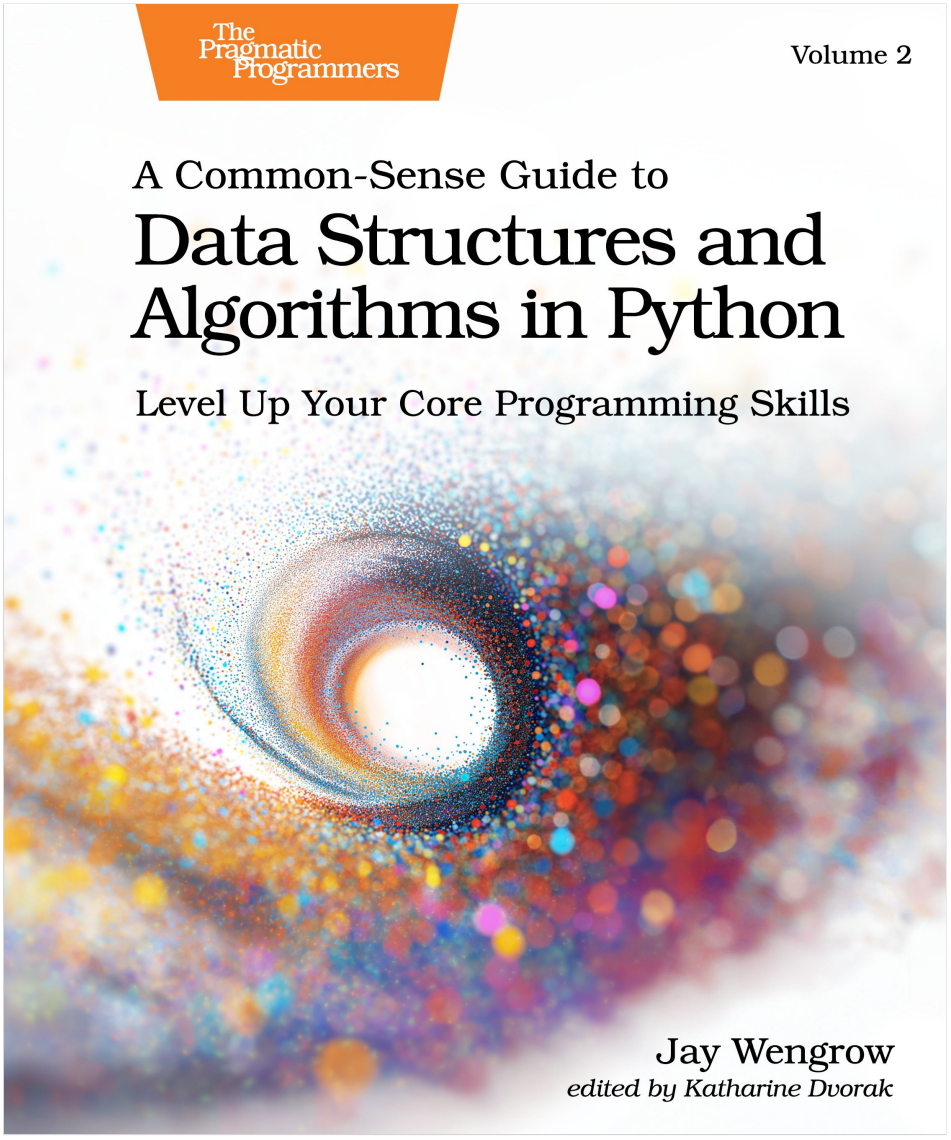


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 1

Getting Things in Order with Mergesort

One of the most fundamental concepts central to understanding data structures and algorithms is understanding time complexity. *A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1* discussed this at great length, placing heavy emphasis on the use of Big O notation as a tool for articulating the speed of algorithms. *Volume 2* takes things to the next level and adds more nuance to the conversation of time complexity. You'll learn that while counting an algorithm's steps and Big O serve as an important model for measuring time complexity, that's not quite the full story. This is because there are additional factors out there that can affect an algorithm's "true" speed. In this chapter, we'll take a look at one of those factors.

We'll also contrast two of the most famous "fast" sorting algorithms—Quicksort and Mergesort—and use the contrast to tease out the limits of the Big O model. Quicksort was covered in Volume 1, Chapter 13, and Mergesort is the main focus of *this* chapter. We'll look at basic array merges, and then discover how they form the backbone of Mergesort. From there, we'll have an important conversation about algorithmic trade-offs. Finally, you'll discover how the counting-steps model is not the end-all of determining an algorithm's true speed.

Ready? Let's dive in.

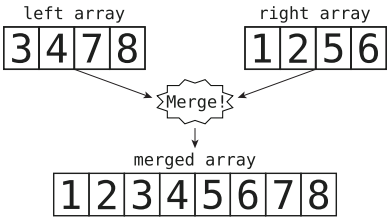
Merging Arrays

Mergesort sorts arrays by relying on another more basic algorithm known as *merging arrays*, or simply *merging*, for short. In this context, to merge arrays means to take two arrays that are already sorted and copy all of their values into a third array and have it so the third array is also completely sorted. Let's look at a basic example.

Let's say we have the two arrays [3, 4, 7, 8] and [1, 2, 5, 6]. Note that each array is already sorted. I can't emphasize this enough: merging arrays only works if the arrays are *already sorted*.

The goal of merging is to take all the values from both arrays and copy them into a third array, which will contain *all* the values in sorted order. The result of merging these two arrays will be: [1, 2, 3, 4, 5, 6, 7, 8].

The following diagram shows the finished product of a merge:



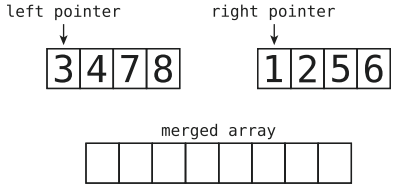
The merging algorithm follows these steps:

1. Initialize a "left pointer" and have it point to the first index of the *left* array.
2. Initialize a "right pointer" and have it point at the first index of the *right* array.
3. Create a third, empty array. This will be the "merged" array. By the end of the merge, the merged array will contain all the values from the left and right arrays and be in sorted order.
4. Run a loop until either the left pointer or right pointer reaches the end of its array. Within the loop, do the following: (a.) Compare the value of the left pointer with the value of the right pointer and determine which value is *lower*; (b.) Take the lower value and append it to the merged array; (c.) Whichever pointer was pointing to the lower value gets incremented so that it points to the next index of its array. (If at any point the two values we're comparing are equal, we can arbitrarily append the value of the *left* array and move its pointer along.)
5. Once the loop is complete, either the left or right array will be "unfinished" in that it will still have values that were not yet copied to the merged array. This triggers the "final phase" of the algorithm.
6. Final phase: take all the remaining values of the "unfinished" array and append them, in order, to the merged array.

Let's now take a look at the merge algorithm in action, using an example.

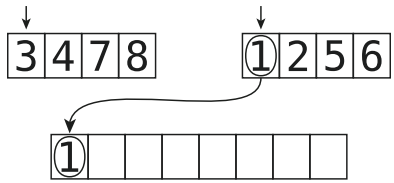
Merging in Action

Following is a diagram representing the arrays we want to merge. Note that the left and right pointers point to the beginning of each of their respective arrays. The merged array starts out as empty:

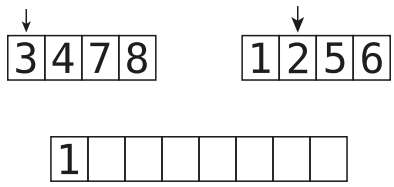


Now, let's walk through the process of merging them. Each of our "steps" below consist of two parts. Part A appends a value to the merged array, while Part B consists of moving either the left or right pointer.

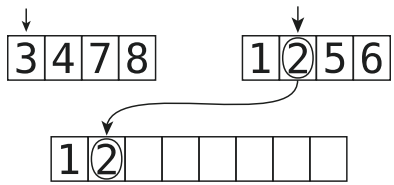
Step 1A: First, we compare the left pointer's value to the right pointer's value. We take whichever value is lower and append it to the array. In this case, the 1 is lower, so we append the 1:



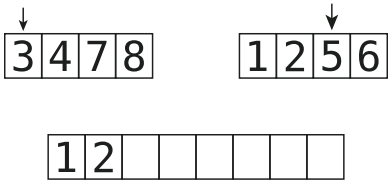
Step 1B: Because we appended a value of the right pointer, we now increment the right pointer so that it points to the next value of the right array:



Step 2A: We compare the left pointer's 3 with the right pointer's 2. Because 2 is lower than 3, we append the 2 to the merged array:

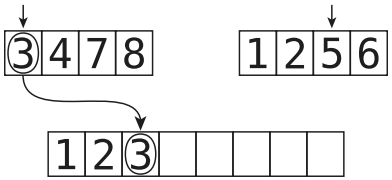


Step 2B: Since we once again appended a value belonging to the right pointer, we move the right pointer another notch rightward:



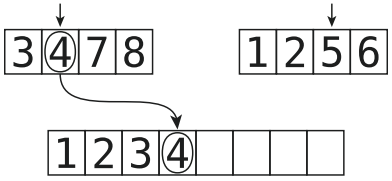
To expedite the remainder of this walkthrough, I'm only going show visuals for the appends, that is, Part A of each step. I'll still mention the pointer movements (Part B), but won't show them in a dedicated diagram.

Step 3: We compare the values of the two pointers. The 3 is lower, so we append it to the merged array:



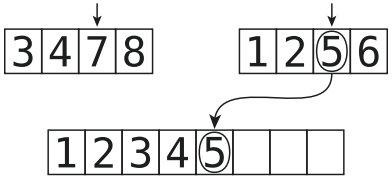
Accordingly, we'll move the left pointer along.

Step 4: We compare the 4 with the 5. We append the 4 to the merged array because it's lower:



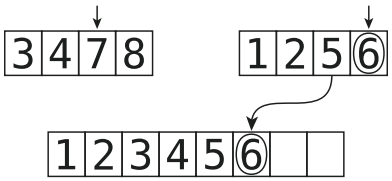
This means we'll also increment the left pointer again.

Step 5: We now compare the 7 with the 5. The right pointer's 5 is lower, so that's the value we add to the merged array:



At this point, we'll move the right pointer one notch rightward.

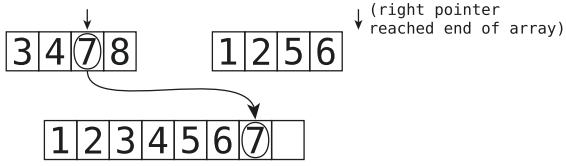
Step 6: Next, we compare the 7 with the 6. We copy the 6 to the merged array:



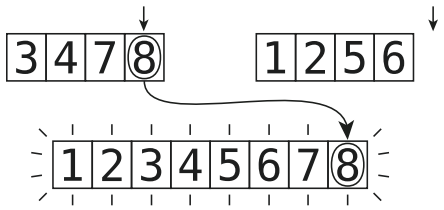
The 6 came from the right pointer, so we move that pointer another notch to the right.

Step 7: This is a noteworthy step, because the right pointer has now moved beyond the end of the right array. This means we no longer need to perform any comparisons, since we've exhausted all the values from the right array.

This triggers the final phase of the merge algorithm, in which we simply take all the values of the remaining array (in this case, the left array) and append each one to the merged array. So let's go ahead and append the 7 to the merged array:



Step 8: Similarly, we append the 8 to the merged array:



The merge algorithm is now complete! The merged array contains all the values from the left and right arrays, *and* is also sorted.