CSE 332: Data Structures & Parallelism
Lecture 12: Sorting

Arthur Liu

Summer 2022

Outline

- Sorting and more sorting

Sorting

Great general pre-processing step

- Binary Search
- Let's us find the k^{th} element in $O(1)$ time for any k.

Also, a convenient way to discuss algorithm design principles.

Three goals

Three things you might want in a sorting algorithm:

- In-Place
  - Only use $O(1)$ extra 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.
  - Example: sort by first name, then by last name.
- Fast

$[0 \ 3 \ 7_0 \ 11 \ 7_1]$

$[0 \ 3 \ 7_0 \ 7_1 \ 11]$

Insertion Sort

How you sort a hand of cards.

Maintain a sorted subarray at the front.

Start with one element.

While(your subarray is not the full array)

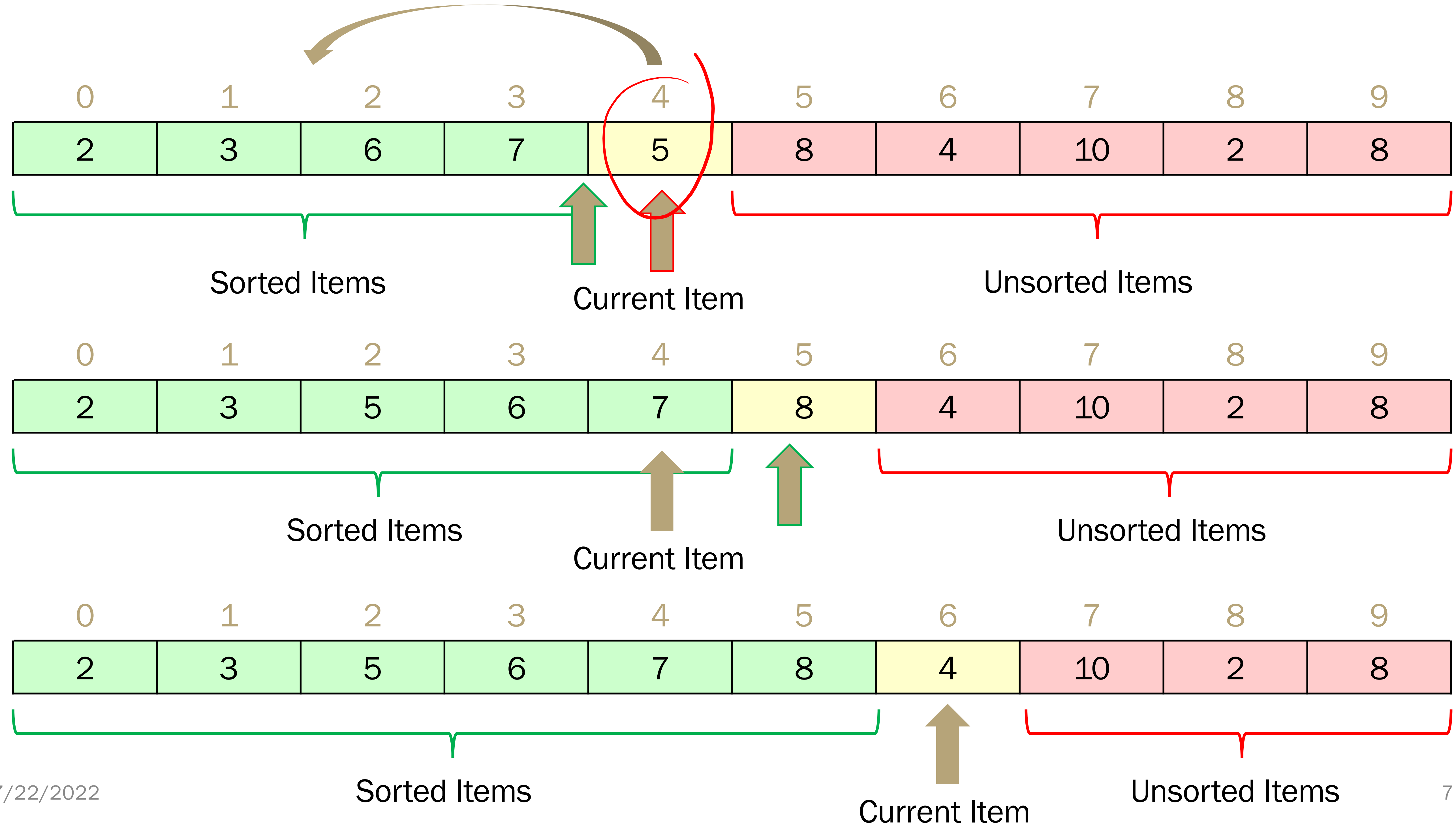
    Take the next element not in your subarray

    Insert it into the sorted subarray

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
    }
}
```

Insertion Sort



Insertion Sort Analysis

Stable? Yes! (If you're careful)

In Place? Yes!

Running time:

- Best Case: $O(n)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$

$$\sum_{i=0}^{n} i = O(n^2)$$

_________ 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.
  - Append it at the end of the sorted part.

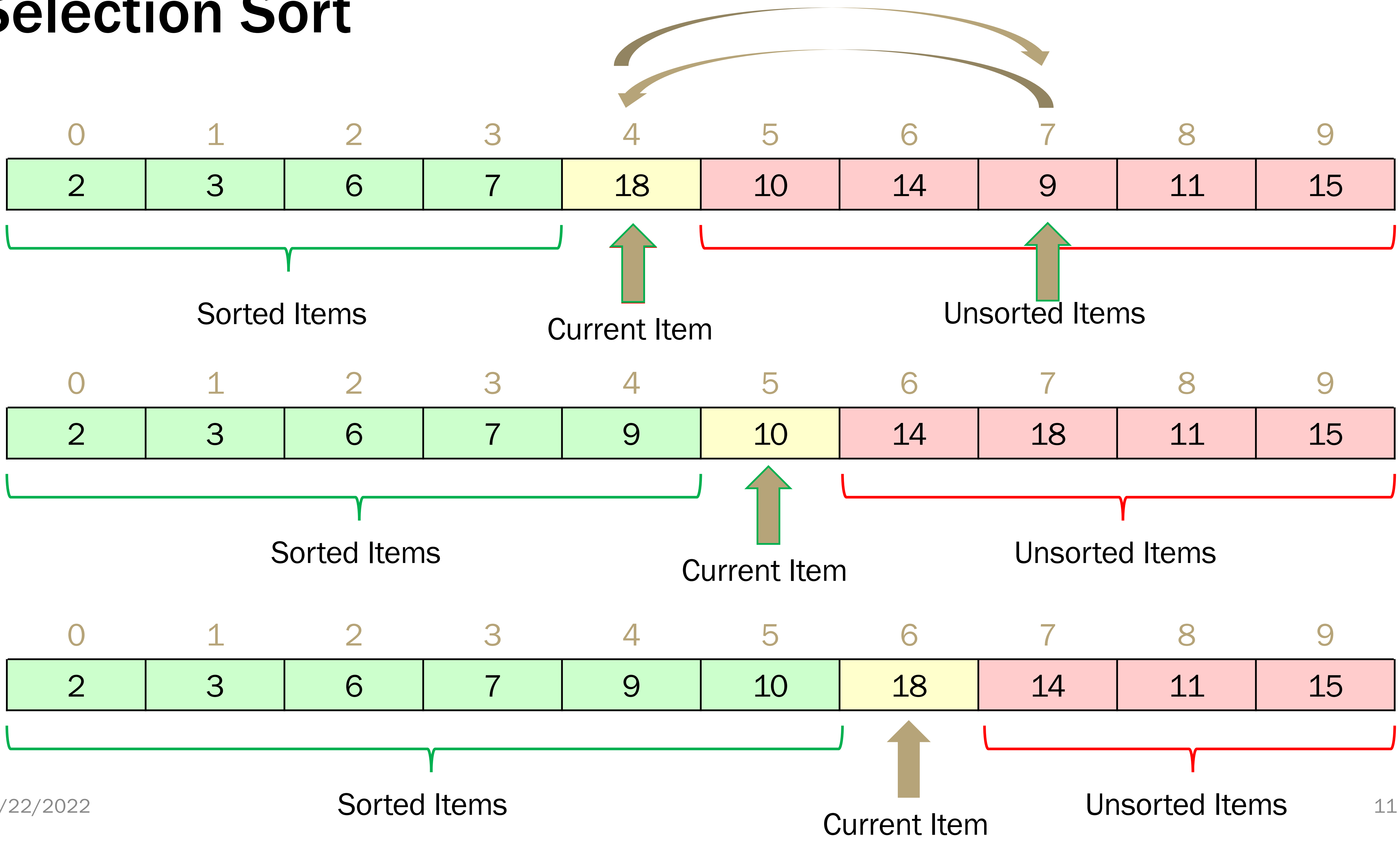
Selection Sort

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 the remainder of the array*
  - Append it at the end of the sorted part.

Running time $O(n^2)$

Selection Sort



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 the remainder of the array*
  - Append it at the end of the sorted part.

Running time $O(n^2)$

Can we do better? With a data structure?

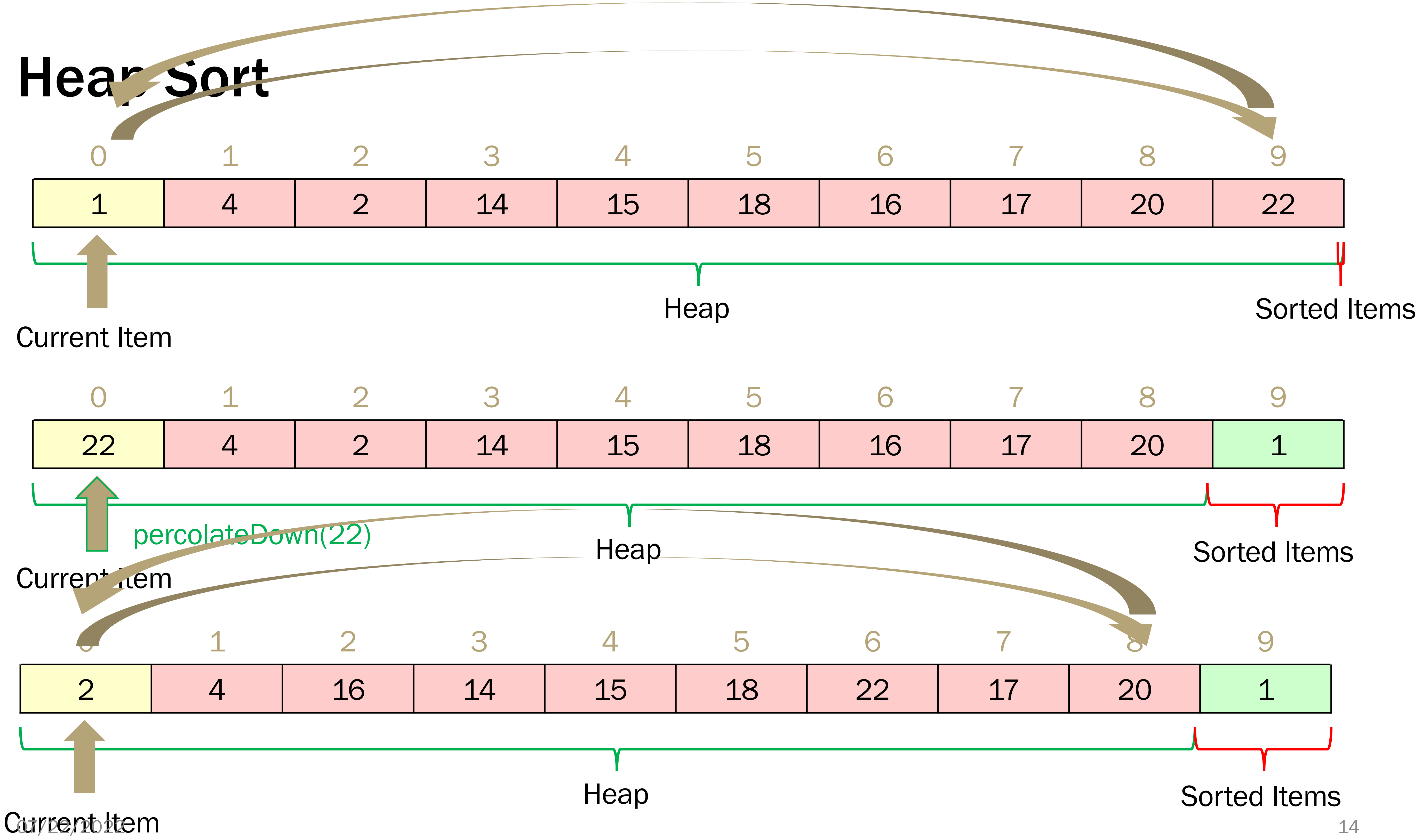
___Heap___ Sort

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*
  - Append it at the end of the sorted part.

Running time $O(n \log n)$

Heap Sort


0 1 2 3 4 5 6 7 8 9
1 4 2 14 15 18 16 17 20 22
Heap
Sorted Items
Current Item
0 1 2 3 4 5 6 7 8 9
22 4 2 14 15 18 16 17 20 1
percolateDown(22)
Heap
Sorted Items
Current Item
0 1 2 3 4 5 6 7 8 9
2 4 16 14 15 18 22 17 20 1
Heap
Sorted Items
Current Item

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? $O(n \log n)$

Heap Sort

- Our first step is to make a heap. Does using `buildHeap` instead of inserts improve the running time?
- Not in a big-O sense (though we did by a constant factor).

In place: Yes

Stable: No

Quick Recap

	Run-time	Stable	Space
Insertion Sort	Best Case: $O(N)$ Worst Case: $O(N^2)$ Average Case: $O(N^2)$	Yes	$O(1)$
Selection Sort	$O(N^2)$	No	$O(1)$
Heap Sort	$O(N \log N)$	No	$O(1)$

Poll Everywhere

pollev.com/artliu

We just saw Heap Sort, what about an "AVL Sort"?

1. How would the algorithm work?

2. What is the worst-case runtime?

3. Would this be a good alternative to heap 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

Divide and conquer

Very important technique in algorithm design

1. Divide problem into smaller parts

2. Solve the parts independently
   - Think recursion
   - Or potential parallelism

3. Combine solution of parts to produce overall solution

Ex: Sort each half of the array, combine together; to sort each half, split into halves…

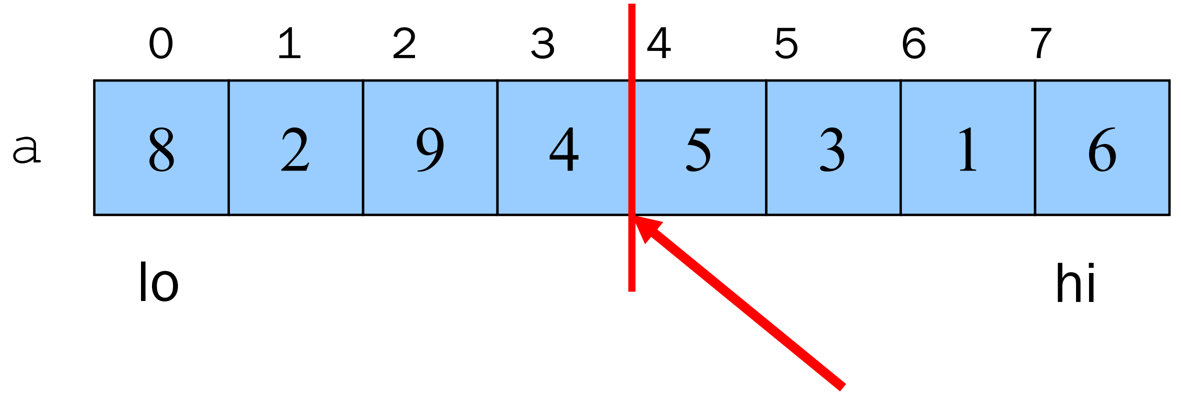
Divide-and-conquer sorting

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 those less-than pivot and those greater-than pivot
   Sort the two divisions (recursively on each)
   Answer is [***sorted-less-than*** then ***pivot*** then ***sorted-greater-than***]

Mergesort

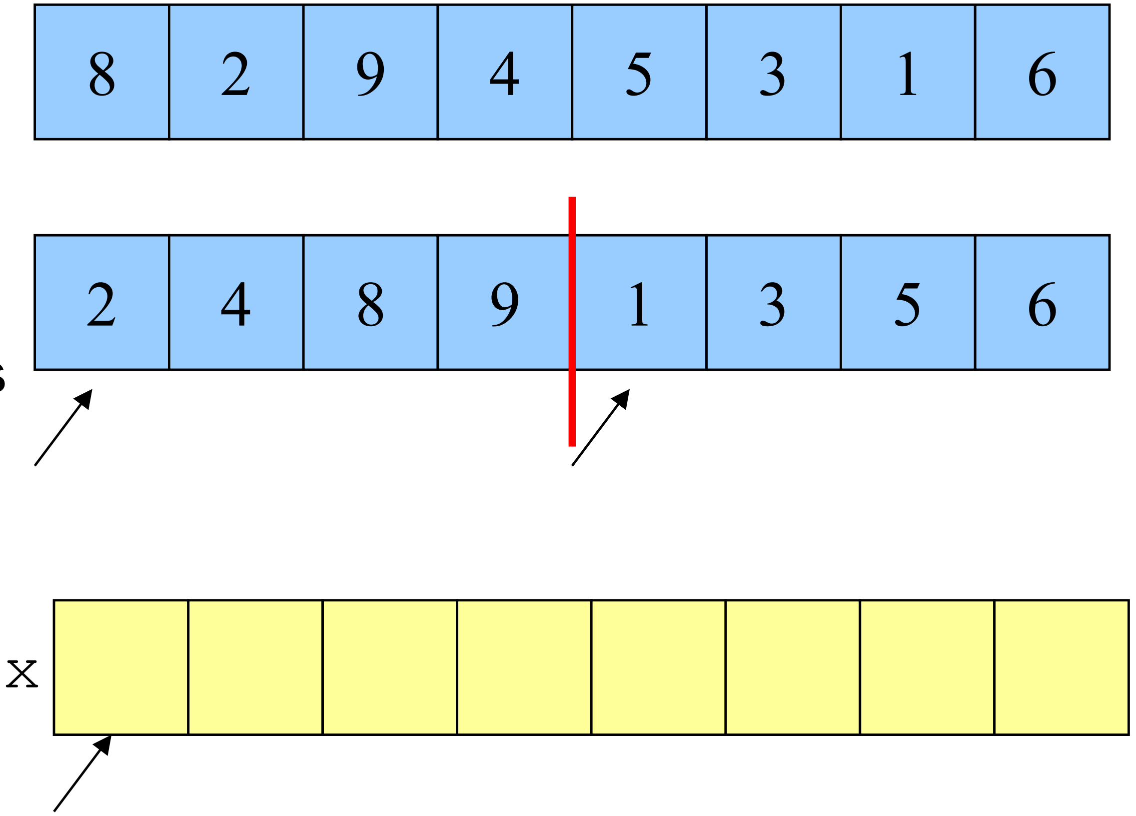


- To sort array from position **`lo`** to position **`hi`**:
  - If range is 1 element long, it's sorted! (Base case)
  - Else, split into two halves:
    - Sort from **`lo`** to **`(hi+lo)/2`**
    - Sort from **`(hi+lo)/2`** to **`hi`**
    - Merge the two halves together

- Merging takes two sorted parts and sorts everything
  - *O*(*n*) but requires auxiliary space...

Example, focus on merging

Start with: a



After we return from left and right recursive calls (pretend it works for now) a

Merge:

Use 3 “fingers” aux

and 1 more array

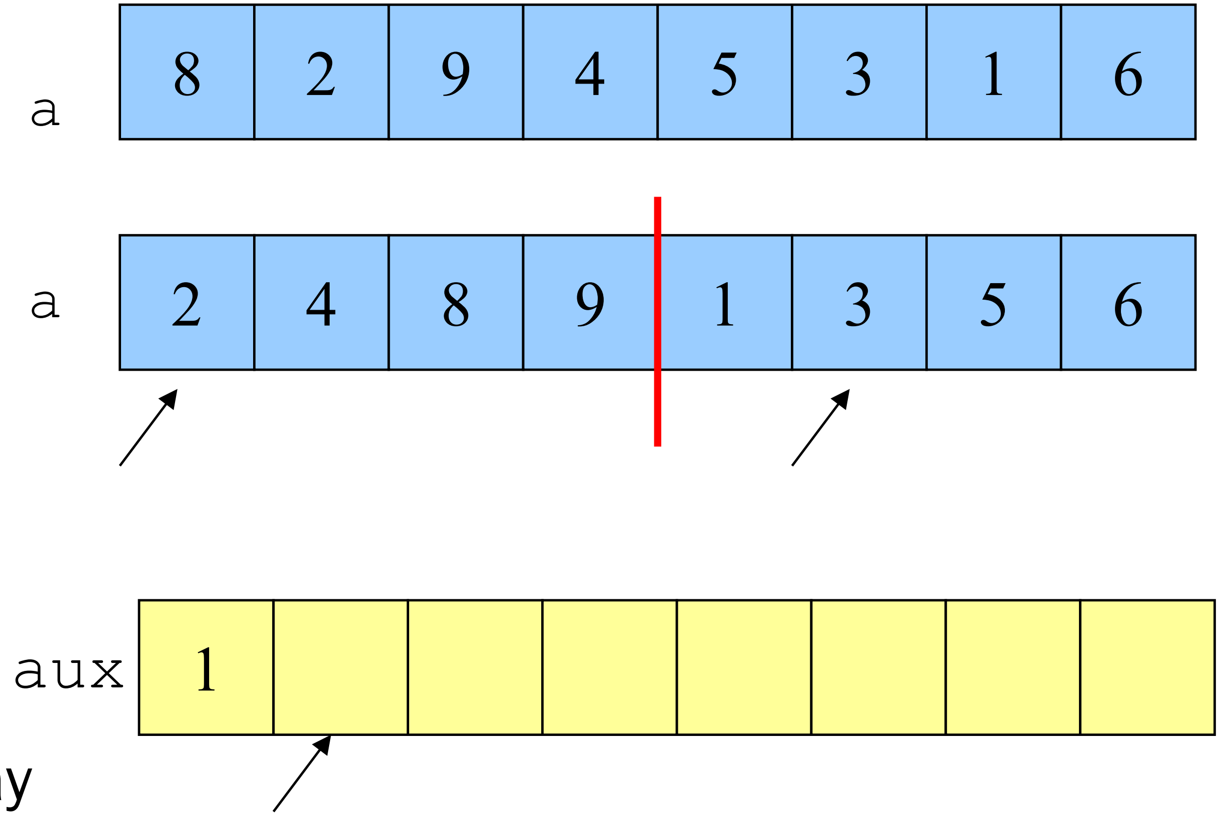
(After merge, copy back to original array)

Example, focus on merging

Start
with:

After recursion:

(not magic ☺)



Merge:

Use 3 “fingers”

and 1 more array

(After merge,
copy back to
original array)

Example, focus on merging

Start with:

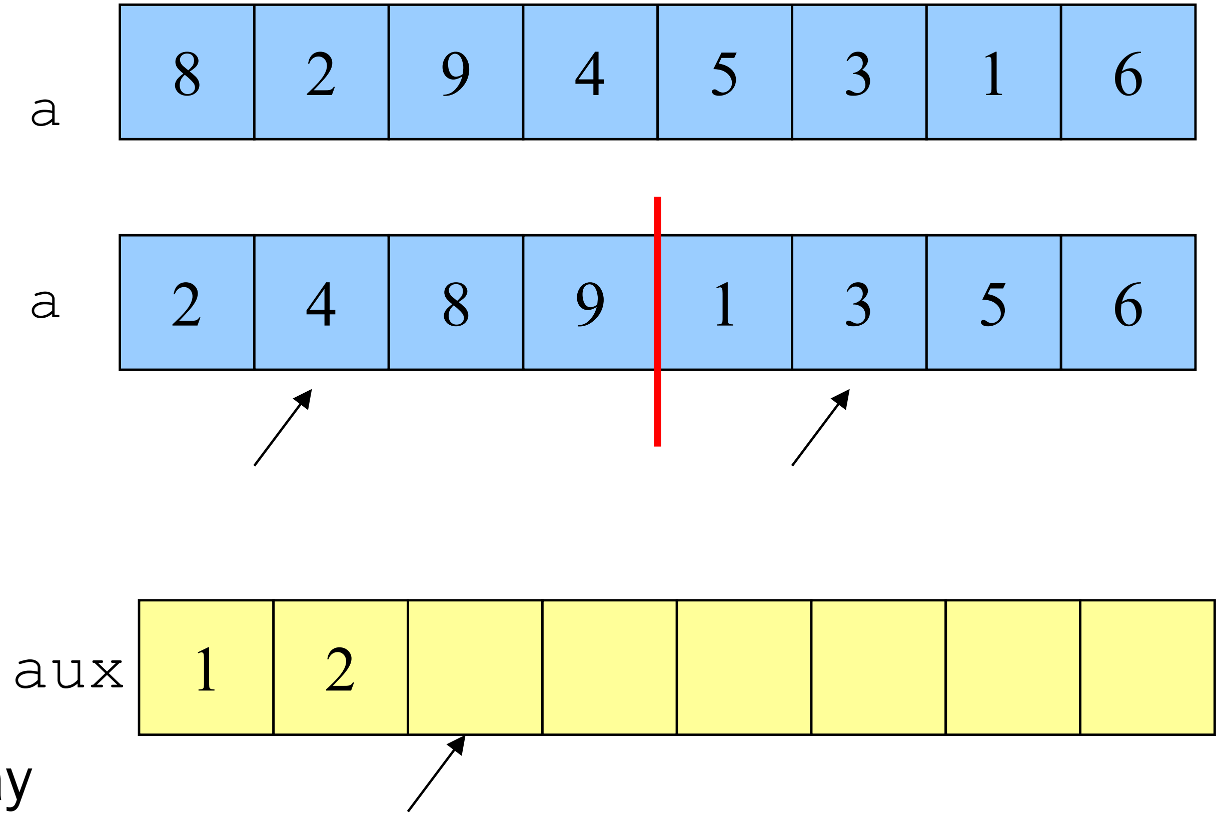
After recursion:

(not magic ☺)

Merge:

Use 3 “fingers”

and 1 more array



(After merge,
copy back to
original array)

Example, focus on merging

Start
with:

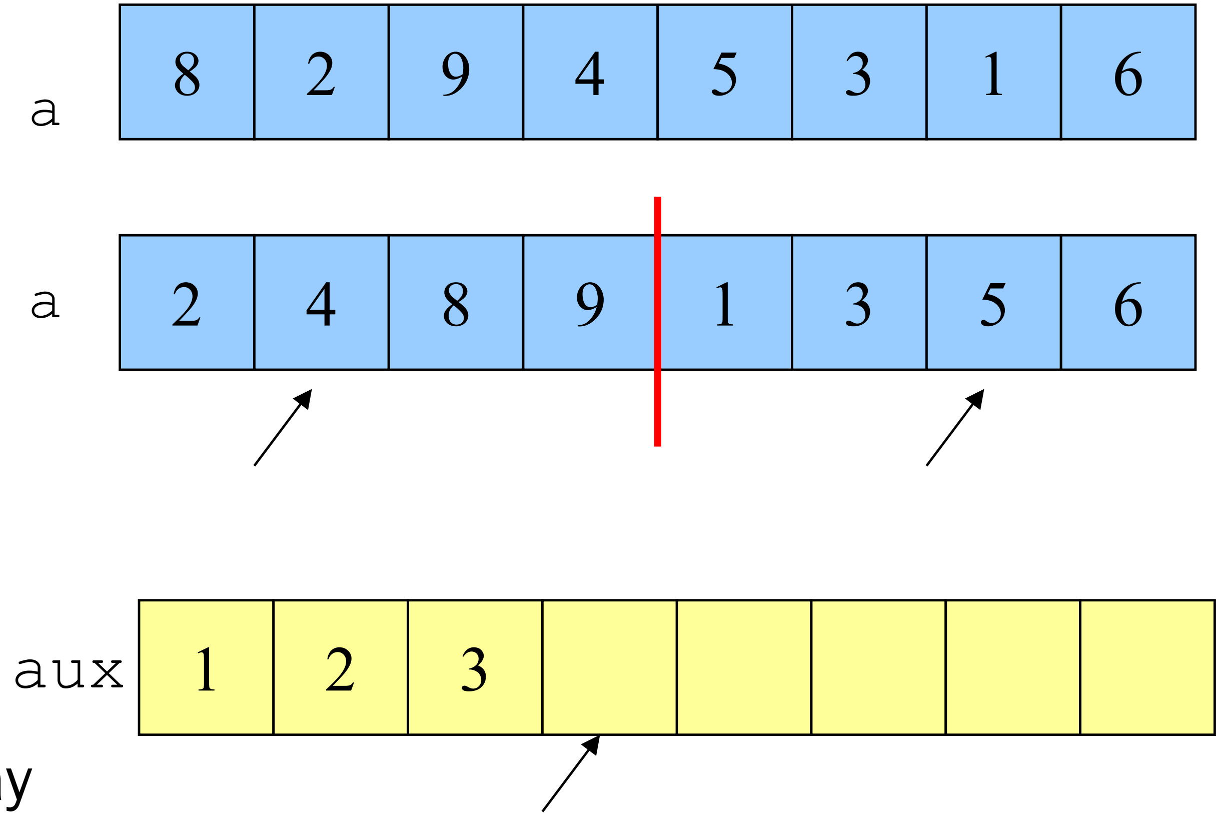
After recursion:

(not magic ☺)

Merge:

Use 3 “fingers”

and 1 more array



(After merge,
copy back to
original array)

Example, focus on merging

Start
with:

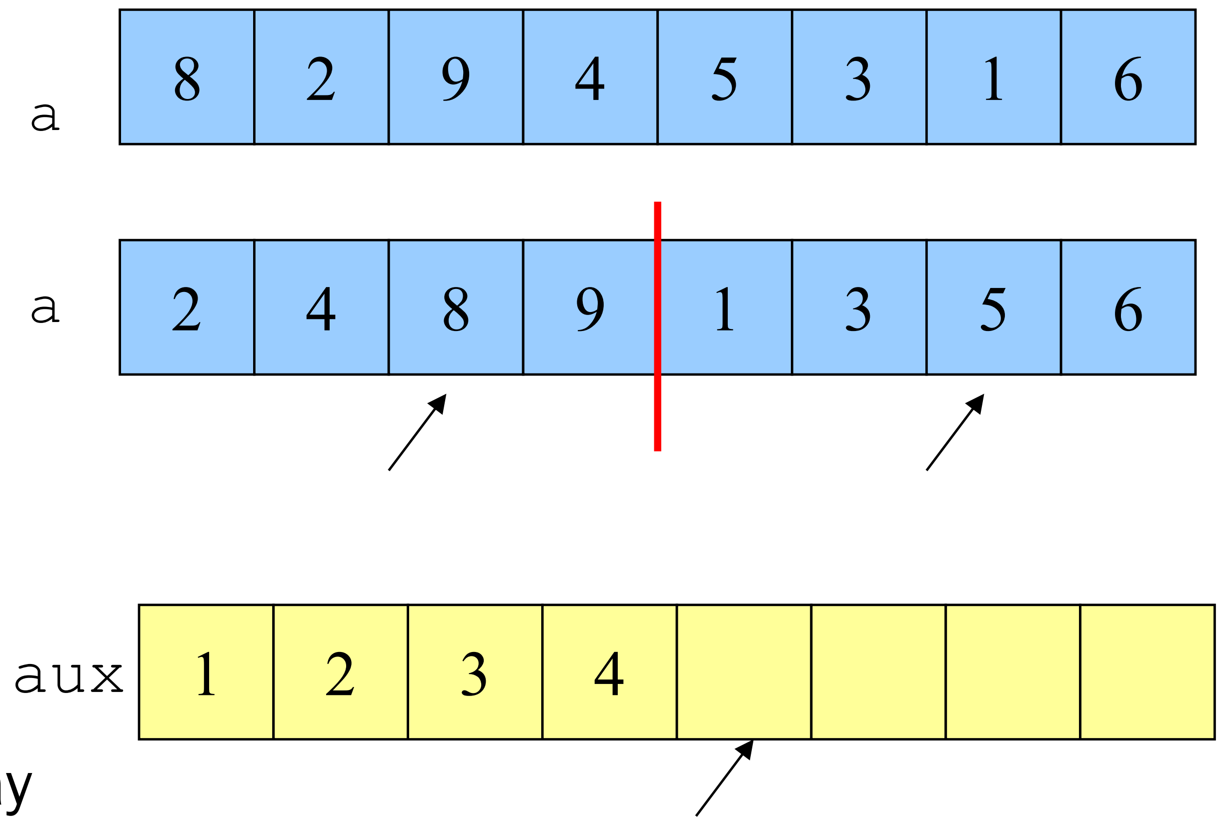
After recursion:

(not magic ☺)

Merge:

Use 3 “fingers”

and 1 more array



(After merge,
copy back to
original array)

Example, focus on merging

Start
with:

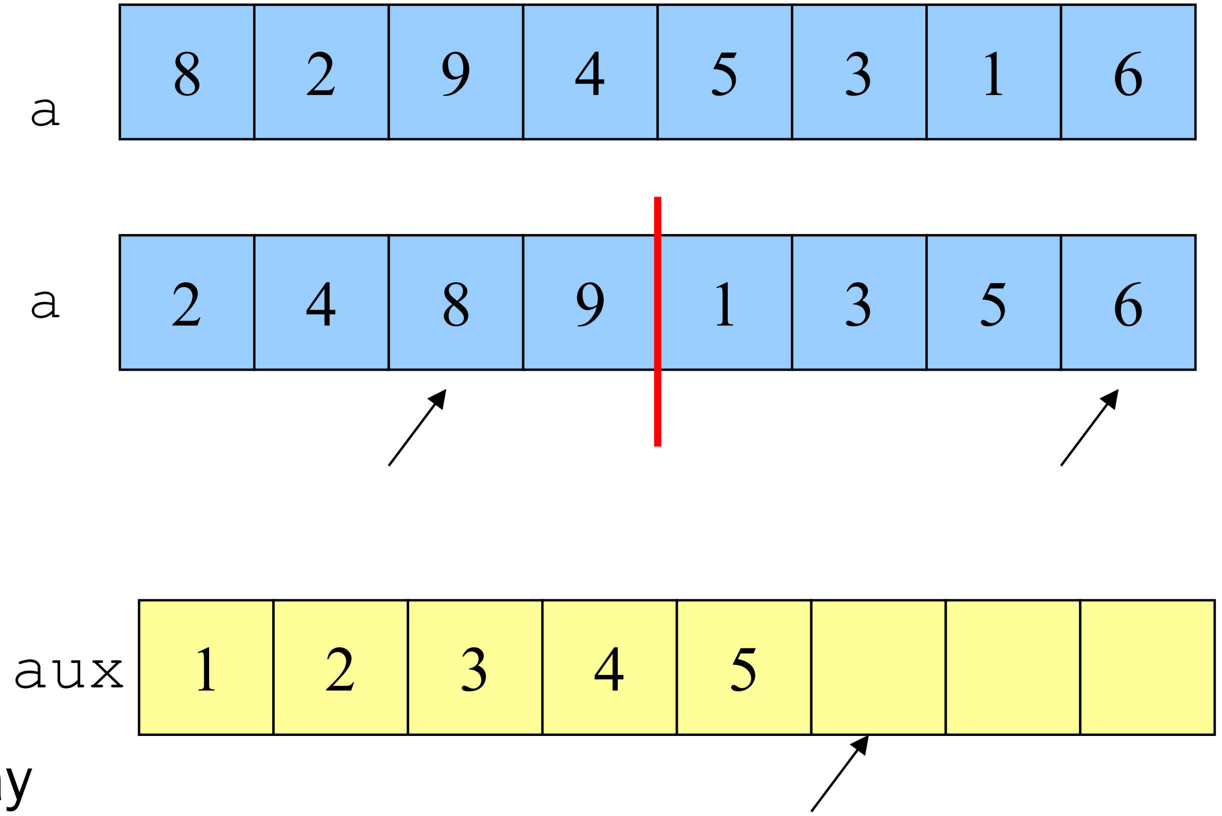
After recursion:

(not magic ☺)

Merge:

Use 3 “fingers”

and 1 more array



(After merge,
copy back to
original array)

Example, focus on merging

Start
with:

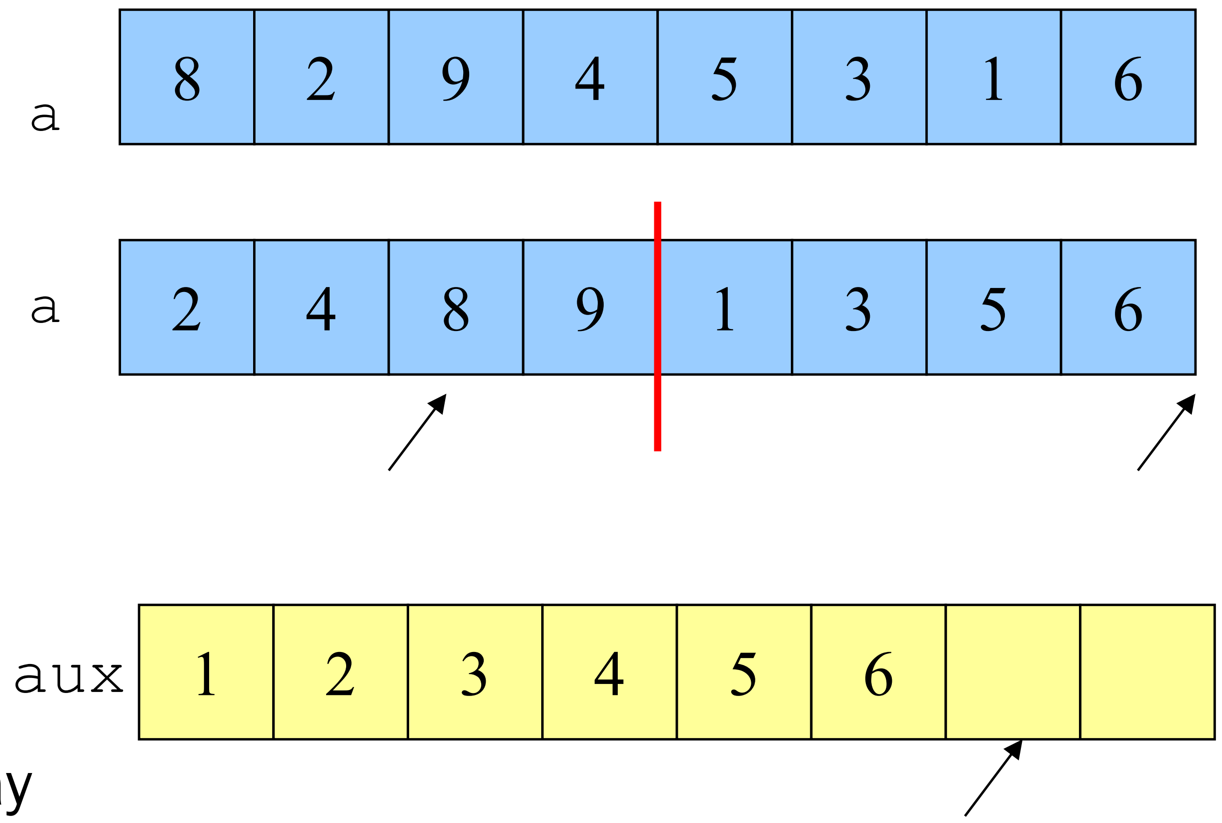
After recursion:

(not magic ☺)

Merge:

Use 3 “fingers”

and 1 more array



(After merge,
copy back to
original array)

Example, focus on merging

Start with:

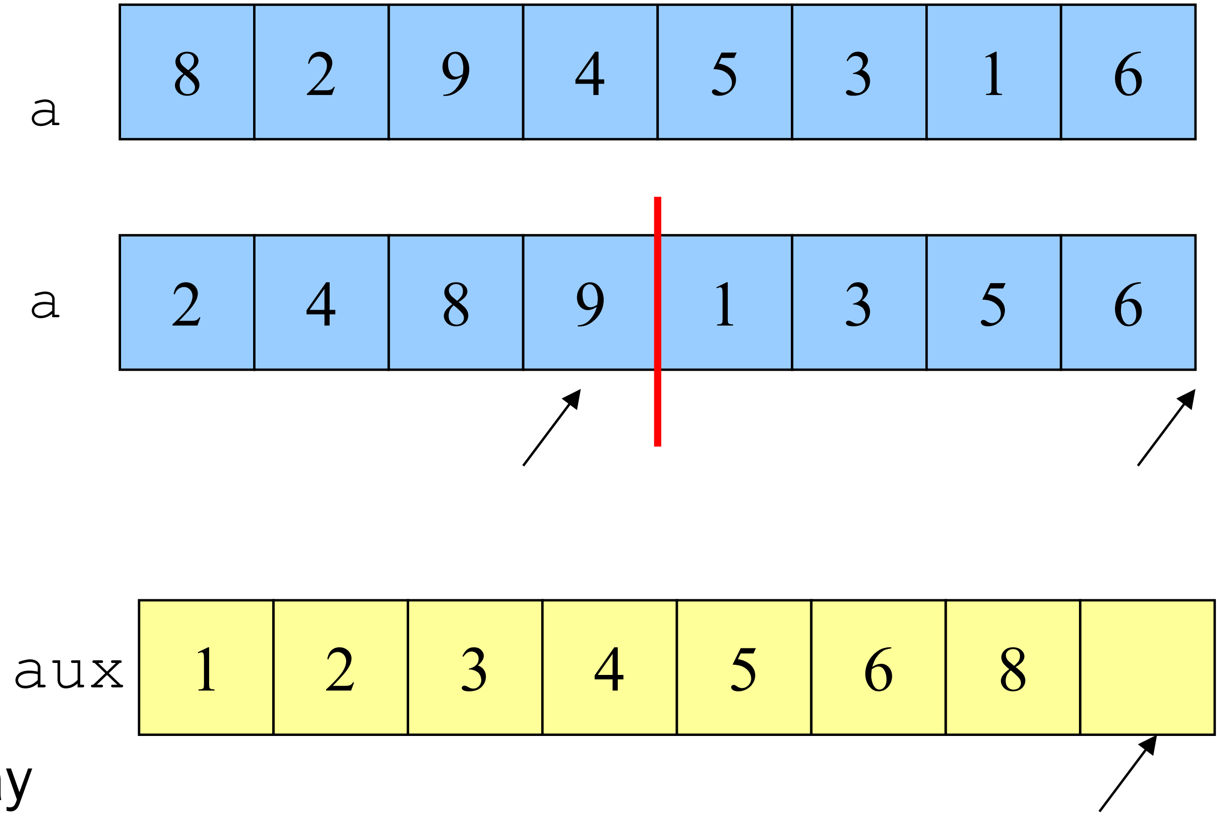
After recursion:

(not magic ☺)

Merge:

Use 3 “fingers”

and 1 more array



(After merge,
copy back to
original array)

Example, focus on merging

Start with:

a	8	2	9	4	5	3	1	6

After recursion:

(not magic ☺)

a	2	4	8	9	1	3	5	6

Merge:

Use 3 “fingers”

and 1 more array

aux	1	2	3	4	5	6	8	9

(After merge,
copy back to
original array)

Example, focus on merging

Start with:

After recursion:

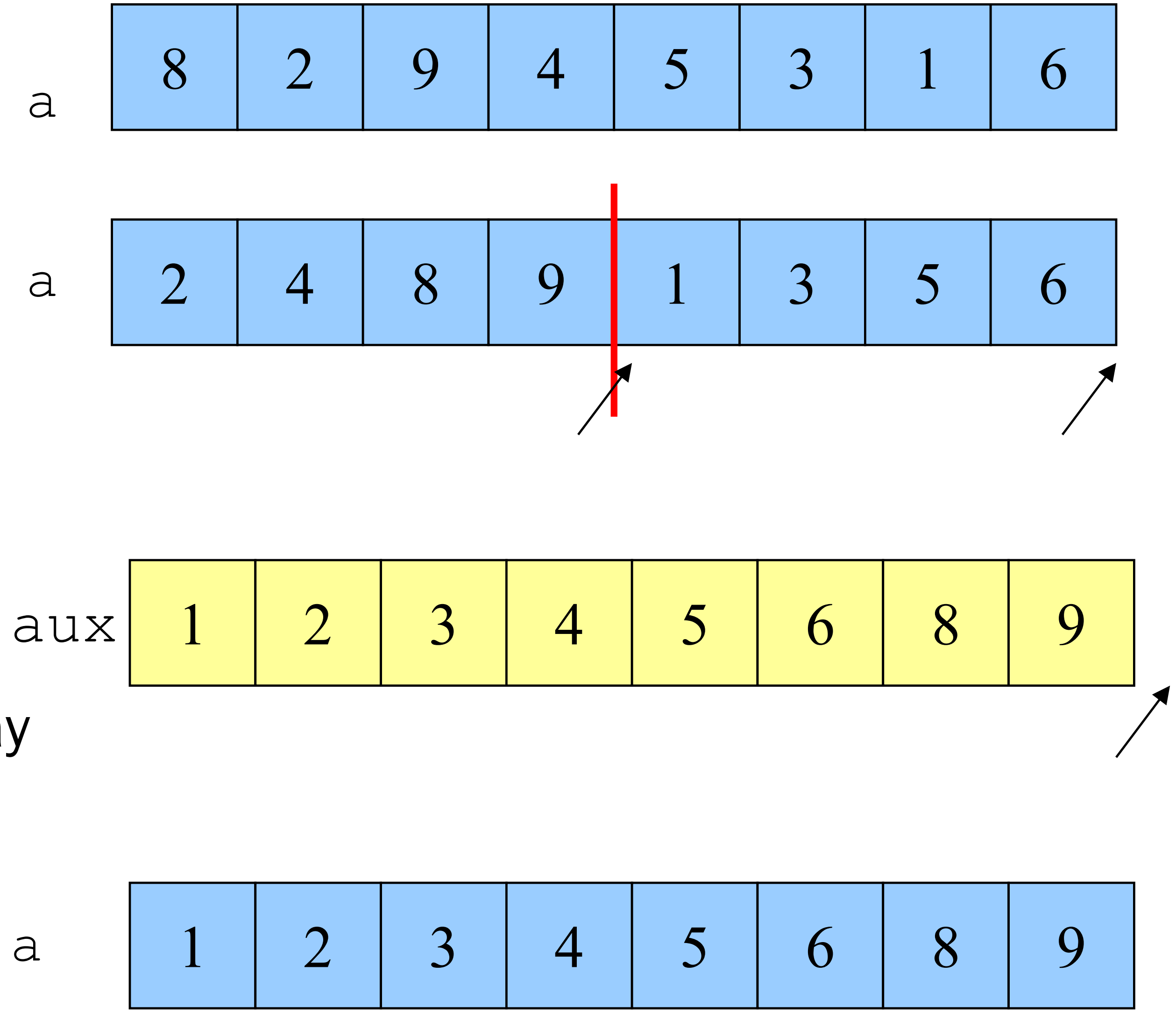
(not magic ☺)

Merge:

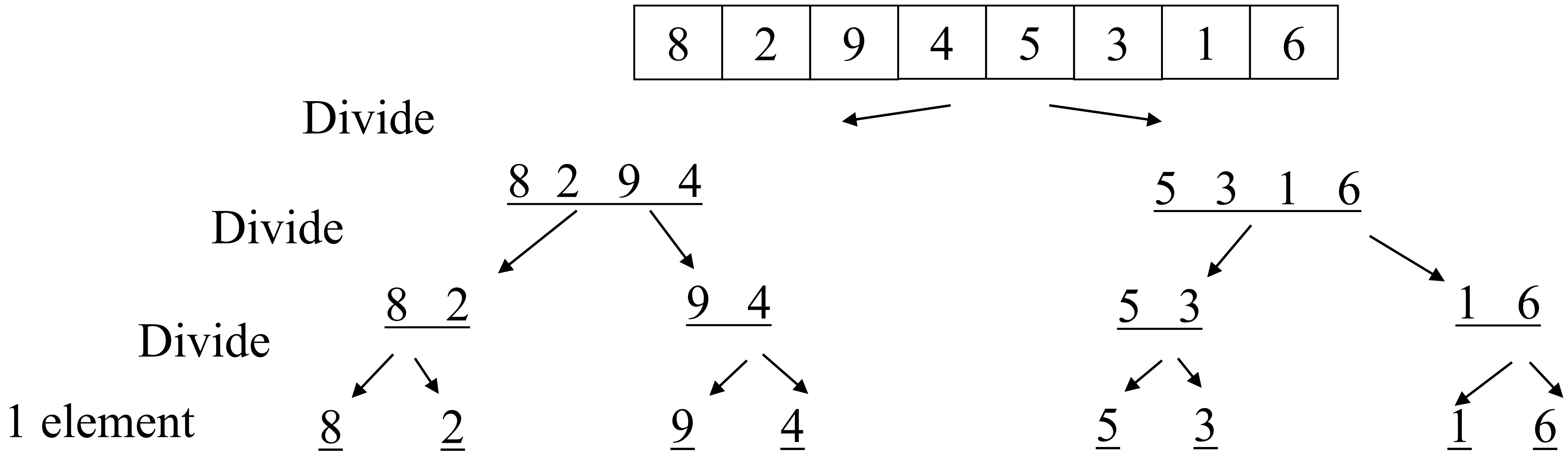
Use 3 "fingers"

and 1 more array

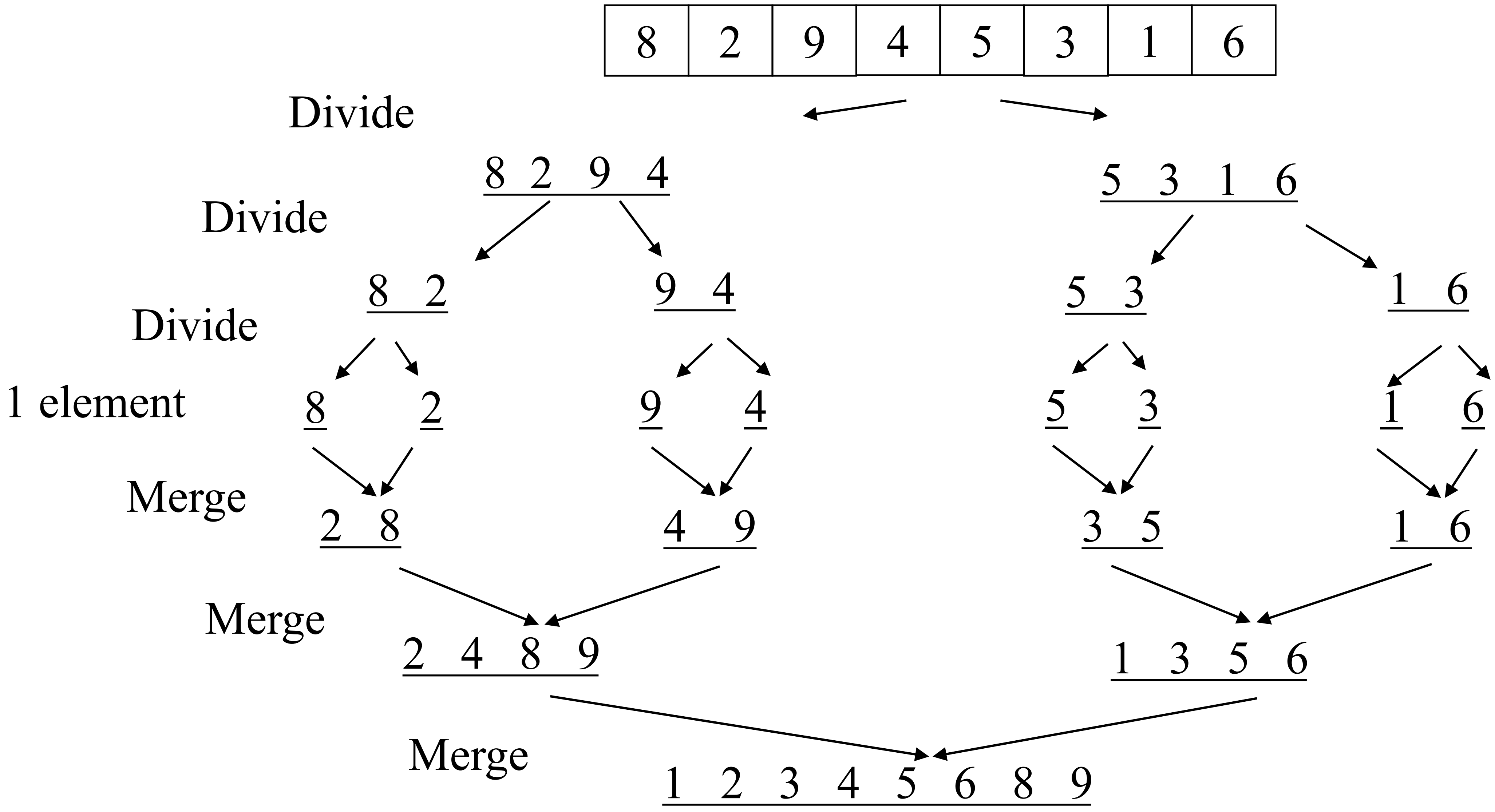
(After merge, copy back to original array)



Mergesort example: Recursively splitting list in half



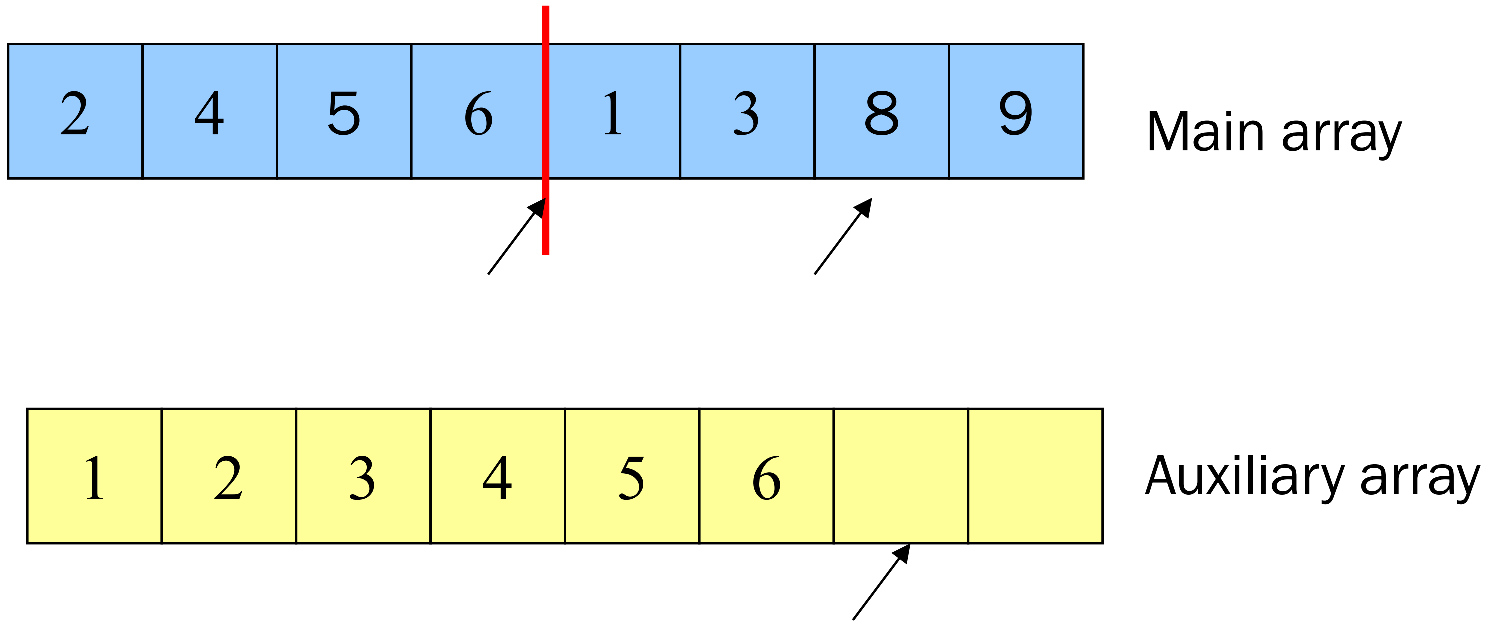
Mergesort example: Merge as we return from recursive calls



When a recursive call ends, it's sub-arrays are each in order; just need to merge them in order together

Mergesort, some details: saving a little time

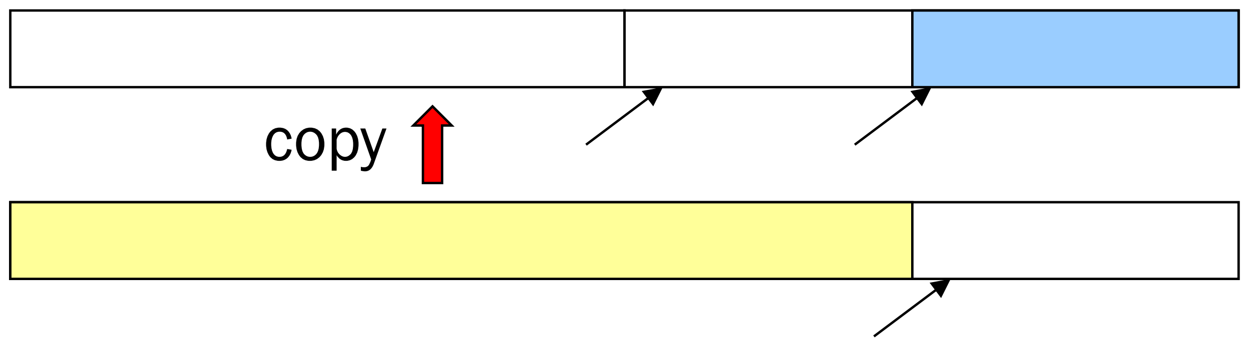
- What if the final steps of our merging looked like the following:



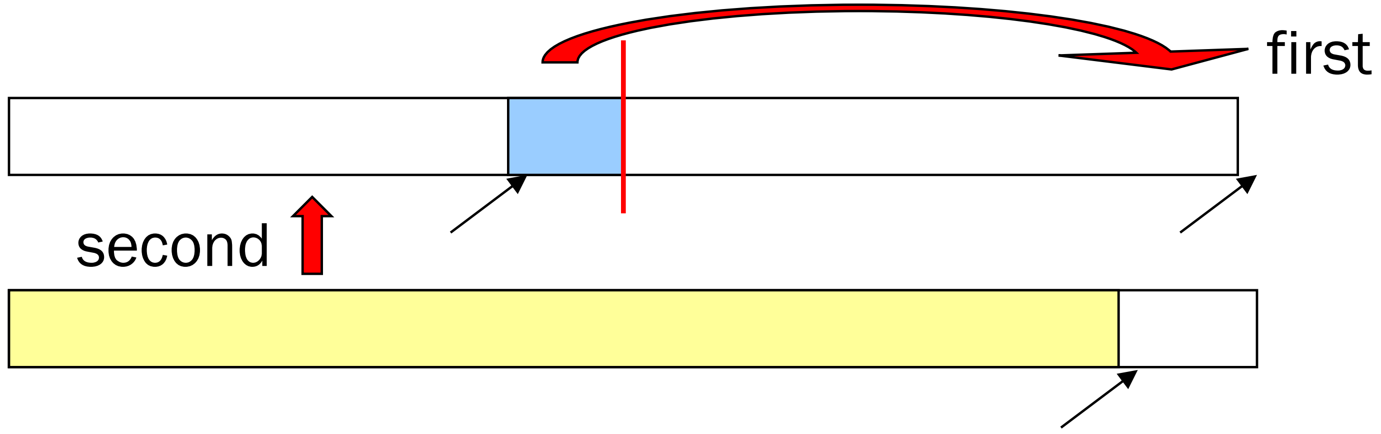
- Seems kind of wasteful to copy 8 & 9 to the auxiliary array just to copy them immediately back…

Mergesort, some details: saving a little time

- Unnecessary to copy remainder over to auxiliary array
  - If left-side finishes first, just stop the merge & copy the auxiliary array:



  - If right-side finishes first, copy remainder directly into right side, then copy auxiliary array



Some details: saving space / copying

Simplest / worst approach:

Use a new auxiliary array of size **`(hi-lo)`** for every merge

Returning from a recursive call? Allocate a new array!

Better:

Reuse same auxiliary array of size **`n`** for every merging stage

Allocate auxiliary array at beginning, use throughout

Best (but a little tricky), (saves time):

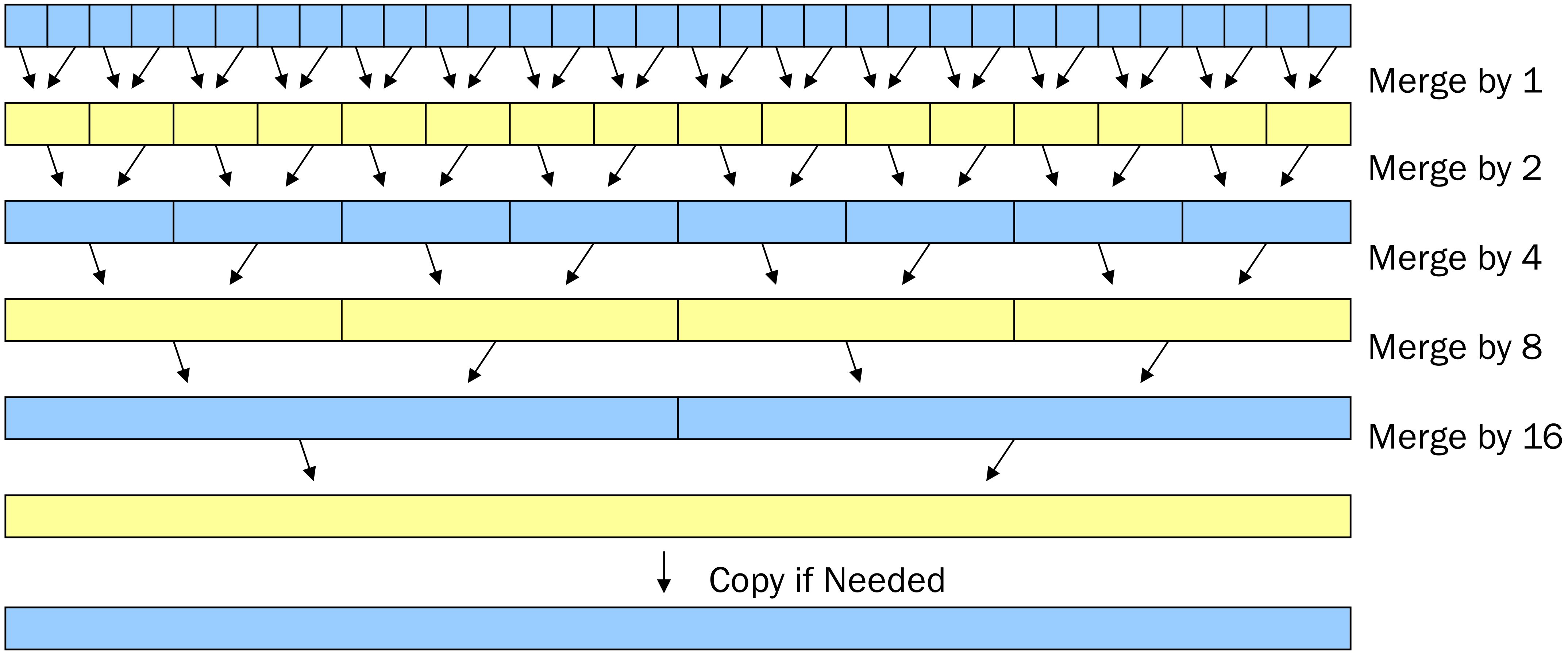
Don't copy back – at 2^{nd}, 4^{th}, 6^{th}, … merging stages, use the original array as the auxiliary array and vice-versa

- Need one copy at end if number of stages is odd

Picture of the "best" from previous slide: Allocate one auxiliary array, switch each step

First recurse down to lists of size 1

As we return from the recursion, switch off arrays



Linked lists and big data

We defined the sorting problem as over an array, but sometimes you want to sort linked lists

One approach:

- Convert to array: $O(n)$
- Sort: $O(n \texttt{ log } n)$
- Convert back to list: $O(n)$

Or: mergesort works very nicely on linked lists directly

- heapsort and quicksort do not
- insertion sort and selection sort do but they're slower

Mergesort is also the sort of choice for external sorting

- Linear merges minimize disk accesses

Mergesort Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time (and space):

To sort n elements, we:

- Return immediately if n=1
- Else do 2 subproblems of size $n/2$ and then an $O(n)$ merge

Recurrence relation?

Mergesort Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time (and space):

To sort *n* elements, we:

- Return immediately if $n=1$
- Else do 2 subproblems of size $n/2$ and then an $O(n)$ merge

Recurrence relation:

$$T(1) = c_1$$

$$T(n) = 2T(n/2) + c_2 n + c_3$$

Mergesort Recurrence

(For simplicity let constants be 1 – no effect on asymptotic answer)

$T(1) = 1$

$T(n) = 2T(n/2) + n$

$= 2(2T(n/4) + n/2) + n$

$= 4T(n/4) + 2n$

$= 4(2T(n/8) + n/4) + 2n$

$= 8T(n/8) + 3n$

…. (after k expansions)

$= 2^k T(n/2^k) + kn$

So total is $2^k T(n/2^k) + kn$ where

$n/2^k = 1$, i.e., $\log n = k$

That is, $2^{\log n} T(1) + n \log n$

$= n + n \log n$

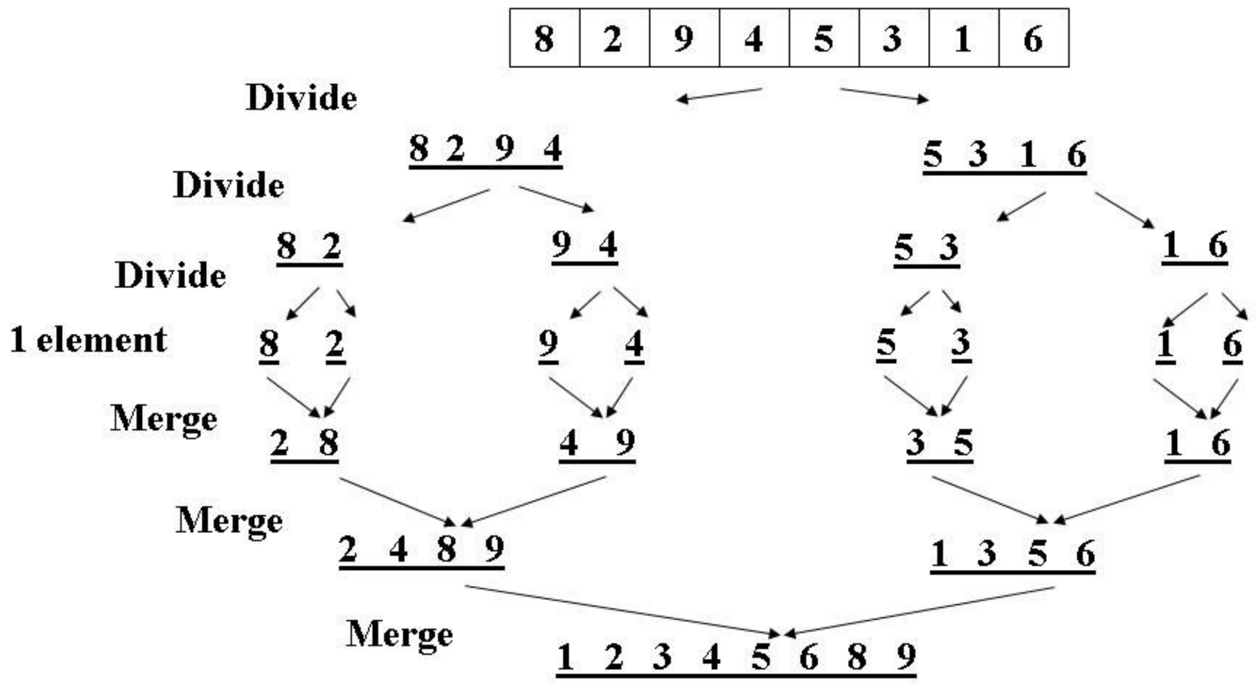
$= O(n \log n)$

Or more intuitively…

This recurrence comes up often enough you should just "know" it's $O(n \log n)$

Merge sort is relatively easy to intuit (best, worst, and average):

- The recursion "tree" will have `log` n height
- At each level we do a *total* amount of merging equal to n



Quicksort

- Also uses divide-and-conquer
  - Recursively chop into halves
  - But, instead of doing all the work as we merge together, we'll do all the work as we recursively split into halves
  - Also unlike MergeSort, does not need auxiliary space

- $O(n \texttt{ log } n)$ on average ☺, but $O(n^2)$ worst-case ☹
  - MergeSort is always O(nlogn)
  - So why use QuickSort?

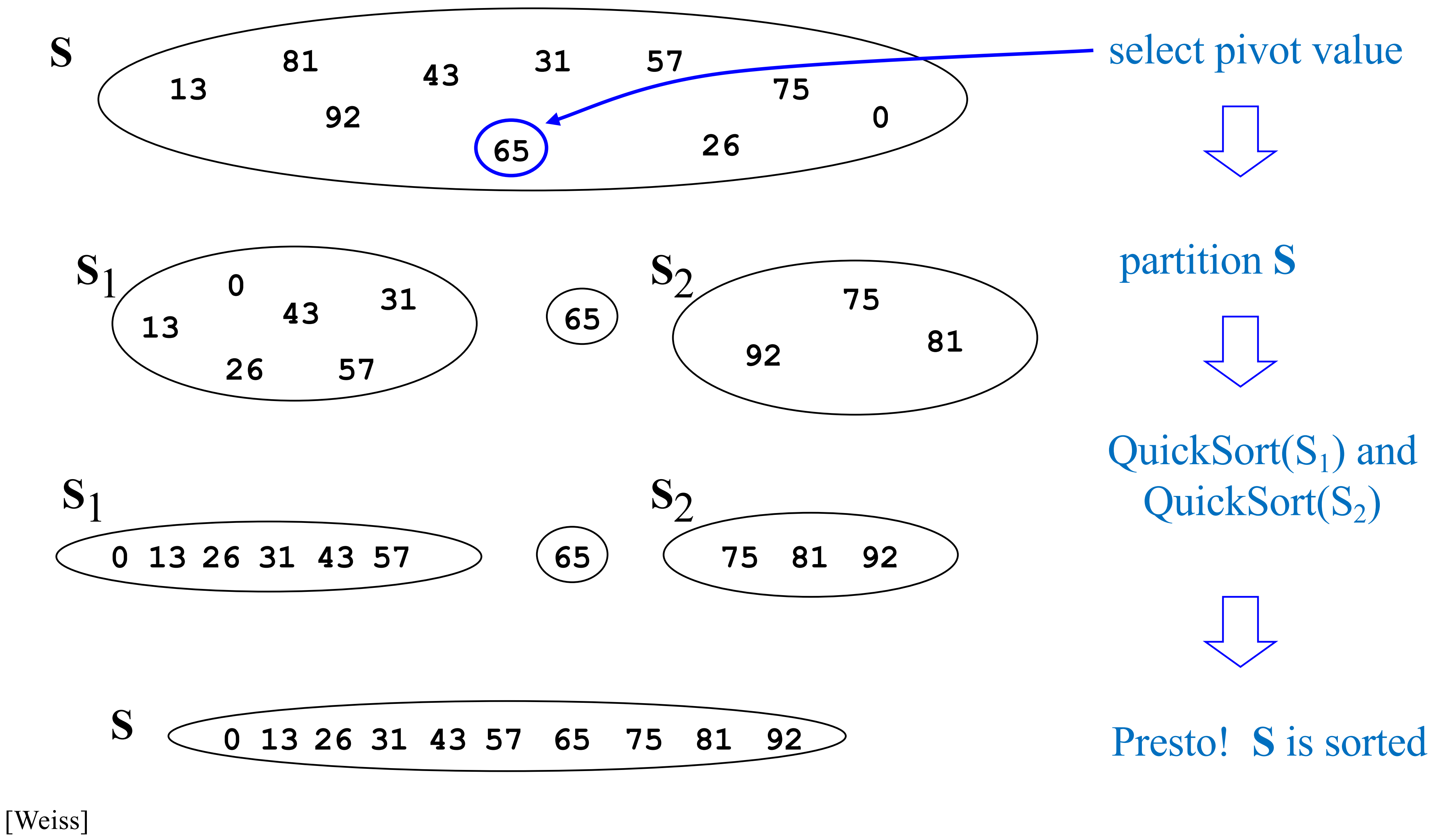
- Can be faster than mergesort
  - Often believed to be faster
  - Quicksort does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

Quicksort Overview

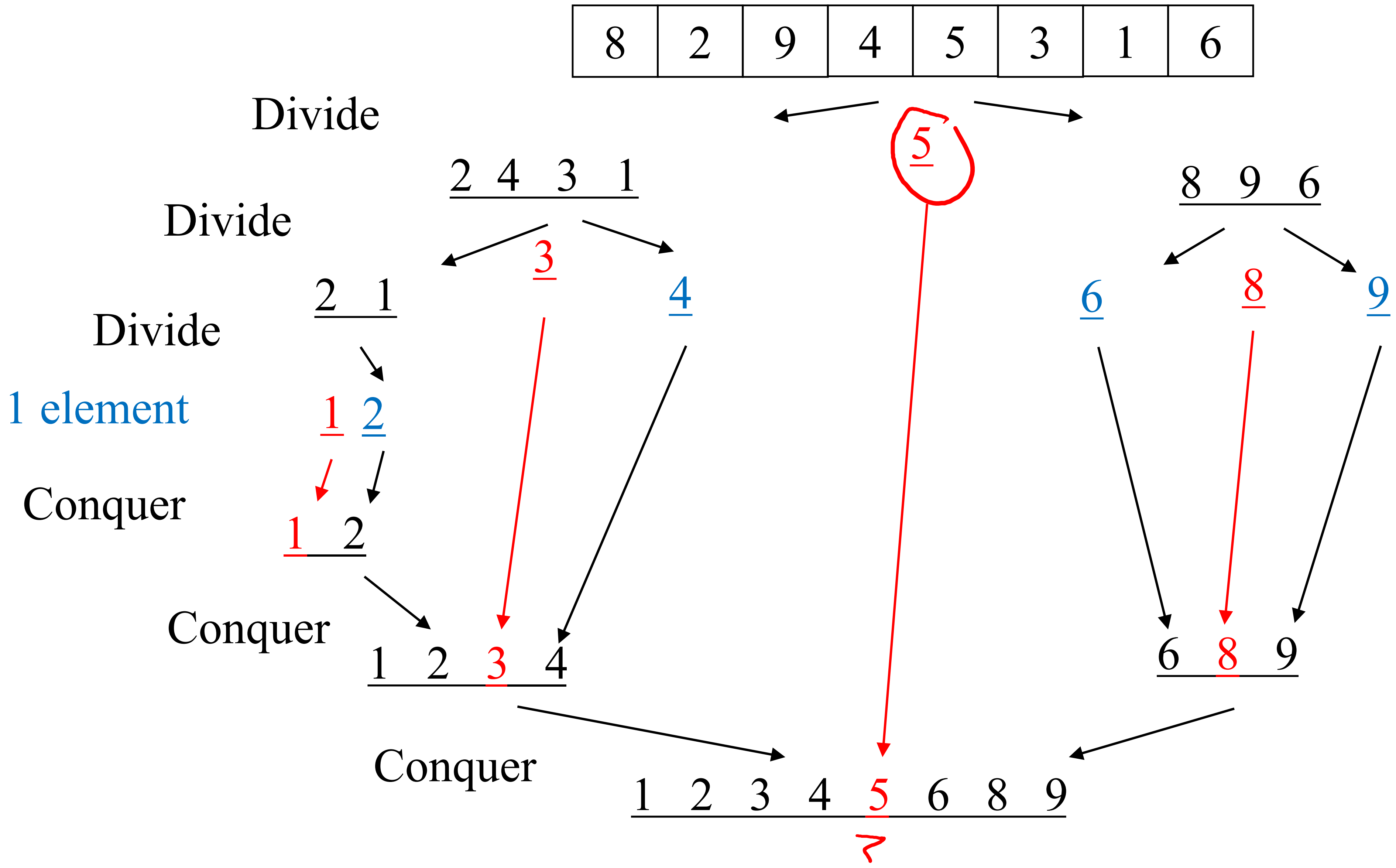
1. Pick a pivot element
   - Hopefully an element ~median
   - Good QuickSort performance depends on good choice of pivot; we'll see why later, and talk about good pivot selection later
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than 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)

Quicksort: Think in terms of sets



Quicksort Example, showing recursion



Quicksort Details

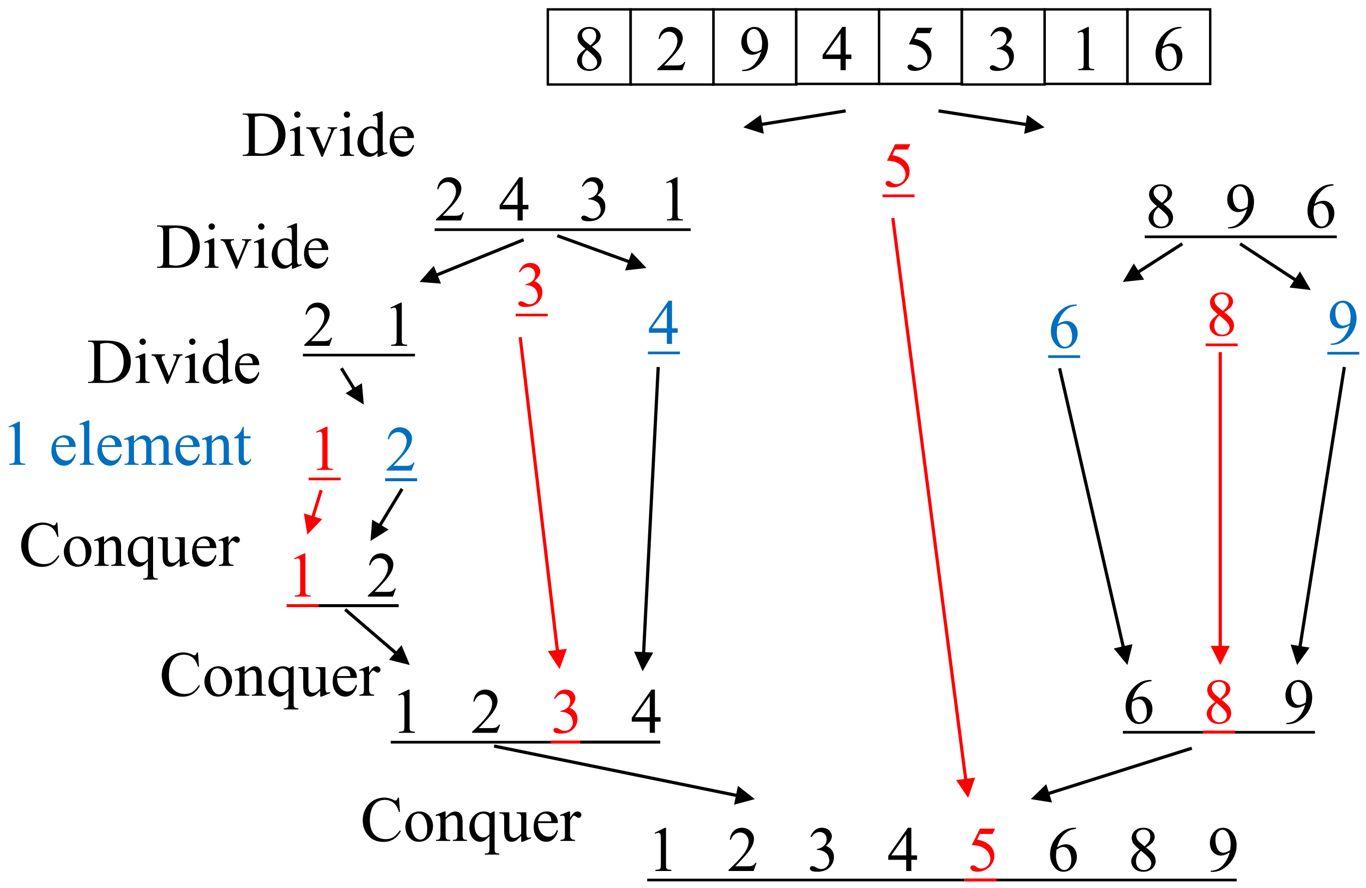
We have not yet explained:

- 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
  - Reduce to problem of size 1 smaller
  - $O(n^2)$

Quicksort: Potential pivot rules

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

- Pick `arr[lo]` or `arr[hi-1]`
  - Fast, but worst-case is (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

- That is, given 8, 4, 2, 9, 3, 5, 7 and pivot 5
  - Dividing into left half & right half (based on pivot)

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

- Ideas?

Hoare Partitioning

- One approach (there are slightly fancier ones):
  1. Swap pivot with **`arr[lo]`**; move it ‘out of the way’
  2. Use two fingers **`i`** and **`j`**, starting at **`lo+1`** and **`hi-1`** (start & end of range, apart from pivot)
  3. Move from right until we hit something less than the pivot; belongs on left side
     Move from left until we hit something greater than the pivot; belongs on right side
     Swap these two; keep moving inward

     ```
     while (i < j)
        if (arr[j] > pivot) j--
        else if (arr[i] <= pivot) i++
        else swap arr[i] with arr[j]
     ```
  4. Put pivot back in middle (Swap with **`arr[i]`**)

Quicksort 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**

Quicksort 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

Quicksort Analysis

Best-case?

Worst-case?

Average-case?

Quicksort Analysis

Best-case: Pivot is always the median

$$T(0) = T(1) = 1$$

$$T(n) = 2T(n/2) + n \quad \text{(linear-time partition)}$$

Same recurrence as mergesort: $O(n \log n)$

Worst-case: Pivot is always smallest or largest element

$$T(0) = T(1) = 1$$

$$T(n) = T(n-1) + n \quad + \cancel{T(0)}$$

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)

Is QuickSort Stable?

It depends on how we partition the elements…

Naïve Partitioning (Stable, requires more space!)

- Two passes over data
  1. Store all values in temporary array that are smaller than pivot
  2. (Add pivot, then) Store all values in temporary array that are larger than pivot

Hoare's Partitioning Scheme (NOT Stable)

- Pointer swapping (what we just saw)

Quicksort Cutoffs

n^2

- For small *n*, all that recursion tends to cost more than doing a quadratic sort
  - Remember asymptotic complexity is for large *n*
  - Also, recursive calls add a lot of overhead for small n
- Common engineering technique: switch to a different algorithm for subproblems 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

Quicksort 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

Quick Recap

	Run-time	Stable	Space
Insertion Sort	Best Case: $O(N)$ Worst Case: $O(N^2)$ Average Case: $O(N^2)$	Yes	$O(1)$
Selection Sort	$O(N^2)$	No	$O(1)$
Heap Sort	$O(N \log N)$	No	$O(1)$
Merge Sort	$O(N \log N)$	Yes	$O(N)$
Quick Sort (Hoare's Partition)	Best Case: $O(N \log N)$ Worst Case: $O(N^2)$ Average Case: $O(N \log N)$	No	$O(1)$

The Big Picture

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

Lower Bound

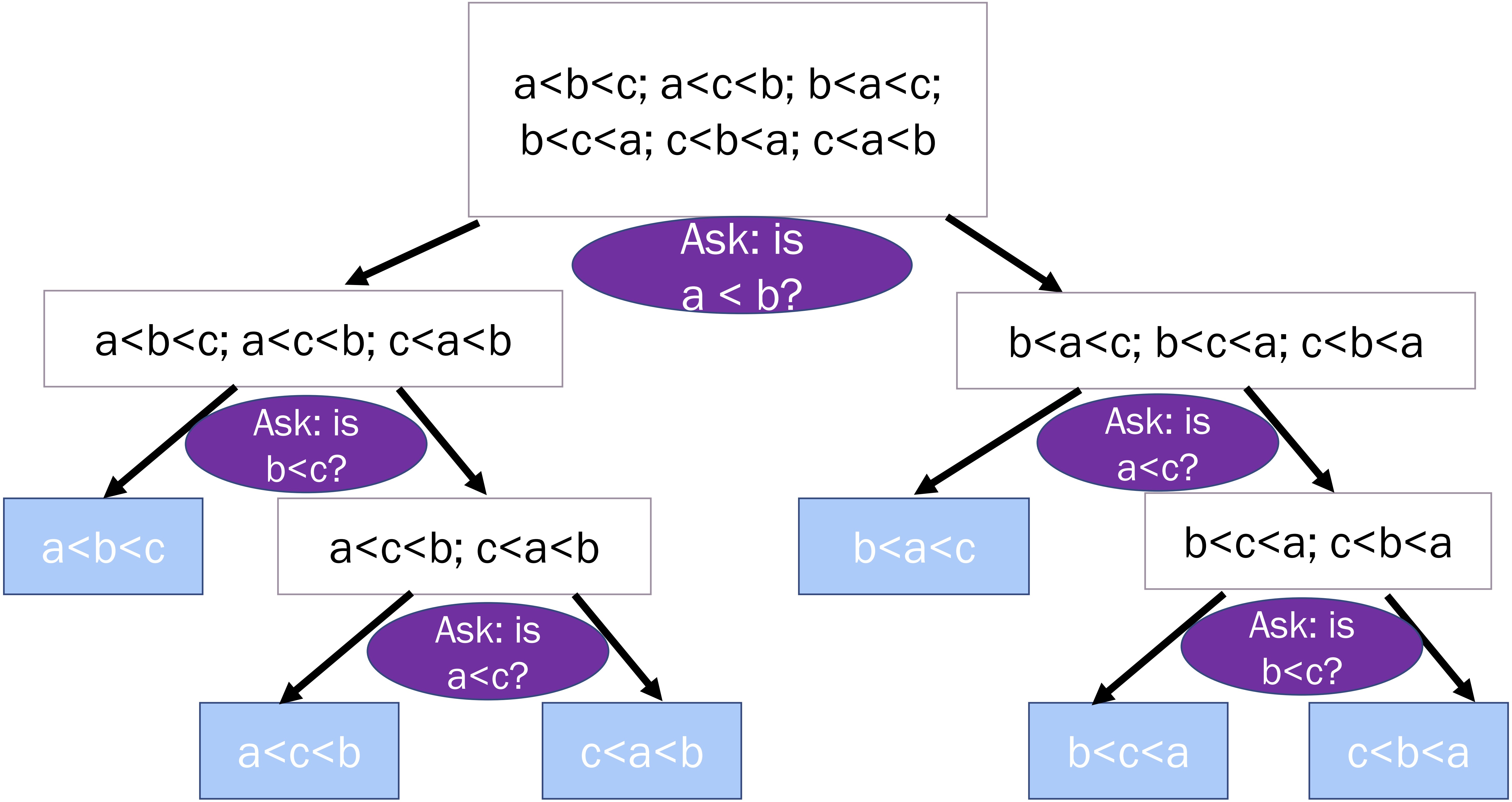
- We keep hitting $O(n \log n)$ in the worst case.
- Can we do better?
- Or is this $O(n \log n)$ pattern a fundamental barrier?
- Without more information about our data set, we cannot do better.

Comparison Sorting Lower Bound
Any sorting algorithm which only interacts with its input by comparing elements must take $\Omega(\text{n log n})$ time.

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.

$n!$


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

Poll Everywhere

pollev.com/artliu

Complete the Proof

- How many operations can we guarantee in the worst case?
  (In terms of the decision tree diagram of our arbitrary sorting algorithm)
- 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?

height of tree

- How tall is the tree if the array is length n?

$n!$

$\log_2(x)$ $\boxed{\log_2(n!)}$

- What's the simplified $\Omega(\,)$?

$$\log(n!) = \log(n \cdot n{-}1 \cdot n{-}2 \cdots) = \overbrace{\log(n) + \log(n-1) + \log(n-2) \cdots \log(1)}^{n}$$

$$\geq \log\left(\tfrac{n}{2}\right) + \log\left(\tfrac{n}{2}\right) \cdots \log\left(\tfrac{n}{2}\right) \quad \tfrac{n}{2} \text{ times}$$

$$\geq \tfrac{n}{2}\log\left(\tfrac{n}{2}\right)$$

$$\Omega(n \log n)$$

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 $\Omega()$?

$$\log_2(n!) = \log_2(n) + \log_2(n-1) + \log_2(n-2) + \cdots + \log_2(1)$$

$\geq \log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2}\right) + \cdots + \log_2\left(\frac{n}{2}\right)$ (only $n/2$ copies)

- $\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = n/2(\log_2(n) - 1) = \Omega(n \log n)$

Takeaways

$\Omega(1)$ $\Omega(n)$

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.

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.

But what if we know that all of our data points are small integers?

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

Bucket Sort (aka Bin Sort)

~~4~~	~~3~~	~~1~~	~~2~~	~~1~~	~~1~~	~~2~~	~~3~~	~~4~~	~~2~~

Assumption: We only have values 1, 2, 3, 4

1	2	3	4
III	III	II	II

1	1	1	2	2	2	3	3	4	4

Formalizing: 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.ka. 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:

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

Bucket sort illustrates a more general trick: How might you implement a heap for a small range of integer priorities in a similar manner...

- 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		→ The Bee Movie
2		
3		→ Harry Potter
4		
5		→ Casablanca → Star Wars

- Example: Movie ratings: 1=bad,… 5=excellent
- Input=

  ~~5: Casablanca~~

  ~~3: Harry Potter movies~~

  ~~1: The Bee Movie~~

  ~~5: Star Wars~~

Result: 1: The Bee Movie, 3: Harry Potter, 5: Casablanca, 5: Star Wars

This result is stable; Casablanca still before Star Wars

Radix Sort

- For each digit (starting at the ones place)
  - Run a “bucket sort” with respect to that digit
  - Keep the sort stable!

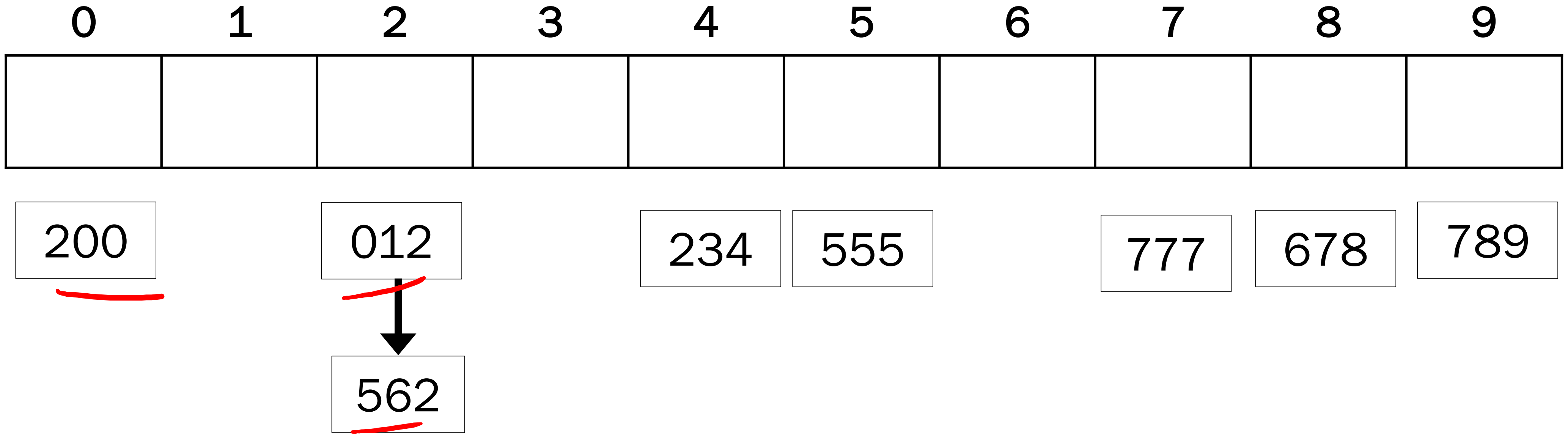
Radix Sort: Ones Place

012	234	789	555	678	200	777	562

0	1	2	3	4	5	6	7	8	9
200		012		234	555		777	678	789
		562							

Radix Sort: Ones Place

012	234	789	555	678	200	777	562



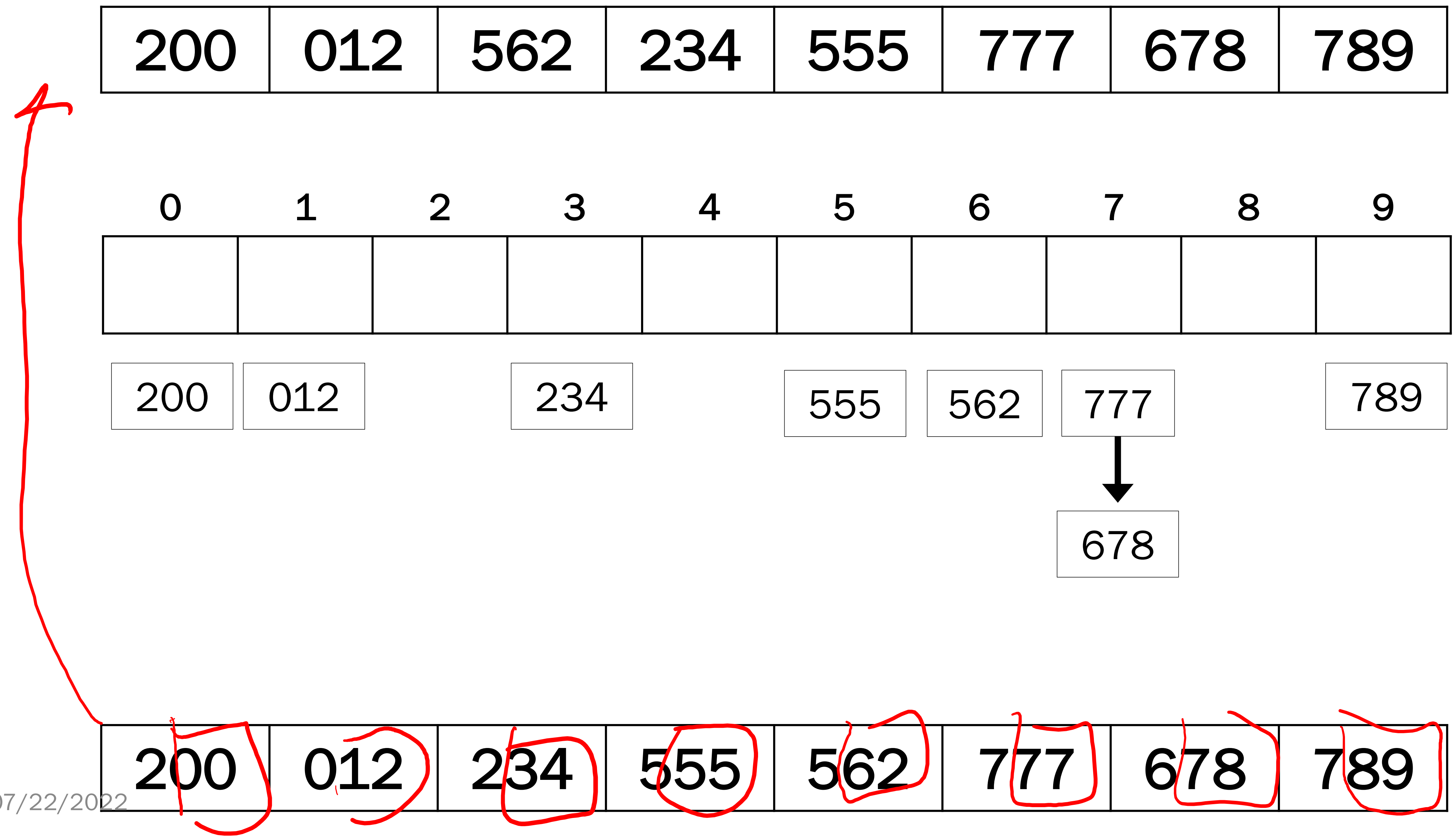
200	012	562	234	555	777	678	789

Radix Sort: Tens Place

200	012	562	234	555	777	678	789

0	1	2	3	4	5	6	7	8	9

Radix Sort: Tens Place



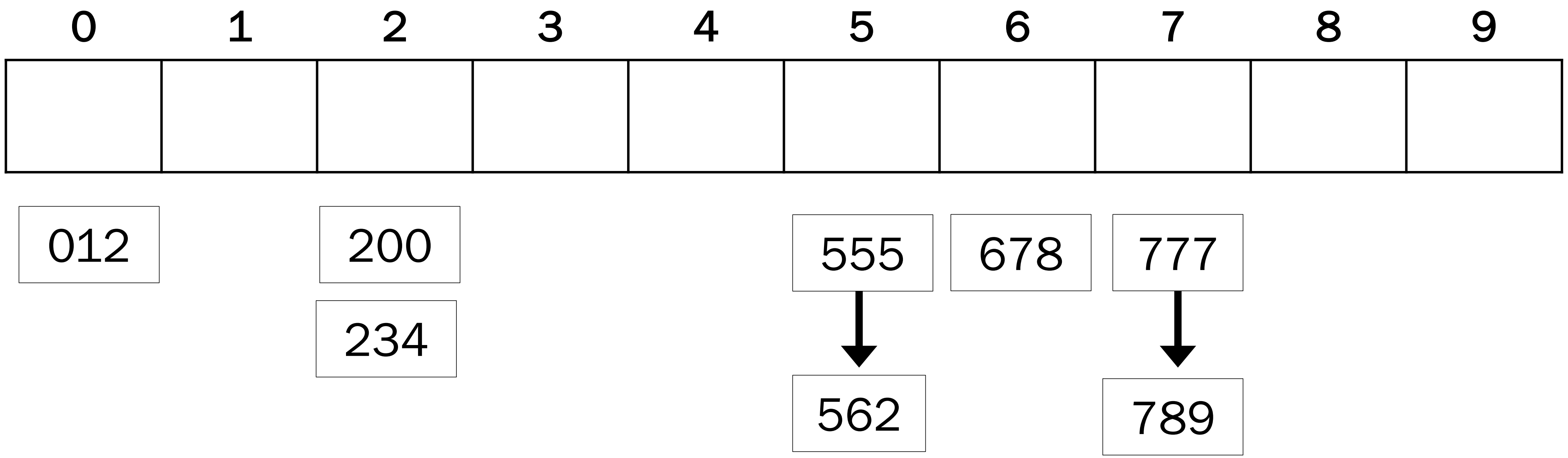
Radix Sort: Hundreds Place

200	012	234	555	562	777	678	789

0	1	2	3	4	5	6	7	8	9

Radix Sort: Hundreds Place

200	012	234	555	562	777	678	789



012	200	234	555	562	678	777	789

Radix Sort

Performance depends on:

- Input size: *n*
- Number of buckets = Radix: *B*
  - e.g. Base 10 number: 10; binary number: 2; Alpha-numeric char: 62
- Number of passes = “Digits”: *P*
  - e.g. Phone Number: 10; Person’s name: ?

$\boxed{12}\boxed{34}$

0 99

- Work per pass is 1 bucket sort: $O(B+n)$
  - Each pass is a Bucket Sort
- Total runtime is $O(P(B+n))$
  - We do ‘P’ passes, each of which is a Bucket Sort

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
- $O(n \texttt{ log } n)$ 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
- $\Omega(n \texttt{ log } n)$ 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 our Sorts! Try to do this without looking at your notes

	Run-time	Stable	Space
Insertion Sort			
Selection Sort			
Heap Sort			
Merge Sort			
Quick Sort (Hoare's Partition)			
Radix Sort			

Poll Everywhere

pollev.com/artliu

Sorting our Sorts! Try to do this without looking at your notes

	Run-time	Stable	Space
Insertion Sort	Best Case: O(N) Worst Case: $O(N^2)$ Average Case: $O(N^2)$	Yes	O(1)
Selection Sort	$O(N^2)$	No	O(1)
Heap Sort	O(N log N)	No	O(1)
Merge Sort	O(N log N)	Yes	O(N)
Quick Sort (Hoare's Partition)	Best Case: O(N log N) Worst Case: $O(N^2)$ Average Case: O(N log N)	No	O(1)
Radix Sort	O(P(B + N))	Yes	O(B + N)