Announcements

- HW4: Due tomorrow!

- Final in EXACTLY 2 weeks.
  - Start studying

CSE373: Data Structure & Algorithms

Beyond Comparison Sorting

Hunter Zahn

Summer 2016

Introduction to Sorting

- Stacks, queues, priority queues, and dictionaries all focused on providing one element at a time
- But often we know we want “all the things” in some order
  - Humans can sort, but computers can sort fast
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - List of countries ordered by population
    - Search engine results by relevance
    - …
- Algorithms have different asymptotic and constant-factor trade-offs
  - No single “best” sort for all scenarios
  - Knowing one way to sort just isn’t enough

More Reasons to Sort

General technique in computing:

Preprocess data to make subsequent operations faster

Example: Sort the data so that you can

- Find the **k**th largest in constant time for any **k**
- Perform binary search to find elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change (and how much it will change)
- How much data there is

The main problem, stated carefully

For now, assume we have *n* comparable elements in an array and we want to rearrange them to be in increasing order

Input:

- An array **`A`** of data records
- A key value in each data record
- A comparison function

Effect:

- Reorganize the elements of **`A`** such that for any **`i`** and **`j`**,
- if **`i < j`** then **`A[i] ≤ A[j]`**
- (Also, **`A`** must have exactly the same data it started with)
- Could also sort in reverse order, of course

An algorithm doing this is a comparison sort

Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort	Heap sort		Bucket sort	External sorting
Selection sort	Merge sort		Radix sort	
Shell sort	Quick sort (avg)			
…	…			

Visualization

Insertion Sort

- Idea: At step **k**, put the **k**th element in the correct position among the first **k** elements

- Alternate way of saying this:
  - Sort first two elements
  - Now insert 3rd element in order
  - Now insert 4th element in order
  - …

- "Loop invariant": when loop index is **i**, first **i** elements are sorted

- Time?

Best-case $O(n)$	Worst-case $O(n^2)$	"Average" case $O(n^2)$
start sorted	start reverse sorted	(see text)

Selection sort

- Idea: At step **`k`**, find the smallest element among the not-yet-sorted elements and put it at position k
- Alternate way of saying this:
  - Find smallest element, put it 1st
  - Find next smallest element, put it 2nd
  - Find next smallest element, put it 3rd
  - …
- "Loop invariant": when loop index is **`i`**, first **`i`** elements are the **`i`** smallest elements in sorted order

- Time?

  Best-case $O(n^2)$ Worst-case $O(n^2)$ "Average" case $O(n^2)$

  Always $T(1) = 1$ and $T(n) = n + T(n-1)$

Bubble Sort

- Not intuitive – It's unlikely that you'd come up with bubble sort
- It doesn't have good asymptotic complexity: $O(n^2)$
- It's not particularly efficient with respect to common factors

Basically, almost everything it is good at some other algorithm is at least as good at

Heap sort

- Sorting with a heap is easy:
  - `insert` each `arr[i]`, or better yet use `buildHeap`
  - ```
    for(i=0; i < arr.length; i++)
        arr[i] = deleteMin();
    ```
- Worst-case running time: $O(n \log n)$
- We have the array-to-sort and the heap
  - So this is not an in-place sort
  - There's a trick to make it in-place…
    ```

In-place heap sort sort

But this reverse sorts – how would you fix that?

- Treat the initial array as a heap (via **`buildHeap`**)
- When you delete the **`i`**th element, put it at **`arr[n-i]`**
    - That array location isn't needed for the heap anymore!

4	7	5	9	8	6	10	3	2	1

heap part sorted part

`arr[n-i]=`
`deleteMin()`

5	7	6	9	8	10	4	3	2	1

heap part sorted part

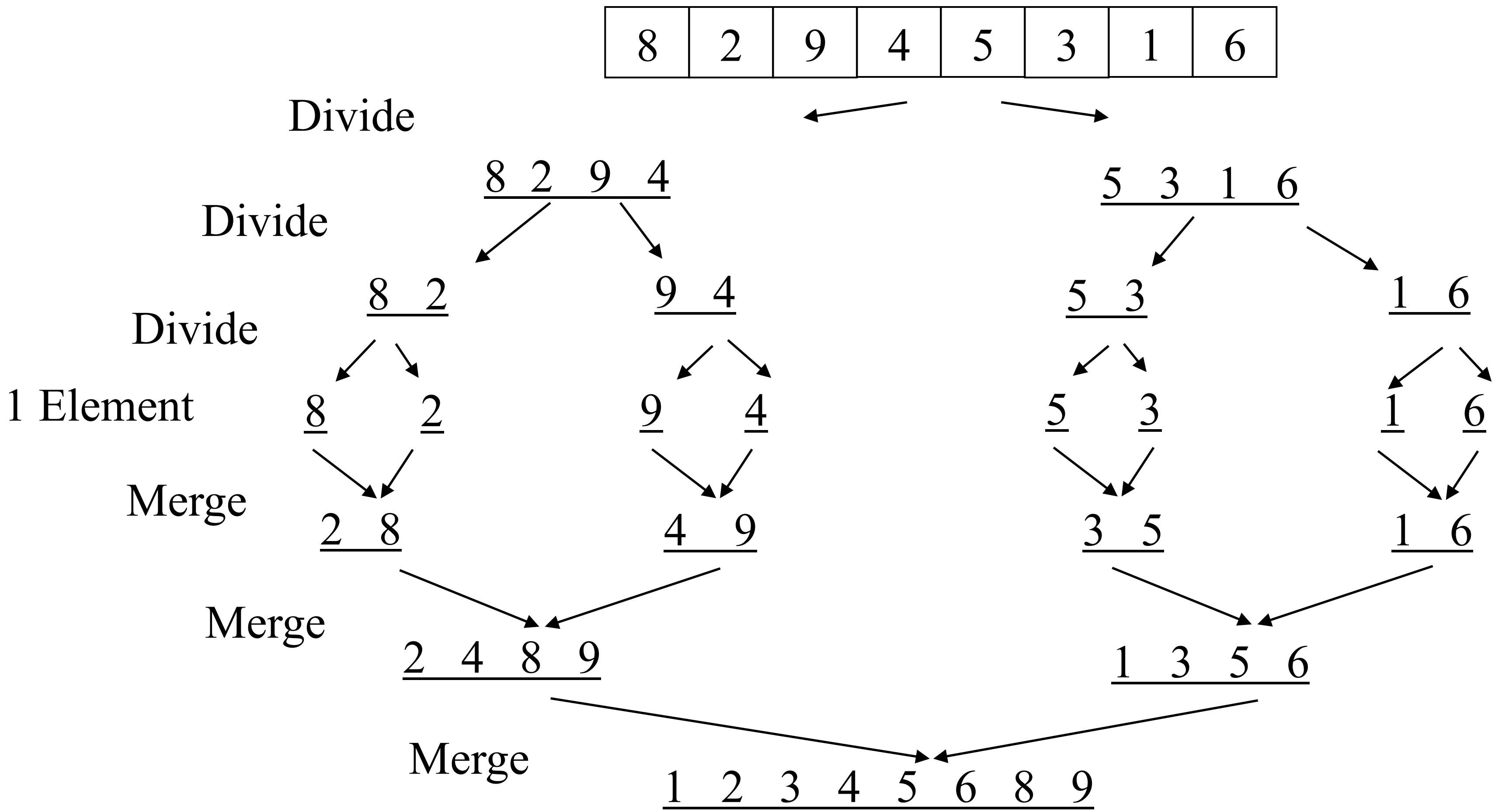
Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

1. Mergesort: Sort the left half of the elements (recursively)
   Sort the right half of the elements (recursively)
   Merge the two sorted halves into a sorted whole

2. Quicksort: Pick a "pivot" element
   Divide elements into less-than pivot
   and greater-than pivot
   Sort the two divisions (recursively on each)
   Answer is sorted-less-than then pivot then
   sorted-greater-than

Example, Showing Recursion



Quicksort

- Also uses divide-and-conquer
  – Recursively chop into two pieces
  – Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  – Unlike merge sort, does not need auxiliary space
- $O(n \texttt{ log } n)$ on average ☺, but $O(n^2)$ worst-case ☹
- Faster than merge sort in practice?
  – Often believed so
  – Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

Quicksort Overview

1. Pick a pivot element

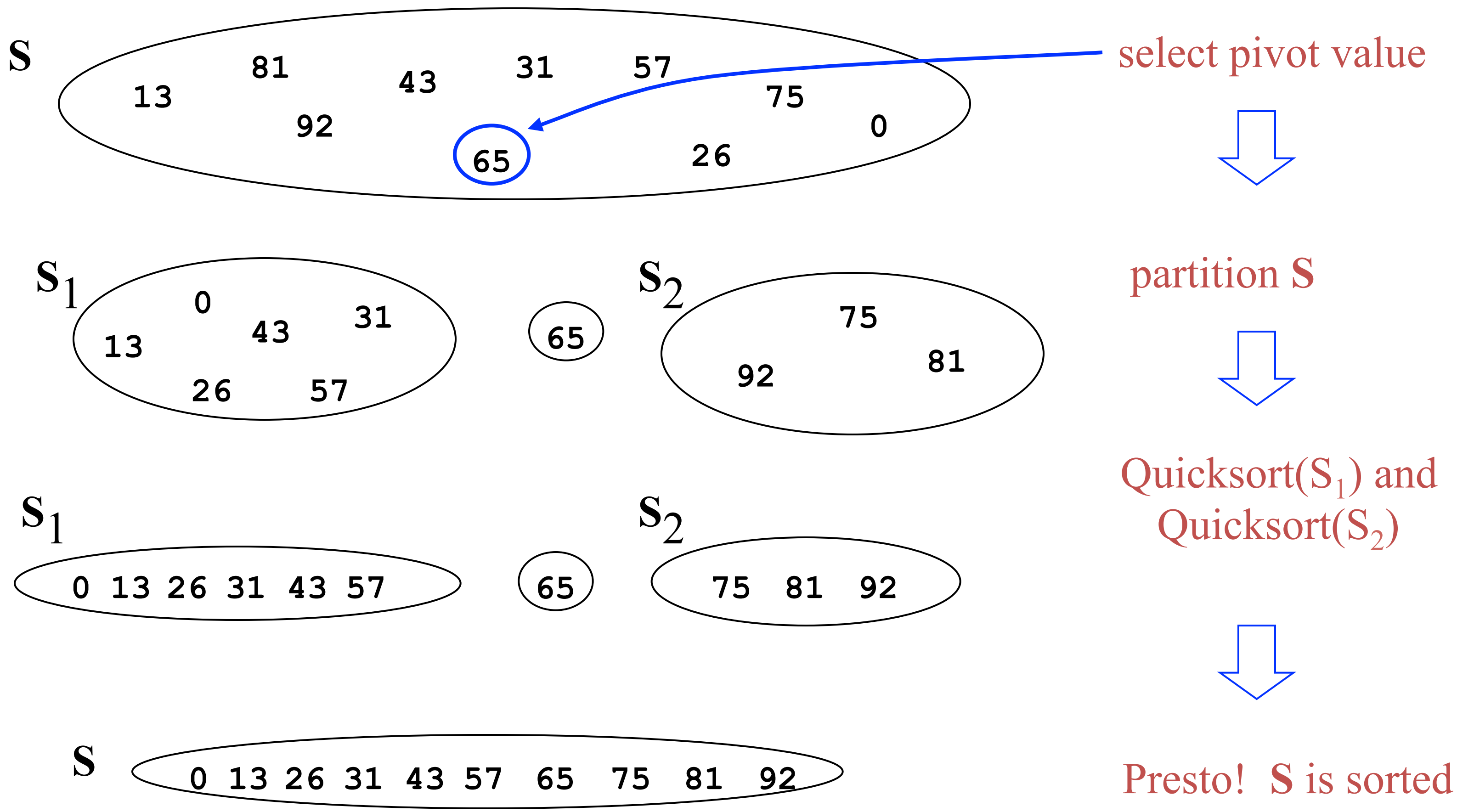
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot

3. Recursively sort A and C

4. The answer is, “as simple as A, B, C”

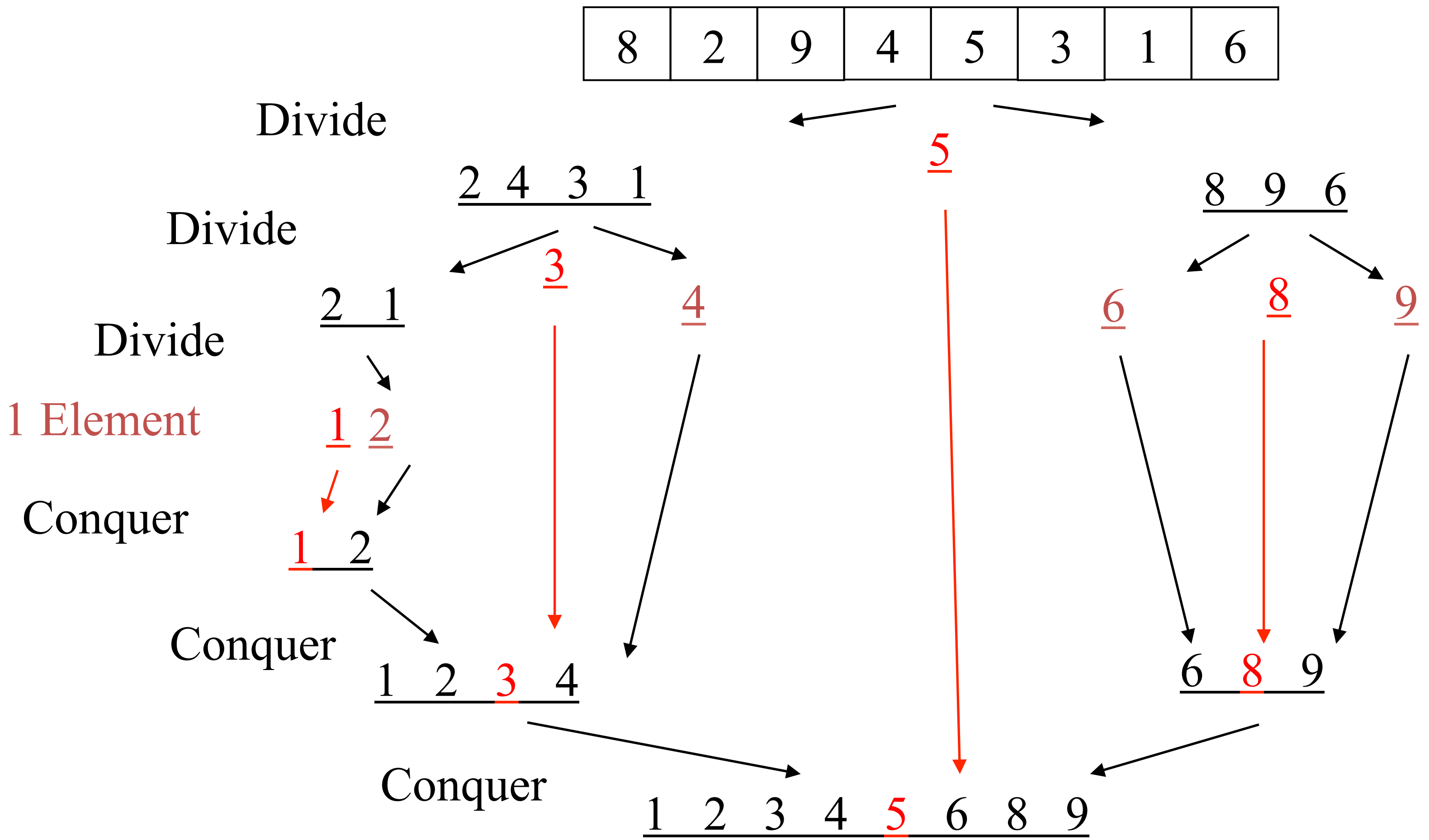
(Alas, there are some details lurking in this algorithm)

Think in Terms of Sets



[Weiss]

Example, Showing Recursion



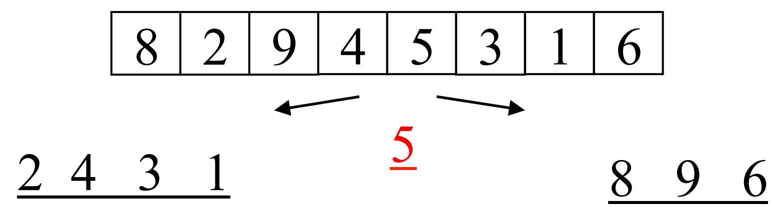
Details

Have not yet explained:

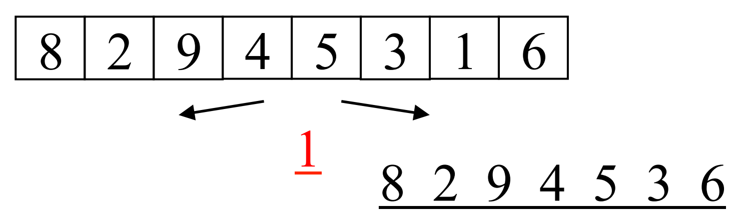
- How to pick the pivot element
  - Any choice is correct: data will end up sorted
  - But as analysis will show, want the two partitions to be about equal in size
- How to implement partitioning
  - In linear time
  - In place

Pivots

- Best pivot?
  - Median
  - Halve each time



- Worst pivot?
  - Greatest/least element
  - Problem of size n - 1
  - $O(n^2)$



Potential pivot rules

While sorting **`arr`** from **`lo`** (inclusive) to **`hi`** (exclusive)…

- Pick **`arr[lo]`** or **`arr[hi-1]`**
  - Fast, but worst-case occurs with mostly sorted input
- Pick random element in the range
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach
- Median of 3, e.g., **`arr[lo], arr[hi-1], arr[(hi+lo)/2]`**
  - Common heuristic that tends to work well

Partitioning

- Conceptually simple, but hardest part to code up correctly
  - After picking pivot, need to partition in linear time in place

- One approach (there are slightly fancier ones):
  1. Swap pivot with **`arr[lo]`**
  2. Use two fingers **`i`** and **`j`**, starting at **`lo+1`** and **`hi-1`**
  3.
```
3. while (i < j)
       if (arr[j] > pivot) j--
       else if (arr[i] < pivot) i++
       else swap arr[i] with arr[j]
```
  4. Swap pivot with **`arr[i]`** *

*skip step 4 if pivot ends up being least element

Example

- **Step one**: pick pivot as median of 3
  - **`lo`** = 0, **`hi`** = 10

0	1	2	3	4	5	6	7	8	9
8	1	4	9	**0**	3	5	2	7	**6**

- Step two: move pivot to the `lo` position

0	1	2	3	4	5	6	7	8	9
6	1	4	9	**0**	3	5	2	7	**8**

Example

Often have more than one swap during partition – this is a short example

Now partition in place

6	1	4	9	**0**	3	5	2	7	**8**

Move fingers

6	1	4	9	**0**	3	5	2	7	**8**

Swap

6	1	4	2	**0**	3	5	9	7	**8**

Move fingers

6	1	4	2	**0**	3	5	9	7	**8**

Move pivot

5	1	4	2	**0**	3	6	9	7	**8**

Analysis

- **Best-case**: Pivot is always the median
  $T(0)=T(1)=1$
  $T(n)=2T(n/2) + n$ -- linear-time partition
  Same recurrence as mergesort: $O(n \log n)$

- **Worst-case**: Pivot is always smallest or largest element
  $T(0)=T(1)=1$
  $T(n) = 1T(n-1) + n$
  Basically same recurrence as selection sort: $O(n^2)$

- **Average-case** (e.g., with random pivot)
  - $O(n \log n)$, not responsible for proof (in text)

Cutoffs

- For small *n*, all that recursion tends to cost more than doing a quadratic sort
  - Remember asymptotic complexity is for large *n*

- Common engineering technique: switch algorithm below a cutoff
  - Reasonable rule of thumb: use insertion sort for $n < 10$

- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
    - Switch to sequential algorithm
  - None of this affects asymptotic complexity

Cutoff skeleton

```
void quicksort(int[] arr, int lo, int hi) {
  if(hi – lo < CUTOFF)
     insertionSort(arr,lo,hi);
  else
     …
}
```

Notice how this cuts out the vast majority of the recursive calls

- Think of the recursive calls to quicksort as a tree
- Trims out the bottom layers of the tree

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures…

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort Selection sort Shell sort …	Heap sort Merge sort Quick sort (avg) …		Bucket sort Radix sort	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 time
- 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: we *know* that this is *impossible*
    - ***Assuming* our comparison *model***: The only operation an algorithm can perform on data items is a 2-element comparison

A General View of Sorting

- Assume we have *n* elements to sort
  - For simplicity, assume none are equal (no duplicates)

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

- Example, ***n*=3**

  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, *n*-1 for next, *n*-2 for next, …
  - $n(n\text{-}1)(n\text{-}2)\ldots(2)(1) = \boldsymbol{n!}$ possible orderings

Counting Comparisons

- So *every* sorting algorithm has to "find" the right answer among the n! possible answers
  - Starts "knowing nothing", "anything is possible"
  - Gains information with each comparison
  - **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities
  - Must narrow answer down to a single possibility

- **What we can show**:

 Any sorting algorithm must do *at least* $(1/2)n\log n - (1/2)n$

 (which is $\Omega(n \log n)$) comparisons

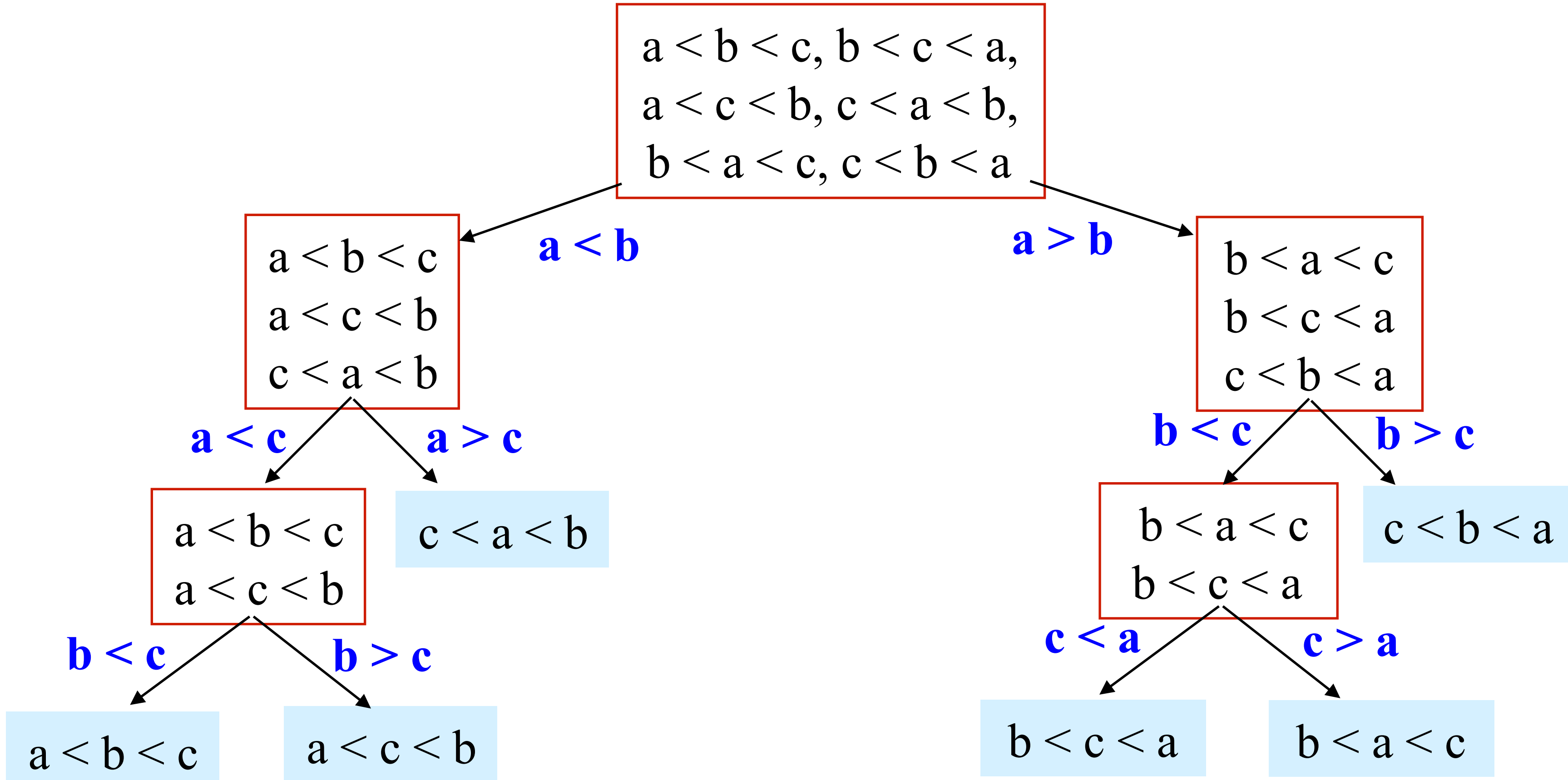
  - Otherwise there are at least two permutations among the n! possible that cannot yet be distinguished, so the algorithm would have to guess and could be wrong [incorrect algorithm]

Optional: 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

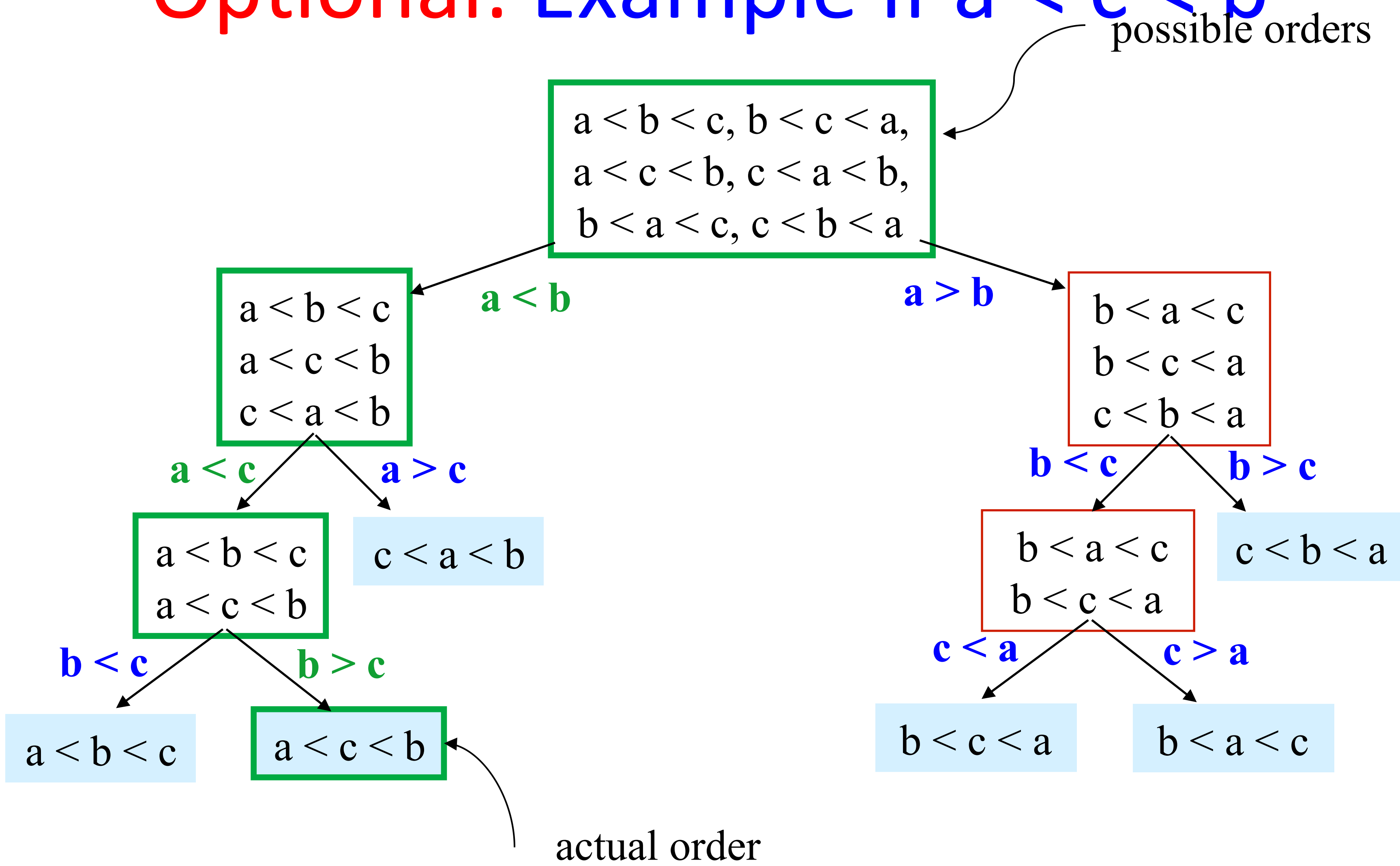
- Can represent this process as a *decision tree*
  - Nodes contain "set of remaining possibilities"
    - Root: None of the n! options yet eliminated
  - Edges are "answers from a comparison"
  - The algorithm does not actually build the tree; it's what our *proof* uses to represent "the most the algorithm could know so far" as the algorithm progresses

Optional: 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

Optional: Example if a < c < b



Optional: What the Decision Tree Tells Us

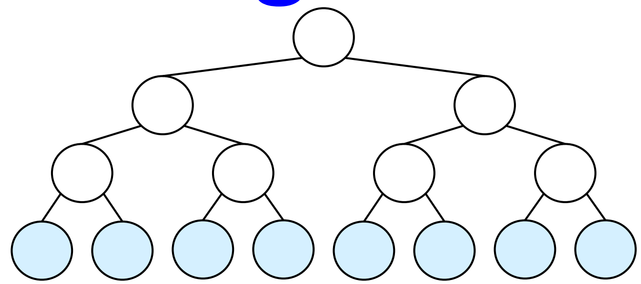
- A binary tree because each comparison has 2 outcomes
  - (We assume no duplicate elements)
  - (Would have 1 outcome if algorithm asks redundant questions) This means that poorly implemented algorithms could yield deeper trees (categorically bad)

- 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

Optional: Where are we

- **Proven**: No comparison sort can have worst-case running time better than the height of a binary tree with n! leaves
  - A comparison sort could be worse than this height, but it cannot be better
- **Now**: a binary tree with n! leaves has height $\Omega(n \log n)$
  - Height could be more, but cannot be less
  - Factorial function grows very quickly
- **Conclusion**: Comparison sorting is $\Omega(n \log n)$
  - An amazing computer-science result: proves all the clever programming in the world cannot comparison-sort in linear time

Optional: Height lower bound



- 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

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

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

$\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2)$ drop smaller terms

$\geq \log_2 (n/2) + \log_2 (n/2) + \ldots + \log_2 (n/2)$ shrink terms to $\log_2 (n/2)$

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

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

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

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

Height, or # of comparisons made bounded by $n \log n$

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures…

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort Selection sort Shell sort …	Heap sort Merge sort Quick sort (avg) …		Bucket sort Radix sort	External sorting

How???

- Change the model – assume more than "compare(a,b)"

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*
  - 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 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

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

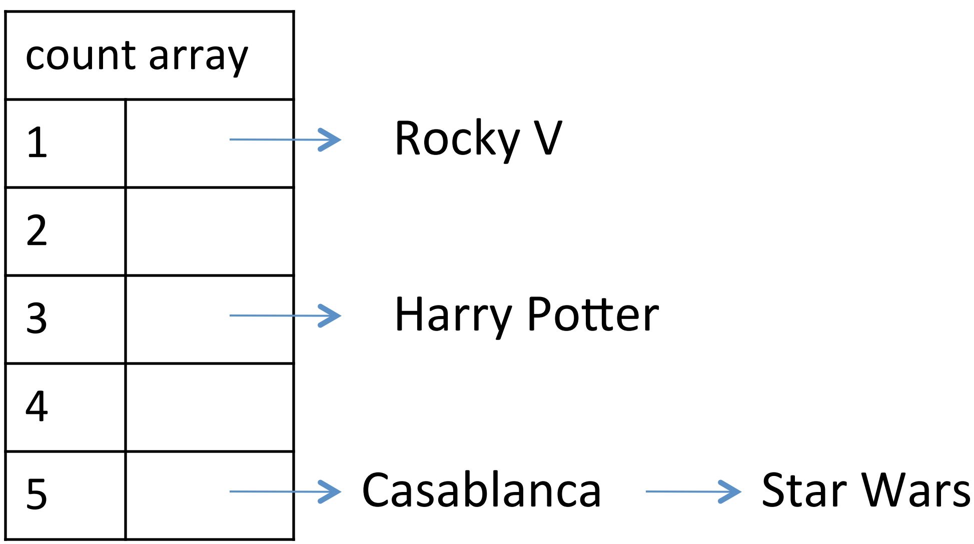
- Good when K is smaller (or not much larger) than n
  - We don't spend time doing comparisons of duplicates

- Bad when K is much larger than n
  - Wasted space; wasted time during 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 keys; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, insert in *O*(1) (at beginning, or keep pointer to last element)



- Example: Movie ratings; scale 1-5;1=bad, 5=excellent

  Input=

  5: Casablanca

  3: Harry Potter movies

  5: Star Wars Original Trilogy

  1: Rocky V

- Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
- Easy to keep '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
    - Keeping sort *stable*
  - Do one pass per digit
  - Invariant: After k passes (digits), the last k digits are sorted

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3				537	478	9
			143				67	38	

Input: 478
537
9
721
3
38
143
67

First pass:

bucket sort by ones digit

Order now: 721
3
143
537
67
478
38
9

Example

0	1	2	3	4	5	6	7	8	9
	721		**3** 143				**537** 67	**478** 38	**9**

Radix = 10

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Order was:

721
3
143
537
67
478
38
9

Second pass:

stable bucket sort by tens digit

Order now:

3
9
721
537
38
143
67
478

Example

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Radix = 10

0	1	2	3	4	5	6	7	8	9
3 9 38 67	143			478	537		721		

Order was:

3
9
721
537
38
143
67
478

Third pass:

stable bucket sort by 100s digit

Order now:

3
9
38
67
143
478
537
721

Analysis

Input size: n

Number of buckets = Radix: B

Number of passes = "Digits": P

Work per pass is 1 bucket sort: $O(B+n)$

Total work is $\boldsymbol{O(P(B+n))}$

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

- Example: Strings of English letters up to length 15
  - Run-time proportional to: $15*(52 + n)$
  - This is less than n log n only if $n > 33{,}000$
  - Of course, cross-over point depends on constant factors of the implementations
    - And radix sort can have poor locality properties

Sorting massive data

- Note: If data is on disk (ie too big to fit in main memory), reading and writing are much slower
- Need sorting algorithms that minimize disk 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
- Mergesort is the basis of massive sorting
- Mergesort can leverage multiple disks

Last Slide on Sorting

- 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
- Ω $(n$ `log` $n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of possible key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!