Lecture 23: Introduction to Sorting II

CSE 373: Data Structures and Algorithms

Warm Up

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

Heap Sort

https://www.youtube.com/watch?v=Xw2D9aJRBY4

1. run Floyd's buildHeap on your data
2. call removeMin n times

```
public void heapSort(input) {
   E[] heap = buildHeap(input)
   E[] output = new E[n]
   for (n)
      output[i] =
removeMin(heap)
}
```

Worst case runtime?

Best case runtime?

In-practice runtime?

Stable? No

In-place? If we get clever…

In Place Heap Sort

0	1	2	3	4	5	6	7	8	9
1	4	2	14	15	18	16	17	20	22

Current Item

Heap

Sorted Items

0	1	2	3	4	5	6	7	8	9
22	4	2	14	15	18	16	17	20	1

Current Item

percolateDown(22)

Heap

Sorted Items

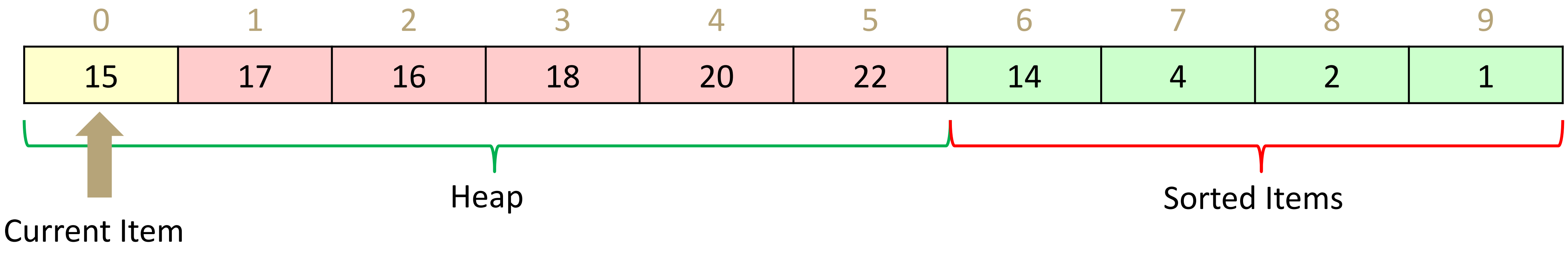
0	1	2	3	4	5	6	7	8	9
2	4	16	14	15	18	22	17	20	1

Current Item

Heap

Sorted Items

In Place Heap Sort



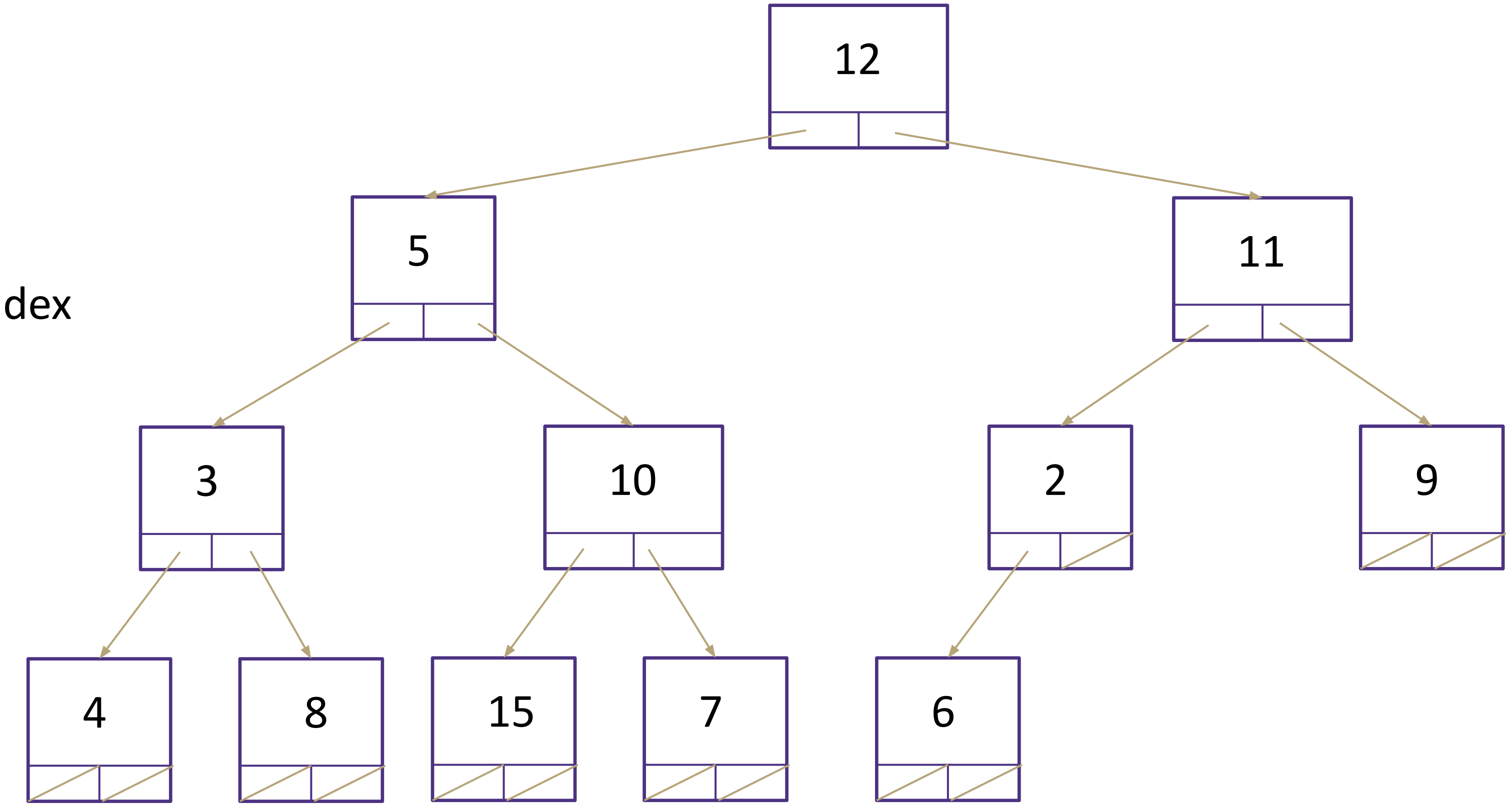

```
public void inPlaceHeapSort(input) {
   buildHeap(input) // alters original array
   for (n : input)
      input[n – i - 1] = removeMin(heap)
}
```

Worst case runtime?	O(nlogn)
Best case runtime?	O(nlogn)
In-practice runtime?	O(nlogn)
Stable?	No
In-place?	Yes

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index

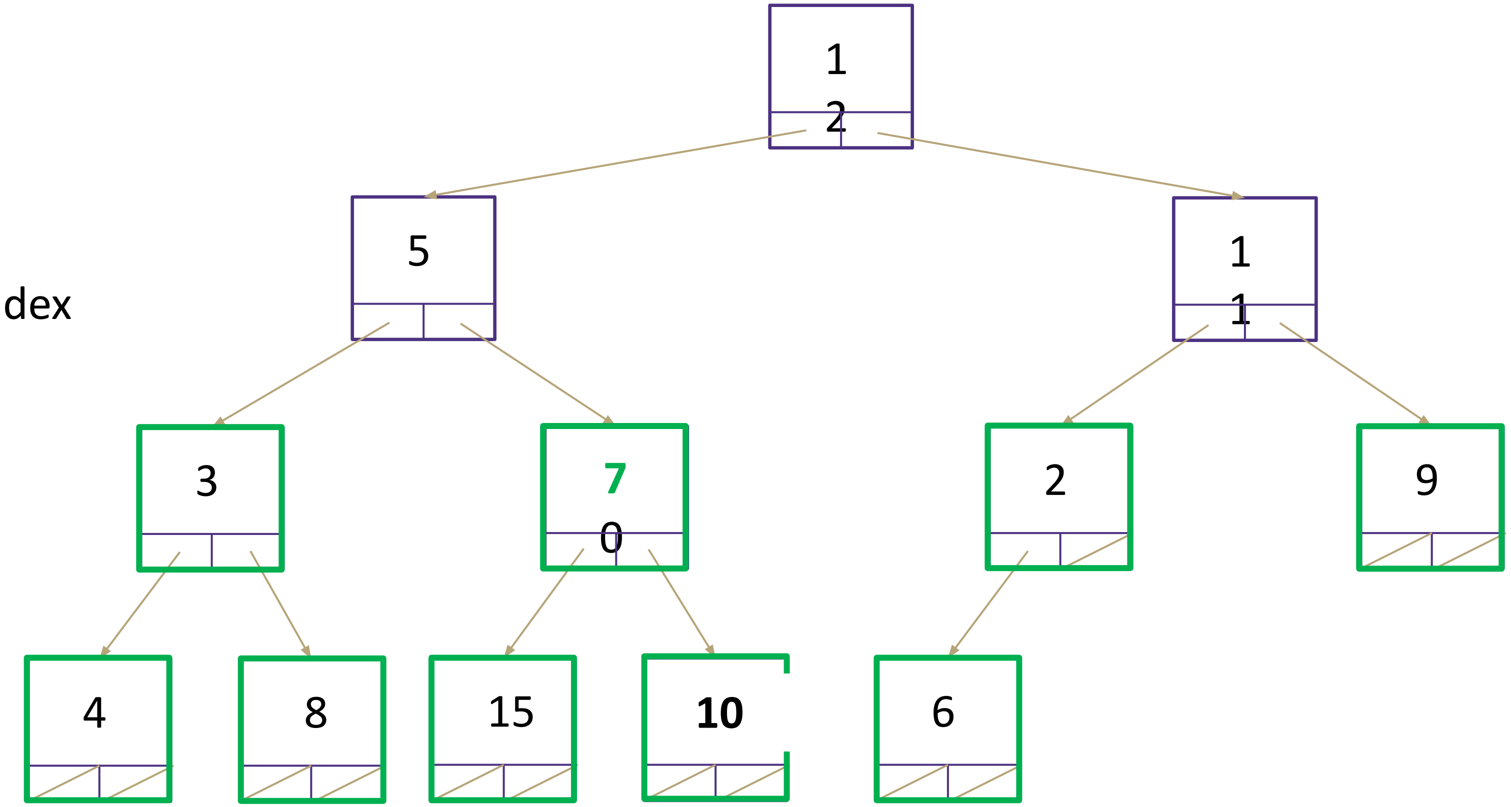


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3



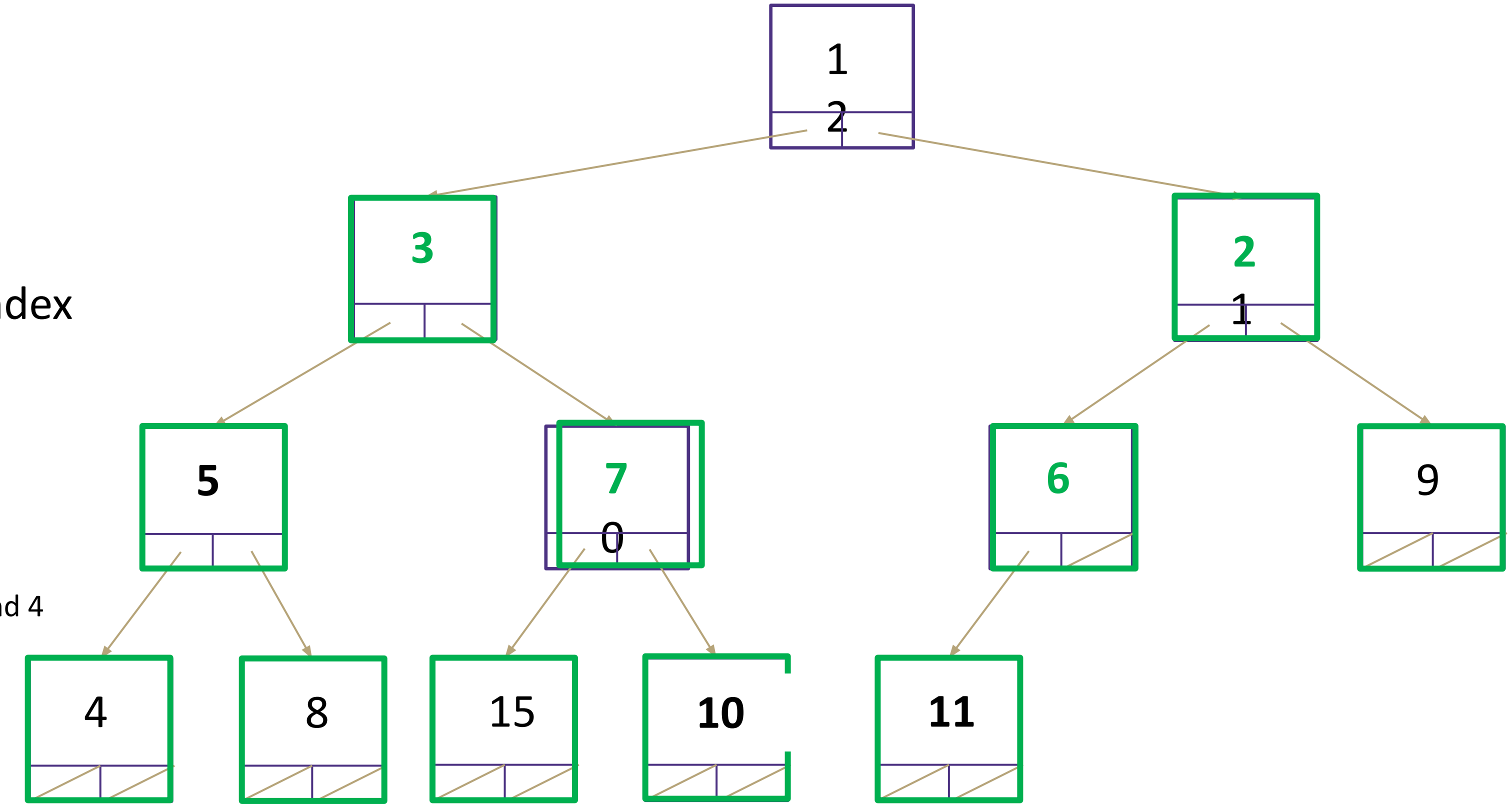
0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2

keep percolating down
like normal here and swap 5 and 4

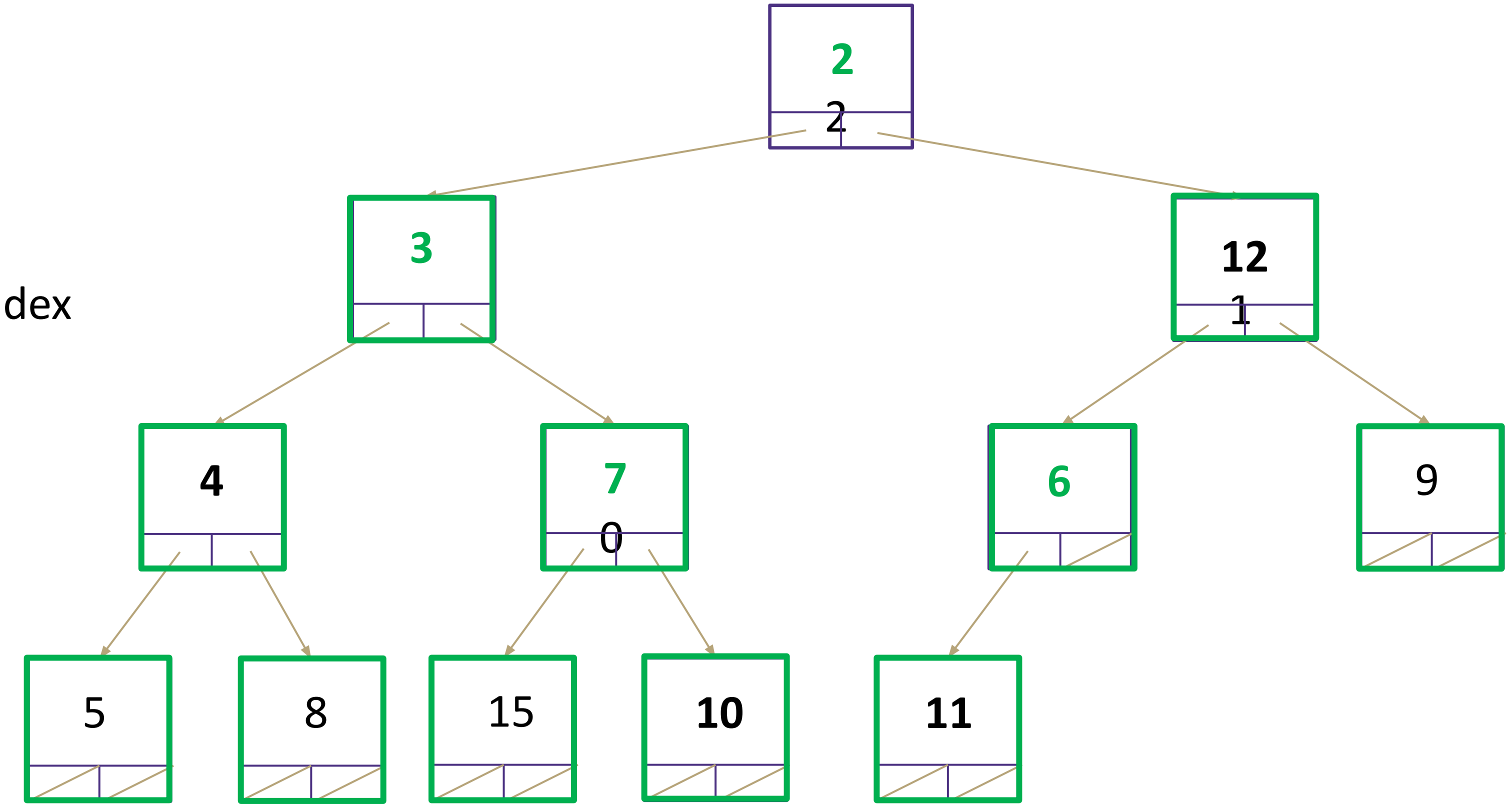


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	7	2	9	4	8	15	10	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
    1. percolateDown level 4
    2. percolateDown level 3
    3. percolateDown level 2
    4. percolateDown level 1

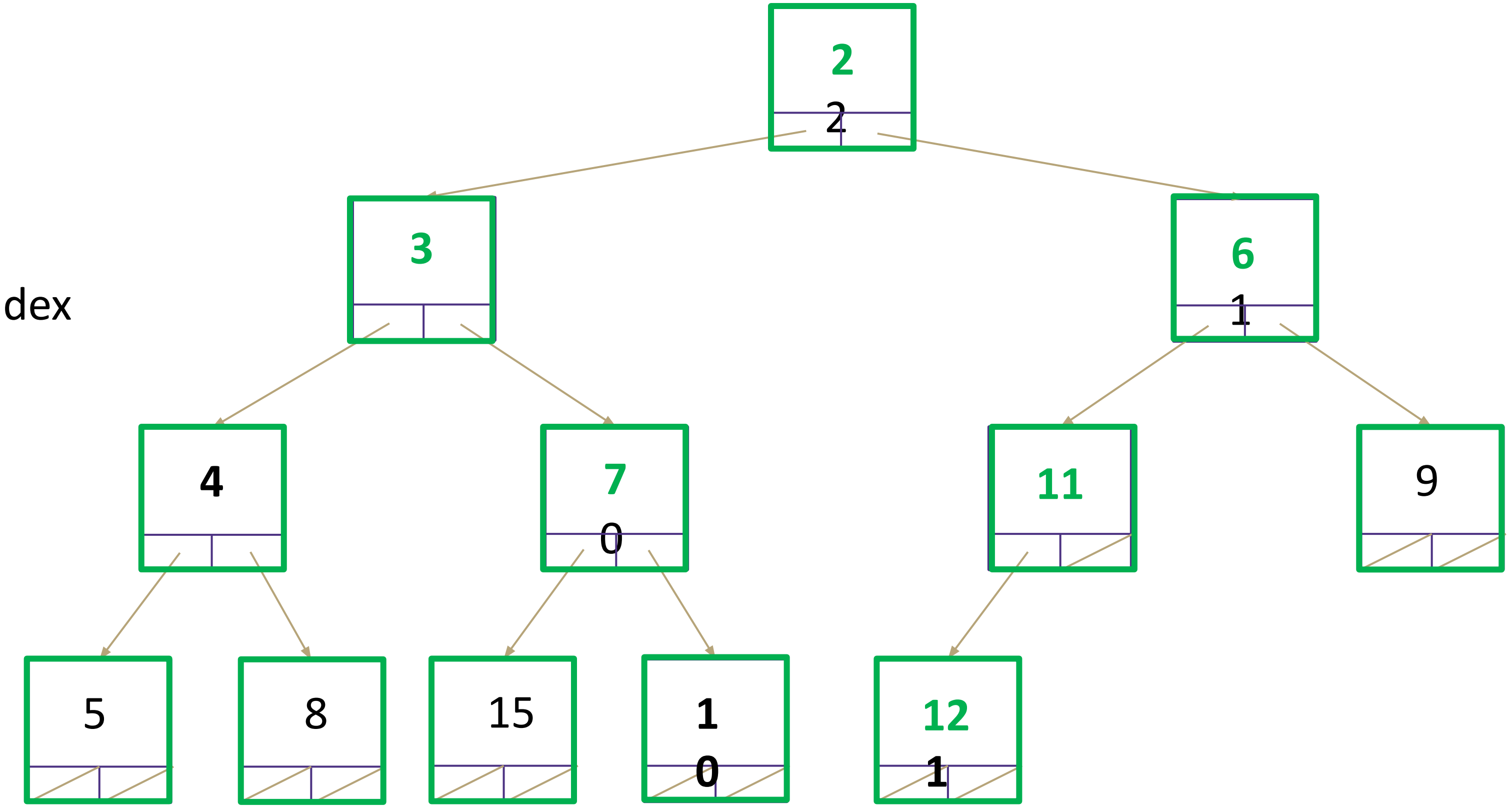


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	3	2	4	7	6	9	5	8	15	10	11		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2
   4. percolateDown level 1



0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	6	4	7	11	9	5	8	15	10	12		

Is It Really Faster?

Assume the tree is **perfect**

- the proof for complete trees just gives a different constant factor.

percolateDown() doesn't take $\log n$ steps each time!

Half the nodes of the tree are leaves

- Leaves run percolate down in constant time

1/4 of the nodes have at most 1 level to travel

1/8 the nodes have at most 2 levels to travel

etc...

work(n) $\approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 1 \cdot (\log n)$

Closed form Floyd's buildHeap

find a pattern -> powers of 2

Summation!

? = upper limit should give last term

We don't have a summation for this! Let's make it look more like a summation we do know.

Infinite geometric series

Floyd's buildHeap runs in O(n) time!

Announcements

EX 5 due today, EX 6 out

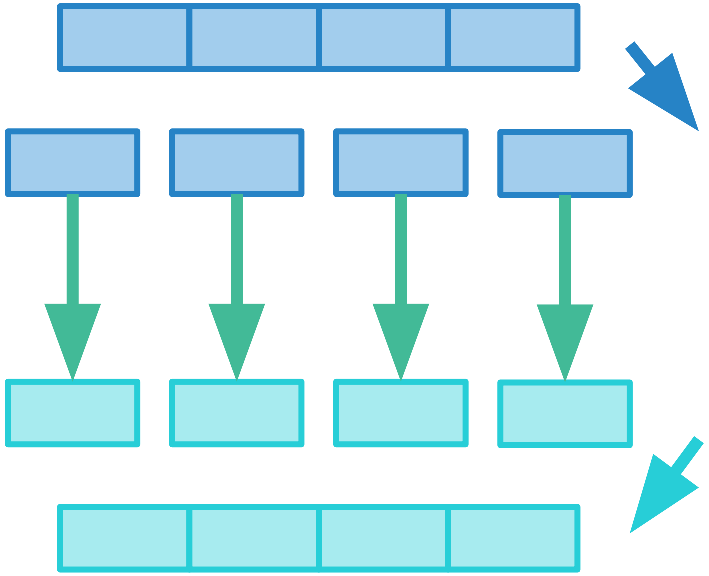
P4 checkpoint Sunday at midnight

- P4 due Wed 6/1
- OH are still too quiet

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
}
```

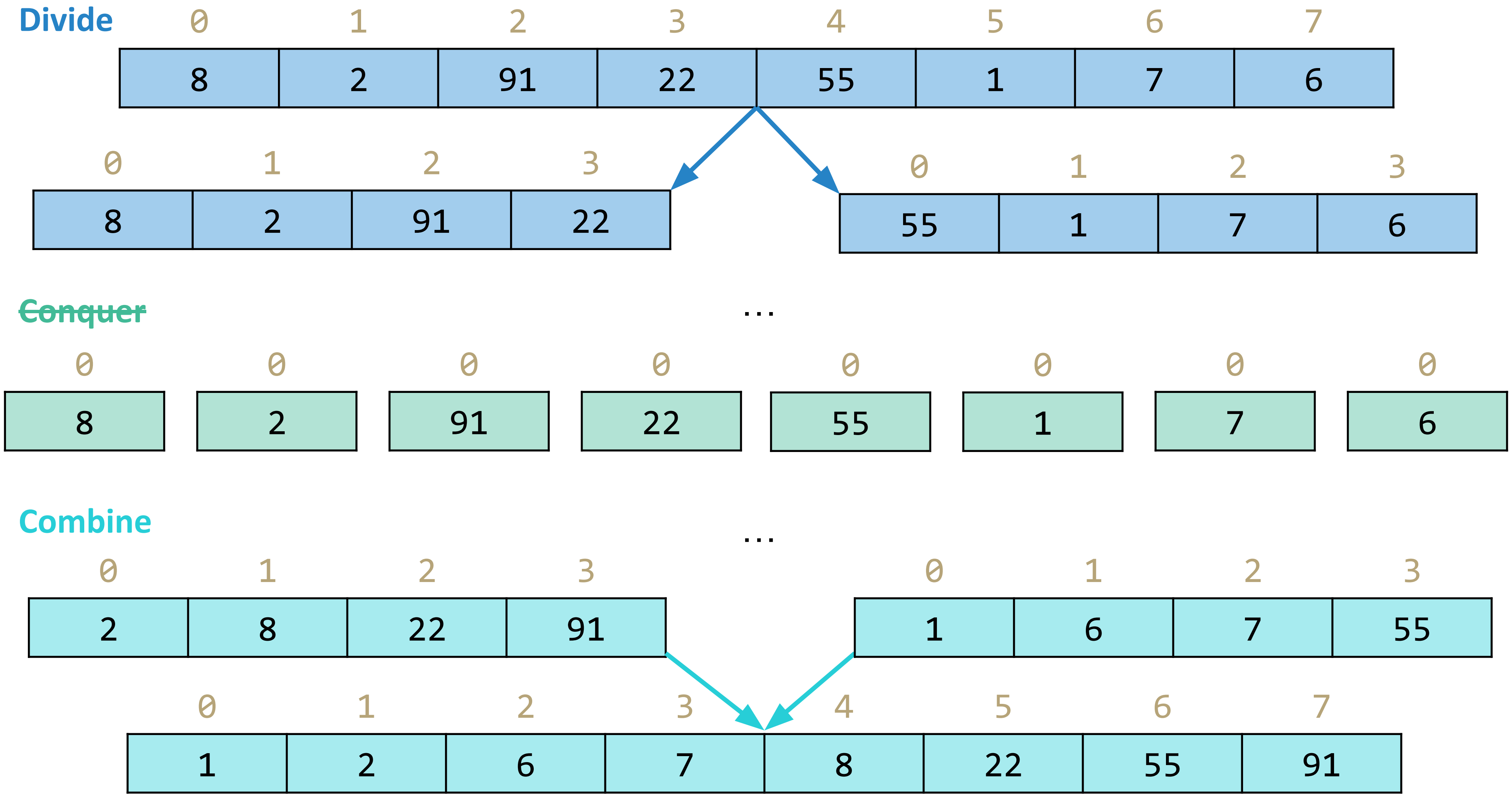
Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

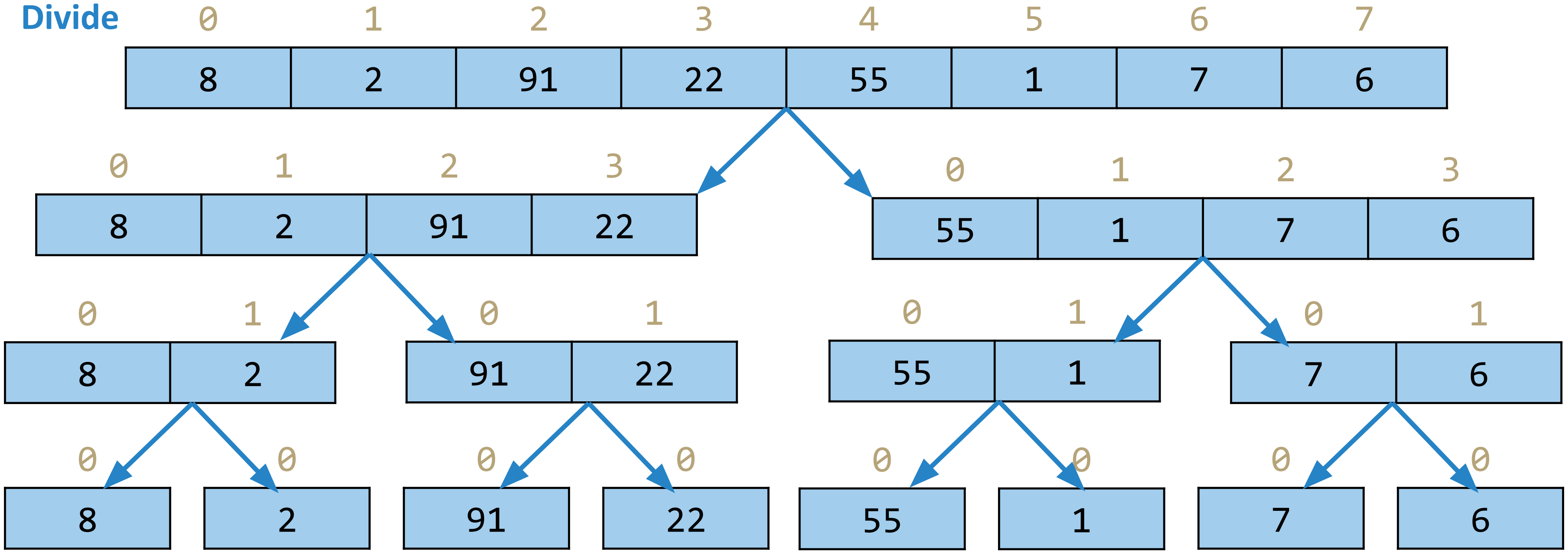
Simply divide in half each time

No extra conquer work needed!

The actual sorting happens here!



Merge Sort: Divide Step



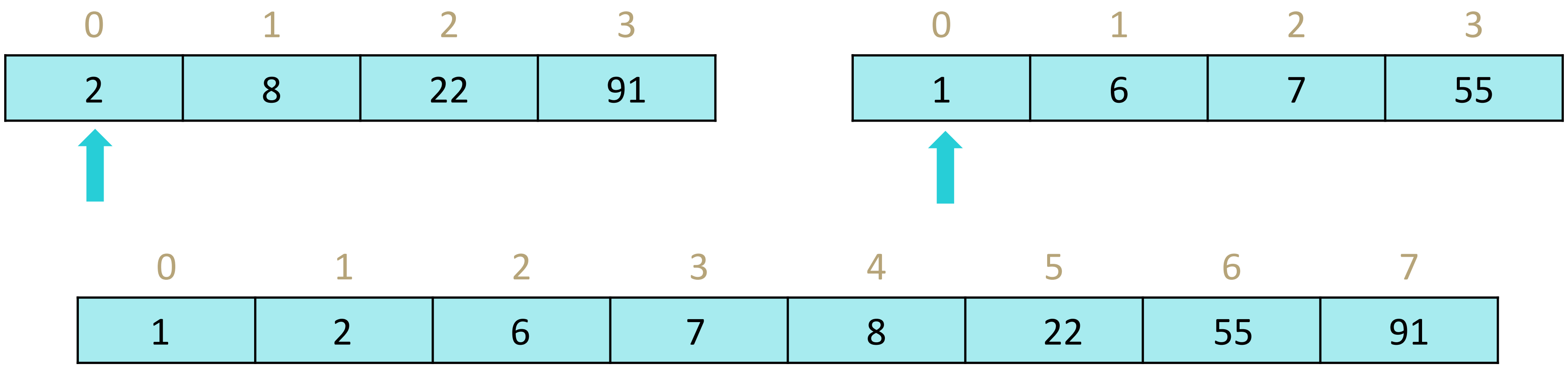
Recursive Case: split the array in half and recurse on both halves

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

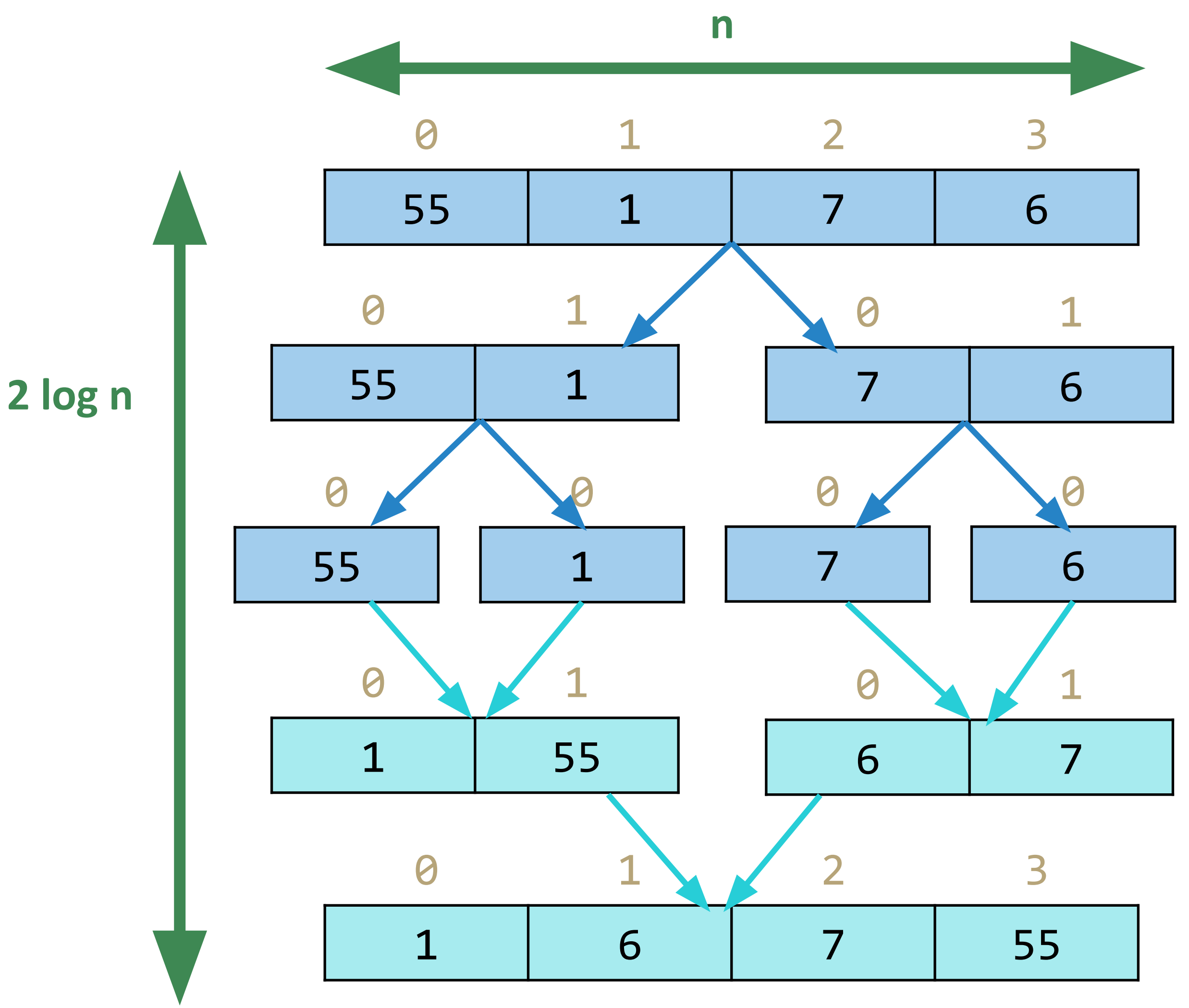
$= \Theta(n \log n)$

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!

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!

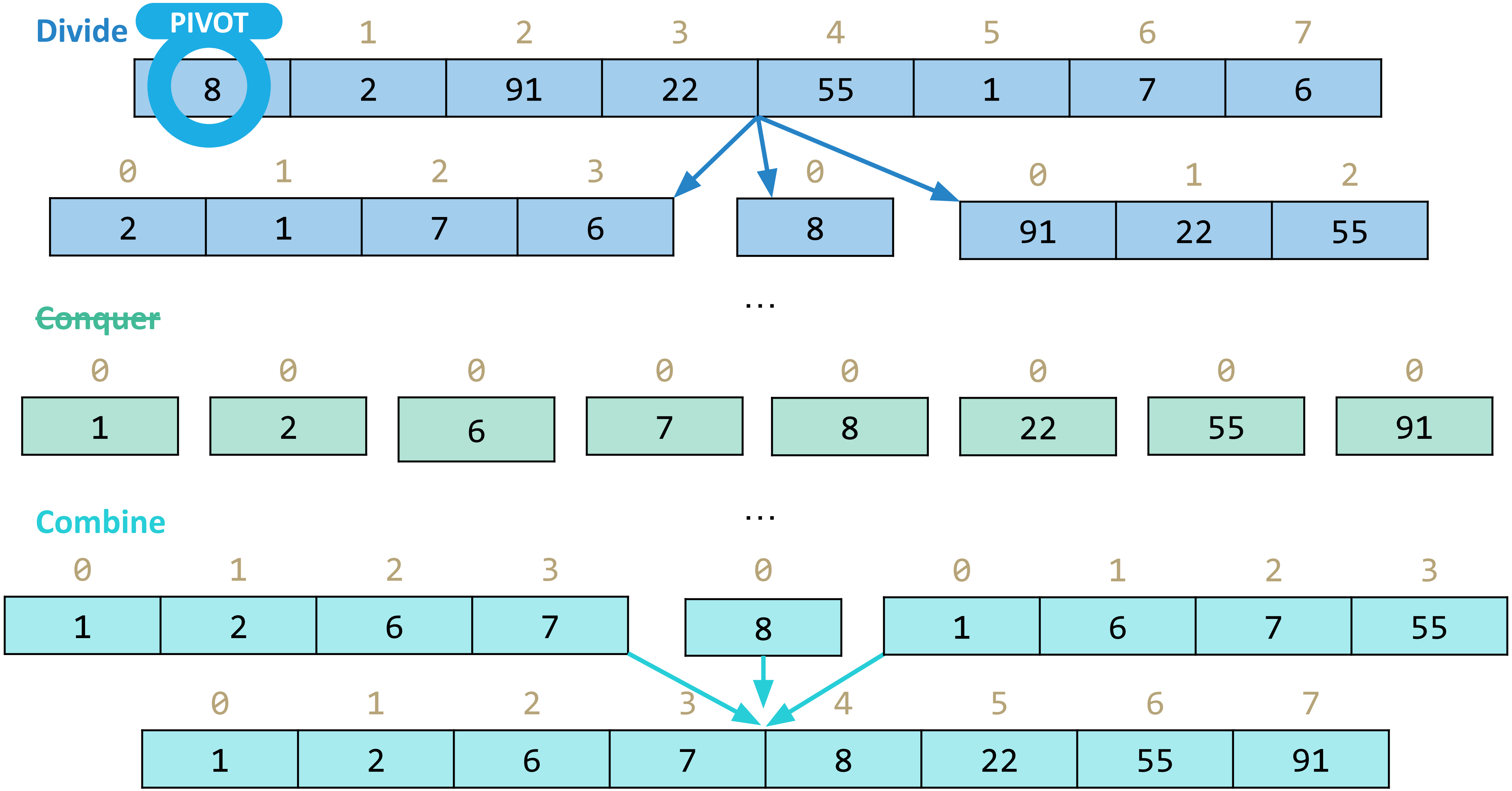
https://www.youtube.com/watch?v=ywWBy6J5gz8

Quick Sort (v1)

Choose a “pivot” element, partition array relative to it!

Again, no extra conquer step needed!

Simply concatenate the now-sorted arrays!



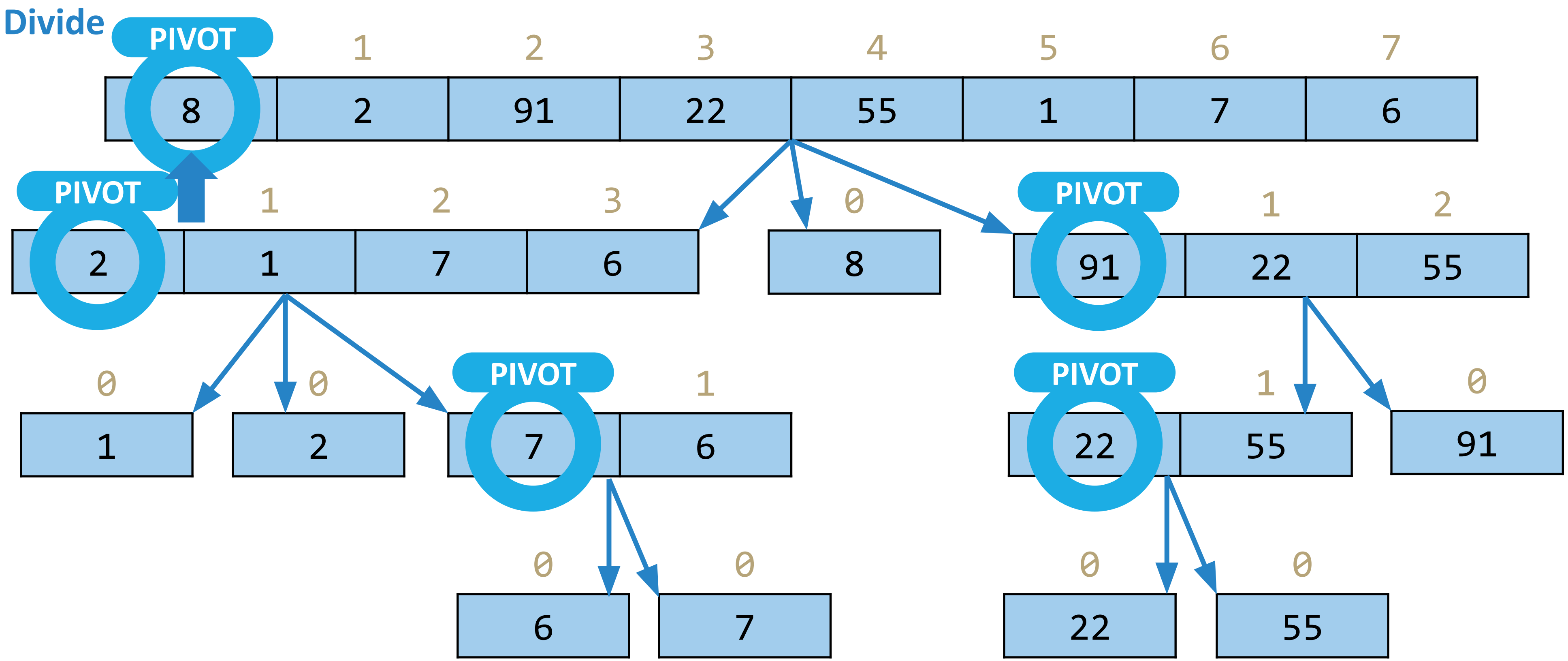
Quick Sort (v1): Divide Step

Recursive Case:

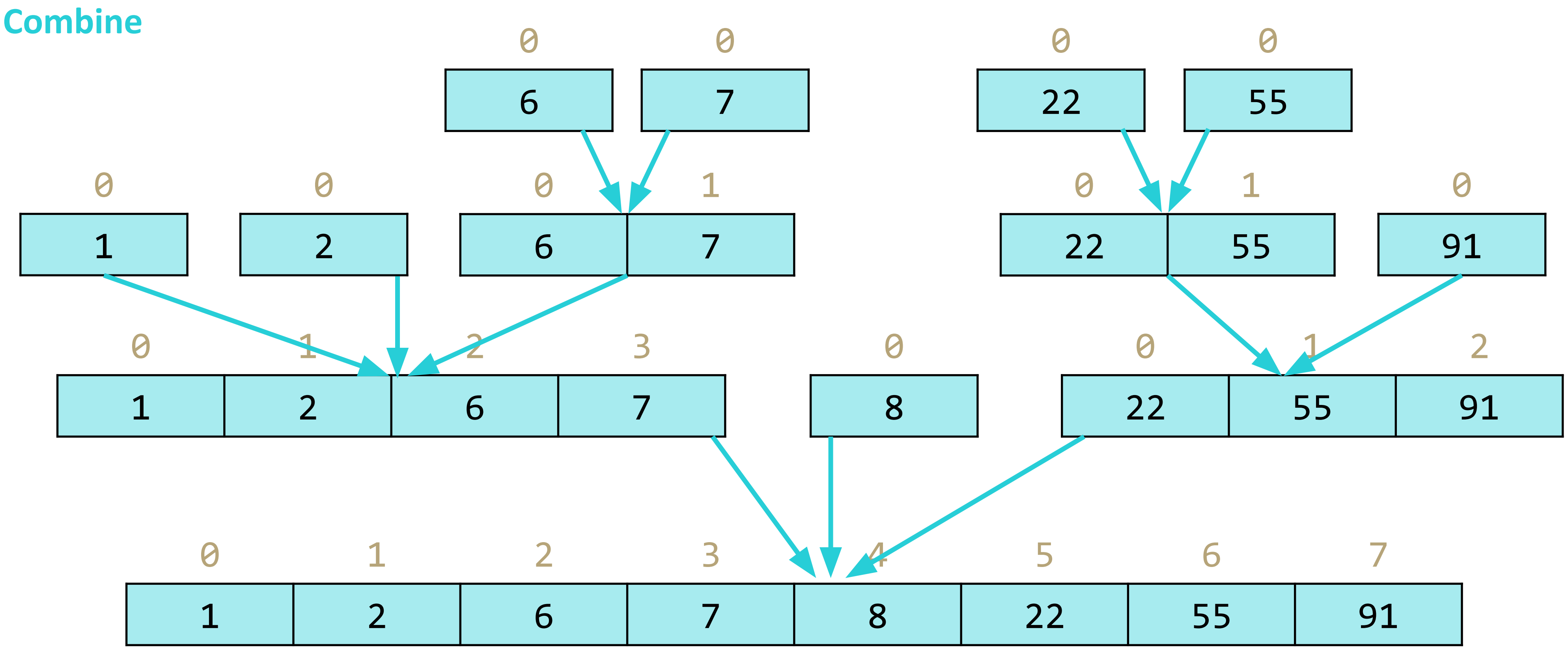
- Choose a "pivot" element
- Partition: linear scan through array, add smaller elements to one array and larger elements to another
- Recursively partition

Base Case:

- When array hits size 1, stop dividing.



Quick Sort (v1): Combine Step

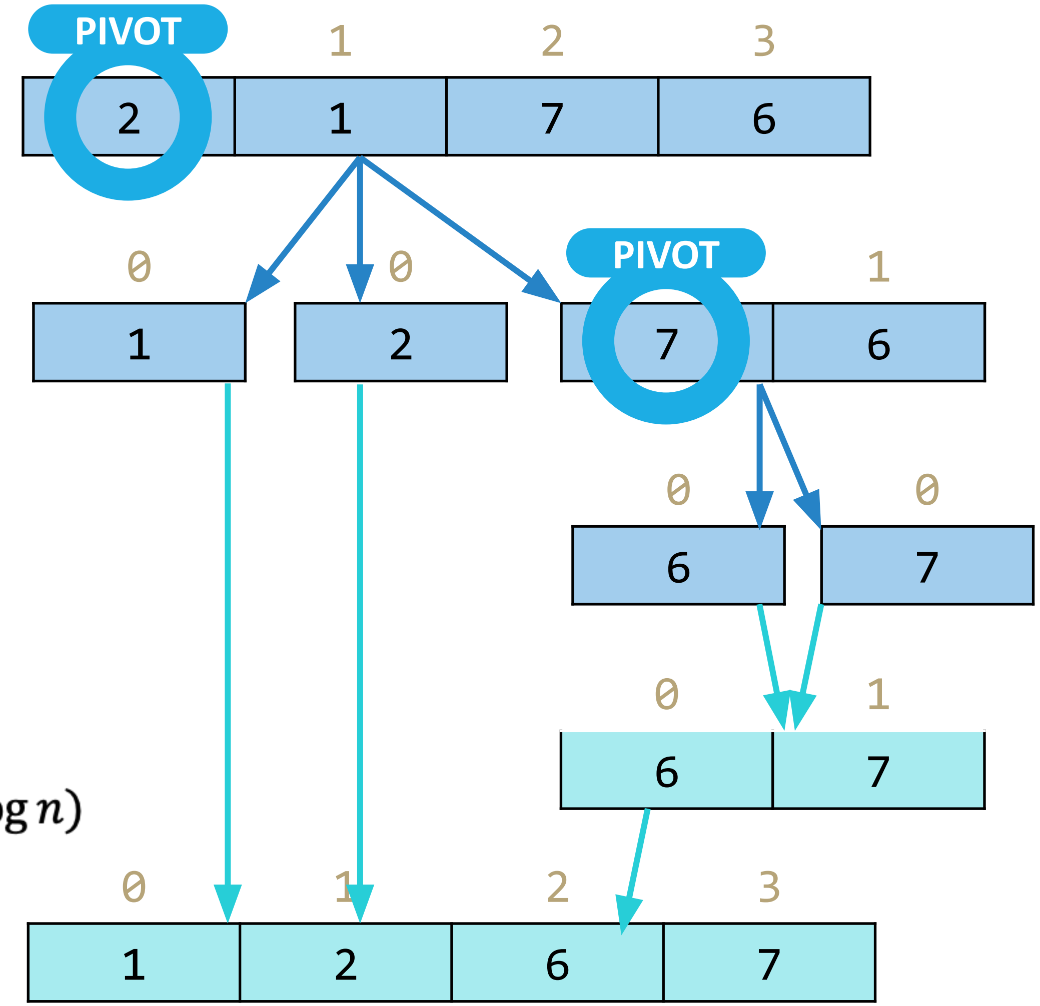


Simply concatenate the arrays that were created earlier! Partition step already left them in order ☺️

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

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? 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

- 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

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)

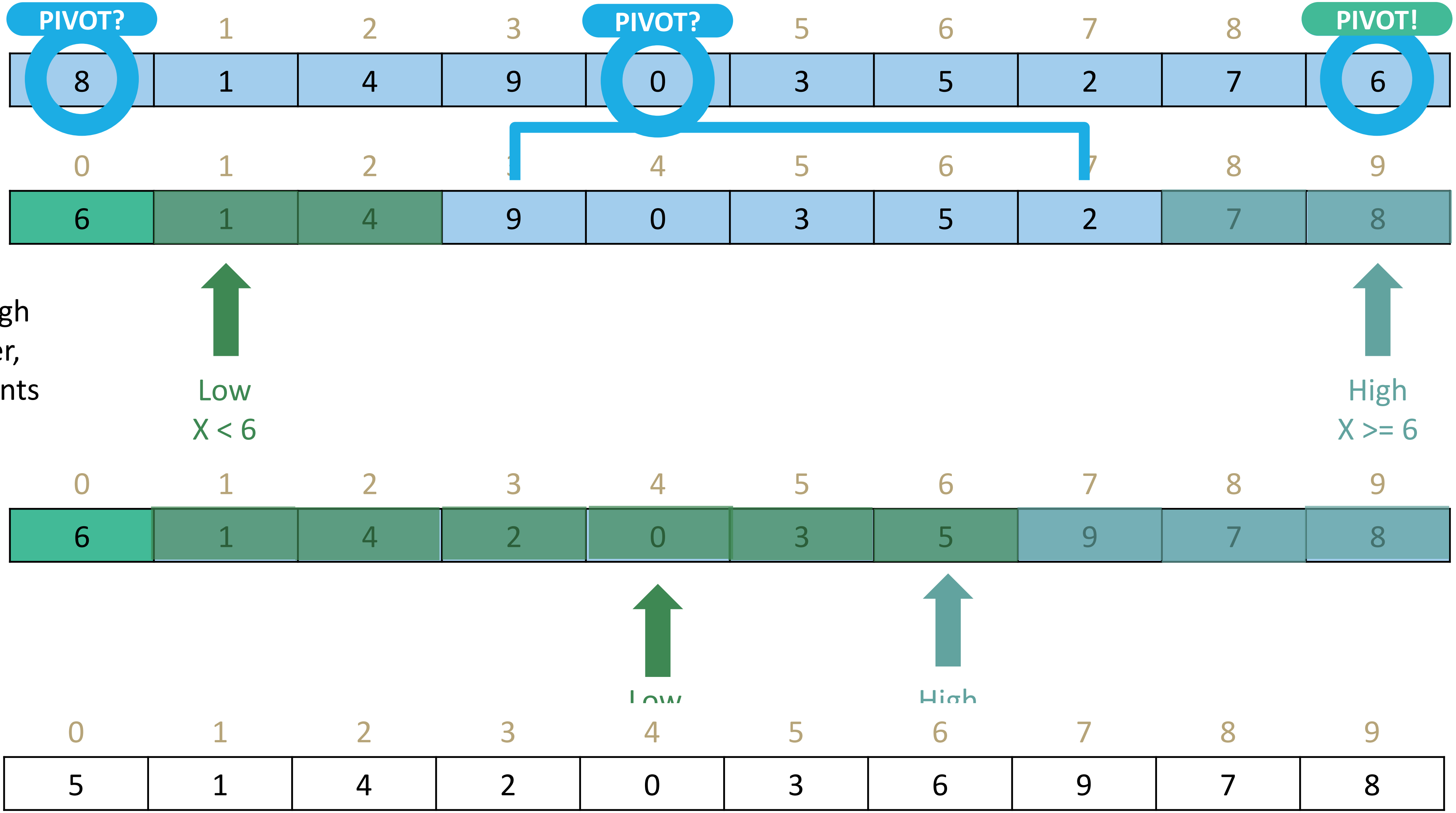
Divide

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?

Best case runtime?

In-practice runtime?

Stable? No

In-place? Yes

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

Can we do better?

We'd really like to avoid hitting the worst case.

Key to getting a good running time, is always cutting the array (about) in half.

How do we choose a good pivot?

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 as a pivot

Pivots

Just take the first element
- fast to find a pivot
- But (e.g.) nearly sorted lists get $\Omega(n^2)$ behavior overall

Take the median of the first, last, and middle element
- Guaranteed to not have the absolute smallest value.
- On real data, this works quite well…
- But worst case is still $\Omega(n^2)$

Median of three is a common choice in practice

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 as a pivot
- somewhat slow constant factors
- Get $O(n \log n)$ running time with probability at least $1 - 1/n^2$
- "adversaries" can't make it more likely that we hit the worst case.

Sorting: Summary

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

BEST

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

IN-PLACE STABLE SPACE

Selection Sort

WORST

BEST

Minimizes array writes, otherwise never preferred.

IN-PLACE SPACE

STRATEGY 2:
IMPOSE STRUCTURE

Heap Sort

WORST

BEST

Always good runtimes

IN-PLACE SPACE

STRATEGY 3:
DIVIDE AND CONQUER

Merge Sort

WORST

BEST

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

STABLE SPACE

Quick Sort

WORST

BEST

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

IN-PLACE SPACE

Insertion Sort

WORST

BEST

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

IN-PLACE STABLE SPACE

Merge Sort

WORST

BEST

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

STABLE SPACE

Heap Sort

WORST

BEST

Always good runtimes

IN-PLACE SPACE

Selection Sort

WORST

BEST

Minimizes array writes, otherwise never preferred.

IN-PLACE SPACE

Quick Sort

WORST

BEST

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

IN-PLACE SPACE

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!

DANCE EDITION

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