Lecture 22: Introduction to Sorting II

CSE 373: Data Structures and Algorithms

Warm Up

Slido Event #3138899
https://app.sli.do/event/dpfe5mrHKsbDYxqz3hBD6R

What sorting algorithm do the following steps represent. The steps are not necessarily consecutive but they are in the correct sequence

Sort 1

```
[23, 37, 48, 34, 11, 34, 37, 34, 23, 39, 41, 47]

[11, 37, 48, 34, 23, 34, 37, 34, 23, 39, 41, 47]

[11, 23, 24, 34, 48, 34, 37, 34, 44, 39, 41, 47]

[11, 23, 24, 34, 34, 37, 48, 44, 37, 39, 41, 47]
```

Sort 2

```
[2, 27, 18, 12, 14, 43, 8, 5, 41, 32, 48, 10, 37]

[2, 27, 12, 18, 14, 43, 8, 5, 41, 32, 48, 10, 37]

[2, 8, 12, 14, 18, 27, 43, 5, 41, 32, 48, 10, 37]

[2, 8, 12, 14, 18, 27, 43, 5, 10, 32, 37, 41, 48]
```

https://visualgo.net/en/sorting

Announcements

Final Exam Friday May 26th in class

<u>Topics</u>

Classes Week 5 – Week 8

- Heaps
- Graphs
  - Graph Modeling
  - BFS/DFS
  - Topological Sort
  - Dijkstra's
  - MSTs
- Disjoint Sets
- Sorting Algorithms

Intro to Sorting
Selection Sort
Insertion 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!

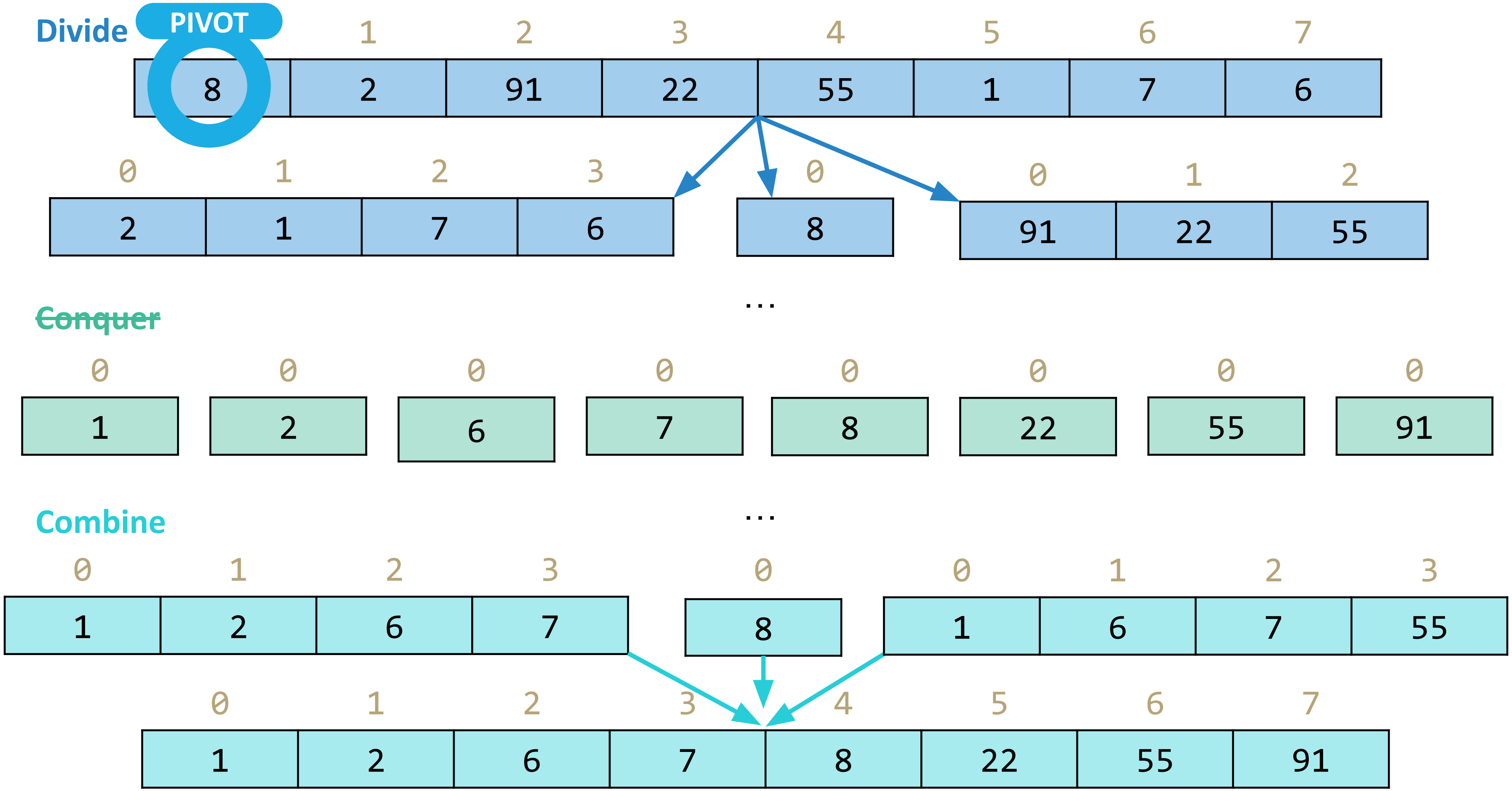
Quick Sort (v1)

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

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

Again, no extra conquer step needed!

Simply concatenate the now-sorted arrays!



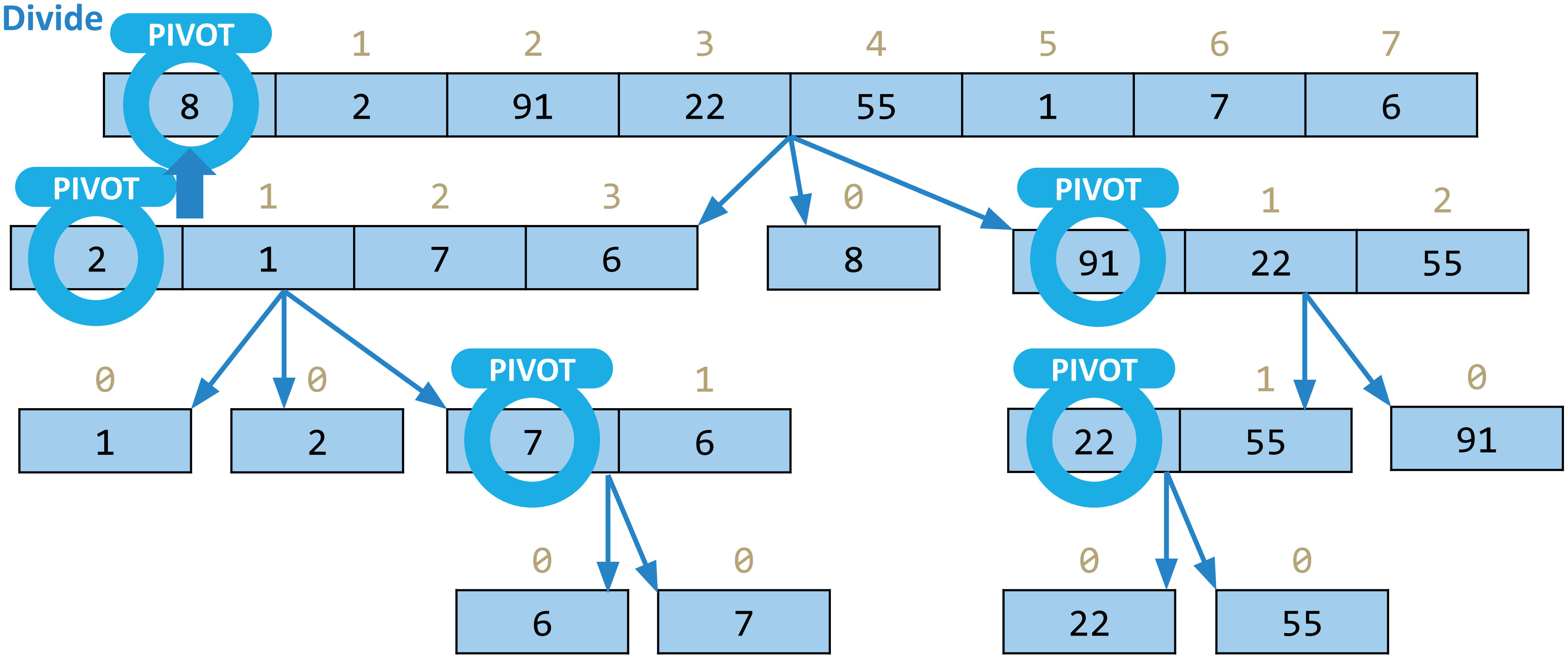
Quick Sort (v1): Divide Step

Recursive Case:

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

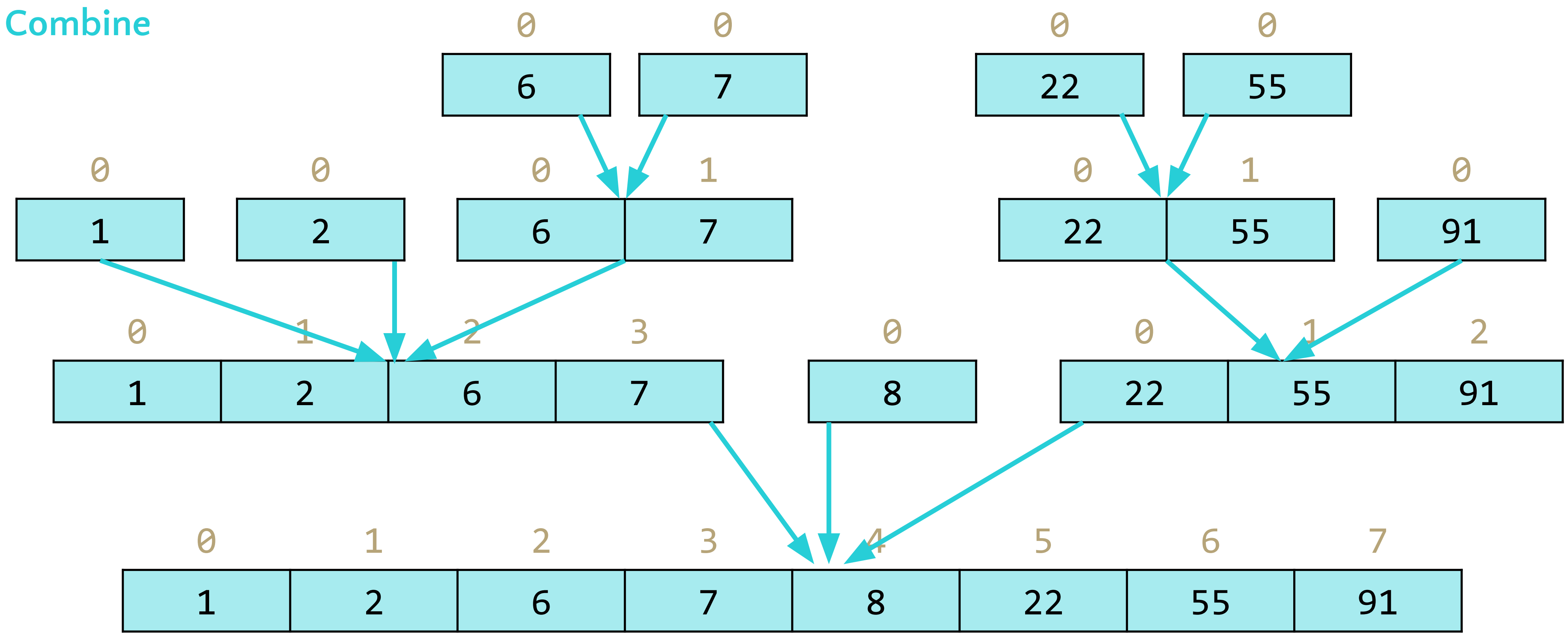
Base Case:

- When array hits size 1, stop dividing



Quick Sort (v1): Combine Step

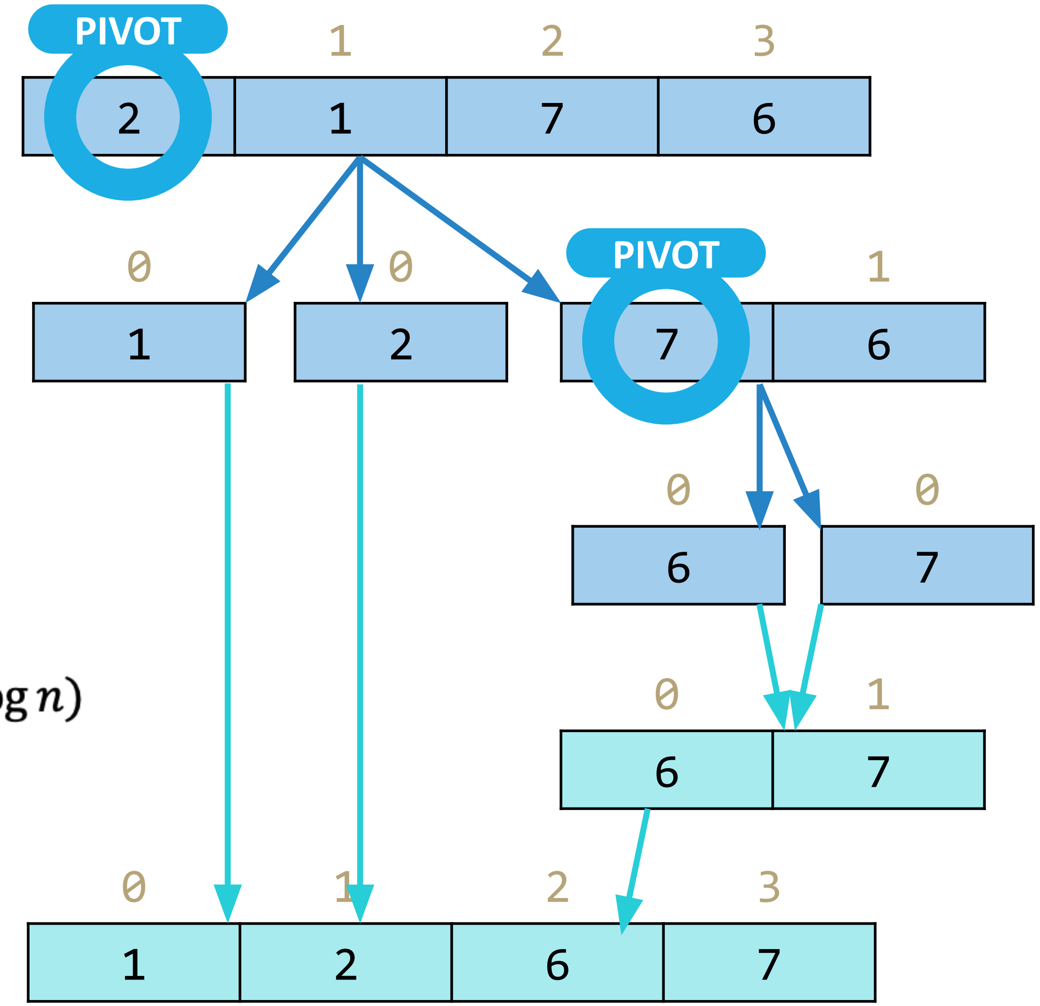
Simply concatenate the arrays that were created earlier! Partition step already left them in order ☺️



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!

Useful for: Fast sorting of primitives!
([This is what Java uses for Primitives])

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 $\mathbf{1\text{–}1/n^2}$
- No simple worst-case input (e.g. sorted, reverse sorted)

Quick Sort (v2: In-Place)

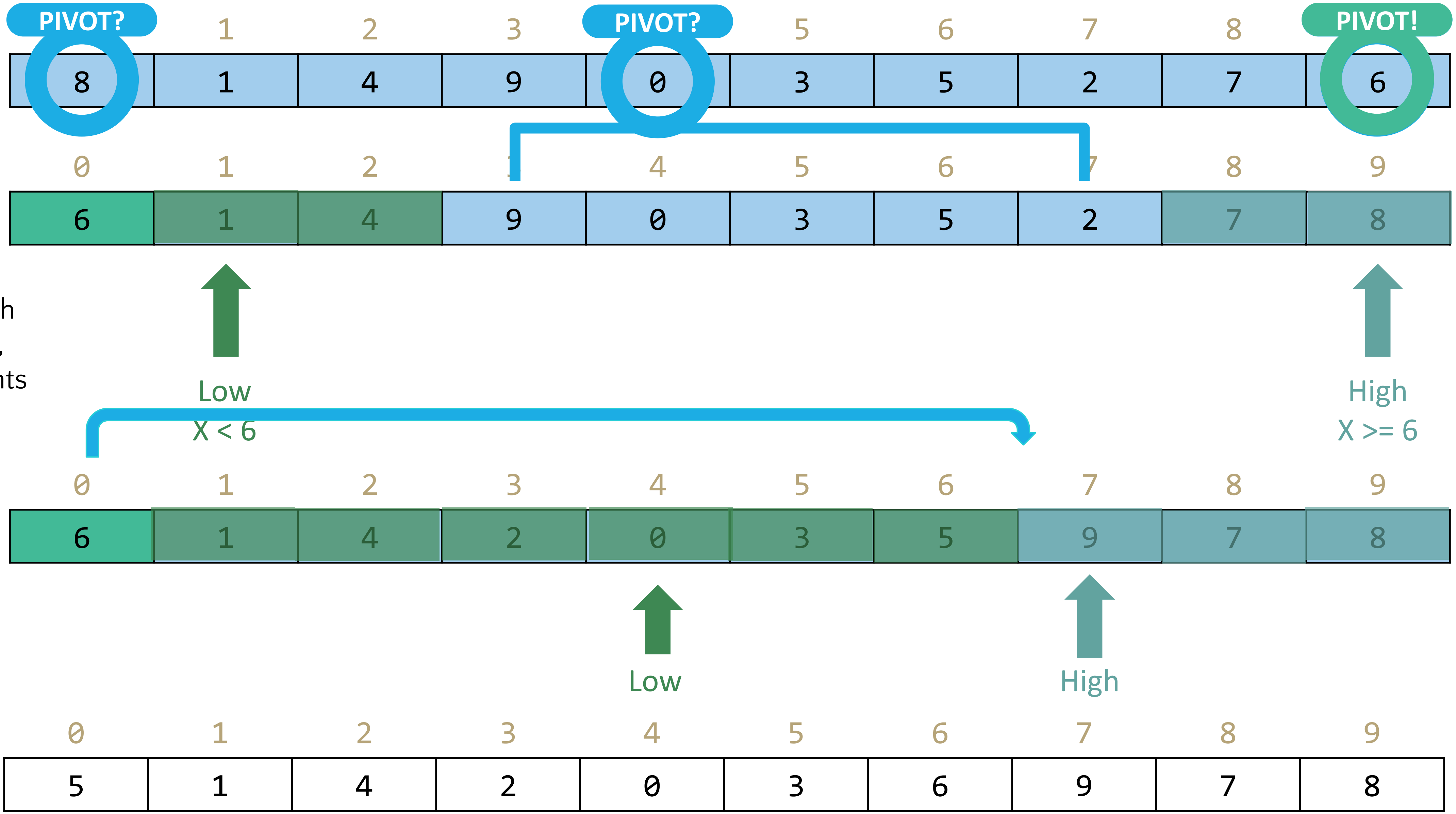
Divide

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!

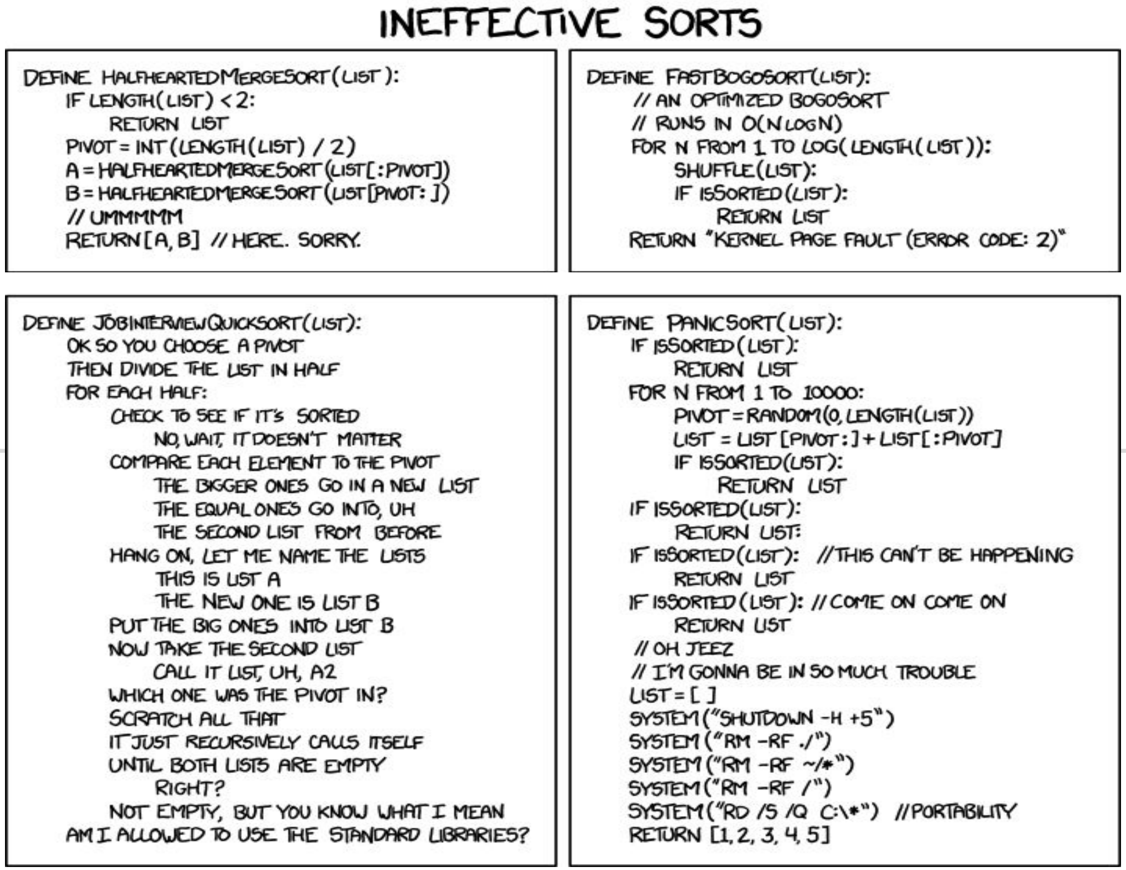


Heap Sort

Bucket Sort

Radix Sort

Sorting Summary



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

Principle 3

Selection sort:
After *k* iterations of the loop, the *k* smallest elements of the array are (sorted) in indices 0, … , k-1
Runs in $\Theta(n^2)$ time no matter what

Using data structures
- Speed up our existing ideas

If only we had a data structure that was good at getting the smallest item remaining in our dataset…
- We do!

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

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

Heap

Sorted Items

Current Item

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

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

Heap Sort

Bucket Sort

Radix Sort

Sorting Summary

Bucket Sort (aka Bin Sort)

- If all values are ints known to be in the range of 1 – K
- Create array of size K and put each element in its proper bucket ("scatter")
    - If elements are only ints simply store count of ints in each bucket
- Output results via linear pass through array of buckets ("gather")

[5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5]

O(n)

1	3
2	1
3	2
4	4
5	3

O(K + n)

[1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 5]

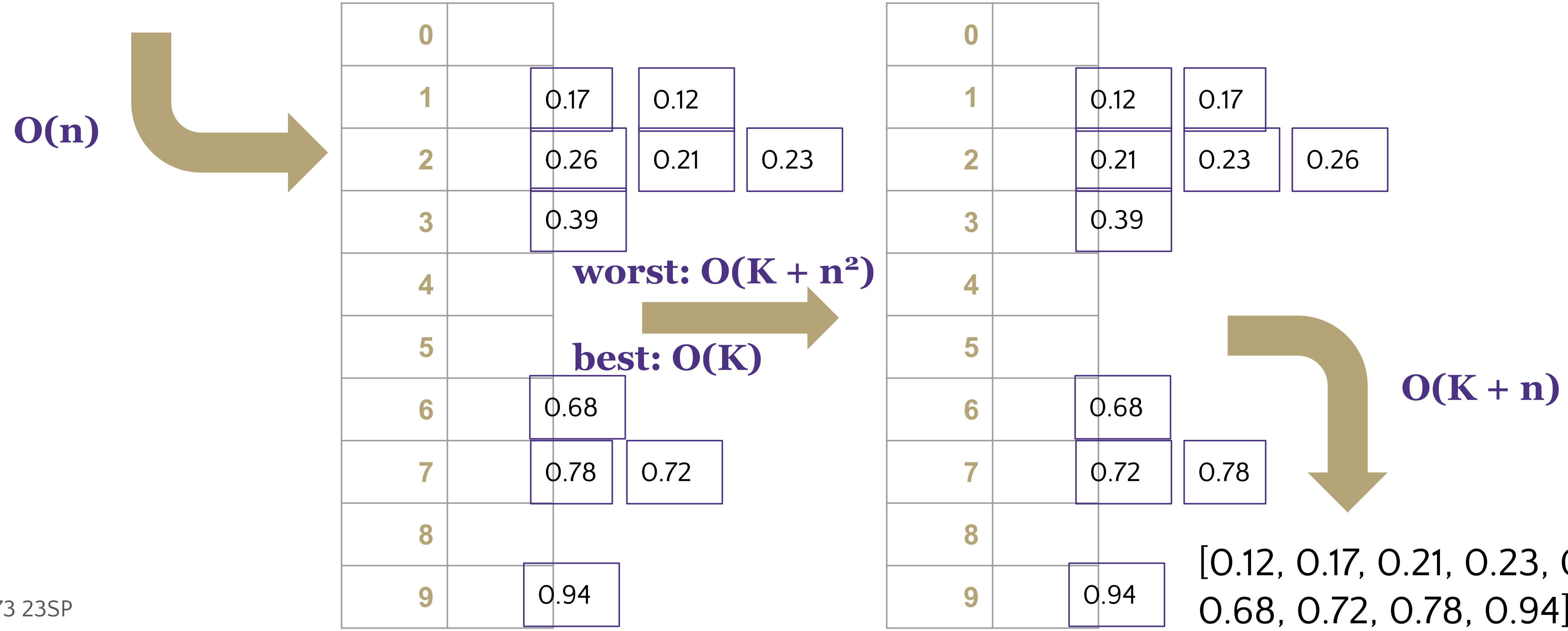
Total Runtime: O(K + n)

[Bucket Sort Example Video]

Bucket Sort with Data

- Instead of using int counts, make buckets of array of lists
- put items into bucket, use **insertion sort** to sort individual buckets

[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]



[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]

Bucket Sort

```
function bucketSort(array, k) is
    buckets ← new array of k empty lists
    M ← 1 + the maximum key value in the array
    for i = 0 to length(array) do
        insert array[i] into buckets[floor(k × array[i] / M)]
    for i = 0 to k do
        nextSort(buckets[i])
    return the concatenation of buckets[0], ...., buckets[k]
```

Worst case runtime?	O(K + n) for ints O(K + n^2) for data if insertion sort is used
Best case runtime?	O(n)
In-practice runtime?	O(n) if K ≅ n, always for ints, and if values are evenly distributed for data
Stable?	Can be because insertion sort
In-place?	No
Useful for:	When range, K, is smaller or not much larger than n (not many duplicates) Not good when K >> N, wasted space

Heap Sort

Bucket Sort

Radix Sort

Sorting Summary

Moving away from comparison sorts

So far we've learned about comparison sorts

- work on any comparable object
- have a best case lower bound of $\Omega(n\log n)$

This is because to sort using comparisons requires all elements to be compared against one another

- n runtime to process all values into some ordered structure (tree)
- logn runtime to remove items from structure in sorted order

What if we didn't need to compare each element, what if we built a sort based on inherent knowledge about the ordering of specific data types ie numbers

Specialized Sorts (“Niche Sorts”)

Sorting algorithms that only work on data types with ordering already known to computer logic: numbers

- Bucket Sort for ints
- Radix Sort

Radix Sort

- Radix = "the base of a number system"
  - We will use "10" as we are comfortable with 10 based systems
  - Could use any value, such as 128 for ASCII strings
- Idea
  - Bucket sort on one digit at a time
    - Only works on sequences of countable data: ints, doubles, stings
  - Number of buckets = radix
  - Start with least significant digit, do one pass of bucket sort per digit
- Fun fact: invented in 1890 as part of US census

Input: [170, 45, 75, 90, 802, 24, 2, 66]

ones: [170, 90, 802, 2, 24, 45, 75, 66]

tens: [802, 2, 24, 45, 66, 170, 75, 90]

hundreds: [2, 24, 45, 66, 75, 90, 170, 802]

Example Walk Through Video

Radix Sort

[478, 537, 9, 721, 3, 38, 143, 67]

O(n)

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

O(n)

[721, 3, 143, 537, 67, 478, 38, 9]

O(n)

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

O(n)

[3, 9, 721, 537, 38, 143, 67, 478]

O(n)

0	**00**3, **00**9, **0**38, **0**67
1	143
2	
3	
4	478
5	
6	537
7	721
8	
9	

O(n)

[3, 9, 38, 67, 143, 478, 537, 721]

Radix Sort

Worst case runtime?	O(n)
Best case runtime?	O(n)
In-practice runtime?	O(n)
Stable?	Yes
In-place?	No
Useful for:	Sorting ints

Heap Sort

Bucket Sort

Radix Sort

Sorting Summary

Sorting: Summary

	Best-Case	Worst-Case	Space	Stable
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	O(n)	$O(n^2)$	O(1)	Yes
Heap Sort	O(nlogn)	O(nlogn)	O(n)	No
In-Place Heap Sort	O(nlogn)	O(nlogn)	O(1)	No
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn) O(n)* optimized	Yes
Quick Sort	O(nlogn)	$O(n^2)$	O(n)	No
In-place Quick Sort	O(nlogn)	$O(n^2)$	O(1)	No
Bucket Sort	O(n)	$O(n^2)$	O(K+n)	Yes
Radix	O(n)	O(n)	O(n)	Yes

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

What Else is There?

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!

Counting Sort (Wikipedia)

External Sorting Algorithms (Wikipedia) – For big data™

Questions?

That’s all!