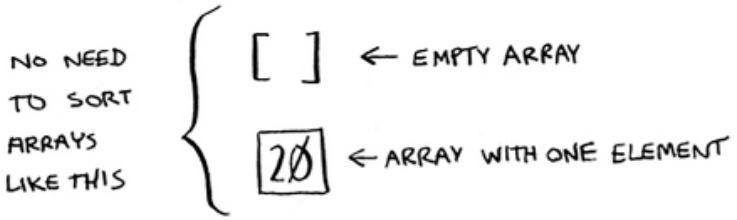
Quicksort

Quicksort is a sorting algorithm. It's much faster than selection sort and is frequently used in real life. Quicksort also uses divide and conquer.

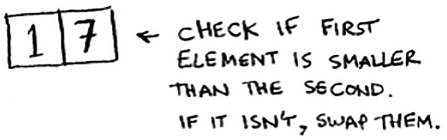
Let's use quicksort to sort an array. What's the simplest array that a sorting algorithm can handle (remember my tip from the previous section)? Well, some arrays don't need to be sorted at all.



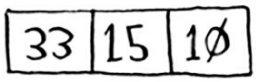
Empty arrays and arrays with just one element will be the base case. You can just return those arrays as is—there's nothing to sort:

```
def quicksort(array):
  if len(array) < 2:
    return array
```

Let's look at bigger arrays. An array with two elements is pretty easy to sort, too.



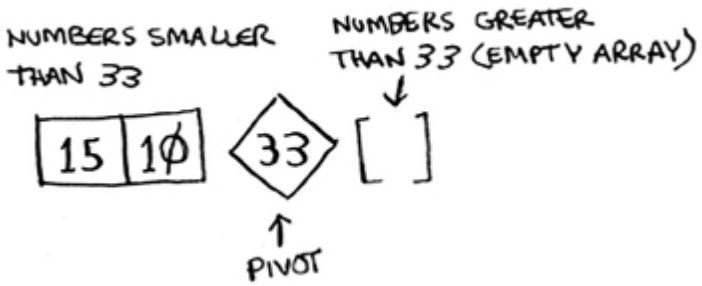
What about an array of three elements?



Remember, you're using D&C. So you want to break down this array until you're at the base case. Here's how quicksort works. First, pick an element from the array. This element is called the *pivot*.

We'll talk about how to pick a good pivot later. For now, let's say the first item in the array is the pivot.

Now find the elements smaller than the pivot and the elements larger than the pivot.



This is called *partitioning*. Now you have

- A sub-array of all the numbers less than the pivot
- The pivot
- A sub-array of all the numbers greater than the pivot

The two sub-arrays aren't sorted. They're just partitioned. But if they *were* sorted, then sorting the whole array would be pretty easy.

If the sub-arrays are sorted, then you can combine the whole thing like this—`left array + pivot + right array`—and you get a sorted array. In this case, it's `[10, 15] + [33] + [] = [10, 15, 33]`, which is a sorted array.

How do you sort the sub-arrays? Well, the quicksort base case already knows how to sort empty arrays (the right sub-array), and it can recursively sort arrays of two elements (the left sub-array). So if you call quicksort on the two sub-arrays and then combine the results, you get a sorted array!

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33]         #A
```

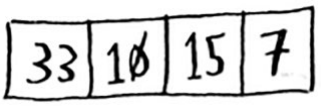
#A A sorted array

This will work with any pivot. Suppose you choose 15 as the pivot instead.

Both sub-arrays have only one element, and you know how to sort those. So now you know how to sort an array of three elements. Here are the steps:

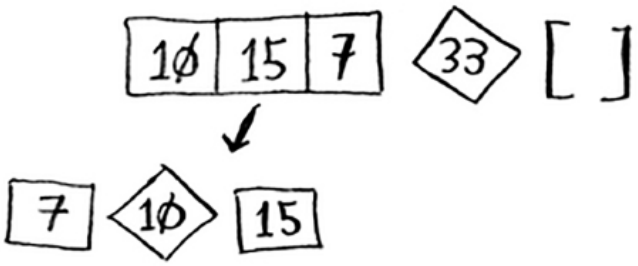
1. Pick a pivot.
2. Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
3. Call quicksort recursively on the two sub-arrays.

What about an array of four elements?

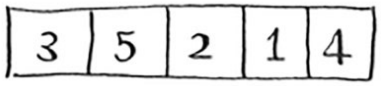


Suppose you choose 33 as the pivot again.

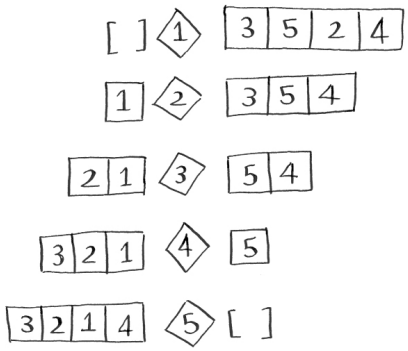
The array on the left has three elements. You already know how to sort an array of three elements: call quicksort on it recursively.



So you can sort an array of four elements. And if you can sort an array of four elements, you can sort an array of five elements. Why is that? Suppose you have this array of five elements.

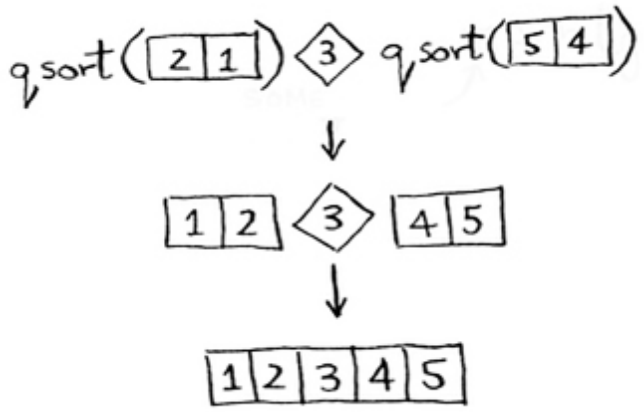


Here are all the ways you can partition this array, depending on what pivot you choose.

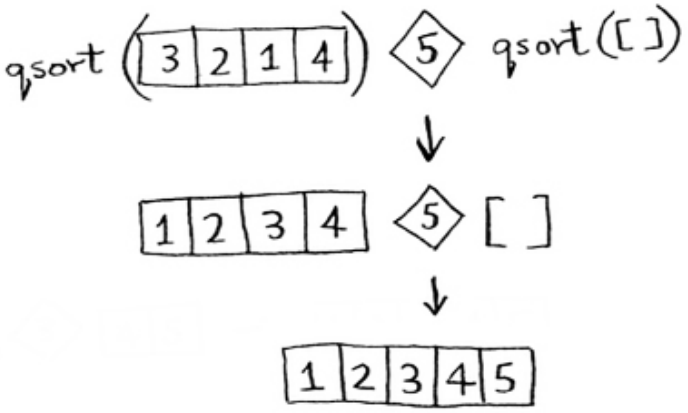


Notice that all of these sub-arrays have somewhere between 0 and 4 elements. And you already know how to sort an array of 0 to 4 elements using quicksort! So no matter what pivot you pick, you can call quicksort recursively on the two sub-arrays.

For example, suppose you pick 3 as the pivot. You call quicksort on the sub-arrays.



The sub-arrays get sorted, and then you combine the whole thing to get a sorted array. This works even if you choose 5 as the pivot.



This works with any element as the pivot. So you can sort an array of five elements. Using the same logic, you can sort an array of six elements, and so on.

Inductive proofs

You just got a sneak peak into *inductive proofs!* Inductive proofs are one way to prove that your algorithm works. Each inductive proof has two steps: the base case and the inductive case. Sound familiar? For example, suppose I want to prove that I can climb to the top of a ladder. In the inductive case, if my legs are on a rung, I can put my legs on the next rung. So if I'm on rung 2, I can climb to rung 3. That's the inductive case. For the base case, I'll say that my legs are on rung 1. Therefore, I can climb the entire ladder, going up one rung at a time.

You use similar reasoning for quicksort. In the base case, I showed that the algorithm works for the base case: arrays of size 0 and 1. In the inductive case, I showed that if quicksort works for an array of size 1, it will work for an array of size 2. And if it works for arrays of size 2, it will work for arrays of size 3, and so on. Then I can say that quicksort will work for all arrays of any size. I won't go deeper into inductive proofs here, but they're fun and go hand-in-hand with D&C.

Here's the code for quicksort:

```
def quicksort(array):
  if len(array) < 2:
    return array          #A
  else:
    pivot = array[0]        #B
    less = [i for i in array[1:] if i <= pivot]    #C

    greater = [i for i in array[1:] if i > pivot]   #D
    return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10, 5, 2, 3]))
```

#A Base case: arrays with 0 or 1 element are already "sorted."

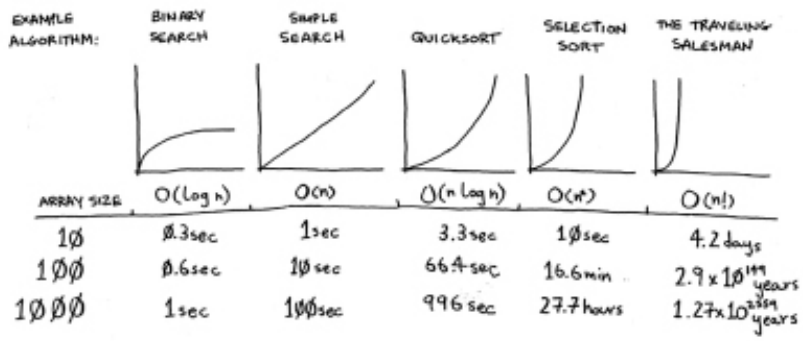
#B Recursive case

#C Sub-array of all the elements less than the pivot

#D Sub-array of all the elements greater than the pivot

Big O notation revisited

Quicksort is unique because its speed depends on the pivot you choose. Before I talk about quicksort, let's look at the most common Big O run times again.



Estimates based on a slow computer that performs 10 operations per second

The example times in this chart are estimates if you perform 10 operations per second. These graphs aren't precise—they're just there to give you a sense of how different these run times are. In reality, your computer can do way more than 10 operations per second.

Each run time also has an example algorithm attached. Check out selection sort, which you learned in chapter 2. It's O(n^2). That's a pretty slow algorithm.

There's another sorting algorithm called *merge sort*, which is O(n log n). Much faster! Quicksort is a tricky case. In the worst case, quicksort takes O(n^2) time.

It's as slow as selection sort! But that's the worst case. In the average case, quicksort takes O(n log n) time. So you might be wondering:

What do *worst case* and *average case* mean here?

If quicksort is O(n log n) on average, but merge sort is O(n log n) always, why not use merge sort? Isn't it faster?

Merge sort vs. quicksort

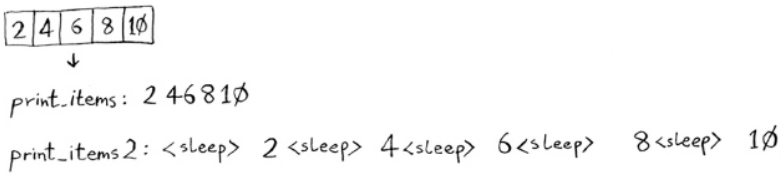
Suppose you have this simple function to print every item in a list:

```
def print_items(myList):
  for item in myList:
    print(item)
```

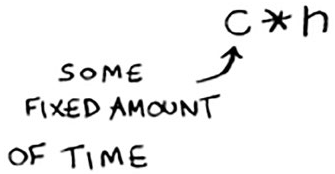
This function goes through every item in the list and prints it out. Because it loops over the whole list once, this function runs in O(n) time. Now, suppose you change this function so it sleeps for 1 second before it prints out an item:

```
from time import sleep
def print_items2(myList):
  for item in myList:
    sleep(1)
    print(item)
```

Before it prints out an item, it will pause for 1 second. Suppose you print a list of five items using both functions.

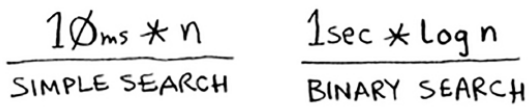


Both functions loop through the list once, so they're both O(*n*) time. Which one do you think will be faster in practice? I think `print_items` will be much faster because it doesn't pause for 1 second before printing an item. So even though both functions are the same speed in Big O notation, `print_items` is faster in practice. When you write Big O notation like O(*n*), it really means this.

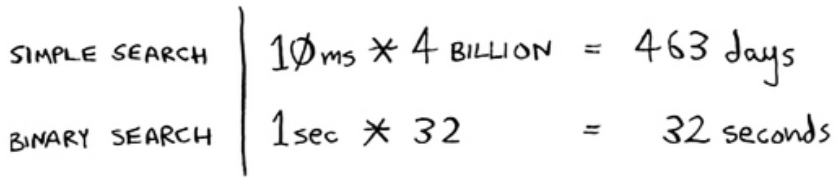


`c` is some fixed amount of time that your algorithm takes. It's called the *constant*. For example, it might be `10 milliseconds * n` for `print_items` versus `1 second * n` for `print_items2`.

You usually ignore that constant, because if two algorithms have different Big O times, the constant doesn't matter. Take binary search and simple search, for example. Suppose both algorithms had these constants.



You might say, "Wow! Simple search has a constant of 10 milliseconds, but binary search has a constant of 1 second. Simple search is way faster!" Now suppose you're searching a list of 4 billion elements. Here are the times.



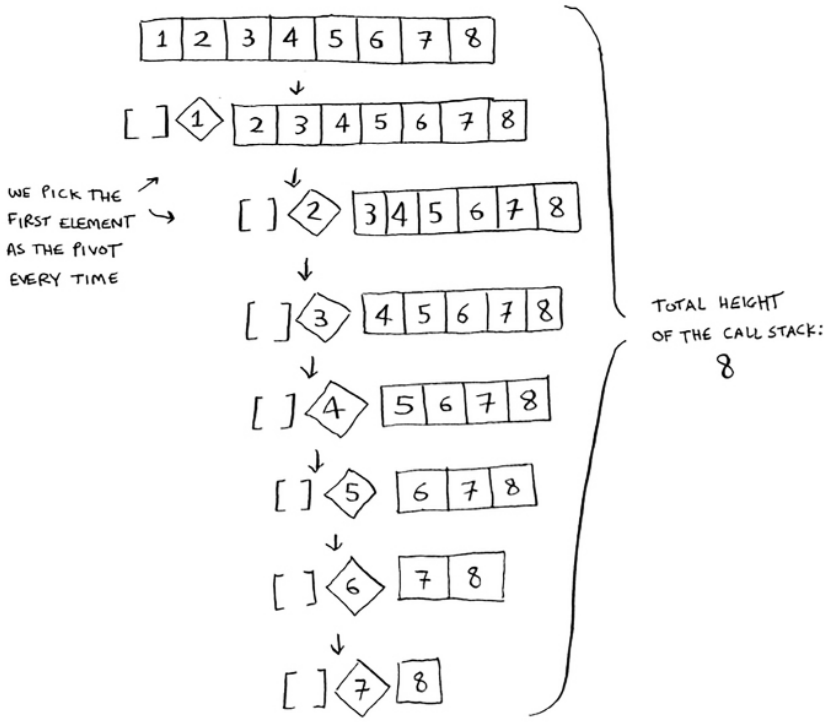
As you can see, binary search is still way faster. That constant didn't make a difference at all.

But sometimes the constant *can* make a difference. Quicksort versus merge sort is one example. Often, the way Quicksort and merge sort are implemented, if they're both O(*n* log *n*) time, quicksort is faster. And quicksort is faster in practice because it hits the average case way more often than the worst case.

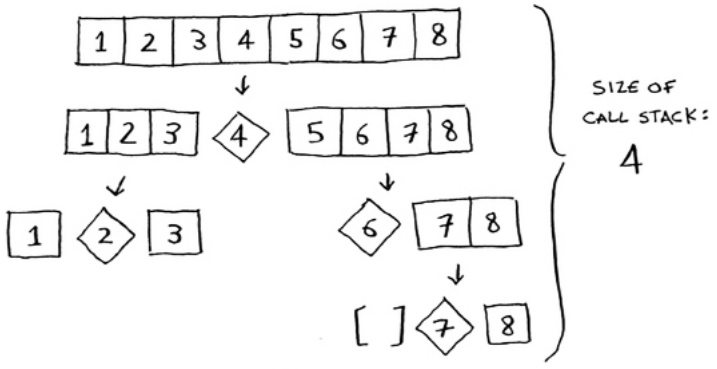
So now you're wondering: what's the average case versus the worst case?

Average case vs. worst case

The performance of quicksort heavily depends on the pivot you choose. Suppose you always choose the first element as the pivot. And you call quicksort with an array that is *already sorted*. Quicksort doesn't check to see whether the input array is already sorted. So it will still try to sort it.



Notice how you're not splitting the array into two halves. Instead, one of the sub-arrays is always empty. So the call stack is really long. Now instead, suppose you always picked the middle element as the pivot. Look at the call stack now.

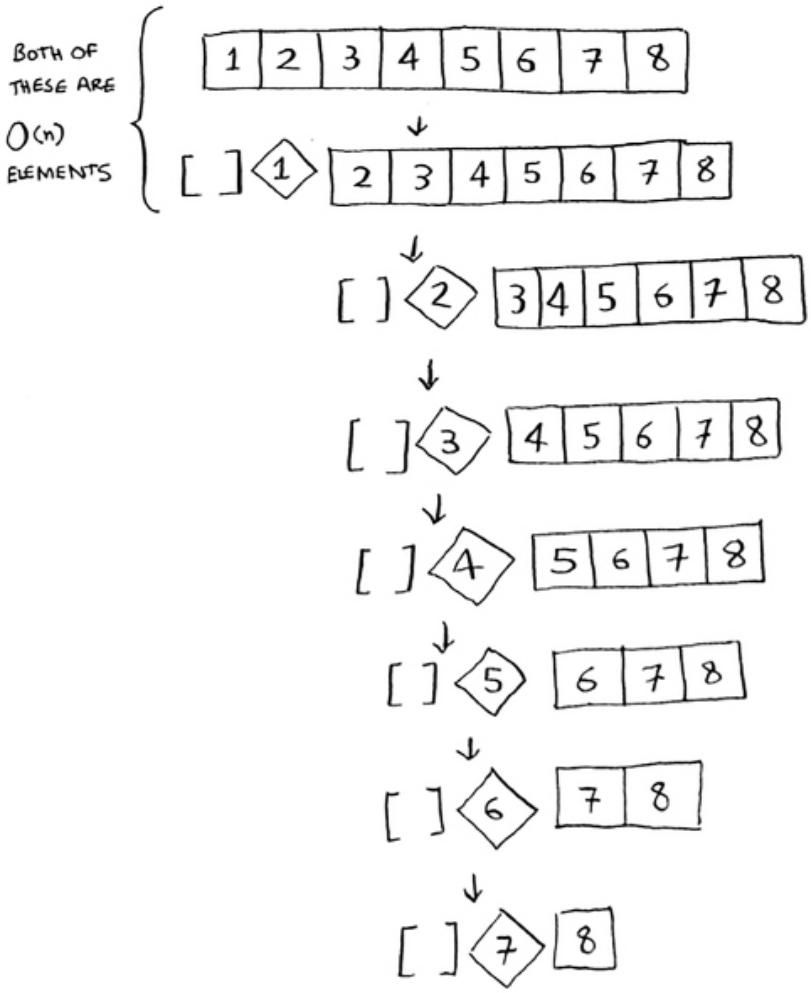


It's so short! Because you divide the array in half every time,

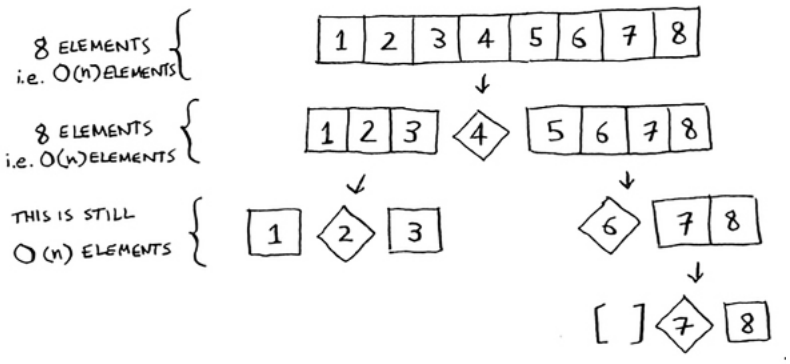
you don’t need to make as many recursive calls. You hit the base case sooner, and the call stack is much shorter.

The first example you saw is the worst-case scenario, and the second example is the best-case scenario. In the worst case, the stack size is O(*n*). In the best case, the stack size is O(log *n*). You can get the best case consistently, as long as you always choose a random element as the pivot. Read on to find out why.

Now look at the first level in the stack. You pick one element as the pivot, and the rest of the elements are divided into sub-arrays. You touch all eight elements in the array. So this first operation takes O(*n*) time. You touched all eight elements on this level of the call stack. But actually, you touch O(*n*) elements on every level of the call stack.



Even if you partition the array differently, you're still touching O(*n*) elements every time.



In this example, there are O(log *n*) levels (the technical way to say that is, "The height of the call stack is O(log *n*)"). And each level takes O(*n*) time. The entire algorithm will take O(*n*) * O(log *n*) = O(*n* log *n*) time. This is the best-case scenario.

In the worst case, there are O(*n*) levels, so the algorithm will take O(*n*) * O(*n*) = O(n^2) time.

Well, guess what? I'm here to tell you that the best case is also the average case. *If you always choose a random element in the array as the pivot*, quicksort will complete in O(*n* log *n*) time on average. Quicksort is one of the fastest sorting algorithms out there, and it's a very good example of D&C.

Exercises

How long would each of these operations take in Big O notation?

1. 4.5 Printing the value of each element in an array.
2. 4.6 Doubling the value of each element in an array.
3. 4.7 Doubling the value of just the first element in an array.
4. 4.8 Creating a multiplication table with all the elements in the array. So if your array is [2, 3, 7, 8, 10], you first multiply every element by 2, then multiply every