Lecture 14: Sorting Algorithms

CSE 373: Data Structures and Algorithms

Administrivia

Robbie's office hours today are cancelled (we have to grade your midterms)

Project 3 Partner form is due tonight (P3 will be released Wednesday)

Project 2 due Wednesday

Exercise 3 due Friday

Sorting

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

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.

Using data structures
- Speed up our existing ideas

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

Principle 1

Invariants/Iterative improvement

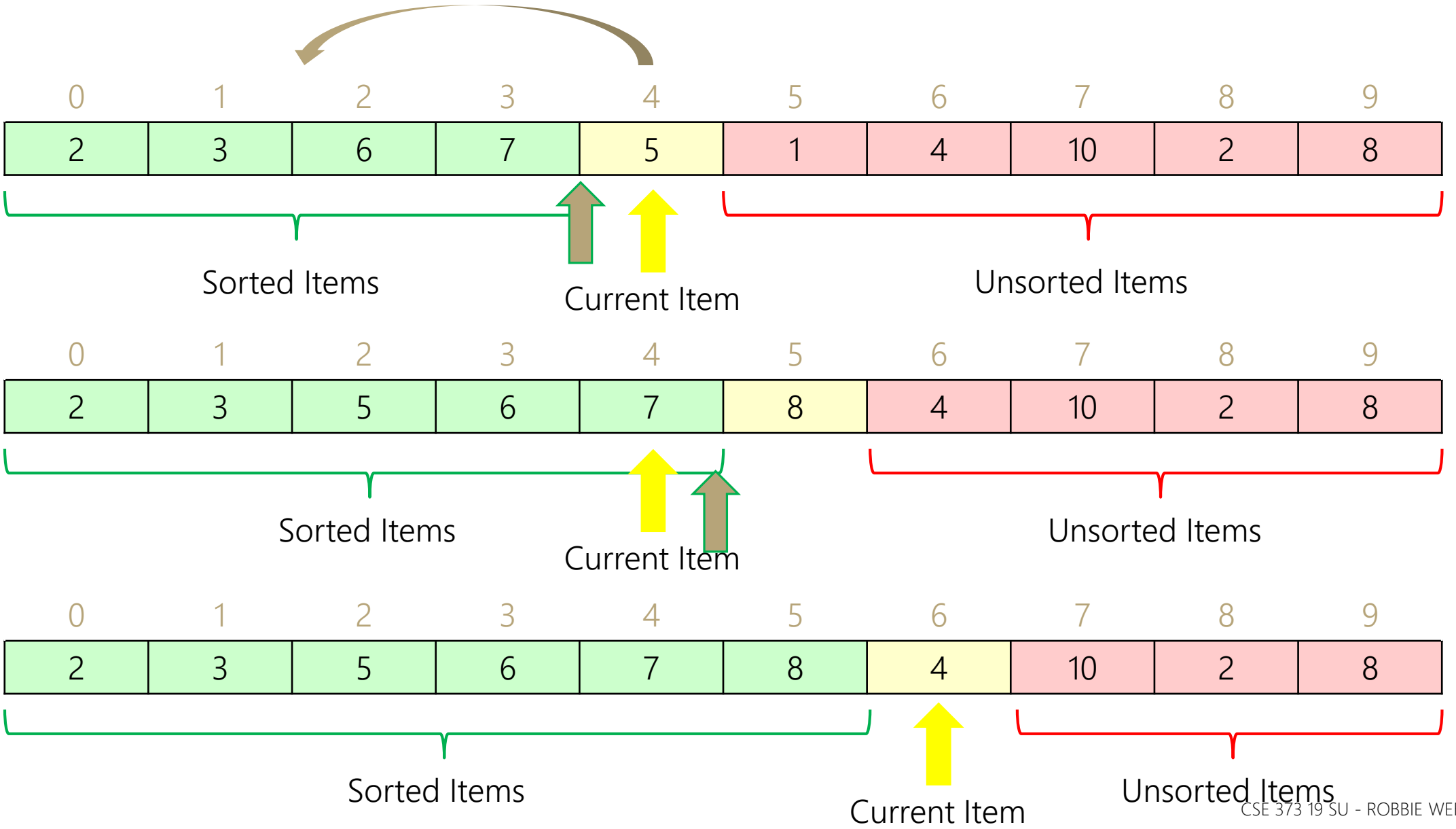
- Step-by-step make one more part of the input your desired output.

We'll write iterative algorithms to satisfy the following invariant:

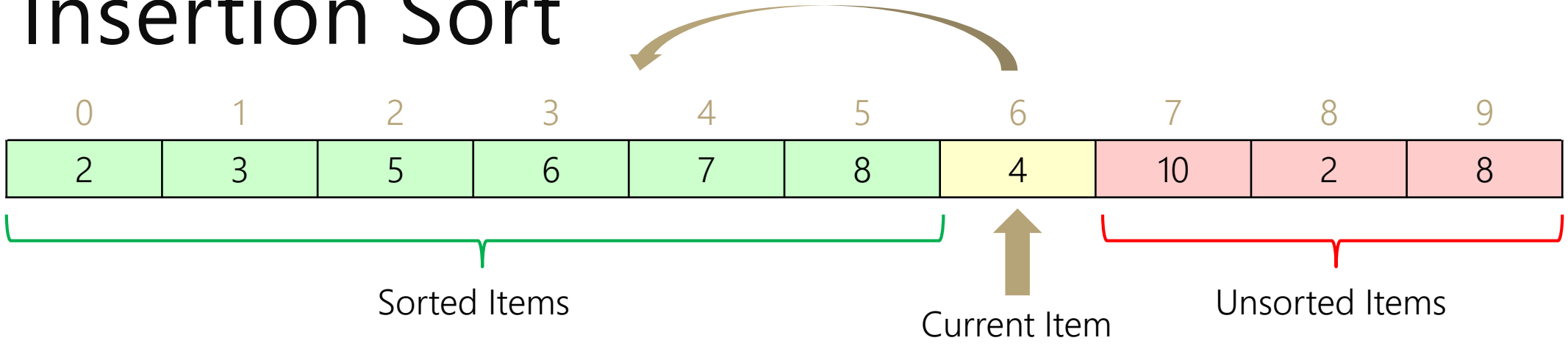
After k iterations of the loop, the first k elements of the array will be sorted.

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?

Best case runtime?

In-practice runtime? $O(n^2)$

Stable? Yes

In-place? Yes

pollEV.com/cse373su19

What are the best case and worst case times?

Satisfying the invariant

We said this would be our invariant:

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

But that wasn't a full description of what happens

Insertion sort:

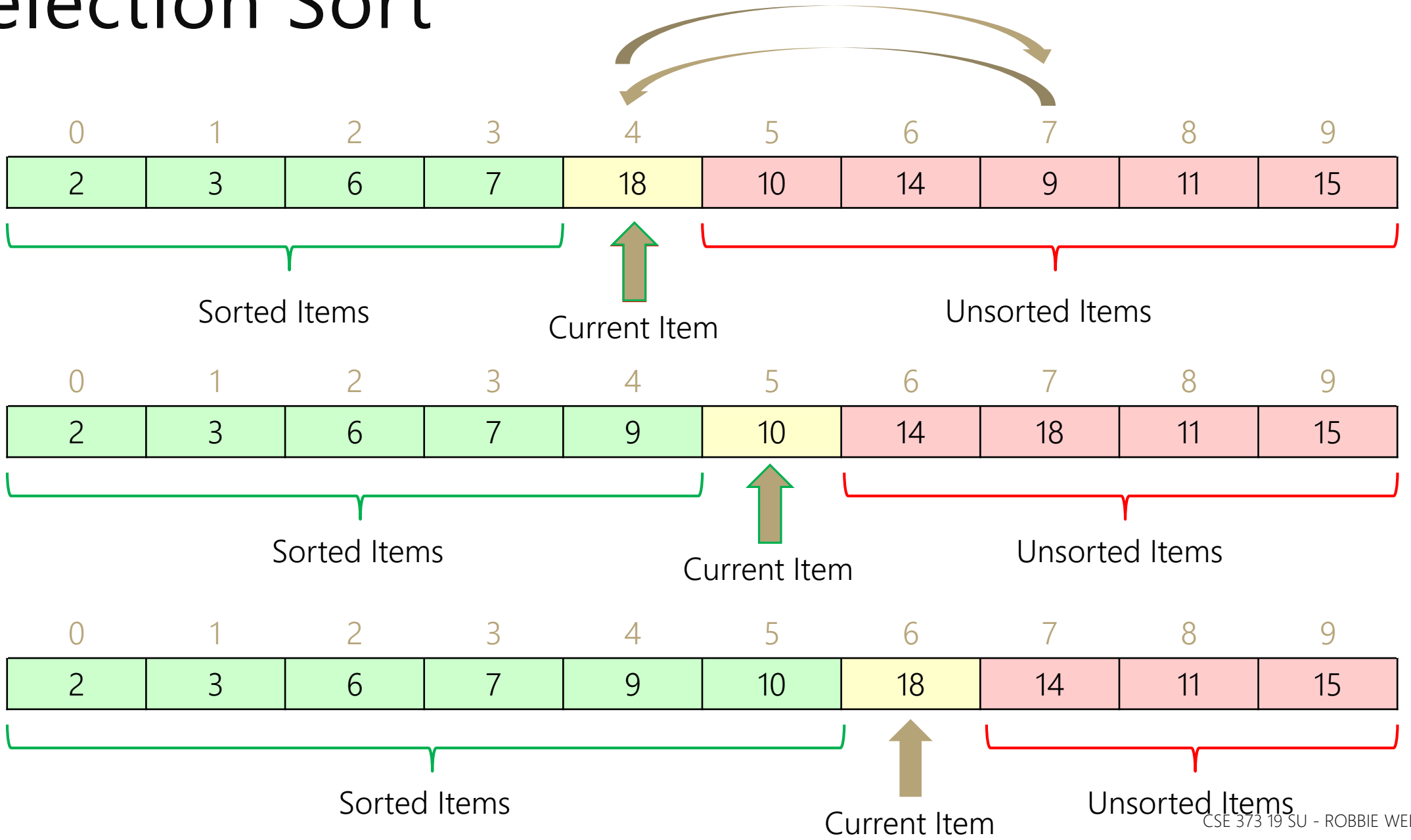
After k iterations of the loop, the elements that started in indices $0, \ldots, k-1$ are now sorted

Selection sort:

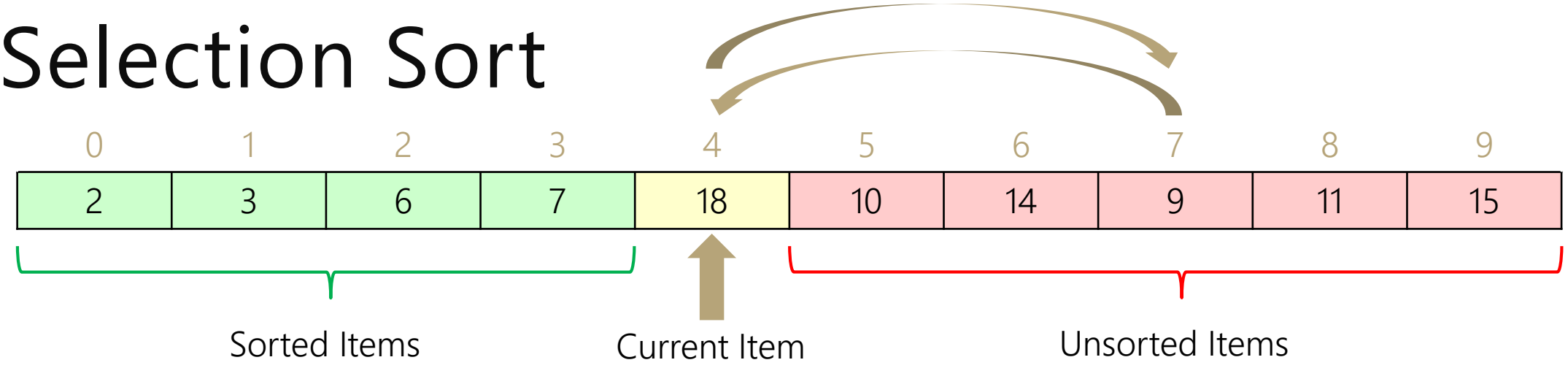
After k iterations of the loop, the k smallest elements of the array are (sorted) in indices $0, \ldots, k-1$

Selection Sort

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



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? $O(n^2)$

Best case runtime? $O(n^2)$

In-practice runtime? $O(n^2)$

Stable? No

In-place? Yes

Principle 2

Selection sort:

After k iterations of the loop, the k smallest elements of the array are (sorted) in indices $0, \ldots, k-1$

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!

Heap Sort

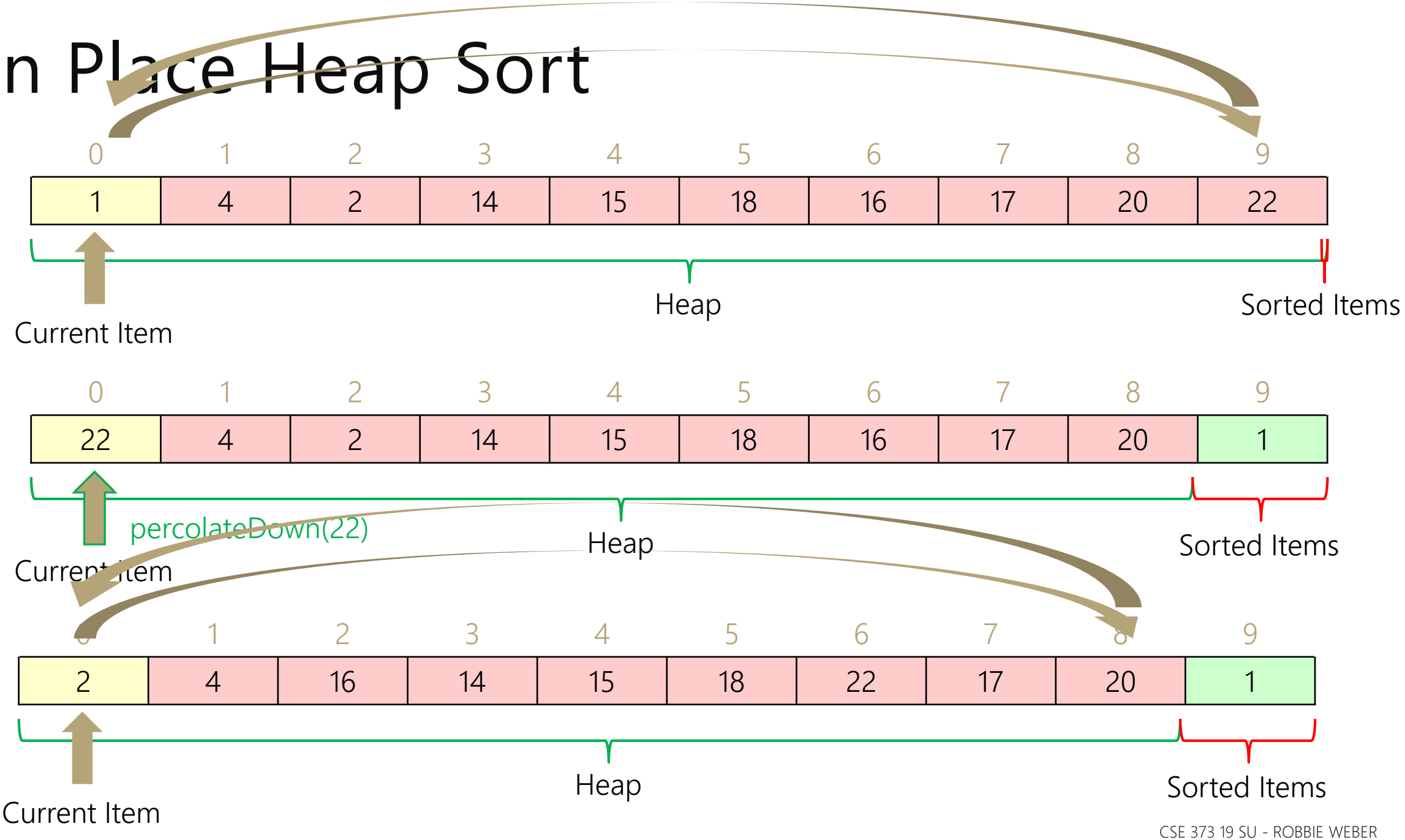
https://www.youtube.com/watch?v=Xw2D9aJRBY4

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

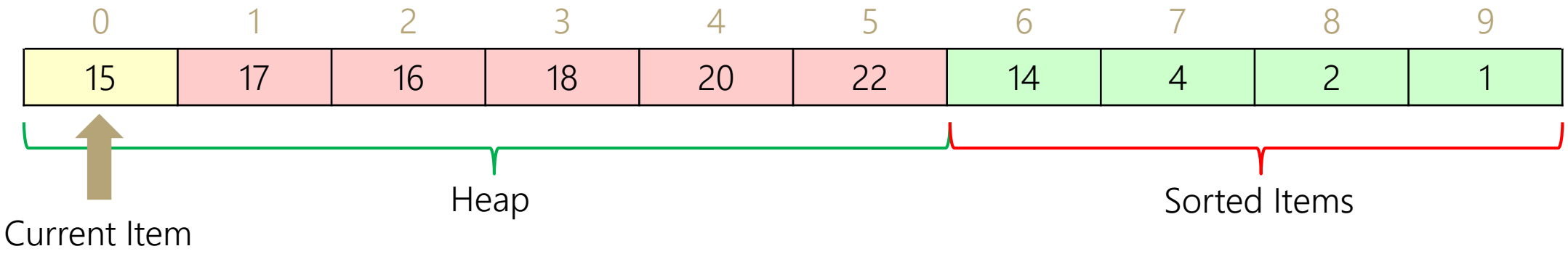
```
public void heapSort(collection) {
   E[] heap = buildHeap(collection)
   E[] output = new E[n]
   for (n)
      output[i] = removeMin(heap)
}
```

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

In Place Heap Sort



In Place Heap Sort




```
public void inPlaceHeapSort(collection) {
   E[] heap = buildHeap(collection)
   for (n)
      output[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?	$O(n \log n)$
Best case runtime?	$O(n \log n)$
In-practice runtime?	$O(n \log n)$
Stable?	No
In-place?	Yes

Principle 3: Divide and Conquer

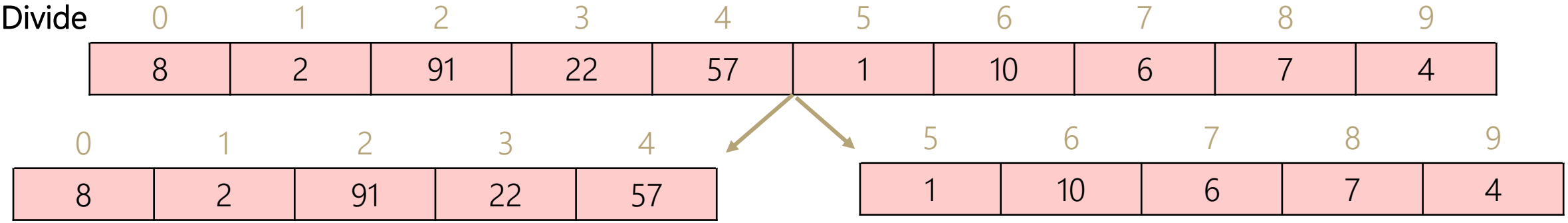
1. Divide your work into smaller pieces recursively
   - Pieces should be smaller versions of the larger problem
2. Conquer the individual pieces
   - Recursion!
   - Until you hit the base case
3. Combine the results of your recursive calls

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

Merge Sort

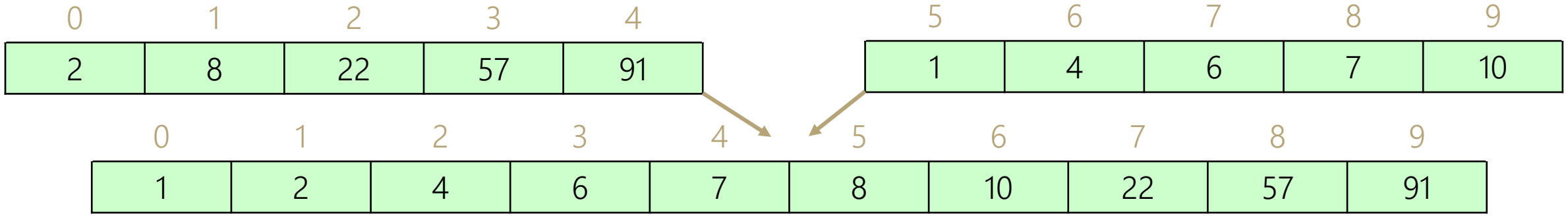
https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide



Sort the pieces through the **magic** of recursion

Combine



Merge Sort

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

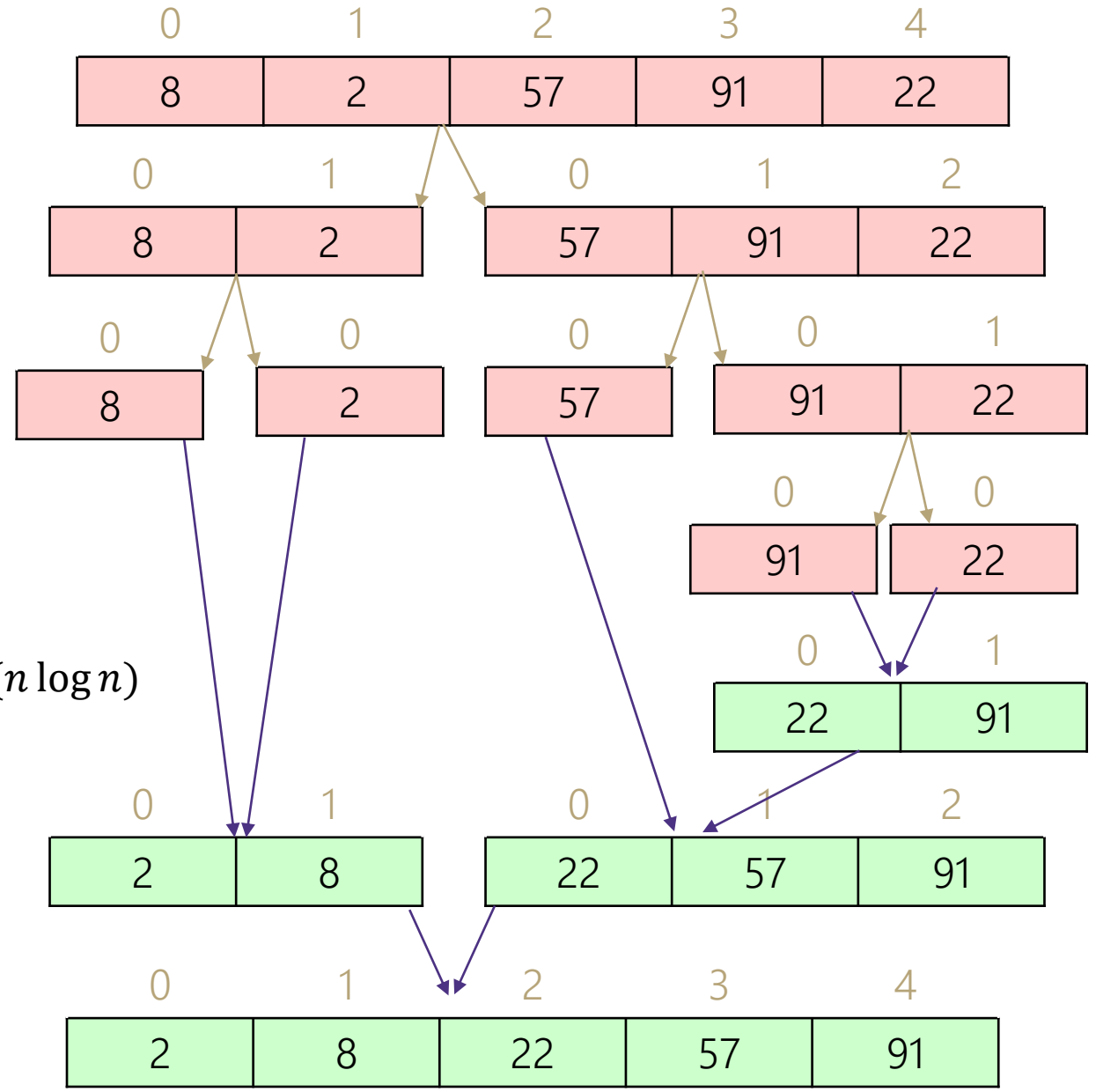
Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases} = O(n \log n)$

Best case runtime? Same

Average runtime? Same

Stable? Yes

In-place? No



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.

- Elements that are "small" and elements that are "large"
- What do I mean by "small" and "large" ?

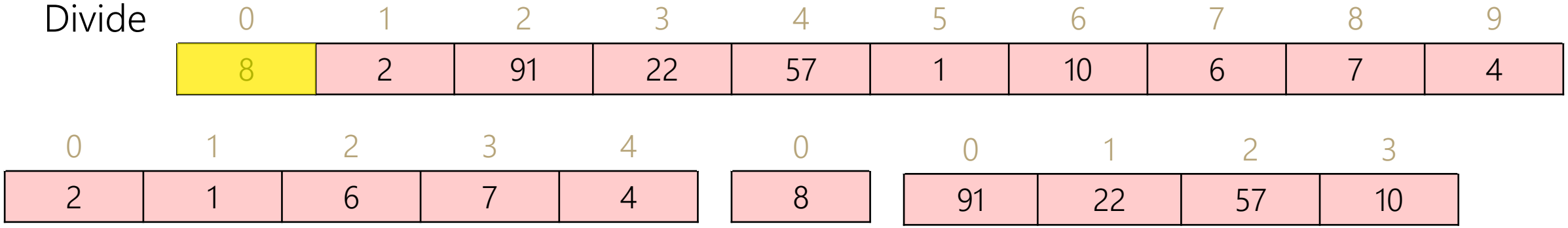
Choose a "pivot" value (an element of the array)

One array has elements smaller than pivot, other has elements larger than pivot.

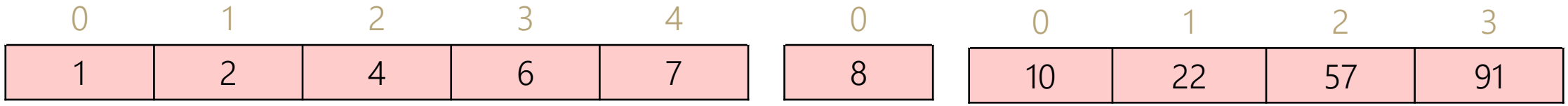
Quick Sort v1

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

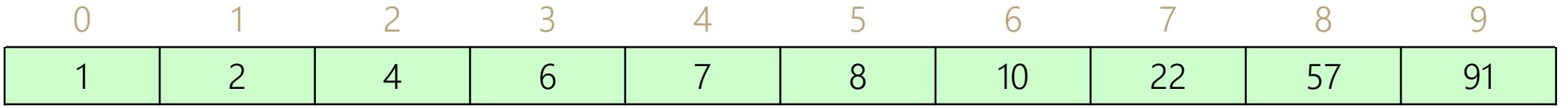
Divide



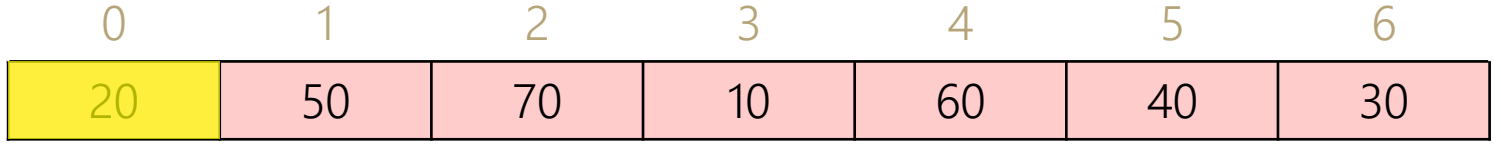
Sort the pieces through the **magic** of recursion

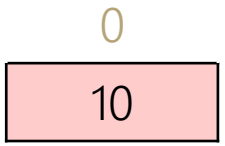


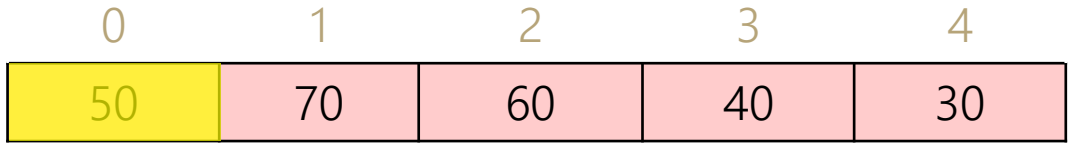
Combine (no extra work if in-place)

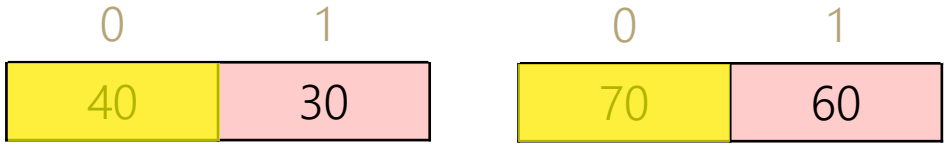


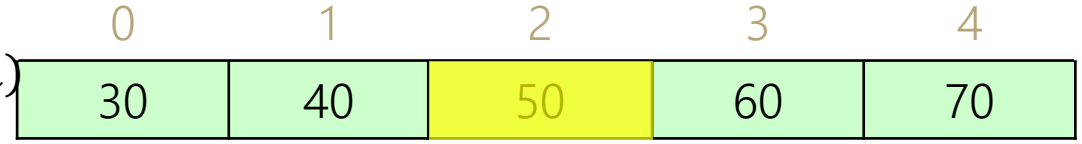
Quick Sort v1

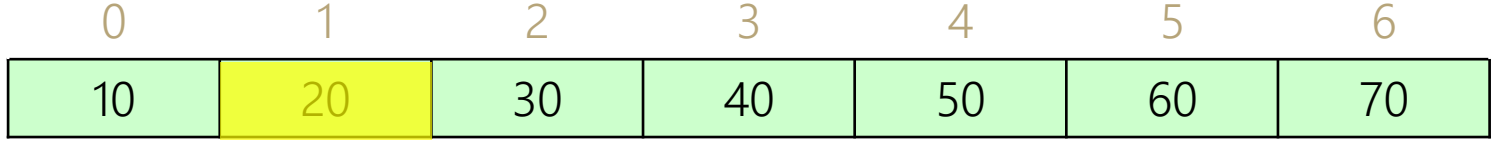













```
quickSort(input) {
   if (input.length == 1)
      return
   else
      pivot = getPivot(input)
      smallerHalf = quickSort(getSmaller(pivot, input))
      largerHalf = quickSort(getBigger(pivot, input))
      return smallerHalf + pivot + largerHalf
}
```

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

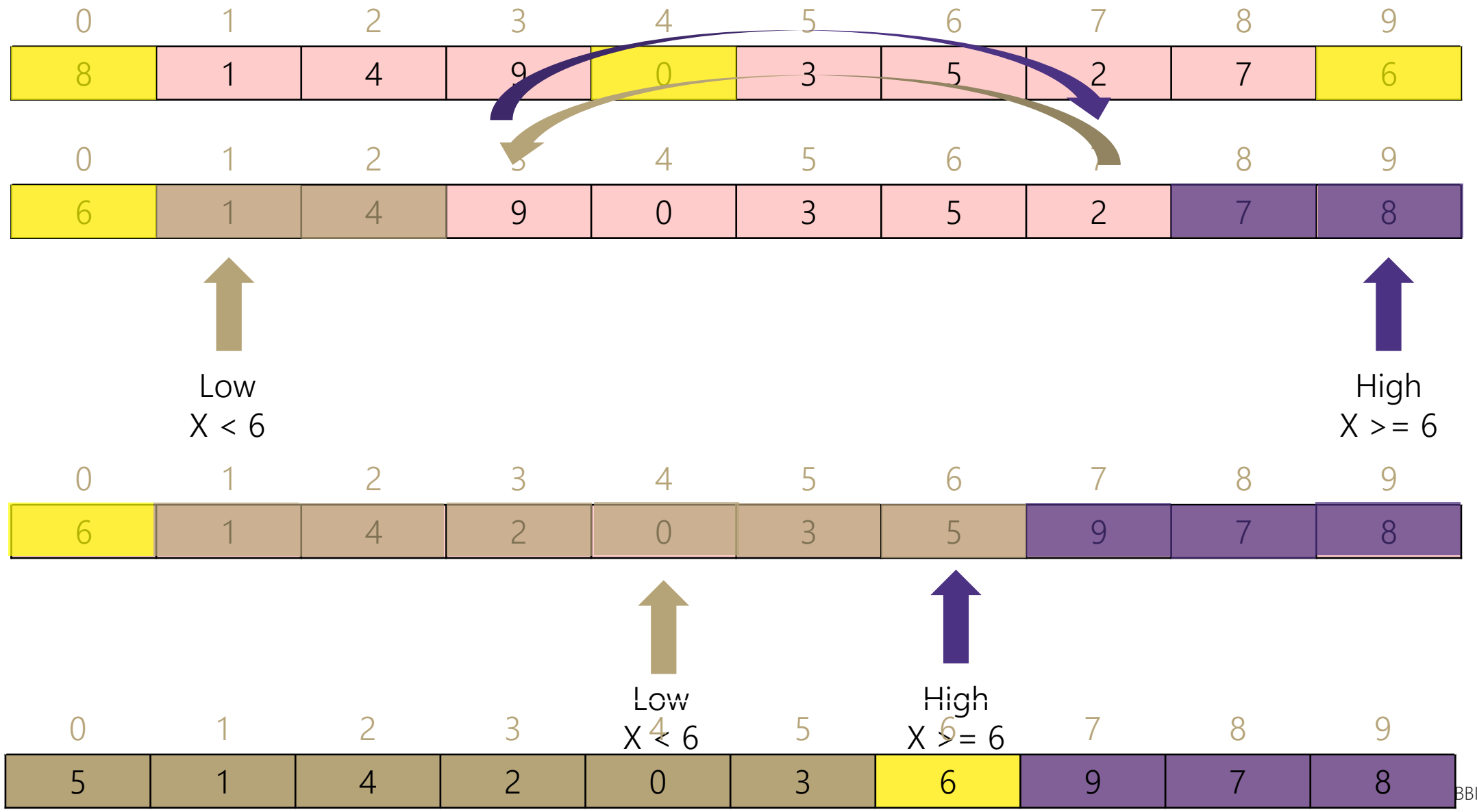
Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$ $= O(n \log n)$

In-practice runtime? Just trust me $O(n \log n)$

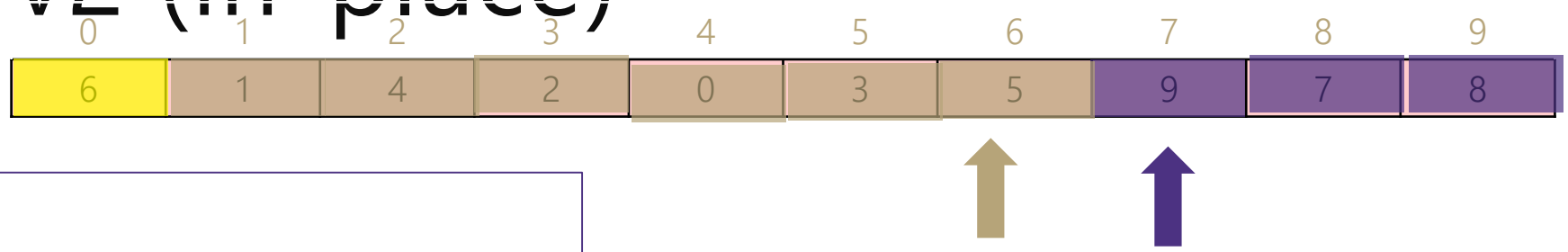
Stable? No

In-place? Can be done

Quick Sort v2 (in-place)



Quick Sort v2 (in-place)




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

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

Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$ = O(nlogn)

In-practice runtime? Just trust me = O(nlogn)

Stable? No

In-place? Yes

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.