Sorting Lower Bound and Non-Comparison Sorts

Data Structures and Parallelism

Announcements

P1 feedback coming soon.

Make sure to look at it before the end of P2.

You can use up to two late days for P2

But watch out for the exercise deadline that Friday.

Writeup: Timing vs. Counting handout on webpage.

Quick Sort

Still Divide and Conquer, but a different idea:

Let's divide the array into "big" values and "small" values

- And recursively sort those

What's "big"?

- Choose an element ("the pivot") anything bigger than that.

How do we pick the pivot?

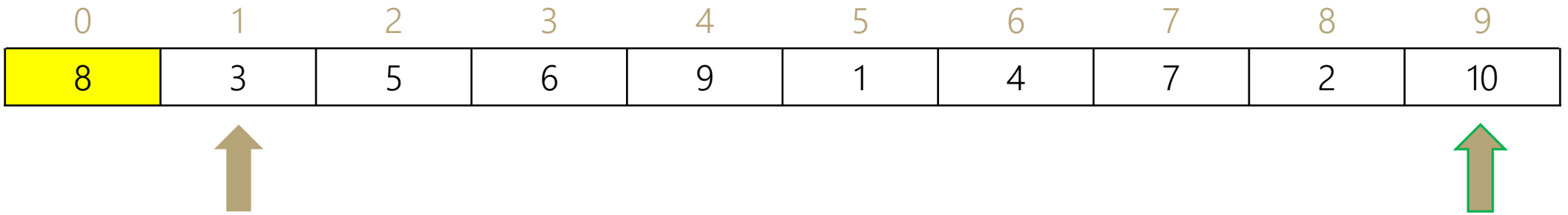
For now, let's just take the first thing in the array:

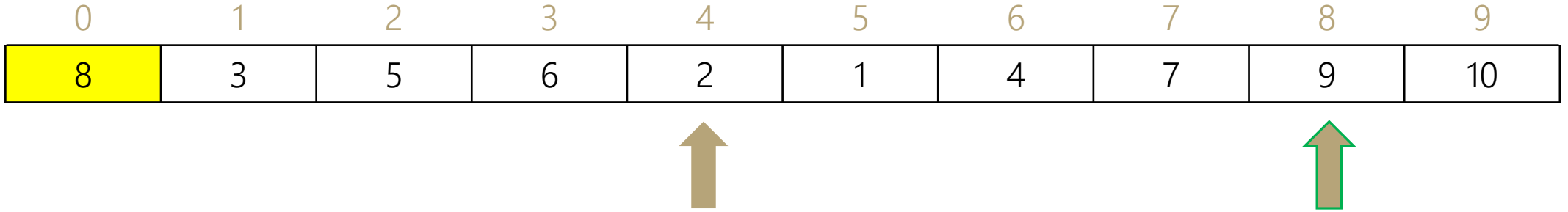
Swapping

How do we divide the array into “bigger than the pivot” and “less than the pivot?”

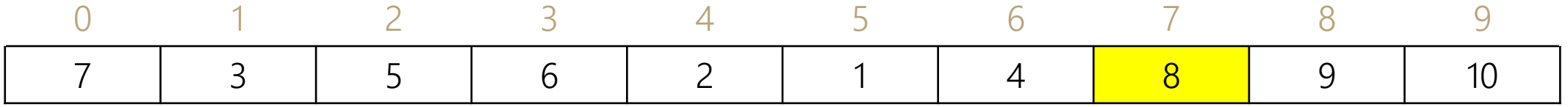
1. Swap the pivot to the far left.
2. Make a pointer i on the left, and j on the right
3. Until i, j meet
   - While $A[i] < \mathbf{pivot}$ move i left
   - While $A[j] > \mathbf{pivot}$ move j right
   - Swap $A[i], A[j]$
4. Swap A[i] or A[i-1] with pivot.

Swapping



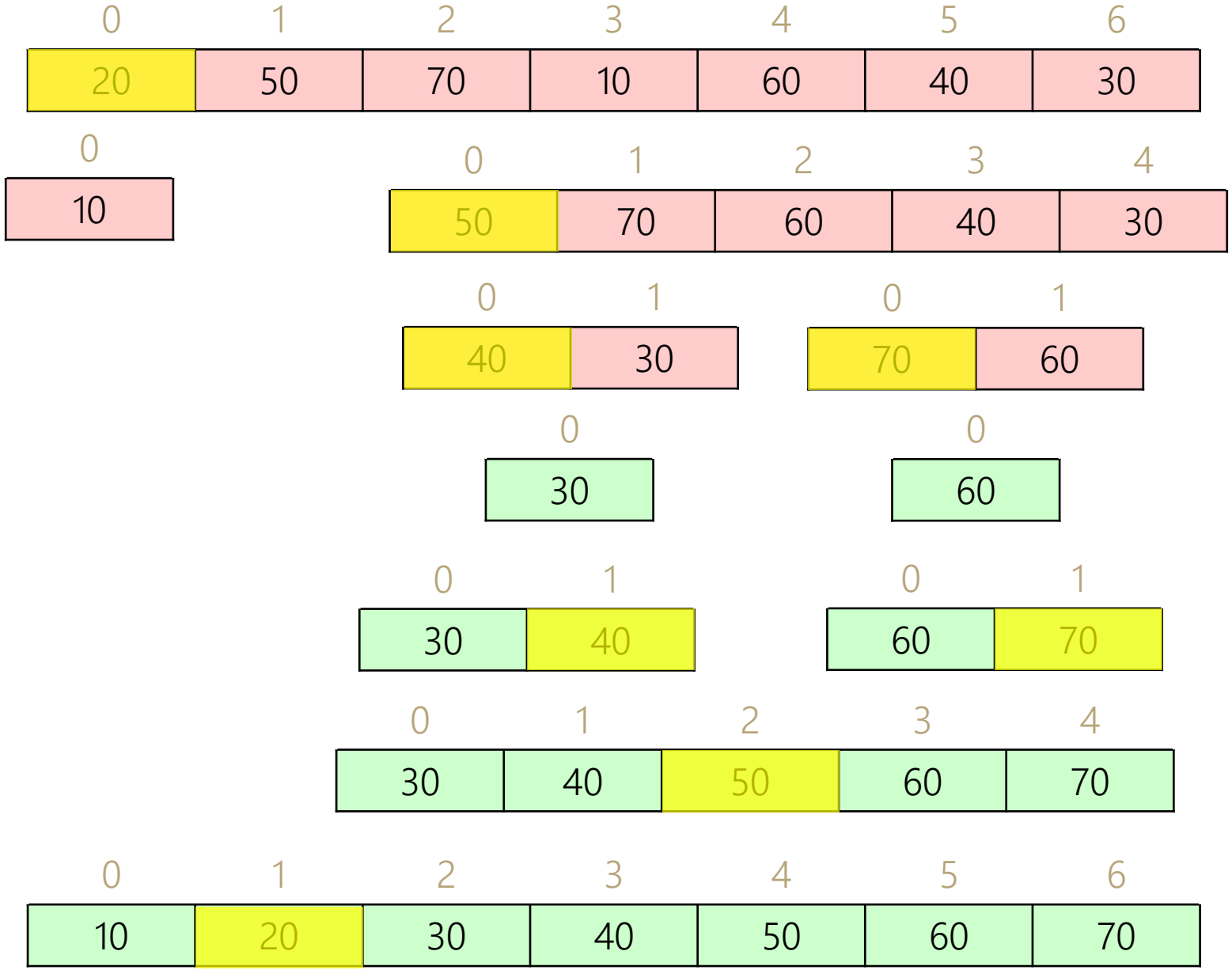


i, j met. $A[i]$ is larger than the pivot, so it belongs on the right, but $A[i-1]$ belongs on the left. Swap pivot and $A[i-1]$.



Quick Sort

https://www.youtube.com/watch?v=ywWBy6J5gz8



Quick Sort Analysis (Take 1)

What is the best case and worst case for a pivot?

- Best case: Picking the median
- Worst case: Picking the smallest or largest element

Recurrences:

Best: $$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c_1 n & \text{if } n \geq 2 \\ c_2 & \text{otherwise} \end{cases}$$

Worst: $$T(n) = \begin{cases} T(n-1) + c_1 n & \text{if } n \geq 2 \\ c_2 & \text{otherwise} \end{cases}$$

Running times:

- Best: $O(n \log n)$
- Worst: $O(n^2)$

Choosing a Pivot

Average case behavior depends on a good pivot.

Pivot ideas:

Just take the first element

Pick an element uniformly at random.

Find the actual median!

Choosing a Pivot

Average case behavior depends on a good pivot.

Pivot ideas:

Just take the first element

- Simple. But an already sorted (or reversed) list will give you a bad time.

Pick an element uniformly at random.

- $O(n \log n)$ running time with probability at least $1 - 1/n^2$.
- Regardless of input!
- Probably too slow in practice :(

Find the actual median!

- You can actually do this in linear time
- Definitely not efficient in practice

Choosing a Pivot

Median of Three

- Take the median of the first, last, and midpoint as the pivot.
- Fast!
- Unlikely to get bad behavior (but definitely still possible)
- Reasonable default choice.

Quick Sort Analysis

Running Time:

- Worst $O(n^2)$
- Best $O(n \log n)$
- Average $O(n \log n)$ (not responsible for the proof, talk to Robbie if you're curious)

In place: Yes

Stable: No.

Lower Bound

We keep hitting $O(n \log n)$ in the worst case.

Can we do better?

Or is this $O(n \log n)$ pattern a fundamental barrier?

Without more information about our data set, we can do no better.

Comparison Sorting Lower Bound

Any sorting algorithm which only interacts with its input by comparing elements must take $\Omega(\mathrm{n} \log \mathrm{n})$ time in the worst case.

Decision Trees

Suppose we have a size 3 array to sort.

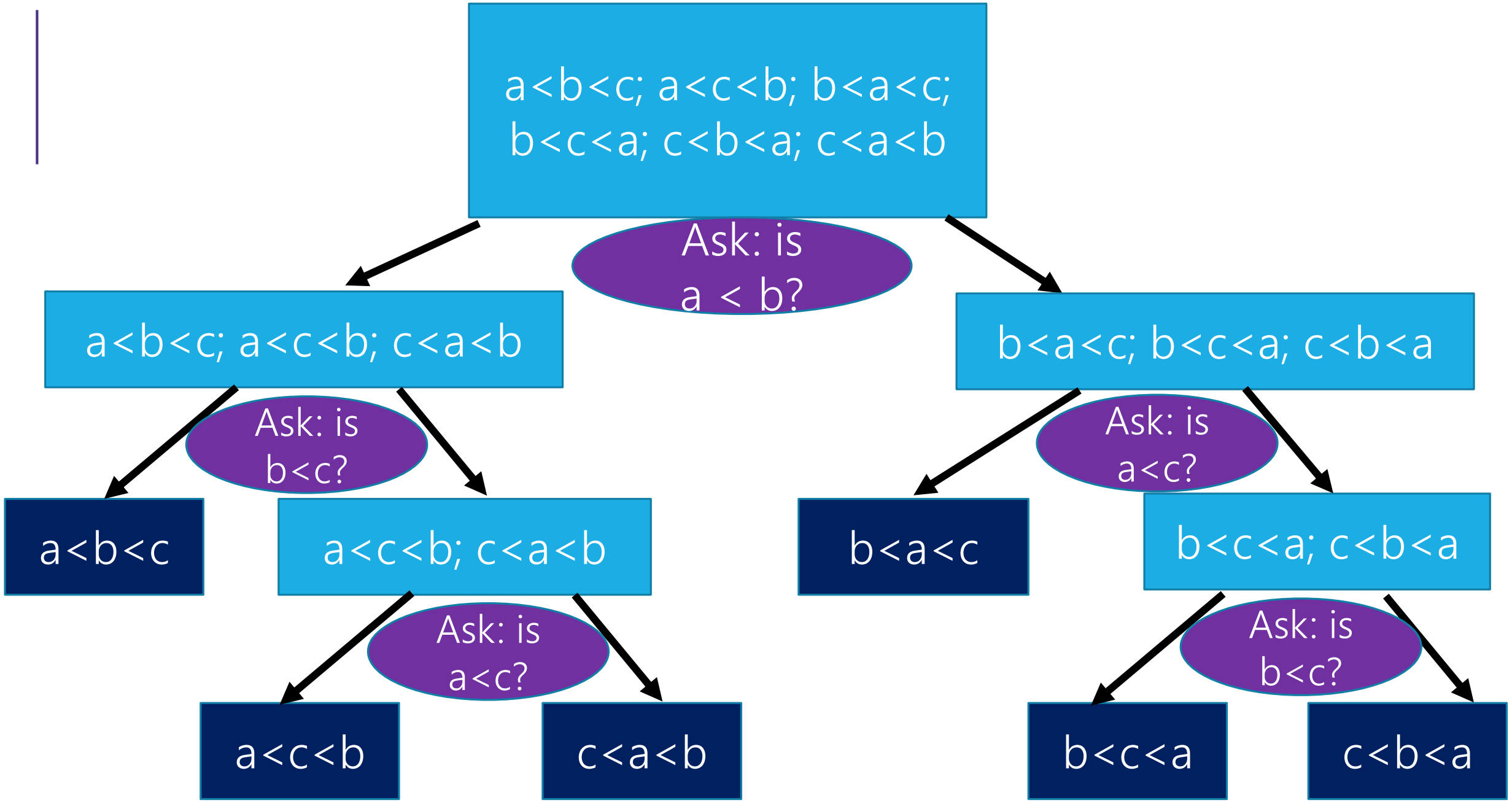
We will figure out which array to return by comparing elements.

When we know what the correct order is, we'll return that array.

In our real algorithm, we're probably moving things around to make the code understandable.

Don't worry about that for the proof.

Whatever tricks we're using to remember what's big and small, it doesn't matter if we don't look first!


a<b<c; a<c<b; b<a<c;
b<c<a; c<b<a; c<a<b
Ask: is
a < b?
a<b<c; a<c<b; c<a<b
b<a<c; b<c<a; c<b<a
Ask: is
b<c?
Ask: is
a<c?
a<b<c
a<c<b; c<a<b
b<a<c
b<c<a; c<b<a
Ask: is
a<c?
Ask: is
b<c?
a<c<b
c<a<b
b<c<a
c<b<a

Complete the Proof

How many operations can we guarantee in the worst case?

How tall is the tree if the array is length n?

What's the simplified $\Omega()$?

Complete the Proof

How many operations can we guarantee in the worst case?

- Equal to the height of the tree.

How tall is the tree if the array is length n?

- One of the children has at least half of the possible inputs.
- What level can we guarantee has an internal node? $\log_2(n!)$

What's the simplified $\Omega()$?

$$\log_2(n!) = \log_2(n) + \log_2(n-1) + \log_2(n-2) + \cdots + \log_2(1)$$

$$\geq \log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2}\right) + \cdots + \log_2\left(\frac{n}{2}\right) \text{ (only } n/2 \text{ copies)}$$

$$\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = n/2(\log_2(n) - 1) = \Omega(n \log n)$$

Takeaways

A tight lower bound like this is **very** rare.

This proof had to argue about every possible algorithm
- that's really hard to do.

We can't come up with a more clever recurrence to sort faster.

This theorem actually says things about data structures
- See exercise 7.

Unless we make some assumptions about our input.

And get information without doing the comparisons.

Avoiding the Lower Bound

Can we avoid using comparisons?

In general probably not.

In your programming projects, definitely not.

But what if we know that all of our data points are small integers?

Bucket Sort (aka Bin Sort)

4	3	1	2	1	1	2	3	4	2

1	2	3	4
3	3	2	2

1	1	1	2	2	2	3	3	4	4

Bucket Sort

Running time?

If we have m possible values and an array of size n?
$O(m+n)$.

How are we beating the lower bound?

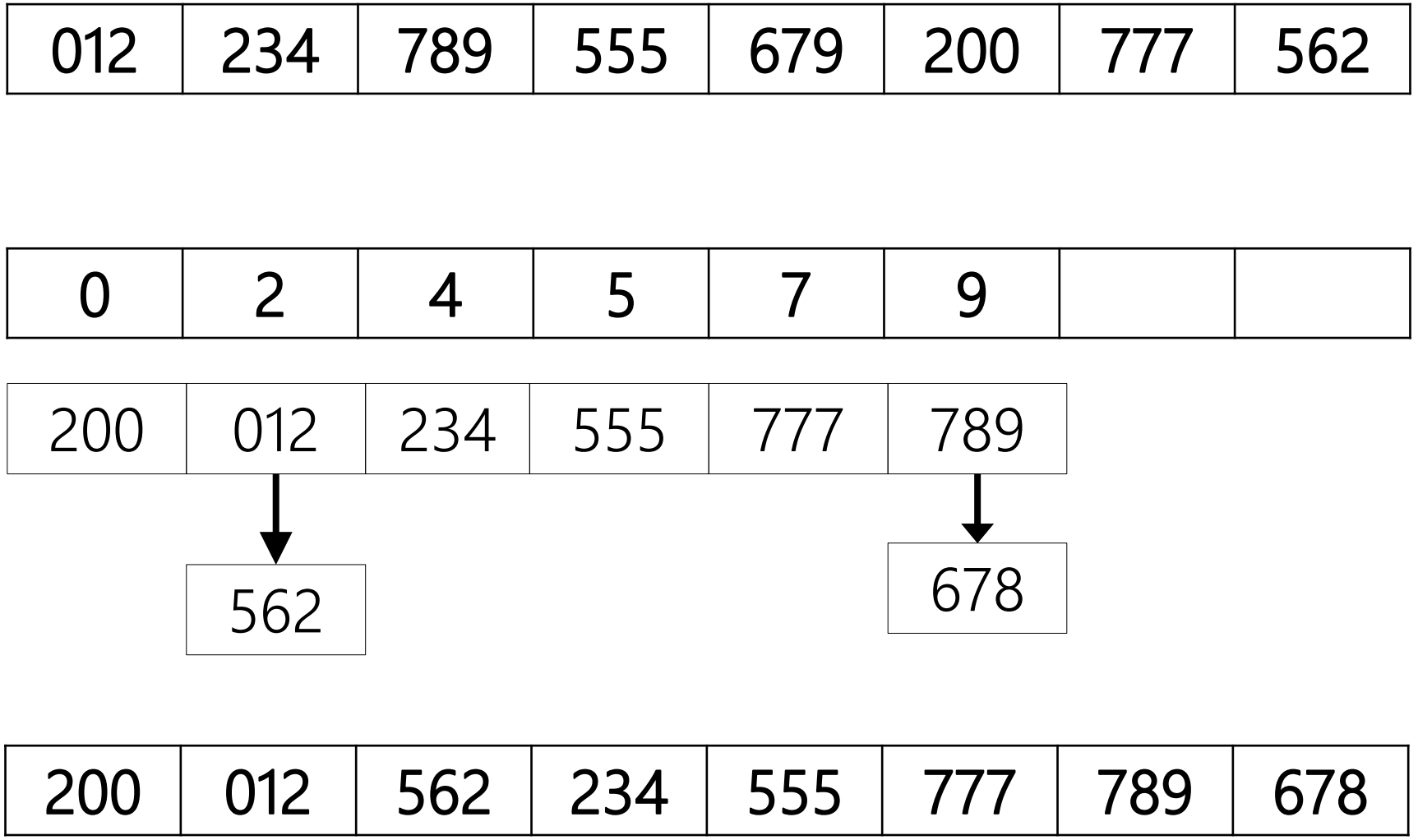
When we place an element, we implicitly compare it to all the others in $O(1)$ time!

Radix Sort

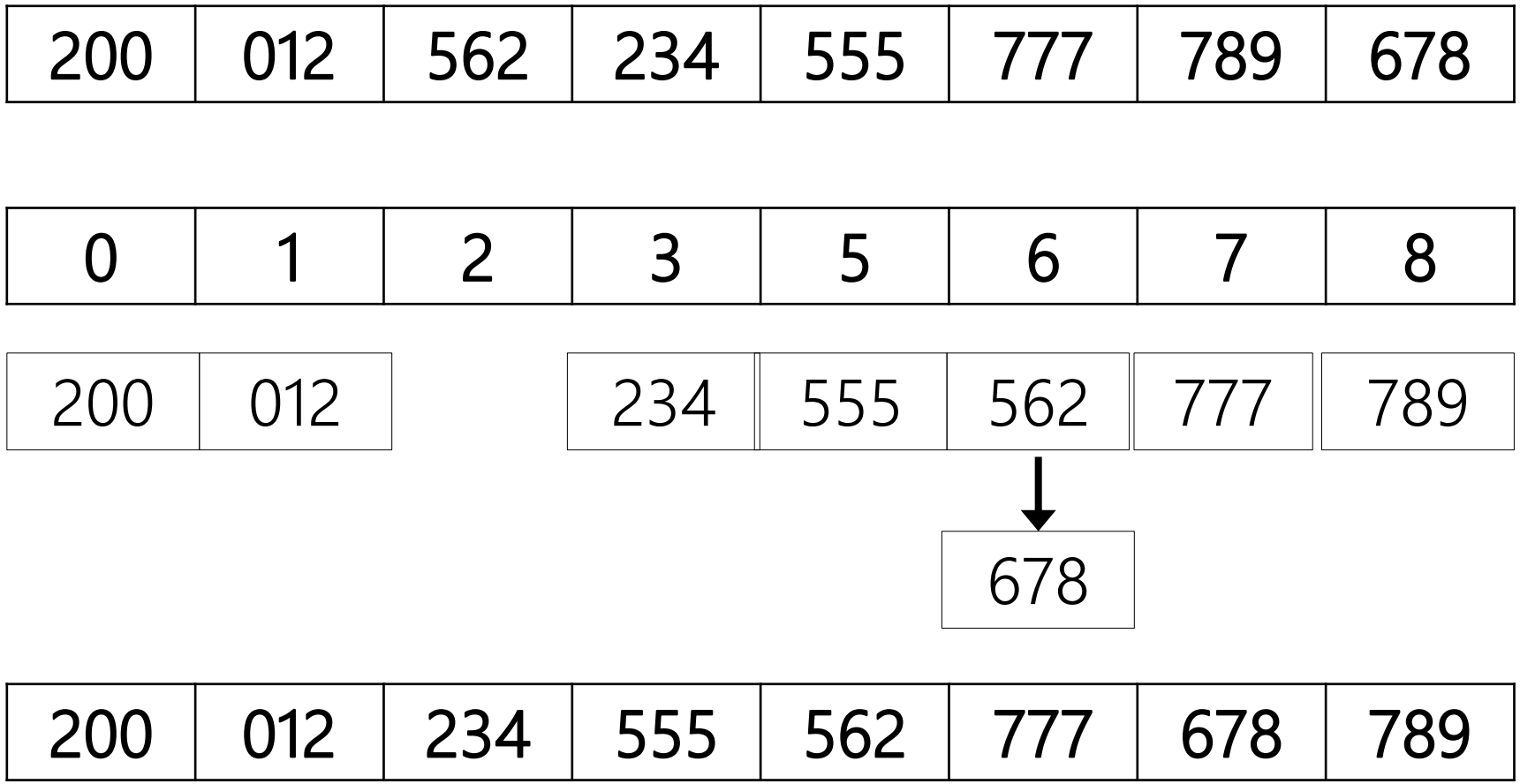
For each digit (starting at the ones place)

- Run a “bucket sort” with respect to that digit
- Keep the sort stable!

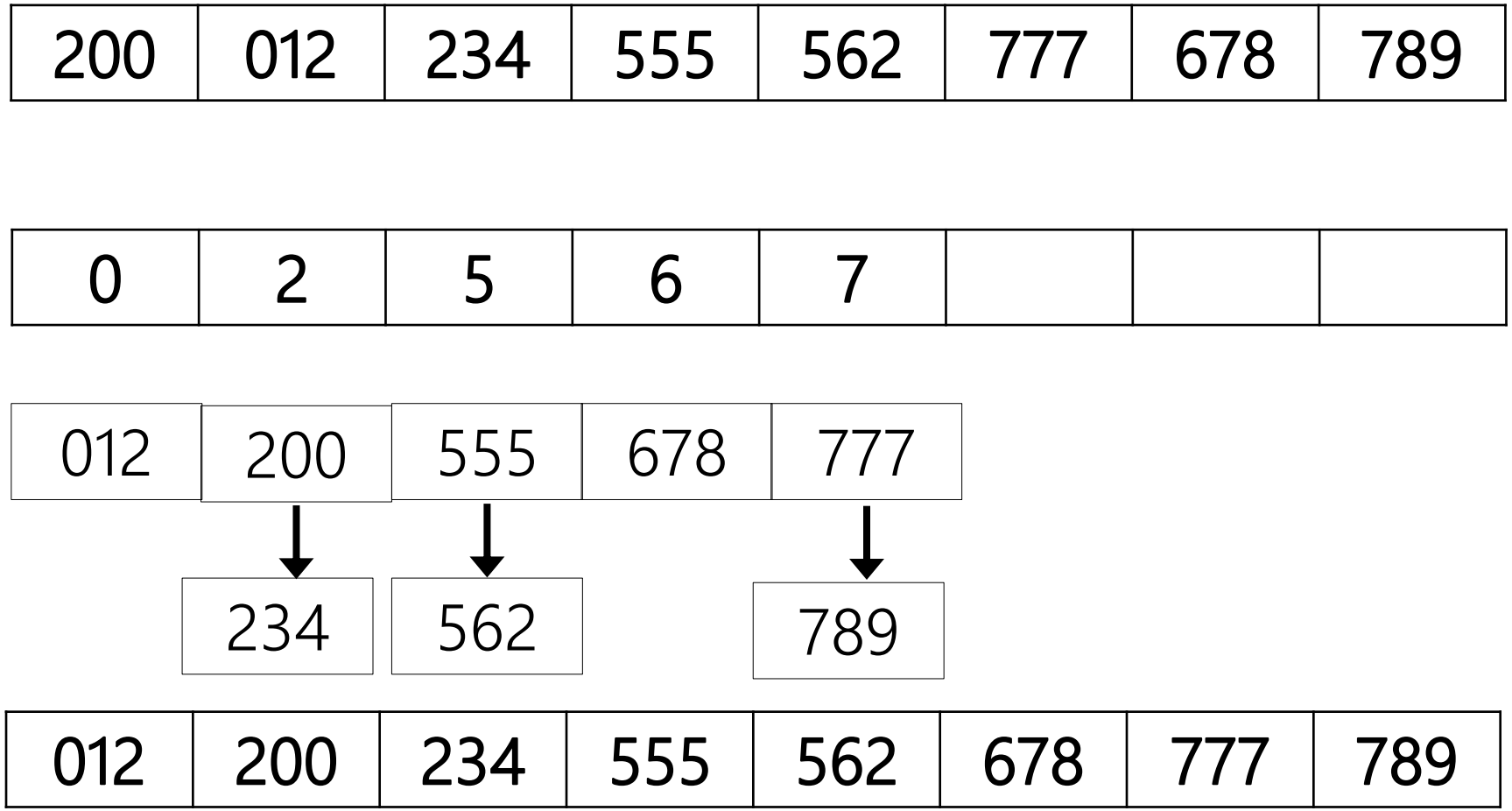
Radix Sort: Ones Place



Radix Sort: Tens Place



Radix Sort: Hundreds Place



Radix Sort

Key idea: by keeping the sorts stable, when we sort by the hundreds place, ties are broken by tens place (then by ones place).

Running time? $O((n + r)d)$

Where d is number of digits in each entry,

r is the radix, i.e. the base of the number system.

How do we avoid the lower bound?

- Same way as bucket sort, we implicitly get free comparison information when we insert into a bucket.

Radix Sort

When can you use it?

ints and strings. As long as they aren’t too large.

Summary

You have a bunch of data. How do you sort it?

Honestly…use your language's default implementation

- It's been carefully optimized.

Unless you really know something about your data, or the situation your in

- Not a lot of extra memory? Use an in place sort.
- Want to sort repeatedly to break ties? Use a stable sort.
- Know your data all falls into a small range? Bucket (or maybe Radix) sort.