Comparisons Sorts

CSE 332 Spring 25
Lecture 12

Linear Probing

First idea: linear probing

h(key) % TableSize full?

Try (h(key) + 1) % TableSize.

Also full? (h(key) + 2) % TableSize.

Also full? (h(key) + 3) % TableSize.

Also full? (h(key) + 4) % TableSize.

…

Example

Insert the hashes: 38, 19, 8, 109, 10 into an empty hash table of size 10.

0	1	2	3	4	5	6	7	8	9
8	109	10						38	19

How Long Does Insert Take?

If $\lambda < 1$ we'll find a spot eventually.

What's the average running time?

Uniform Hashing Assumption

for any pair of elements x,y

the probability that h(x) = h(y) is $\frac{1}{TableSize}$

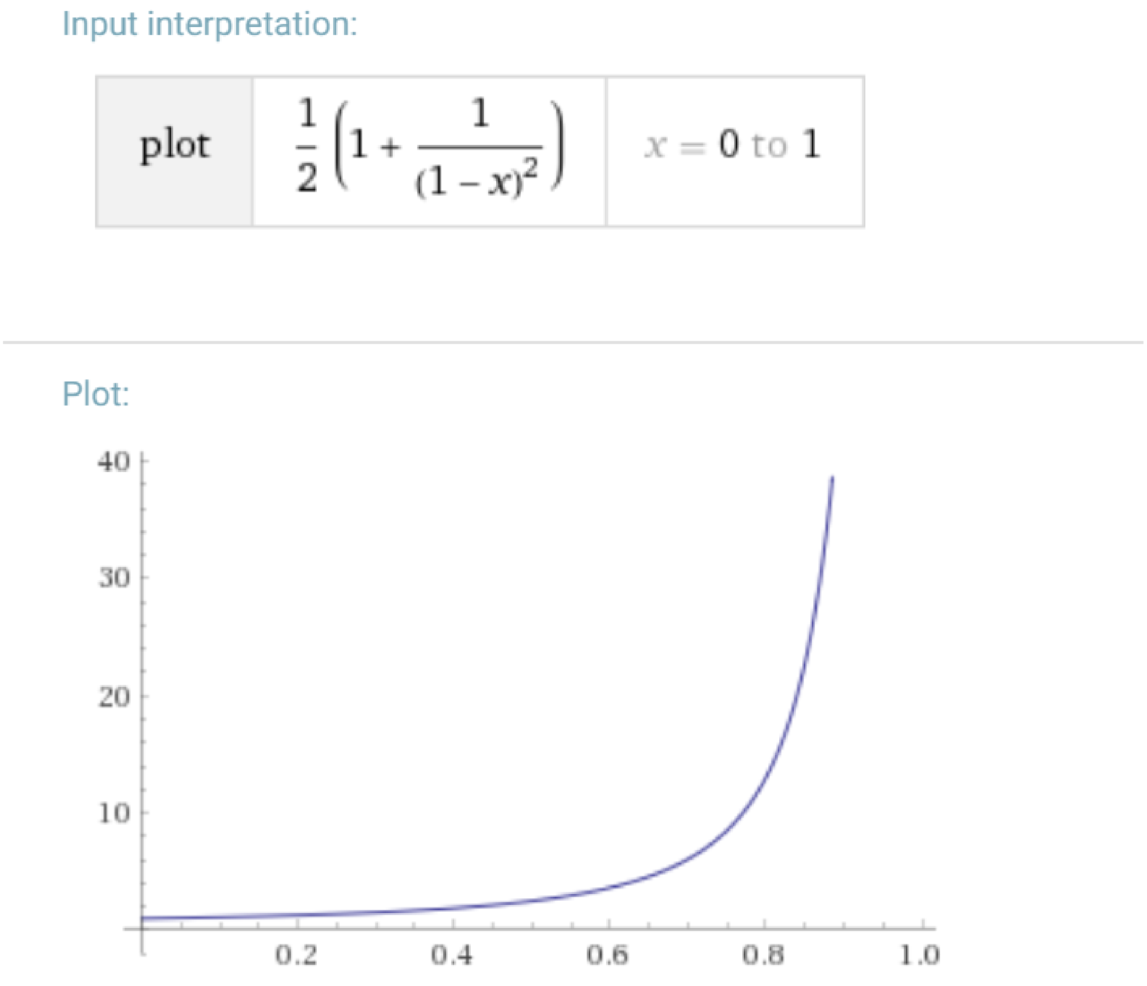
If find is unsuccessful: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$

If find is successful: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$

We won't prove these (they're not even in the textbook)

- Ask Robbie for references if you're really interested.

When to Resize



Input interpretation:

plot	$\frac{1}{2}\left(1+\frac{1}{(1-x)^2}\right)$	$x = 0$ to 1

Plot:

We definitely want to resize before λ gets close to 1.
Taking $\lambda = 0.5$ as a resize point probably avoids the bad end of this curve.

Remember these are the average find times.
Even under UHA, the worst possible find is a bit worse than this *with high probability*.

Why are there so many probes?

The number of probes is a result of **primary clustering**

If a few consecutive spots are filled,

Hashing to any of those spots will make more consecutive filled spots.

Quadratic Probing

Want to avoid primary clustering.

If our spot is full, let’s try to move far away relatively quickly.

h(key) % TableSize full?

Try (h(key) + 1) % TableSize.

Also full? (h(key) + 4) % TableSize.

Also full? (h(key) + 9) % TableSize.

Also full? (h(key) + 16) % TableSize.

…

Example

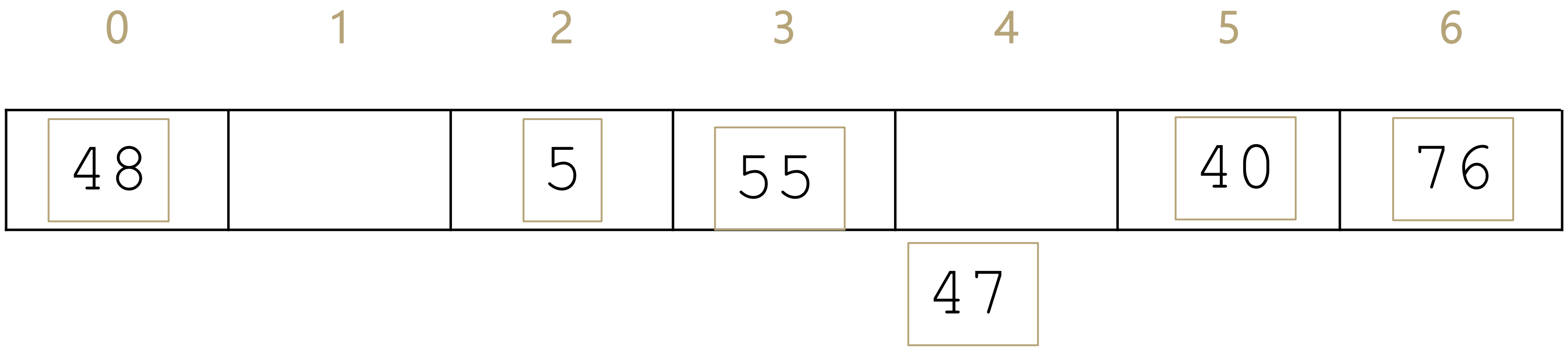
Insert: 89, 18, 49, 58, 79 into an empty hash table of size 10.

0	1	2	3	4	5	6	7	8	9
49		58	79					18	89

Then insert 76, 40, 48, 5, 55,47 into an empty hash table of size 7

0	1	2	3	4	5	6
48		5	55		40	76
				47		

Example



0	1	2	3	4	5	6
48		5	55		40	76

47

47%7=5 (full)
47+1^2 % 7 = 6 (full)
47+2^2 % 7 = 2 (full)
47+3^2 % 7 = 0 (full)
47+4^2 % 7 = 0 (full)
47+5^2 % 7 = 2 (full)
47+6^2 % 7 = 6 (full) UH-OH

Quadratic probing does not check every single location! But if $TableSize$ is prime, it checks half of them!

Quadratic Probing: Proof

Claim: If $\lambda < \frac{1}{2}$, and TableSize is prime then quadratic probing will find an empty slot.

Quadratic Probing: Proof

Claim: If $\lambda < \frac{1}{2}$, and TableSize is prime then quadratic probing will find an empty slot.

Enough to show, first TableSize/2 probes are distinct.

For contradiction, suppose there exists some $i \neq j$ with $i + j < \text{TableSize}$ such that

$$(h(x) + i^2) \bmod \text{TableSize} = (h(x) + j^2) \bmod \text{TableSize}$$

$$i^2 \bmod \text{TableSize} = j^2 \bmod \text{TableSize}$$

$$(i^2 - j^2)\text{mod TableSize} = 0$$

Quadratic Probing: Proof

$(i^2 - j^2)\text{mod TableSize} = 0$

$(i + j)(i - j)\text{mod TableSize} = 0$

Thus TableSize divides $(i + j)(i - j)$

But TableSize is prime, so

TableSize divides $i + j$ or TableSize divides $i - j$

But $i + j < \text{TableSize}$ (and therefore $i - j < \text{TableSize}$ as well).

That’s a contradiction! So no repeated probes exist in the first $TableSize/2$ probes.

Problems

Still have a fairly large amount of probes (we won’t even try to do the analysis)

We don’t have primary clustering, but we do have **secondary clustering**

If you initially hash to the same location, you follow the same set of probes.

Double Hashing

Instead of probing by a fixed value every time, probe by some new hash function!

h(key) % TableSize full?

Try (h(key) + g(key)) % TableSize.

Also full? (h(key) + 2*g(key)) % TableSize.

Also full? (h(key) + 3*g(key)) % TableSize.

Also full? (h(key) + 4*g(key)) % TableSize.

…

Example

Insert the following keys into a table of size 10 with the following hash functions: 13, 28, 33, 147, 43

Primary hash function h(key) = key mod TableSize

Second hash function g(key) = 1 + ((key / TableSize) mod (TableSize-1))

0	1	2	3	4	5	6	7	8	9
			13				33	28	147

43

Running Times

Double Hashing will find lots of possible slots as long as g(key) and TableSize are relatively prime.

Under the uniform hashing assumption:

Expected probes for unsuccessful find: $\frac{1}{1-\lambda}$

Successful: $\frac{1}{1-\lambda}\ln\left(\frac{1}{1-\lambda}\right)$

Derivation beyond the scope of this course.

Ask Robbie for references if you want to learn more.

Summary

Separate Chaining

- Easy to implement
- Running times $O(1 + \lambda)$

Open Addressing

- Uses less memory.
- Various schemes:
- Linear Probing – easiest, but need to resize most frequently
- Quadratic Probing – middle ground
- Double Hashing – need a whole new hash function, but low chance of clustering.

Which you use depends on your application and what you're worried about.

Other Topics

Perfect Hashing –

- if you have fewer than 2^{32} possible keys, have a one-to-one hash function

Hopscotch and cuckoo hashing (more complicated collision resolution strategies)

Other uses of hash functions:

Cryptographic hash functions

- Easy to compute, but hard to tell given hash what the input was.

Check-sums

Locality Sensitive Hashing

- Map "similar" items to similar hashes

Wrap Up

Hash tables have great behavior on average,

As long as we make assumptions about our data set.

But for every hash function, there’s a set of keys you can insert to grind the hash table to a halt.

The number of keys is consistently larger than the number of ints.

An adversary can pick a set of values that all have the same hash.

Wrap Up

Can we avoid the terrible fate of our worst enemies forcing us to have $O(n)$ time dictionary operations?

If you have a lot of enemies, maybe use AVL trees.

But some hash table options:

Cryptographic hash functions – should be hard for adversary to find the collisions.

Randomized families of hash functions – have a bunch of hash functions, randomly choose a different one each time you start a hash table.

Done right – adversary won’t be able to cause as many collisions.

Wrap Up

Hash Tables:

- Efficient find, insert, delete **on average, under some assumptions**
- Items not in sorted order
- Tons of real world uses
- …and really popular in tech interview questions.

Need to pick a good hash function.

- Have someone else do this if possible.
- Balance getting a good distribution and speed of calculation.

Resizing:

- Always make the table size a prime number.
- λ determines when to resize, but depends on collision resolution strategy.

Sorting

Sorting

Common Pre-processing Step

- Lets 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 heap 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.
- Why? Imagine you sort an array by first name, then sort by last name. With a stable sort you get a list sorted by full name! (With an unstable sort the "Smiths" could go in any order).

Fast

Insertion Sort

How you sort a hand of cards.

Maintain a sorted subarray at the front.

- Start with one element. That’s sorted!

While(your sorted subarray is not the full array)

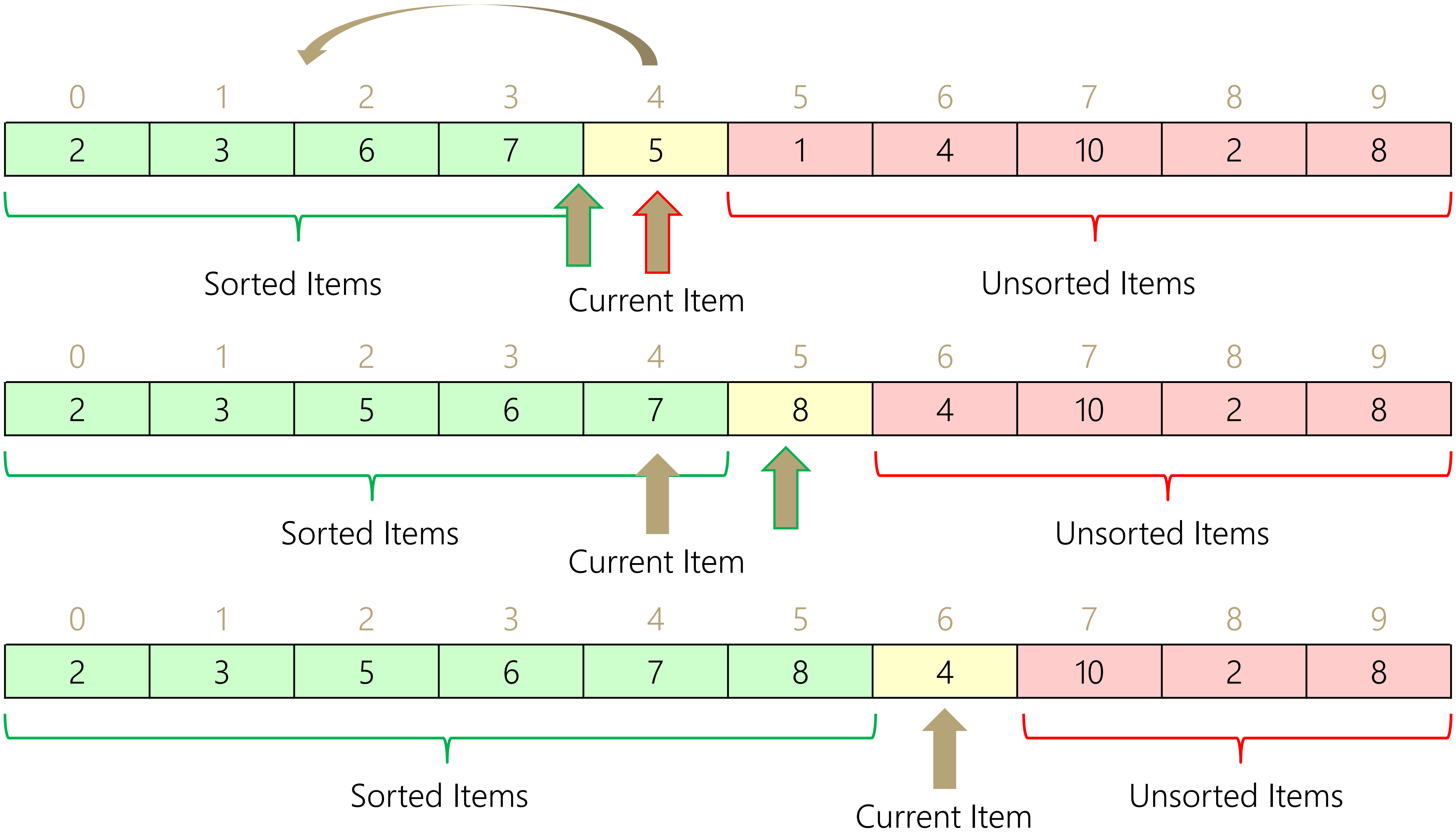
- Take the next element not in your subarray
- Insert it into the sorted subarray, by swapping until it’s in the right spot

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



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

Sorted Items

Current Item

Unsorted Items

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

Sorted Items

Current Item

Unsorted Items

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

Sorted Items

Current Item

Unsorted Items

Insertion Sort Analysis

Stable? Yes! (If you’re careful)

In Place Yes!

Running time:

- What’s the best case and worst case?

```
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 Analysis

Stable? Yes! (If you’re careful)

In Place Yes!

Running time:

- Best Case: $O(n)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$
- Won’t prove average cases (ask Robbie for explanations later if you’re curious); assumption is that all permutations are all equally likely.

___________ 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

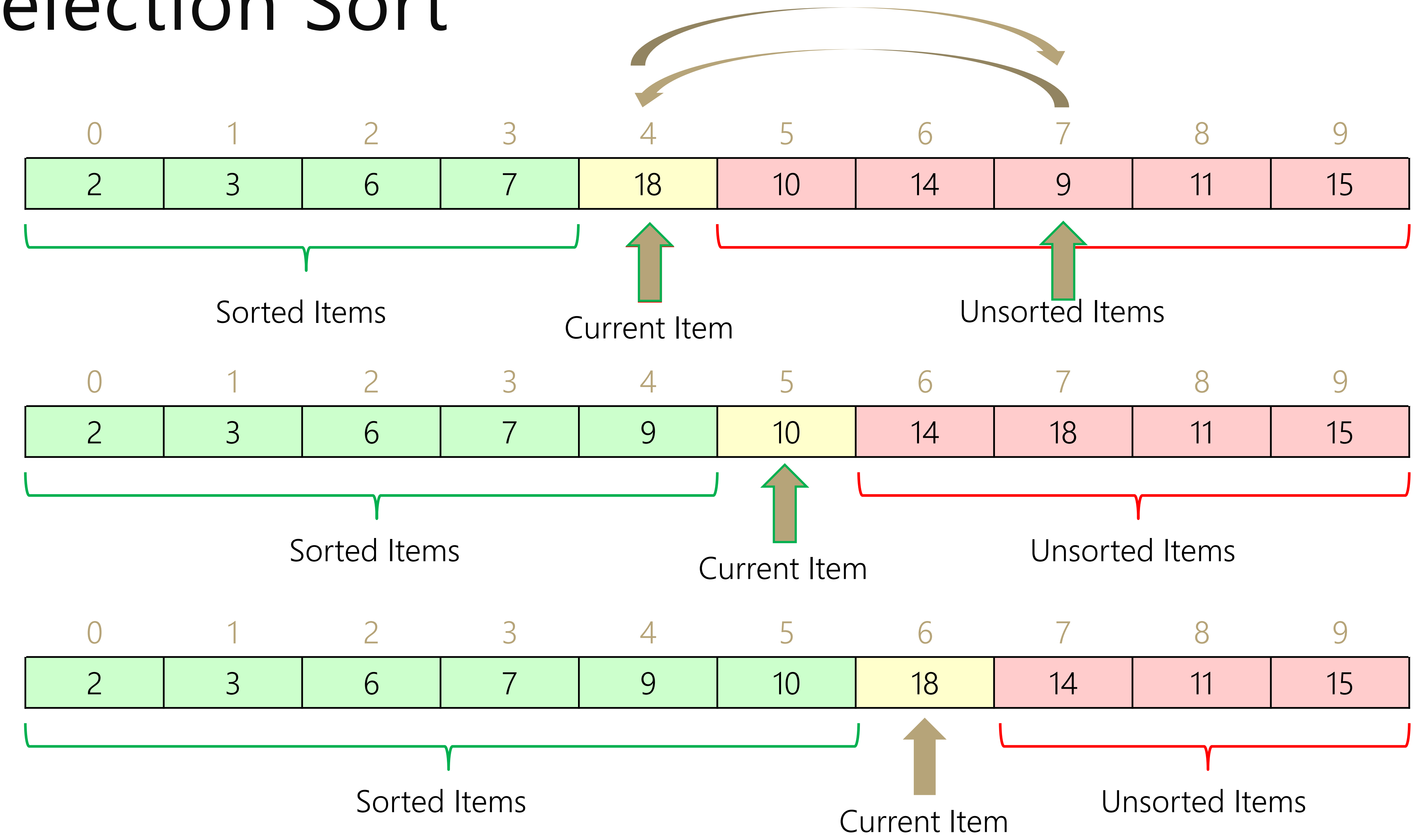
Here’s another idea for a sorting algorithm:

```
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



0	1	2	3	4	5	6	7	8	9
2	3	6	7	18	10	14	9	11	15

Sorted Items

Current Item

Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	14	18	11	15

Sorted Items

Current Item

Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	18	14	11	15

Sorted Items

Current Item

Unsorted Items

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)

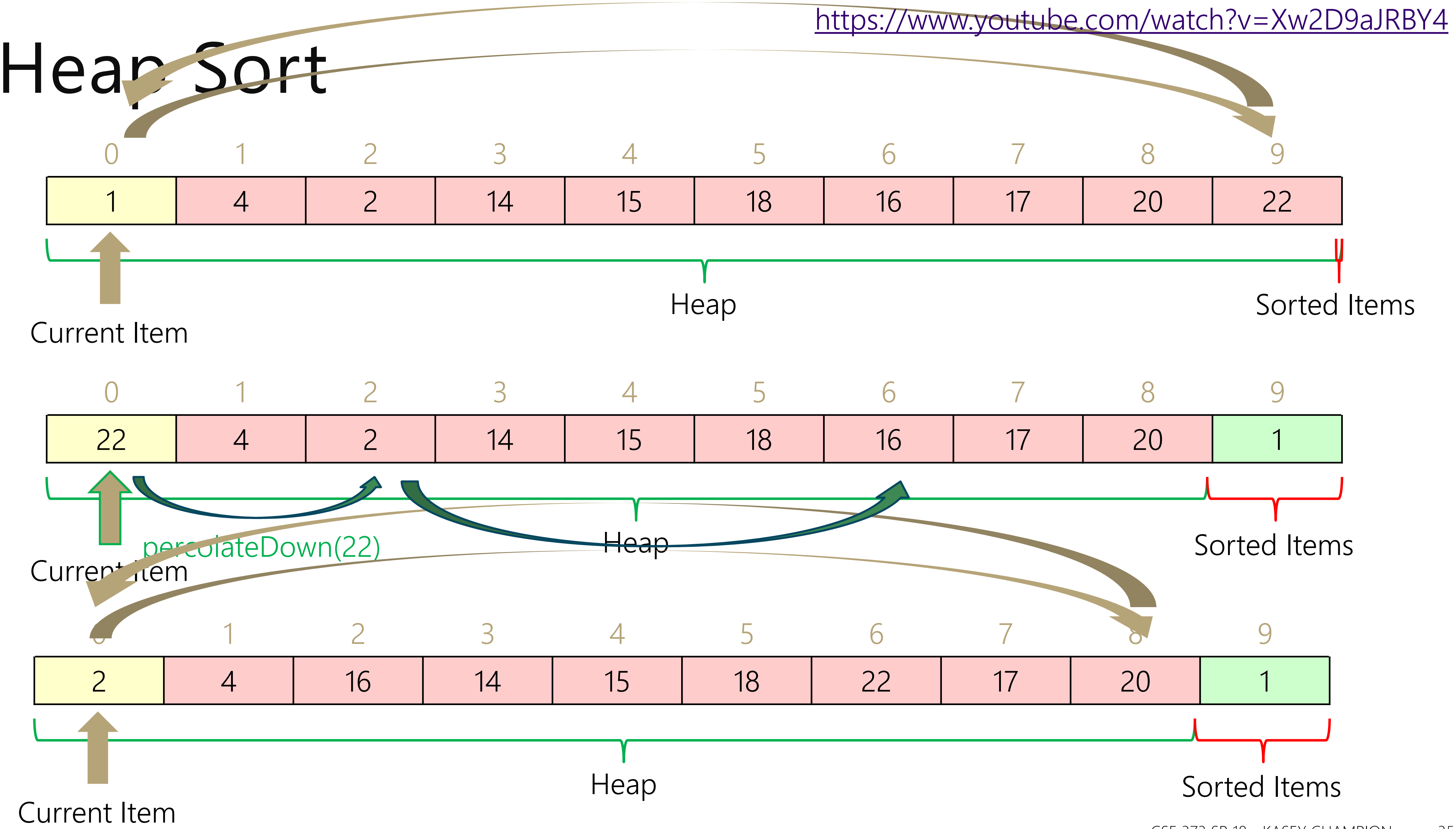
    Find the smallest element remaining in the unsorted part.
    -By calling `removeMin` on the heap

    Insert it at the end of the sorted part.

Running time $O(n \log n)$

Heap Sort

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



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

Heap Sort (Better)

We’re sorting in the wrong order!

- Can reverse at the end…

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

Better to switch to a max-heap and keep the sorted stuff on the right.

What’s our running time?

Worst Case: $O(n \log n)$
Best Case and Average Case are also $O(n \log n)$.

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).

In place: Yes

Stable: No---you'll prove this on your next exercise.

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)

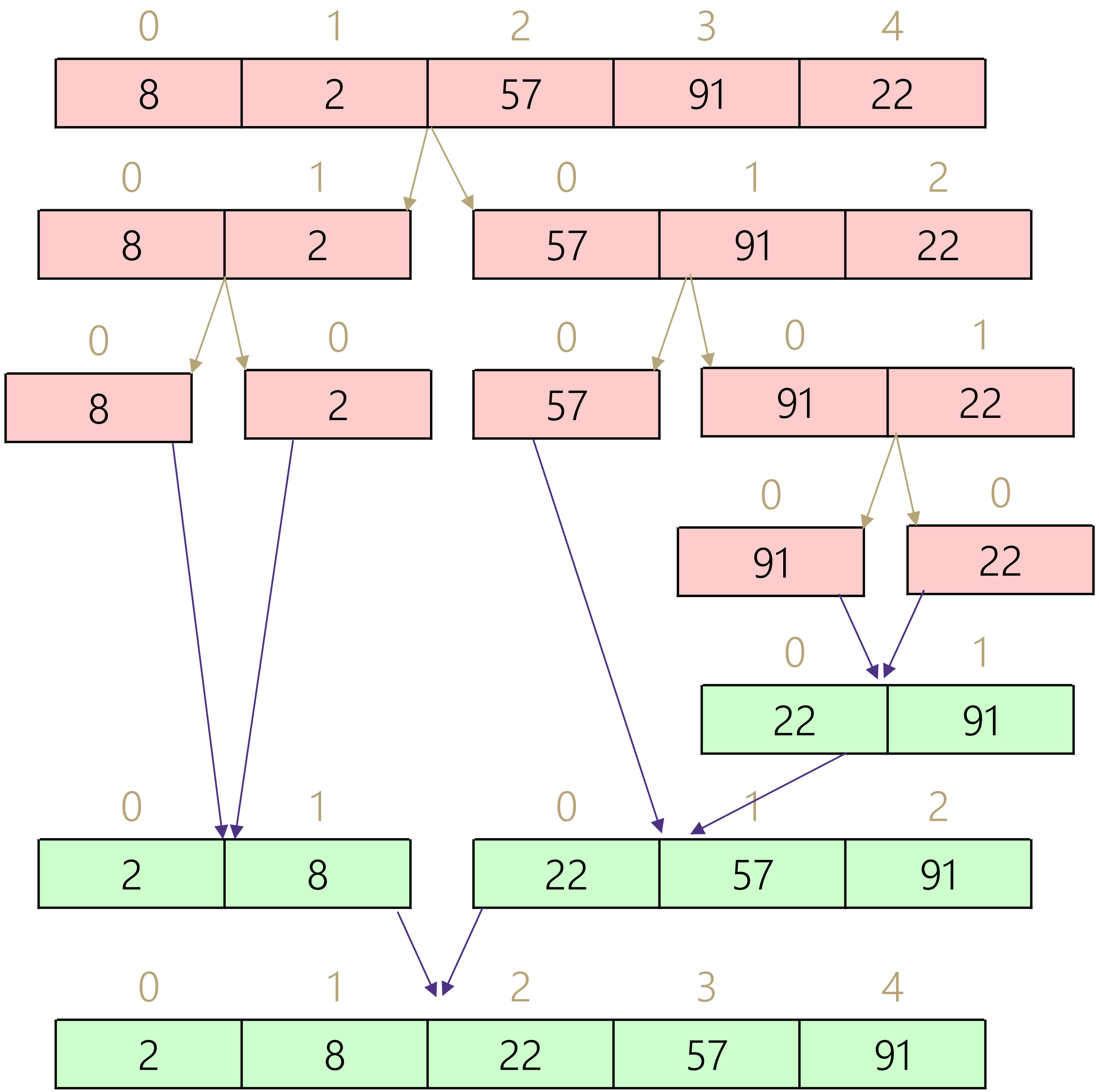
Combine the pieces

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 will 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?

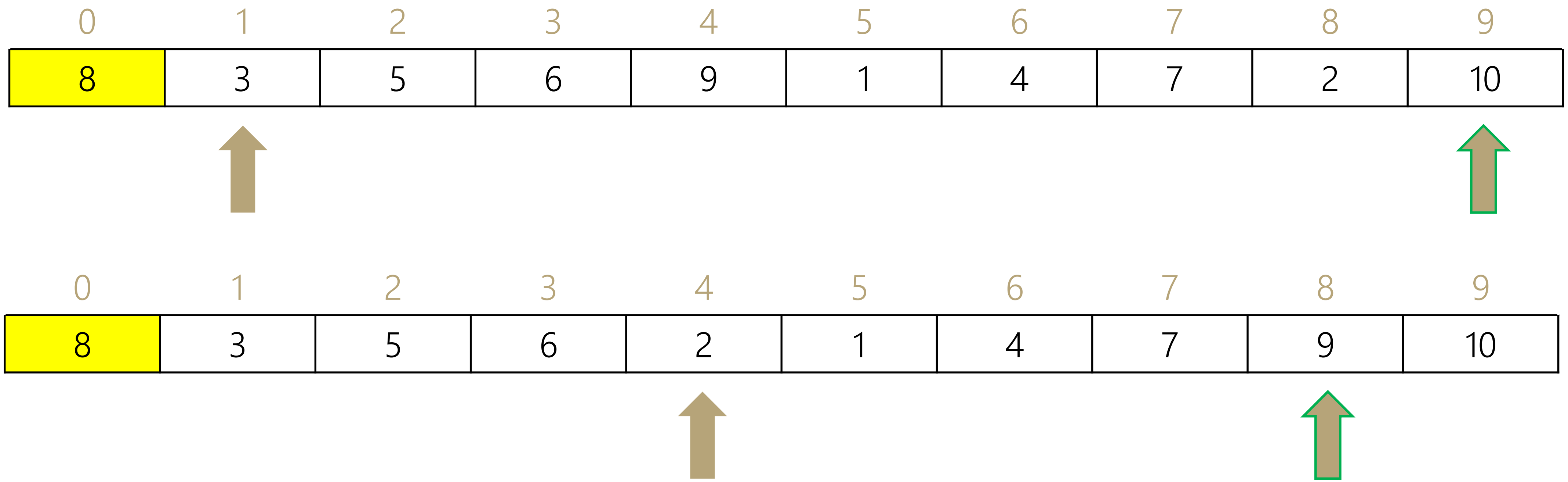
For now, let’s just take the first thing in the array:

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

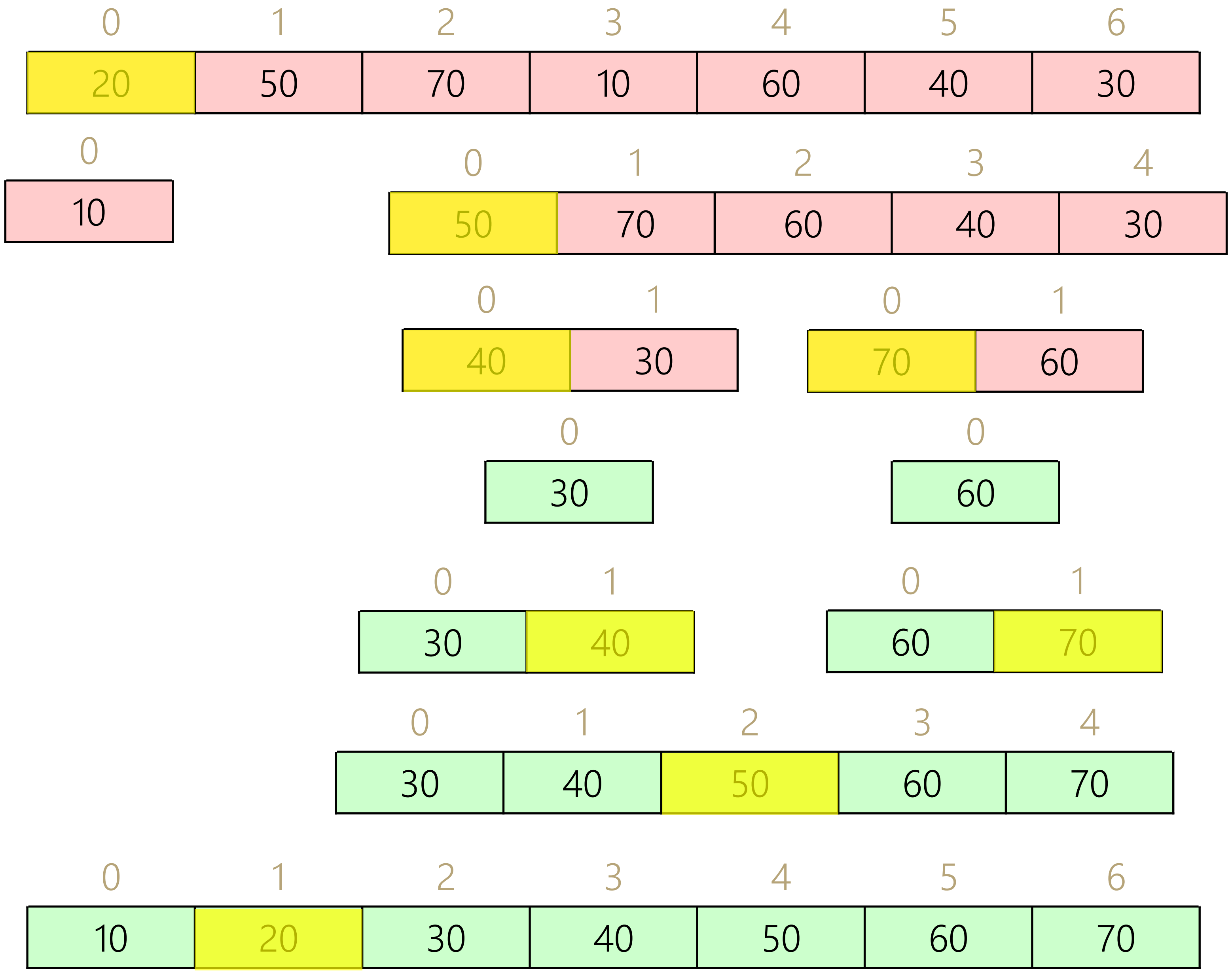
0	1	2	3	4	5	6	7	8	9
8	3	5	6	2	1	4	7	9	10

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



0	1	2	3	4	5	6
20	50	70	10	60	40	30

0
10

0	1	2	3	4
50	70	60	40	30

0	1
40	30

0	1
70	60

0
30

0
60

0	1
30	40

0	1
60	70

0	1	2	3	4
30	40	50	60	70

0	1	2	3	4	5	6
10	20	30	40	50	60	70

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).
- You can say something stronger, with probability $1 - 1/n$, the running time is $O(n \log n)$.

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 next week!