Extracted from:

A Common-Sense Guide to Data Structures and Algorithms, Second Edition

Level Up Your Core Programming Skills

The Pragmatic Bookshelf

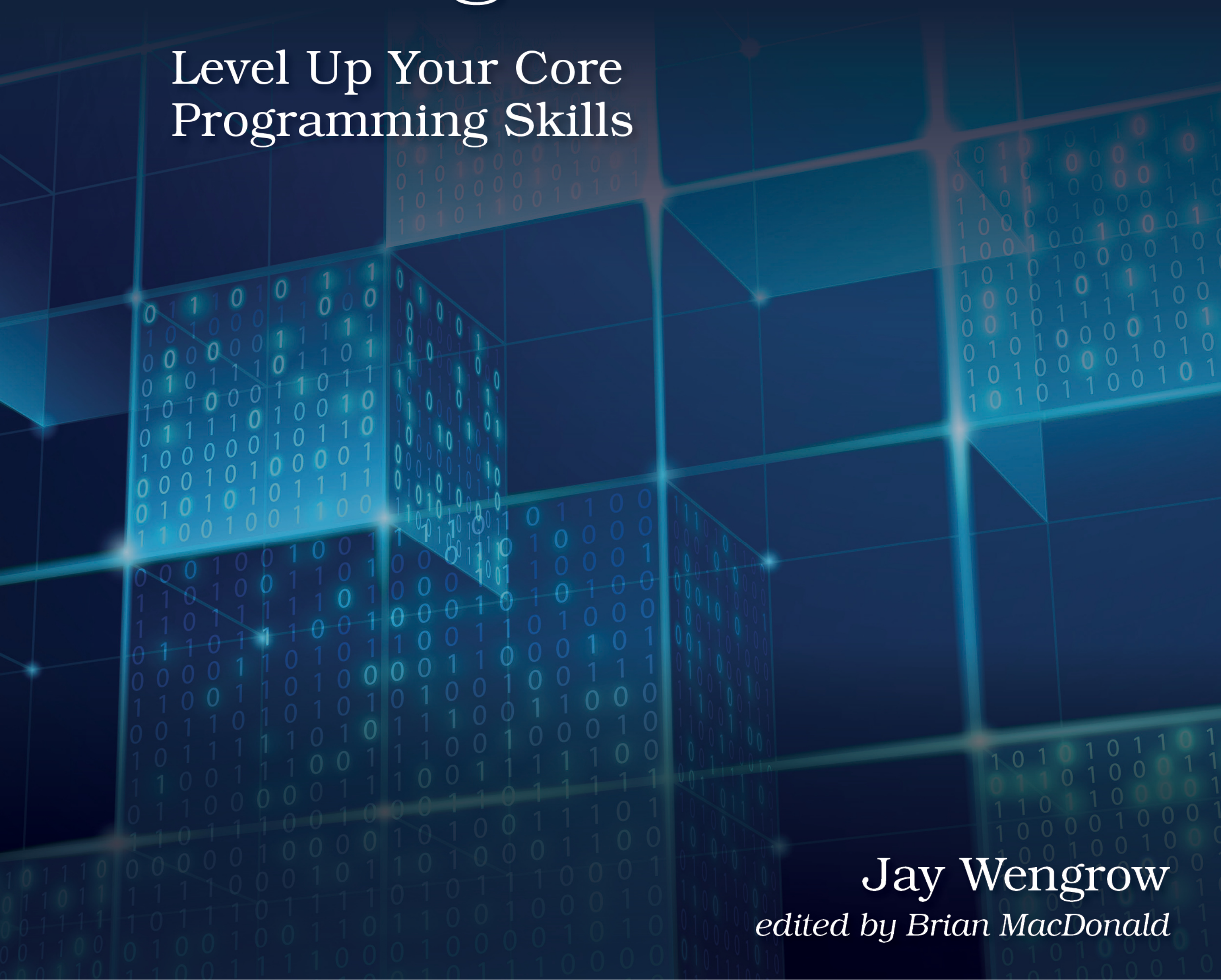
Raleigh, North Carolina

The Pragmatic Programmers

Second Edition

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, Second Edition

Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Brian MacDonald
Copy Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

ISBN-13: 978-1-68050-722-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2020

Bubble Sort

Before jumping into our practical problem, though, we need to first look at 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.

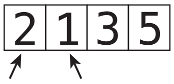
Sorting algorithms have been the subject of extensive research in computer science, and tens of such algorithms have been developed over the years. They all solve the following problem:

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

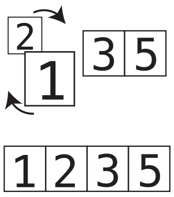
In this and the following chapters, we're going to encounter a number of these sorting algorithms. Some of the first ones you'll learn about are known as "simple sorts," in that they are easy to understand, but are not as efficient as some of the faster sorting algorithms out there.

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

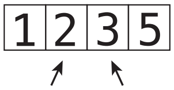
1. Point to two consecutive values in the array. (Initially, we start by pointing to the array's first two values.) Compare the first item with the second one:



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:



4. Repeat Steps 1 through 3 until we reach the end of the array, or if we reach the values that have already been sorted. (This will make more sense in the walk-through that follows.) At this point, we have completed our first *pass-through* of the array. That is, we "passed through" the array by pointing to each of its values until we reached the end.

5. We then move the two pointers back to the first two values of the array, and execute another pass-through of the array by running Steps 1 through 4 again. We keep on executing these pass-throughs until we have a pass-through in which we did not perform any swaps. When this happens, it means our array is fully sorted and our work is done.

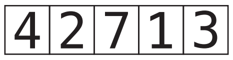
Bubble Sort in Action

Let's walk through a complete example of Bubble Sort.

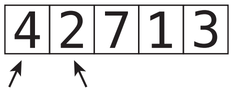
Assume we want to sort the array, [4, 2, 7, 1, 3]. It's currently out of order, and we want to produce an array that contains the same values in ascending order.

Let's begin the first pass-through:

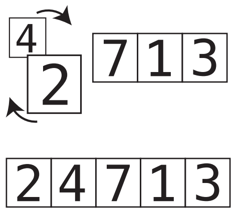
This is our starting array:



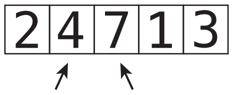
Step 1: First, we compare the 4 and the 2:



Step 2: They're out of order, so we swap them:

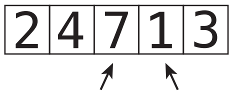


Step 3: Next, we compare the 4 and the 7:

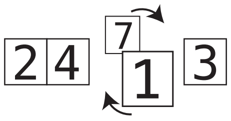


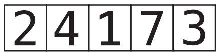
They're in the correct order, so we don't need to perform a swap.

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

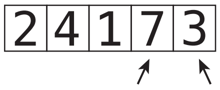


Step 5: They're out of order, so we swap them:

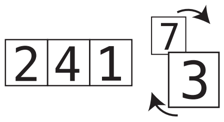


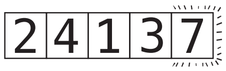


Step 6: We compare the 7 and the 3:



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





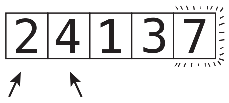
We now know for a fact that the 7 is in its correct position within the array, because we kept moving it along to the right until it reached its proper place. The previous diagram has little lines surrounding the 7 to indicate that the 7 is officially in its correct position.

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

Because we made at least one swap during this pass-through, we need to conduct another pass-through.

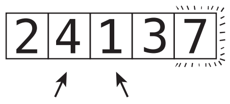
We begin the second pass-through:

Step 8: We compare the 2 and the 4:

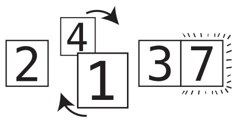


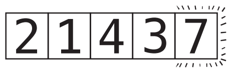
They're in the correct order, so we can move on.

Step 9: We compare the 4 and the 1:

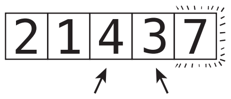


Step 10: They're out of order, so we swap them:

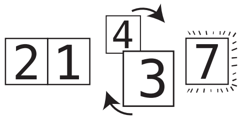


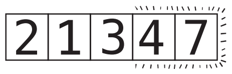


Step 11: We compare the 4 and the 3:



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



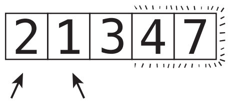


We don't have to compare the 4 and the 7 because we know that the 7 is already in its correct position from the previous pass-through. And now we also know that the 4 has bubbled up to its correct position as well. This concludes our second pass-through.

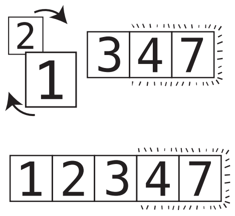
Because we made at least one swap during this pass-through, we need to conduct another pass-through.

We begin the third pass-through:

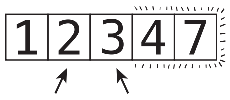
Step 13: We compare the 2 and the 1:



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

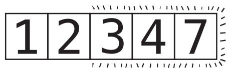


Step 15: We compare the 2 and the 3:



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

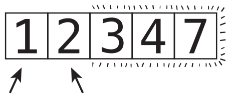
We now know that the 3 has bubbled up to its correct spot:



Since we made at least one swap during this pass-through, we need to perform another one.

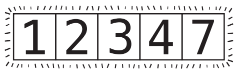
And so begins the fourth pass-through:

Step 16: We compare the 1 and the 2:



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

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



Code Implementation: Bubble Sort

Here's an implementation of Bubble Sort in Python:

```
def bubble_sort(list):
    unsorted_until_index = len(list) - 1
    sorted = False

    while not sorted:
        sorted = True
        for i in range(unsorted_until_index):
            if list[i] > list[i+1]:
                list[i], list[i+1] = list[i+1], list[i]
                sorted = False
        unsorted_until_index -= 1

    return list
```

To use this function, we can pass an unsorted array to it, like so:

```
print(bubble_sort([65, 55, 45, 35, 25, 15, 10]))
```

This function will then return the sorted array.

Let's break the function down line by line to see how it works. I'll explain each line by first providing the explanation, followed by the line of code itself.

The first thing we do is create a variable called unsorted_until_index. This keeps track of the rightmost index of the array that has *not* yet been sorted. When we first start the algorithm, the array is completely unsorted, so we initialize this variable to be the final index in the array:

```
unsorted_until_index = len(list) - 1
```

We also create a variable called sorted that will keep track of whether the array is fully sorted. Of course, when our code first runs, it isn't, so we set it to False:

```
sorted = False
```

We begin a while loop that continues to run as long as the array is not sorted. Each round of this loop represents a pass-through of the array:

```
while not sorted:
```

Next, we preliminarily establish sorted to be True:

```
sorted = True
```

The approach here is that in each pass-through, we'll assume the array is sorted until we encounter a swap, in which case we'll change the variable back to False. If we get through an entire pass-through without having to make any swaps, sorted will remain True, and we'll know that the array is completely sorted.

Within the while loop, we begin a for loop in which we point to each pair of values in the array. We use the variable i as our first pointer, and it starts from the beginning of the array and goes until the index that has not yet been sorted:

```
for i in range(unsorted_until_index):
```

Within this loop, we compare each pair of adjacent values and swap those values if they're out of order. We also change sorted to False if we have to make a swap:

```
if list[i] > list[i+1]:
    list[i], list[i+1] = list[i+1], list[i]
    sorted = False
```

At the end of each pass-through, we know that the value we bubbled up all the way to the right is now in its correct position. Because of this, we decrement the unsorted_until_index by 1, since the index it was already pointing to is now sorted:

```
unsorted_until_index -= 1
```

The while loop ends once sorted is True, meaning the array is completely sorted. Once this is the case, we return the sorted array:

```
return list
```

The Efficiency of Bubble Sort

The Bubble Sort algorithm contains two significant kinds of steps:

- *Comparisons:* two numbers are compared with one another to determine which is greater.
- *Swaps:* two numbers are swapped with one another in order to sort them.

Let's start by determining how many *comparisons* take place in Bubble Sort.

Our example array has five elements. Looking back, you can see that in our first pass-through, we had to make four comparisons between sets of two numbers.

In our second pass-through, we only had to make three comparisons. This is because we didn't have to compare the final two numbers, since we knew that the final number was in the correct spot due to the first pass-through.

In our third pass-through, we made two comparisons, and in our fourth pass-through, we made just one comparison.

So, that's:

4 + 3 + 2 + 1 = 10 comparisons.

To put it this in a way that would hold true for arrays of all sizes, we'd say that for N elements, we make

(N - 1) + (N - 2) + (N - 3) … + 1 comparisons.

Now that we've analyzed the number of comparisons that take place in Bubble Sort, let's analyze the *swaps.*

In a worst-case scenario, where the array is sorted in descending order (the exact opposite of what we want), we'd actually need a swap for each comparison. So, we'd have 10 comparisons and 10 swaps in such a scenario for a grand total of 20 steps.

Let's look at the big picture. With an array containing five values in reverse order, we make 4 + 3 + 2 + 1 = 10 comparisons. Along with the 10 comparisons, we also have 10 swaps, totaling 20 steps.

For such an array with 10 values, we get 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45 comparisons, and another 45 swaps. That's a total of 90 steps.

With an array containing *20* values, we'd have:

19 + 18 + 17 + 16 + 15 + 14 + 13 + 12 + 11 + 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = *190* comparisons, and approximately 190 swaps, for a total of 380 steps.

Notice the inefficiency here. As the number of elements increases, the number of steps grows *exponentially.* We can see this clearly in the following table:

N Data Elements	Max # of Steps
5	20
10	90
20	380
40	1560
80	6320

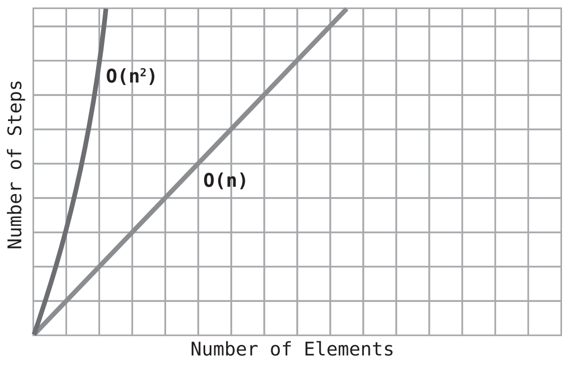
If you look at the growth of steps as N increases, you'll see that it's growing by approximately N^2. Take a look at the following table:

N Data Elements	# of Bubble Sort Steps	N^2
5	20	25
10	90	100
20	380	400
40	1560	1600
80	6320	6400

Let's express the time complexity of Bubble Sort with Big O Notation. Remember, Big O always answers the key question: if there are N data elements, how many steps will the algorithm take?

Because for N values, Bubble Sort takes N^2 steps, in Big O, we say that Bubble Sort has an efficiency of $O(N^2)$.

$O(N^2)$ is considered to be a relatively inefficient algorithm, since as the data increases, the steps increase dramatically. Look at this graph, which compares $O(N^2)$ against the faster O(N):



Note how $O(N^2)$ curves sharply upward in terms of number of steps as the data grows. Compare this with O(N), which plots along a simple, diagonal line.

One last note: $O(N^2)$ is also referred to as *quadratic time.*