Sorting Lower Bound and Non-Comparison Sorts

Data Structures and Parallelism

Announcements

P1 feedback coming soon.

Make sure to look at it before you finish P2.

You (and your partner) can use up to two late days for P2

Writeup: Timing vs. Counting handout on webpage.

- Useful for doing the P2 writeup

Comparison Sorts

Insertion Sort
- Best-case: $O(n)$ Worst: $O(n^2)$ Avg: $O(n^2)$

Selection Sort
- Best-case: $O(n)$ Worst: $O(n^2)$ Avg: $O(n^2)$

Heap Sort
- Best-case: $O(n)^*$ Worst: $O(n\log(n))$ Avg: $O(n\log(n))$

Merge Sort
- Best-case: $O(n\log(n))$ Worst: $O(n\log(n))$ Avg: $O(n\log(n))$

Quick Sort
- Best-case: $O(n\log(n))$ Worst: $O(n^2)$ Avg: $O(n\log(n))$

(* only if all input values are the same)

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.

A Different View of Sorting

Assume we have *n* elements to sort

- And for simplicity, none are equal (no duplicates)

How many ***permutations*** (possible orderings) of the elements?

Example, *n*=3,

A Different View of Sorting

Assume we have *n* elements to sort

- And for simplicity, none are equal (no duplicates)

How many ***permutations*** (possible orderings) of the elements?

Example, *n*=3, six possibilities

a[0]<a[1]<a[2]	a[0]<a[2]<a[1]	a[1]<a[0]<a[2]
a[1]<a[2]<a[0]	a[2]<a[0]<a[1]	a[2]<a[1]<a[0]

In general, *n* choices for least element, then *n*-1 for next, then *n*-2 for next, …

- *n*(*n*-1)(*n*-2)…(2)(1) = *n*! possible orderings

Describing every comparison sort

A different way of thinking of sorting is that the sorting algorithm has to "find" the right answer among the n! possible answers

- Starts "knowing nothing", "anything is possible"
- Gains information with each comparison, eliminating some possiblities
  - Intuition: At best, each comparison can eliminate half of the remaining possibilities
- In the end narrows down to a single possibility

Counting Comparisons

Don't know what the algorithm is, but it cannot make progress without doing comparisons

- Eventually does a first comparison "is $a < b$?"
- Can use the result to decide what second comparison to do
- Etc.: comparison *k* can be chosen based on first *k-1* results

What is the first comparison in:

- Insertion Sort?
- Selection Sort?
- Mergesort?
- Quicksort?

Counting Comparisons

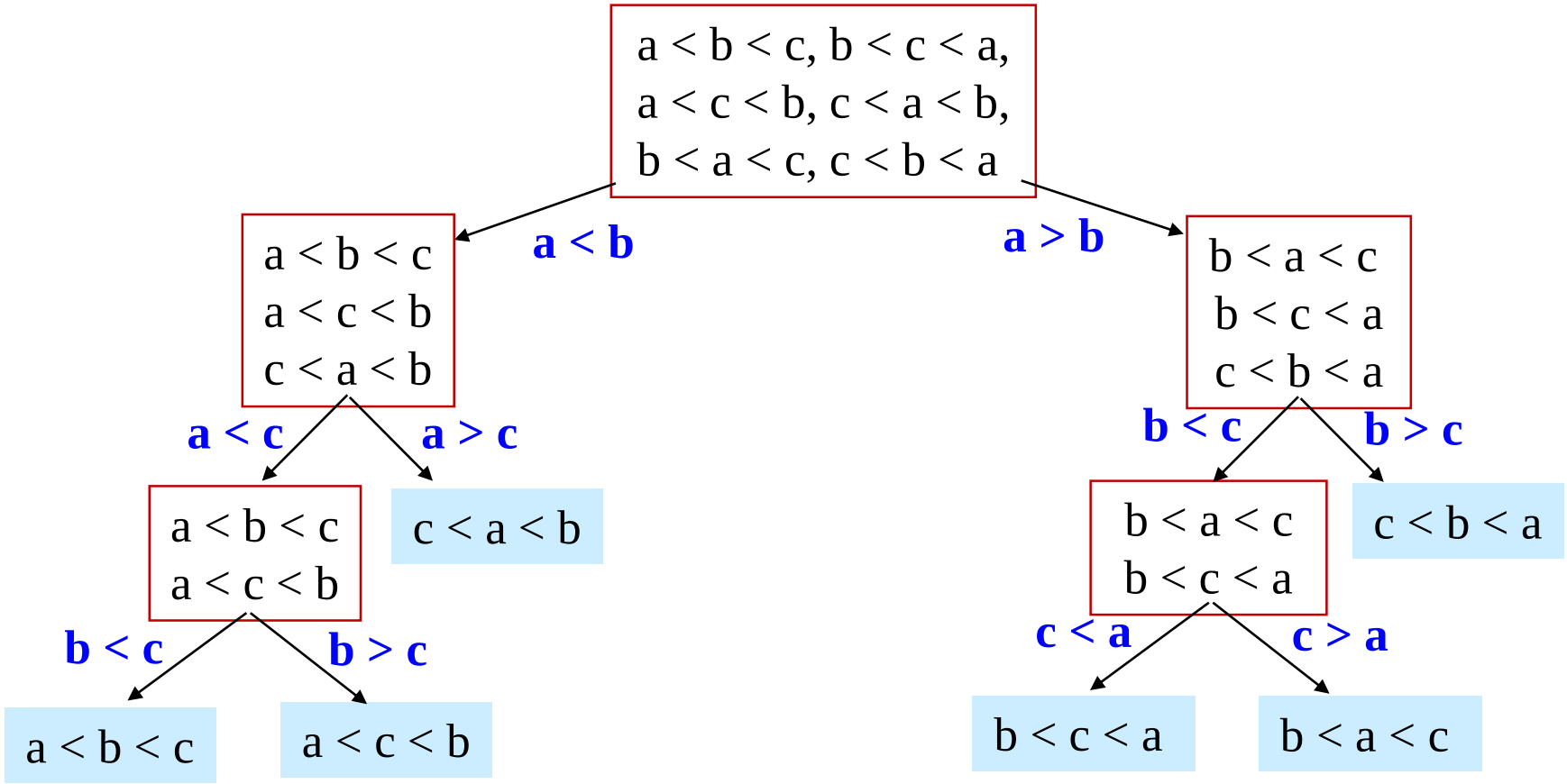
Don't know what the algorithm is, but it cannot make progress without doing comparisons

- Eventually does a first comparison "is $a < b$?"
- Can use the result to decide what second comparison to do
- Etc.: comparison *k* can be chosen based on first *k-1* results

Can represent this process as a *decision tree*

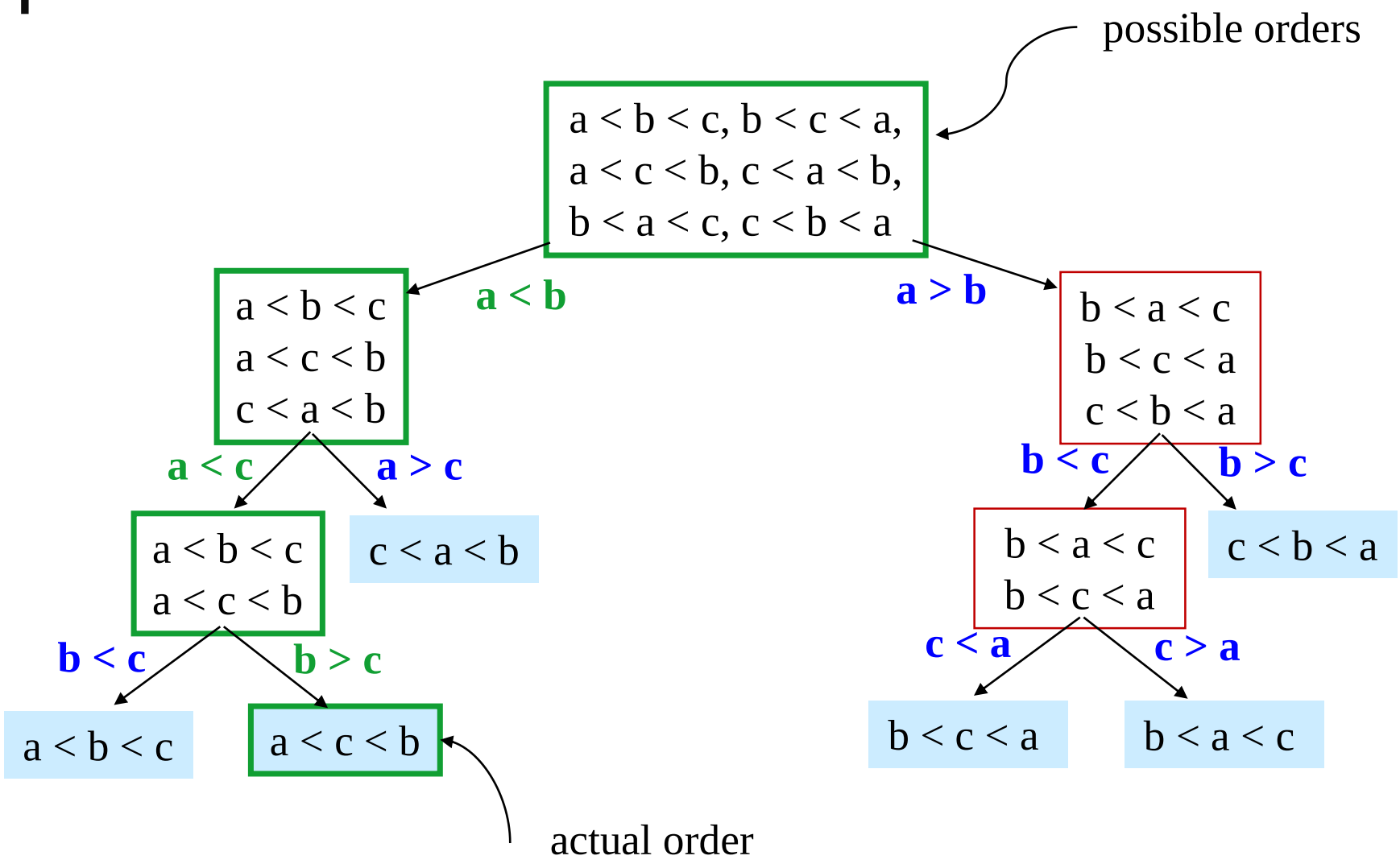
- Nodes contain "set of remaining possibilities"
- At root, anything is possible; no option eliminated
- Edges are "answers from a comparison"
- The algorithm does not actually build the tree; it's what our *proof* uses to represent "the most the algorithm could know so far" as the algorithm progresses

One Decision Tree for n=3



- The leaves contain all the possible orderings of a, b, c
- A different algorithm would lead to a different tree

Example if a < c < b



What the decision tree tells us

A *binary* tree because each comparison has 2 outcomes

- Perform only comparisons between 2 elements; binary result
  - Ex: Is a<b? Yes or no?
- We assume no duplicate elements
- Assume algorithm doesn't ask redundant questions

Because any data is possible, any algorithm needs to ask enough questions to produce all *n*! answers

- Each answer is a different leaf
- So the tree must be big enough to have *n*! leaves
- Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree with *n*! leaves
- So no algorithm can have worst-case running time better than the height of a tree with *n*! leaves
  - Worst-case number-of-comparisons for an algorithm is an input leading to a longest path in algorithm's decision tree

Where are we

Proven: No comparison sort can have worst-case running time better than: the height of a binary tree with *n*! leaves

- Turns out average-case is same asymptotically
- A comparison sort could be worse than this height, but it cannot be better
- Fine, *how tall is a binary tree with n! leaves?*

Now: Show that a binary tree with n! leaves has height $\Omega(n \log n)$

- That is, n log n is the lower bound, the height must be at least this, could be more, (in other words your comparison sorting algorithm could take longer than this, but it won't be faster)
- Factorial function grows very quickly

Then we'll conclude that: (Comparison) Sorting is $\Omega(n \log n)$

- This is an amazing computer-science result: proves all the clever programming in the world can't sort in linear time!

Lower bound on Height

A binary tree of height h has **at most** *how many* leaves?

L ≤ _____

A binary tree with L leaves has height **at least**:

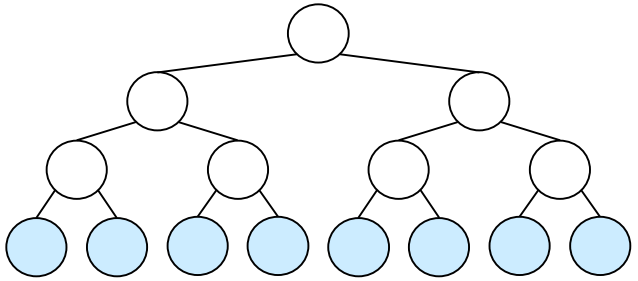
h ≥ _____

The decision tree has how many leaves: _____

So the decision tree has height:

h ≥ _____

Lower bound on height



The height of a binary tree with L leaves is at least $\log_2 L$

So the height of our decision tree, h:

$h \geq \log_2 (n!)$ property of binary trees

$= \log_2 (n*(n-1)*(n-2)...(2)(1))$ definition of factorial

$= \log_2 n + \log_2 (n-1) + ... + \log_2 1$ property of logarithms

$\geq \log_2 n + \log_2 (n-1) + ... + \log_2 (n/2)$ keep first n/2 terms

$\geq (n/2) \log_2 (n/2)$ each of the n/2 terms left is $\geq \log_2 (n/2)$

$= (n/2)(\log_2 n - \log_2 2)$ property of logarithms

$= (1/2)n\log_2 n - (1/2)n$ arithmetic

"=" $\Omega(n \log n)$

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	1	1	2	2	2	3	3	4	4

1	2	3	4

Bucket Sort (aka Bin Sort)

4	3	1	2	1	1	2	3	4	2

1	1	1	2	2	2	3	3	4	4

1	2	3	4
3	3	2	2

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?

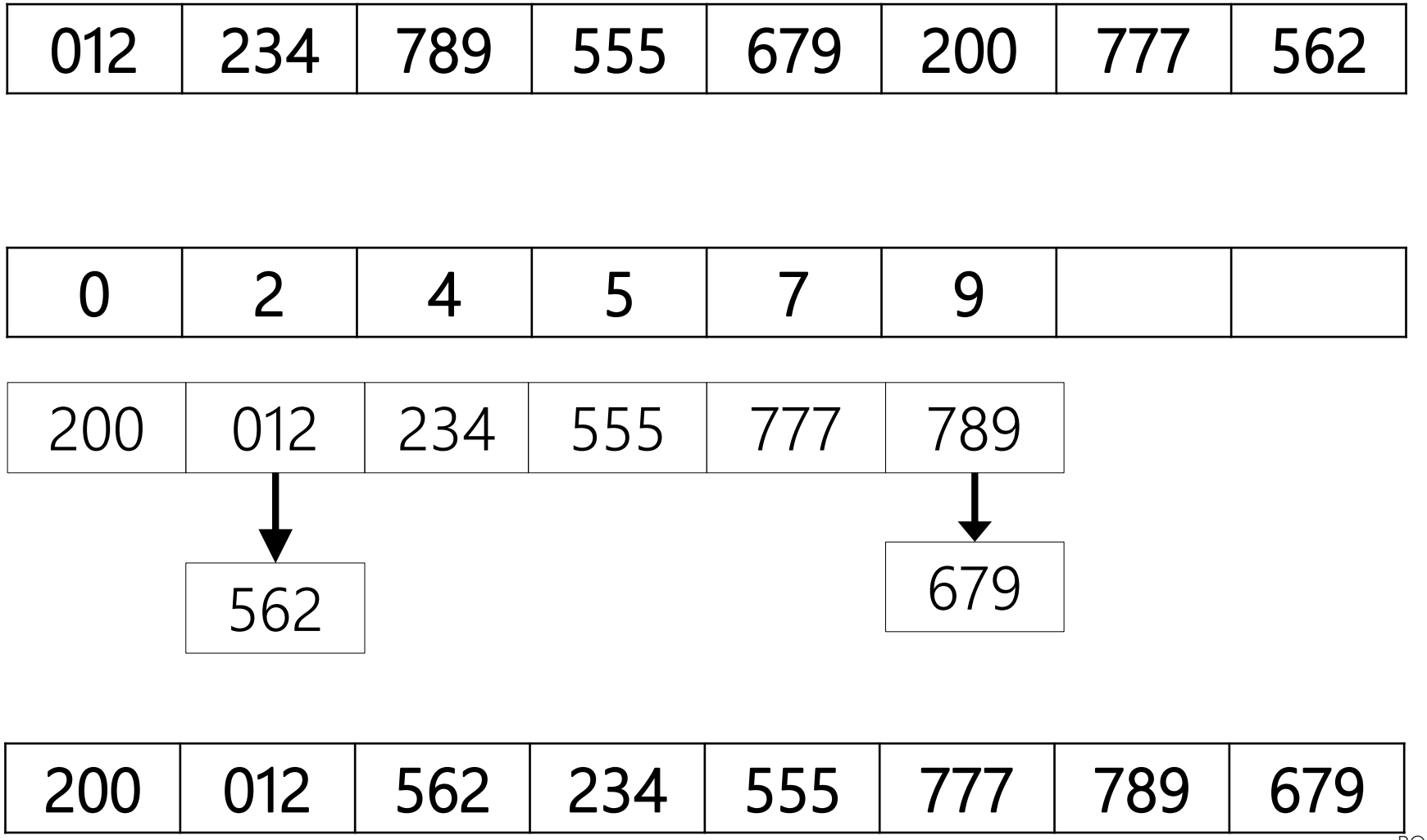
When we place an element, we implicitly compare it to all the others in $O(1)$ time!

Radix Sort

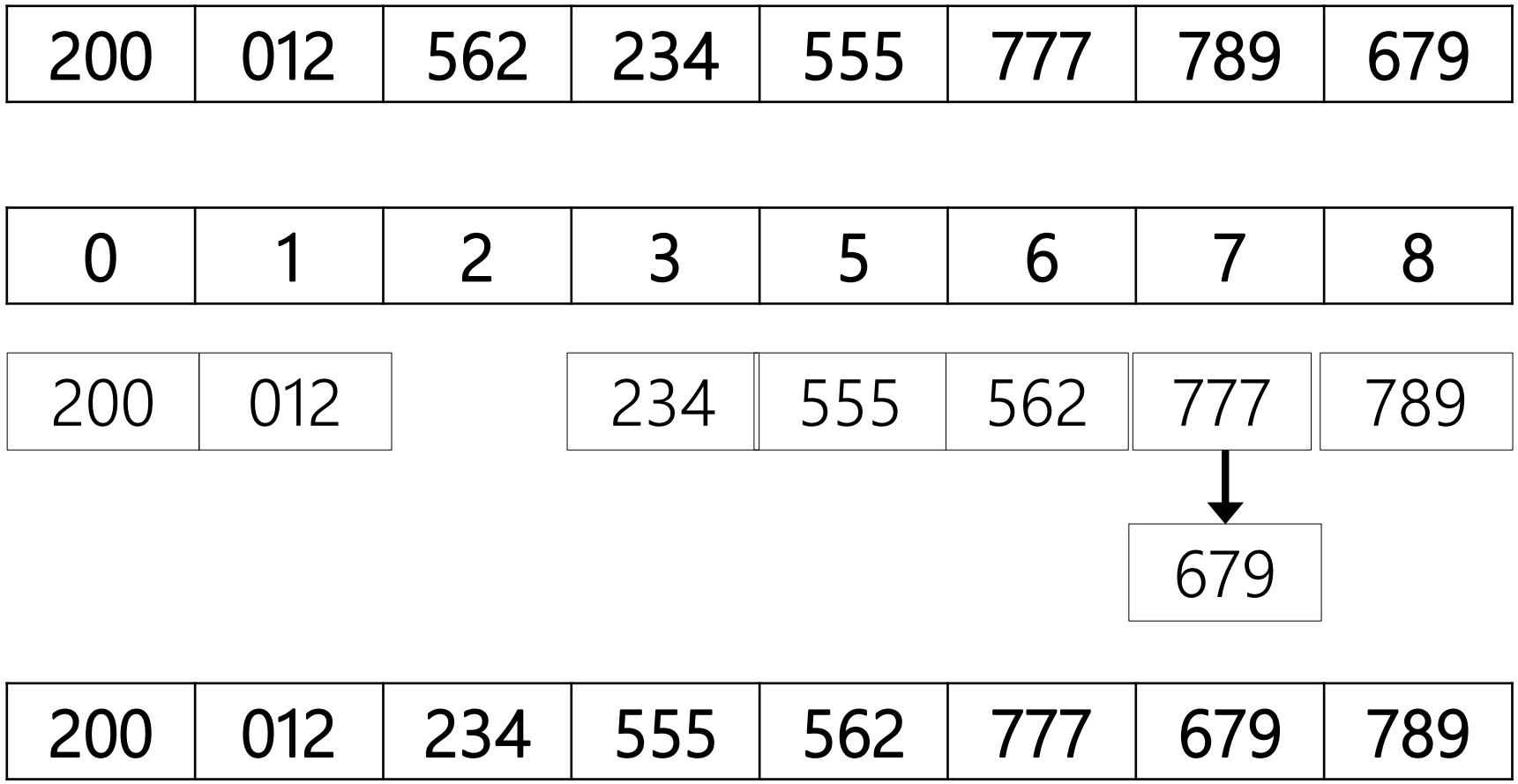
For each digit (starting at the ones place)

- Run a “bucket sort” with respect to that digit\
- Use a separate chaining hash table as your bucket array
- Keep the sort stable!

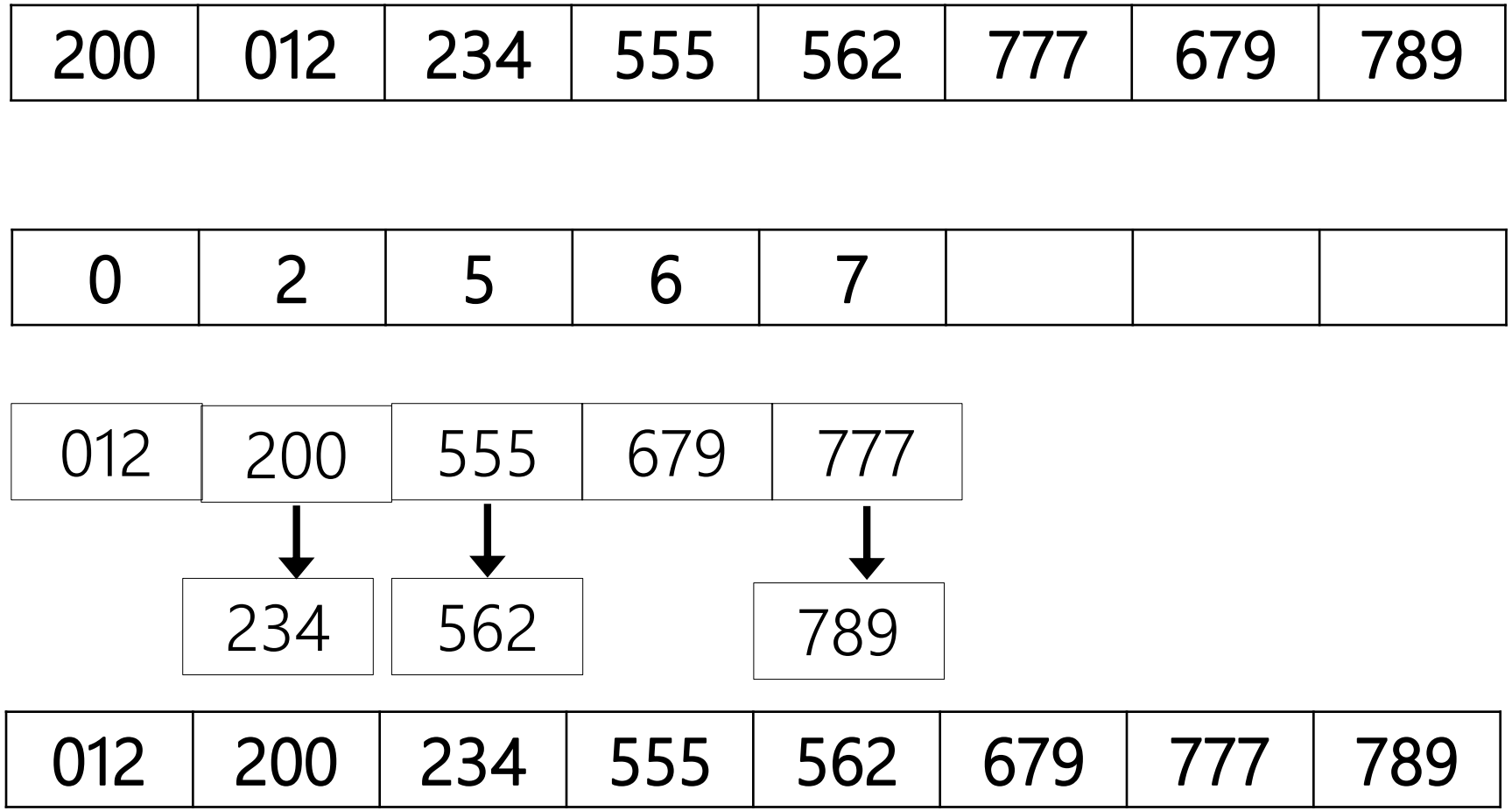
Radix Sort: Ones Place



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.