# Computational Complexity

**Gabe Johnson**

**Applied Data Structures & Algorithms**

**University of Colorado-Boulder**

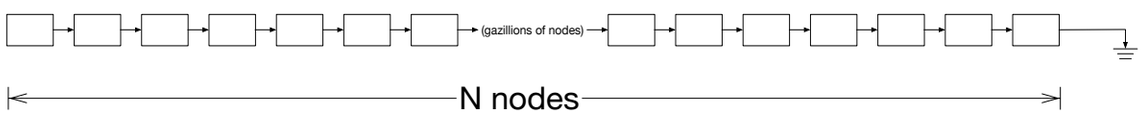
This entire data structures and algorithms course is really a vehicle for three big ideas. You've already seen two of them: pointers and recursion. This sequence introduce the last of those, and we'll spend the rest of the course after this exploring different data structures and algorithms through the lens of these three concepts.

Episode 1

# Complexity & Big Oh

The new topic is of course, computational complexity. This is a way to measure the boundaries on how much some resource is used as a function of the input size.

# Search Unordered List




A simple example is: say we're looking through a gigantic linked list for some specific value. How many nodes will we have to look through to determine if the value is there? We've seen this movie before. Assuming the list isn't sorted or arranged in any clever way, we'll have to access potentially every node.

<next build>

So if... instead of saying 'gigantic list', maybe I'll be a little more mathy about it, and say the list has N items in it, we'd say that searching this unordered list could involve N accesses as we scroll through the list.

**O(n)** Pronounced “big oh of n”, or just “oh of n”

| | |
|---|---|
| **O(n!)** | factorial |
| **O($n^2$)** | polynomial |
| **O(n)** | linear |
| **O(log n)** | logarithmic |
| **O(1)** | constant |

Formally, we'd say this operation is big oh of N. Usually we shorten this to just 'oh of N'.

When we talk about computational complexity, it is always in terms of some input size N. You can have lots of different complexities, like O(log n), O(n!), and even O(1) which means the runtime is constant and doesn't depend on input length.

If you were to look at some algorithm and break it down into great detail, you might actually determine that the worst case scenario is something like...

```
int nifty_function(data_structure input) {
  // nifty implementation
}
```

$4n^3 + 10n^2 + 100000$

$4n^3 + 10n^2 + 100000$

$4n^3 + 10n^2 + 100000$

nifty_function is…

$O(n^3)$

4n^3 + 10n^2 + 100000

From a practical perspective, we might actually care about each of those terms. And I'll actually return to that topic in a few episodes.

But from a compulational complexity, big-oh point of view, we only care about the dominant term, the one with the biggest effect as N becomes huge.

<next build>

So we don't care about the constant 100000, nor do we care about the 10n^2.

<next build>

And even the 4 part of 4n^3 doesn't matter from a notational perspective. We'd just say this algorithm, whatever it is, is

<next build>

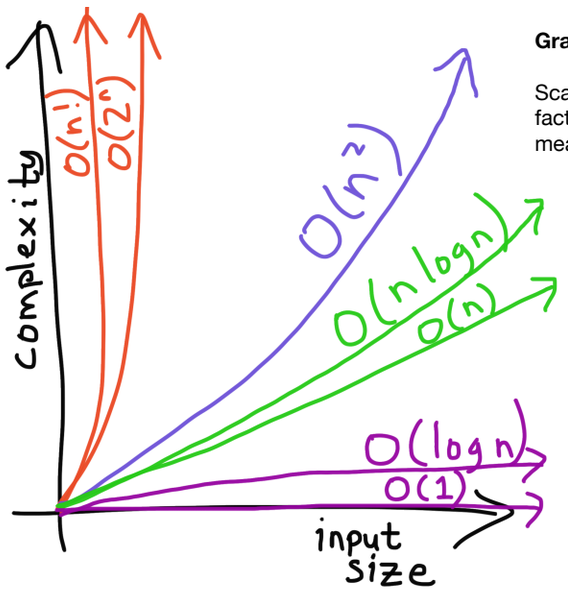
O(n^3).

# Intuition of Complexity

**O($2^n$)** takes longer than **O($n^2$)** takes longer than **O(log n)** takes longer than **O(1)**

where ‘takes longer’ almost always means ‘is worse than’

Eventually you'll take a formal algorithms course, and you'll delve deeply into algorithm analysis. For this class, all I want to impress on you is that there's this thing called computational complexity, and you should develop an intuition for it. Both in understanding why, say, an algorithm of O(n^2) generally takes longer to run than an O(log n) one.




**Graph shows common complexities and how they relate.**

Scale is kinda warped because it is not possible to show factorial and logarithmic on the same graph while conveying meaningful information.

**Factorial and exponential (n! and $2^n$):** becomes absurdly large even with small N.

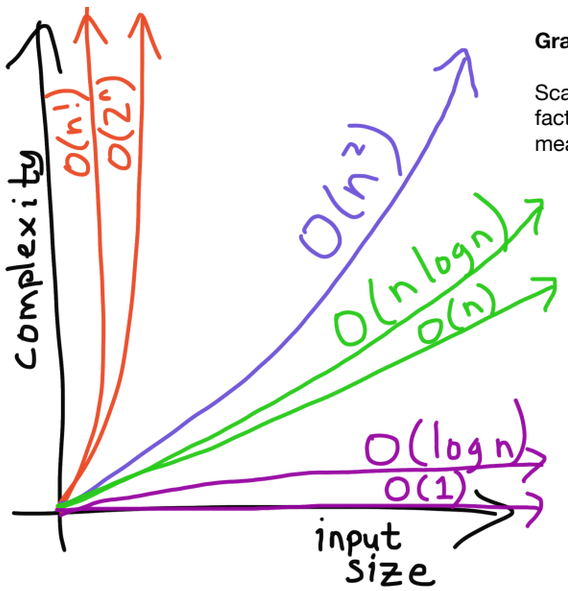
**Polynomial ($n^2$, $n^3$, etc)** grow faster than n, but (depending on use case) might not be terrible.

**Linear and logarithmic (n log n, n, log n)** are where most 'good' algorithms are. *obviously depends on use case.*

**Constant time, O(1),** doesn't depend on input size.

Since I love diagrams, I drew one for you.

This shows the relationship between input size on the X axis with how various complexities grow. This diagram intentionally looks like a six year old drew it, because I don't want you to focus on the fact that the scale is all warped. The point here is that there are huge differences between the runtime complexity of different algorithms. And you can say that one family, like O(n), is categorically more complex than say, O(log n).




**Graph shows common complexities and how they relate.**

Scale is kinda warped because it is not possible to show factorial and logarithmic on the same graph while conveying meaningful information.

**Factorial and exponential (n! and $2^n$):** becomes absurdly large even with small N.

**Polynomial ($n^2$, $n^3$, etc)** grow faster than n, but (depending on use case) might not be terrible.

**Linear and logarithmic (n log n, n, log n)** are where most 'good' algorithms are. *obviously depends on use case.*

**Constant time, O(1),** doesn't depend on input size.

I've said 'family' with respect to complexity functions. This is because once we've analyzed an algorithm and determined that it is O(something), we can group it together with any other O(something) algorithm. This especially makes sense when comparing similar approaches to the same problem.

Like, if you have an algorithm that can search for a value in a data structure in O(log n), and I have an algorithm that does the same thing but it is O(n), yours is categorically faster.

## Big Oh is bounds on resource usage

Most often, big oh refers to runtime. **Time**

Sometimes, big oh refers to memory usage. **Space**

You can use big oh to refer to any resource usage. Time / Space are most common.

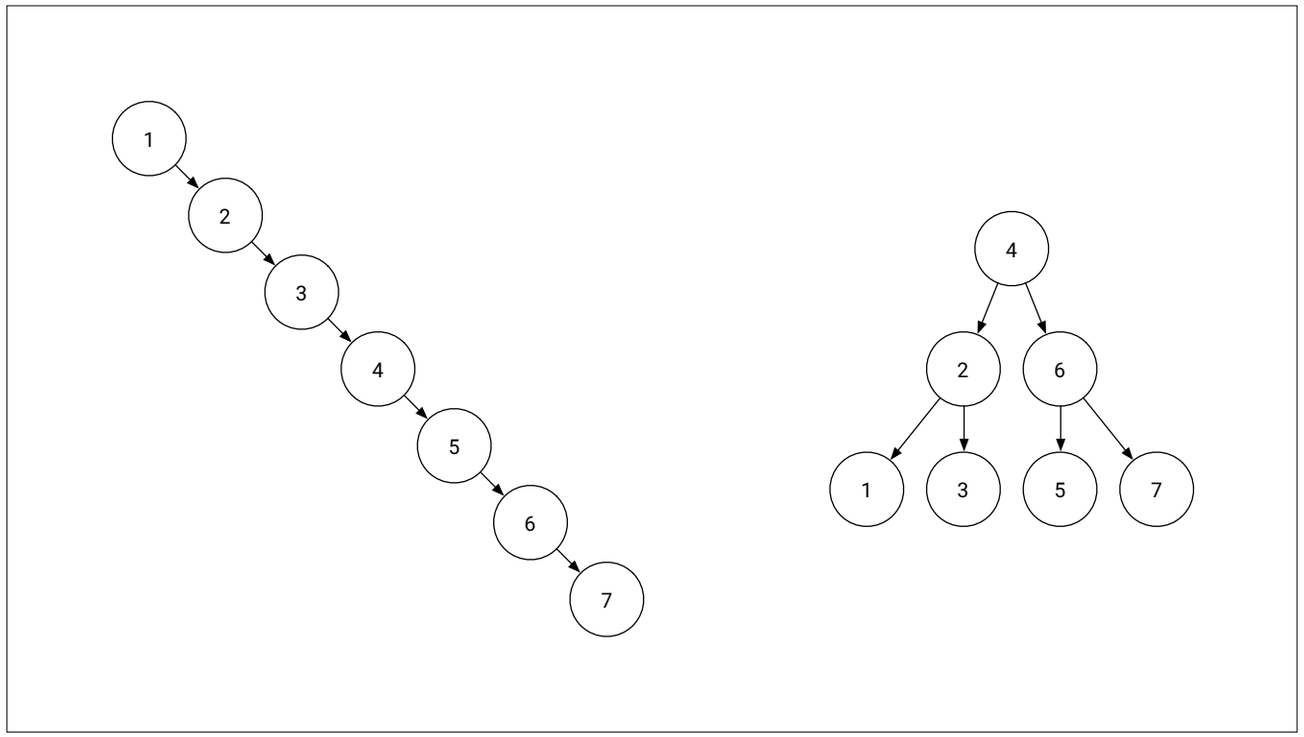
One more important thing to know about computational complexity:

We usually talk about complexity in terms of the **time** an algorithm takes as a function of input size. But we really can use complexity to measure any resource the algorithm uses. Time is the most common, but space is another. And by that we mean memory, disk usage, or whatever other data that has to be stored somewhere.

Episode 2

# Complexity of BST Operations

This is a short episode that applies the complexity concept from the last episode to binary search trees.



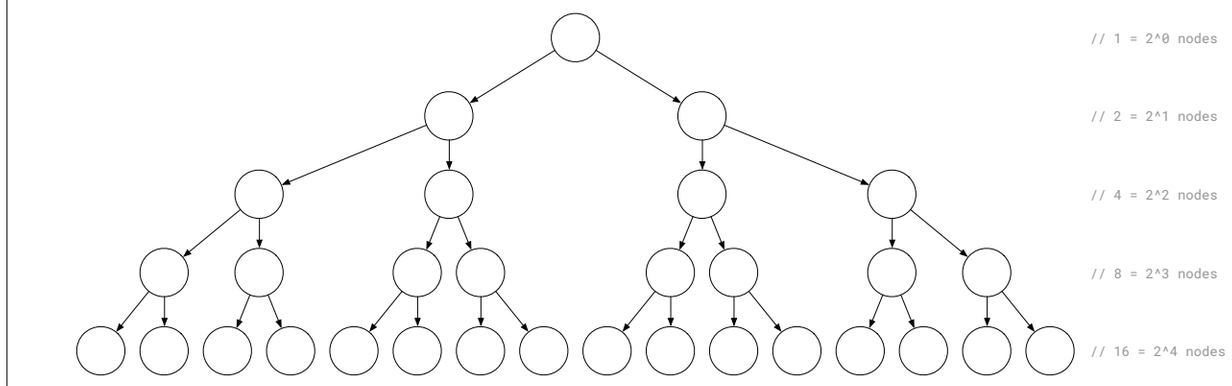

You've seen these two trees before. The one on the left doesn't look very tree-like. It was built from inserting one through seven in that order into a plain old binary search tree.

The one on the right is a balanced tree, it might be a red-black tree, or some other kind of balanced binary tree like a splay tree.

Think about how many nodes you'll have to examine to determine if some value is in these trees. And to keep things interesting, let's say we're looking for 100, which isn't in either tree.

With the unbalanced tree, we have to look at every node, since it is totally unbalanced. This is a worst-case scenario, but it could conceivably happen, like maybe you're adding nodes that are already sorted. This is O(n).

With the balanced tree, we only have to look at three nodes before knowing that 100 isn't in there. This algorithm is O(log n), which is loads better than O(n).




Each layer of a full binary tree has twice as many nodes as the prior layer.

If we have more data, and I think you already understand this intuitively, the unbalanced tree starts looking worse and worse compared to the balanced tree. This is what a perfectly balanced, "full" binary tree looks like. This means all the leaves have the same depth.

This tree only has five layers, but it packs in 31 nodes. A full six layer tree has 63. More generally, a full tree with N layers holds 2^n - 1 nodes.

What this means from a practical perspective is that you can store a huge amount of data, but all of that data is still relatively close to the root. A balanced tree with a billion nodes only has 30 layers. That means 30 steps to search through a billion records.

So... if your tree is balanced, you can make guarantees on runtime.

# Unbalanced vs Balanced

| | Unbalanced | Balanced |
|---|---|---|
| **Insert** | O(n) | O(log n) |
| **Remove** | O(n) | O(log n) |
| **Find** | O(n) | O(log n) |

**O(log n) is common and typically associated with subdivision-based data structures that strive to keep the number of subdivisions to a minimum. Like balanced binary search trees.**

We've seen both unbalanced and balanced binary search trees. You've implemented a plain unbalanced binary search tree, and we've talked about red-black trees, which are balanced.

Let's think about some of the operations on these trees.

The unbalanced tree really doesn't come out very well here. It is still practically better than a linked list, since it will usually have fewer hops. But the runtimes given here are the upper bound of how long it could take.

You'll see log-n runtimes all over the place. They're often associated with divide-and-conquer, subdivision based data structures that strive to keep the number of subdivisions to a minimum. Balanced binary search trees do this, because they try to be as 'short' as possible. A plain old binary search tree doesn't do this, so you can have pathological trees that are totally unbalanced. This is why the complexity of the plain binary search tree operations are actually the same as a linked list's.

Episode 3

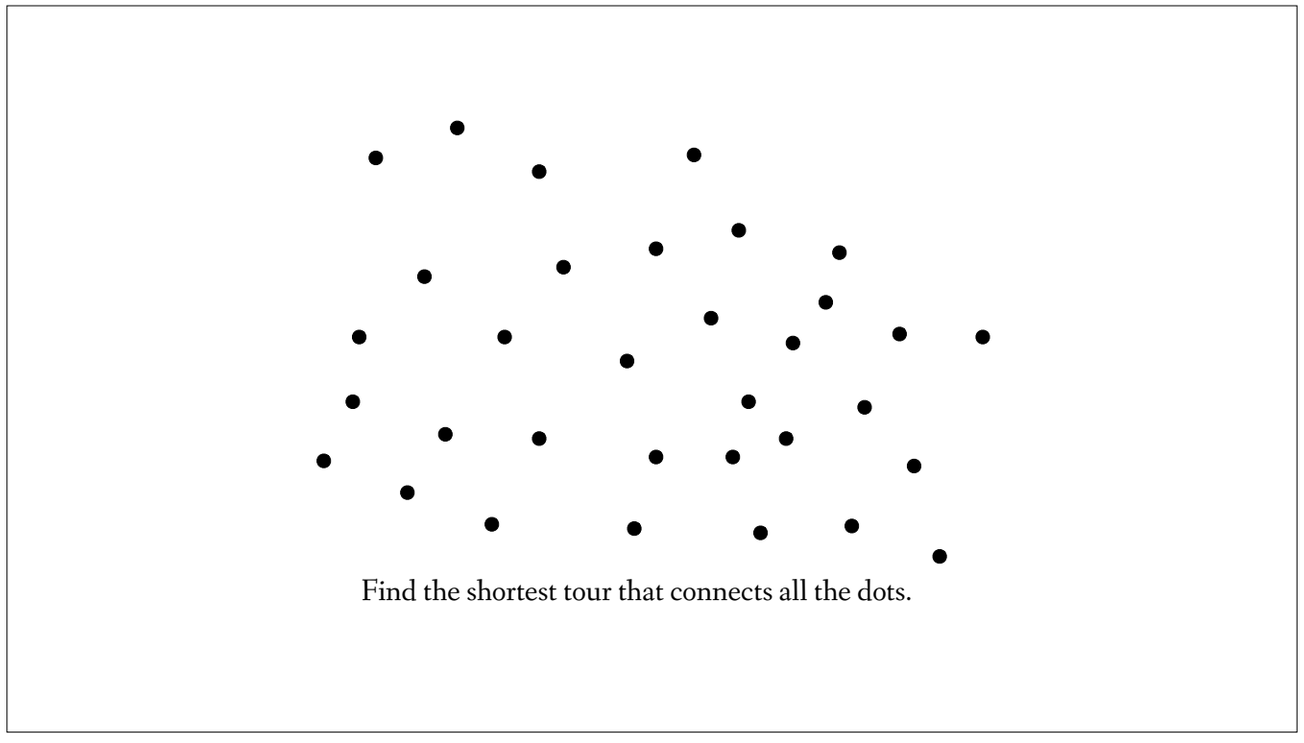
# Hard Problems

It might help understand the full range of complexity by talking about really hard problems that are very complex, and could take years, eons even, to finish using a supercomputer.

# Traveling Salesman Problem (TSP)


To give you a sense of what a hard problem looks like, at least from a complexity perspective, I'm going to explain the Traveling Salesman Problem. It is well-known enough that it has its own three-letter acronym: TSP.

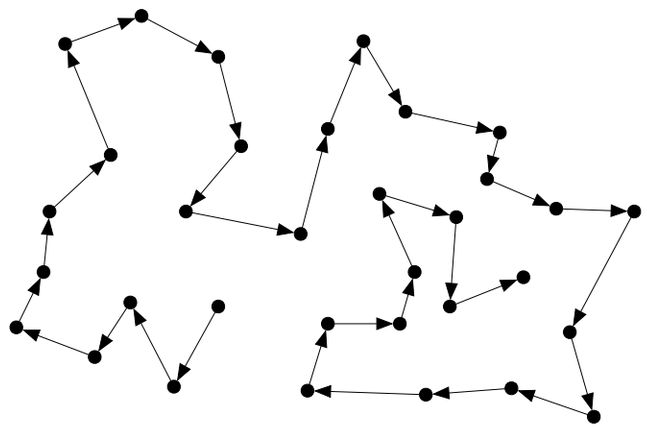
It is called the traveling salesman problem, because in that telling of the story, the idea is to plan a route that minimizes the number of miles driven between cities, and we want the salesman to start and end in his hometown.



Find the shortest tour that connects all the dots.

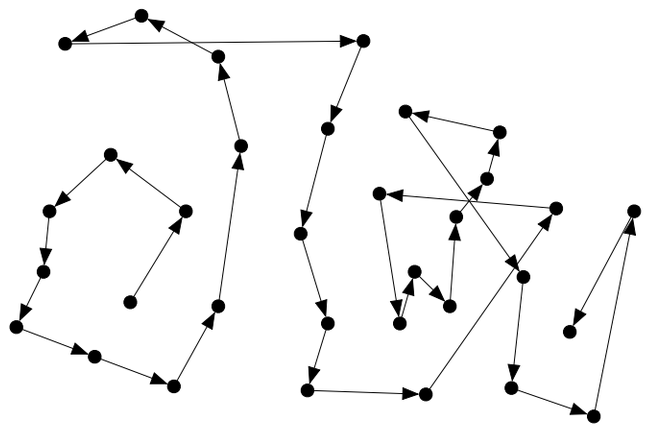
The TSP problem gives you a bunch of dots, and asks for the shortest total route that connects them all, starting and ending at the same dot. In these examples, I'm actually just looking for the shortest route, so I'm ignoring the bit about starting and ending at the same spot.

More generally, TSP-like problems could involve robot arms placing microchips, or vehicles making deliveries in a city. Turns out there are so many problems that are sort of like this, that many many PhDs have been devoted to studying the problem.



One possible tour of all the dots.

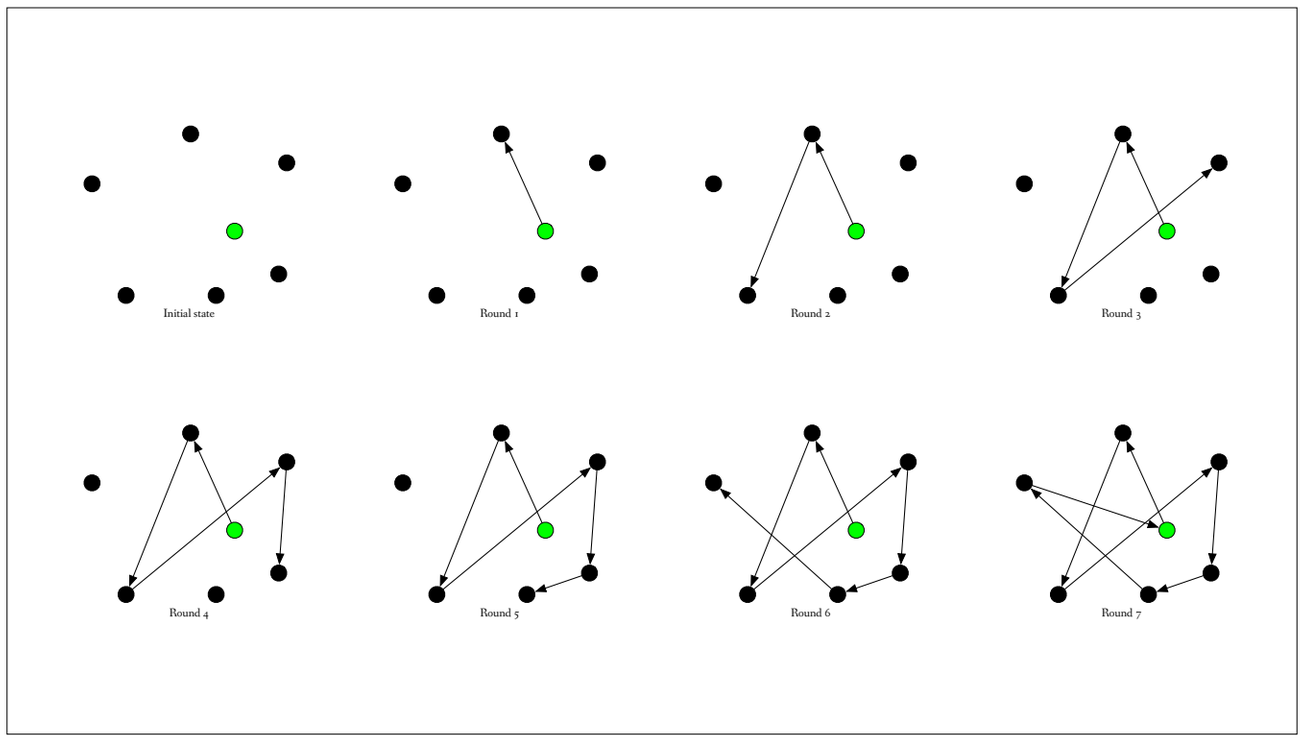
Here's one possible solution. I just eyeballed this one, didn't use any sort of algorithm. Unless you consider eyeballing it to be an algorithm.



Another possible tour of all the dots.

Here's another one. Again, I eyeballed it. I guess you could call it a "monte carlo" or "stochastic" approach, if you want to make it sound fancy.

I don't actually know what the solution to this bunch of dots is. I can tell you that there _is_ a solution, but the bad news is that finding that solution with a formal algorithm might take a long time. And I only have a few dozen dots there. If there's thousands of nodes, it might take a long, long time.




Think about a brute force approach, starting and stopping from some given dot?

One way of doing it is to exhaustively enumerate each possible path, compute their lengths, and keep track of the shortest one.

# Brute Force TSP?

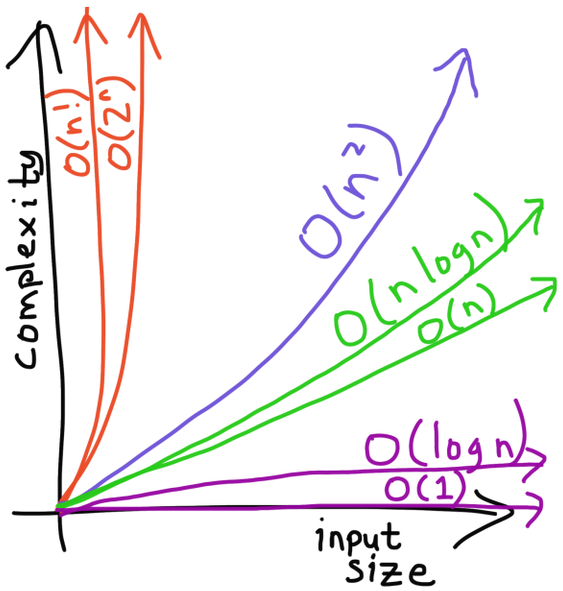
```
func tsp(dots) {
    best_sequence = null
    best_length = infinity
    all_sequences = make_all_permutations(dots)
    for each sequence in all_sequences {
        this_length = compute_length(sequence)
        if this_length < best_length {
            best_sequence = sequence
            best_length = this_length
        }
    }
    return best_sequence, best_length
}
```

We can brute force it with an algorithm like this.

First of all, don't try this at home.

Second of all, what do you think the runtime complexity of this algorithm is? I mean, it looks pretty straightforward, right? Recall that our big-oh notation depends on the length of the input. In this case, dots. So anyway, which part of this algorithm grows with the length of dots?

That `make_all_permutations` function is going to be the determining factor. You may or may not remember, that if you have a seven-letter word, the number of permutations of that word is `7!` -- 7 times 6 times 5, etc, down to one. And `7! = 5040`. For our `make_all_permutations` function, instead of a seven-letter word, we've got the equivalent of an N-letter word. We need to make every possible permutation. If we had 8 dots, that's 40320 permutations. Just 13 dots and we have over 6 billion permutations.




**Brute Force TSP is O(n!)**

**(that's bad...)**

This code has a runtime complexity of O(n!), and it will ruin your day.

Notice that we're making all the permutations, and then storing them in that `all_sequences` variable. So not only does this have an abysmal runtime complexity, it also has O(n!) space complexity as well!

Fun fact! There are only about 10^80 atoms in the universe. I haven't counted them myself, but I have my sources.

So if your N is, say 60, you will need to store almost 10^82 permutations... that's more than than the number of atoms there are in the universe.

# Heuristics

**Intractable problem getting you down?**

## Try a heuristic!

**Settle for an approximate answer!**

***Sure does beat waiting for the heat death of the universe!***

Some algorithms like TSP involve an enormous, basically intractable, amount of computation to get an exact solution. Happily, sometimes we can get away with a "good enough" solution.

You can use heuristics, basically clever little rules, that will lead to a decent solution, and they can run with much better runtimes. These algorithms only give you approximate solutions with no promises as to how far off from the 'true' solution they might be.

Episode 4

# Bubble Sort

So we've finally got to the coding part of this sequence.

We've already seen that binary search trees can help us do things like add values to a data structure efficiently, and that's because we store the data in clever ways that make it possible to do less work. Binary trees aren't the only structures that help out with sorting. Even simple arrays can be used for getting similar effects. Suffice to say, once you have sorted data, operations become much more powerful.

This sequence's homework assignment is about three classic sorting algorithms. No fancy new data structure here. We're using a C++ class called a vector. I'll talk more about vectors in a coding session later on.

**This actually happened**

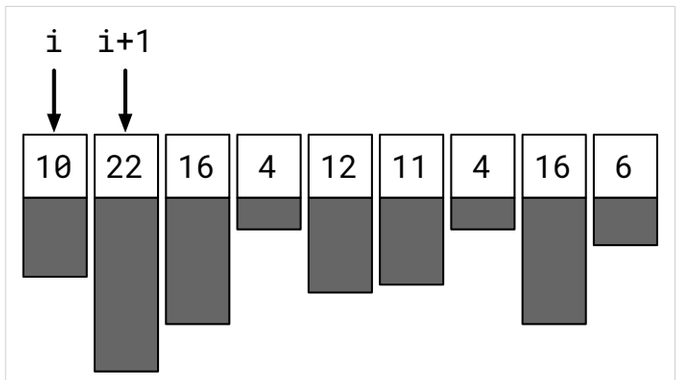

**2008 - Eric Schmidt (CEO of Google) interviewing Senator Obama**

So, this actually happened.

I guess Obama was still campaigning and he visited Google. Eric Schmidt asked him this question in front of like a thousand people, and he somehow knew enough to say that bubble sort isn't the way to go. I don't know if he was coached on that, or if he has a little bit of programming experience.

Anyway, like the man suggested, bubble sort is not necessarily the best thing you can use to sort a list of integers. It does have the nice property of being simple to understand.

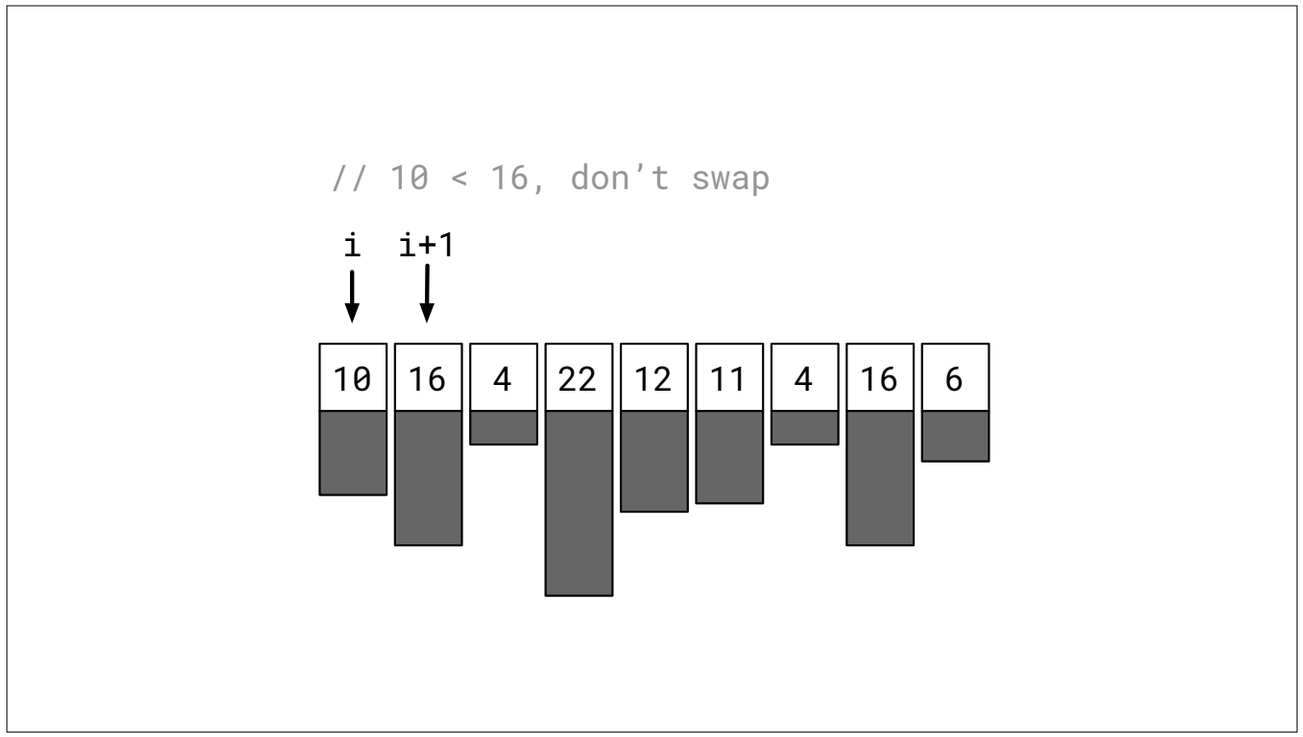
**Bubble Sort:** Start at the beginning, compare each item with the next. When they are out of order, swap them. Then increment your index, and continue until the end of the list.




I'll just show some examples. Here the numbers have bars below them to give a better visualization.

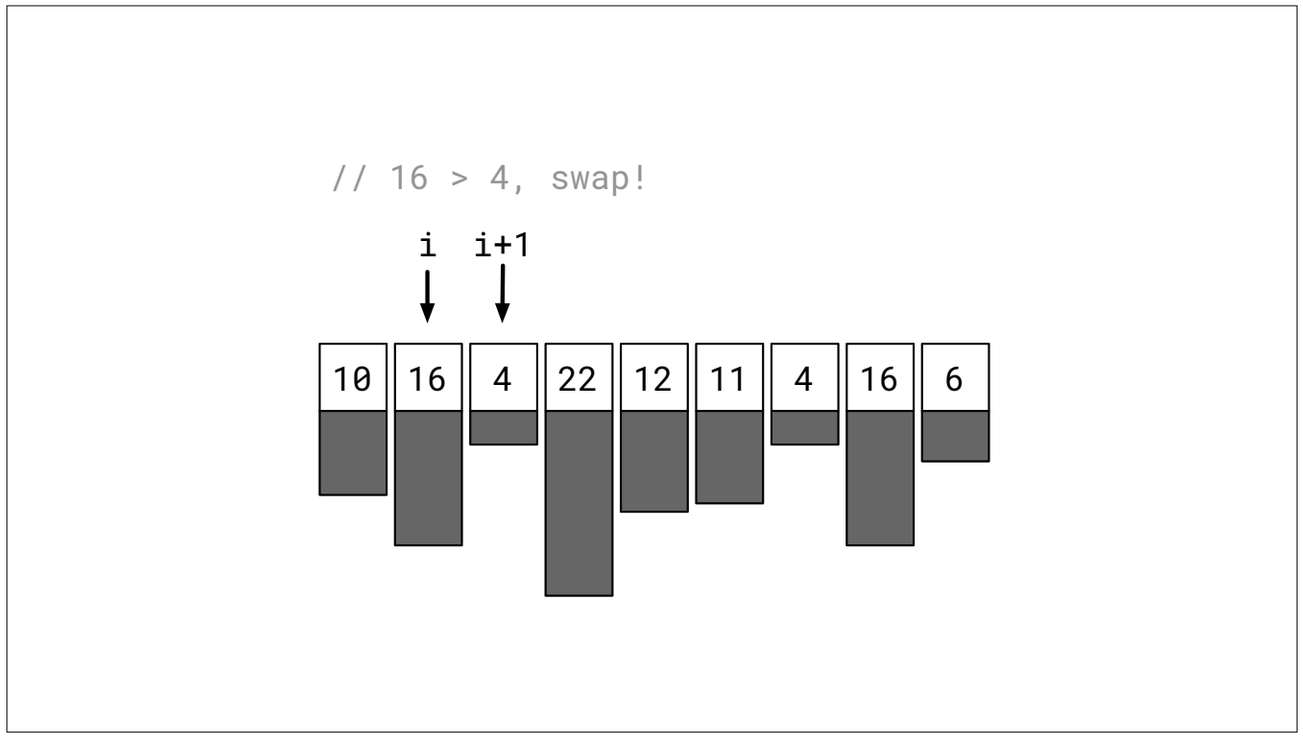
The main idea with bubble sort is that we're going to march through the list in order, and when we see neighboring pairs that aren't in the right order, we swap them, and continue on to the next pair. Then at the end, if the list isn't totally sorted, we do it again.

Not super efficient, but it gets the job done.



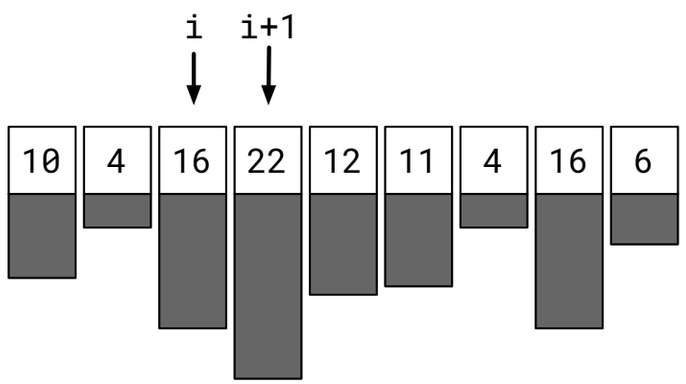

Here's a walkthrough of the first iteration.

We just look at values in i and i+1. When value in i is larger than the one in i+1, we swap the values. If not, we don't mess with them.

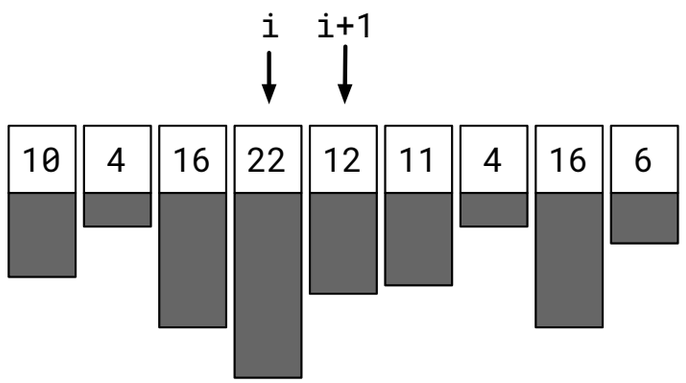



Regardless, the next thing we do is increment i so we're looking at the next pair. We continue this until we reach the end of the list.

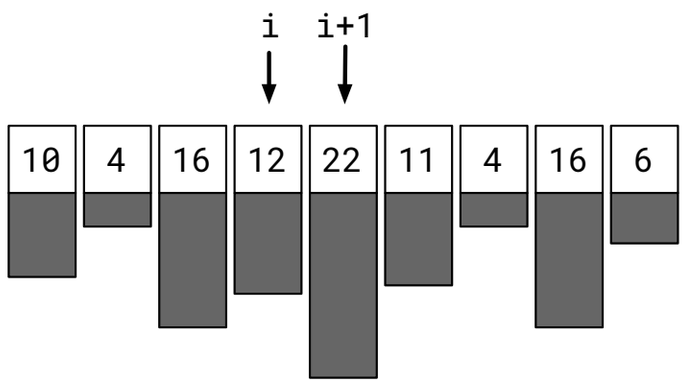
```
// 16 < 22, don't swap
```



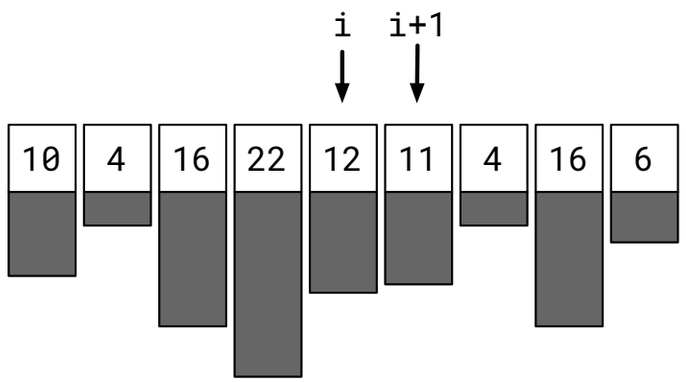
```
// 22 > 12, swap!
```



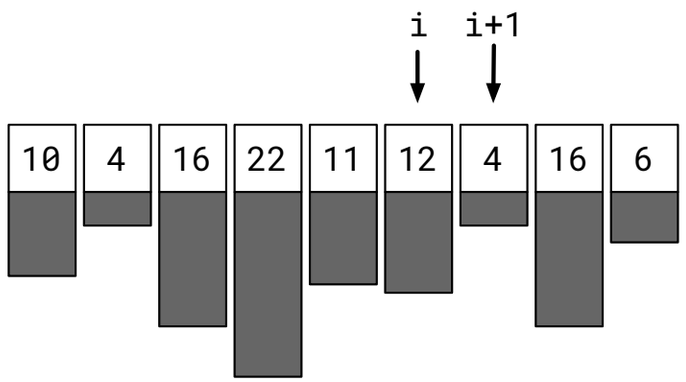
```
// 22 > 12, swap!
```



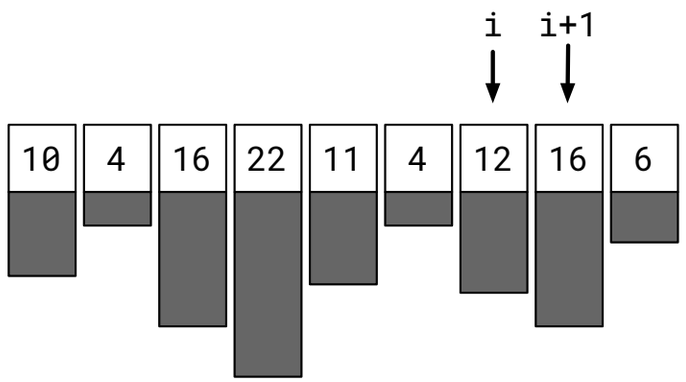
```
// 12 > 11, swap!
```



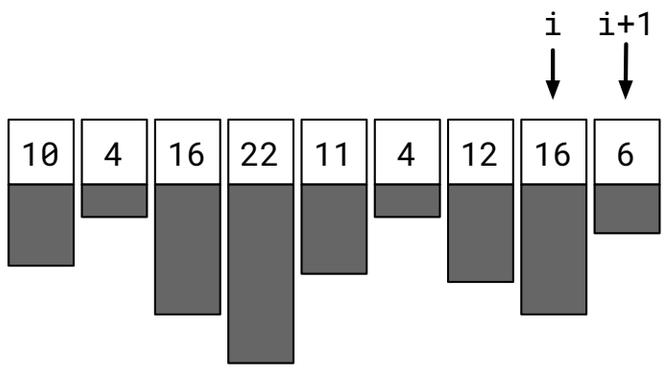
```
// 12 > 4, swap!
```



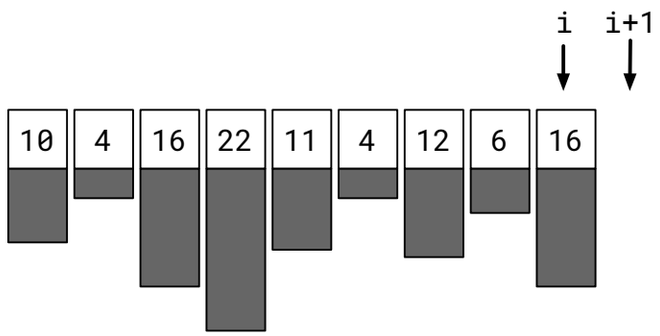
```
// 12 < 16, don't swap
```



```
// 16 > 6, swap!
```

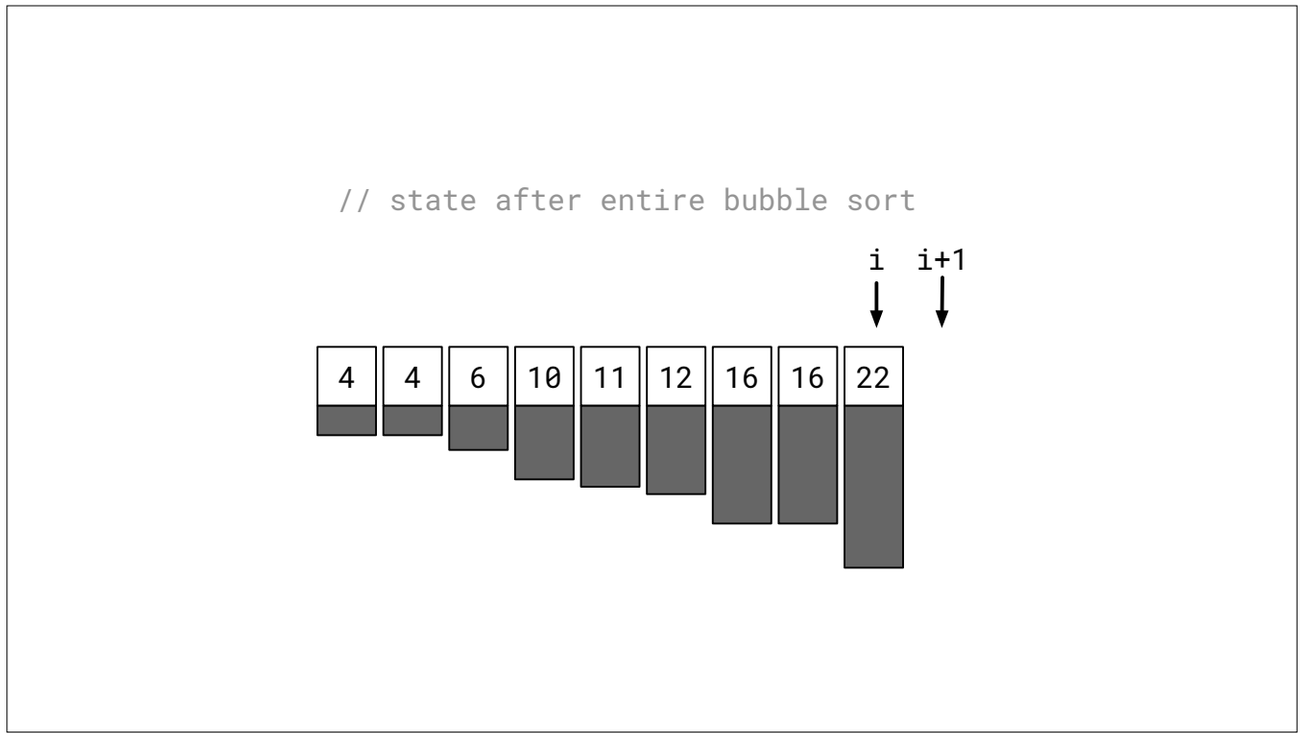


```
// state after first sweep
```



**Since the list is not sorted, reset i=0 and do it over and over again until it is sorted.**

At the end of the list, if the data isn't yet sorted, we just do it again. Really, it is that simple. But the runtime complexity is bad... O(n^2).




After we’ve done this a few more times, it eventually becomes fully sorted.

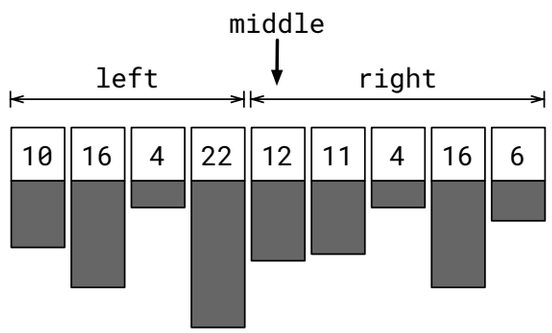
Episode 5

# Merge Sort

Merge sort is another algorithm for sorting lists of data. This one is pretty efficient.

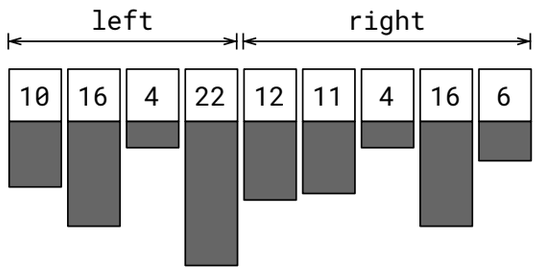
**Merge Sort:** based on chopping a list into smaller lists and recursively merge sorting them. To chop a list, find the middle, and create two new lists based on what is to the left and right.

```
// middle = index of item at
// or near center of list
```




It is based on recursively chopping up the list into smaller and smaller sub-lists and then merging them in sorted order.

Start by picking the cell in the middle. If you're working with a list with an odd number of elements, there's an obvious cell in the middle. If you've got an even number, pick one of the two candidates.




**Mergesort is recursive.**

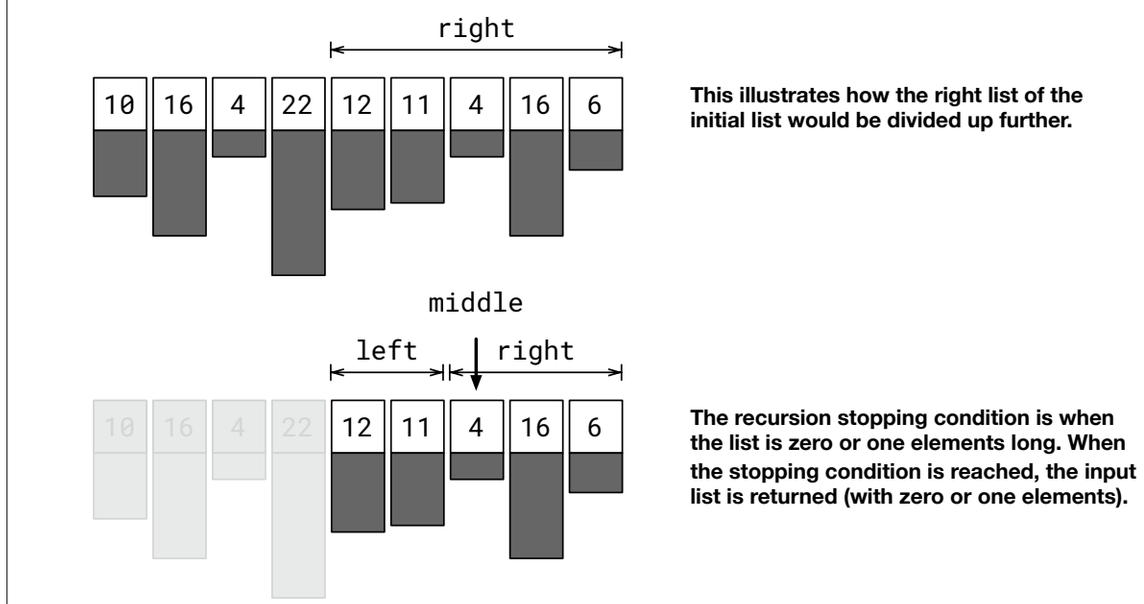
**Each side (left and right) is modified by a recursive call to mergesort and then those results are combined in a merge operation.**

```
// then recursively mergesort these lists
left_list  = mergesort(left_list)
right_list = mergesort(right_list)

// then merge the results
combined = merge(left_list, right_list)
```

**Merge is illustrated in a few slides.**

Merge sort is recursive. From the middle element you can form a left and right (the middle goes along with the right side). Then take left and right, and call merge sort on those lists. Those recursive calls give you two sorted lists, which you can then merge together. I'll get to the merge step in a few slides. But first let's focus on the recursion.



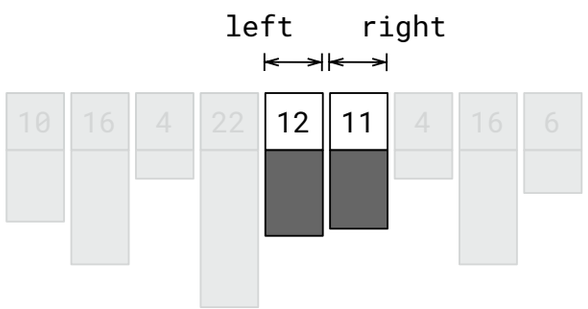

**This illustrates how the right list of the initial list would be divided up further.**

**The recursion stopping condition is when the list is zero or one elements long. When the stopping condition is reached, the input list is returned (with zero or one elements).**

So let's say we're going to recursively call merge sort with the right list. Once inside the second merge sort call, we pick a middle, and make two smaller lists. Here we've got a two-element left side, and a three-element right side.

You continue doing this, recursively calling merge sort with left and right, until the stopping condition.

The stopping condition is when the list has zero or one elements. Depending on how you pick your middle you might not see a zero-element list.

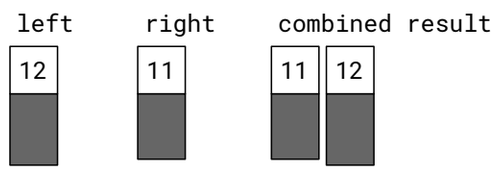



**This is one such stopping condition. The lists are both one element long, so they can't be divided any more.**

**Since we're dividing lists first, before merging, this means the first lists we merge are the zero- and one-element lists.**

**This guarantees that we merge small lists first, then larger ones.**

Here we have left and right each with one element. When we recurse with those, the merge sort will just return the inputs. Remember, merge sort is supposed to return a sorted list, and a list with one element is technically sorted.




**Merging two lists: the idea is to use the left and right lists (which start out sorted), to form a third combined result that is also sorted.**

**Compare the first element in left and right, remove the smaller one, and append to the result. If there's a tie, it doesn't matter which one we choose.**

```
// merge:
//
// compare the first element in left and right,
// remove the smaller of the two, and place it
// at the end of the combined result.
```

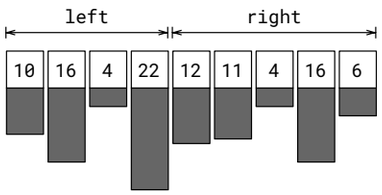
**This first merge is not very exciting because it is only merging single-item lists. But remember it is necessary to start with zero-or one-item lists to ensure sort order is intact.**

So now we 've got out of the recursing stage, at least with these one-element lists, so we can merge them.

The idea with merging two lists is to take the smallest element available from either list, and move it into a combined list. Since the input lists are sorted, it is easy to identify the smallest elements.

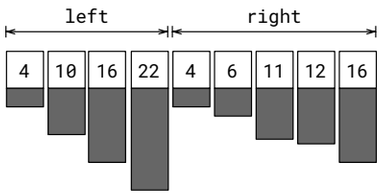
With one-element lists, the merge isn't interesting, it just results in a two-element list that is sorted. We return that list.

**So given the initial list subdivision…**



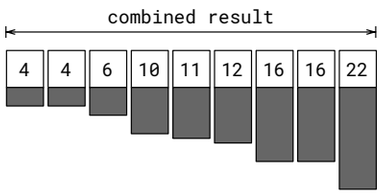

`// start state`

**When its recursive calls return, it looks like this. Last step is to merge…**




`// after recursion`

**After merging, the list is in sorted order.**




`// after merging`

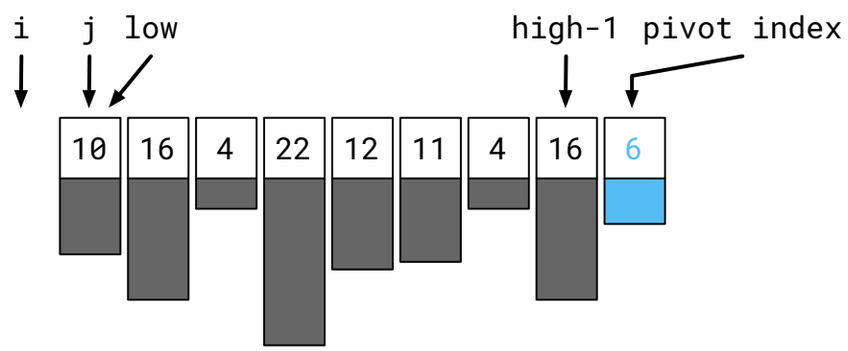
So given the initial list you see here, we chop it up into left and right, and recurse with both of those. When each recursive call gets back, we have sorted versions of left and right, which we can then merge. And then we're left with the original input, but sorted. And we're done!

Episode 6

# Quick Sort

Quick sort is a somewhat trickier sorting algorithm than we've discussed, but it is by no means the trickiest.

**Quick Sort:** Pick a pivot index between the low and high indices. Then partition data by moving everything less than data[pivot] to the left, and everything else to the right. Then recurse to left and right sides.




The idea with quick sort is to split it in two parts called partitions, and recursively quick sort those partitions.

Most of the work happens in the partioning process. We pick a pivot value, and then move all the items lower than that value to the left partition, and those greater than the pivot value go to the right.

**quick sort pseudocode.** most of the work is done in the partition function.

```
// low to high inclusive
quicksort(data, low, high) {
  if low < high { // recursive stopping condition
    p = partition(data, low, high)
    quicksort(data, low, p-1)  // recurse left
    quicksort(data, p+1, high) // recurse right
  }
}
```

```
partition(data, low, high) {
  pivot = data[high]
  i = low - 1
  // low to high-1 inclusive
  for j in range [low, high-1] {
    if data[j] < pivot {
      i = i+1
      swap(data, i, j)
    }
  }
  if data[high] < data[i+1] {
    swap(data, i+1, high)
  }
  return i+1
}
```

Here's the pseudocode for quicksort and the related partition function. You can come back to this slide to see it all together, that will definitely help with the homework. I'll go over both of these now.

```
quicksort(data, low, high) {
  if low < high { // recursive stopping condition
    p = partition(data, low, high)
    quicksort(data, low, p-1)  // recurse left
    quicksort(data, p+1, high) // recurse right
  }
}
```

**The first thing we do is partition the data. This gives us an index that separates ‘left’ from ‘right’.**

**Items smaller than the partition’s value are on the left, larger are on the right. Equal values can go either way.**

**There are lots of ways to implement this, but anything that fits this description is quicksort.**

In the quicksort function, We're given a list of data, and a low and high index.

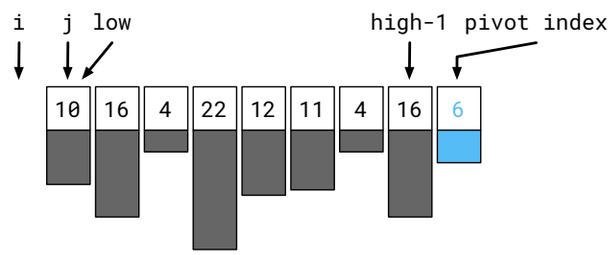
So quicksort is recursive, so having a recursion stopping condition is important. We keep recursing as long as the low index is less than the high index.

First thing to do is partition the data, which gives us a partition index. We then recurse with left and right lists. With quicksort, after partitioning the data, we have values smaller than the partition value, and larger ones on the right. Equal values can go either way.

We then take the left and right sides and recursive with each. Notice that the partition index isn't acutally sent along, we use indices next to it instead.

Also I should point out that there are many subtle variations on quicksort, but anything that fits this general description is quicksort.

```
partition(data, low, high) {
  pivot = data[high]
  i = low - 1
  // low to high-1 inclusive
  for j in range [low, high-1] {
    if data[j] < pivot {
      i = i+1
      swap(data, i, j)
    }
  }
  if data[high] < data[i+1] {
    swap(data, i+1, high)
  }
  return i+1
}
```




**Partitioning for the very first time.** Low indexes the first item, high indexes the last item. We pick pivot index to be high because it makes it easier to follow. We could have picked any pivot index, but then the code would be more complicated.

Let's dive in to partitioning now. We're given a list of data, and then two indices. Low indexes the first item in the range we're working on, high indexes the last item in that range.

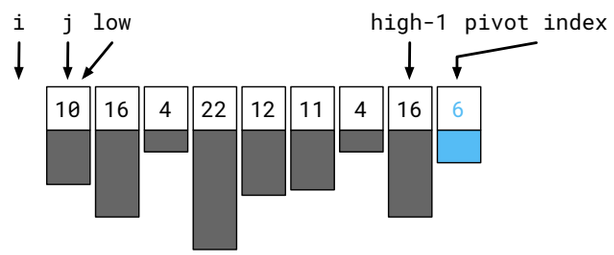
First thing to do is pick a pivot index. We're going to choose the last element, mostly because it makes the algorithm easier to follow. There are variations on quicksort where you use a different pivot index.

So now we have a pivot index, and a pivot value, which in this case is six.

We then use two cursor indices, i and j. We set i to low minus one, which is safe because before we use it for the first time we will increment it so we're addressing a valid element.

J will range from the low index up to high minus one. We stop just shy of the high index because it contains our pivot value, and we'll deal with that element specifically later on.

```
partition(data, low, high) {
  pivot = data[high]
  i = low - 1
  // low to high-1 inclusive
  for j in range [low, high-1] {
    if data[j] < pivot {
      i = i+1
      swap(data, i, j)
    }
  }
  if data[high] < data[i+1] {
    swap(data, i+1, high)
  }
  return i+1
}
```




**Partitioning for the very first time.** Low indexes the first item, high indexes the last item. We pick pivot index to be high because it makes it easier to follow. We could have picked any pivot index, but then the code would be more complicated.

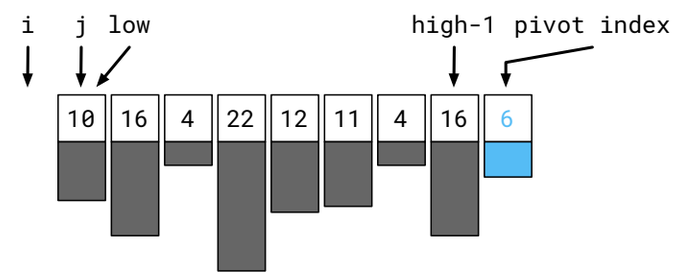
We then loop for the full range of j. Whenever we see the value at j is less than our pivot value of six, we're going to increment i, then swap the values in cells i and j.

After that we handle the pivot cell itself similarly, and then return the partition index. When we return, all the values from the low index up to the partition are less than the partition value (six), and those from the partition up to the high index are larger or equal to six.

Now for a marathon example of the partition process!

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

**10 > 6 - don't swap**

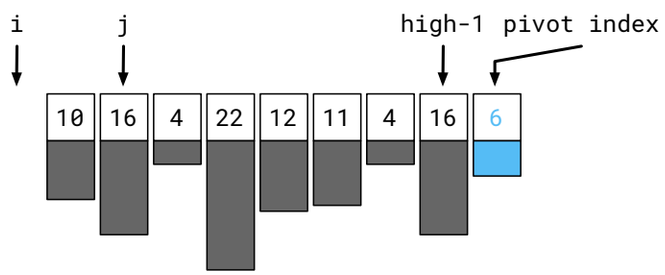



Let's say we're in the partition function, and we've set up our variables and we're now in that loop, the first time around. We compare the value at j with our pivot, and if they're the same, we do a swapping thing.

OK, 10 is larger than 6, don't swap, move along, nothing to see here...

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

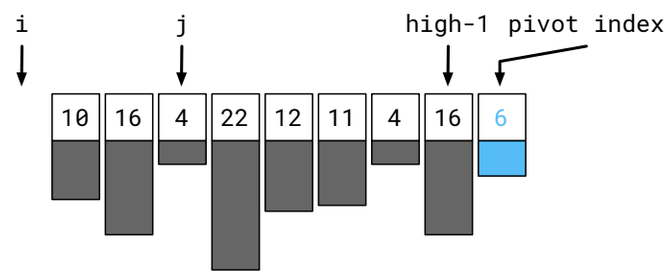
**16 > 6 - don't swap**




Same thing here...

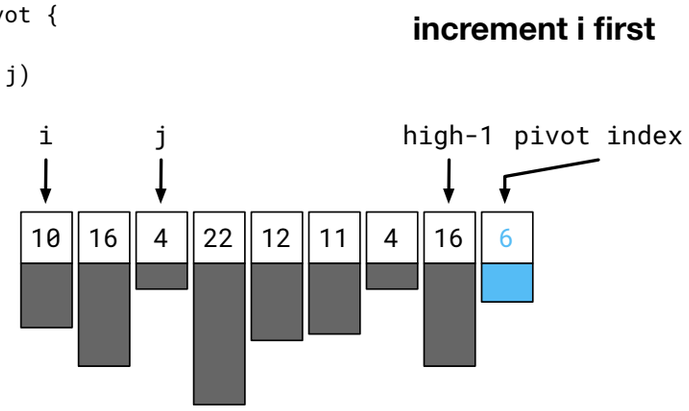
```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

**4 < 6 - enter swap code**




OK, 4 is less than 6, so we enter the swap code.

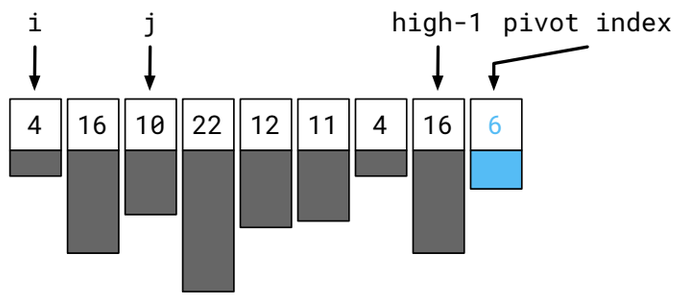
```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```




First, increment i, as promised.

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

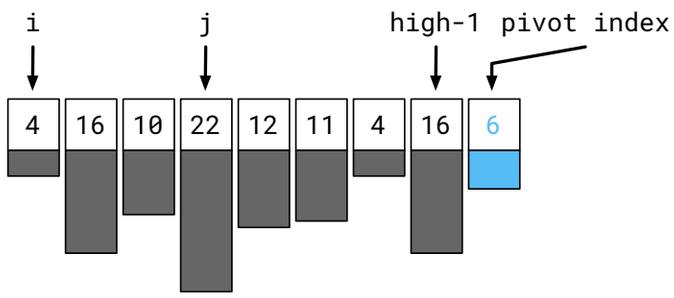
**then swap values in cells i and j**




Then swap values in cells i and j. We leave pivot alone.

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

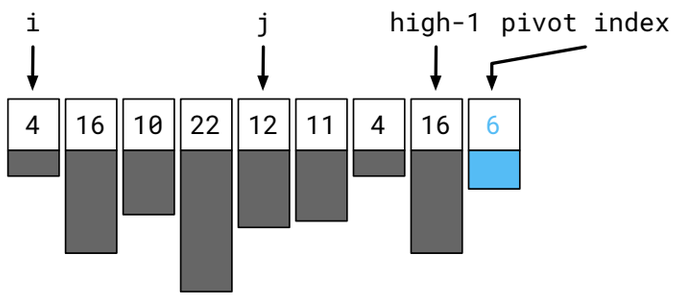
**continue. 22 > 6 - don’t swap**




Then continue like before...

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

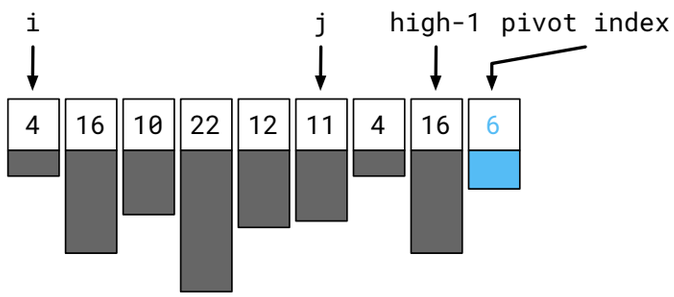
**12 > 6 - don't swap**




Same…

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

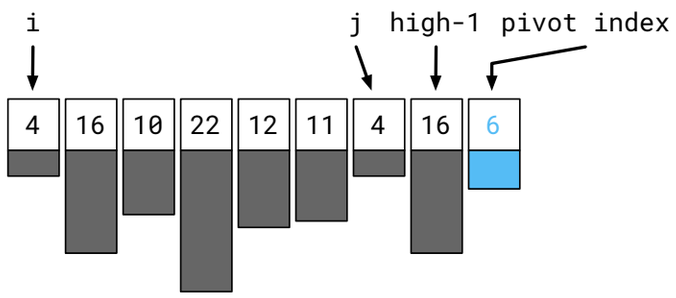
**11 > 6 - don't swap**




Same…

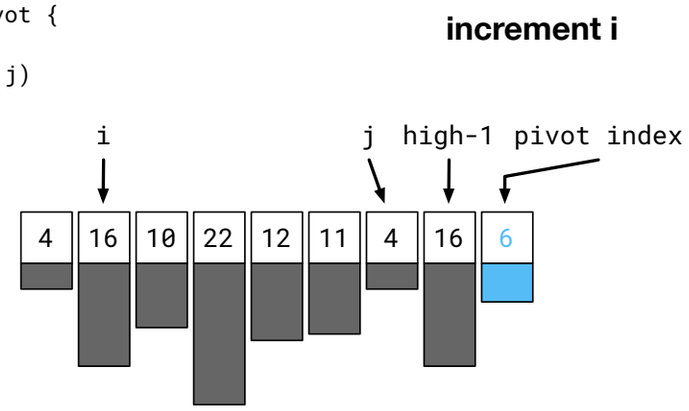
```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

**4 < 6 - enter swap code**




Another swap…

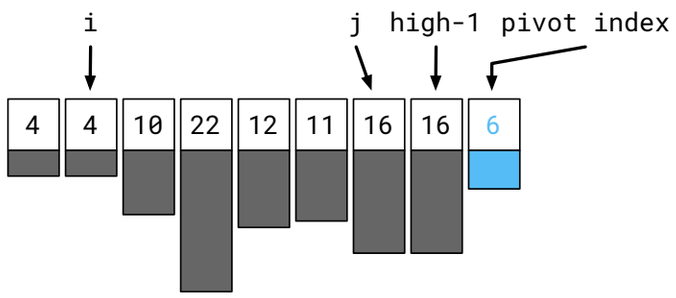
```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```




increment i

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

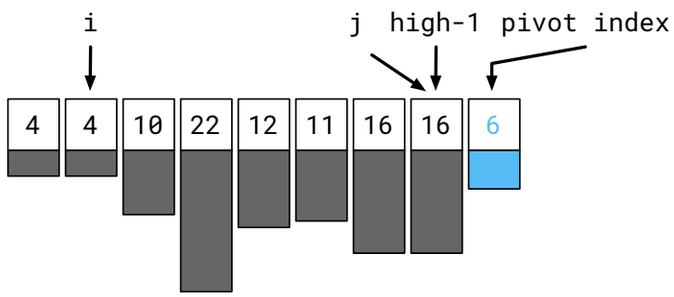
**swap values in i and j**




swap values in i and j…

```
// loop in partition
if data[j] < pivot {
  i = i+1
  swap(data, i, j)
}
```

**continue. 16 > 6 - don’t swap**



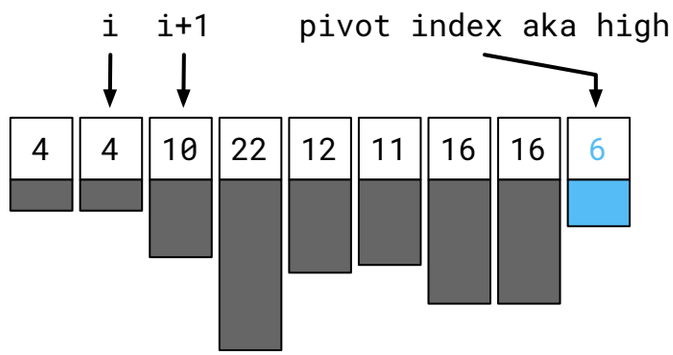

**we’ve reached the end of the loop where we increment j.**

continue…

Now we've reached the end of the loop where we increment j. Time to handle the pivot cell itself.

```
// swap pivot if necessary
if data[high] < data[i+1] {
  swap(data, i+1, high)
}
```

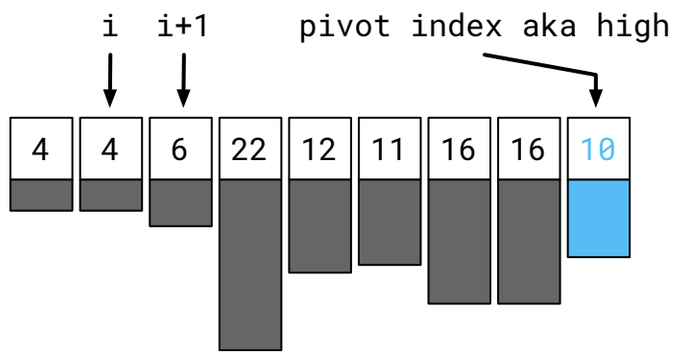
**6 < 10, swap values for pivot and i+1**



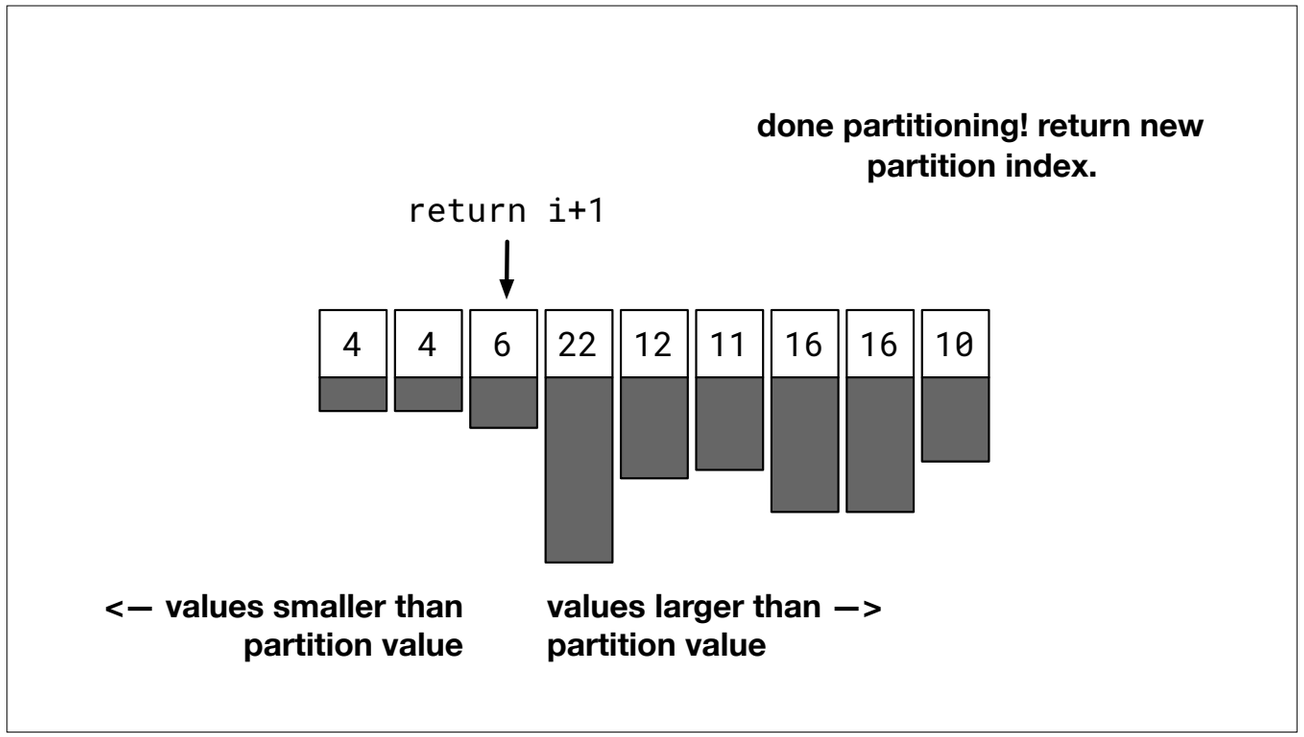

Because as you can see, that six seems awfully weird at the end there. We compare the pivot value with the value at i+1, and if our pivot is smaller, we swap those two.

```
// swap pivot if necessary
if data[high] < data[i+1] {
  swap(data, i+1, high)
}
```

**6 < 10, swap values for pivot and i+1**



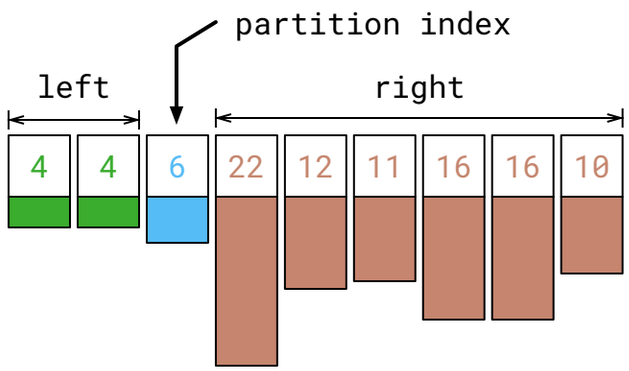

And in this pseudocode we're just using i+1, but we could increment i and use plain old i too.




Then, we're done partitioning! We return the i+1 cell that we just swapped our six into. Looking at this list before we return: values with a smaller value than six are to the left, larger values are to the right.

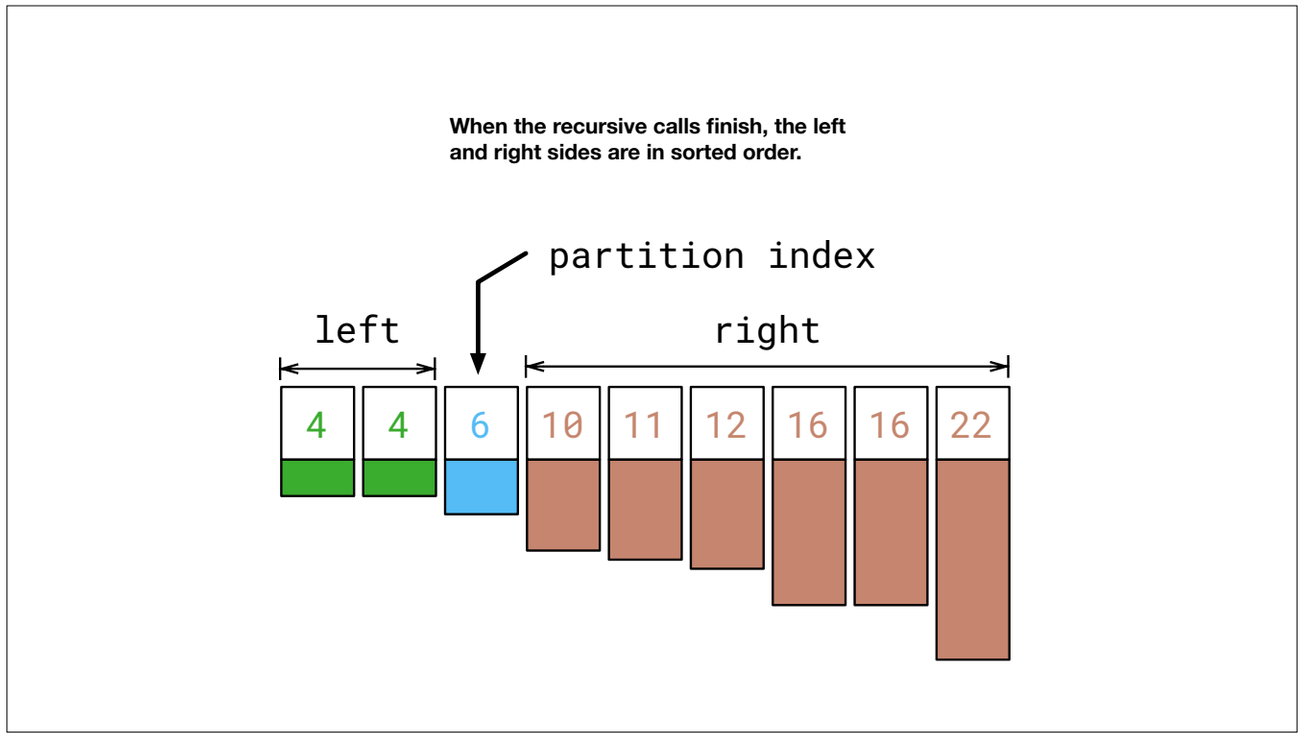
**Now all that is left is to recursively quick sort the left and right sides using the same algorithm but using the pivot to divide the process.**

**quicksort(data, left, p - 1) // recurse with left side**
**quicksort(data, p + 1, right) // recurse with right side**




Now all that we have to do is recursively quick sort the left and right sides of the partition using the same code.

Like I said earlier, we don't actually include the partition index.




When those recursive calls come back, the whole list is sorted! And, that's quicksort.