4 Sorting Algorithms

I think the bubble sort would be the wrong way to go.

Barack Obama

As a computer scientist, in addition to searching for data, you will often have to sort data. **Sorting data** means arranging it in a meaningful way. For example, if you have a list of numbers, you could sort them from the smallest to the largest (ascending). Or, imagine you are building an app that keeps track of the books each user has read. In an app like this, you might want to allow the user to see their books sorted in different ways. For example, you could give them the option to see their books sorted from the shortest book to the longest, oldest to newest, or newest to oldest.

There are many different sorting algorithms to help you sort data, each with strengths and weaknesses. For example, some sort algorithms work best in specific situations, like if an iterable is nearly sorted. In this chapter, you will learn about bubble sort, insertion sort, and merge sort. Other popular sorts include quicksort, shell sort, and heap sort. There are many sorting algorithms you will have to use in only rare circumstances, so after teaching you a few sorts, I spend the remainder of the chapter showing you how to use Python's built-in functions to sort data, something you will often use in real-world programming.

When you are building your programs in the real world, you should almost always use your programming language's built-in sorting function. You should not implement the classic sorting algorithms discussed here (except in rare circumstances) because modern programming languages like Python have built-in sorting functions that are faster. However, learning a few of the classic sorting algorithms will help you better understand time complexity and teach you concepts you can use in situations other than sorting, such as the merge step in a merge sort.

Bubble Sort

Bubble sort is a sorting algorithm where you iterate through a list of numbers, compare each number to the next number, and swap them if they are out of order. Computer scientists call it a bubble sort because the numbers with the highest values "bubble up" to the end of the list, and the numbers with the smallest values move to the beginning of the list as the algorithm progresses.

Say you have the following list:

```
[32, 1, 9, 6]
```

First, you compare 1 and 32:

```
[32, 1, 9, 6]
```

Thirty-two is bigger, so you swap them:

```
[1, 32, 9, 6]
```

Next, you compare 32 and 9:

```
[1, 32, 9, 6]
```

Thirty-two is larger, so you swap them again:

```
[1, 9, 32, 6]
```

Finally, you compare 32 and 6:

```
[1, 9, 32, 6]
```

Once again, you swap them:

```
[1, 9, 6, 32]
```

As you can see, 32 "bubbled up" to the end of the list. However, your list is still not in order because 9 and 6 are not in the right spots. So, your algorithm starts at the beginning again and compares 1 and 9:

```
[1, 9, 6, 32]
```

Nothing happens because 1 is not greater than 9. Next, it compares 9 and 6:

```
[1, 9, 6, 32]
```

Nine is greater than 6, so you swap them, and your list is now in order:

```
[1, 6, 9, 32]
```

In a bubble sort, the largest number in your list will move to the end of your list at the end of your algorithm's first iteration, but if the smallest number in your list starts at the end, it will take multiple passes for your algorithm to move it to the beginning of your list. For instance, in this example, 32 ended up at the end of your list after one iteration. Say your list started like this, though:

```
[32, 6, 9, 1]
```

In this case, it will take four iterations to move 1 from the end of the list to the beginning.

It can be helpful to use a bubble sort visualizer to better understand how this algorithm works. There are many bubble sort visualizers you can search for on the internet that can cement your understanding of how this sorting algorithm works. I recommend doing this for the sorting algorithms you will learn about in this chapter.

Here is how to write a bubble sort algorithm in Python:

```
def bubble_sort(a_list):
    list_length = len(a_list) - 1
    for i in range(list_length):
        for j in range(list_length):
            if a_list[j] > a_list[j + 1]:
                a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
    return a_list
```

Your function `bubble_sort` takes a list of numbers, called `a_list`, as a parameter:

```
def bubble_sort(a_list):
```

Inside your function, you get your list's length, subtract 1, and save the result in `list_length` to control how many iterations your algorithm will make:

```
list_length = len(a_list) - 1
```

Your function has two nested loops so that you can iterate through your list and make comparisons:

```
for i in range(list_length):
    for j in range(list_length)
```

Inside your inner `for` loop, you use an `if` statement to compare the current number to the one after it by adding 1 to the current number's index:

```
if a_list[j] > a_list[j + 1]:
```

This line of code is the current number:

```
a_list[j]
```

This line of code is the next number in your list:

```
a_list[j + 1]
```

If the current number is greater than the next number, you swap them. The following Python syntax allows you to swap two items without loading one of the items into a temporary variable:

```
a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
```

Your algorithm's comparisons all happen inside your inner loop. Your outer loop is there only to keep your algorithm running for as many swaps as it takes to put your list in order. Take, for example, the list from the beginning of the section:

```
[32, 1, 9, 6]
```

After one inner-loop iteration, your list looks like this:

```
[1, 9, 6, 32]
```

Recall, however, that this list is not in order. If you had only your inner loop, your algorithm would end prematurely, and your list would not be in order. That is why you need your outer loop: to start your algorithm's inner loop over from the beginning and repeat it until your list is in order.

You can improve the efficiency of your bubble sort by including `- i` in your second `for` loop. That way, your inner loop will not compare the last two numbers the first time through the loop; the second time through the loop, it will not compare the last three numbers; and so on.

```
def bubble_sort(a_list):
    list_length = len(a_list) - 1
    for i in range(list_length):
        for j in range(list_length - i):
            if a_list[j] > a_list[j + 1]:
                a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
    return a_list
```

You don't have to make these comparisons because the highest numbers bubble to the end of your list. For instance, in the initial example at the beginning of this section, you saw 32 bubbled up to the end of your list after the first iteration of your sort. That means the largest number is at the end of your list after the first iteration; the second largest number is at the next-to-the-last spot after the second iteration, and so forth. This means you do not need to compare the other numbers to them and can end your loop early. For example, take a look at the list from the beginning of this section:

```
[32, 1, 9, 6]
```

After one full inner-loop iteration, it looks like this:

```
[1, 9, 6, 32]
```

After one iteration, the largest number moves to the end of your list, so you no longer have to compare numbers to 32 because you know it is the largest number in your list.

The second time through your inner loop, the second-largest value will move to its final position, second from the end, etc.

```
[1, 6, 9, 32]
```

So, each time through your inner loop, your inner loop can terminate sooner.

You can also make your bubble sort more efficient by adding a variable that keeps track of whether your algorithm made any swaps in your inner loop. If you get through an inner loop with no swaps, your list is sorted, and you can exit the loop and return your list without any further processing.

```
def bubble_sort(a_list):
    list_length = len(a_list) - 1
    for i in range(list_length):
        no_swaps = True
        for j in range(list_length - i):
            if a_list[j] > a_list[j + 1]:
                a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
                no_swaps = False
        if no_swaps:
            return a_list
    return a_list
```

In this case, you added a variable called `no_swaps` that starts as `True` at the beginning of your inner loop. If you swap two numbers inside your inner loop, you set it to `False`. If you make it through your inner loop and `no_swaps` is `True`, your list is sorted, and you can end your algorithm. This small change makes your bubble sort run significantly faster when a list starts nearly sorted.

When to Use Bubble Sort

In this implementation of bubble sort, your algorithm sorts numbers, but you can also write a bubble sort (or any other sort) that sorts strings. For example, you could modify a bubble sort to sort strings alphabetically by each word's first letter.

A bubble sort's main advantage is how simple it is to implement, making it a good starting point for teaching sorting algorithms. Because a bubble sort relies on two nested `for` loops, its time complexity is O(*n***2), which means while it can be acceptable for small sets of data, it is not an efficient choice for larger data sets.

A bubble sort is also stable. A **stable sort** is one that does not disturb any sequences other than the one specified by the sort key. For example, say you have a database containing records for these four animals:

Akita

Bear

Tiger

Albatross

If you sort by the first letter of the first character, you expect this output:

Stable Sort

Akita

Albatross

Bear

Tiger

This output is an example of a stable sort because Akita and Albatross are in the same order as in the original list, even though your sort looked only at the first letter of each name. An unstable sort might disrupt the original order of the two animal names that start with *A*, so Albatross might come before Akita even though Akita came before Albatross in the original list.

Unstable Sort

Albatross

Akita

Bear

Tiger

In other words, in a stable sort, when there are two equal keys, the items maintain their original order.

Despite its stability, because a bubble sort is O(*n***2) and because there are other, more efficient sorts you can use (which you are about to learn), you are unlikely to see anyone using a bubble sort outside of the classroom.

Insertion Sort

An **insertion sort** is a sorting algorithm where you sort data like you sort a deck of cards. First, you split a list of numbers into two: a sorted left half and an unsorted right half. Then, you sort the left half the same way you sort a hand of playing cards. For example, when you sort a hand of five cards in ascending order, you go through your cards one by one, inserting each card to the right of every card lower than it.

Here is how an insertion sort algorithm works on a list with four elements: 6, 5, 8, and 2. Your algorithm starts with the second item in your list, which is 5:

```
[6, 5, 8, 2]
```

Next, you compare the current item to the previous item in your list. Six is greater than 5, so you swap them:

```
[5, 6, 8, 2]
```

Now the left half of your list is sorted, but the right half is not:

```
[5, 6, 8, 2]
```

Then, you move to the third item in your list. Six is not greater than 8, so you don't swap 8 and 6:

```
[5, 6, 8, 2]
```

Because the left half of your list is sorted, you do not have to compare 8 and 5:

```
[5, 6, 8, 2]
```

Next, you compare 8 and 2:

```
[5, 6, 8, 2]
```

Because 8 is greater than 2, you go one by one through the sorted left half of the list, comparing 2 to each number until it arrives at the front and the entire list is sorted:

```
[2, 5, 6, 8]
```