# Comparisons Sorts

CSE 332 Spring 2025
Lecture 13

# Announcements

| | Monday | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|---|
| This Week | | | MIDTERM :O | | TODAY<br>Ex 5 due<br>Ex 6 out |
| Next Week | Ex 7 out | | | | Ex 6 due |

Don’t discuss midterm yet (makeups happening next week)

We’ll release solutions once everyone has taken it.

Sorting exercise (re-)hidden, it’ll be out tonight

# Three goals

Three things you might want in a sorting algorithm:

In-Place

- Only use $O(1)$ extra heap memory.
- Sorted array given back in the input array.

Stable

- If a appears before b in the initial array and a.compareTo(b) == 0, then a appears before b in the final array.
- Why? Imagine you sort an array by first name, then sort by last name. With a stable sort you get a list sorted by full name! (With an unstable sort the "Smiths" could go in any order).

Fast

# Insertion Sort

```
for(i from 1 to n-1){
    int index = i
    while(a[index-1] > a[index]){
        swap(a[index-1], a[index])
        index = index-1
    }
}
```

# Selection Sort

Here's another idea for a sorting algorithm:

Maintain a sorted subarray

While(subarray is not full array)

Find the smallest element remaining in the unsorted part.

- By scanning through the remaining array

Insert it at the end of the sorted part.

Running time $O(n^2)$

In-Place: Yes; Stable: Yes.

# Heap Sort

Here's another idea for a sorting algorithm:

Maintain a sorted subarray; **Make the unsorted part a min-heap**

While(subarray is not full array)

Find the smallest element remaining in the unsorted part.

- By calling `removeMin` on the heap

Insert it at the end of the sorted part.

Running time $O(n \log n)$

# Heap Sort (Better)

We're sorting in the wrong order!

- Could reverse at the end.

Our heap implementation will implicitly assume that the heap is on the left of the array.

Switch to a max-heap, and keep the sorted stuff on the right.

What's our running time? $O(n \log n)$

# A Different Idea

So far our sorting algorithms:

- Start with an (empty) sorted array
- Add something to it.

Different idea: Divide And Conquer:

Split up array (somehow)

Sort the pieces (recursively)

Combine the pieces

https://www.youtube.com/watch?v=XaqR3G_NVoo

# Merge Sort Pseudocode

```
mergeSort(input) {
    if (input.length == 1)
        return
    else
        leftHalf = mergeSort(new [0, ..., mid])
        rightHalf = mergeSort(new [mid + 1, ...])
        return merge(leftHalf, rightHalf)
}
```

# How Do We Merge?

Turn two sorted lists into one sorted list:

Start from the small end of each list.

Copy the smaller into the combined list

Move that pointer one spot to the right.

| | | |
|---|---|---|
| 3 | 15 | 27 |

| | | |
|---|---|---|
| 5 | 12 | 30 |

| | | | | | |
|---|---|---|---|---|---|
| 3 | 5 | 12 | 15 | 27 | 30 |

# Merge Sort Analysis

Running Time:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c_1 n & \text{if } n \geq 1 \\ c_2 & \text{otherwise} \end{cases}$$

This is a closed form you will have memorized by the end of the quarter.

The closed form is $\Theta(n \log n)$.

Stable: yes! (if you merge correctly)

In place: no.

# 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

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 3 | 5 | 6 | 9 | 1 | 4 | 7 | 2 | 10 |

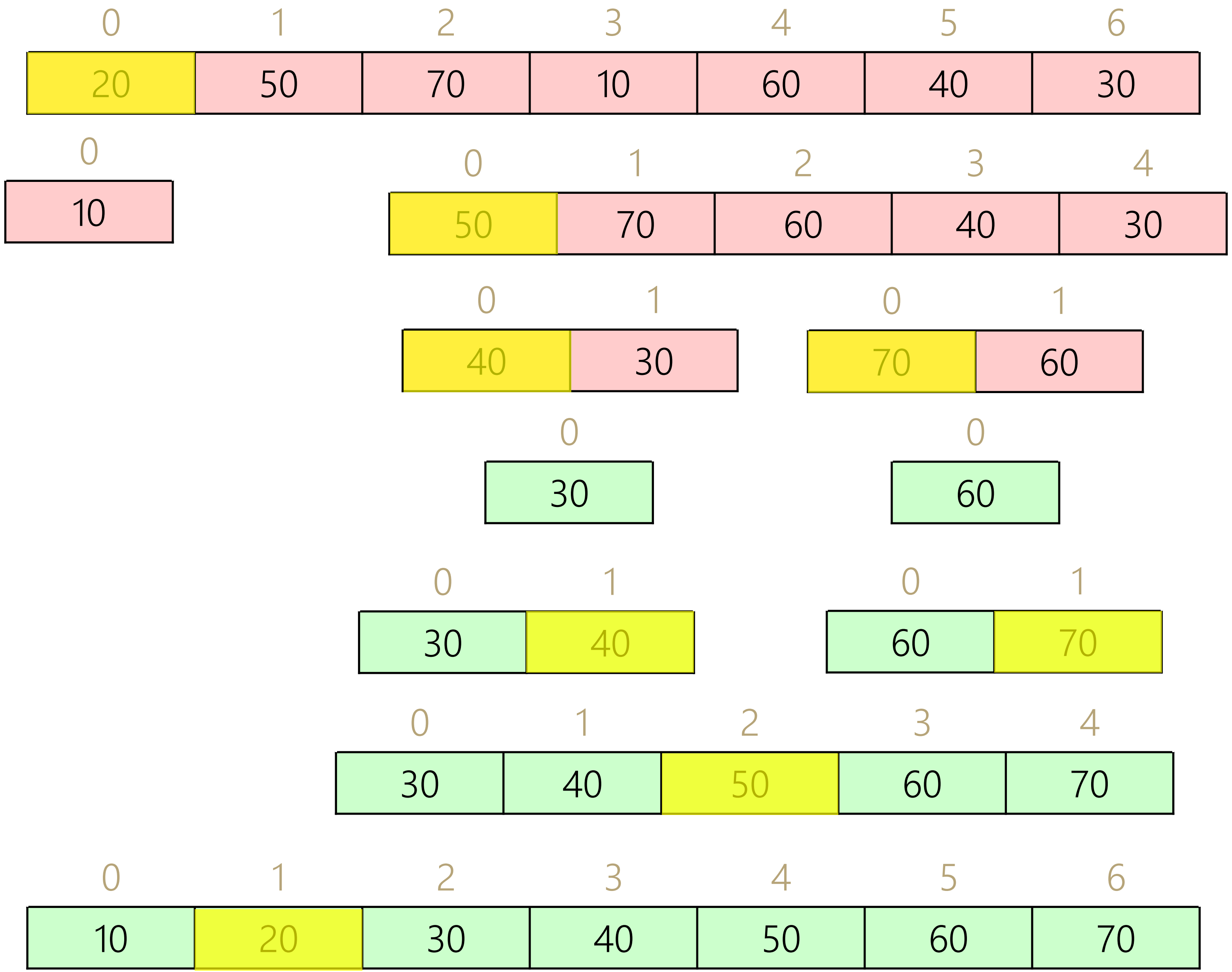
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 3 | 5 | 6 | 2 | 1 | 4 | 7 | 9 | 10 |

$i, j$ 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 $A[i-1]$.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 3 | 5 | 6 | 2 | 1 | 4 | 8 | 9 | 10 |

# 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:
- Worst case:

Recurrences:

Best:

Worst:

Running times:

- Best:
- Worst:

# 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

- 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.

*QuickSort does create $\Theta(\log n)$ memory on the call-stack (maintaining all the recursive calls). Our definition of in-place doesn't count that memory, there's $\Theta(1)$ "heap" memory.

# Constant Factors

Why is QuickSort called QuickSort---the worst case is slower than mergesort!

The constant factors for QuickSort tend to be better on average. Creating new memory is expensive, and it turns out that simple pivot selection rules are VERY likely to avoid worst-case. It really is quick in practice!!

Most practical implementations of recursive sorts set the base case larger than 1, and run an iterative sort (e.g., insertion sort) instead.

- Creating a recursive call is actually pretty expensive---for n=10, say insertion sort is probably faster.
- Big-O is same as the “main” sort, regardless of what you do for the base case.

# Lower Bound

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

- Merge Sort, Heap Sort, Quick Sort with a guaranteed good pivot

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

# Proving the lower-bound

**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.

Our proof will use something called a "decision-tree."

It's a diagram showing the decisions our code will make (think "if-else branches").

We'll argue that any algorithm that takes $o(n \log n)$ time makes a mistake.

# Decision Trees

Suppose we have a size 3 array to sort.

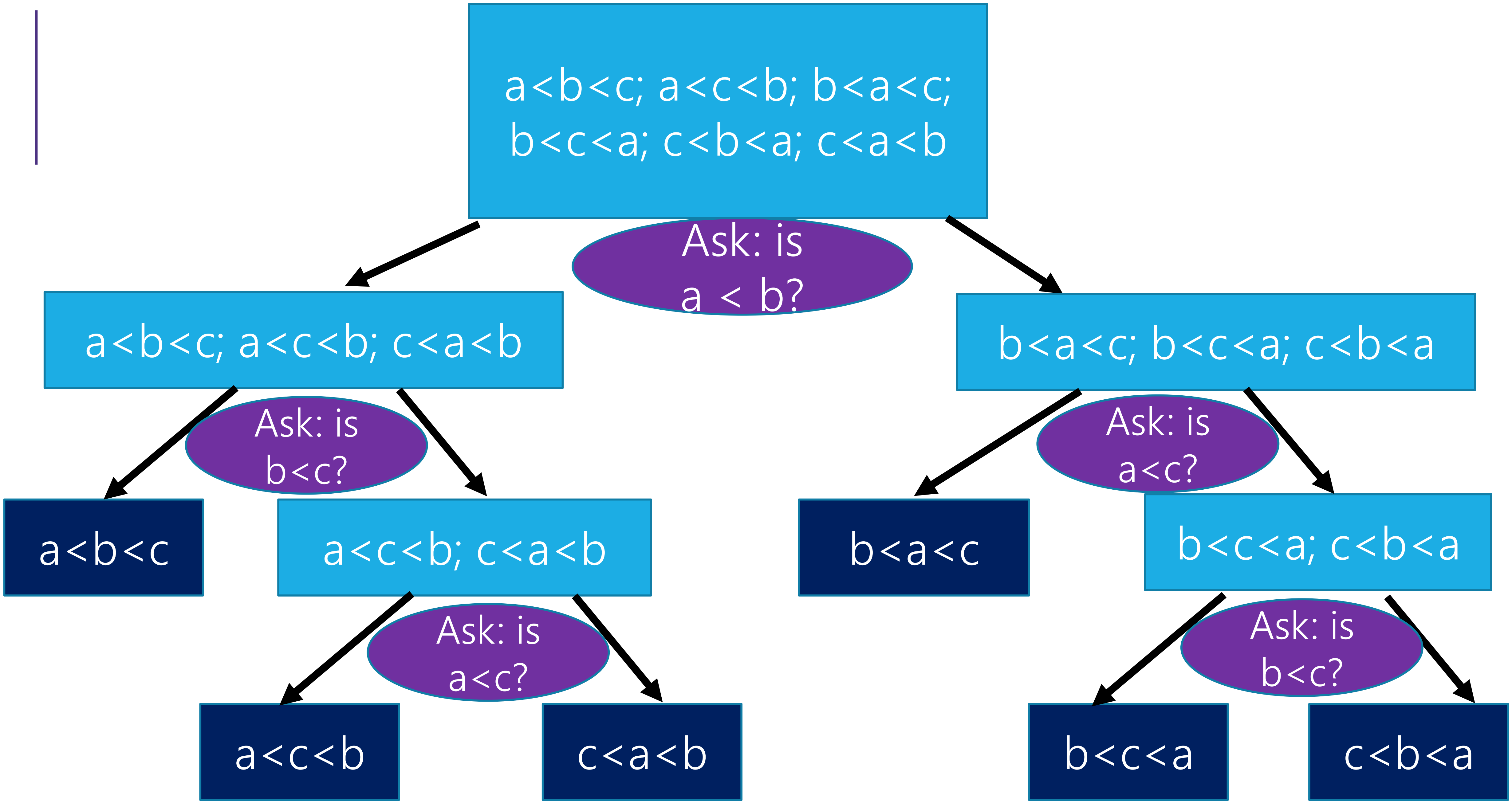
We will figure out which array to return by comparing elements.

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, too!

- You'll prove it yourselves in an upcoming exercise.

Unless we make some assumptions about our input.

And get information without doing the comparisons.

# Wrap-up

# 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.

| Sort | Best-Case | Average-Case | Worst-Case | In-Place? | Stable? | Other Notes |
|---|---|---|---|---|---|---|
| Insertion | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | Yes | Yes | Common choice for small $n$ or structured (maybe already sorted) data |
| Selection | $O(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | Yes | Yes | Many comparisons, but few swaps…mostly educational purposes not practical |
| Heap | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | Yes | No | Best-Case analysis assumes elements are distinct. |
| Merge | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | No | Yes | Common choice when reliability is key, or stability is needed. |
| Quick | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | Yes | No | Common default choice due to great constant factors |
| Bucket | $\Theta(n + m)$ | $\Theta(n + m)$ | $\Theta(n + m)$ | No | Yes | Only works when all entries are ints between $0$ and $m$; non-comparison based. |
| Radix | $\Theta(n(r + d))$ | $\Theta(n(r + d))$ | $\Theta(n(r + d))$ | No | Yes | Only works when entries are $d$ digits long in base $r$. non-comparison based. |

# Non-Comparison Sorts

# Avoiding the Lower Bound

Can we avoid using comparisons?

In general, probably not.

- If you're trying to write the most general code, 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 input 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!

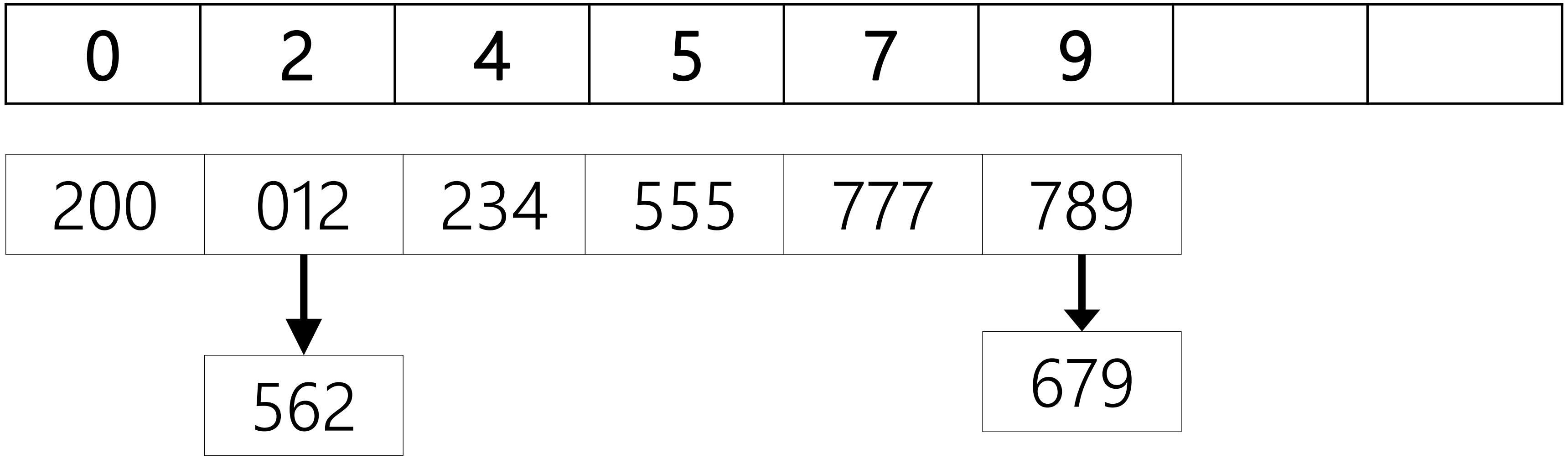
# Radix Sort

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 012 | 234 | 789 | 555 | 679 | 200 | 777 | 562 |




Radix Sort by Ones Place
Keep Stable: Later elements go at <u>end</u> of linked list

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 200 | 012 | 562 | 234 | 555 | 777 | 789 | 679 |

Copy back in new order.

# Radix Sort: Tens Place

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 200 | 012 | 562 | 234 | 555 | 777 | 789 | 679 |

| 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 200 | 012 | | 234 | 555 | 562 | 777 | 789 |
| | | | | | | 679 | |

Radix Sort by Tens Place
Keep Stable: Later elements go at <u>end</u> of linked list

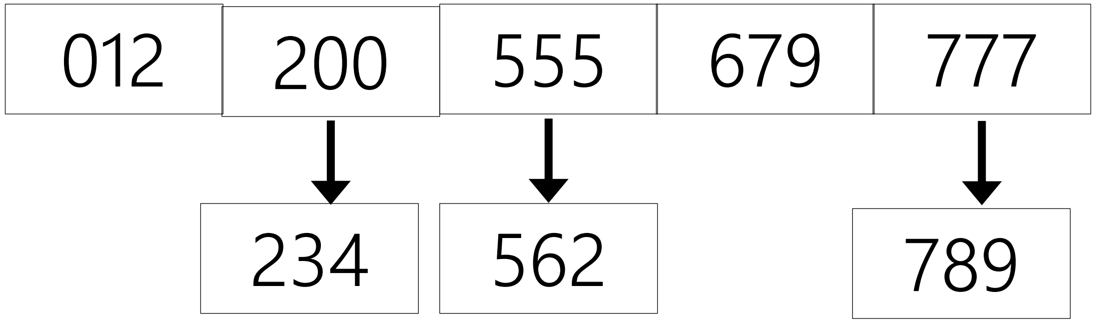
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 200 | 012 | 234 | 555 | 562 | 777 | 679 | 789 |

Copy back in new order. Sorted by tens, then ones.

# Radix Sort: Hundreds Place

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 200 | 012 | 234 | 555 | 562 | 777 | 679 | 789 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 5 | 6 | 7 | | | |




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 012 | 200 | 234 | 555 | 562 | 679 | 777 | 789 |

# 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.