Lecture 21: Introduction to Sorting

CSE 373: Data Structures and Algorithms

Warm Up

Slido Event #2559450
https://app.sli.do/event/szeVCNPUW5uXG81RSji8Lk

If I handed you a stack of papers and asked you to sort them by author name alphabetically, how would you do it?

Selection Sort – I would flip through the stack from front to back looking for the first name, then pull it to the front. Then I would flip through again looking for the second name and put it behind the first and so on until all were sorted

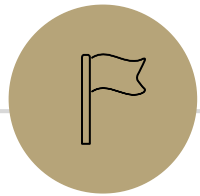
Insertion Sort – I look at the first two papers and put them in sorted order, then look at the third and put it in sorted order with the previous two and continue until the whole stack is in sorted order

Merge Sort – I would spread the papers out on the ground and break them into subsections, sort the subsections section by section then put them all back together

Bucket Sort – I would start by putting the papers into groups based on the first letter of the author's name until I had 26 piles, then I would sort within those piles and put them all back together

Announcements

- EX5 due today
- EX6 releases today
- Final exam on Friday May 26th



Intro to Sorting

Selection Sort

Insertion Sort

Merge Sort

Quick Sort

Where are we?

This course is “data structures and algorithms”

Data structures

- Organize our data so we can process it effectively

Algorithms

- Actually process our data!

We’re going to start focusing on algorithms

We’ll start with sorting

- A very common, generally-useful preprocessing step
- And a convenient way to discuss a few different ideas for designing algorithms.

Types of Sorts

Comparison Sorts

Compare two elements at a time

General sort, works for most types of elements

What does this mean?
compareTo() works for your elements

- And for our running times to be correct, compareTo() must run in O(1) time

Niche Sorts aka "linear sorts"

Leverages specific properties about the items in the list to achieve faster runtimes

niche sorts typically run O(n) time

For example, we're sorting small integers, or short strings

In this class we'll focus on comparison sorts

Sorting Goals

In-place sort

A sorting algorithm is in-place if it allocates O(1) extra memory

Modifies input array (can't copy data into new array)

Useful to minimize memory usage

Stable sort

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort

Why do we care?

- "data exploration" Client code will want to sort by multiple features and "break ties" with secondary features

[**(8, "fox")**, (9, "dog"), (4, "wolf"), **(8, "cow")**]

[(4, "wolf"), **(8, "fox")**, **(8, "cow")**, (9, "dog")] Stable

[(4, "wolf"), **(8, "cow")**, **(8, "fox")**, (9, "dog")] Unstable

Speed

Of course, we want our algorithms to be fast.

Sorting is so common, that we often start caring about constant factors.

SO MANY SORTS!

Quicksort, Merge sort, in-place merge sort, heap sort, insertion sort, intro sort, selection sort, timsort, cubesort, shell sort, bubble sort, binary tree sort, cycle sort, library sort, patience sorting, smoothsort, strand sort, tournament sort, cocktail sort, comb sort, gnome sort, block sort, stackoverflow sort, odd-even sort, pigeonhole sort, bucket sort, counting sort, radix sort, spreadsort, burstsort, flashsort, postman sort, bead sort, simple pancake sort, spaghetti sort, sorting network, bitonic sort, bogosort, stooge sort, insertion sort, slow sort, rainbow sort…

Goals

Algorithm Design (like writing invariants) is more art than science.

We'll do a little bit of designing our own algorithms

- Take CSE 417 (usually runs in Winter) for more

Mostly we'll understand how existing algorithms work

Understand their pros and cons

- Design decisions!

Practice how to apply those algorithms to solve problems

Algorithm Design Patterns

Algorithms don't just come out of thin air.

There are common patterns we use to design new algorithms.

Many of them are applicable to sorting (we'll see more patterns later in the quarter)

Invariants/Iterative improvement

- Step-by-step make one more part of the input your desired output.

Divide and conquer

- Split your input
- Solve each part (recursively)
- Combine solved parts into a single

Using data structures

- Speed up our existing ideas

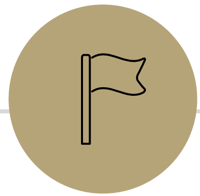
Principle 1

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.



Intro to Sorting

Selection Sort

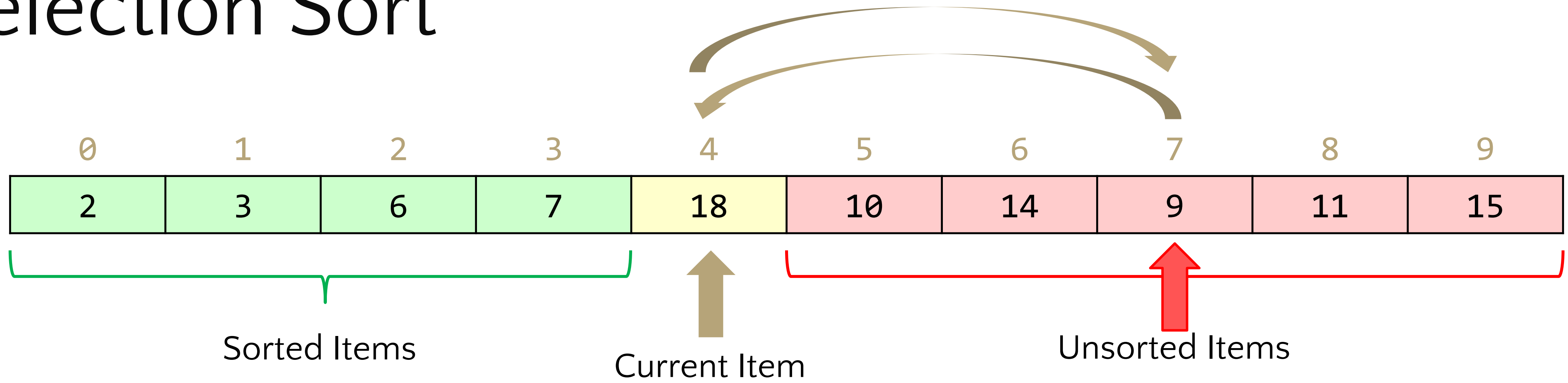
Insertion Sort

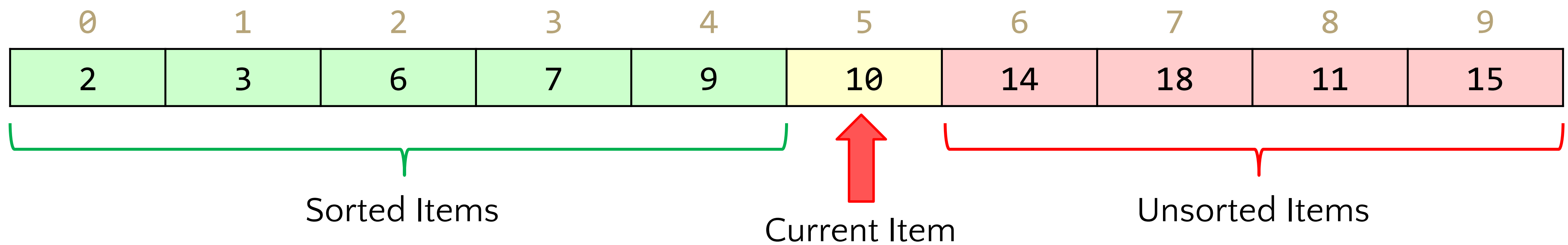
Merge Sort

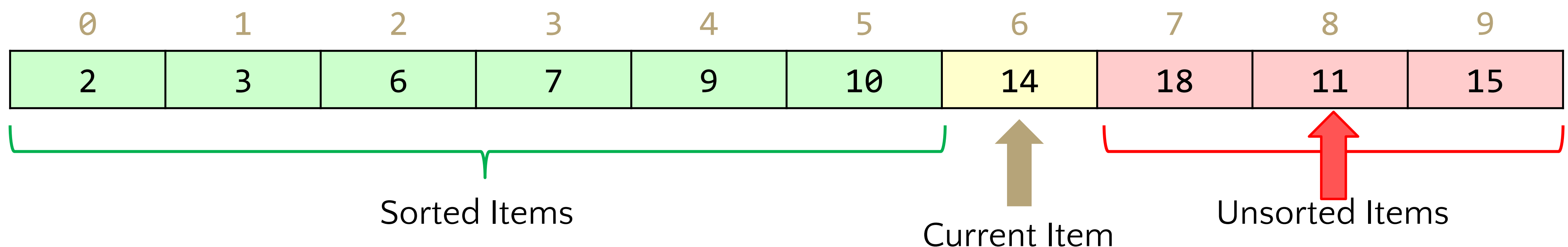
Quick Sort

https://www.youtube.com/watch?v=Ns4TPTC8whw

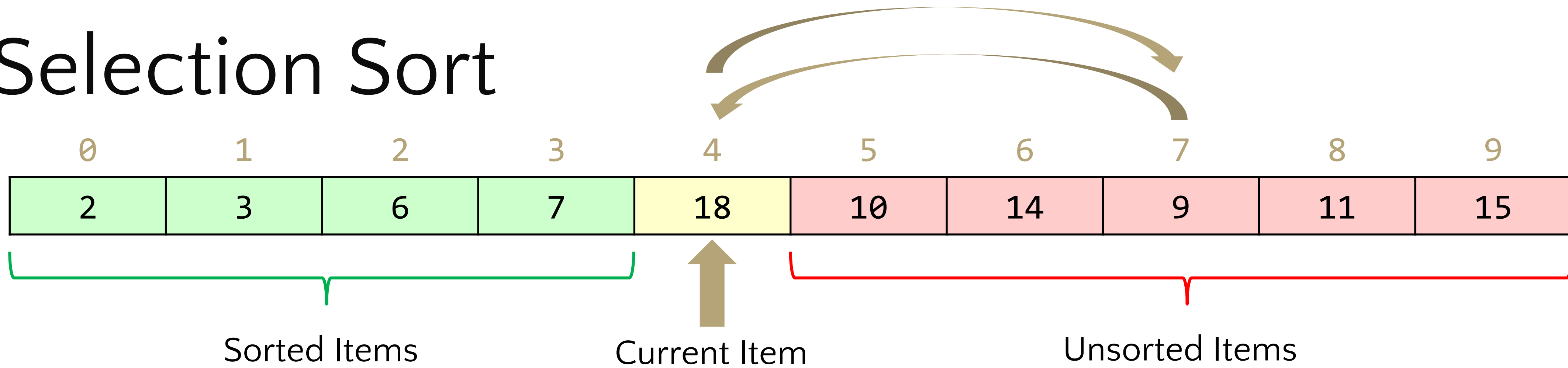
Selection Sort







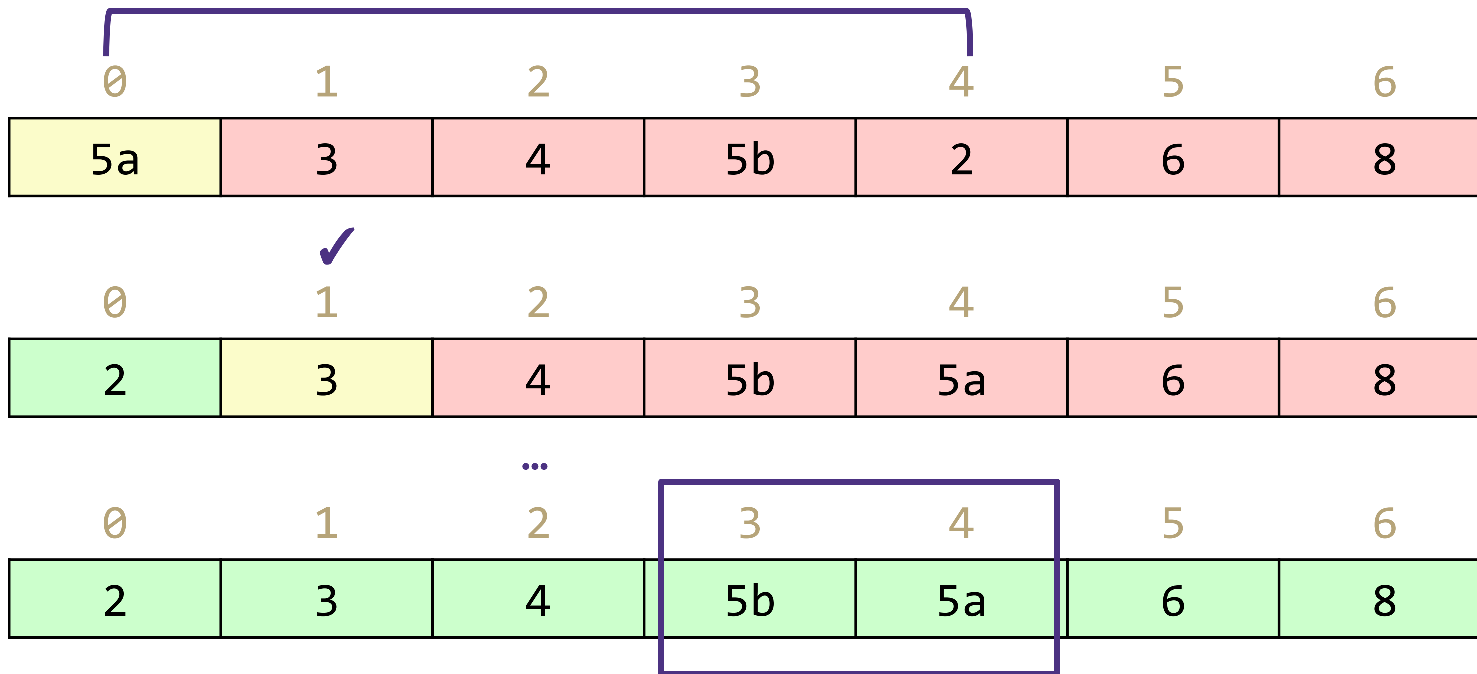
Selection Sort



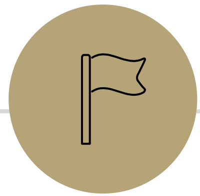

```
public void selectionSort(collection) {
   for (entire list)
      int newIndex = findNextMin(currentItem);
      swap(newIndex, currentItem);
}
public int findNextMin(currentItem) {
   min = currentItem
   for (unsorted list)
      if (item < min)
         min = currentItem
   return min
}
public int swap(newIndex, currentItem) {
   temp = currentItem
   currentItem = newIndex
   newIndex = currentItem
}
```

Worst case runtime?	$\Theta(n^2)$
Best case runtime?	$\Theta(n^2)$
In-practice runtime?	$\Theta(n^2)$
Stable?	No
In-place?	Yes
Useful for:	Top K sort without needing extra space

Selection Sort Stability



*Swapping non-adjacent items can result in instability of sorting algorithms



Intro to Sorting

Selection Sort

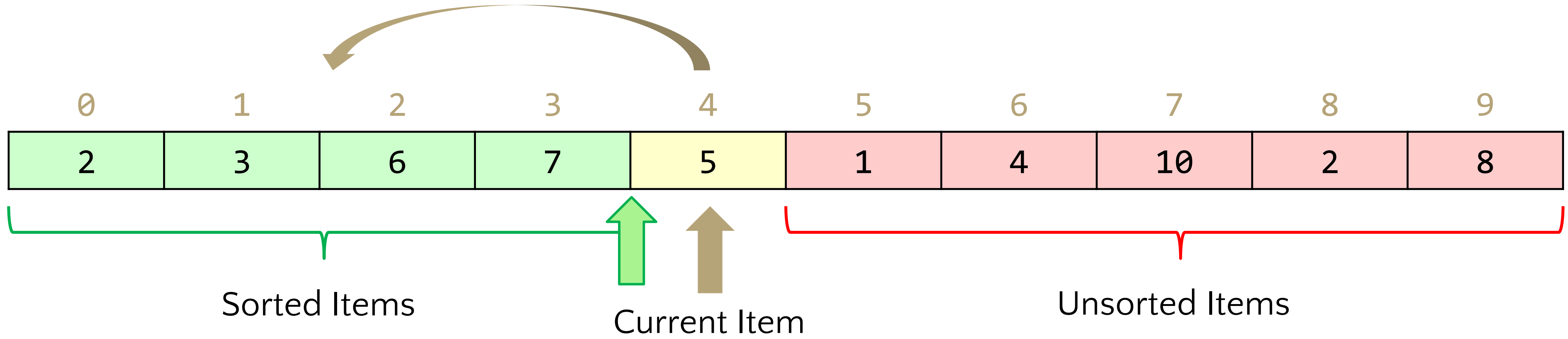
Insertion Sort

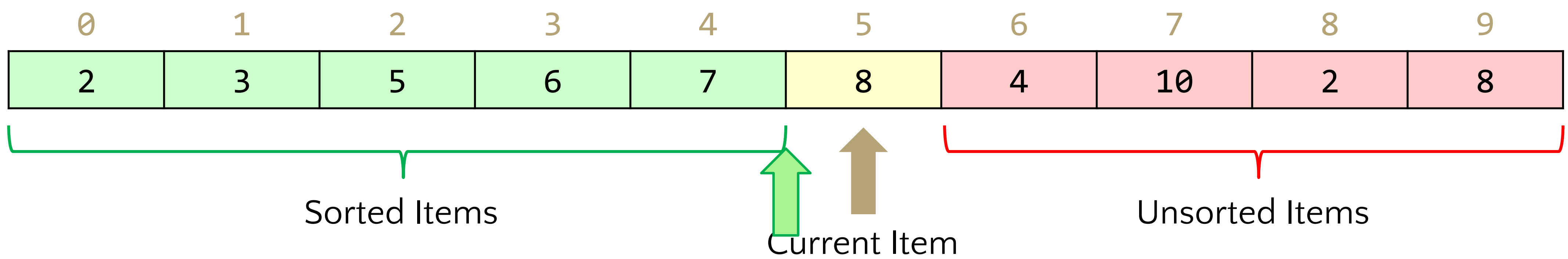
Merge Sort

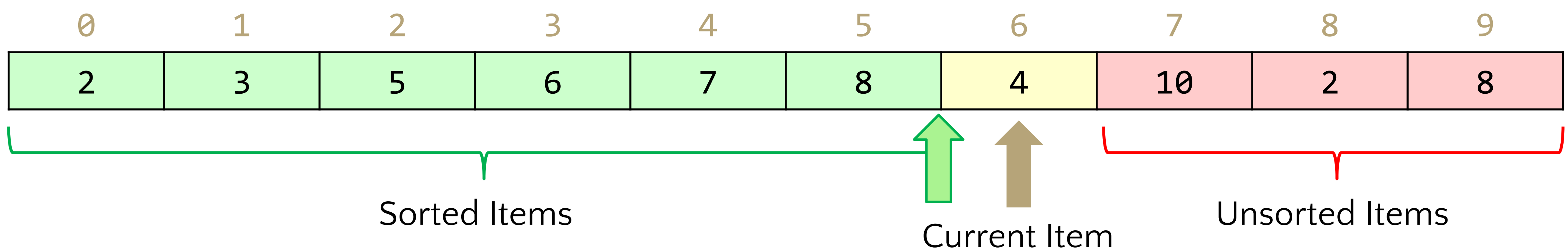
Quick Sort

Insertion Sort

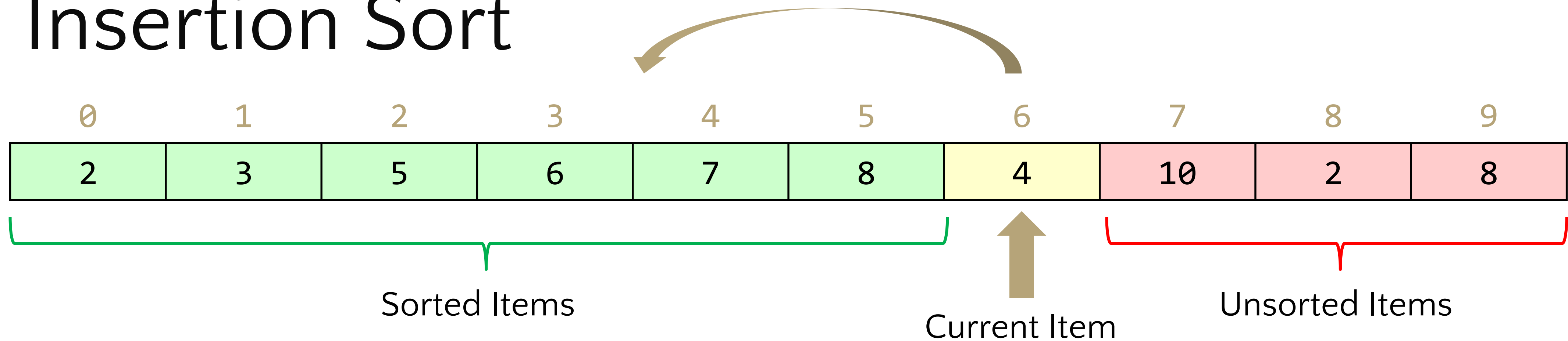
https://www.youtube.com/watch?v=ROalU379l3U







Insertion Sort




```
public void insertionSort(collection) {
   for (entire list)
      if(currentItem is smaller than largestSorted)
         int newIndex = findSpot(currentItem);
         shift(newIndex, currentItem);
}
public int findSpot(currentItem) {
   for (sorted list going backwards)
      if (spot found) return
}
public void shift(newIndex, currentItem) {
   for (i = currentItem > newIndex)
      item[i+1] = item[i]
   item[newIndex] = currentItem
}
```

Worst case runtime? $\Theta(n^2)$

Best case runtime? $\Theta(n)$

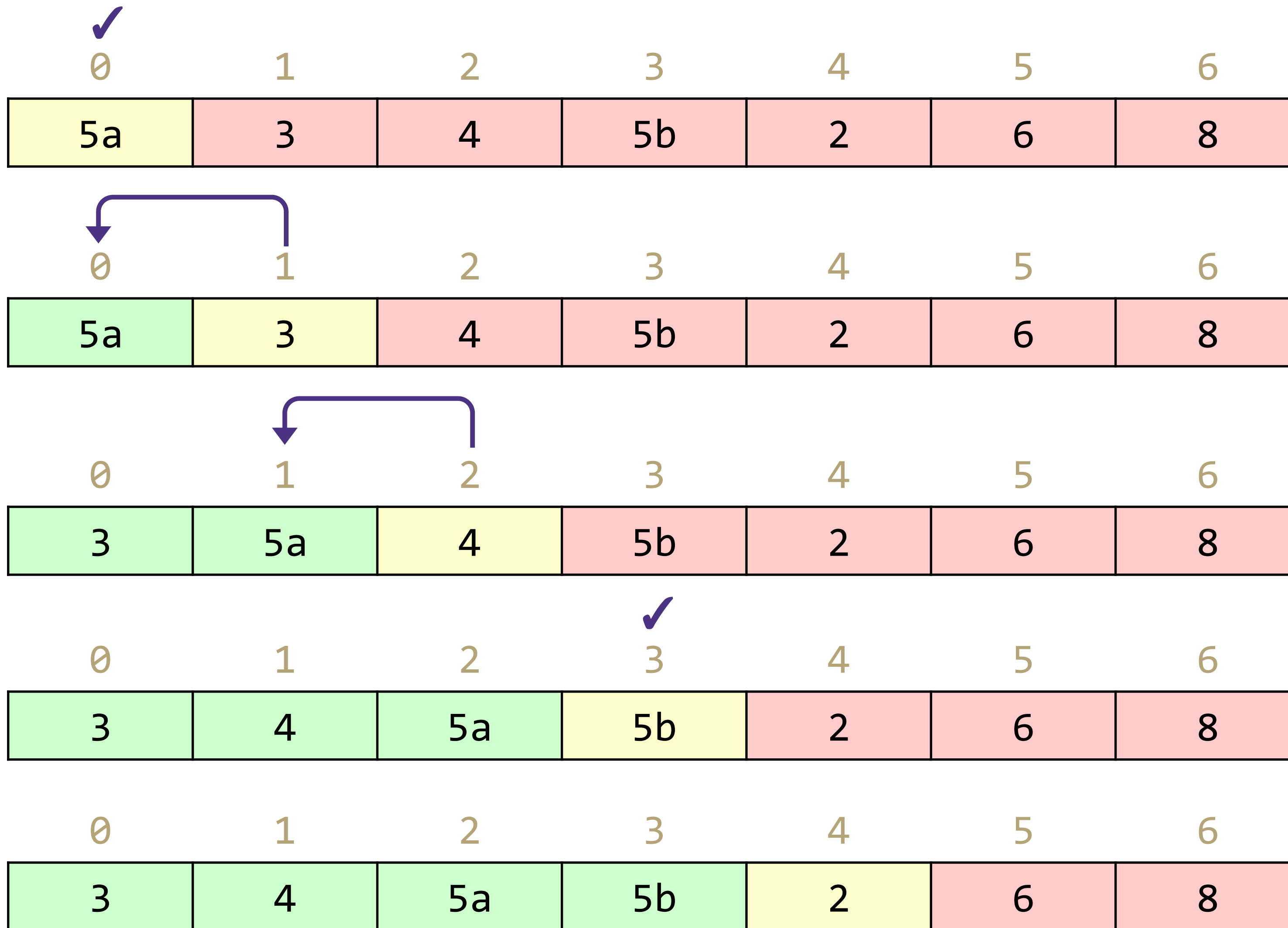
In-practice runtime? $\Theta(n^2)$

Stable? Yes

In-place? Yes

Useful for: Mostly sorted collections of primitives

Insertion Sort Stability



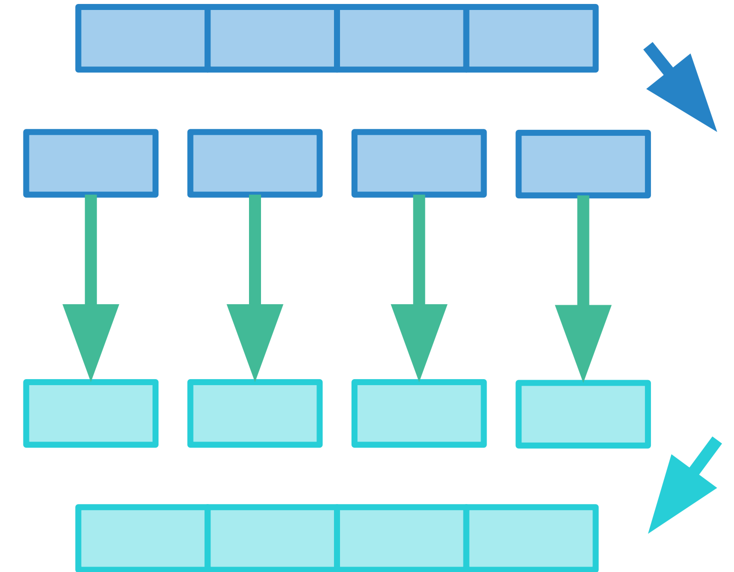
Insertion sort is stable

- All swaps happen between adjacent items to get current item into correct relative position within sorted portion of array
- Duplicates will always be compared against one another in their original orientation, thus it can be maintained with proper if logic

Principle 2: 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

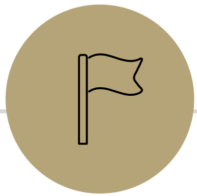


```
divideAndConquer(input) {
  if (small enough to solve):
    conquer, solve, return results
  else:
    divide input into a smaller pieces
    recurse on smaller pieces
    combine results and return
}
```

Intro to Sorting

Selection Sort

Insertion Sort



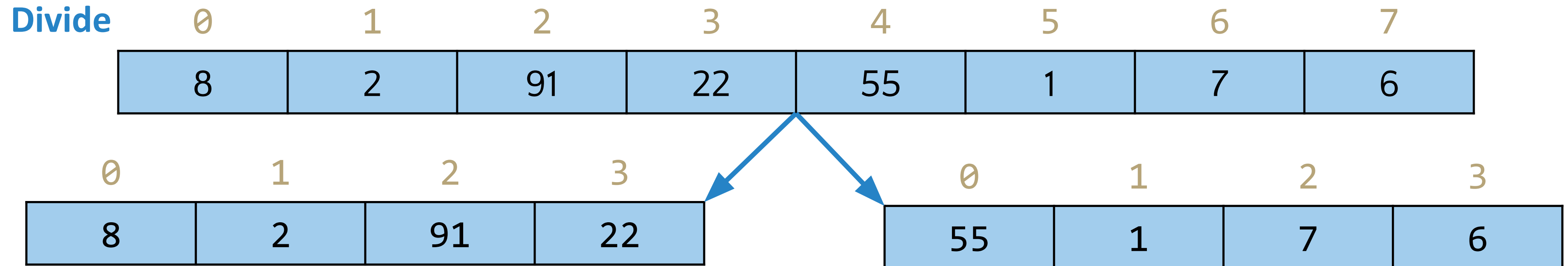
Merge Sort

Quick Sort

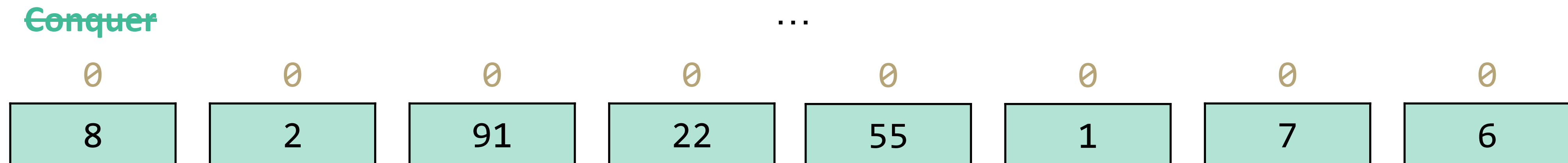
Merge Sort

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

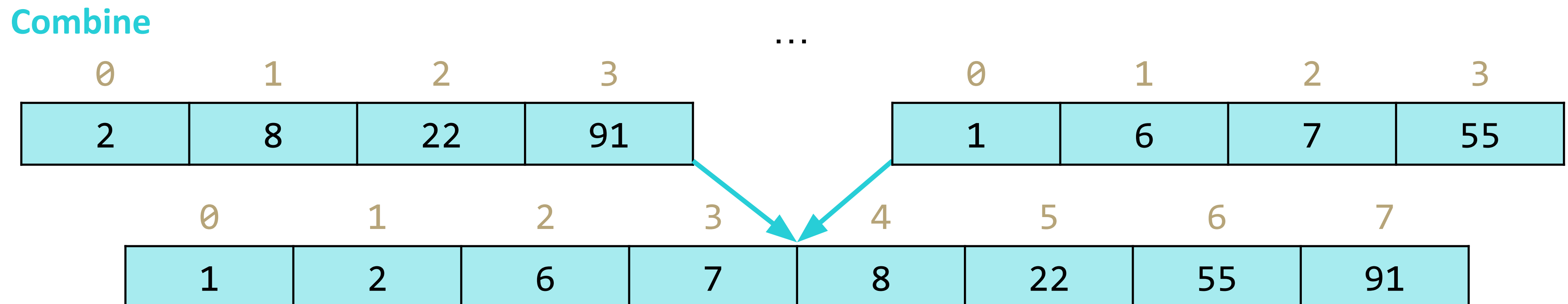
Simply divide in half each time



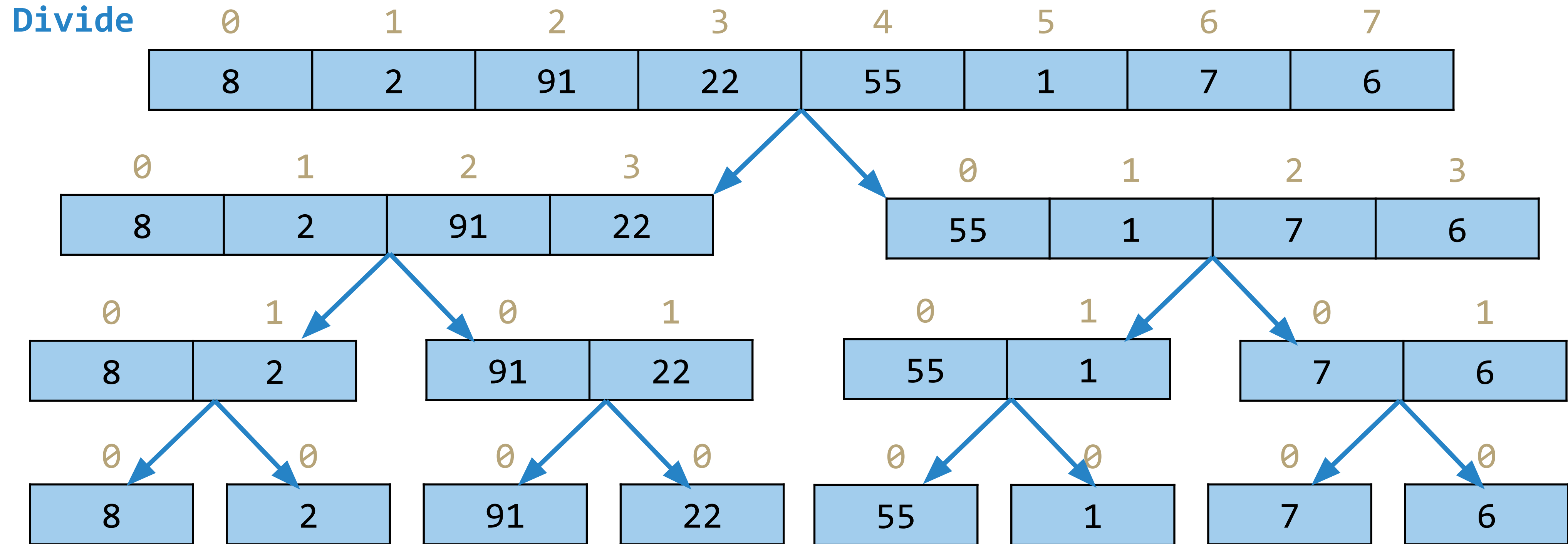
No extra conquer work needed!



The actual sorting happens here!



Merge Sort: Divide Step

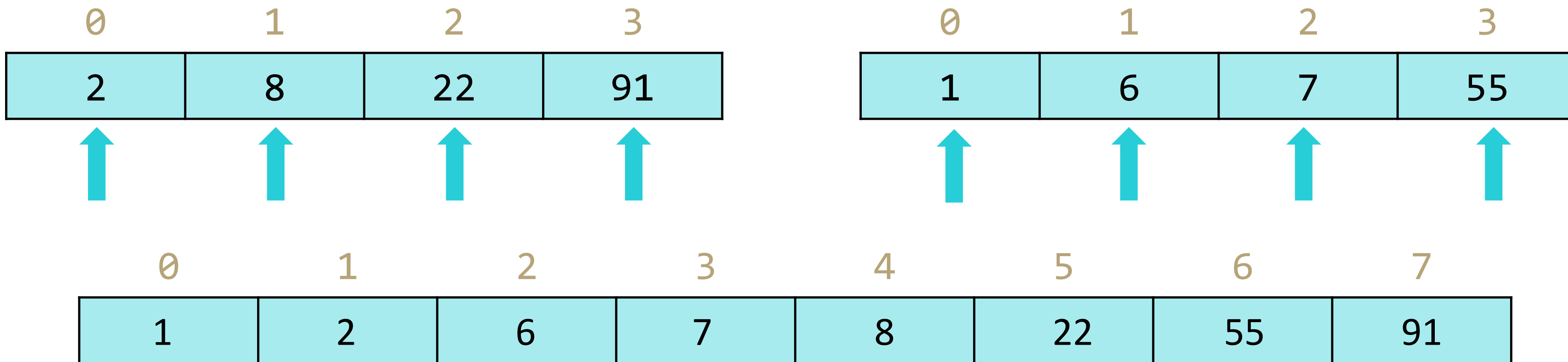


Recursive Case: split the array in half and recurse on both halves

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



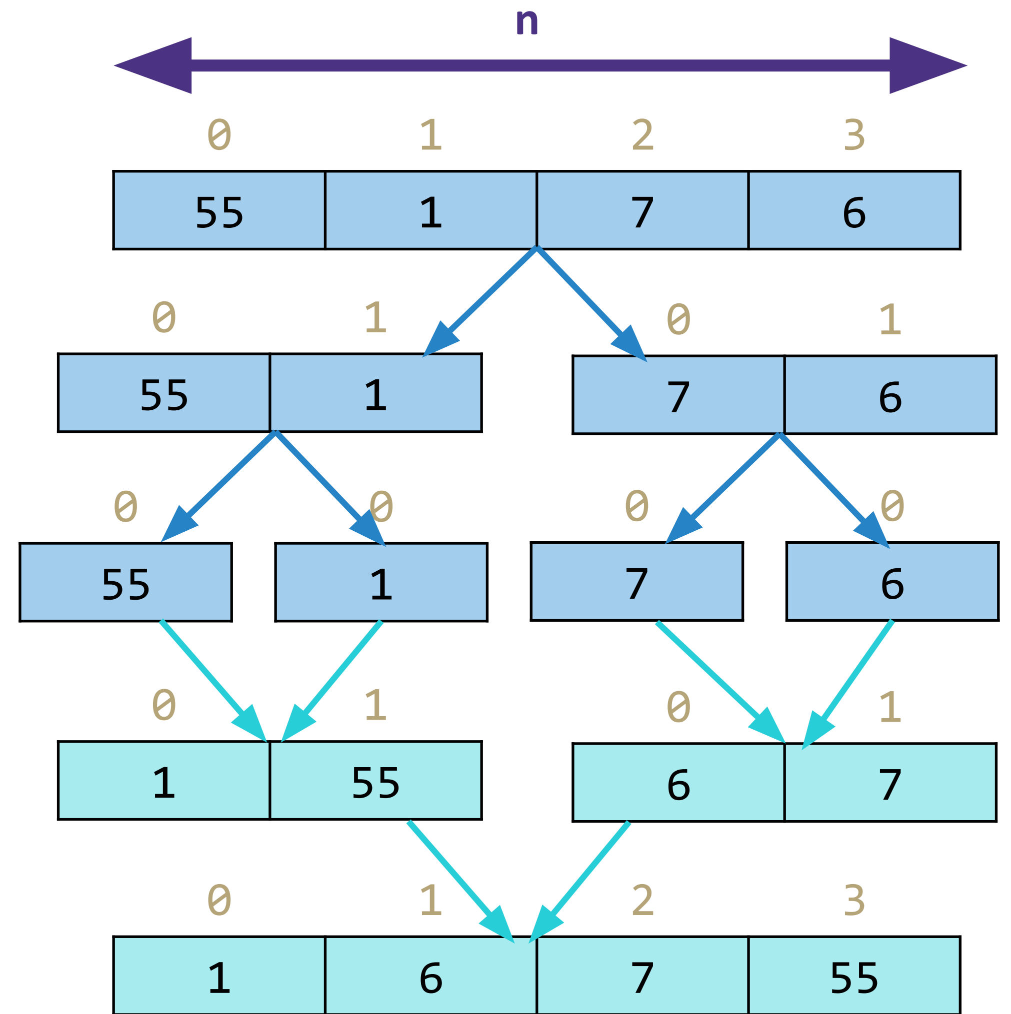
Combining two *sorted* arrays:

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

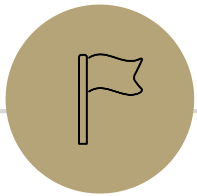
In-place? No

2 Constant size Input

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

Useful for: Predictable sort times regardless of sorted nature (This is what Java uses for Objects, kinda)

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.

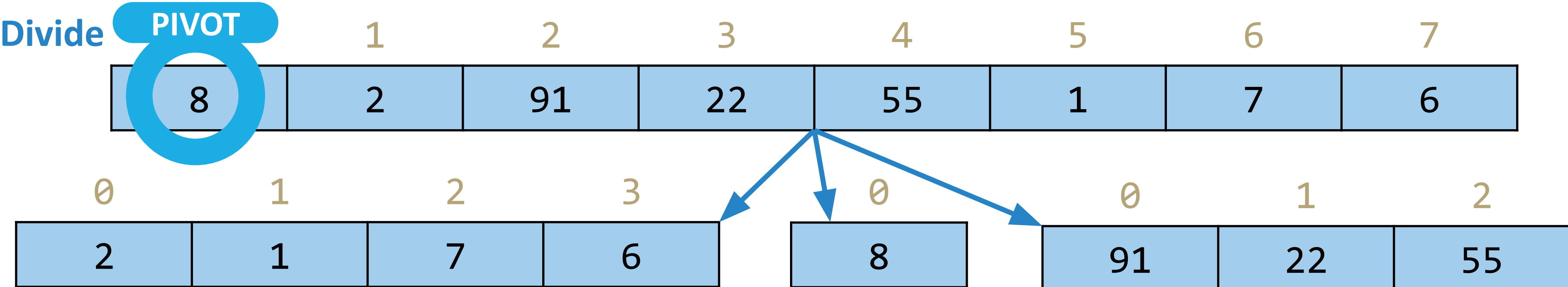
Quicksort:

- 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!

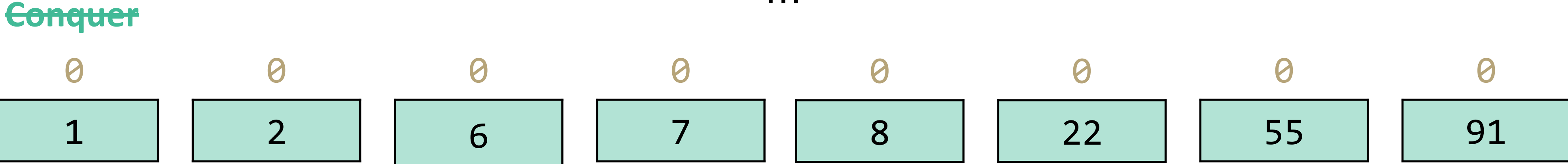
Quick Sort (v1)

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

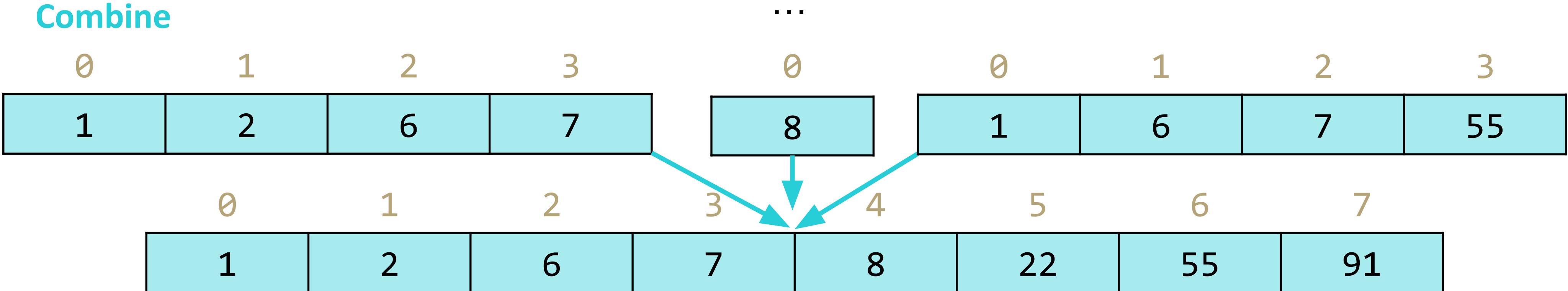
Choose a "pivot" element, partition array relative to it!



Again, no extra conquer step needed!



Simply concatenate the now-sorted arrays!



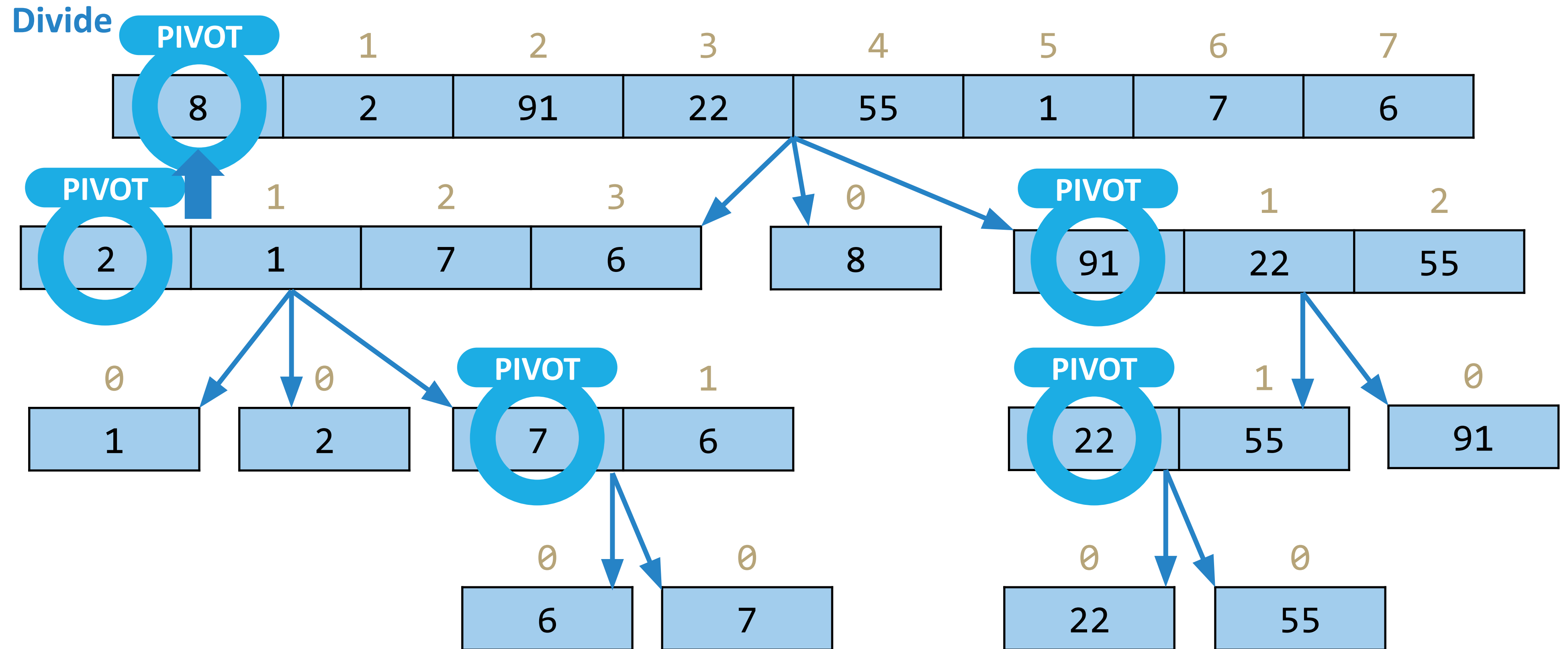
Quick Sort (v1): Divide Step

Recursive Case:

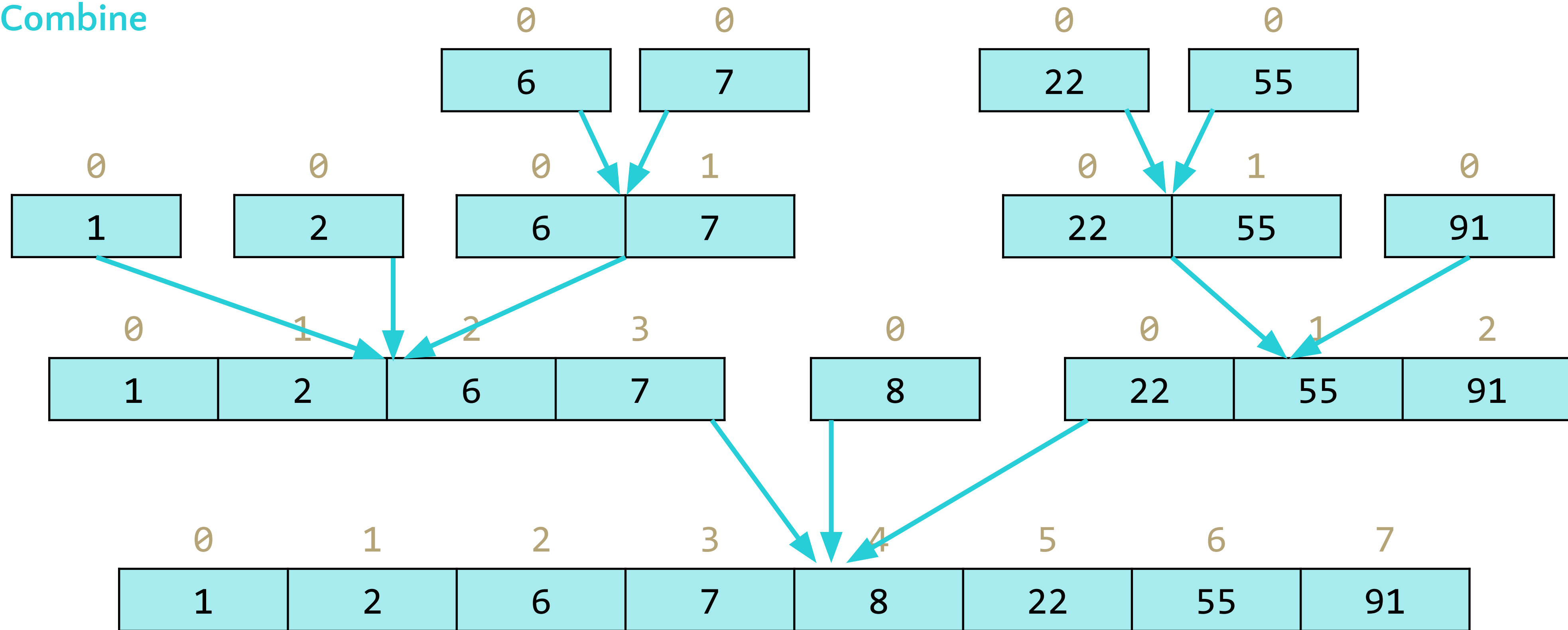
- Choose a “pivot” element
- Partition: linear scan through array, add smaller elements to one array and larger elements to another
- Recursively partition

Base Case:

- When array hits size 1, stop dividing



Quick Sort (v1): Combine Step

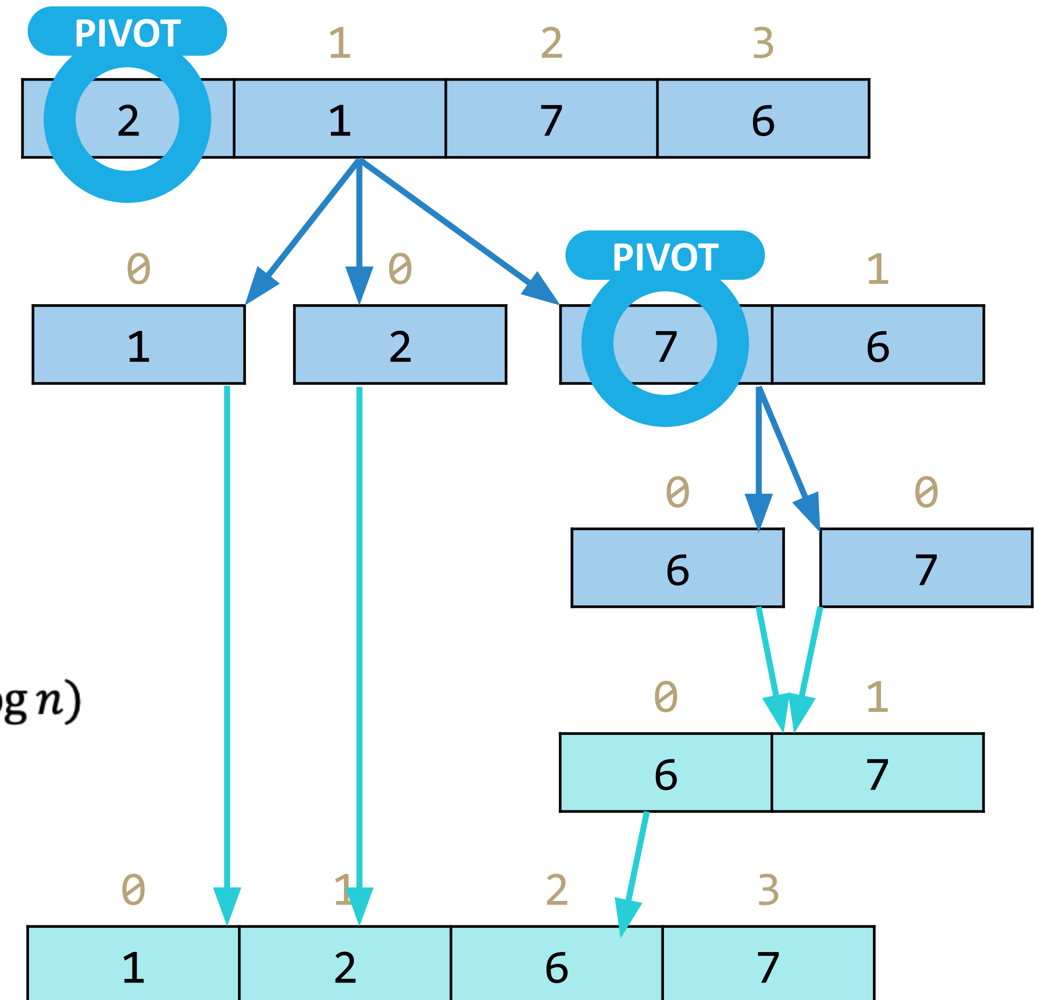


Simply concatenate the arrays that were created earlier! Partition step already left them in order ☺️

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!

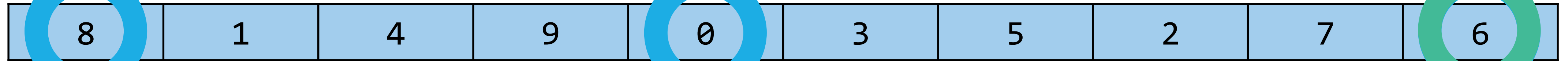
Pick a random element

- Get $O(n \log n)$ runtime with probability at least $\mathbf{1-1/n^2}$
- No simple worst-case input (e.g. sorted, reverse sorted)

Quick Sort (v2: In-Place)

Divide

Select a pivot



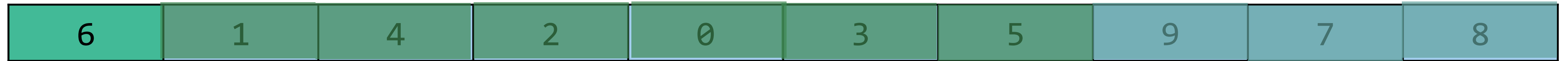
Move pivot out of the way

Bring low and high pointers together, swapping elements if needed

Low
X < 6

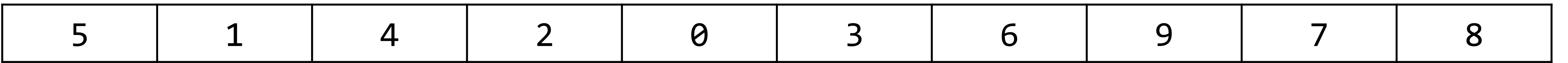
High
X >= 6

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



Low

High



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?

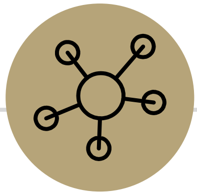
Best case runtime?

In-practice runtime?

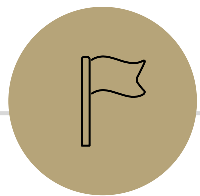
Stable? No

In-place? Yes

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



Questions?



That's all!