



CSE 332: Data Structures & Parallelism

Lecture 15: Beyond Comparison Sorting

Richard Jiang
Spring 2020

Today

- Sorting
 - The Lower Bound of Comparison Sorting
 - Bucket Sort
 - Radix Sort

The Big Picture

**Simple
algorithms:
 $O(n^2)$**

Insertion sort
Selection sort
Shell sort
...

**Fancier
algorithms:
 $O(n \log n)$**

Heap sort
Merge sort
Quick sort (avg)
...

**Comparison
lower bound:
 $\Omega(n \log n)$**

**Specialized
algorithms:
 $O(n)$**

Bucket sort
Radix sort

**Handling
huge data
sets**

**External
sorting**

How fast can we sort?

- Heapsort & mergesort have $O(n \log n)$ worst-case running time
- Quicksort has $O(n \log n)$ average-case running times
- These bounds are all tight, actually $\Theta(n \log n)$
- So maybe we need to dream up another algorithm with a lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$
 - Instead: *prove* that this is *impossible*
 - *Assuming* our comparison *model*: The only operation an algorithm can perform on data items is a 2-element comparison

A Different View of Sorting

- Assume we have n elements to sort [a, b, c...]
 - And for simplicity, none are equal (no duplicates)
- **Sorting** is like finding one specific ordering out of all possible ordering of elements
- How many **permutations** (possible orderings) are there?
 - Example, $n=3$

A Different View of Sorting

- Assume we have n elements to sort [a, b, c...]
 - And for simplicity, none are equal (no duplicates)
- **Sorting** is like finding one specific ordering out of all possible ordering of elements
- How many **permutations** (possible orderings) are there?
 - Example, $n=3$

$a < b < c$	$a < c < b$	$b < a < c$
$b < c < a$	$c < a < b$	$c < b < a$
- In general, n choices for the first element, then $n-1$ for next, then $n-2$ for next, ...
 - $n(n-1)(n-2)\dots(2)(1) = n!$ possible orderings

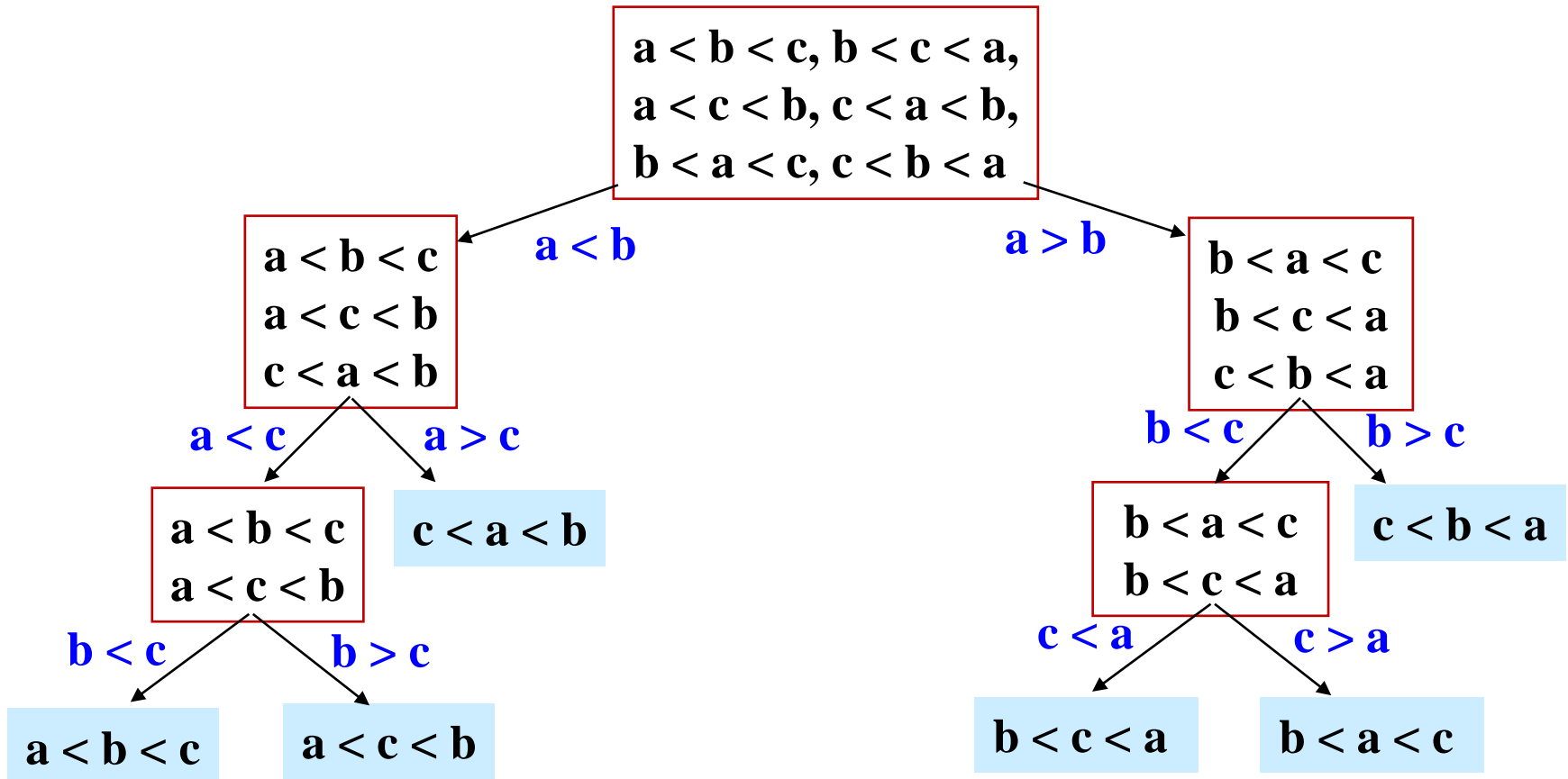
Describing every comparison sort

- A different way of thinking of sorting is that the sorting algorithm has to “find” the right order among the $n!$ possible answers
 - Starts “knowing nothing”, “anything is possible”
 - Gains information with each comparison, eliminating some possibilities
 - Comparisons are binary, $a < b$ or $b < a$
 - Intuition: At best, each comparison can eliminate half of the remaining possibilities
 - In the end narrows down to a single possibility
- Where are the comparisons in:
 - Insertion Sort?
 - Quicksort?

Representing Comparison Sorts

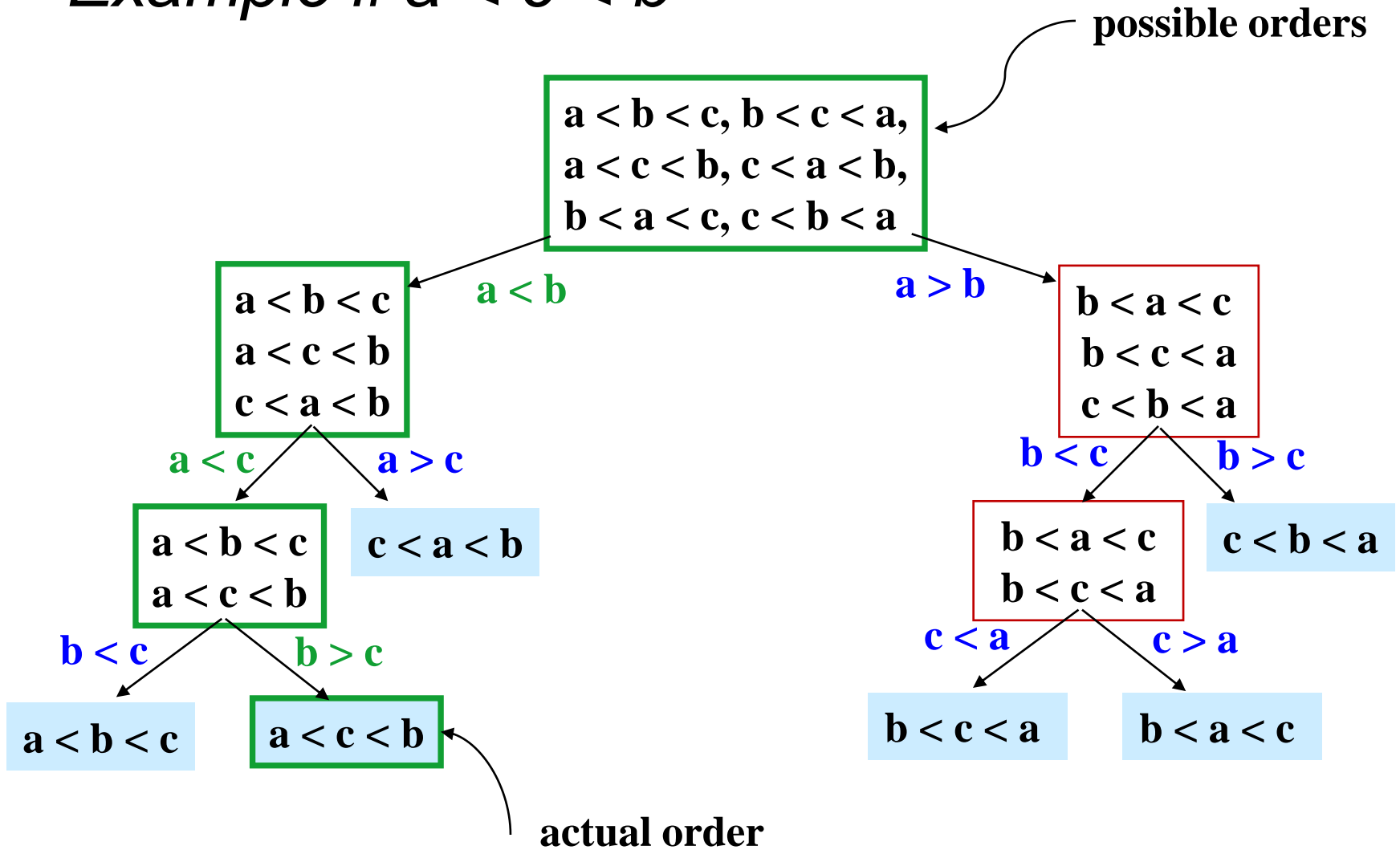
- Let's represent these binary comparisons as a binary tree!
- Called a *Decision Tree*
 - Nodes contain a “set of remaining possible orderings”
 - The root contains all possible orderings, anything is possible
 - The leaves contain one specific ordering
 - Edges are “answers from a comparison”
 - The 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

- Because any order is possible, any algorithm needs to ask enough questions to produce all $n!$ answers
 - Each answer/ordering is a different leaf
 - So the binary 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
 - The length of this path is the number of comparison operations needed
 - So no algorithm can have worst-case running time better than the height of a complete 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

Lower bound on Height

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

$$L \leq 2^h$$

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

$$h \geq \log_2 L$$

- The decision tree has how many leaves: _____

- So the decision tree has height:

$$h \geq \underline{\hspace{10em}}$$

Lower bound on Height

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

$$L \leq 2^h$$

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

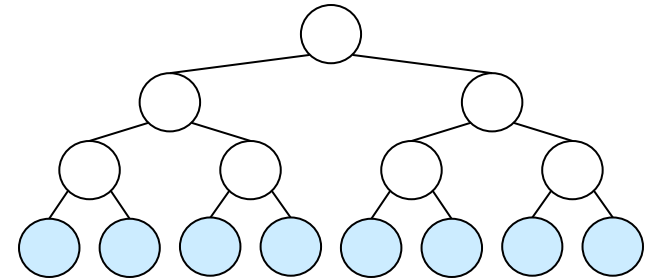
$$h \geq \log_2 L$$

- The decision tree has how many leaves: $N!$

- So the decision tree has height:

$$h \geq \log_2 N!$$

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 :

$$h \geq \log_2 (n!)$$

$$= \log_2 (n \cdot (n-1) \cdot (n-2) \dots (2)(1))$$

$$= \log_2 n + \log_2 (n-1) + \dots + \log_2 1$$

$$\geq \log_2 n + \log_2 (n-1) + \dots + \log_2 (n/2)$$

$$\geq (n/2) \log_2 (n/2) \quad \text{each of the } n/2 \text{ terms left is } \geq \log_2 (n/2)$$

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

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

$$\text{"=" } \Omega (n \log n)$$

property of binary trees

definition of factorial

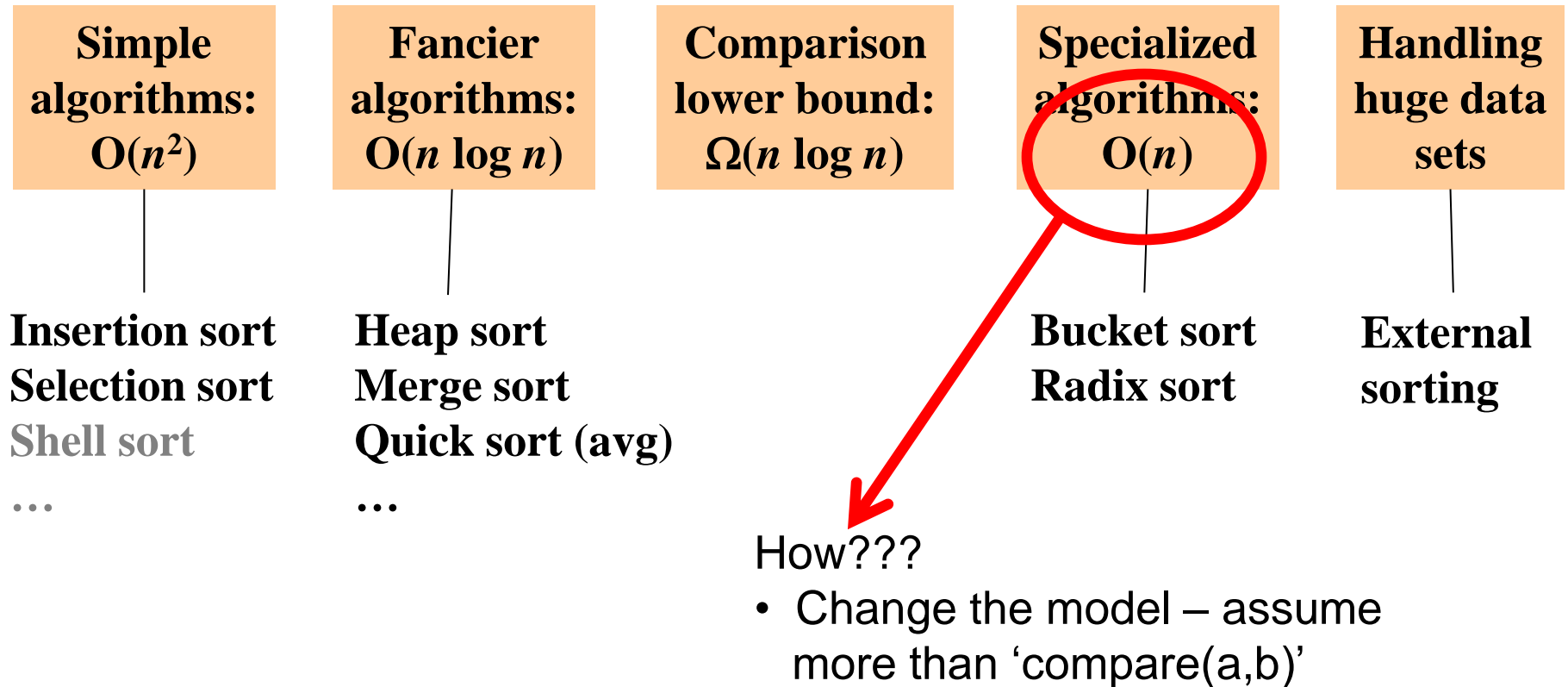
property of logarithms

keep first $n/2$ terms

property of logarithms

arithmetic

The Big Picture



BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and K (or any small range),
 - Create an array of size K , and put each element in its proper bucket (a.k.a. bin)
 - If data is only integers, no need to store more than a *count* of how many times that bucket has been used
- Output result via linear pass through array of buckets

count array	
1	
2	
3	
4	
5	

- Example:
K=5
input (5,1,3,4,3,2,1,1,5,4,5)
output: _____

BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and K (or any small range),
 - Create an array of size K , and put each element in its proper bucket (a.k.a. bin)
 - If data is only integers, no need to store more than a *count* of how many times that bucket has been used
- Output result via linear pass through array of buckets

count array	
1	3
2	1
3	2
4	2
5	3

- Example:

$K=5$

input (5,1,3,4,3,2,1,1,5,4,5)

output: 1,1,1,2,3,3,4,4,5,5,5

What is the running time?

How did the model change?

Analyzing bucket sort

- Overall: $O(n+K)$
 - Linear in n , but also linear in K
 - $\Omega(n \log n)$ lower bound does not apply because **this is not a comparison sort**
- Good when range, K , is smaller (or not much larger) than n
 - (We don't spend time doing lots of comparisons of duplicates!)
- Bad when K is much larger than n
 - Wasted space; wasted time during final linear $O(K)$ pass
- For data in addition to integer keys, use list at each bucket

Bucket Sort with Data

- Most real lists aren't just #'s; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, place at end $O(1)$ (keep pointer to last element)

count array	
1	→ Rocky V
2	
3	→ Harry Potter
4	
5	→ Casablanca → Star Wars

- Example: Movie ratings:
1=bad,... 5=excellent
- Input=
 - 5: Casablanca
 - 3: Harry Potter movies
 - 1: Rocky V
 - 5: Star Wars

Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
This result is **stable**; Casablanca still before Star Wars

Radix sort

- Radix = “the base of a number system”
 - Examples will use 10 because we are used to that
 - In implementations use larger numbers
 - For example, for ASCII strings, might use 128
- Idea:
 - Bucket sort on one digit at a time
 - Number of buckets = radix
 - Starting with *least* significant digit, sort with Bucket Sort
 - Keeping sort *stable*
 - Do one pass per digit
- **Invariant:** After k passes, the last k digits are sorted
- Aside: Origins go back to the 1890 U.S. census

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9

Input: 333
143
591
65
332
491

First pass:

1. bucket sort by ones digit
2. Iterate thru and collect into a list
 - List is sorted by first digit

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	591 491	332	333 143		65				

Input: 333
143
591
65
332
491

First pass:

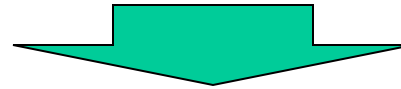
1. bucket sort by ones digit
2. Iterate thru and collect into a list
 - List is sorted by first digit

Order now: 591
491
332
333
143
65

Example

0	1	2	3	4	5	6	7	8	9
	591	332	333		65				
	491		143						

Radix = 10



0	1	2	3	4	5	6	7	8	9

Order now: 591

491

332

333

143

65

Second pass:

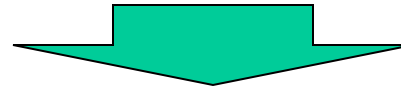
stable bucket sort by tens digit

If we chop off the 100's place,
these #s are sorted

Example

0	1	2	3	4	5	6	7	8	9
	591	332	333		65				
	491		143						

Radix = 10



0	1	2	3	4	5	6	7	8	9
			332	143		65			591
			333						491

Order now: 591
491
332
333
143
65

Second pass:

stable bucket sort by tens digit

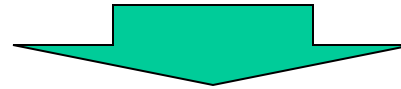
If we chop off the 100's place,
these #s are sorted

Order now: 332
333
143
65
591
491

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
			332	143		65			591
			333						491



0	1	2	3	4	5	6	7	8	9
65	143		332	491	591				
			333						

Order now: 332

333

143

65

591

491

Order now: 65

143

332

333

491

591

Third pass:

stable bucket sort by 100s digit

Only 3 digits: We're done!

Analysis of Radix Sort

Performance depends on:

- Input size: n
- Number of buckets = Radix: B
 - e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = “Digits”: P
 - e.g. Ages of people: 3; Phone #: 10; Person’s name: ?
- Work per pass is 1 bucket sort: _____
 - Each pass is a Bucket Sort
- Total work is _____
 - We do ‘P’ passes, each of which is a Bucket Sort

Analysis of Radix Sort

Performance depends on:

- Input size: n
- Number of buckets = Radix: B
 - e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = “Digits”: P
 - e.g. Ages of people: 3; Phone #: 10; Person’s name: ?
- Work per pass is 1 bucket sort: $O(B+n)$
 - Each pass is a Bucket Sort
- Total work is $O(P(B+n))$
 - We do ‘P’ passes, each of which is a Bucket Sort

Comparison to Comparison Sorts

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
 - Approximate run-time: $15 \cdot (52 + n)$
 - This is less than $n \log n$ only if $n > 33,000$
 - Of course, cross-over point depends on constant factors of the implementations plus P and B
 - And radix sort can have poor locality properties
- Not really practical for many classes of keys
 - Strings: Lots of buckets

Recap: Features of Sorting Algorithms

In-place

- Sorted items occupy the same space as the original items. (No copying required, only $O(1)$ extra space if any.)

Stable

- Items in input with the same value end up in the same order as when they began.

Examples:

- Merge Sort - not in place, stable
- Quick Sort - in place, not stable

Sorting Summary

- Simple $O(n^2)$ sorts can be fastest for small n
 - selection sort, insertion sort (latter linear for mostly-sorted)
 - good for “below a cut-off” to help divide-and-conquer sorts
- $O(n \log n)$ sorts
 - heap sort, in-place but not stable nor parallelizable
 - merge sort, not in place but stable and works as external sort
 - quick sort, in place but not stable and $O(n^2)$ in worst-case
 - often fastest, but depends on costs of comparisons/copies
- $\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
 - Bucket sort good for small number of key values
 - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!

Sorting massive data: External Sorting

Need sorting algorithms that **minimize disk/tape access** time:

- Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
- Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access

Basic Idea:

- Load chunk of data into Memory, sort, store this “run” on disk/tape
- Use the Merge routine from Mergesort to merge runs
- Repeat until you have only one run (one sorted chunk)

- Mergesort can leverage multiple disks
- Weiss gives some examples