorithms have been developed over the years. They all solve the following problem:

Given an array of unsorted numbers, how can we sort them so that they end up in ascending order?

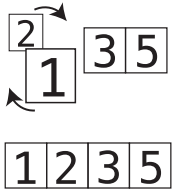
In this and the following chapters, we're going to encounter a number of these sorting algorithms. Some of the first ones we'll be learning about are known as "simple sorts," in that they are easier to understand, but are not as efficient as some of the faster sorting algorithms out there.

Bubble Sort is a very basic sorting algorithm, and follows these steps:

1. Point to two consecutive items in the array. (Initially, we start at the very beginning of the array and point to its first two items.) Compare the first item with the second one:

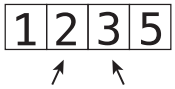


2. If the two items are out of order (in other words, the left value is greater than the right value), swap them:



   (If they already happen to be in the correct order, do nothing for this step.)

3. Move the "pointers" one cell to the right:



   Repeat steps 1 and 2 until we reach the end of the array or any items that have already been sorted.

4. Repeat steps 1 through 3 until we have a round in which we didn't have to make any swaps. This means that the array is in order.

   Each time we repeat steps 1 through 3 is known as a *passthrough*. That is, we "passed through" the primary steps of the algorithm, and will repeat the same process until the array is fully sorted.

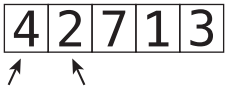
Bubble Sort in Action

Let's walk through a complete example. Assume that we wanted to sort the array [4, 2, 7, 1, 3]. It's currently out of order, and we want to produce an array containing the same values in the correct, ascending order.

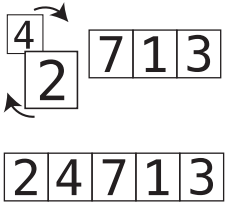
Let's begin Passthrough #1:

This is our starting array:

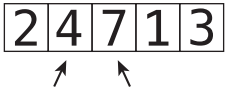
Step #1: First, we compare the 4 and the 2. They're out of order:



Step #2: So we swap them:



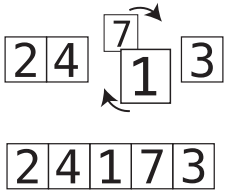
Step #3: Next, we compare the 4 and the 7:



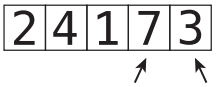
They're in the correct order, so we don't need to perform any swaps.

Step #4: We now compare the 7 and the 1:

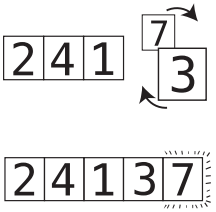
Step #5: They're out of order, so we swap them:



Step #6: We compare the 7 and the 3:



Step #7: They're out of order, so we swap them:



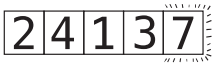
We now know for a fact that the 7 is in its correct position within the array, because we kept moving it along to the right until it reached its proper place. We've put little lines surrounding it to indicate this fact.

This is actually the reason that this algorithm is called *Bubble* Sort: in each passthrough, the highest unsorted value "bubbles" up to its correct position.

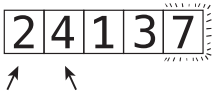
Since we made at least one swap during this passthrough, we need to conduct another one.

We begin Passthrough #2:

The 7 is already in the correct position:

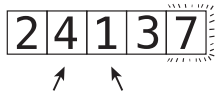


Step #8: We begin by comparing the 2 and the 4:

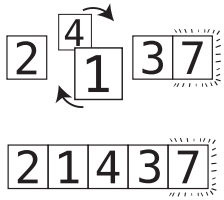


They're in the correct order, so we can move on.

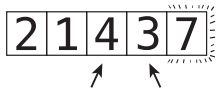
Step #9: We compare the 4 and the 1:



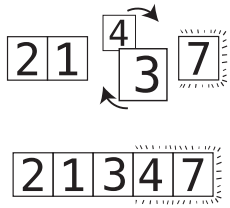
Step #10: They're out of order, so we swap them:



Step #11: We compare the 4 and the 3:



Step #12: They're out of order, so we swap them:



We don't have to compare the 4 and the 7 because we know that the 7 is already in its correct position from the previous passthrough. And now we also know that the 4 is bubbled up to its correct position as well. This concludes our second passthrough. Since we made at least one swap during this passthrough, we need to conduct another one.

We now begin Passthrough #3:

Step #13: We compare the 2 and the 1:

Step #14: They're out of order, so we swap them:

Step #15: We compare the 2 and the 3:

They're in the correct order, so we don't need to swap them.

We now know that the 3 has bubbled up to its correct spot. Since we made at least one swap during this passthrough, we need to perform another one.

And so begins Passthrough #4:

Step #16: We compare the 1 and the 2:

Since they're in order, we don't need to swap. We can end this passthrough, since all the remaining values are already correctly sorted.

Now that we've made a passthrough that didn't require any swaps, we know that our array is completely sorted: