CSE 373: Data Structures & Algorithms

More Sorting and Beyond Comparison Sorting

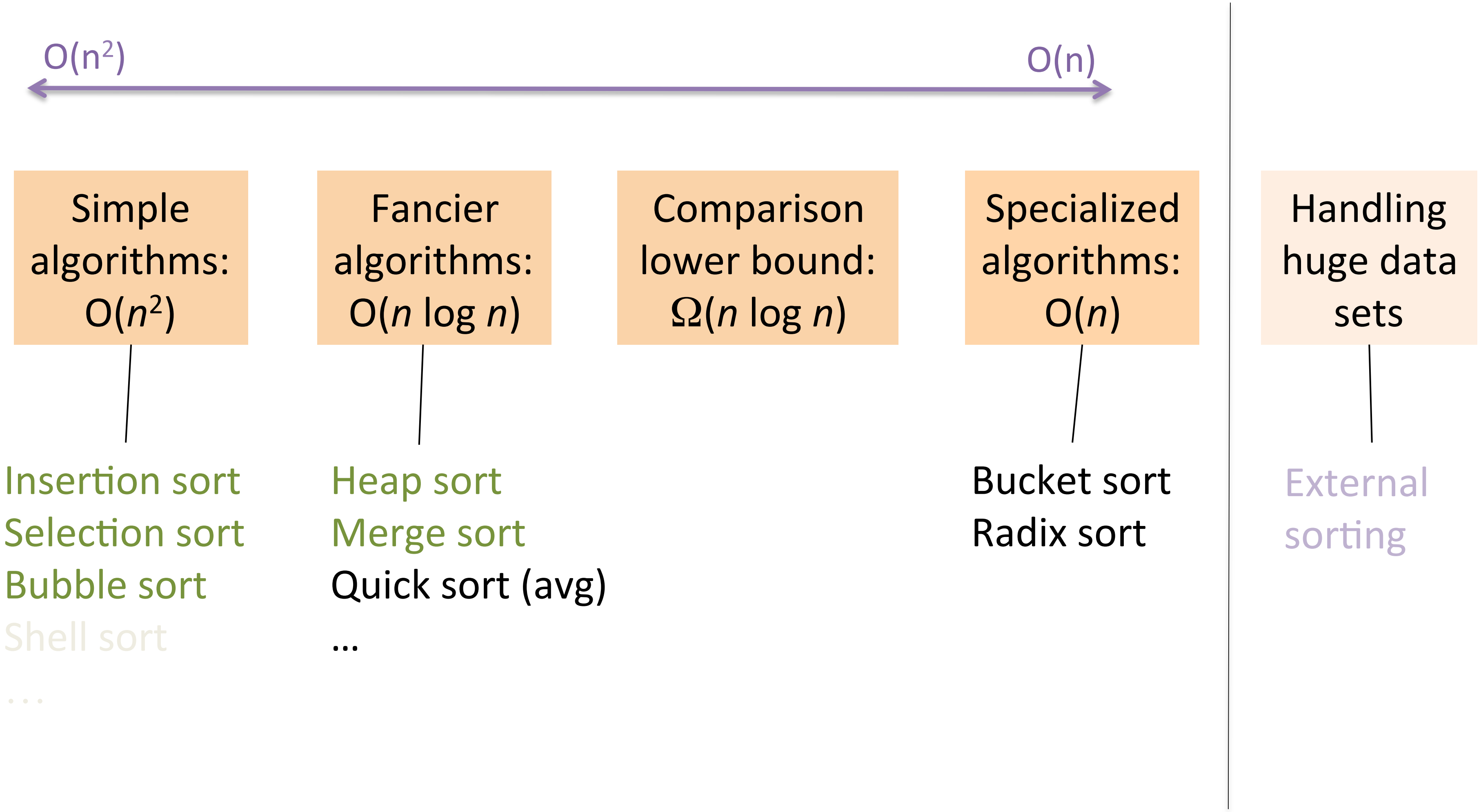
Riley Porter

Winter 2017

Course Logistics

- HW5 due in a couple days → more graphs! Don't forget about the write-up!
- HW6 out later today → sorting (and to a lesser degree reading specs/other files/ tests).
- Final exam in 2 weeks!

Review: Sorting: The Big Picture



$O(n^2)$ ⟷ $O(n)$

Simple algorithms: $O(n^2)$
- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- …

Fancier algorithms: $O(n \log n)$
- Heap sort
- Merge sort
- Quick sort (avg)
- …

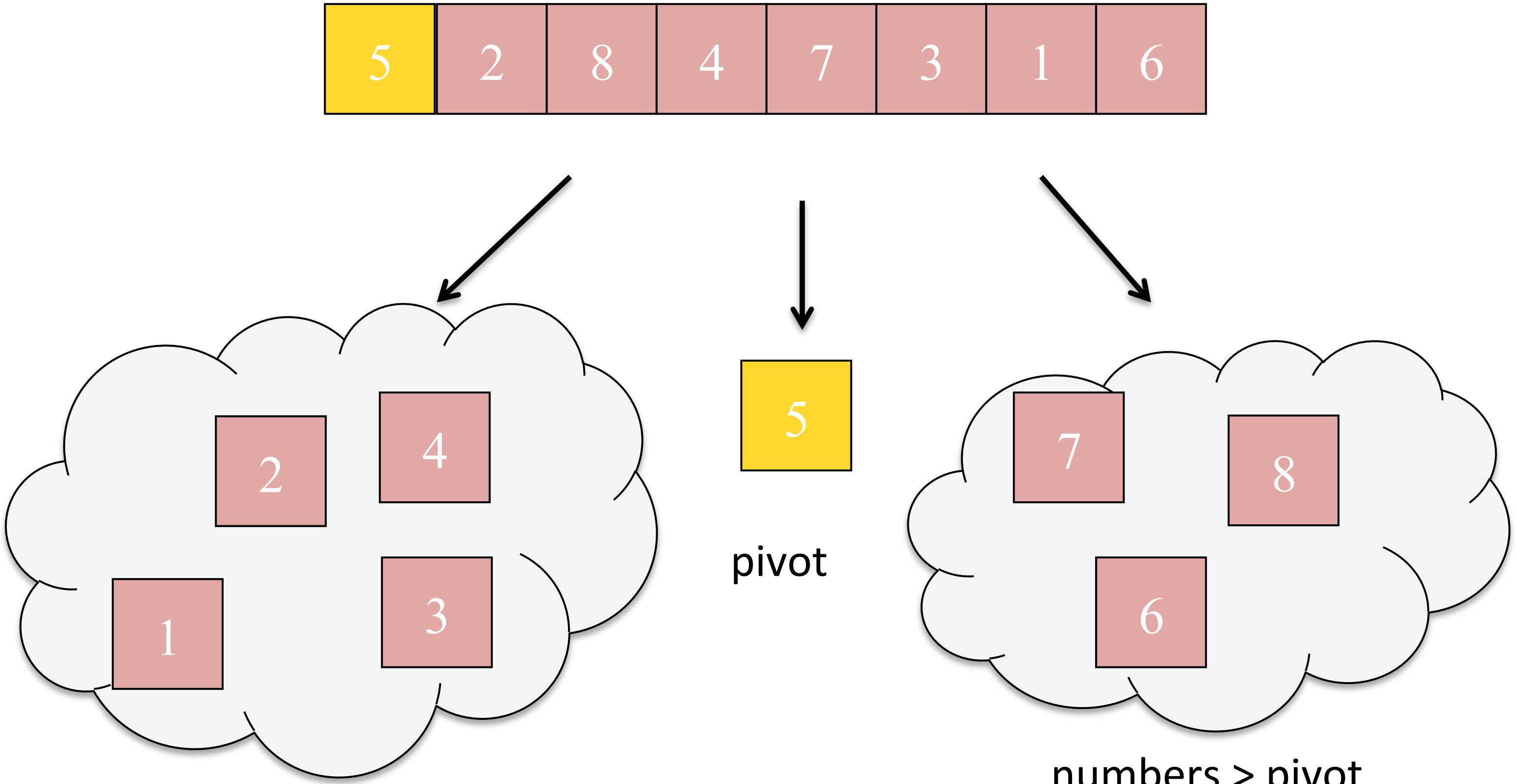
Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$
- Bucket sort
- Radix sort

Handling huge data sets
- External sorting

Quick Sort

Divide: Split array around a ‘pivot’



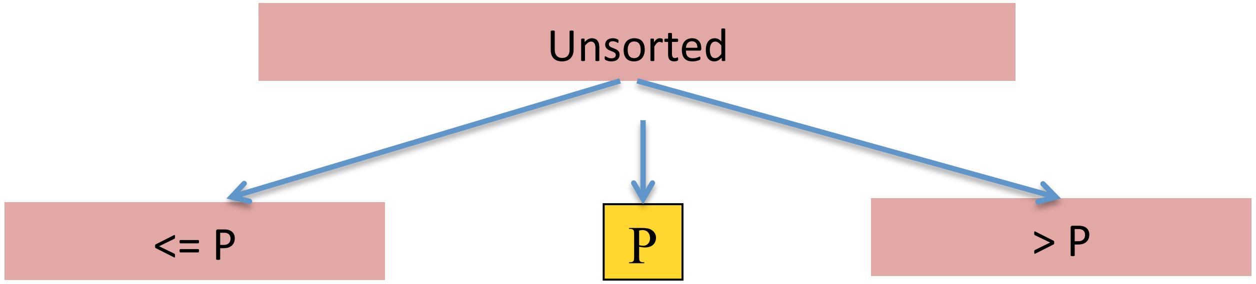
pivot

numbers > pivot

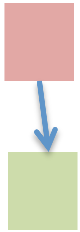
numbers <= pivot

Quick Sort

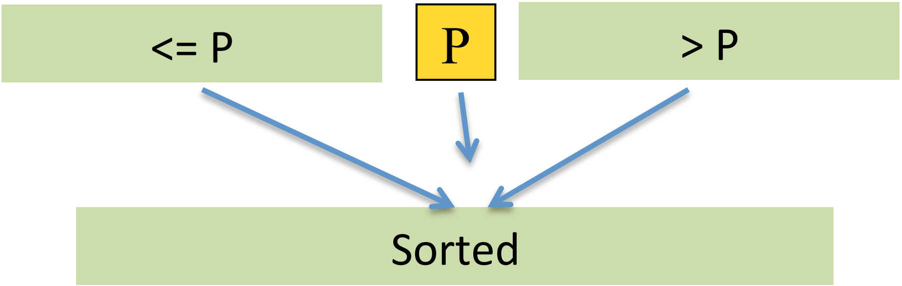
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot

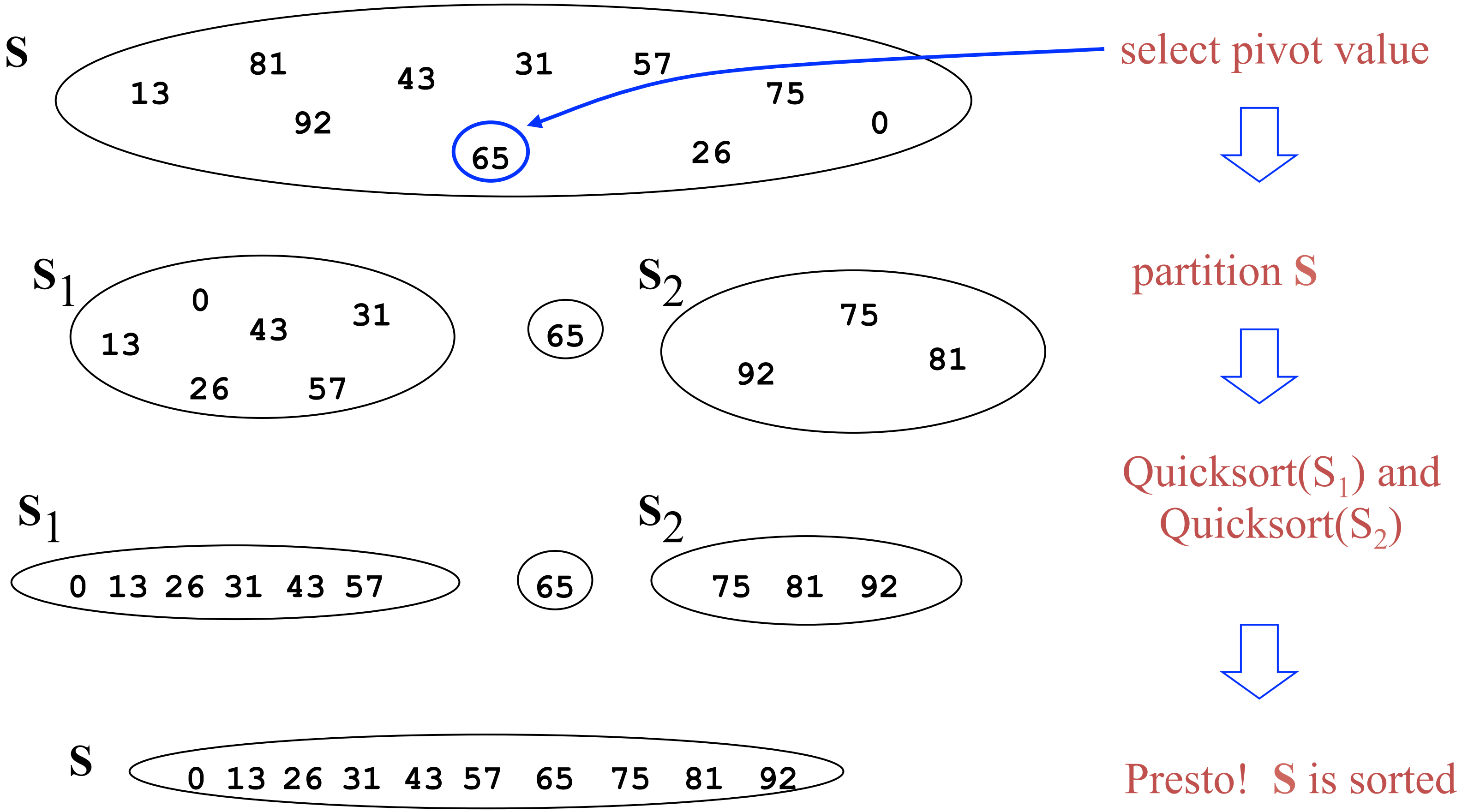


Quick Sort Pseudocode

Core idea: Pick some item from the array and call it the pivot. Put all items smaller in the pivot into one group and all items larger in the other and recursively sort. If the array has size 0 or 1, just return it unchanged.

```
quicksort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```

Think in Terms of Sets



select pivot value

partition S

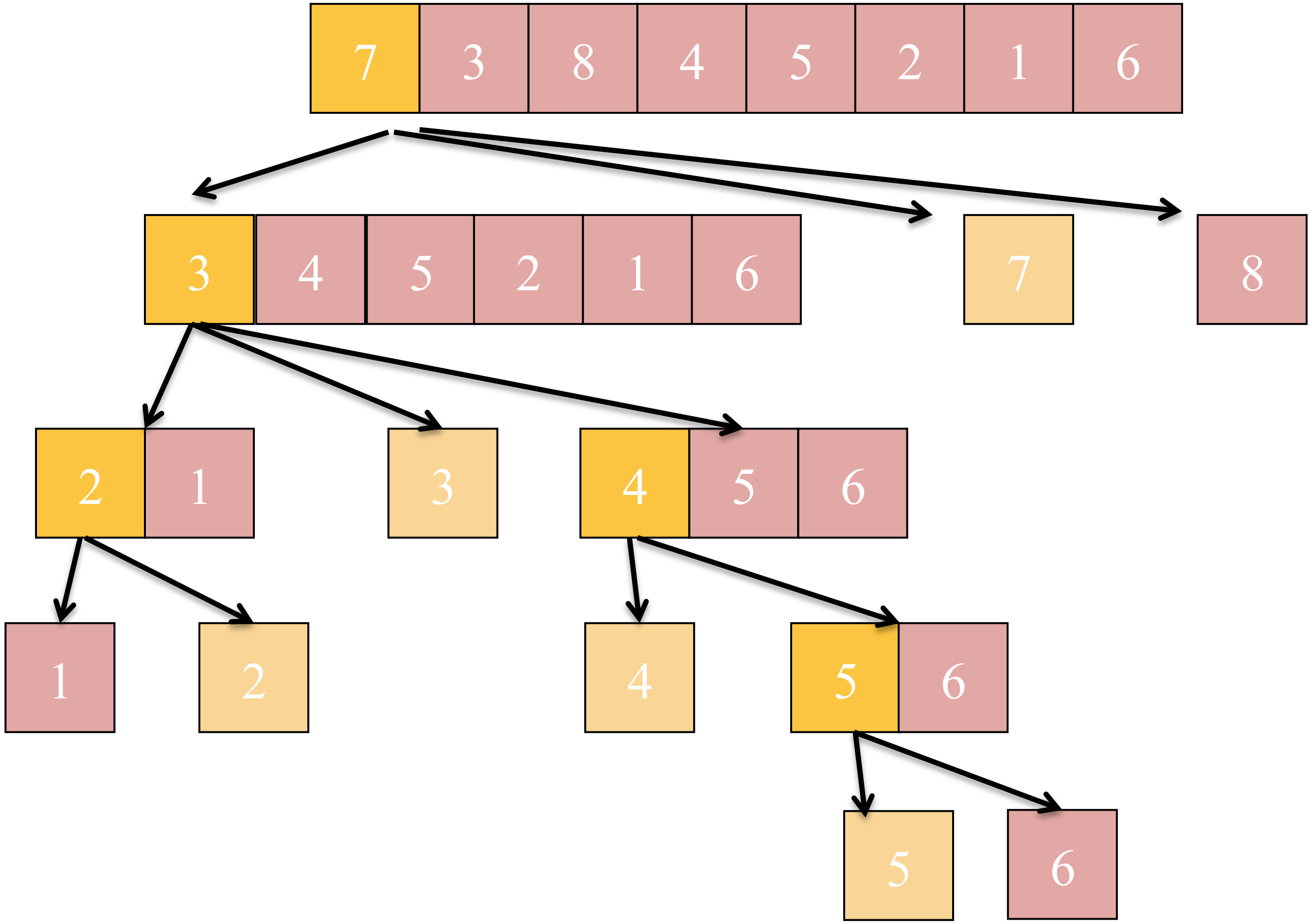
Quicksort(S_1) and Quicksort(S_2)

Presto! S is sorted

[Weiss]

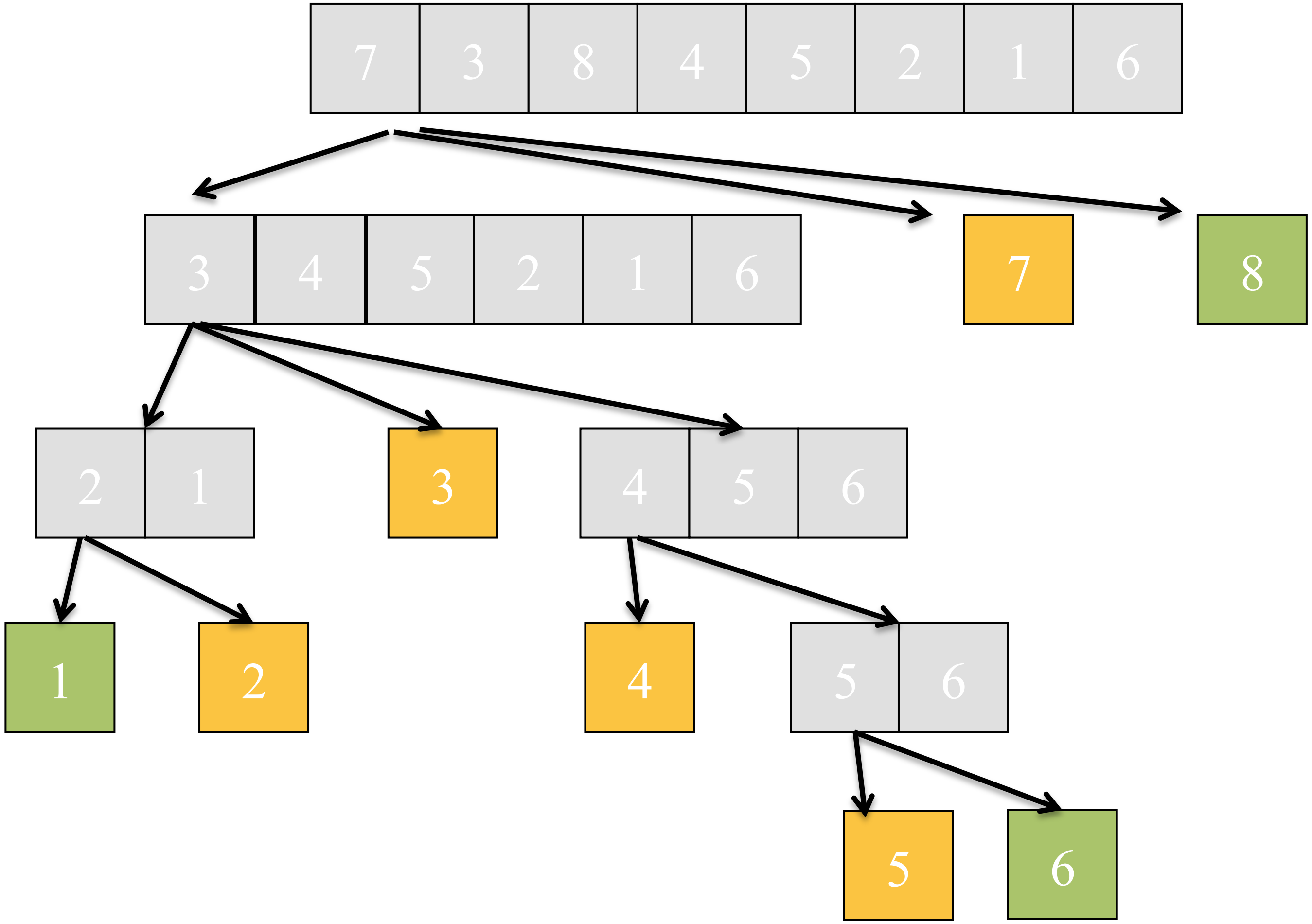
Quick Sort Example: Divide

Pivot rule: pick the element at index 0



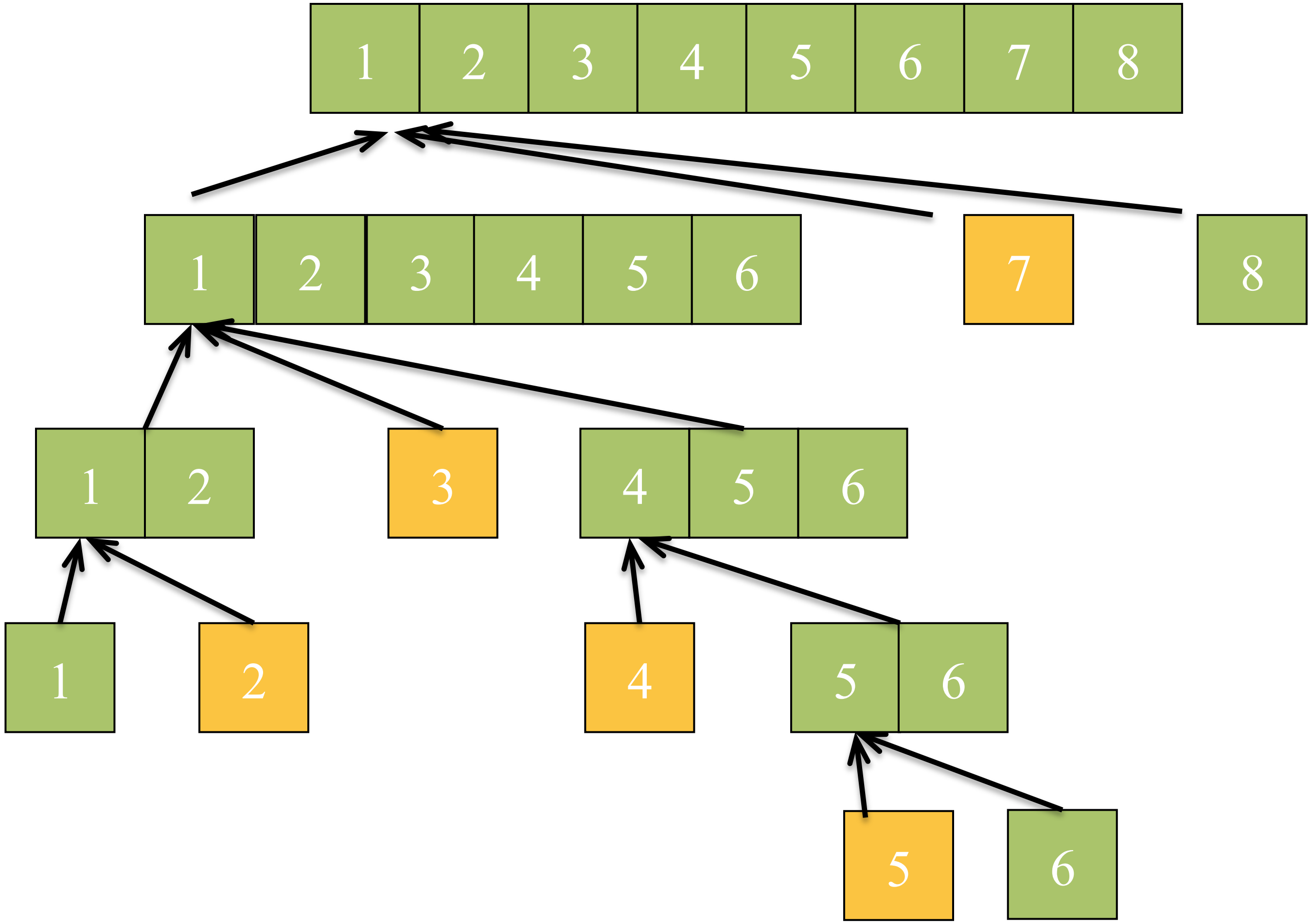
Quick Sort Example: Combine

Combine: this is the order of the elements we'll care about when combining



Quick Sort Example: Combine

Combine: put left partition < pivot < right partition



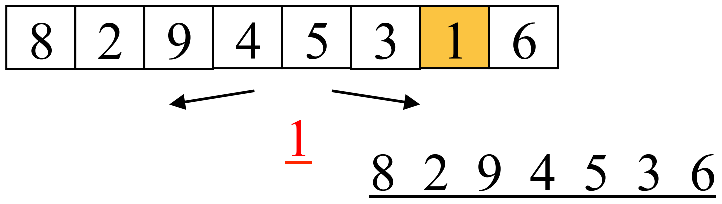
Details

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

- Worst pivot?
  - Greatest/least element
  - Recurse on problem of size n – 1

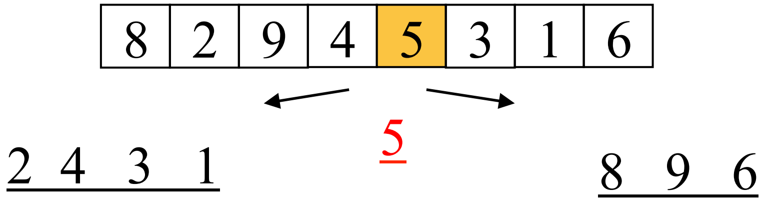


8	2	9	4	5	3	1	6

1

8 2 9 4 5 3 6

- Best pivot?
  - Median
  - Halve each time



8	2	9	4	5	3	1	6

2 4 3 1

5

8 9 6

Potential pivot rules

- **Pick first or last element**: fast, but worst-case occurs with mostly sorted input (as we've seen)

- **Try looping through the array:** we'll get a good value, but that's slow and hard to implement

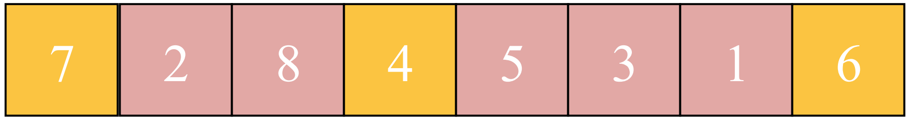
- **Pick random element:** cool, does as well as any technique, but (pseudo)random number generation can be slow

- **Pick the median of first, middle, and last:** Easy to implement and is a common heuristic that tends to work well

  e.g., **`arr[lo], arr[hi-1], arr[(hi+lo)/2]`**

Median Pivot Example

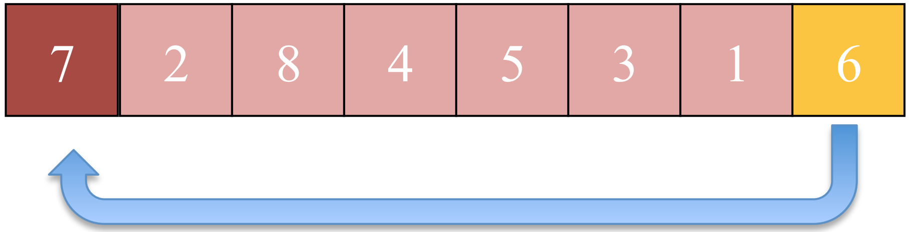
Pick the median of first, middle, and last



7	2	8	4	5	3	1	6

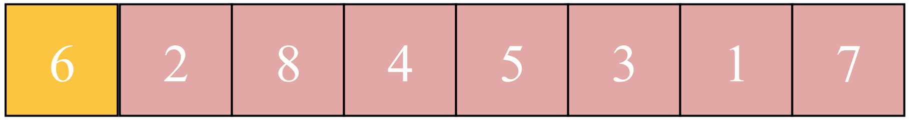
Median = 6

Swap the median with the first value



7	2	8	4	5	3	1	6

Pivot is now at index 0, and we’re ready to go



6	2	8	4	5	3	1	7

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. Put pivot in index **`lo`**
  2. Use two pointers **`i`** and **`j`**, starting at **`lo+1`** and **`hi-1`**

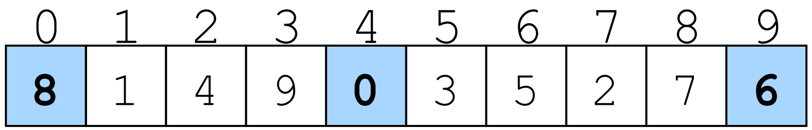
```
3. while (i < j)
       if (arr[j] > pivot) j--
       else if (arr[i] < pivot) i++
       else swap arr[i] with arr[j]
```

  4. Swap pivot with **`arr[i]`** *

*skip step 4 if pivot ends up being least element

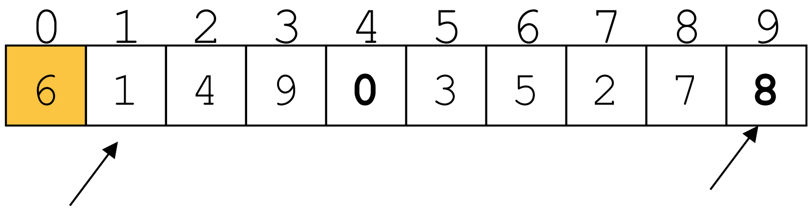
Example

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



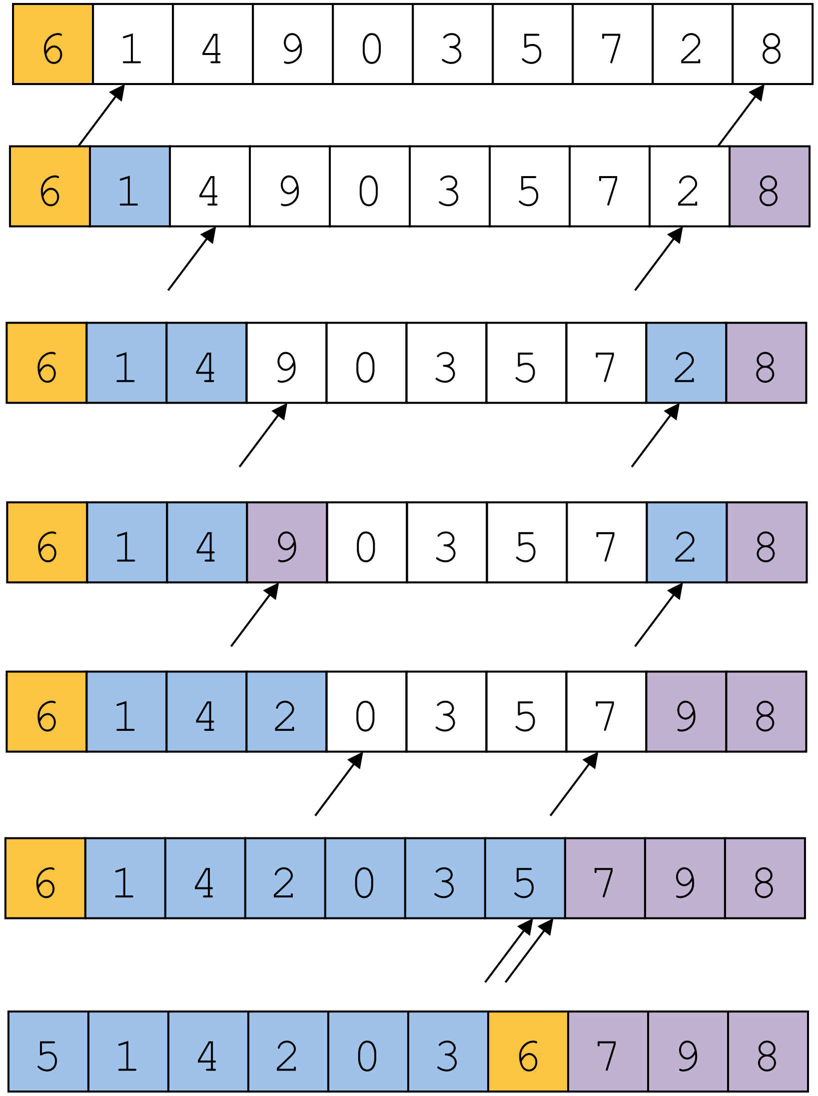
0	1	2	3	4	5	6	7	8	9
8	1	4	9	**0**	3	5	2	7	**6**

- 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**

Quick Sort Partition Example



Quick Sort Analysis

- **Best-case**: Pivot is always the median, split data in half

  Same as mergesort: $O(n \log n)$, O(n) partition work for O(log(n)) levels

- **Worst-case**: Pivot is always smallest or largest element

  Basically same as selection sort: $O(n^2)$

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

Quick Sort Analysis

- **In-place:** Yep! We can use a couple pointers and partition the array in place, recursing on different **`lo`** and **`hi`** `indices`

- **Stable**: Not necessarily. Depends on how you handle equal values when partitioning. A stable version of quick sort uses some extra storage for partitioning.

Divide and Conquer: Cutoffs

- For small *n*, all that recursion tends to cost more than doing a simple, 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:
  - Cutoffs are also the norm with parallel algorithms
    - Switch to sequential algorithm
  - None of this affects asymptotic complexity

Cutoff Pseudocode

```
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

Cool Comparison Sorting Links

- Visualization of sorts on different inputs: http://www.sorting-algorithms.com/
- Visualization of sorting with sound: https://www.youtube.com/watch?v=t8g-iYGHpEA
- Sorting via dance: https://www.youtube.com/watch?v=XaqR3G_NVoo
- XKCD Ineffective sorts:

  https://xkcd.com/1185/

Sorting: 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 time
- These bounds are all tight, actually $\Theta(n \log n)$
- ***Assuming* our comparison *model***: The only operation an algorithm can perform on data items is a 2-element comparison. There is no lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$

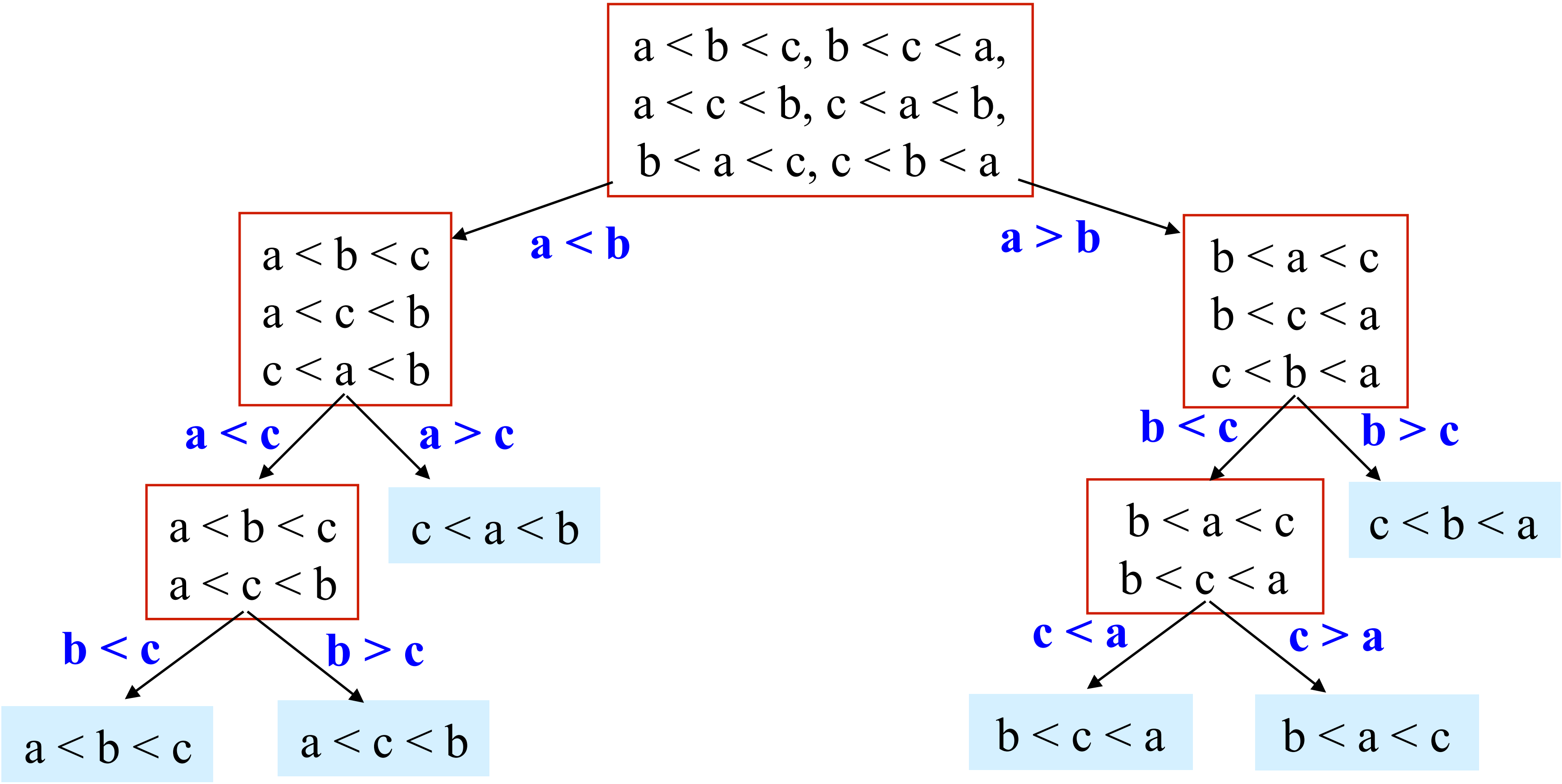
Counting Comparisons

- No matter what the algorithm is, it cannot make progress without doing comparisons

- **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities of possible orderings

- Can represent this process as a *decision tree*
  - Nodes contain "set of remaining possibilities"
  - 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

Decision Tree for n = 3


a < b < c, b < c < a,
a < c < b, c < a < b,
b < a < c, c < b < a

a < b

a > b

a < b < c
a < c < b
c < a < b

b < a < c
b < c < a
c < b < a

a < c

a > c

b < c

b > c

a < b < c
a < c < b

c < a < b

b < a < c
b < c < a

c < b < a

b < c

b > c

c < a

c > a

a < b < c

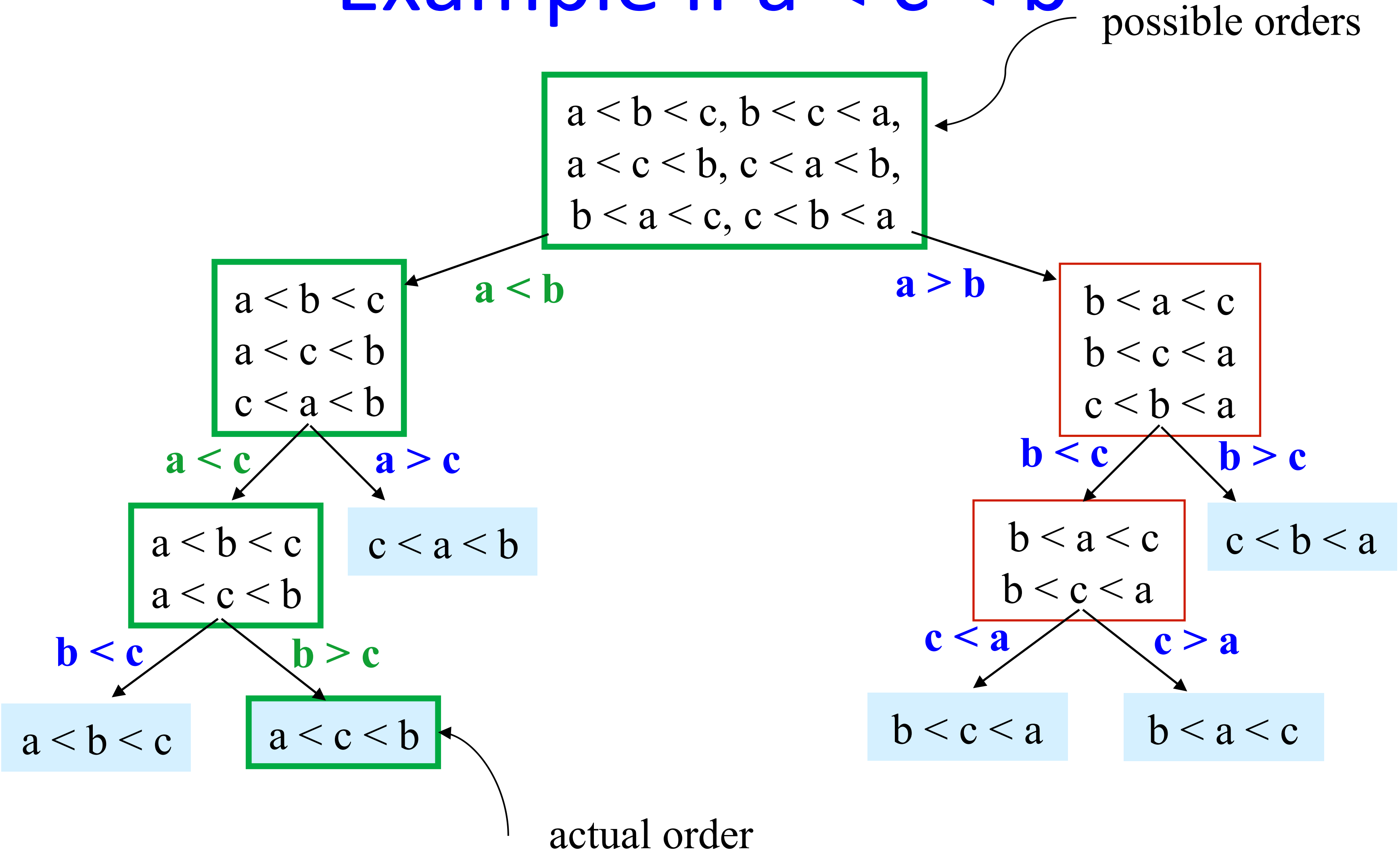
a < c < b

b < c < a

b < a < c

- The leaves contain all the possible orderings of a, b, c

Example if a < c < b



possible orders

a < b < c, b < c < a,
a < c < b, c < a < b,
b < a < c, c < b < a

a < b

a < b < c
a < c < b
c < a < b

a > b

b < a < c
b < c < a
c < b < a

a < c

a < b < c
a < c < b

a > c

c < a < b

b < c

b < a < c
b < c < a

b > c

c < b < a

b < c

a < b < c

b > c

a < c < b

c < a

b < c < a

c > a

b < a < c

actual order

What the Decision Tree Tells Us

- A binary tree because each comparison has 2 outcomes (we're comparing 2 elements at a time)
- Because any data is possible, any algorithm needs to ask enough questions to produce all orderings.

The facts we can get from that:

1. Each ordering is a different leaf (only one is correct)
2. Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree. Worst number of comparisons is the longest path from root-to-leaf in the decision tree for input size n
3. There is no worst-case running time better than the height of a tree with <*num possible orderings*> leaves

How many possible orderings?

- Assume we have *n* elements to sort. How many *permutations* of the elements (possible orderings)?
  - For simplicity, assume none are equal (no duplicates)

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]

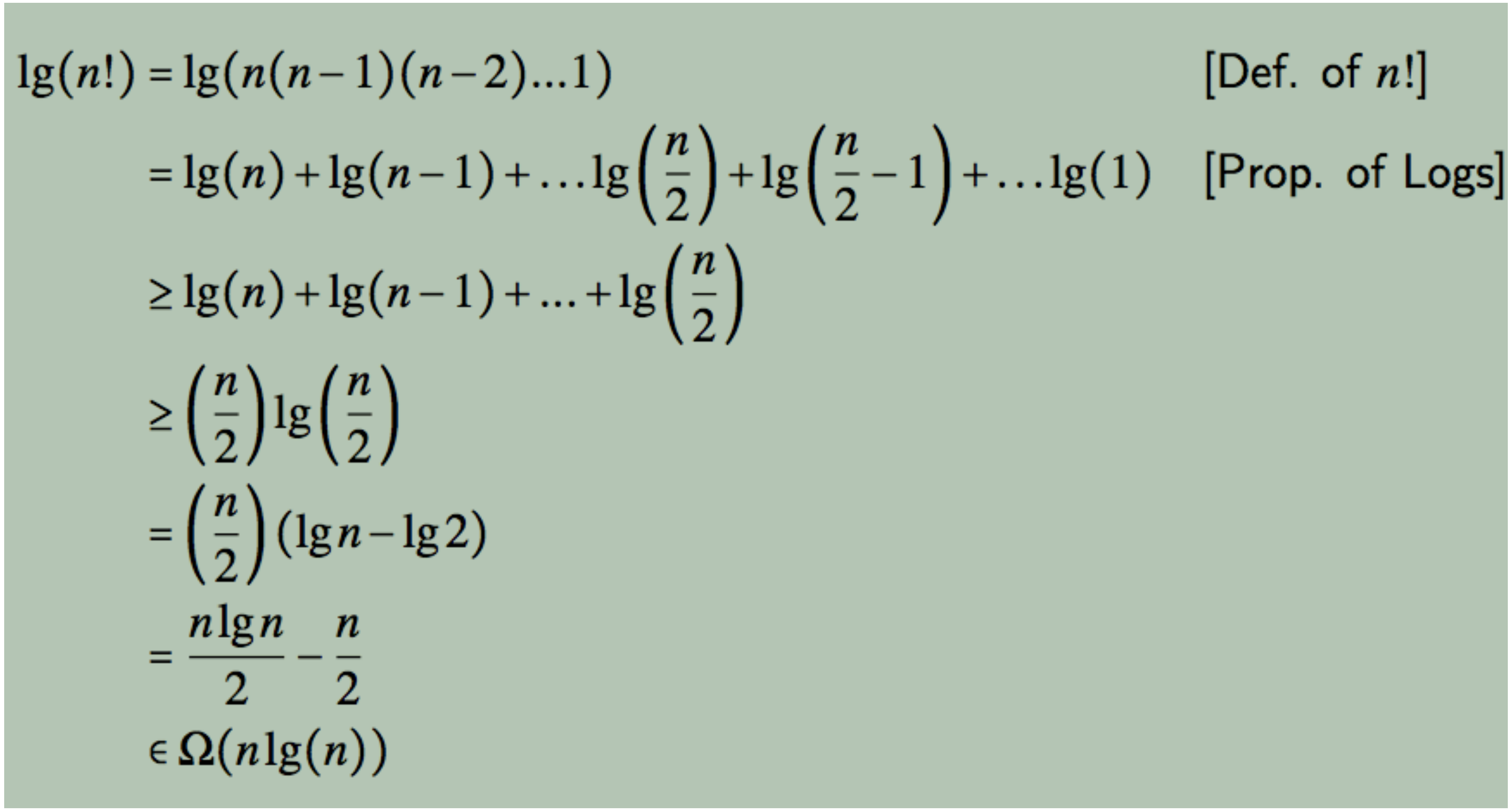
In general, *n* choices for least element, *n*-1 for next, *n*-2 for next, …

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

That means with n! possible leaves, **best height for tree is log(n!)**, given that **best case tree** splits leaves in half at each branch

What does that mean for runtime?

That proves runtime is at least **Ω**(**`log`** (*n!)*). Can we write that more clearly?



$$\begin{aligned}
\lg(n!) &= \lg(n(n-1)(n-2)\ldots 1) && \text{[Def. of } n!\text{]}\\
&= \lg(n)+\lg(n-1)+\ldots\lg\left(\frac{n}{2}\right)+\lg\left(\frac{n}{2}-1\right)+\ldots\lg(1) && \text{[Prop. of Logs]}\\
&\geq \lg(n)+\lg(n-1)+\ldots+\lg\left(\frac{n}{2}\right)\\
&\geq \left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right)\\
&= \left(\frac{n}{2}\right)(\lg n-\lg 2)\\
&= \frac{n\lg n}{2}-\frac{n}{2}\\
&\in \Omega(n\lg(n))
\end{aligned}$$

Nice! Any sorting algorithm must do *at best* (1/2)*(*n*`log` *n* – *n)* comparisons: **Ω**(*n*`log` *n*)

Sorting: 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

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*

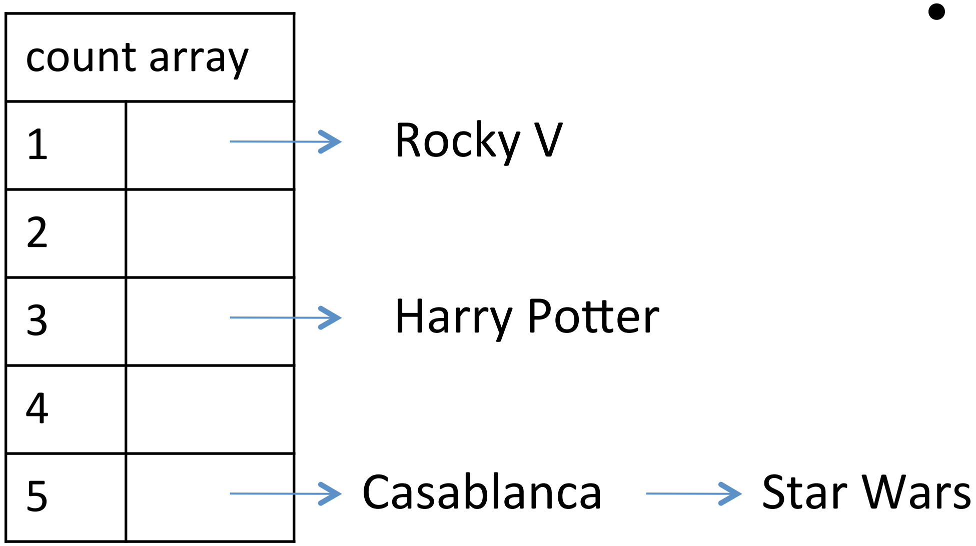
- Good when *K* is smaller (or not much larger) than *n*
  - We don't spend time doing comparisons of duplicates

- 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 non integers

- 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)



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

- Example: Movie ratings; scale 1-5

  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

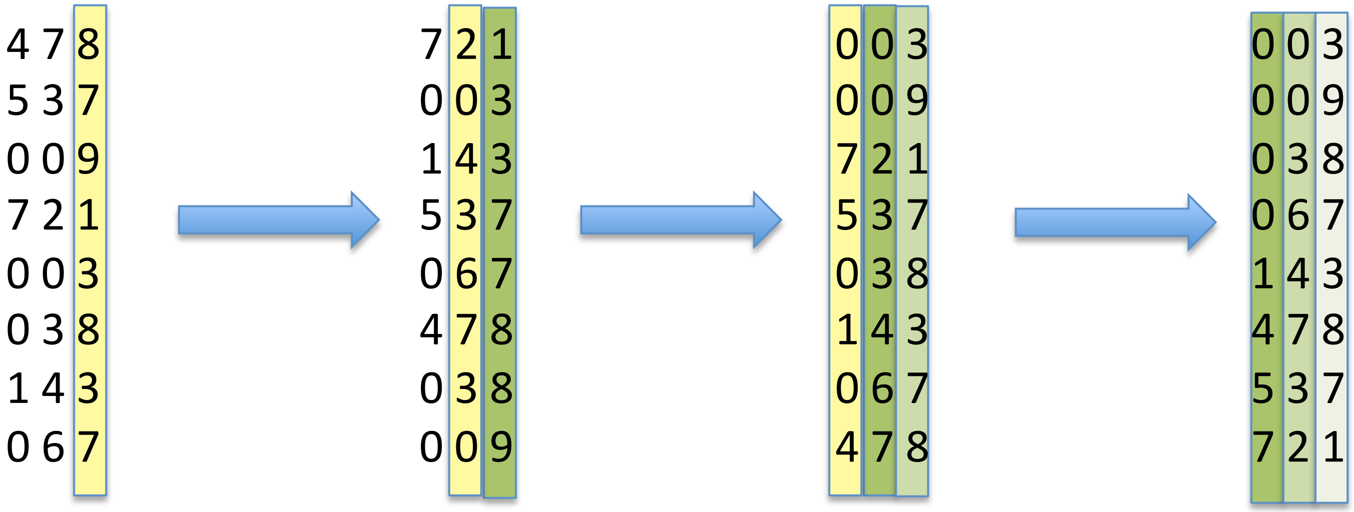
- Radix = “the base of a number system”
  - Examples will use base 10 because we are used to that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128
- **Idea**:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least*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

Radix Sort Example

Radix = 10

Input: 478, 537, 9, 721, 3, 38, 143, 67

3 passes (input is 3 digits at max), on each pass, stable sort the input highlighted in yellow



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

Example

Radix = 10

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

Input:
478
537
9
721
3
38
143
67

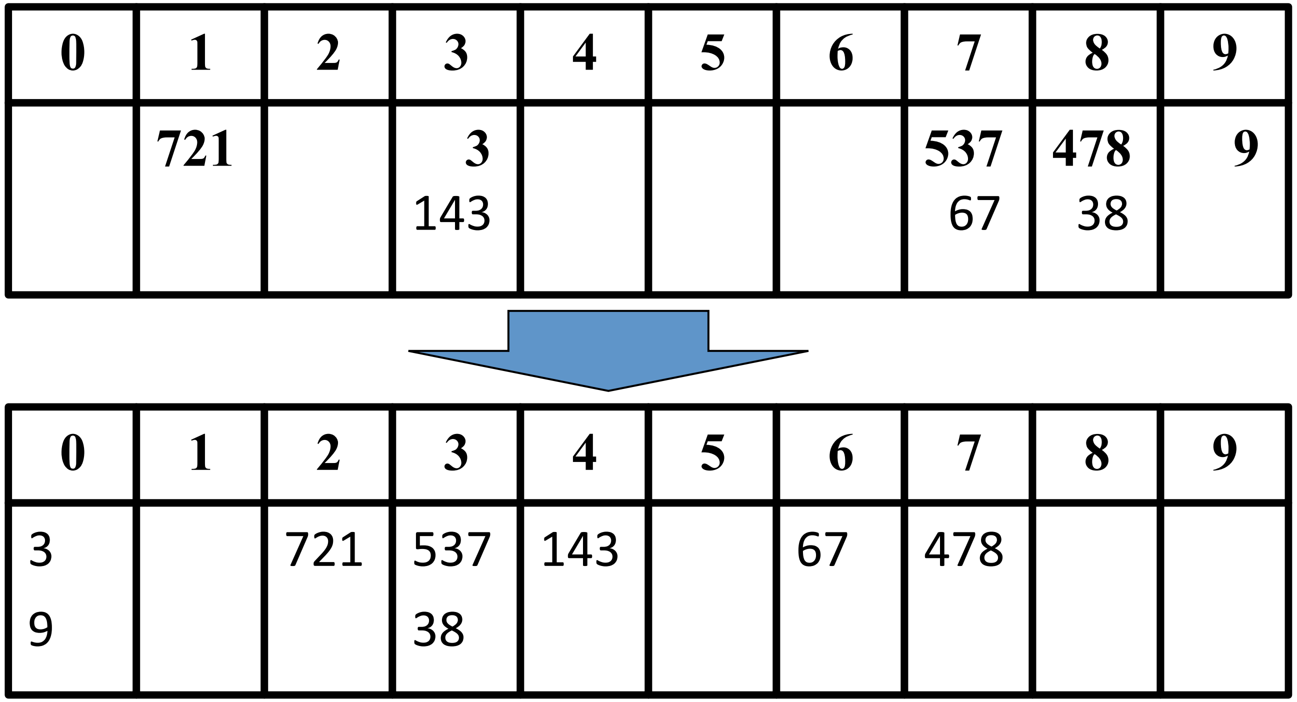
First pass:

bucket sort by ones digit

Order now:
721
003
143
537
067
478
038
009

Example

Radix = 10



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

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

Order was:

721
003
143
537
067
478
038
009

Second pass:

stable bucket sort by tens digit

Order now:

003
009
721
537
038
143
067
478

Example

Radix = 10

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

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

Order was:

003
009
721
537
038
143
067
478

Third pass:

stable bucket sort by 100s digit

Order now:

003
009
038
067
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

Sorting Takeaways

- 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!