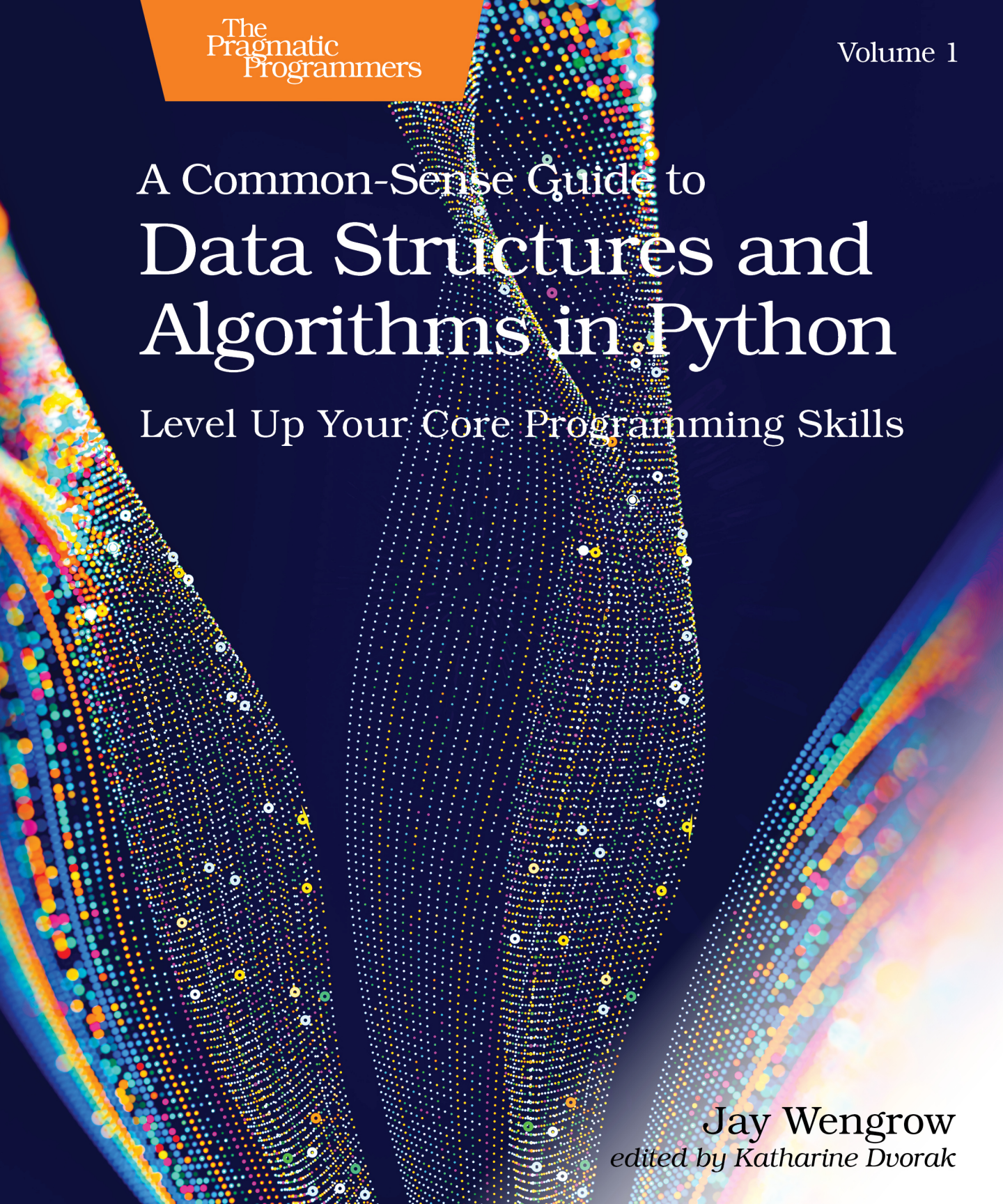
Extracted from:

A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1

Level Up Your Core Programming Skills

The Pragmatic Bookshelf

Dallas, Texas


The Pragmatic Programmers
Volume 1
A Common-Sense Guide to
Data Structures and Algorithms in Python
Level Up Your Core Programming Skills
Jay Wengrow
edited by Katharine Dvorak

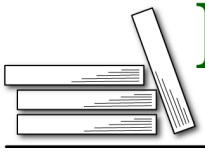
A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1

Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf

Dallas, Texas



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

Publisher: Dave Thomas
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
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: 979-8-88865-035-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2023

CHAPTER 2

Why Algorithms Matter

In the previous chapter, we took a look at our first data structures and saw how choosing the right data structure can affect the performance of our code. Even two data structures that seem so similar, such as the array and the set, can have very different levels of efficiency.

In this chapter, we're going to discover that even if we decide on a particular data structure, another major factor can affect the efficiency of our code: the proper selection of which *algorithm* to use.

Although the word *algorithm* sounds like something complex, it really isn't. An algorithm is simply *a set of instructions for completing a specific task*.

Even a process as simple as preparing a bowl of cereal is technically an algorithm, as it involves following a defined set of steps to achieve the task at hand. The cereal-preparation algorithm follows these four steps (for me, at least):

1. Grab a bowl.
2. Pour cereal into the bowl.
3. Pour milk into the bowl.
4. Dip a spoon into the bowl.

By following these steps in this particular order, we can now enjoy our breakfast.

When applied to computing, an algorithm refers to the set of instructions given to a computer to achieve a particular task. When we write any code, then, we're creating algorithms for the computer to follow and execute.

We can also express algorithms using plain English to set out the details of the instructions we plan on providing the computer. Throughout this book, I'll use both plain English as well as code to show how various algorithms work.

Sometimes, it's possible to have two different algorithms that accomplish the same task. We saw an example of this at the beginning of Chapter 1, Why Data Structures Matter, on page ?, where we had two different approaches for printing out even numbers. In that case, one algorithm had twice as many steps as the other.

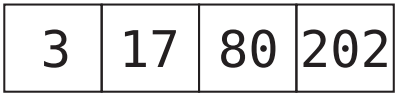
In this chapter, we'll encounter another two algorithms that solve the same problem. In this case, though, one algorithm will be faster than the other by *orders of magnitude*.

To explore these new algorithms, we'll need to take a look at a new data structure.

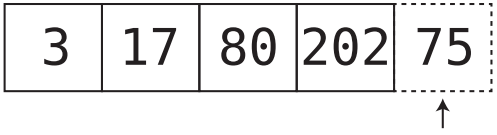
Ordered Arrays

The *ordered array* is almost identical to the classic array we saw in the previous chapter. The only difference is that ordered arrays require that the values are always kept—you guessed it—*in order*; that is, every time a value is added, it gets placed in the proper cell so that the values in the array remain sorted.

For example, let's take the array `[3, 17, 80, 202]`:

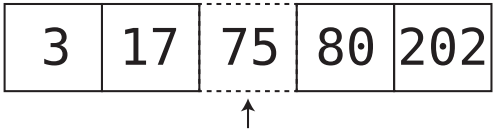


Assume we want to insert the value 75 into the array. If this array were a classic array, we could insert the 75 at the end, as follows:



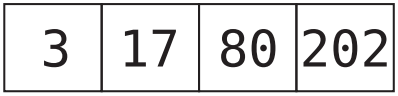
As we saw in the previous chapter, the computer can accomplish this in a single step.

On the other hand, if this were an *ordered array*, we'd have no choice but to insert the 75 in the proper spot so that the values remain in ascending order:

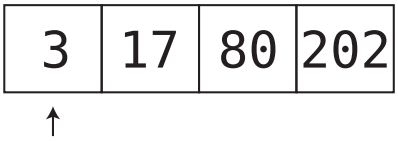


Now, this is easier said than done. The computer cannot simply drop the 75 into the right slot in a single step, because it first has to *find* the right place to insert the 75 and then shift the other values to make room for it. Let's break down this process step by step.

Let's start again with our original ordered array:



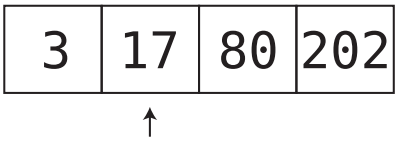
Step 1: We check the value at index 0 to determine whether the value we want to insert—the 75—should go to its left or to its right:



Because 75 is greater than 3, we know that the 75 will be inserted somewhere to its right. However, we don't know yet exactly which cell it should be inserted into, so we need to check the next cell.

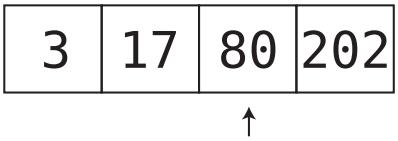
We'll call this type of step a *comparison*, where we compare the value we're inserting to a number already present in the ordered array.

Step 2: We inspect the value at the next cell:



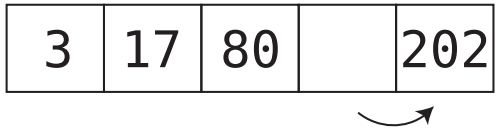
Since 75 is greater than 17, we need to move on.

Step 3: We check the value at the next cell:

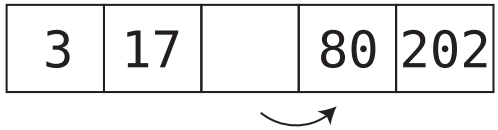


We've encountered the value 80, which is *greater* than the 75 we wish to insert. Since we've reached the first value that is greater than 75, we can conclude that the 75 must be placed immediately to the left of this 80 to maintain the order of this ordered array. To do this, we need to shift data to make room for the 75.

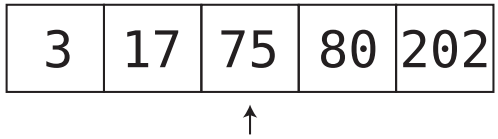
Step 4: Move the final value to the right:



Step 5: Move the next-to-last value to the right:



Step 6: We can finally insert the 75 into its correct spot:



It emerges that when inserting into an ordered array, we need to always conduct a search before the actual insertion to determine the correct spot for the insertion. This is one difference in performance between a classic array and an ordered array.

We can see in this example that there were initially four elements and that insertion took six steps. In terms of N, we'd say that for N elements in an ordered array, the insertion took N + 2 steps in total.

Interestingly, the number of steps for insertion remains similar no matter where in the ordered array our new value ends up. If our value ends up toward the beginning of the ordered array, we have fewer comparisons and more shifts. If our value ends up toward the end, we get more comparisons but fewer shifts. The fewest steps occur when the new value winds up at the very end, since no shifts are necessary. In this case, we take N steps to compare the new value with all N existing values, plus one step for the insertion itself, yielding a total of N + 1 steps.

While insertion is less efficient for an ordered array than for a classic array, the ordered array has a secret superpower when it comes to searching.