LEC 24

CSE 373

Sorting II

BEFORE WE START

Instructor Hunter Schafer

TAs
Ken Aragon
Khushi Chaudhari
Joyce Elauria
Santino Iannone
Leona Kazi
Nathan Lipiarski
Sam Long
Amanda Park
Paul Pham
Mitchell Szeto
Batina Shikhalieva
Ryan Siu
Elena Spasova
Alex Teng
Blarry Wang
Aileen Zeng

Learning Objectives

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

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

Lecture Outline

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

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: Definitions

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

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*

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

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

3	5	4	8	2

Not in-place sort building up in auxiliary array

		4	8	2

3	5			

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)

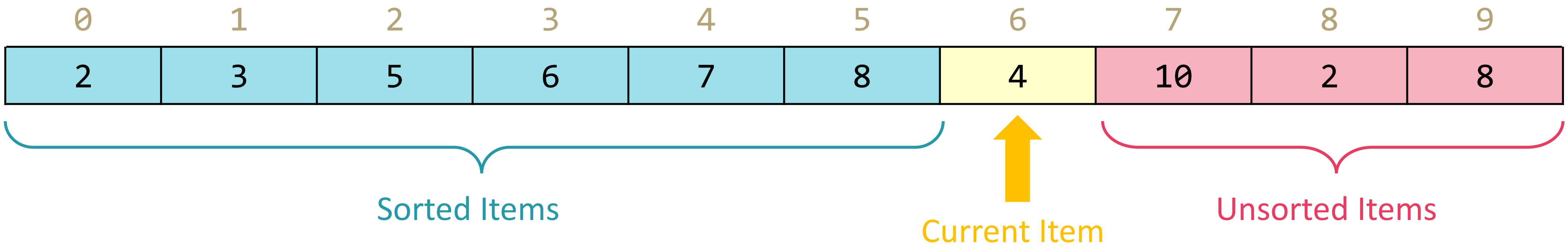
```
void insertionSort(list) {
   for each current in list:
      target = findSpot(current)
      shift(target, current)
}
```

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

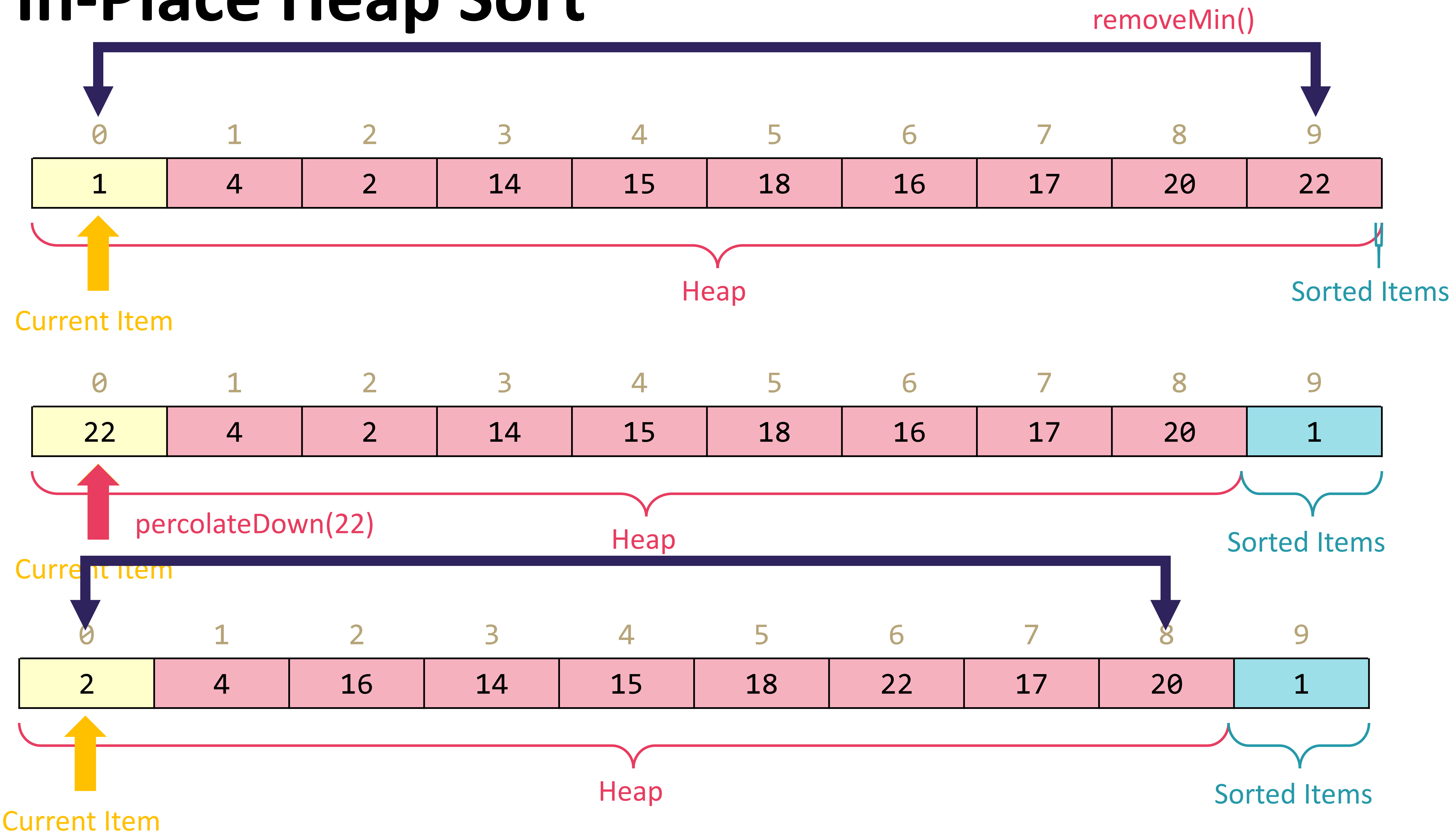
- Then **shift** everything over to make space

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)



In-Place Heap Sort



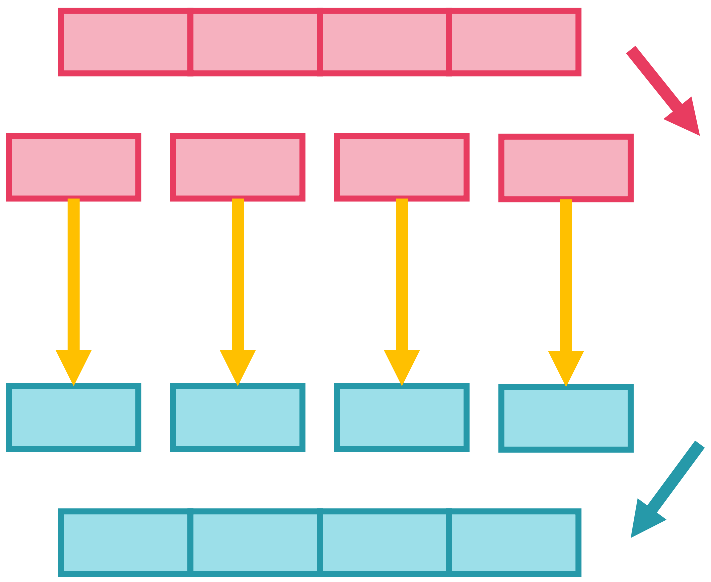
Lecture Outline

- *Review* Definitions, Insertion, Selection
- **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



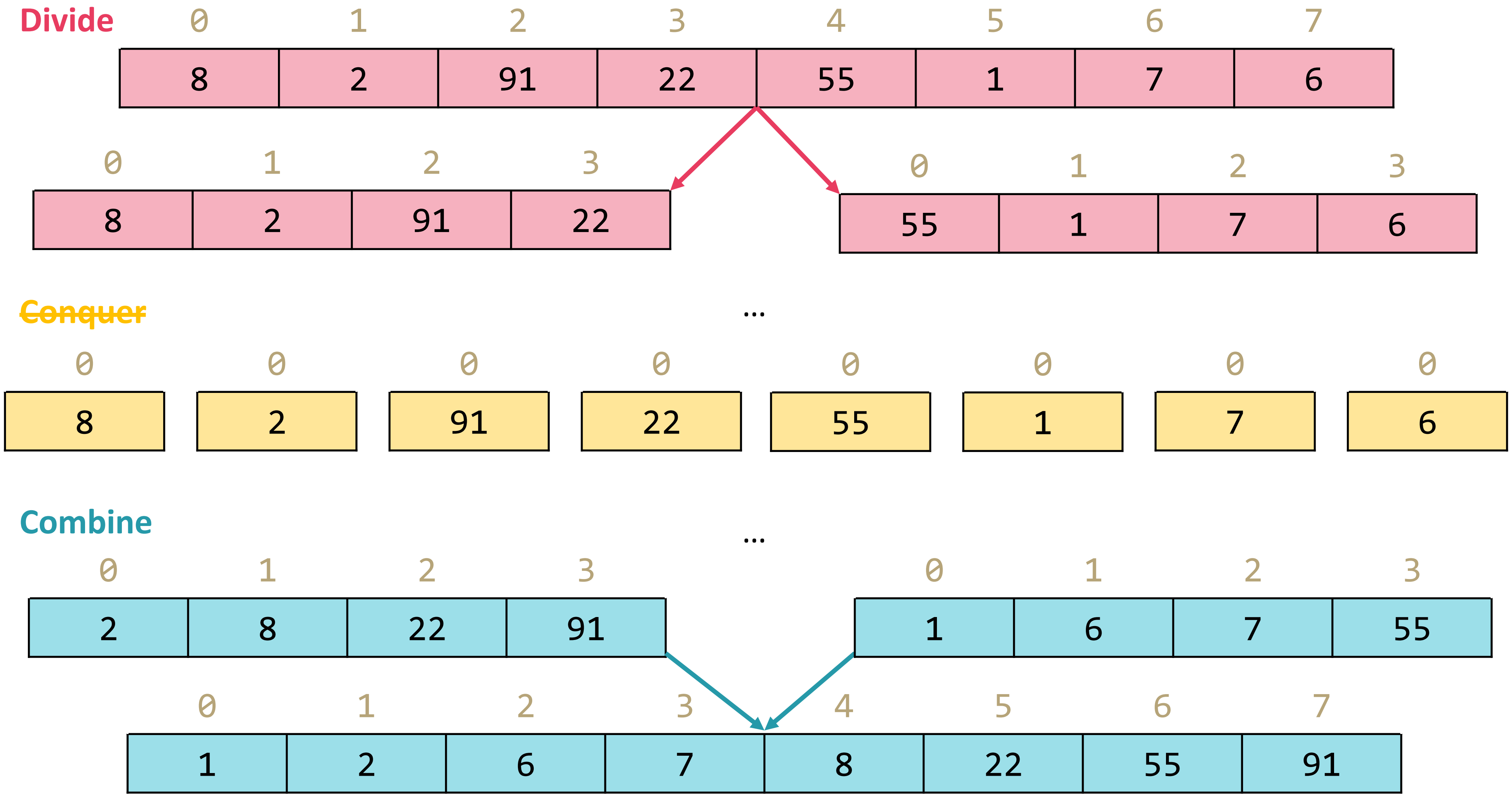
```
divideAndConquer(input) {
  if (small enough to solve):
    conquer, solve, return results
  else:
    divide input into a smaller pieces
    recurse on smaller pieces
    combine results and return
}
```

Merge Sort

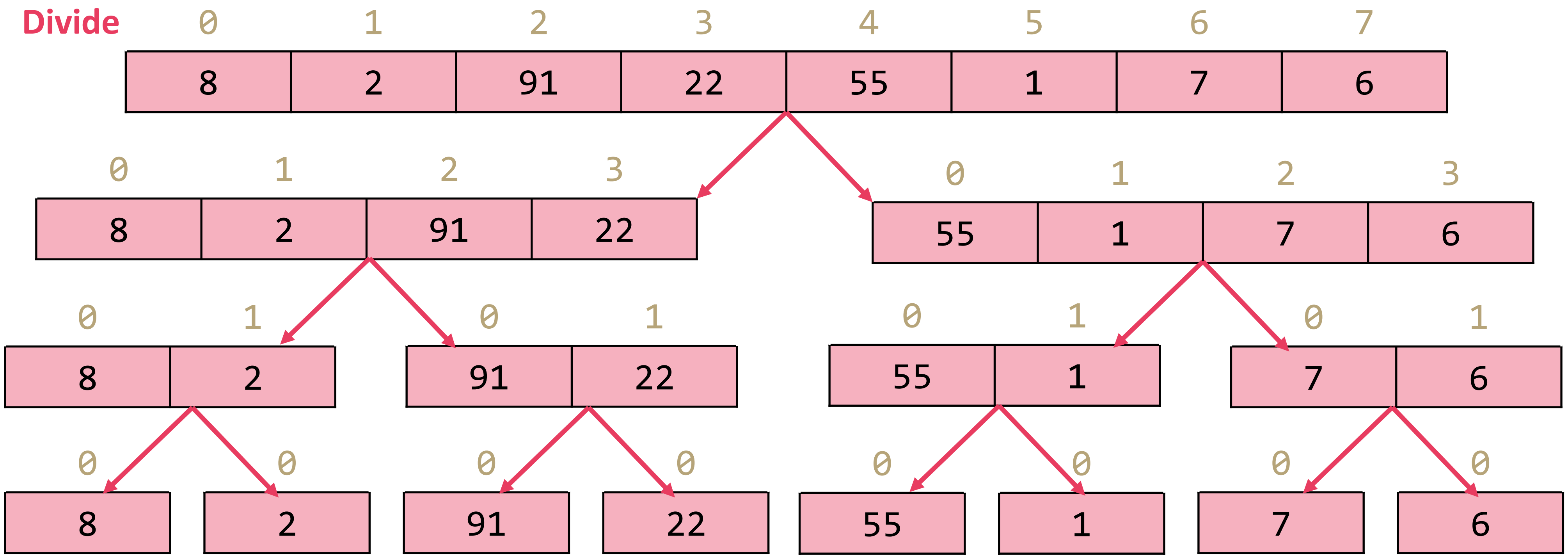
Simply divide in half each time

No extra conquer work needed!

The actual sorting happens here!



Merge Sort: Divide Step



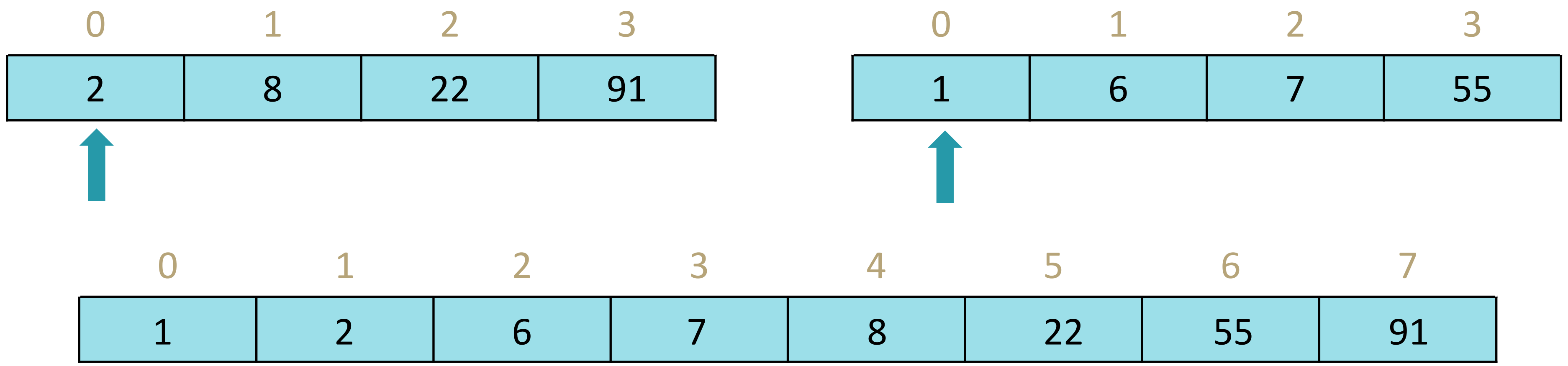
Recursive Case: split the array in half and recurse on both halves

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

Combine



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

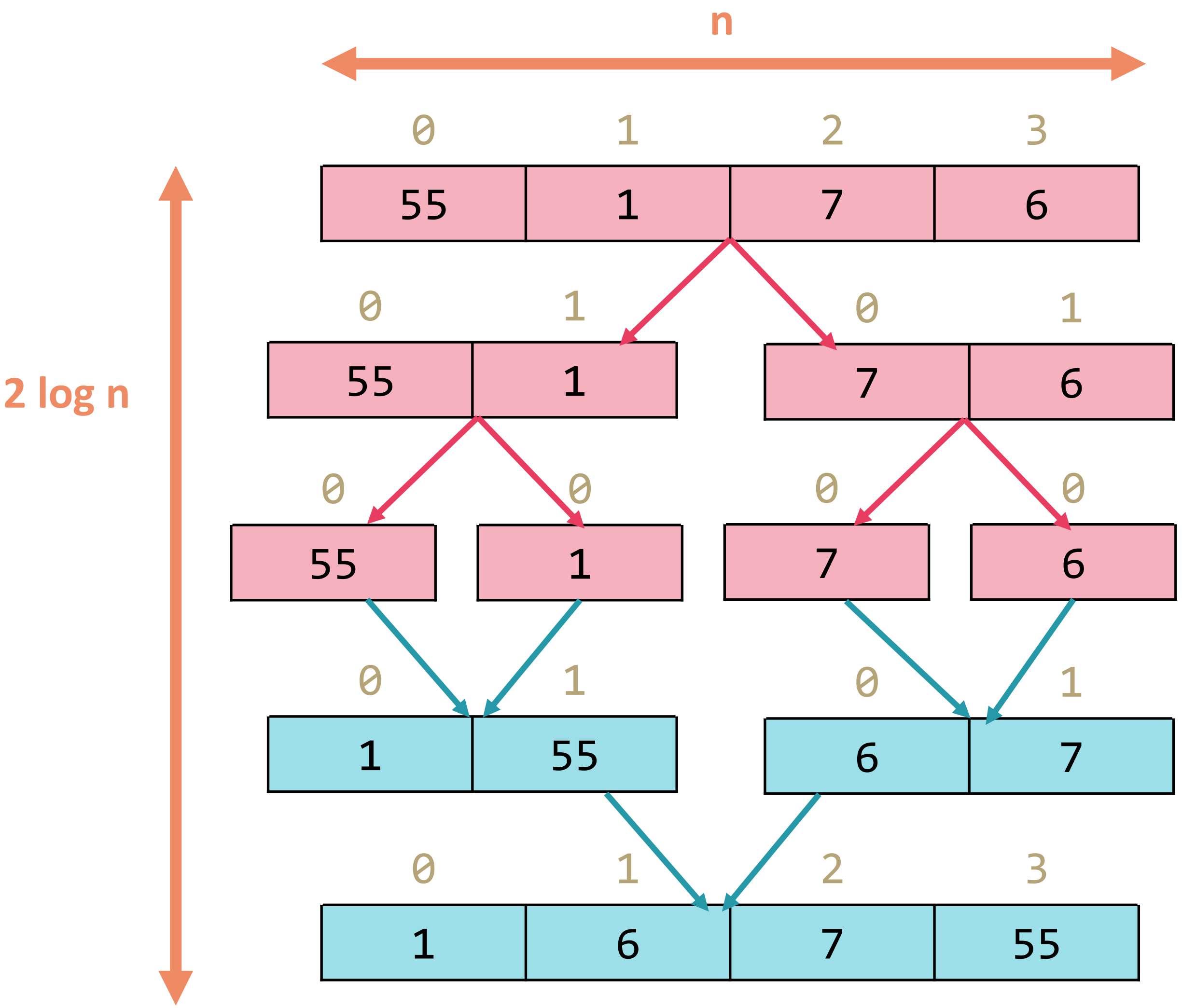
$= \Theta(n \log n)$

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!

Lecture Outline

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

Divide and Conquer

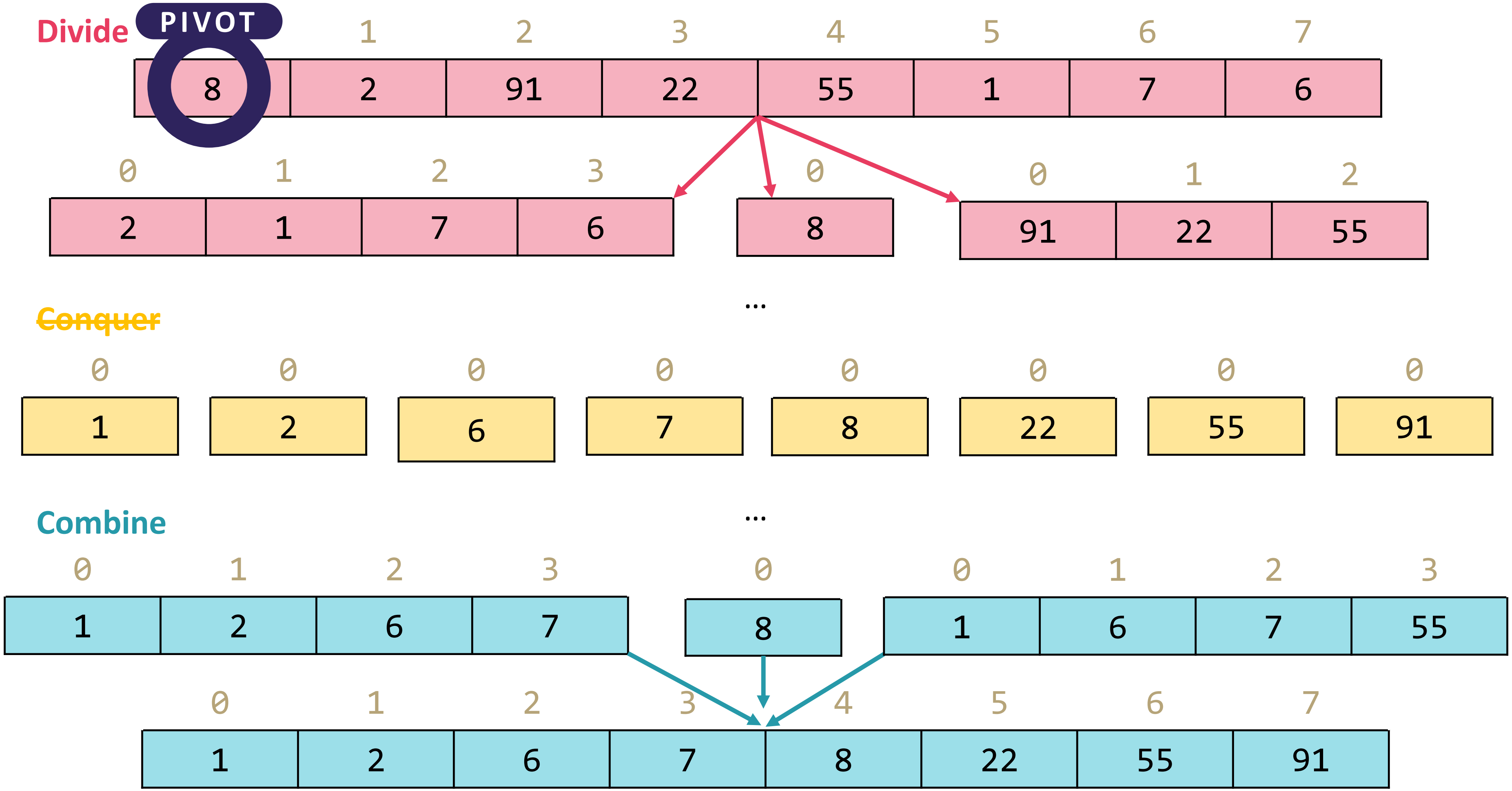
- There's more than one way to divide!
- Mergesort:
  - Split into two arrays.
  - Elements that just happened to be on the left and that happened to be on the right.
- Quicksort:
  - Split into two arrays.
  - Roughly, elements that are "small" and elements that are "large"
  - How to define "small" 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!

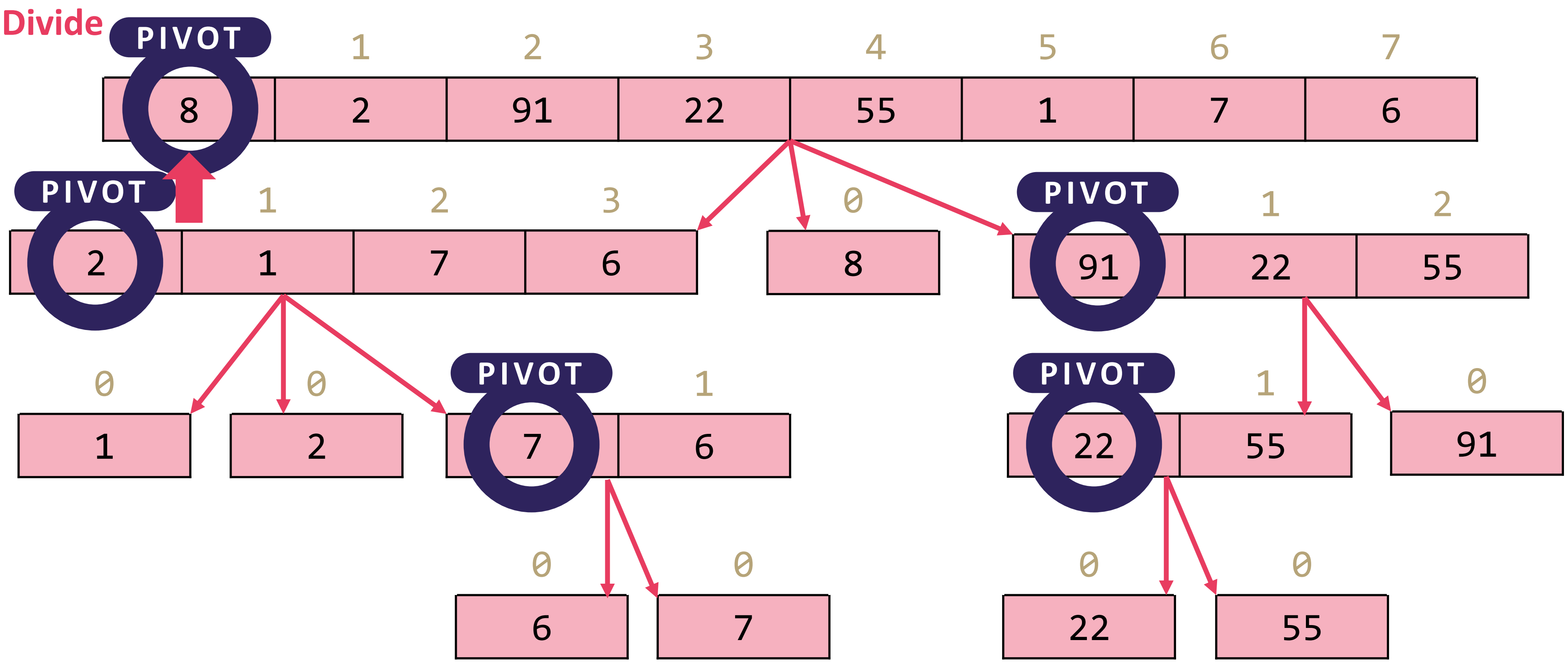
Simply concatenate the now-sorted arrays!



Quick Sort (v1): Divide Step

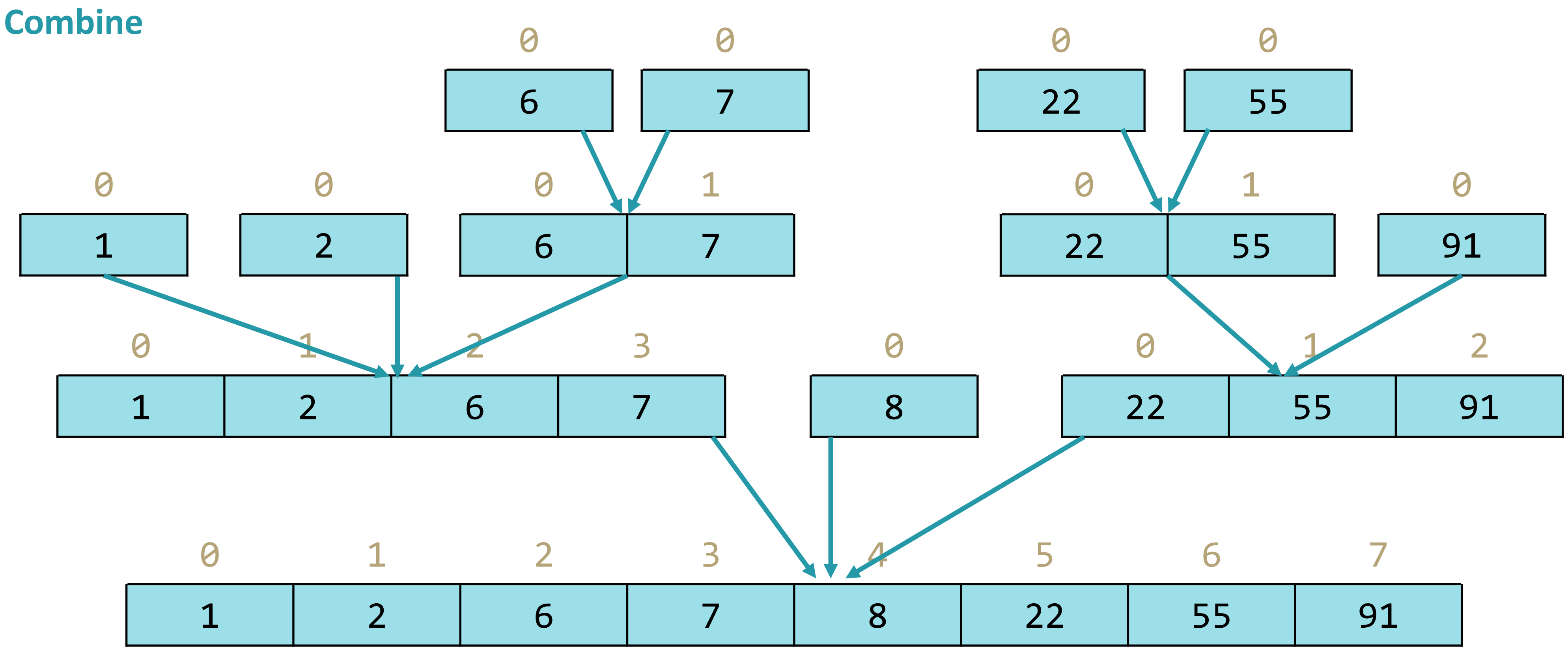
Recursive Case:
- Choose a "pivot" element
- Partition: linear scan through array, add smaller elements to one array and larger elements to another
- Recursively partition

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 ☺

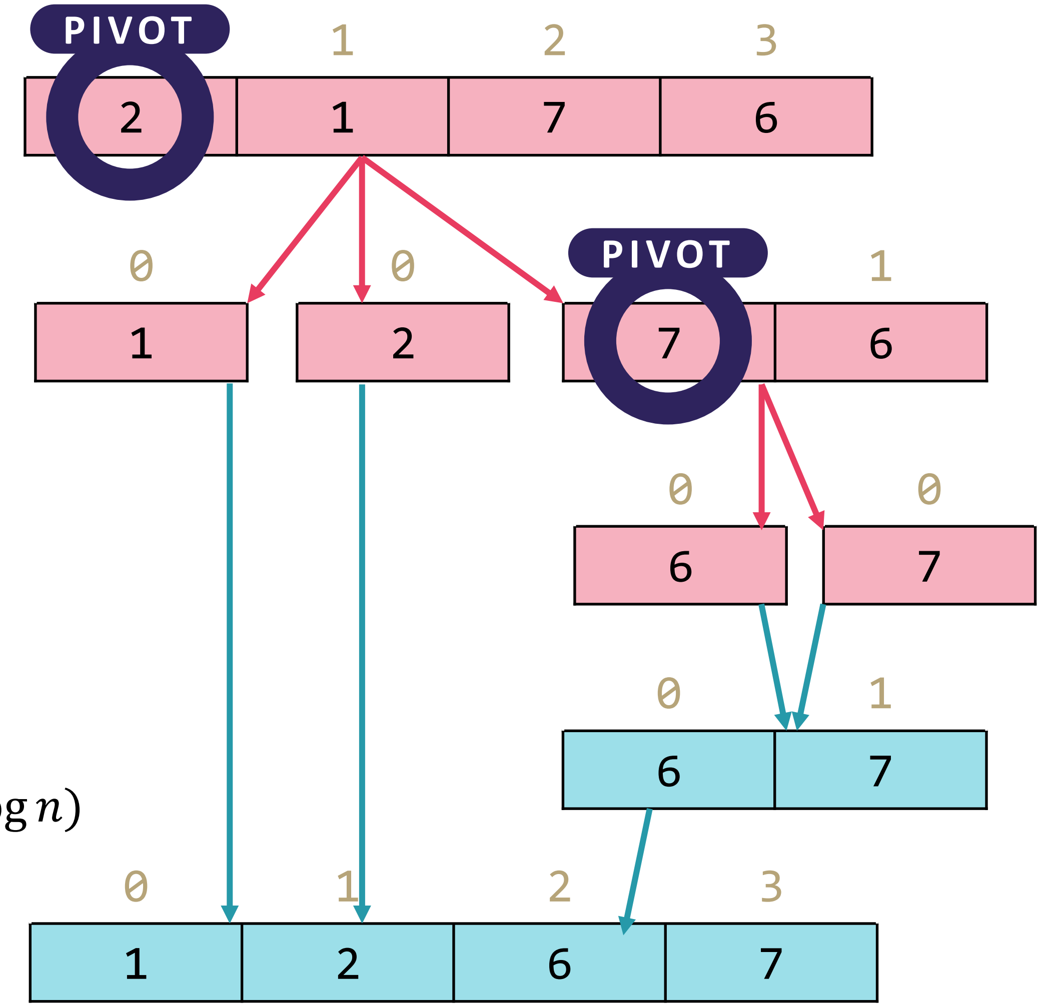


Quick Sort (v1)

```
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: Pivot only chops off one value

Best case: Pivot divides each array in half



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!

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 full array
  - Take the median of the first, last, and middle element
  - 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 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.
- Take the median of the first, last, and middle element

Most commonly used

  - 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!
- 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)

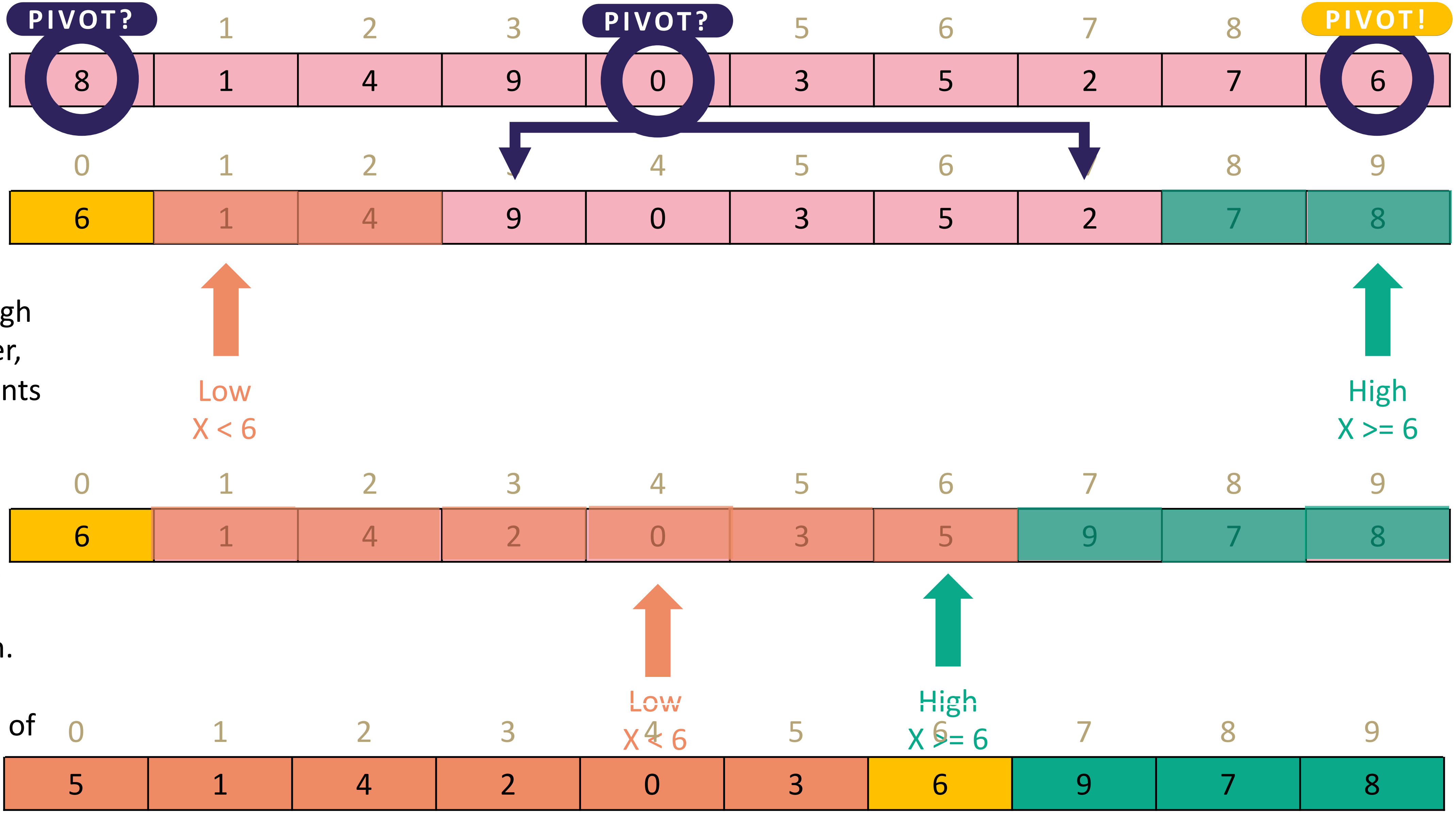
Divide

Select a pivot

Move pivot out of the way

Bring low and high pointers together, swapping elements if needed

Meeting point is where pivot belongs; swap in.
Now recurse on smaller portions of same array!



Quick Sort (v2: In-Place)

```
quickSort(list) {
   if (list.length == 1):
      return list
   else:
      pivot = choosePivot(list)
      smallerPart, largerPart = partition(pivot, list)
      smallerPart = quickSort(smallerPart)
      largerPart = quickSort(largerPart)
      return smallerPart + pivot + largerPart
}
```

choosePivot:
- Use one of the pivot selection strategies

partition:
- For in-place Quick Sort, series of swaps to build both partitions at once
- Tricky part: moving pivot out of the way and moving it back!
- Similar to Merge Sort divide step: two pointers, only move smaller one

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

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

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

Stable? No

In-place? Yes

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

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

* They actually use Tim Sort, which is very similar to Merge Sort in theory, but has some minor details different

STRATEGY 1:
ITERATIVE IMPROVEMENT

Insertion Sort

WORST $\Theta(n^2)$
BEST $\Theta(n)$

Simple, stable, low-overhead, great if already sorted.

IN-PLACE STABLE SPACE $\Theta(1)$

Selection Sort

WORST $\Theta(n^2)$
BEST $\Theta(n^2)$

Minimizes array writes, otherwise never preferred.

IN-PLACE SPACE $\Theta(1)$

STRATEGY 2:
IMPOSE STRUCTURE

Heap Sort

WORST $\Theta(n \log n)$
BEST $\Theta(n)$

Always good runtimes

IN-PLACE SPACE $\Theta(1)$

STRATEGY 3:
DIVIDE AND CONQUER

Merge Sort

WORST $\Theta(n \log n)$
BEST $\Theta(n \log n)$

Stable, very reliable! In-place variant is slower.

STABLE SPACE $\Theta(n)$

Quick Sort

WORST $\Theta(n^2)$
BEST $\Theta(n \log n)$

Fastest in practice (constant factors), bad worst case.

IN-PLACE SPACE $\Theta(1)$

Insertion Sort

WORST $\theta(n^2)$

BEST $\theta(n)$

Simple, stable, low-overhead, great if already sorted.

IN-PLACE STABLE SPACE $\theta(1)$

Heap Sort

WORST $\theta(n \log n)$

BEST $\theta(n)$

Always good runtimes

IN-PLACE SPACE $\theta(1)$

Merge Sort

WORST $\theta(n \log n)$

BEST $\theta(n \log n)$

Stable, very reliable! In-place variant is slower.

STABLE SPACE $\theta(n)$

Selection Sort

WORST $\theta(n^2)$

BEST $\theta(n^2)$

Minimizes array writes, otherwise never preferred.

IN-PLACE SPACE $\theta(1)$

Quick Sort

WORST $\theta(n^2)$

BEST $\theta(n \log n)$

Fastest in practice (constant factors), bad worst case.

IN-PLACE SPACE $\theta(1)$

Can we do better than n log n?

- For comparison sorts, **NO**. A proven lower bound!
  - Intuition: n elements to sort, no faster way to find “right place” than log n
- However, niche sorts can do better in specific situations!

Many cool niche sorts beyond the scope of 373!

Radix Sort (Wikipedia, VisuAlgo) - Go digit-by-digit in integer data. Only 10 digits, so no need to compare!

Counting Sort (Wikipedia)

Bucket Sort (Wikipedia)

External Sorting Algorithms (Wikipedia) - For big 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 DANCE EDITION

- 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