LEC 21

CSE 373

Sorting II

BEFORE WE START

Which of the following is true about Insertion sort and Selection sort?

a) Both algorithms run in n^2 (quadratic) time on an already-sorted array
b) Insertion sort benefits from using a Linked List instead of an underlying array
c) Insertion sort is always faster than selection sort
d) We swap elements in the same order for both algorithms

pollev.com/uwcse373

Instructor Aaron Johnston

TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- EX4 due ***Monday* 8/17**
  - Focuses on MSTs and sorting
  - You'll need today's lecture for problem 1
- P4 due in 1 week: Wednesday 8/19
  - Don't get caught in the maze of time management in week 9!
- TA-led Industry Panel Q&A: Friday 4:30 – 5:30!
  - Come chat with a panel of your amazing TAs! Learn about their backgrounds/experiences and ask about careers in technology, finding internships, or preparing for interviews.
- Exam II next Friday! Logistics & topics list released on website
  - This Friday is the last day of content for the exam
  - Additional review materials published next Monday

Learning Objectives

After this lecture, you should be able to...

1. Implement Heap Sort, describe its runtime, and implement the in-place variant
2. Implement Merge Sort, and derive its runtimes
3. Trace through Quick Sort, derive its runtimes, and trace through the in-place variant
4. Evaluate the best algorithm to use based on properties of input data (already sorted, multiple fields, etc.)

Lecture Outline

- **Sorting**
  - *Review* **Definitions, Insertion, Selection**
  - Heap Sort
  - Merge Sort
  - Quick Sort

Review Sorting: Definitions

A sort is **stable** if the relative order of *equivalent* keys is maintained after sorting

An **in-place** sort modifies the input array directly, as opposed to building up an auxiliary data structure

Input

Anita *2010*	Basia *2018*	Caris *2019*	Duska *2020*	Duska *2015*	Anita *2016*

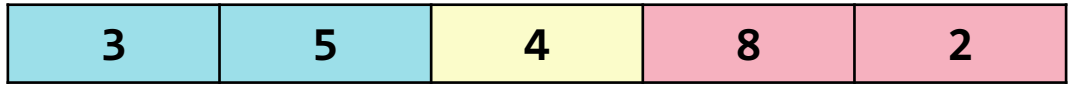
Stable sort using name as key

Anita *2010*	Anita *2016*	Basia *2018*	Caris *2019*	Duska *2020*	Duska *2015*

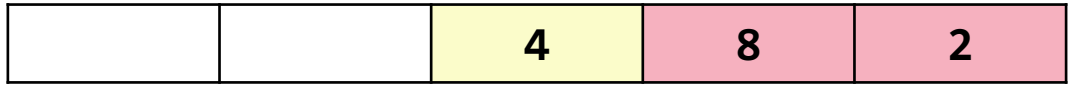
Unstable sort using name as key

Anita *2016*	Anita *2010*	Basia *2018*	Caris *2019*	Duska *2015*	Duska *2020*

In-Place sort building up result in partition of same array



Not in-place sort building up in auxiliary array



Review Sorting: Ordering Relations

- An **ordering relation** < for keys a, b, and c has the following properties:
  - Law of Trichotomy: Exactly one of a < b, a = b, b < a is true
  - Law of Transitivity: If a < b, and b < c, then a < c
- Determined by the data type *AND* the application!

Ints

2, 6, 4, 5, 8, 9

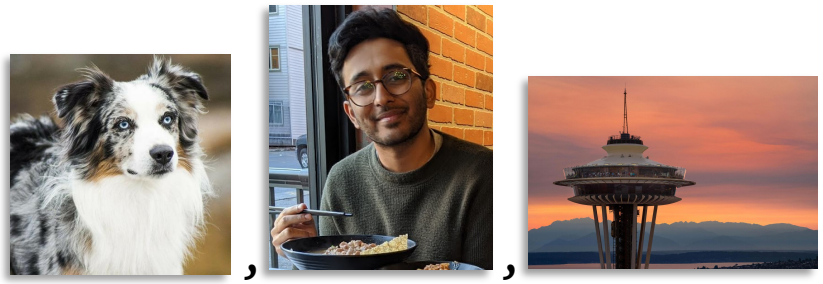
- **Increasing**: Could sort using int definition of <
- **Decreasing**: Could sort using int definition of >

Movies

Coco *2017*, Tangled *2010*, Inside Out *2015*

- **Netflix library**: Could sort by title (or star rating)
- **IMDB actor credits**: Could sort by year
- Could sort by some combo of both!

Image Data



- **File system**: Could sort by image size, last modified
- **Design**: Could sort by average color of pixels
- **Google Search Index**: Could sort by subject

Review Sorting Strategy 1: Iterative Improvement

- Invariants/Iterative improvement
  - Step-by-step make one more part of the input your desired output.

- We'll write iterative algorithms to satisfy the following invariant:
- After k iterations of the loop, the first k elements of the array will be sorted.

INVARIANT

Iterative Improvement

After k iterations of the loop, the first k elements of the array will be sorted

Review Selection vs. Insertion Sort

```
void selectionSort(list) {
   for each current in list:
      target = findNextMin(current)
      swap(target, current)
}
```

"Look through unsorted to **select** the smallest item to replace the current item"

- Then **swap** the two elements

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n^2)$
In-practice runtime? $\Theta(n^2)$
Stable? No
In-place? Yes

Minimizes writing to an array (doesn't have to shift everything)

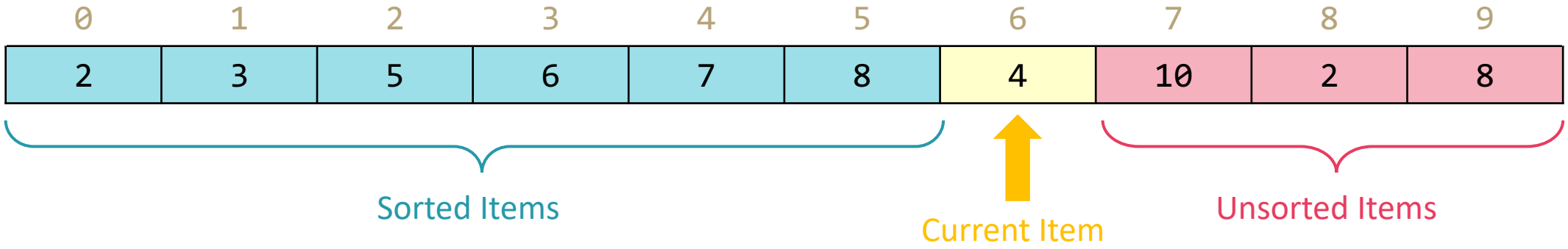
```
void insertionSort(list) {
   for each current in list:
      target = findSpot(current)
      shift(target, current)
}
```

"Look through sorted to **insert** the current item in the spot where it belongs"

- Then **shift** everything over to make space

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n)$
In-practice runtime? $\Theta(n^2)$
Stable? Yes
In-place? Yes

Almost always preferred: Stable & can take advantage of an already-sorted list.
(LinkedList means no shifting ☺, though doesn't change asymptotic runtime)



Lecture Outline

- **Sorting**
  - *Review* Definitions, Insertion, Selection
  - **Heap Sort**
  - Merge Sort
  - Quick Sort

Sorting Strategy 2: Impose Structure

- Consider what contributes to Selection sort runtime of $\Theta(n^2)$
  - Unavoidable n iterations to consider each element
  - Finding next minimum element to swap requires a $\Theta(n)$ linear scan! Could we do better?

$\Theta(n)$ iterations →
$\Theta(n)$ →

```
void selectionSort(list) {
   for each current in list:
      target = findNextMin(current)
      swap(target, current)
}
int findNextMin(current) {
   min = current
   for each item in unsorted items:
      if (item < min):
         min = current
   return min
}
```

- If only we knew a way to *structure* our *data* to make it fast to find the smallest item remaining in our dataset...

MIN PRIORITY QUEUE ADT

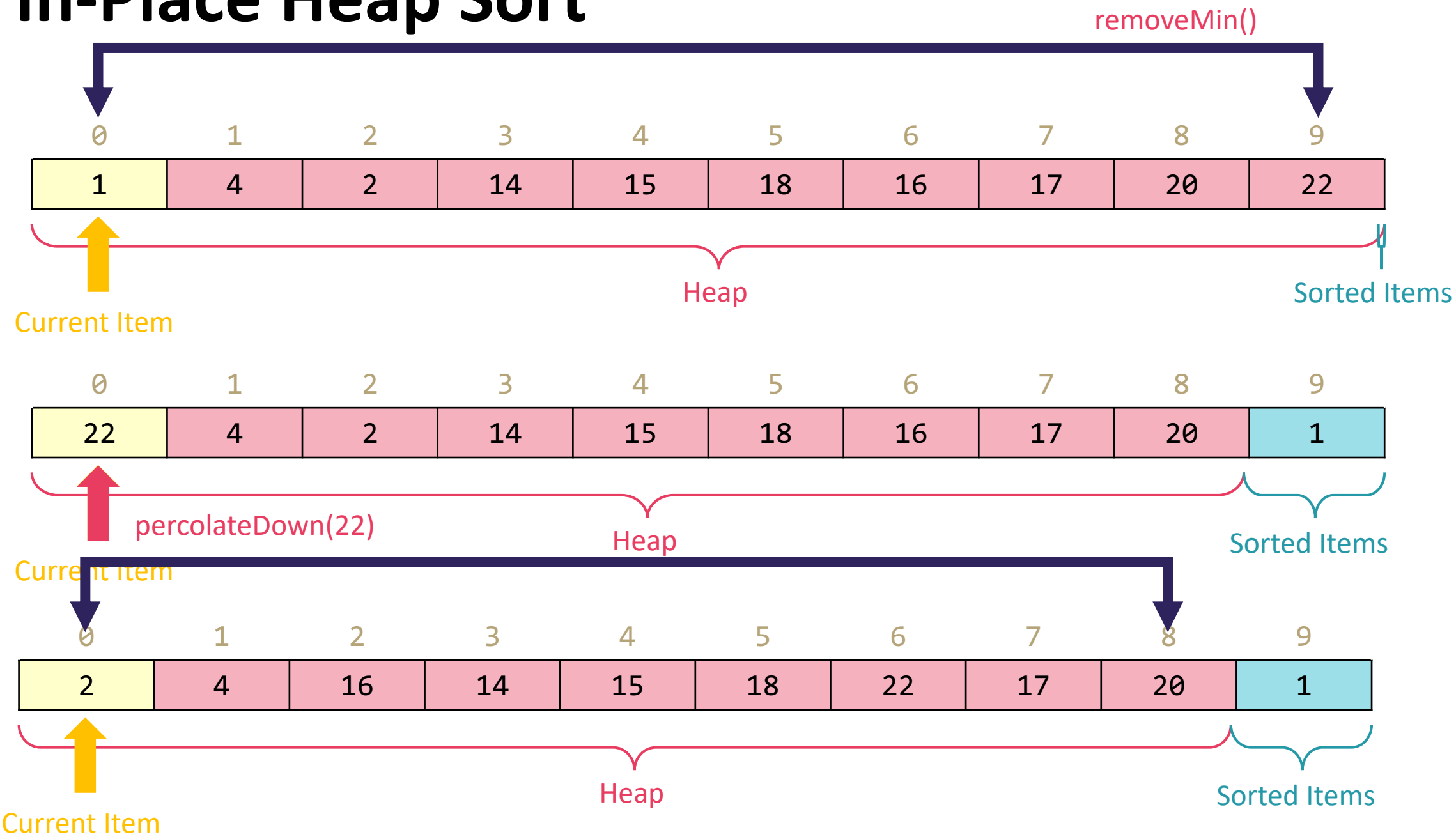
Heap Sort

1. Run Floyd's `buildHeap` on your data
2. Call `removeMin` n times to pull out every element

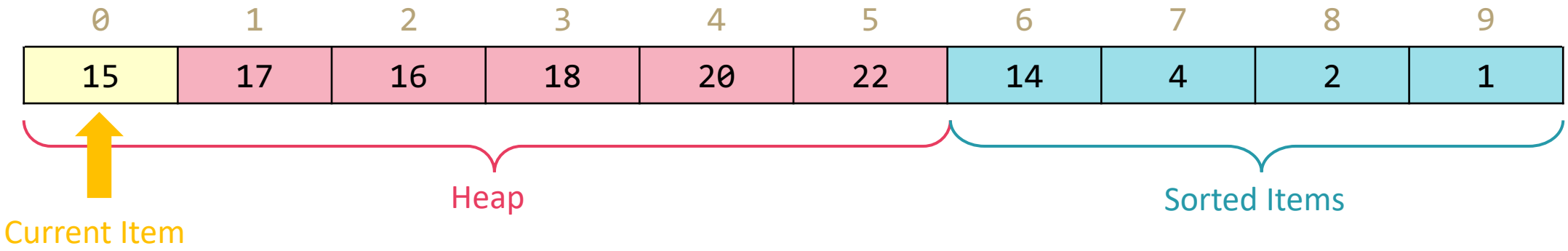
```
void heapSort(list) {
   E[] heap = buildHeap(list)
   E[] output = new E[n]
   for (i = 0; i < n; i++):
      output[i] = removeMin(heap)
}
```

Worst case runtime?	$\Theta(n \log n)$
Best case runtime?	$\Theta(n)$
In-practice runtime?	$\Theta(n \log n)$
Stable?	No
In-place?	If we get clever…

In-Place Heap Sort



In Place Heap Sort




```
void inPlaceHeapSort(list) {
   buildHeap(list) // alters original array
   for (n : list)
      list[n – i - 1] = removeMin(heap part of list)
}
```

Complication: final array is reversed! Lots of fixes:
- Run reverse afterwards ($O(n)$)
- Use a max heap
- Reverse compare function to emulate max heap

Worst case runtime?	$\Theta(n \log n)$
Best case runtime?	$\Theta(n)$
In-practice runtime?	$\Theta(n \log n)$
Stable?	No
In-place?	Yes

Lecture Outline

- **Sorting**
  - *Review* Definitions, Insertion, Selection
  - Heap Sort
  - **Merge Sort**
  - Quick Sort

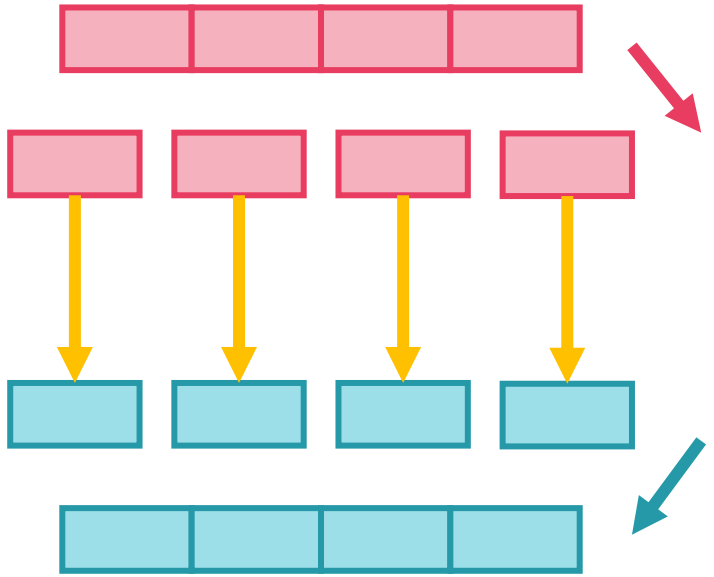
Sorting Strategy 3: Divide and Conquer

General recipe:

1. **Divide** your work into smaller pieces recursively

2. **Conquer** the recursive subproblems
   - In many algorithms, conquering a subproblem requires no extra work beyond recursively dividing and combining it!

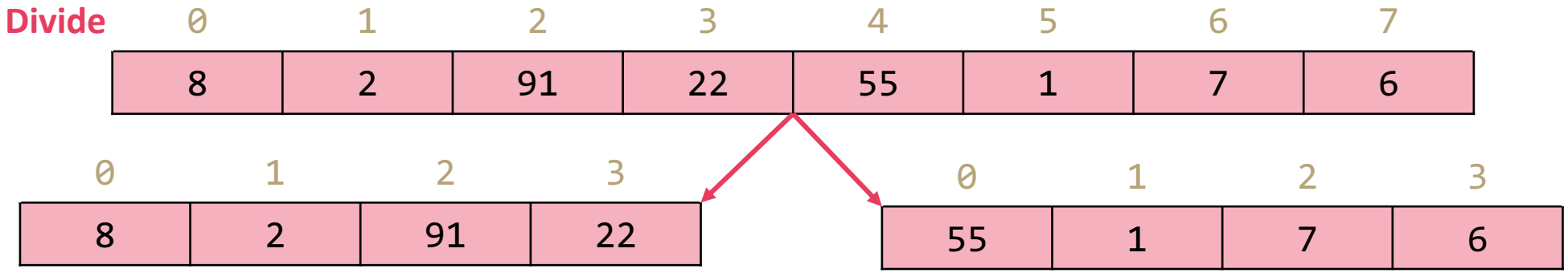
3. **Combine** the results of your recursive calls



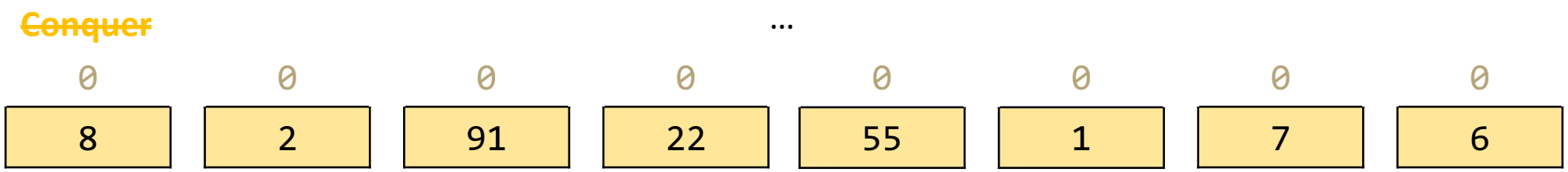
```
divideAndConquer(input) {
  if (small enough to solve):
    conquer, solve, return results
  else:
    divide input into a smaller pieces
    recurse on smaller pieces
    combine results and return
}
```

Merge Sort

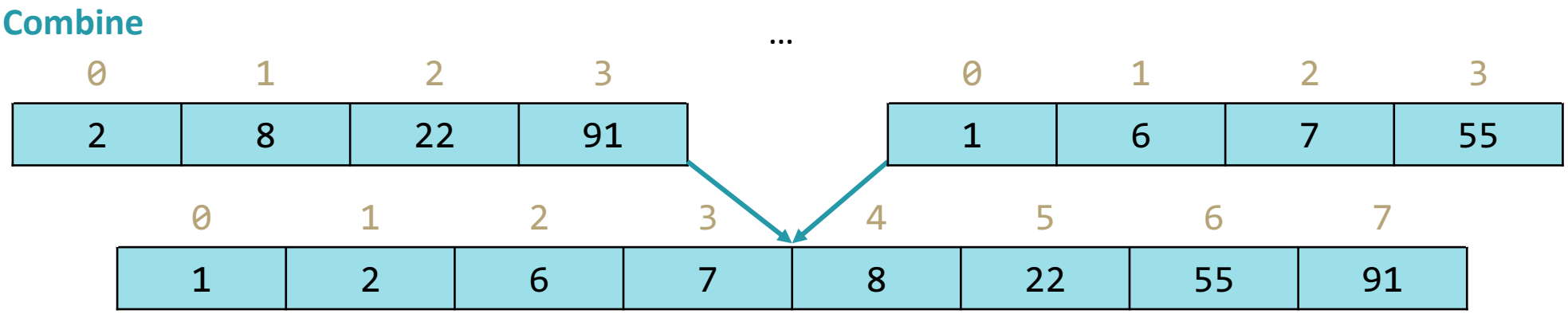
Simply divide in half each time



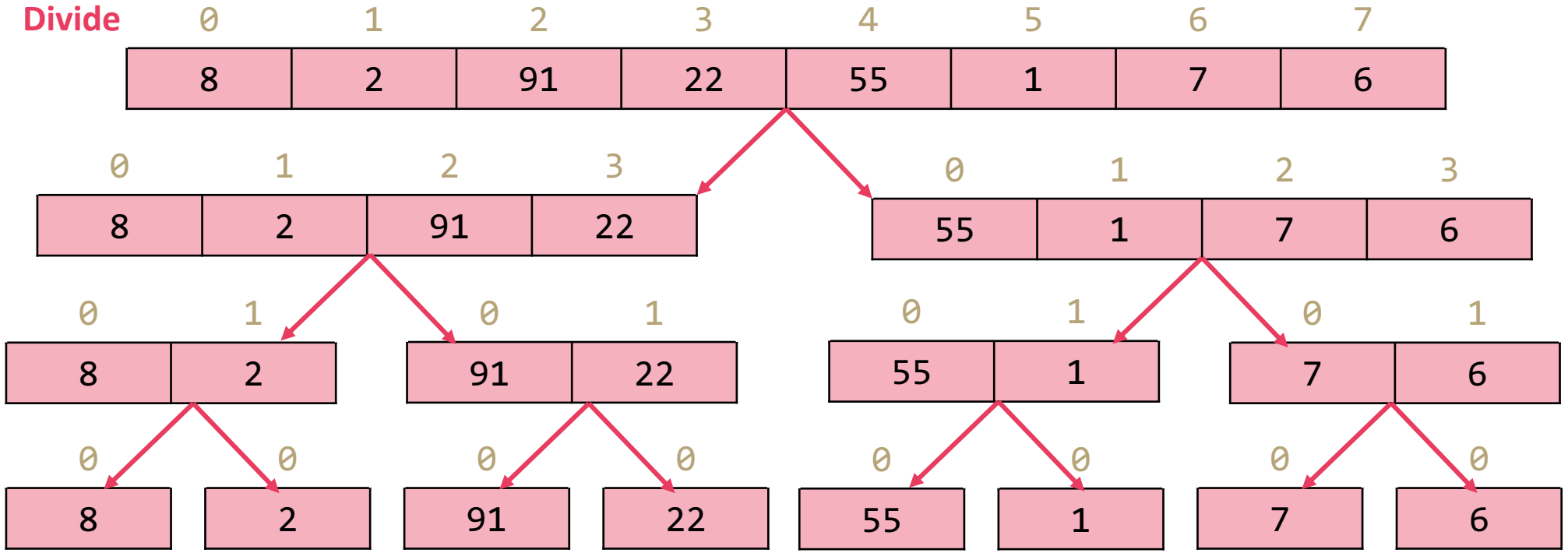
No extra conquer work needed!



The actual sorting happens here!



Merge Sort: Divide Step

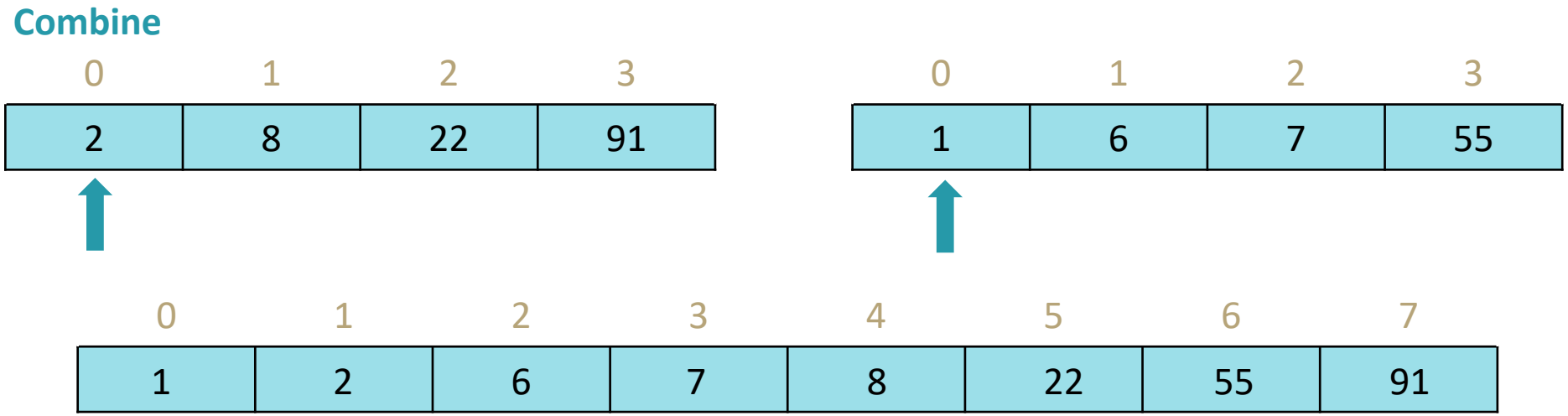


Recursive Case: split the array in half and recurse on both halves

Base Case: when array hits size 1, stop dividing. In Merge Sort, no additional work to conquer: everything gets sorted in combine step!

Sort the pieces through the magic of recursion

Merge Sort: Combine Step



Combining two *sorted* arrays:

1. Initialize **pointers** to start of both arrays
2. Repeat until all elements are added:
    1. Add whichever is smaller to the result array
    2. Move that pointer forward one spot

Works because we only move the smaller pointer – then ”reconsider” the larger against a new value, and because the arrays are sorted we never have to backtrack!

Merge Sort

```
mergeSort(list) {
   if (list.length == 1):
      return list
   else:
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```

Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$

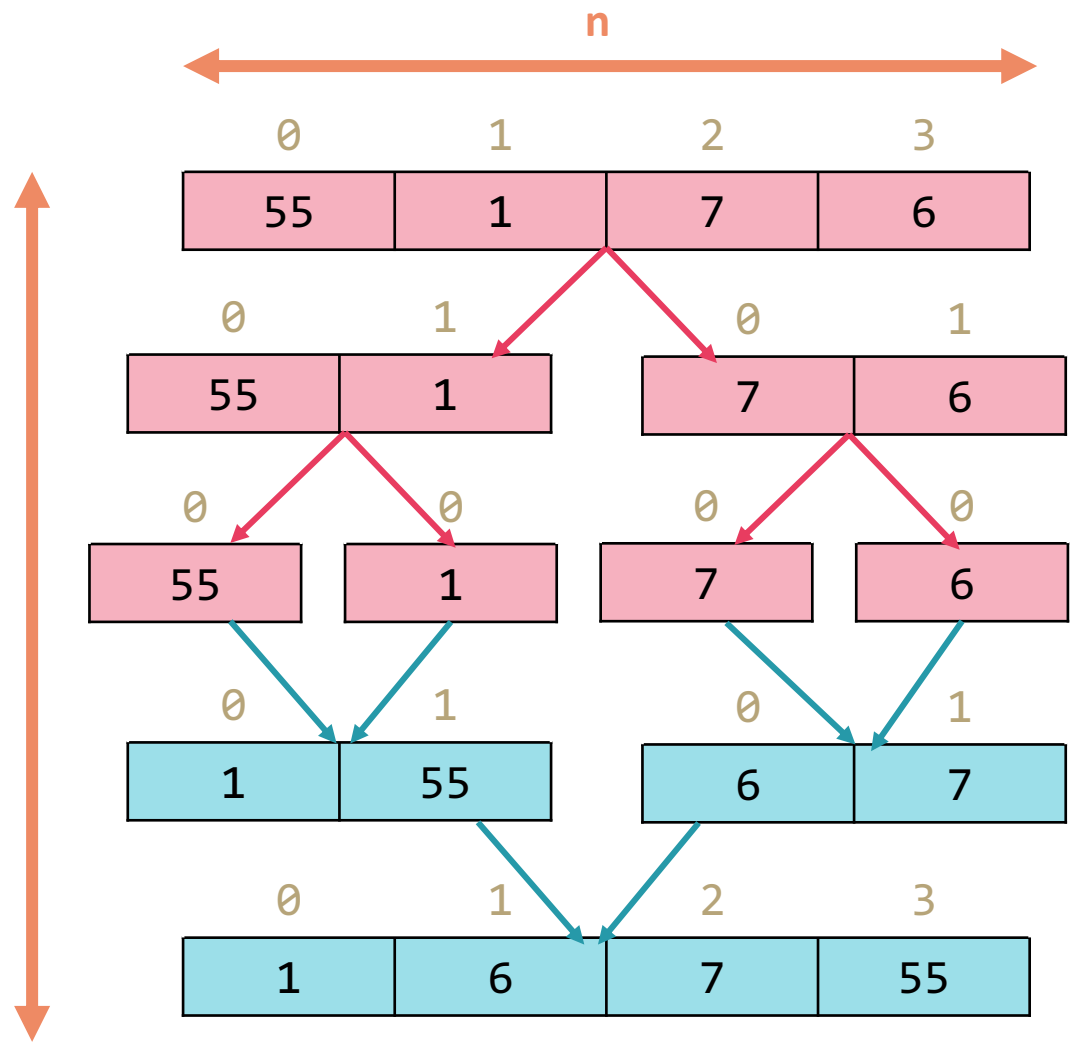
$= \Theta(n \log n)$

Best case runtime? Same

In Practice runtime? Same

Stable? Yes

In-place? No



2 **Constant size Input**

Don't forget your old friends, the 3 recursive patterns!

CORRECTED AFTER LECTURE

Lecture Outline

- **Sorting**
  - *Review* Definitions, Insertion, Selection
  - Heap Sort
  - Merge Sort
  - **Quick Sort** 

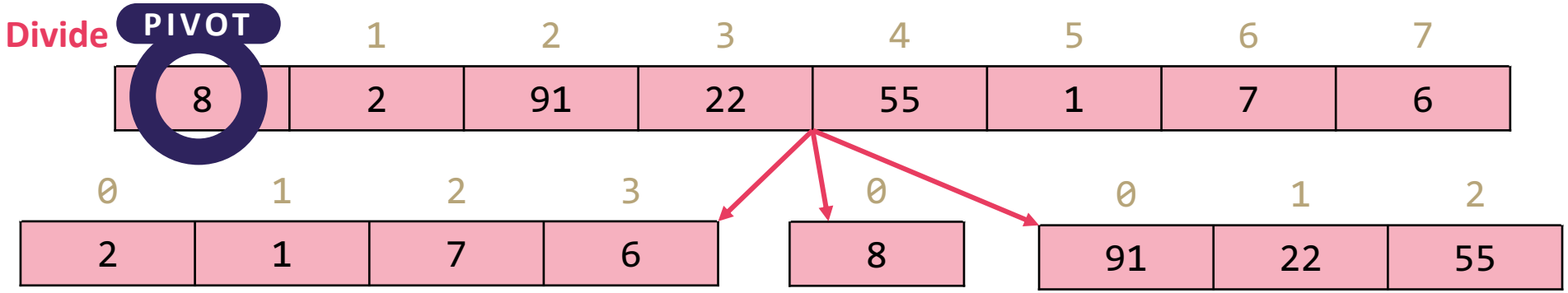
Divide and Conquer

- There's more than one way to divide!
- Mergesort:
  - Split into two arrays.
  - Elements that just happened to be on the left and that happened to be on the right.

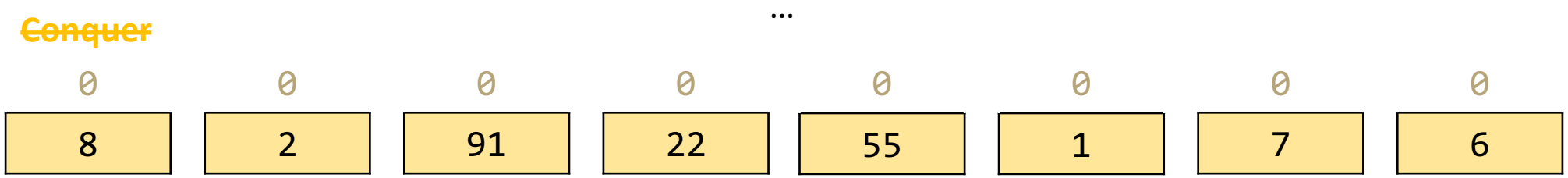
- Quicksort:
  - Split into two arrays.
  - Roughly, elements that are "small" and elements that are "large"
  - How to define "small" and "large"? Choose a "**pivot**" value in the array that will **partition** the two arrays!

Quick Sort (v1)

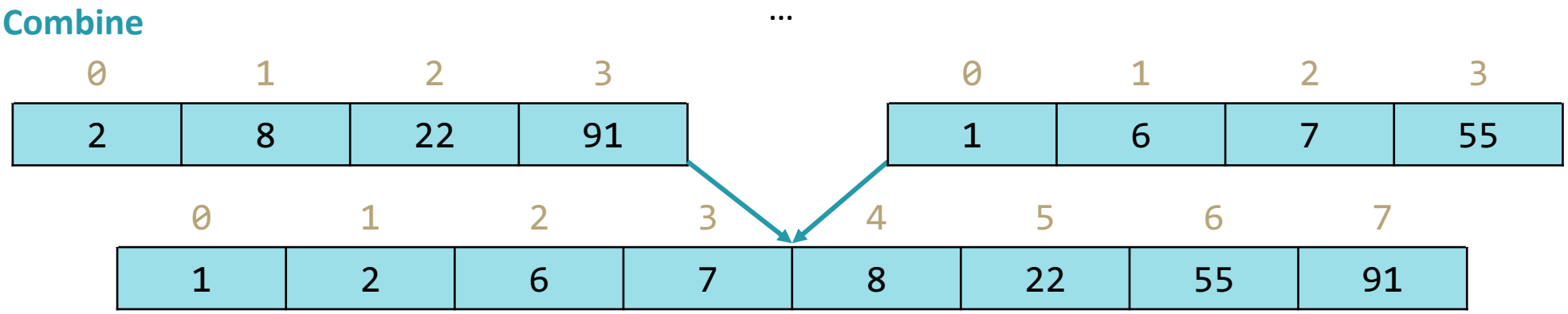
Choose a "pivot" element, partition array relative to it!



Again, no extra conquer step needed!



Simply concatenate the now-sorted arrays!



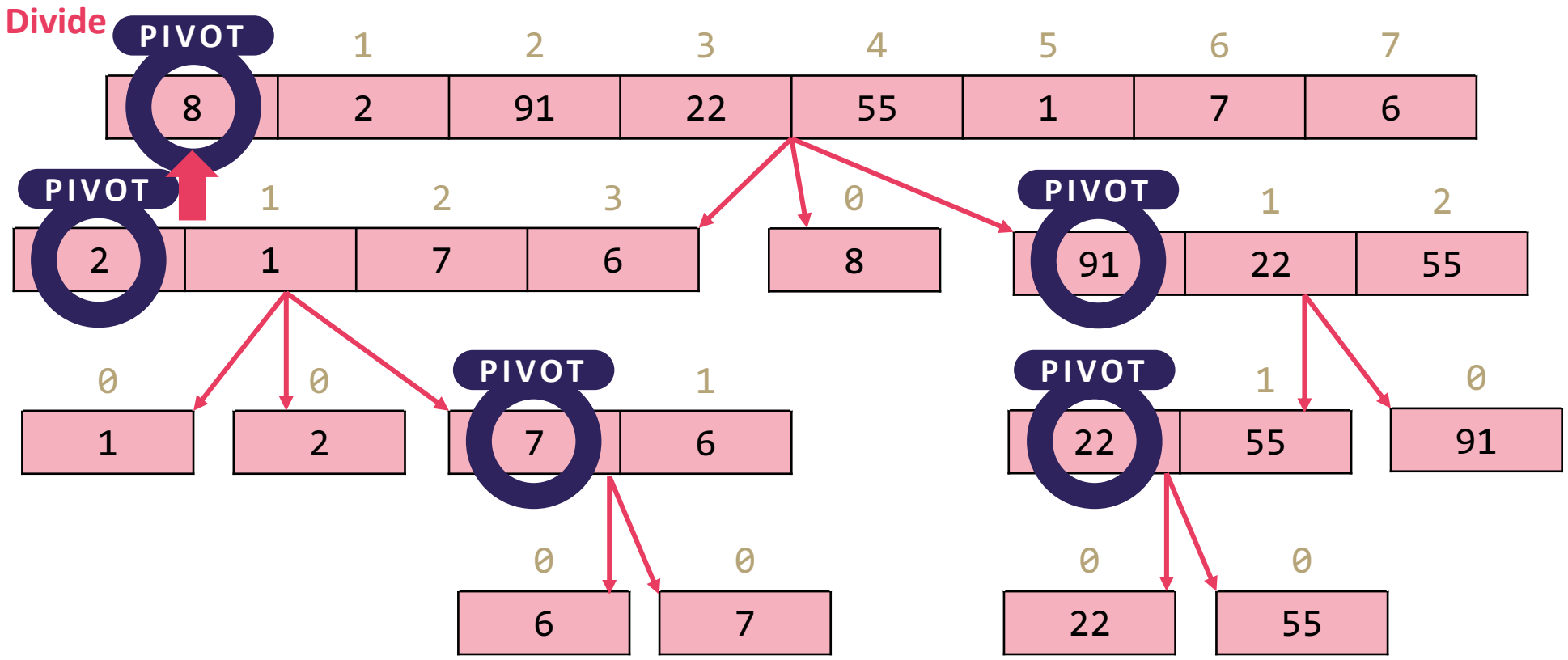
Quick Sort (v1): Divide Step

Recursive Case:

- Choose a "pivot" element
- Partition: linear scan through array, add smaller elements to one array and larger elements to another
- Recursively partition

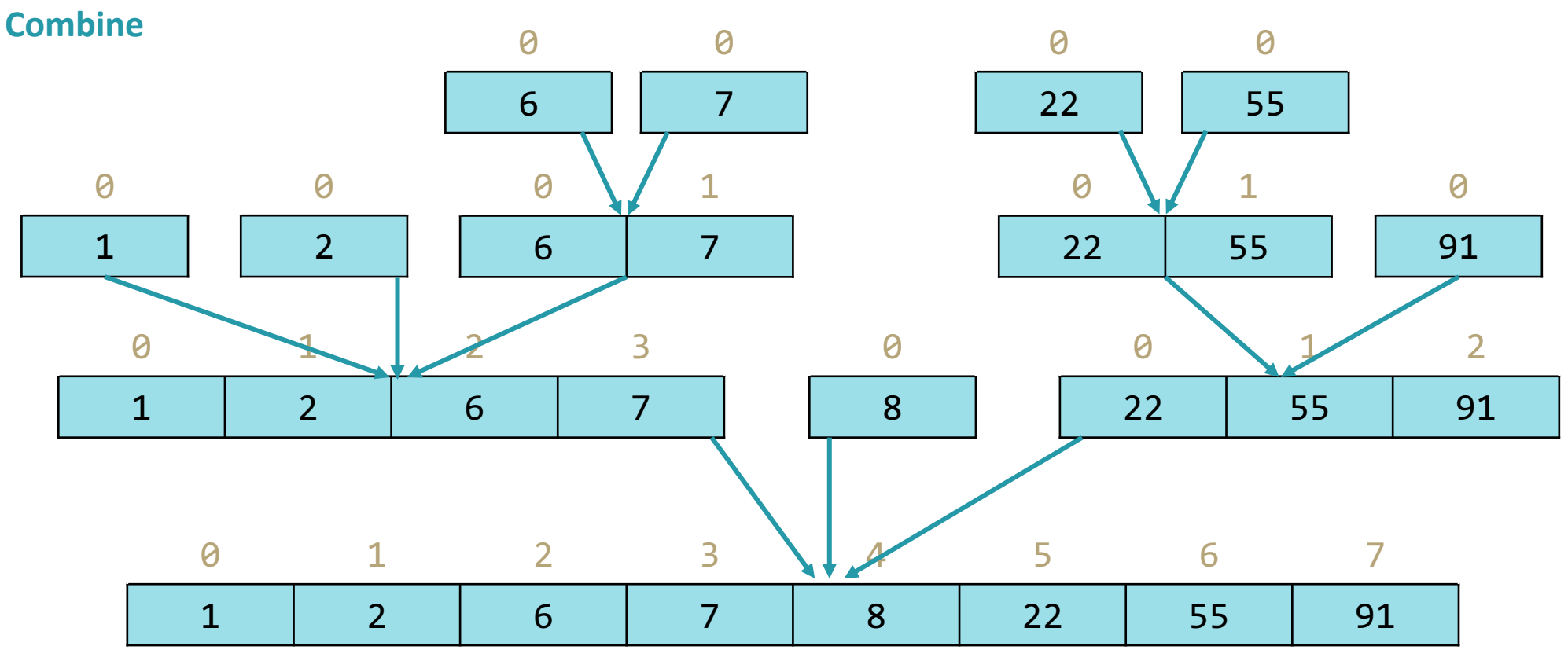
Base Case:

- When array hits size 1, stop dividing.



Quick Sort (v1): Combine Step

Simply concatenate the arrays that were created earlier! Partition step already left them in order ☺



Quick Sort (v1)

```
quickSort(list) {
   if (list.length == 1):
      return list
   else:
      pivot = choosePivot(list)
      smallerHalf = quickSort(getSmaller(pivot, list))
      largerHalf = quickSort(getBigger(pivot, list))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + n & \text{otherwise} \end{cases} = \Theta(n^2)$

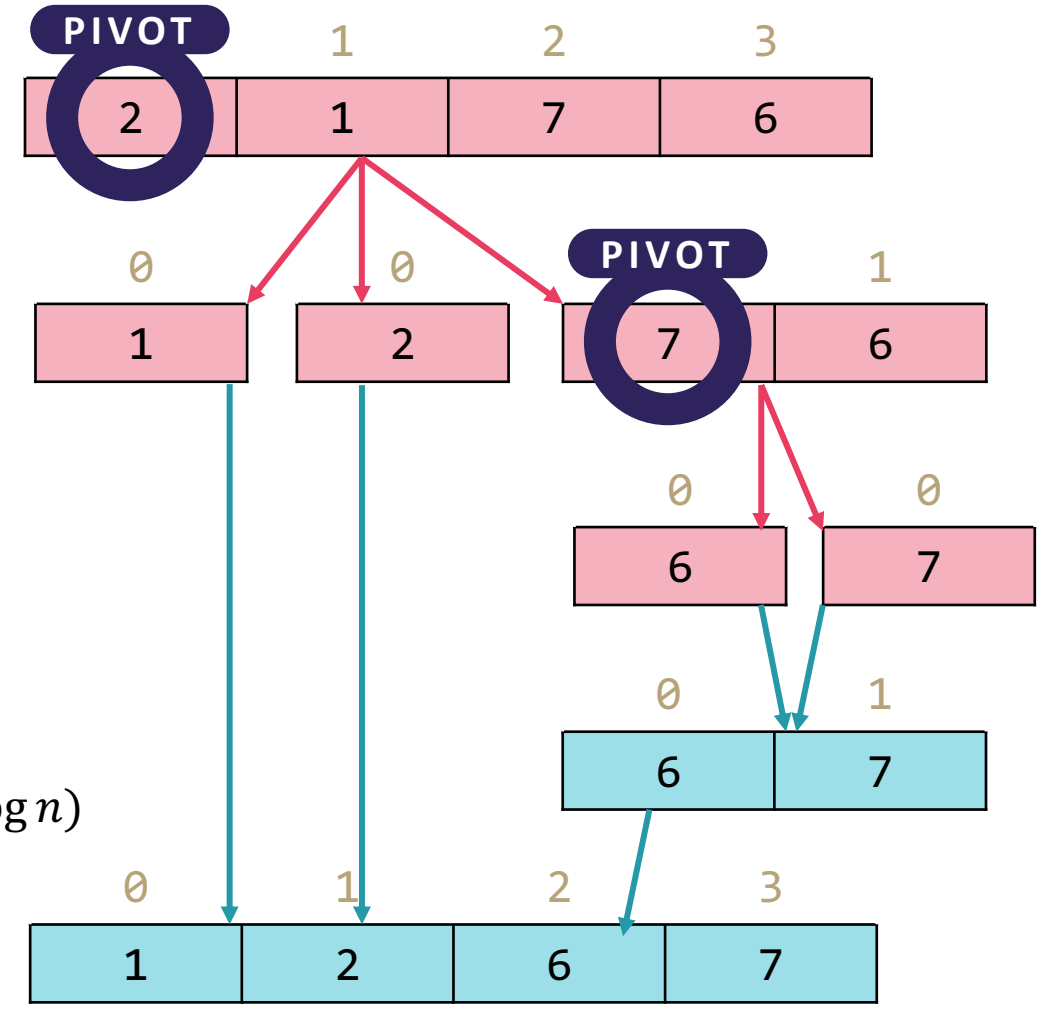
Best case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases} = \Theta(n \log n)$

In-practice runtime? Just trust me: $\Theta(n \log n)$
(absurd amount of math to get here)

Stable? No

In-place? Can be done!

Worst case: Pivot only chops off one value
Best case: Pivot divides each array in half



Can we do better?

- How to avoid hitting the worst case?
  - It all comes down to the pivot. If the pivot divides each array in half, we get better behavior

- Here are four options for finding a pivot. What are the tradeoffs?
  - Just take the first element
  - Take the median of the first, last, and middle element
  - Take the median of the full array
  - Pick a random element

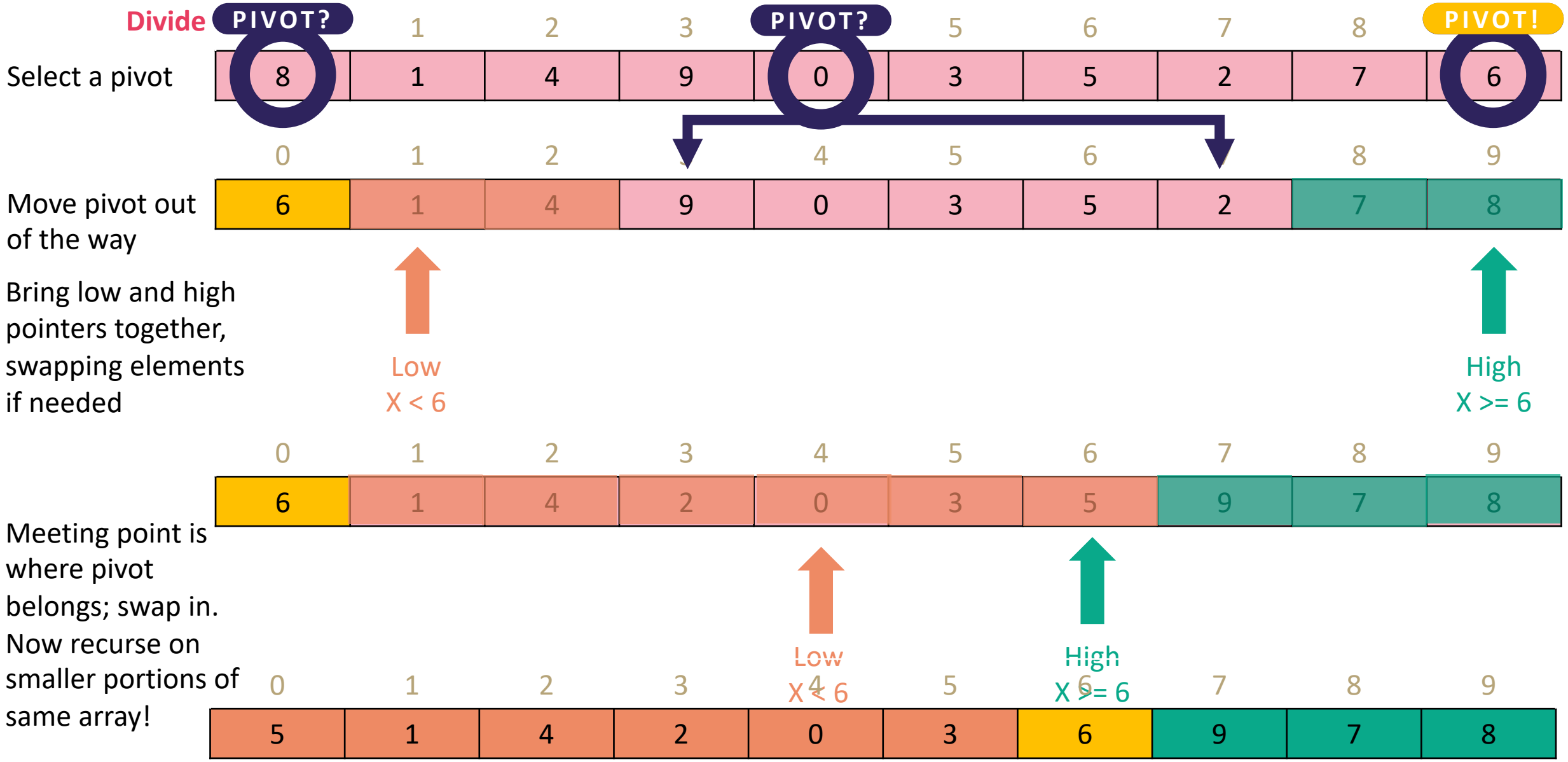
Strategies for Choosing a Pivot

- Just take the first element
  - Very fast!
  - But has worst case: for example, sorted lists have $\Omega(n^2)$ behavior
- Take the median of the first, last, and middle element
  - Makes pivot slightly more content-aware, at least won't select very smallest/largest
  - Worst case is still $\Omega(n^2)$, but on real-world data tends to perform well!

Most commonly used

- Take the median of the full array
  - Can actually find the median in $O(n)$ time (google QuickSelect). It's **complicated.**
  - $O(n\ log\ n)$ even in the worst case… but the constant factors are **awful**. No one does quicksort this way.
- Pick a random element
  - Get $O(n \log n)$ runtime with probability at least $1 - 1/n^2$
  - No simple worst-case input (e.g. sorted, reverse sorted)

Quick Sort (v2: In-Place)



Quick Sort (v2: In-Place)

```
quickSort(list) {
   if (list.length == 1):
      return list
   else:
      pivot = choosePivot(list)
      smallerHalf = quickSort(getSmaller(pivot, list))
      largerHalf = quickSort(getBigger(pivot, list))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + n & \text{otherwise} \end{cases} = \Theta(n^2)$

Best case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases} = \Theta(n \log n)$

In-practice runtime? Just trust me: $\Theta(n \log n)$
(absurd amount of math to get here)

Stable? No

In-place? Yes

Sorting: Summary

	Best-Case	Worst-Case	Space	Stable
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	No
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(1)$	Yes
Heap Sort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n)$	No
In-Place Heap Sort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(1)$	No
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$ $\Theta(n)$* optimized	Yes
Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	No
In-place Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(1)$	No

What does Java do?

- Actually uses a combination of *3 different sorts*:
  - If objects: use Merge Sort (stable!)
  - If primitives: use Dual Pivot Quick Sort
  - If "reasonably short" array of primitives: use Insertion Sort
    - Researchers say 48 elements

Key Takeaway: No single sorting algorithm is "the best"!

- Different sorts have different properties in different situations
- The "best sort" is one that is well-suited to your data

But Don't Take it From Me…

Here are some excellent visualizations for the sorting algorithms we've talked about!

Comparing Sorting Algorithms

- Different Types of Input Data: https://www.toptal.com/developers/sorting-algorithms
- More Thorough Walkthrough: https://visualgo.net/en/sorting?slide=1

Comparing Sorting Algorithms

- Insertion Sort: https://www.youtube.com/watch?v=ROalU379l3U
- Selection Sort: https://www.youtube.com/watch?v=Ns4TPTC8whw
- Heap Sort: https://www.youtube.com/watch?v=Xw2D9aJRBY4
- Merge Sort: https://www.youtube.com/watch?v=XaqR3G_NVoo
- Quick Sort: https://www.youtube.com/watch?v=ywWBy6J5gz8