## Big O: Count the Steps

Instead of focusing on units of time, Big O achieves consistency by focusing only on the *number of steps* that an algorithm takes.

In Chapter 1, *Why Data Structures Matter*, we discovered that reading from an array takes just one step, no matter how large the array is. The way to express this in Big O Notation is:

O(1)

Many pronounce this verbally as "Big Oh of 1." Others call it "Order of 1." My personal preference is "Oh of 1." While there is no standardized way to *pronounce* Big O Notation, there is only one way to *write* it.

O(1) simply means that the algorithm takes the same number of steps no matter how much data there is. In this case, reading from an array always takes just one step no matter how much data the array contains. On an old computer, that step may have taken twenty minutes, and on today's hardware it may take just a nanosecond. But in both cases, the algorithm takes just a single step.

Other operations that fall under the category of O(1) are the insertion and deletion of a value at the end of an array. As we've seen, each of these operations takes just one step for arrays of any size, so we'd describe their efficiency as O(1).

Let's examine how Big O Notation would describe the efficiency of linear search. Recall that linear search is the process of searching an array for a particular value by checking each cell, one at a time. In a worst-case scenario, linear search will take as many steps as there are elements in the array. As we've previously phrased it: for N elements in the array, linear search can take up to a maximum of N steps.

The appropriate way to express this in Big O Notation is:

O(N)

I pronounce this as "Oh of N."

O(N) is the “Big O” way of saying that for N elements inside an array, the algorithm would take N steps to complete. It’s that simple.

**So Where's the Math?**

As I mentioned, in this book, I’m taking an easy-to-understand approach to the topic of Big O. That’s not the only way to do it; if you were to take a traditional college course on algorithms, you’d probably be introduced to Big O from a mathematical perspective. Big O is originally a concept from mathematics, and therefore it’s often described in mathematical terms. For example, one way of describing Big O is that it describes the upper bound of the growth rate of a function, or that if a function g(x) grows no faster than a function f(x), then g is said to be a member of O(f). Depending on your mathematics background, that either makes sense, or doesn’t help very much. I’ve written this book so that you don’t need as much math to understand the concept.

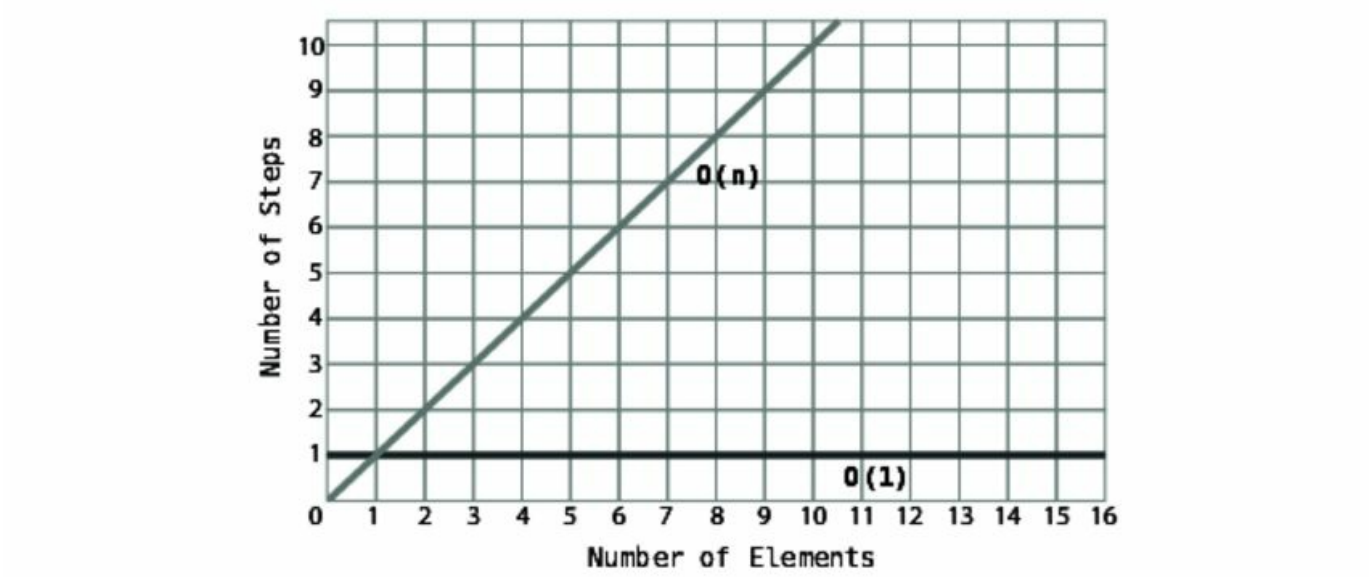
If you want to dig further into the math behind Big O, check out *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (MIT Press, 2009) for a full mathematical explanation. Justin Abrahms provides a pretty good definition in his article: https://justin.abrah.ms/computer-science/understanding-big-o-formal-definition.html. Also, the Wikipedia article on Big O (https://en.wikipedia.org/wiki/Big_O_notation) takes a fairly heavy mathematical approach.

# Constant Time vs. Linear Time

Now that we've encountered O(N), we can begin to see that Big O Notation does more than simply describe the number of steps that an algorithm takes, such as a hard number such as 22 or 400. Rather, it describes how many steps an algorithm takes *based on the number of data elements that the algorithm is acting upon*. Another way of saying this is that Big O answers the following question: *how does the number of steps change as the data increases?*

An algorithm that is O(N) will take as many steps as there are elements of data. So when an array increases in size by one element, an O(N) algorithm will increase by one step. An algorithm that is O(1) will take the same number of steps no matter how large the array gets.

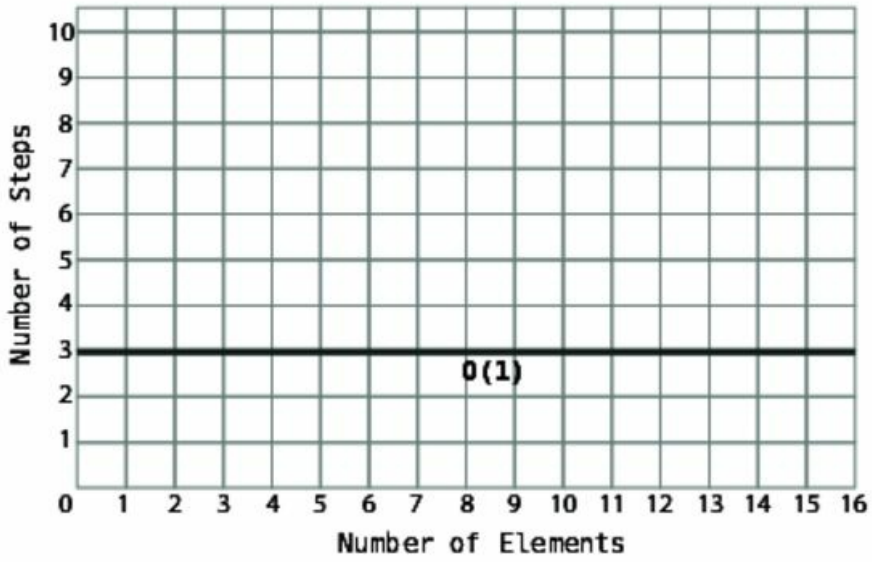
Look at how these two types of algorithms are plotted on a graph.




You'll see that O(N) makes a perfect diagonal line. This is because for every additional piece of data, the algorithm takes one additional step. Accordingly, the more data, the more steps the algorithm will take. For the record, O(N) is also known as *linear time*.

Contrast this with O(1), which is a perfect horizontal line, since the number of steps in the algorithm remains constant no matter how much data there is. Because of this, O(1) is also referred to as *constant time*.

As Big O is primarily concerned about how an algorithm performs across varying amounts of data, an important point emerges: an algorithm can be described as O(1) even if it takes more than one step. Let's say that a particular algorithm always takes *three* steps, rather than one—but it always takes these three steps no matter how much data there is. On a graph, such an algorithm would look like this:

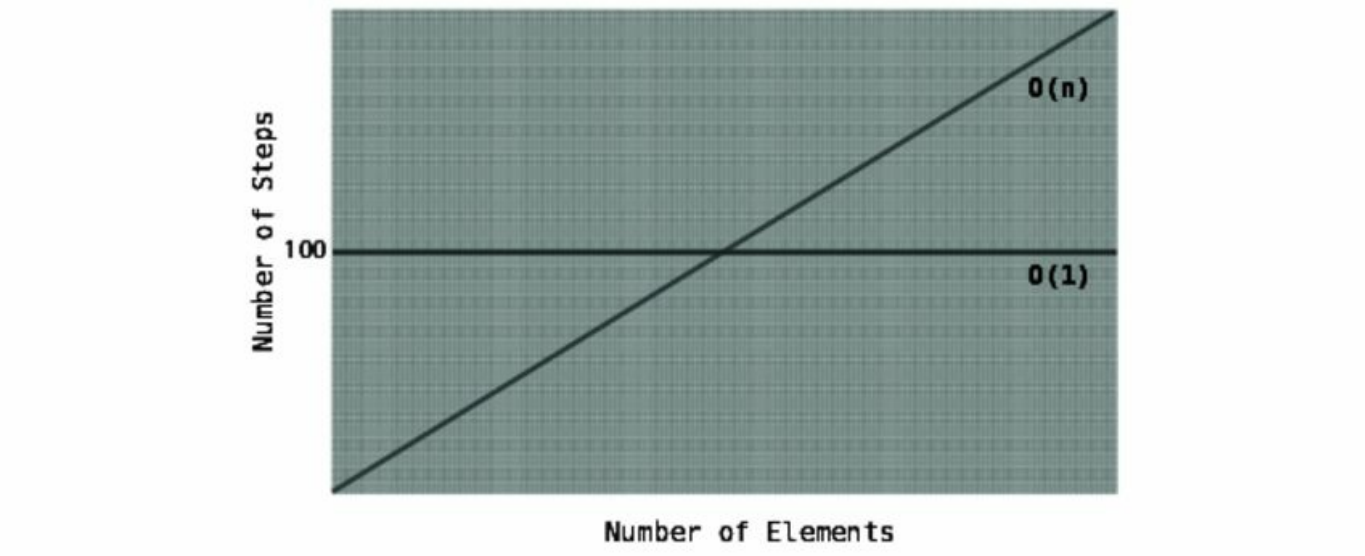



Because the number of steps remains constant no matter how much data there is, this would also be considered constant time and be described by Big O Notation as O(1). Even though the algorithm technically takes three steps rather than one step, Big O Notation considers that trivial. O(1) is the way to describe *any* algorithm that doesn't change its number of steps even when the data increases.

If a three-step algorithm is considered O(1) as long as it remains constant, it follows that even a constant 100-step algorithm would be expressed as O(1) as well. While a 100-step algorithm is less efficient than a one-step algorithm, the fact that it is O(1) still makes it *more efficient* than any O(N) algorithm.

Why is this?

See the following graph:




As the graph depicts, for an array of fewer than one hundred elements, O(N) algorithm takes fewer steps than the O(1) 100-step algorithm. At exactly one hundred elements, the two algorithms take the same number of steps (100). But here's the key point: for *all arrays greater than one hundred*, the O(N) algorithm takes more steps.

Because there will always be *some* amount of data in which the tides turn, and O(N) takes more steps from that point until infinity, O(N) is considered to be, on the whole, less efficient than O(1).

The same is true for an O(1) algorithm that always takes one million steps. As the data increases, there will inevitably reach a point where O(N) becomes less efficient than the O(1) algorithm, and will remain so up until an infinite amount of data.

## Same Algorithm, Different Scenarios

As we learned in the previous chapters, linear search isn't *always* O(N). It's true that if the item we're looking for is in the final cell of the array, it will take N steps to find it. But where the item we're searching for is found in the *first* cell of the array, linear search will find the item in just one step. Technically, this would be described as O(1). If we were to describe the efficiency of linear search in its totality, we'd say that linear search is O(1) in a *best-case* scenario, and O(N) in a *worst-case* scenario.

While Big O effectively describes both the best- and worst-case scenarios of a given algorithm, Big O Notation generally refers to *worst-case scenario* unless specified otherwise. This is why most references will describe linear search as being O(N) even though it *can* be O(1) in a best-case scenario.

The reason for this is that this "pessimistic" approach can be a useful tool: knowing exactly how inefficient an algorithm can get in a worst-case scenario prepares us for the worst and may have a strong impact on our choices.

Chapter 4

# Speeding Up Your Code with Big O

Big O Notation is a great tool for comparing competing algorithms, as it gives an objective way to measure them. We've already been able to use it to quantify the difference between binary search vs. linear search, as binary search is O(log N)—a much faster algorithm than linear search, which is O(N).

However, there may not always be two clear alternatives when writing everyday code. Like most programmers, you probably use whatever approach pops into your head first. With Big O, you have the opportunity to compare your algorithm to *general algorithms out there in the world*, and you can say to yourself, "Is this a fast or slow algorithm as far as algorithms generally go?"

If you find that Big O labels your algorithm as a "slow" one, you can now take a step back and try to figure out if there's a way to optimize it by trying to get it to fall under a faster category of Big O. This may not always be possible, of course, but it's certainly worth thinking about before concluding that it's not.

In this chapter, we'll write some code to solve a practical problem, and then measure our algorithm using Big O. We'll then see if we might be able to modify the algorithm in order to give it a nice efficiency bump. (Spoiler: we will.)

# 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.

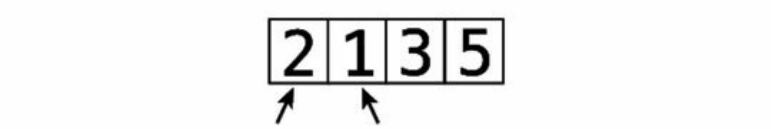
*Sorting algorithms* have been the subject of extensive research in computer science, and tens of such algorithms have been developed over the years. They all solve the following problem:

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

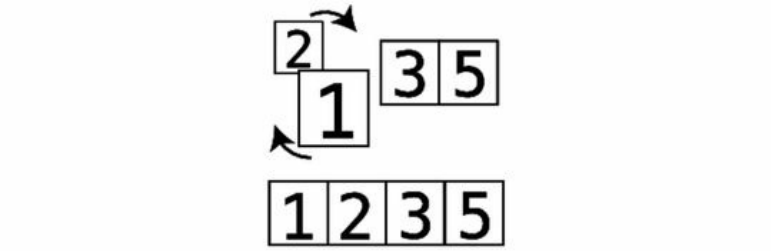
In this and the following chapters, we're going to encounter a number of these sorting algorithms. Some of the first ones we'll be learning about are known as "simple sorts," in that they are easier to understand, but are not as efficient as some of the faster sorting algorithms out there.

*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:

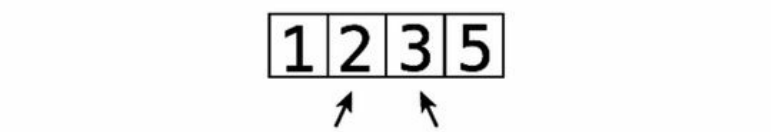



2. If the two items are out of order (in other words, the left value is greater than the right value), swap them:




(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.

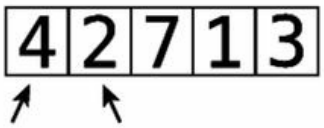
## Bubble Sort in Action

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.

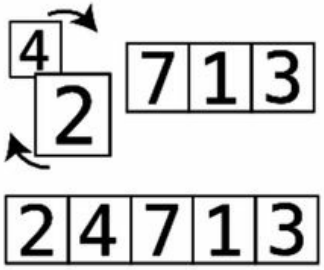
Let's begin Passthrough #1:

This is our starting array:

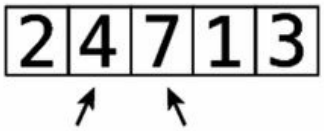

Step #1: First, we compare the 4 and the 2. They're out of order:




Step #2: So we swap them:

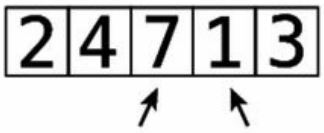



Step #3: Next, we compare the 4 and the 7:

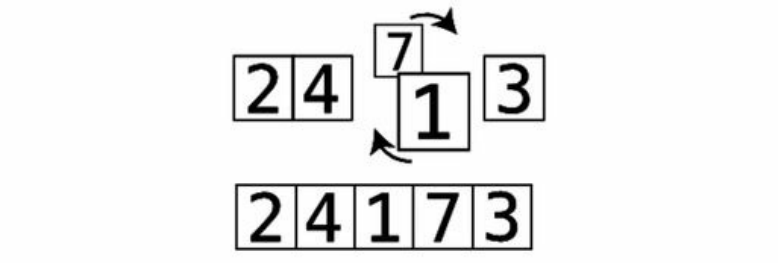



They're in the correct order, so we don't need to perform any swaps.

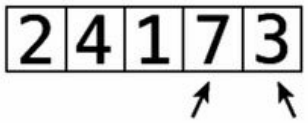
Step #4: We now compare the 7 and the 1:



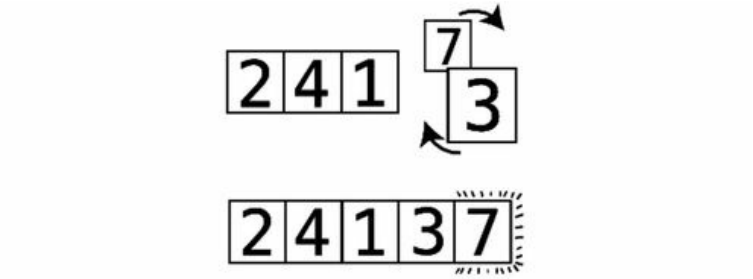

Step #5: They're out of order, so we swap them:




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




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



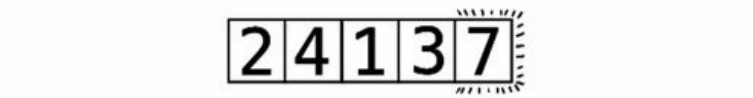

We now know for a fact that the 7 is in its correct position within the array, because we kept moving it along to the right until it reached its proper place. We've put little lines surrounding it to indicate this fact.

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.

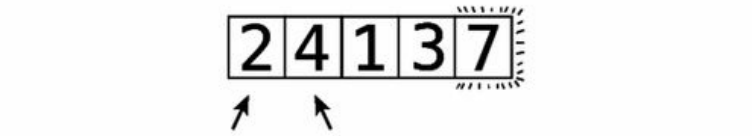
Since we made at least one swap during this passthrough, we need to conduct another one.

We begin Passthrough #2:

The 7 is already in the correct position:

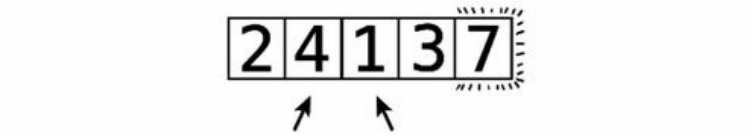



Step #8: We being by comparing the 2 and the 4:

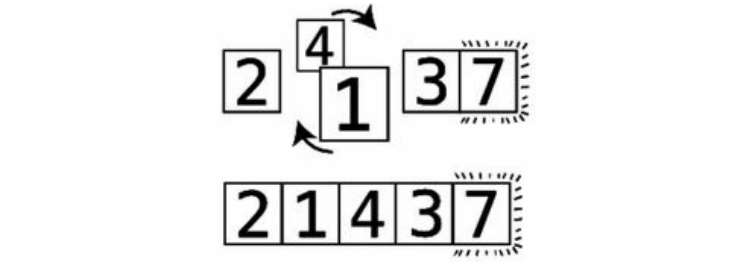



They’re in the correct order, so we can move on.

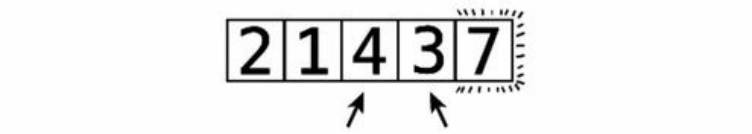
Step #9: We compare the 4 and the 1:



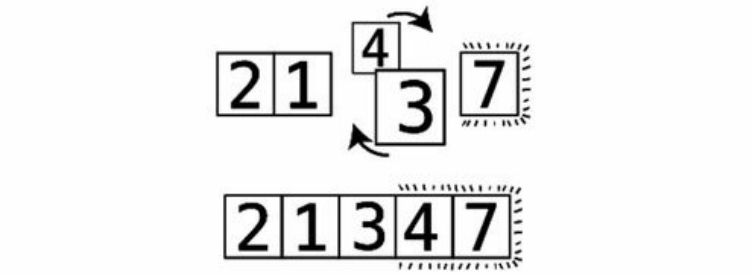

Step #10: They’re out of order, so we swap them:




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




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

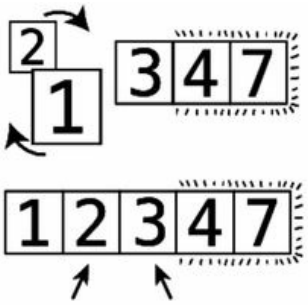



We don’t have to compare the 4 and the 7 because we know that the 7 is already in its correct position from the previous passthrough. And now we also know that the 4 is bubbled up to its correct position as well. This concludes our second passthrough. Since we made at least one swap during this passthrough,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


1 2 3 4 7

## Bubble Sort Implemented

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]:
                sorted = False
                list[i], list[i+1] = list[i+1], list[i]
        unsorted_until_index = unsorted_until_index - 1

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

Let's break this down line by line. We'll first present the line of code, followed by its explanation.

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

We keep track of up to which index is still unsorted with the `unsorted_until_index` variable. At the beginning, the array is totally unsorted, so we initialize this variable to be the final index in the array.

```
sorted = False
```

We also create a `sorted` variable that will allow us to keep track whether the array is fully sorted. Of course, when our code first runs, it isn't.

```
while not sorted:
    sorted = True
```

We begin a `while` loop that will last as long as the array is not sorted. Next, we preliminarily establish `sorted` to be `True`. We'll change this back to `False` as soon as we have to make any swaps. If we get through an entire passthrough without having

to make any swaps, we'll know that the array is completely sorted.

```
    for i in range(unsorted_until_index):
        if list[i] > list[i+1]:
            sorted = False
            list[i], list[i+1] = list[i+1], list[i]
```

Within the **while** loop, we begin a **for** loop that starts from the beginning of the array and goes until the index that has not yet been sorted. Within this loop, we compare every pair of adjacent values, and swap them if they're out of order. We also change **sorted** to **False** if we have to make a swap.

```
    unsorted_until_index = unsorted_until_index - 1
```

By this line of code, we've completed another passthrough, and can safely assume that the value we've bubbled up 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.

Each round of the **while** loop represents another passthrough, and we run it until we know that our array is fully sorted.

## The Efficiency of Bubble Sort

The Bubble Sort algorithm contains two 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 passthrough, we had to make four comparisons between sets of two numbers.

In our second passthrough, we had to make only 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 passthrough.

In our third passthrough, we made two comparisons, and in our fourth passthrough, we made just one comparison.

So, that's:

4 + 3 + 2 + 1 = 10 comparisons.

To put it more generally, 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 not just randomly shuffled, but sorted in descending order (the exact opposite of what we want), we'd actually need a swap for each comparison. So we'd have ten comparisons and ten swaps in such a scenario for a grand total of twenty steps.

So let's look at the complete picture. With an array containing ten elements in reverse order, we'd have:

9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45 comparisons, and another forty-five swaps. That's a total of ninety steps.

With an array containing *twenty* elements, 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 increase, the number of steps grows exponentially. We can see this clearly with the following table:

| **N data elements** | **Max # of steps** |
|---|---|
| 5 | 20 |
| 10 | 90 |
| 20 | 380 |
| 40 | 1560 |
| 80 | 6320 |

If you look precisely at the growth of steps as N increases, you'll see that it's growing by approximately $N^2$.

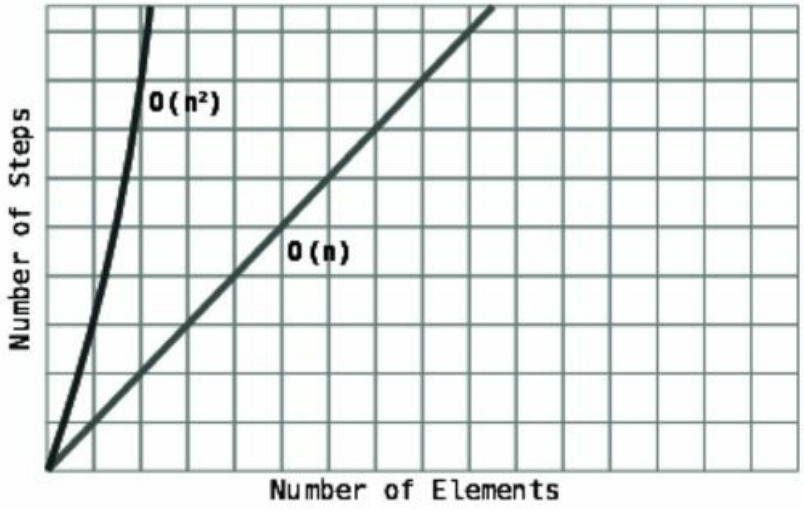
| **N data elements** | **# of Bubble Sort steps** | **$N^2$** |
|---|---|---|
| 5 | 20 | 25 |
| 10 | 90 | 100 |
| 20 | 380 | 400 |
| 40 | 1560 | 1600 |
| 80 | 6320 | 6400 |

Therefore, in Big O Notation, we would say that Bubble Sort has an efficiency of $O(N^2)$.

Said more officially: in an $O(N^2)$ algorithm, for N data elements, there are roughly

$N^2$ steps.

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




Note how $O(N^2)$ curves sharply upwards 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*.

## Wrapping Up

It's clear that having a solid understanding of Big O Notation can allow us to identify slow code and select the faster of two competing algorithms. However, there are situations in which Big O Notation will have us believe that two algorithms have the same speed, while one is actually faster. In the next chapter, we're going to learn how to evaluate the efficiencies of various algorithms even when Big O isn't nuanced enough to do so.

Chapter 5

# Optimizing Code with and Without Big O

We've seen that Big O is a great tool for contrasting algorithms and determining which algorithm should be used for a given situation. However, it's certainly not the *only* tool. In fact, there are times where two competing algorithms may be described in exactly the same way using Big O Notation, yet one algorithm is significantly faster than the other.

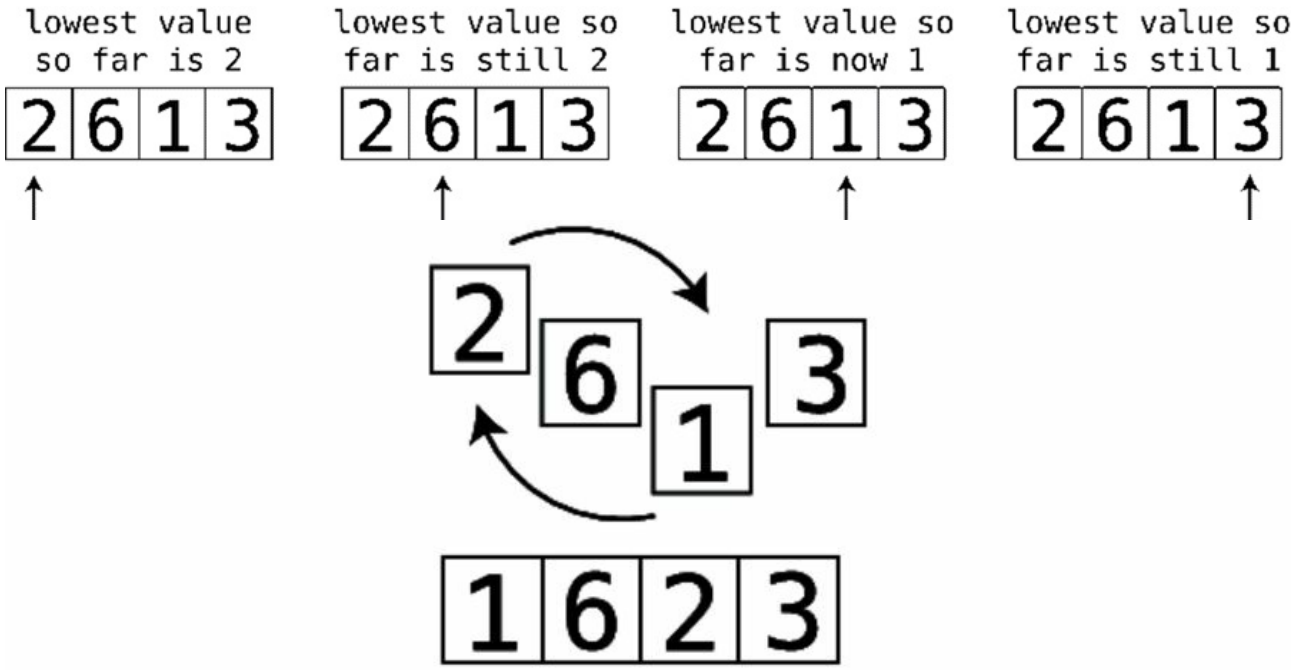
In this chapter, we're going to learn how to discern between two algorithms that *seem* to have the same efficiency, and select the faster of the two.

## Selection Sort

In the previous chapter, we explored an algorithm for sorting data known as Bubble Sort, which had an efficiency of $O(N^2)$. We're now going to dig into another sorting algorithm called *Selection Sort*, and see how it measures up to Bubble Sort.

The steps of Selection Sort are as follows:

1. We check each cell of the array from left to right to determine which value is least. As we move from cell to cell, we keep in a variable the lowest value we've encountered so far. (Really, we keep track of its index, but for the purposes of the following diagrams, we'll just focus on the actual value.) If we encounter a cell that contains a value that is even less than the one in our variable, we replace it so that the variable now points to the new index. See the following diagram:




2. Once we've determined which index contains the lowest value, we swap that index with the value we began the passthrough with. This would be index 0 in the first passthrough, index 1 in the second passthrough, and so on and so forth. In the next diagram, we make the swap of the first passthrough:

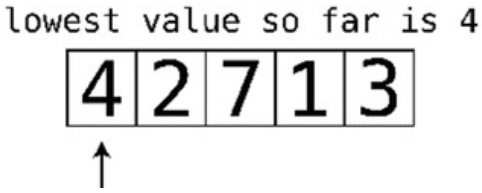
3. Repeat steps 1 and 2 until all the data is sorted.

## Selection Sort in Action

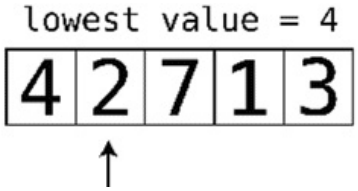
Using the example array **[4, 2, 7, 1, 3]**, our steps would be as follows:

We begin our first passthrough:

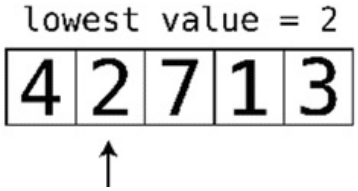
We set things up by inspecting the value at index 0. By definition, it's the lowest value in the array that we've encountered so far (as it's the *only* value we've encountered so far), so we keep track of its index in a variable:



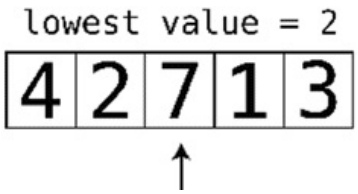

Step #1: We compare the 2 with the lowest value so far (which happens to be 4):



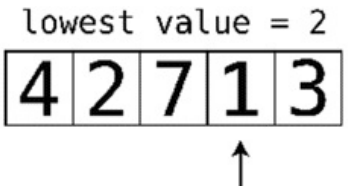

The 2 is even less than the 4, so it becomes the lowest value so far:



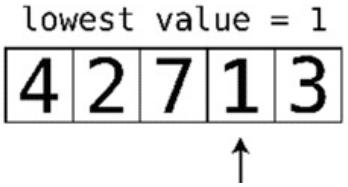

Step #2: We compare the next value—the 7—with the lowest value so far. The 7 is greater than the 2, so 2 remains our lowest value:



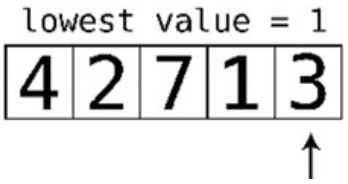

Step #3: We compare the 1 with the lowest value so far:



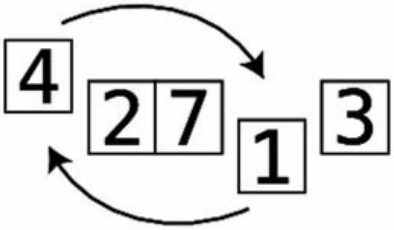

Since the 1 is even less than the 2, the 1 becomes our new lowest value:




Step #4: We compare the 3 to the lowest value so far, which is the 1. We've reached the end of the array, and we've determined that 1 is the lowest value out of the entire array:



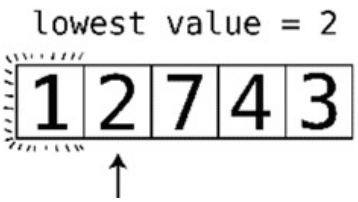

Step #5: Since 1 is the lowest value, we swap it with whatever value is at index 0—the index we began this passthrough with:



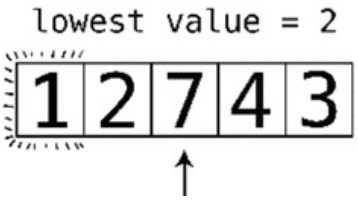

We have now determined that the 1 is in its correct place within the array:


We are now ready to begin our second passthrough:

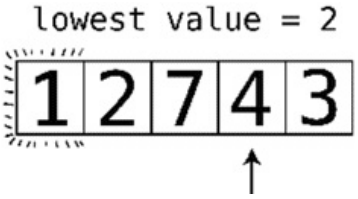
Setup: the first cell—index 0—is already sorted, so this passthrough begins at the next cell, which is index 1. The value at index 1 is the number 2, and it is the lowest value we've encountered in this passthrough so far:



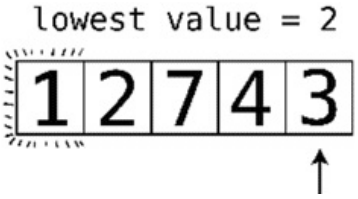

Step #6: We compare the 7 with the lowest value so far. The 2 is less than the 7, so the 2 remains our lowest value:




Step #7: We compare the 4 with the lowest value so far. The 2 is less than the 4, so the 2 remains our lowest value:



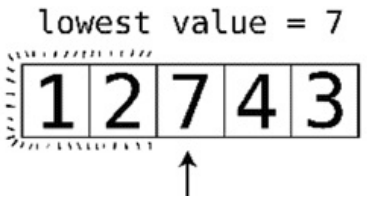

Step #8: We compare the 3 with the lowest value so far. The 2 is less than the 3, so the 2 remains our lowest value:



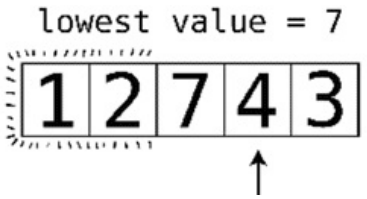

We've reached the end of the array. We don't need to perform any swaps in this passthrough, and we can therefore conclude that the 2 is in its correct spot. This ends our second passthrough, leaving us with:


We now begin Passthrough #3:

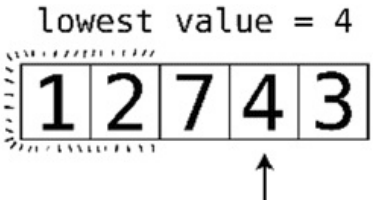
Setup: we begin at index 2, which contains the value 7. The 7 is the lowest value we've encountered so far in this passthrough:



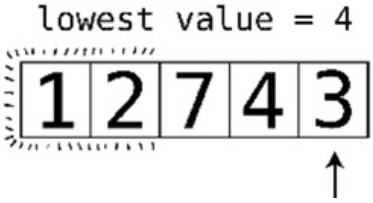

Step #9: We compare the 4 with the 7:



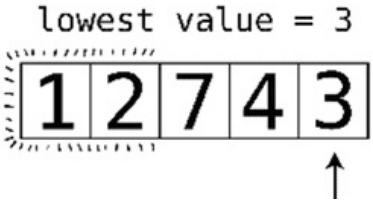

We note that 4 is our new lowest value:



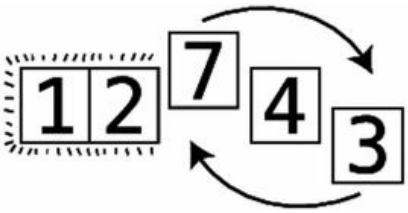

Step #10: We encounter the 3, which is even lower than the 4:




3 is now our new lowest value:



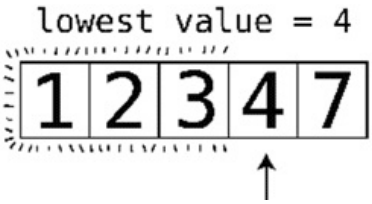

Step #11: We've reached the end of the array, so we swap the 3 with the value that we started our passthrough at, which is the 7:



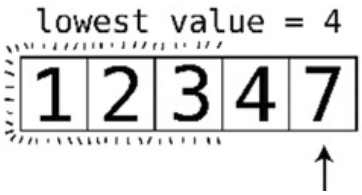

We now know that the 3 is in the correct place within the array:


While you and I can both see that the entire array is correctly sorted at this point, the *computer* does not know this yet, so it must begin a fourth passthrough:

Setup: we begin the passthrough with index 3. The 4 is the lowest value so far:




Step #12: We compare the 7 with the 4:




The 4 remains the lowest value we've encountered in this passthrough so far, so we don't need any swaps and we know it's in the correct place.

Since all the cells besides the last one are correctly sorted, that must mean that the last cell is also in the correct order, and our entire array is properly sorted:

## Selection Sort Implemented

Here's a JavaScript implementation of Selection Sort:

```
function selectionSort(array) {
  for(var i = 0; i < array.length; i++) {
    var lowestNumberIndex = i;
    for(var j = i + 1; j < array.length; j++) {
      if(array[j] < array[lowestNumberIndex]) {
        lowestNumberIndex = j;
      }
    }

    if(lowestNumberIndex != i) {
      var temp = array[i];
      array[i] = array[lowestNumberIndex];
      array[lowestNumberIndex] = temp;
    }
  }
  return array;
}
```

Let's break this down. We'll first present the line of code, followed by its explanation.

```
for(var i = 0; i < array.length; i++) {
```

Here, we have an outer loop that represents each passthrough of Selection Sort. We then begin keeping track of the *index* containing the lowest value we encounter so far with:

```
var lowestNumberIndex = i;
```

which sets `lowestNumberIndex` to be whatever index `i` represents. Note that we're actually tracking the index of the lowest number instead of the actual number itself. This index will be 0 at the beginning of the first passthrough, 1 at the beginning of the second, and so on.

```
for(var j = i + 1; j < array.length; j++) {
```

kicks off an inner `for` loop that starts at `i + 1`.

```
if(array[j] < array[lowestNumberIndex]) {
  lowestNumberIndex = j;
}
```

checks each element of the array that has not yet been sorted and looks for the lowest number. It does this by keeping track of the index of the lowest number it found so far in the `lowestNumberIndex` variable.

By the end of the inner loop, we've determined the index of the lowest number not yet sorted.

```
if(lowestNumberIndex != i) {
  var temp = array[i];
  array[i] = array[lowestNumberIndex];
  array[lowestNumberIndex] = temp;
}
```

We then check to see if this lowest number is already in its correct place (`i`). If not, we swap the lowest number with the number that's in the position that the lowest number should be at.

## The Efficiency of Selection Sort

Selection Sort contains two types of steps: comparisons and swaps. That is, we compare each element with the lowest number we've encountered in each passthrough, and we swap the lowest number into its correct position.

Looking back at our example of an array containing five elements, we had to make a total of ten comparisons. Let's break it down.

| **Passthrough #** | **# of comparisons** |
|---|---|
| 1 | 4 comparisons |
| 2 | 3 comparisons |
| 3 | 2 comparisons |
| 4 | 1 comparison |

So that's a grand total of 4 + 3 + 2 + 1 = 10 comparisons.

To put it more generally, we'd say that for N elements, we make

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

As for *swaps*, however, we only need to make a maximum of one swap per passthrough. This is because in each passthrough, we make either one or zero swaps, depending on whether the lowest number of that passthrough is already in the correct position. Contrast this with Bubble Sort, where in a worst-case scenario—an array in descending order—we have to make a swap for *each and every* comparison.

Here's the side-by-side comparison between Bubble Sort and Selection Sort:

| **N elements** | **Max # of steps in Bubble Sort** | **Max # of steps in Selection Sort** |
|---|---|---|
| 5 | 20 | 14 (10 comparisons + 4 swaps) |
| 10 | 90 | 54 (45 comparisons + 9 swaps) |
| 20 | 380 | 199 (180 comparisons + 19 swaps) |
| 40 | 1560 | 819 (780 comparisons + 39 swaps) |

| | | |
|---|---|---|
| 80 | 6320 | 3239 (3160 comparisons + 79 swaps) |

From this comparison, it’s clear that Selection Sort contains about half the number of steps that Bubble Sort does, indicating that Selection Sort is twice as fast.

## Ignoring Constants

But here's the funny thing: in the world of Big O Notation, Selection Sort and Bubble Sort are described in exactly the same way.

Again, Big O Notation describes how many steps are required relative to the number of data elements. So it would seem at first glance that the number of steps in Selection Sort are roughly *half of whatever $N^2$ is*. It would therefore seem reasonable that we'd describe the efficiency of Selection Sort as being $O(N^2 / 2)$. That is, for N data elements, there are $N^2 / 2$ steps. The following table bears this out:

| **N elements** | **$N^2 / 2$** | **Max # of steps in Selection Sort** |
|---|---|---|
| 5 | $5^2 / 2 = 12.5$ | 14 |
| 10 | $10^2 / 2 = 50$ | 54 |
| 20 | $20^2 / 2 = 200$ | 199 |
| 40 | $40^2 / 2 = 800$ | 819 |
| 80 | $80^2 / 2 = 3200$ | 3239 |

In reality, however, Selection Sort is described in Big O as $O(N^2)$, just like Bubble Sort. This is because of a major rule of Big O that we're now introducing for the first time:

*Big O Notation ignores constants.*

This is simply a mathematical way of saying that Big O Notation never includes regular numbers that aren't an exponent.

In our case, what should technically be $O(N^2 / 2)$ becomes simply $O(N^2)$. Similarly, O(2N) would become O(N), and O(N / 2) would also become O(N). Even O(100N), which is *100 times slower than O(N)*, would also be referred to as O(N).

Offhand, it would seem that this rule would render Big O Notation entirely useless, as you can have two algorithms that are described in the same exact way with Big O, and yet one can be *100 times faster* than the other. And that's exactly what we're seeing here with Selection Sort and Bubble Sort. Both are described in Big O as

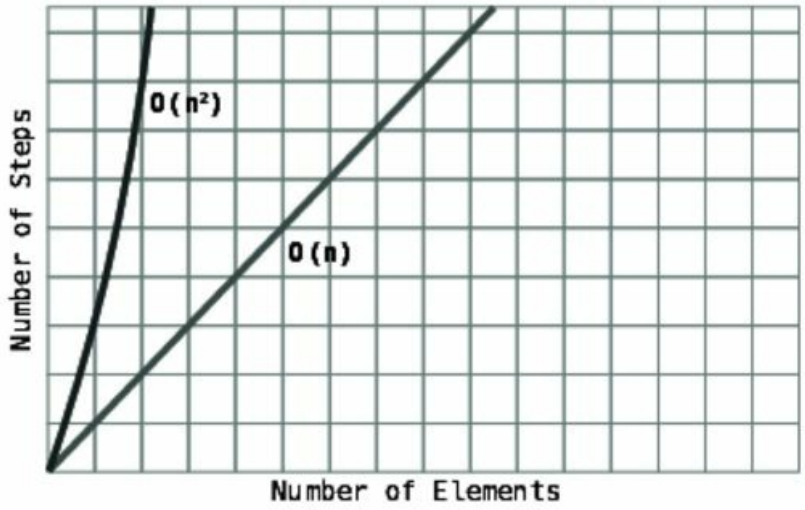
$O(N^2)$, but Selection Sort is actually twice as fast as Bubble Sort. And indeed, if given the choice between these two options, Selection Sort is the better choice.

So, what gives?

## The Role of Big O

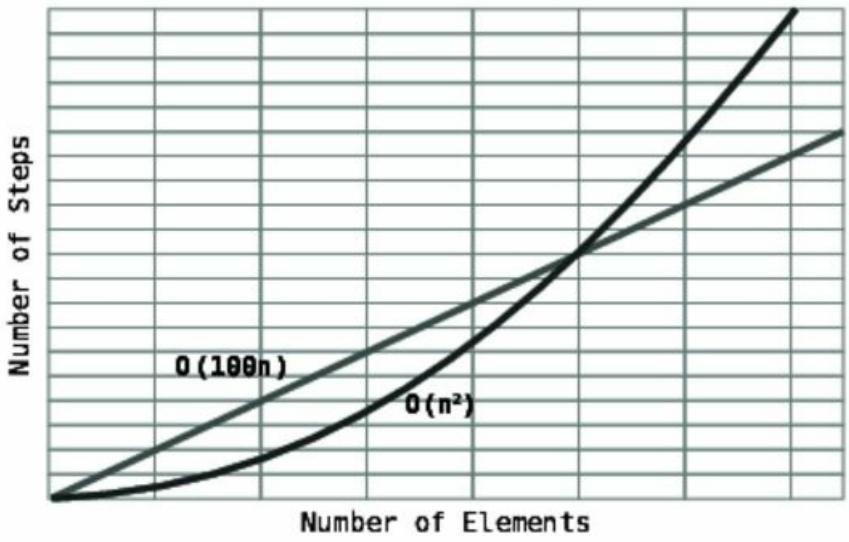
Despite the fact that Big O doesn't distinguish between Bubble Sort and Selection Sort, it is still very important, because it serves as a great way to classify the *long-term growth rate* of algorithms. That is, for *some amount of data*, O(N) will be always be faster than $O(N^2)$. And this is true no matter whether the O(N) is really O(2N) or even O(100N) under the hood. It is a fact that there is *some amount* of data at which even O(100N) will become faster than $O(N^2)$. (We've seen essentially the same concept in Chapter 3, *Oh Yes! Big O Notation* when comparing a 100-step algorithm with O(N), but we'll reiterate it here in our current context.)

Look at the first graph, in which we compare O(N) with $O(N^2)$.




We've seen this graph in the previous chapter. It depicts how O(N) is faster than $O(N^2)$ for *all* amounts of data.

Now take a look at the second graph, where we compare O(100N) with $O(N^2)$.




In this second graph, we see that $O(N^2)$ is faster than O(100N) for certain amounts of data, but after a point, even O(100N) becomes faster and remains faster for all increasing amounts of data from that point onward.

It is for this very reason that Big O ignores constants. The purpose of Big O is that for different classifications, *there will be a point* at which one classification supersedes the other in speed, and will remain faster forever. When that point occurs exactly, however, is not the concern of Big O.

Because of this, there really is no such thing as O(100N)—it is simply written as O(N).

Similarly, with large amounts of data, O(log N) will always be faster than O(N), even if the given O(log N) algorithm is actually O(2 * log N) under the hood.

So Big O is an extremely useful tool, because if two algorithms fall under different classifications of Big O, you'll generally know which algorithm to use since with large amounts of data, one algorithm is guaranteed to be faster than the other at a certain point.

However, the main takeaway of this chapter is that when two algorithms fall under the *same* classification of Big O, it doesn't necessarily mean that both algorithms process at the same speed. After all, Bubble Sort is twice as slow as Selection Sort even though both are $O(N^2)$. So while Big O is perfect for contrasting algorithms

that fall under different classifications of Big O, when two algorithms fall under the *same* classification, further analysis is required to determine which algorithm is faster.

## A Practical Example

Let's say you're tasked with writing a Ruby application that takes an array and creates a new array out of *every other element* from the original array. It might be tempting to use the `each_with_index` method available to arrays to loop through the original array as follows:

```
def every_other(array)
  new_array = []

  array.each_with_index do |element, index|
    new_array << element if index.even?
  end

  return new_array
end
```

In this implementation, we iterate through each element of the original array and only add the element to the new array if the index of that element is an even number.

When analyzing the steps taking place here, we can see that there are really two types of steps. We have one type of step in which we look up each element of the array, and another type of step in which we add elements to the new array.

We perform N array lookups, since we loop through each and every element of the array. We only perform N / 2 insertions, though, since we only insert every other element into the new array. Since we have N lookups, and we have N / 2 insertions, we would say that our algorithm technically has an efficiency of O(N + (N / 2)), which we can also rephrase as O(1.5N). But since Big O Notation throws out the constants, we would say that our algorithm is simply O(N).

While this algorithm does work, we always want to take a step back and see if there's room for any optimization. And in fact, we can.

Instead of iterating through each element of the array and checking whether the index is an even number, we can instead simply look up every other element of the array in the first place:

```ruby
def every_other(array)
  new_array = []
  index = 0

  while index < array.length
    new_array << array[index]
    index += 2
  end

  return new_array
end
```

In this second implementation, we use a **while** loop to skip over each element, rather than check each one. It turns out that for N elements, there are N / 2 lookups and N / 2 insertions into the new array. Like the first implementation, we'd say that the algorithm is O(N).

However, our first implementation truly takes 1.5N steps, while our second implementation only takes N steps, making our second implementation significantly faster. While the first implementation is more idiomatic in the way Ruby programmers write their code, if we're dealing with large amounts of data, it's worth considering using the second implementation to get a significant performance boost.

## Wrapping Up

We now have some very powerful analysis tools at our disposal. We can use Big O to determine broadly how efficient an algorithm is, and we can also compare two algorithms that fall within one classification of Big O.

However, there is another important factor to take into account when comparing the efficiencies of two algorithms. Until now, we've focused on how slow an algorithm is in a worst-case scenario. Now, worst-case scenarios, by definition, don't happen all the time. On average, the scenarios that occur are—well—average-case scenarios. In the next chapter, we'll learn how to take all scenarios into account.

Chapter 6

# Optimizing for Optimistic Scenarios

When evaluating the efficiency of an algorithm, we've focused primarily on how many steps the algorithm would take in a worst-case scenario. The rationale behind this is simple: if you're prepared for the worst, things will turn out okay.

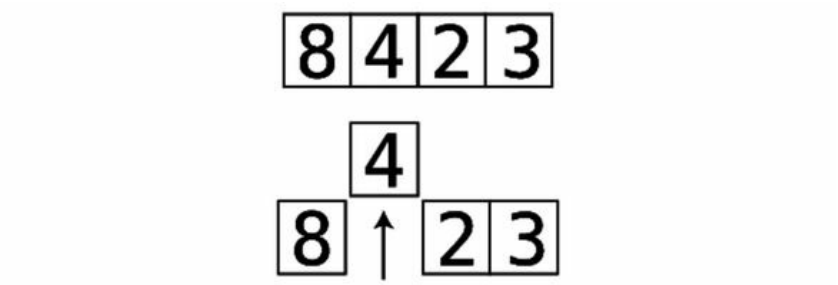
However, we'll discover in this chapter that the worst-case scenario isn't the *only* situation worth considering. Being able to consider *all* scenarios is an important skill that can help you choose the appropriate algorithm for every situation.

# Insertion Sort

We've previously encountered two different sorting algorithms: Bubble Sort and Selection Sort. Both have efficiencies of $O(N^2)$, but Selection Sort is actually twice as fast. Now we'll learn about a third sorting algorithm called *Insertion Sort* that will reveal the power of analyzing scenarios beyond the worst case.

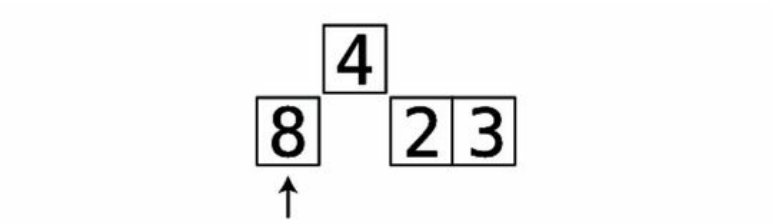
Insertion Sort consists of the following steps:

1. In the first passthrough, we temporarily remove the value at index 1 (the second cell) and store it in a temporary variable. This will leave a gap at that index, since it contains no value:

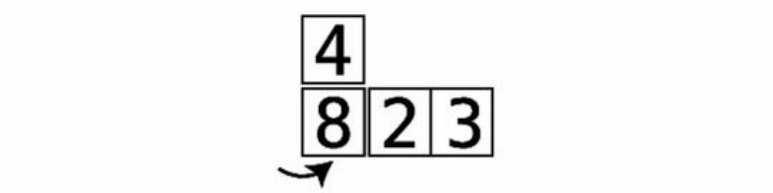



   In subsequent passthroughs, we remove the values at the subsequent indexes.

2. We then begin a shifting phase, where we take each value to the left of the gap, and compare it to the value in the temporary variable:



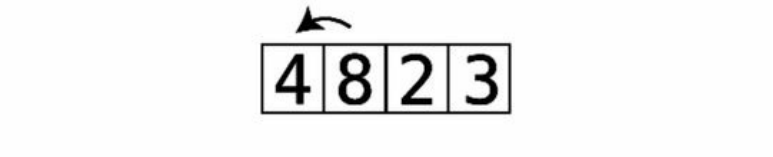

   If the value to the left of the gap is greater than the temporary variable, we shift that value to the right:




   As we shift values to the right, inherently, the gap moves leftwards. As soon as

we encounter a value that is lower than the temporarily removed value, or we reach the left end of the array, this shifting phase is over.

3. We then insert the temporarily removed value into the current gap:

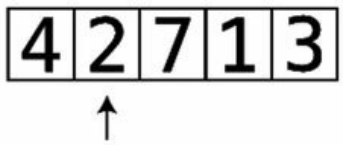



4. We repeat steps 1 through 3 until the array is fully sorted.

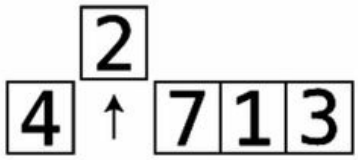
## Insertion Sort in Action

Let's apply Insertion Sort to the array: `[4, 2, 7, 1, 3]`.

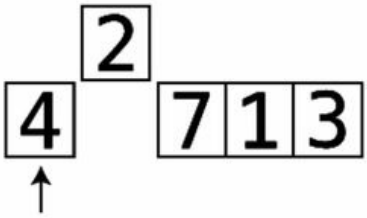
We begin the first passthrough by inspecting the value at index 1. This happens to contain the value 2:



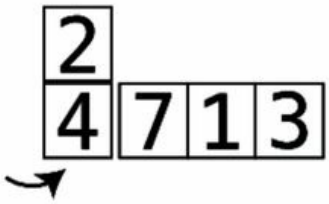

Setup: we temporarily remove the 2, and keep it inside a variable called `temp_value`. We represent this value by shifting it above the rest of the array:




Step #1: We compare the 4 to the `temp_value`, which is 2:

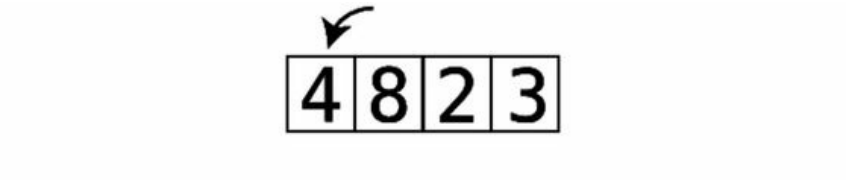



Step #2: Since 4 is greater than 2, we shift the 4 to the right:



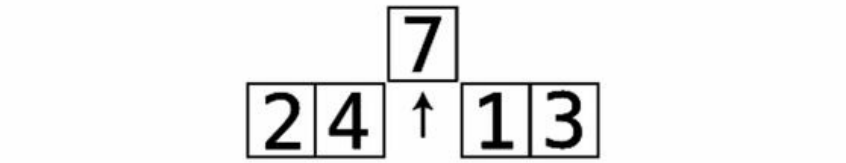

There's nothing left to shift, as the gap is now at the left end of the array.

Step #3: We insert the `temp_value` back into the array, completing our first passthrough:

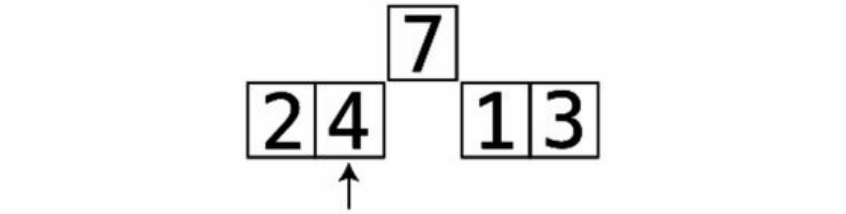



We begin Passthrough #2:

Setup: in our second passthrough, we temporarily remove the value at index 2. We'll store this in `temp_value`. In this case, the `temp_value` is 7:




Step #4: We compare the 4 to the `temp_value`:

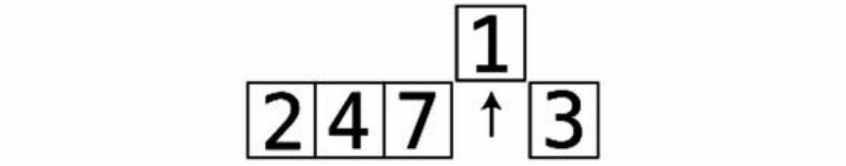



4 is lower, so we won't shift it. Since we reached a value that is less than the `temp_value`, this shifting phase is over.

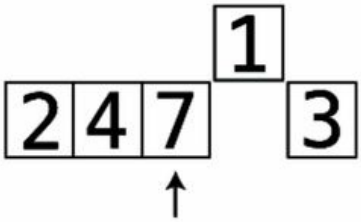
Step #5: We insert the `temp_value` back into the gap, ending the second passthrough:


Passthrough #3:

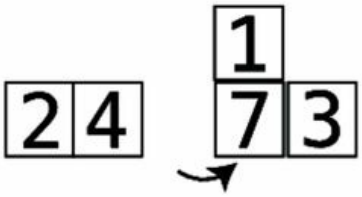
Setup: we temporarily remove the 1, and store it in `temp_value`:



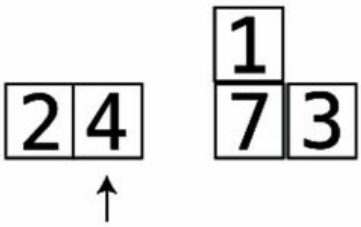

Step #6: We compare the 7 to the `temp_value`:



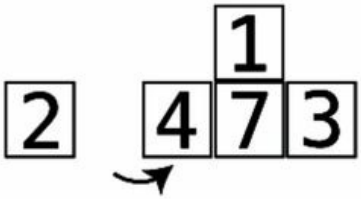

Step #7: 7 is greater than 1, so we shift the 7 to the right:



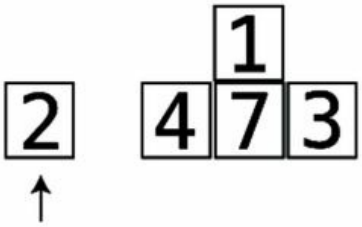

Step #8: We compare the 4 to the `temp_value`:



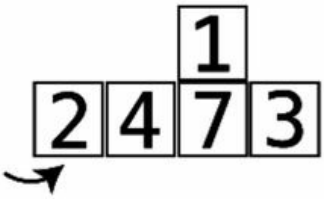

Step #9: 4 is greater than 1, so we shift it as well:



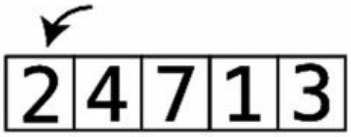

Step #10: We compare the 2 to the `temp_value`:




Step #11: The 2 is greater, so we shift it:

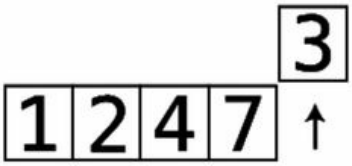


Step #12: The gap has reached the left end of the array, so we insert the **temp_value** into the gap, concluding this passthrough:

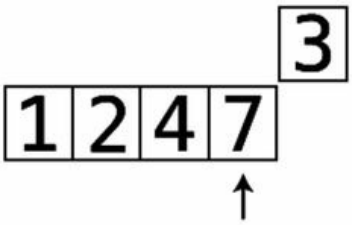



Passthrough #4:

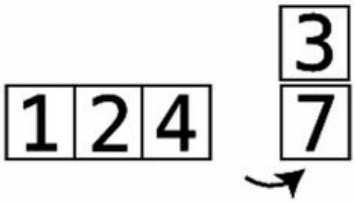
Setup: we temporarily remove the value from index 4, making it our **temp_value**. This is the value 3:



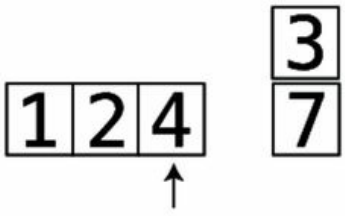

Step #13: We compare the 7 to the **temp_value**:



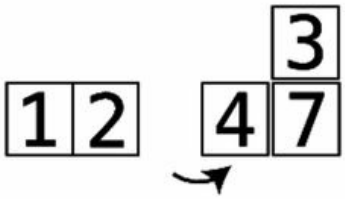

Step #14: The 7 is greater, so we shift the 7 to the right:



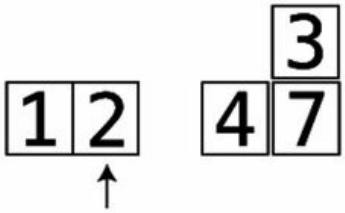

Step #15: We compare the 4 to the **temp_value**:




Step #16: The 4 is greater than the 3, so we shift the 4:




Step #17: We compare the 2 to the `temp_value`. 2 is less than 3, so our shifting phase is complete:




Step #18: We insert the `temp_value` back into the gap:


Our array is now fully sorted:

## Insertion Sort Implemented

Here is a Python implementation of Insertion Sort:

```
def insertion_sort(array):
   for index in range(1, len(array)):

     position = index
     temp_value = array[index]

     while position > 0 and array[position - 1] > temp_value:
         array[position] = array[position - 1]
         position = position - 1

     array[position] = temp_value
```

Let's walk through this step by step. We'll first present the line of code, followed by its explanation.

```
for index in range(1, len(array)):
```

First, we start a loop beginning at index 1 that runs through the entire array. The current index is kept in the variable `index`.

```
position = index
temp_value = array[index]
```

Next, we mark a `position` at whatever `index` currently is. We also assign the value at that index to the variable `temp_value`.

```
while position > 0 and array[position - 1] > temp_value:
    array[position] = array[position - 1]
    position = position - 1
```

We then begin an inner `while` loop. We check whether the value to the left of `position` is greater than the `temp_value`. If it is, we then use `array[position] = array[position - 1]` to shift that left value one cell to the right, and then decrement `position` by one. We then check whether the value to the left of the new `position` is greater than `temp_value`, and keep repeating this process until we find a value that is

less than the **temp_value**.

```
array[position] = temp_value
```

Finally, we drop the **temp_value** into the gap within the array.

## The Efficiency of Insertion Sort

There are four types of steps that occur in Insertion Sort: removals, comparisons, shifts, and insertions. To analyze the efficiency of Insertion Sort, we need to tally up each of these steps.

First, let's dig into *comparisons*. A comparison takes place each time we compare a value to the left of the gap with the `temp_value`.

In a worst-case scenario, where the array is sorted in reverse order, we have to compare every number to the left of `temp_value` with `temp_value` in each passthrough. This is because each value to the left of `temp_value` will always be greater than `temp_value`, so the passthrough will only end when the gap reaches the left end of the array.

During the first passthrough, in which `temp_value` is the value at index 1, a maximum of one comparison is made, since there is only one value to the left of the `temp_value`. On the second passthrough, a maximum of two comparisons are made, and so on and so forth. On the final passthrough, we need to compare the `temp_value` with every single value in the array besides `temp_value` itself. In other words, if there are N elements in the array, a maximum of N - 1 comparisons are made in the final passthrough.

We can, therefore, formulate the total number of comparisons as:

1 + 2 + 3 + … + N - 1 comparisons.

In our example of an array containing five elements, that's a maximum of:

1 + 2 + 3 + 4 = 10 comparisons.

For an array containing ten elements, there would be:

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 comparisons.

(For an array containing twenty elements, there would be a total of 190 comparisons,

and so on.)

When examining this pattern, it emerges that for an array containing N elements, there are *approximately* $N^2 / 2$ comparisons. ($10^2 / 2$ is 50, and $20^2 / 2$ is 200.)

Let's continue analyzing the other types of steps.

*Shifts* occur each time we move a value one cell to the right. When an array is sorted in reverse order, there will be as many shifts as there are comparisons, since every comparison will force us to shift a value to the right.

Let's add up comparisons and shifts for a worst-case scenario:

$N^2 / 2$ comparisons

+ $N^2 / 2$ shifts

---

$N^2$ steps

Removing and inserting the `temp_value` from the array happen once per passthrough. Since there are always N - 1 passthroughs, we can conclude that there are N - 1 removals and N - 1 insertions.

So now we've got:

$N^2$ comparisons & shifts combined

N - 1 removals

+ N - 1 insertions

---

$N^2 + 2N - 2$ steps

We've already learned one major rule of Big O—that Big O ignores constants. With this rule in mind, we'd—at first glance—simplify this to $O(N^2 + N)$.

However, there is another major rule of Big O that we'll reveal now:

*Big O Notation only takes into account the highest order of N.*

That is, if we have some algorithm that takes $N^4 + N^3 + N^2 + N$ steps, we only consider $N^4$ to be significant—and just call it $O(N^4)$. Why is this?

Look at the following table:

| **N** | **$N^2$** | **$N^3$** | **$N^4$** |
|---|---|---|---|
| 2 | 4 | 8 | 16 |
| 5 | 25 | 125 | 625 |
| 10 | 100 | 1,000 | 10,000 |
| 100 | 10,000 | 1,000,000 | 100,000,000 |
| 1,000 | 1,000,000 | 1,000,000,000 | 1,000,000,000,000 |

As N increases, $N^4$ becomes so much more significant than any other order of N. When N is 1,000, $N^4$ is 1,000 times greater than $N^3$. Because of this, we only consider the greatest order of N.

In our example, then, $O(N^2 + N)$ simply becomes $O(N^2)$.

It emerges that in a worst-case scenario, Insertion Sort has the same time complexity as Bubble Sort and Selection Sort. They're all $O(N^2)$.

We noted in the previous chapter that although Bubble Sort and Selection Sort are both $O(N^2)$, Selection Sort is faster since Selection Sort has $N^2 / 2$ steps compared with Bubble Sort's $N^2$ steps. At first glance, then, we'd say that Insertion Sort is as slow as Bubble Sort, since it too has $N^2$ steps. (It's really $N^2 + 2N - 2$ steps.)

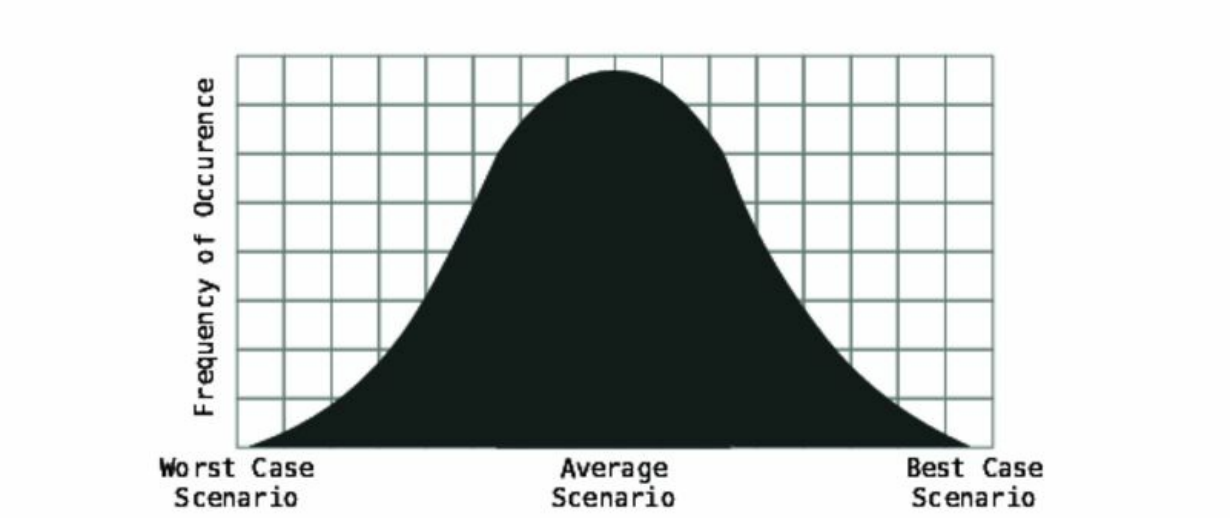
If we'd stop the book here, you'd walk away thinking that Selection Sort is the best choice out of the three, since it's twice as fast as either Bubble Sort or Insertion Sort. But it's actually not that simple.

## The Average Case

Indeed, in a worst-case scenario, Selection Sort *is* faster than Insertion Sort. However, it is critical that we also take into account the *average-case scenario*.

Why?

By definition, the cases that occur most frequently are average scenarios. The worst- and best-case scenarios happen only rarely. Let's look at this simple bell curve:




Best- and worst-case scenarios happen relatively infrequently. In the real world, however, average scenarios are what occur most of the time.

And this makes a lot of sense. Think of a randomly sorted array. What are the odds that the values will occur in perfect ascending or descending order? It's much more likely that the values will be all over the place.

Let's examine Insertion Sort in context of all scenarios.

We've looked at how Insertion Sort performs in a worst-case scenario—where the array sorted in descending order. In the worst case, we pointed out that in each passthrough, we compare and shift every value that we encounter. (We calculated this to be a total of $N^2$ comparisons and shifts.)

In the best-case scenario, where the data is already sorted in ascending order, we end

up making just one comparison per passthrough, and not a single shift, since each value is already in its correct place.

Where data is randomly sorted, however, we'll have passthroughs in which we compare and shift all of the data, some of the data, or possibly none of data. If you'll look at our preceding walkthrough example, you'll notice that in Passthroughs #1 and #3, we compare and shift all the data we encounter. In Passthrough #4, we compare and shift some of it, and in Passthrough #2, we make just one comparison and shift no data at all.

While in the worst-case scenario, we compare and shift *all* the data, and in the best-case scenario, we shift *none* of the data (and just make one comparison per passthrough), for the average scenario, we can say that in the aggregate, we probably compare and shift about *half* of the data.

So if Insertion Sort takes $N^2$ steps for the worst-case scenario, we'd say that it takes about $N^2 / 2$ steps for the average scenario. (In terms of Big O, however, both scenarios are $O(N^2)$.)

Let's dive into some specific examples.

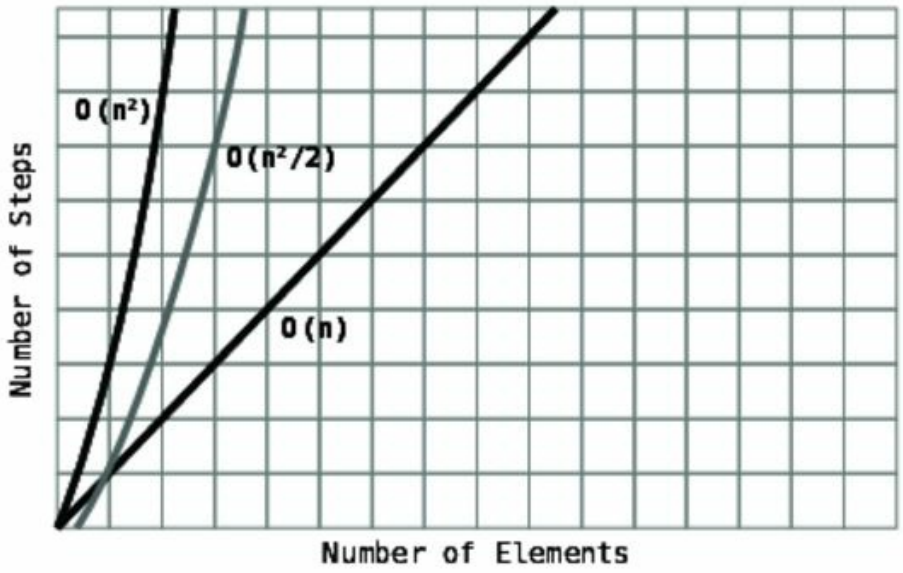
The array `[1, 2, 3, 4]` is already presorted, which is the best case. The worst case for the same data would be `[4, 3, 2, 1]`, and an example of an average case might be `[1, 3, 4, 2]`.

In the worst case, there are six comparisons and six shifts, for a total of twelve steps. In the average case, there are four comparisons and two shifts, for a total of six steps. In the best case, there are three comparisons and zero shifts.

We can now see that the performance of Insertion Sort *varies greatly* based on the scenario. In the worst-case scenario, Insertion Sort takes $N^2$ steps. In an average scenario, it takes $N^2 / 2$ steps. And in the best-case scenario, it takes about N steps.

This variance is because some passthroughs compare all the data to the left of the `temp_value`, while other passthroughs end early, due to encountering a value that is less than the `temp_value`.

You can see these three types of performance in this graph:




Compare this with Selection Sort. Selection Sort takes $N^2 / 2$ steps in *all* cases, from worst to average to best-case scenarios. This is because Selection Sort doesn't have any mechanism for ending a passthrough early at any point. Each passthrough compares every value to the right of the chosen index no matter what.

So which is better: Selection Sort or Insertion Sort? The answer is: well, it depends. In an average case—where an array is randomly sorted—they perform similarly. If you have reason to assume that you'll be dealing with data that is *mostly* sorted, Insertion Sort will be a better choice. If you have reason to assume that you'll be dealing with data that is mostly sorted in reverse order, Selection Sort will be faster. If you have no idea what the data will be like, that's essentially an average case, and both will be equal.

## Wrapping Up

Having the ability to discern between best-, average-, and worst-case scenarios is a key skill in choosing the best algorithm for your needs, as well as taking existing algorithms and optimizing them further to make them significantly faster. Remember, while it's good to be prepared for the worst case, average cases are what happen most of the time.

In the next chapter, we're going to learn about a new data structure that is similar to the array, but has nuances that can allow it to be more performant in certain circumstances. Just as you now have the ability to choose between two competing algorithms for a given use case, you'll also need the ability to choose between two competing data structures, as one may have better performance than the other.