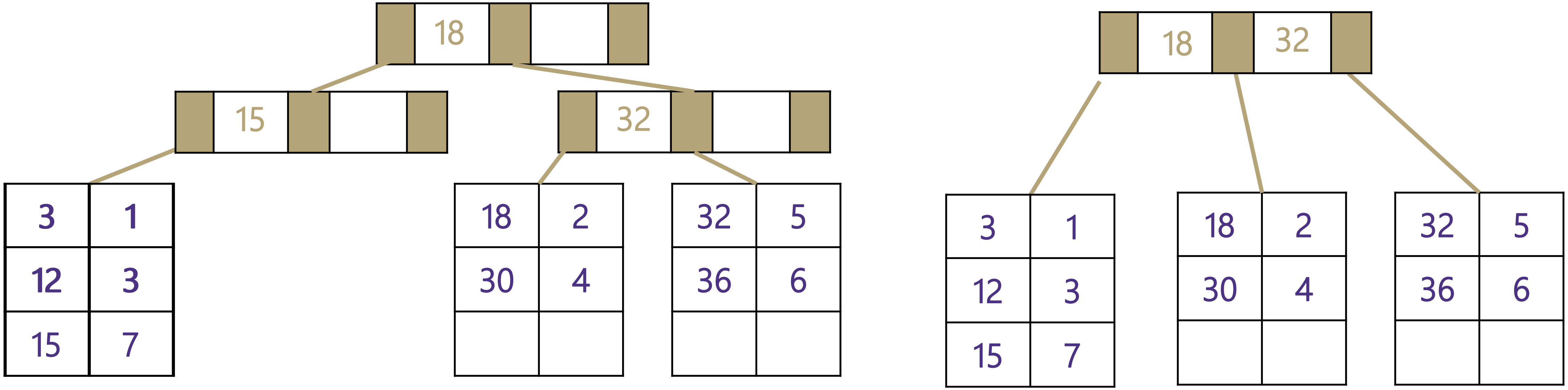
Comparisons Sorts

Data Structures and Parallelism

Deletion – Clarification

Delete 14. Merge up the tree.
Update "signpost" to be smallest key in its right subtree.



Sorting

General Pre-processing Step

Let's us find the k^{th} element in $O(1)$ time for any k.

Also a convenient way to discuss algorithm design principles.

Three goals

Three things you might want in a sorting algorithm:

In-Place

- Only use $O(1)$ extra memory.
- Sorted array given back in the input array.

Stable

- If a appears before b in the initial array and a.compareTo(b) == 0
- Then a appears before b in the final array.
- Example: sort by first name, then by last name.

Fast

Insertion Sort

How you sort a hand of cards.

Maintain a sorted subarray at the front.

Start with one element.

While(your subarray is not the full array)

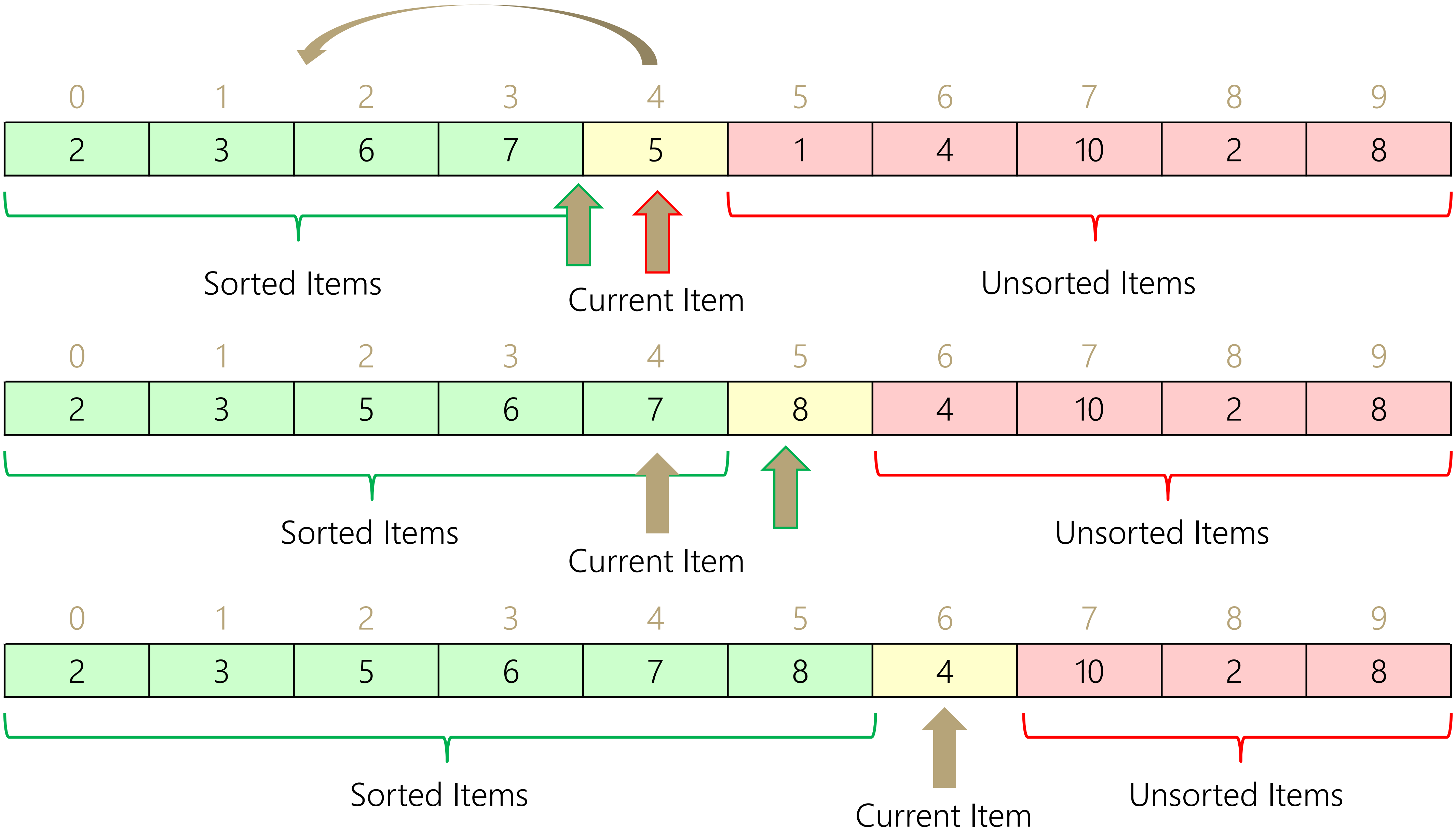
- Take the next element not in your subarray
- Insert it into the sorted subarray

Insertion Sort

```
for(i from 1 to n-1){
    int index = i
    while(a[index-1] > a[index]){
        swap(a[index-1], a[index])
        index = index-1
    }
}
```

Insertion Sort

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



Insertion Sort Analysis

Stable? Yes! (If you're careful)

In Place Yes!

Running time:

- Best Case: $O(n)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$

___________ Sort

Here’s another idea for a sorting algorithm:

Maintain a sorted subarray

While(subarray is not full array)

    Find the smallest element remaining in the unsorted part.

    Insert it at the end of the sorted part.

Selection Sort

Here's another idea for a sorting algorithm:

Maintain a sorted subarray

While(subarray is not full array)

  Find the smallest element remaining in the unsorted part.

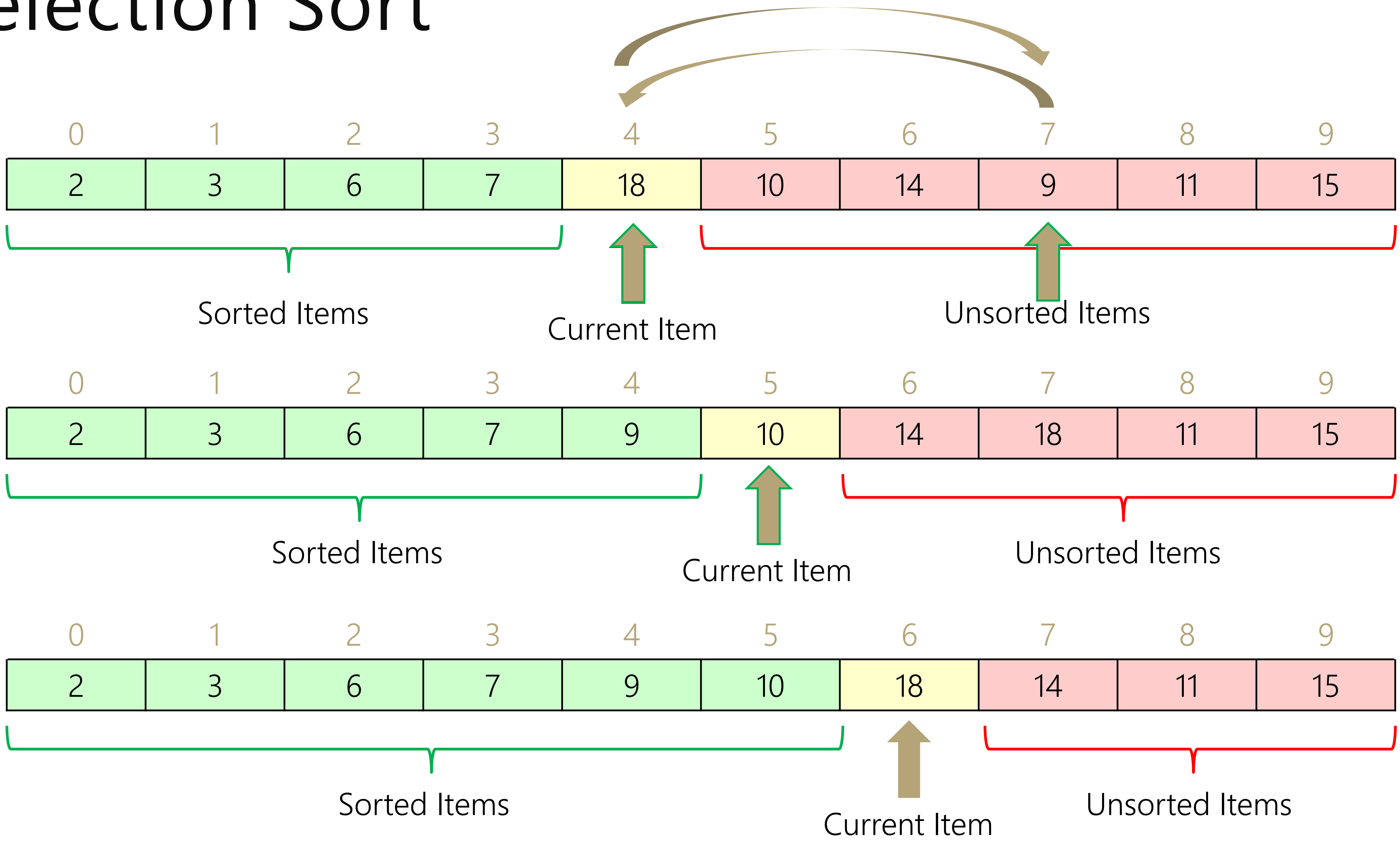
  - By scanning through the remaining array

  Insert it at the end of the sorted part.

Running time $O(n^2)$

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

Selection Sort



Selection Sort

Here's another idea for a sorting algorithm:

Maintain a sorted subarray

While(subarray is not full array)

Find the smallest element remaining in the unsorted part.

- By scanning through the remaining array

Insert it at the end of the sorted part.

Running time $O(n^2)$

Can we do better? With a data structure?

Heap Sort

Here's another idea for a sorting algorithm:

Maintain a sorted subarray; **Make the unsorted part a min-heap**

While(subarray is not full array)

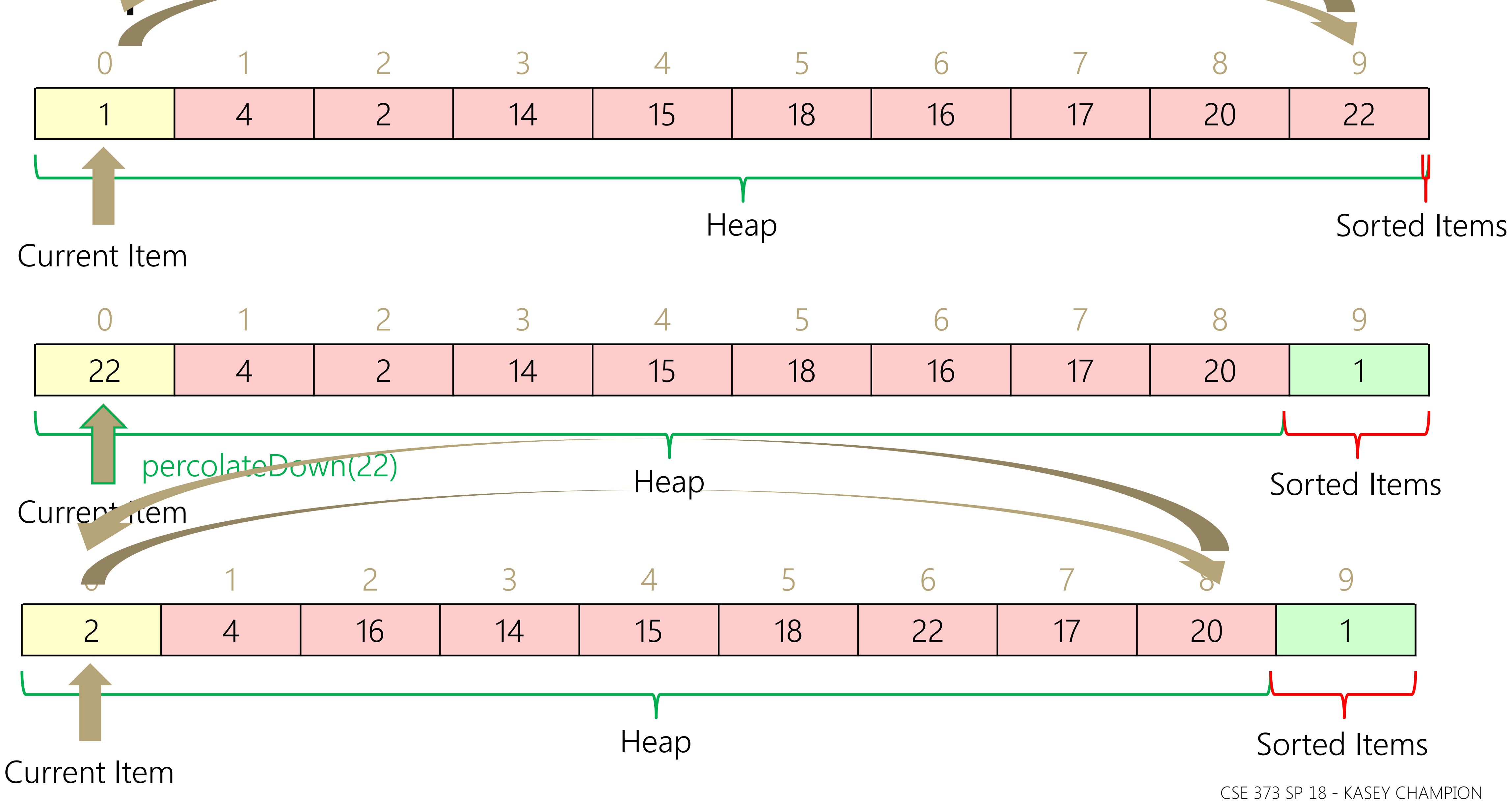
Find the smallest element remaining in the unsorted part.
- By calling `removeMin` on the heap

Insert it at the end of the sorted part.

Running time $O(n \log n)$

Heap Sort

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



Heap Sort (Better)

We're sorting in the wrong order!

- Could reverse at the end.

Our heap implementation will implicitly assume that the heap is on the left of the array.

Switch to a max-heap, and keep the sorted stuff on the right.

What's our running time? $O(n \log n)$

Heap Sort

Our first step is to make a heap. Does using `buildHeap` instead of inserts improve the running time?

Not in a big-O sense (though we did by a constant factor).

Exercise 7 will show some sorting problems where `buildHeap` does give you a better $O()$ bound.

In place: Yes

Stable: No

A Different Idea

So far our sorting algorithms:

- Start with an (empty) sorted array
- Add something to it.

Different idea: Divide And Conquer:

Split up array (somehow)

Sort the pieces (recursively)

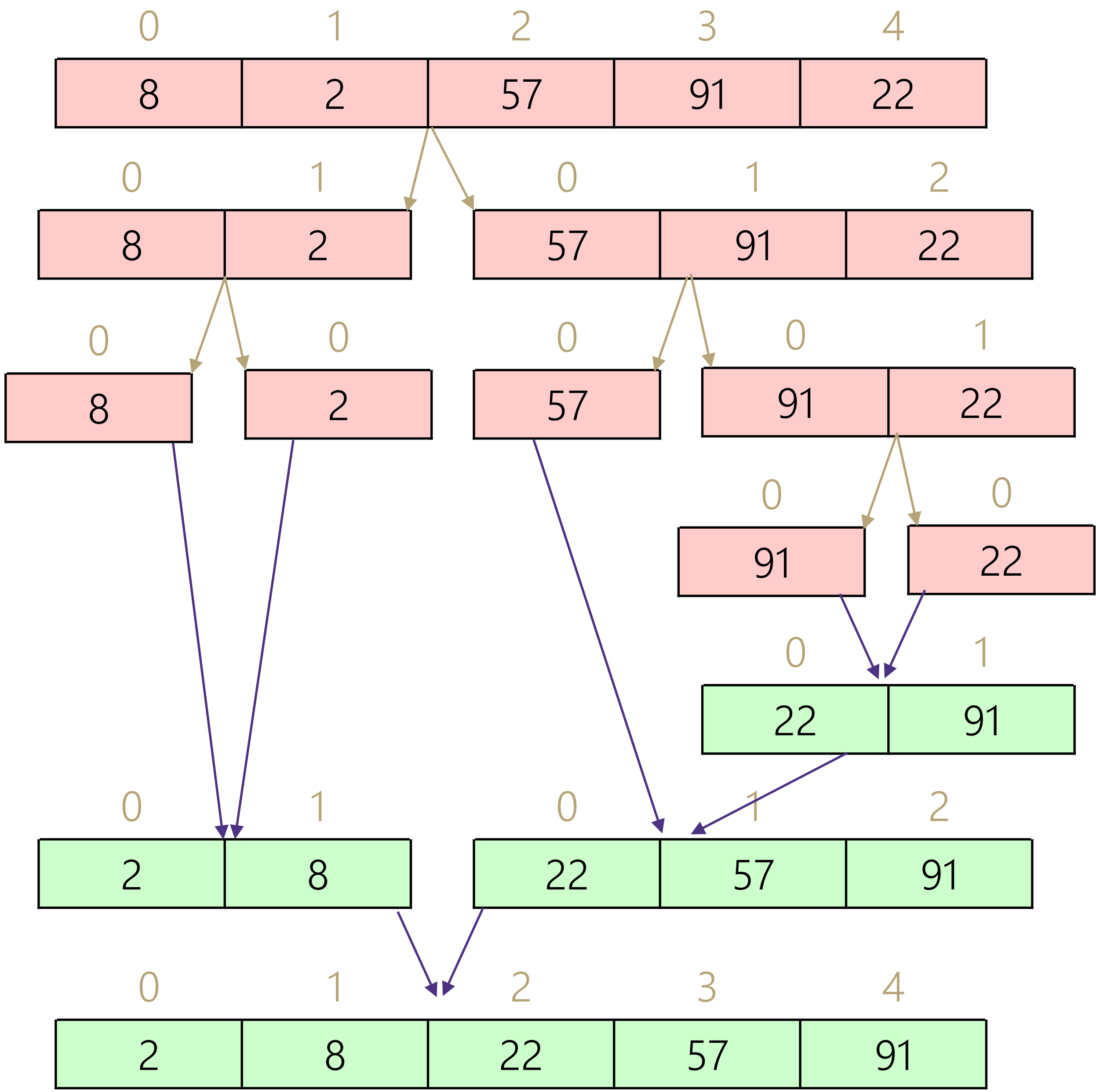
Combine the pieces

Merge Sort

Split array in the middle

Sort the two halves

Merge them together



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

Merge Sort Pseudocode

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

How Do We Merge?

Turn two sorted lists into one sorted list:

Start from the small end of each list.

Copy the smaller into the combined list

Move that pointer one spot to the right.

3	15	27

5	12	30

3	5	12	15	27	30

Merge Sort Analysis

Running Time:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c_1 n & \text{if } n \geq 1 \\ c_2 & \text{otherwise} \end{cases}$$

This is a closed form you should have memorized by the end of the quarter.

The closed form is $\Theta(n \log n)$.

Stable: yes! (if you merge correctly)

In place: no.

Some Optimizations

We need extra memory to do the merge

It's inefficient to make a new array every time

Instead have a single auxiliary array

- Keep reusing it as the merging space

Even better: make a single auxiliary array

- Have the original array and the auxiliary array "alternate" being the list and the merging space.

Quick Sort

Still Divide and Conquer, but a different idea:

Let's divide the array into "big" values and "small" values

- And recursively sort those

What's "big"?

- Choose an element ("the pivot") anything bigger than that.

How do we pick the pivot?

For now, let's just take the first thing in the array:

Swapping

How do we divide the array into “bigger than the pivot” and “less than the pivot?”

1. Swap the pivot to the far left.
2. Make a pointer i on the left, and j on the right
3. Until i, j meet
   - While $A[i] < \mathbf{pivot}$ move i left
   - While $A[j] > \mathbf{pivot}$ move j right
   - Swap $A[i], A[j]$
4. Swap A[i] or A[i-1] with pivot.

Swapping

0	1	2	3	4	5	6	7	8	9
8	3	5	6	9	1	4	7	2	10

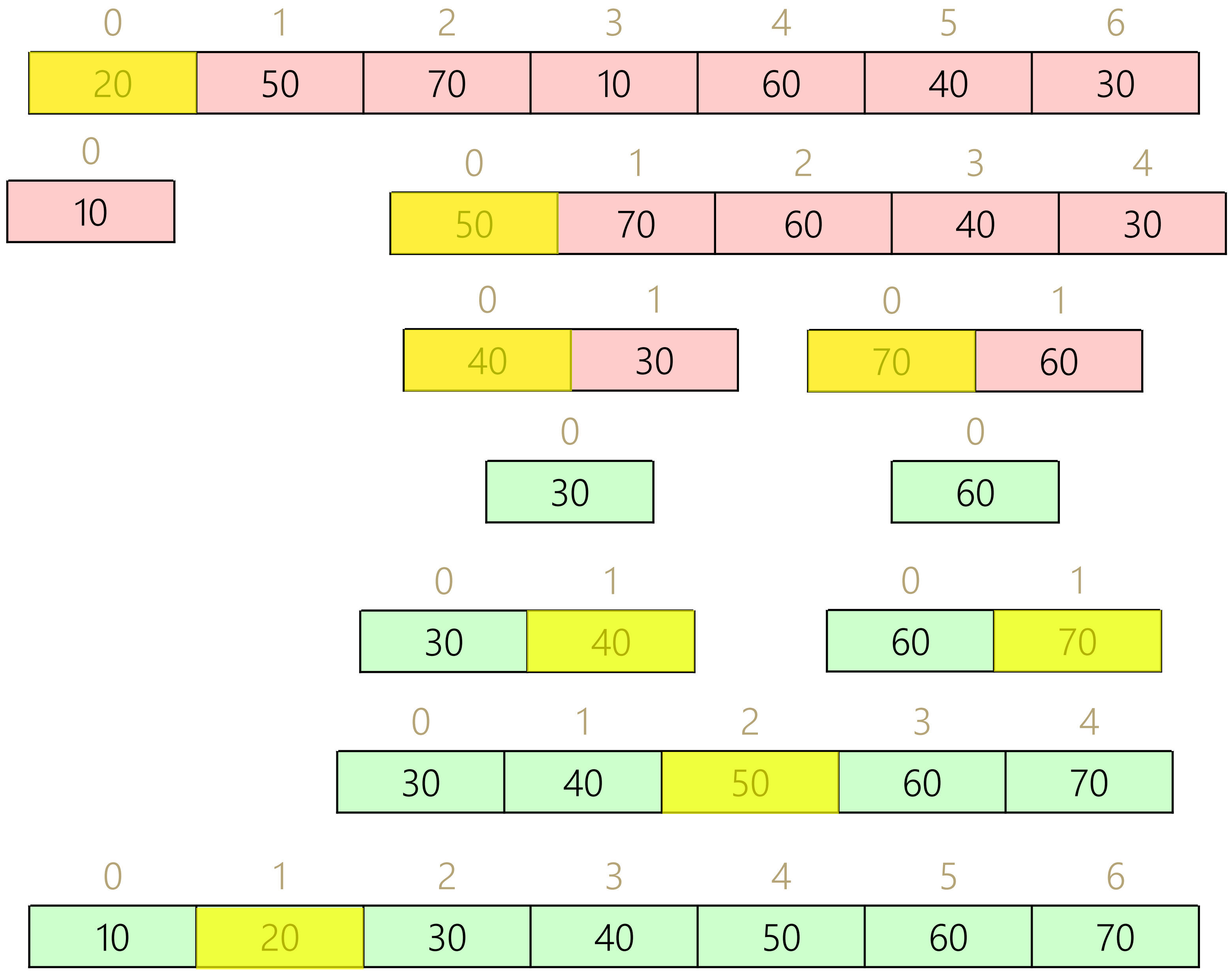
0	1	2	3	4	5	6	7	8	9
8	3	5	6	2	1	4	7	9	10

i, j met. $A[i]$ is larger than the pivot, so it belongs on the right, but $A[i-1]$ belongs on the left. Swap pivot and $A[i-1]$.

0	1	2	3	4	5	6	7	8	9
7	3	5	6	2	1	4	8	9	10

Quick Sort

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



Quick Sort Analysis (Take 1)

What is the best case and worst case for a pivot?

- Best case: Picking the median
- Worst case: Picking the smallest or largest element

Recurrences:

Best:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c_1 n & \text{if } n \geq 2 \\ c_2 & \text{otherwise} \end{cases}$$

Worst:

$$T(n) = \begin{cases} T(n-1) + c_1 n & \text{if } n \geq 2 \\ c_2 & \text{otherwise} \end{cases}$$

Running times:

- Best: $O(n \log n)$
- Worst: $O(n^2)$

Choosing a Pivot

Average case behavior depends on a good pivot.

Pivot ideas:

Just take the first element

- Simple. But an already sorted (or reversed) list will give you a bad time.

Pick an element uniformly at random.

- $O(n \log n)$ running time with probability at least $1 - 1/n^2$.
- Regardless of input!
- Probably too slow in practice :(

Find the actual median!

- You can actually do this in linear time
- Definitely not efficient in practice

Choosing a Pivot

Median of Three

- Take the median of the first, last, and midpoint as the pivot.
- Fast!
- Unlikely to get bad behavior (but definitely still possible)
- Reasonable default choice.

Quick Sort Analysis

Running Time:

- Worst $O(n^2)$
- Best $O(n \log n)$
- Average $O(n \log n)$ (not responsible for the proof, talk to Robbie if you're curious)

In place: Yes

Stable: No.

Lower Bound

We keep hitting $O(n \log n)$ in the worst case.

Can we do better?

Or is this $O(n \log n)$ pattern a fundamental barrier?

Without more information about our data set, we can do no better.

Comparison Sorting Lower Bound

Any sorting algorithm which only interacts with its input by comparing elements must take $\Omega(\mathrm{n} \log \mathrm{n})$ time.

We'll prove this theorem on Friday!