CS 161 - Section 2

CA : [Name of the CA]

Section 2 agenda

- Recap lectures:
  - Recurrence Relations
    - The Master Method
    - The Substitution Method
- Space complexity
- Handout

Recurrence Relations

- Divide and conquer algorithms are often recursive in nature – need to know how to solve recurrence relations for runtime analysis

- Two methods:
  - Master Method
  - Substitution Method

Remember that we can (mostly) ignore floors and ceilings in this class: $\left\lfloor\frac{n}{b}\right\rfloor$ or $\left\lceil\frac{n}{b}\right\rceil$ work too.

The Master Method

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Three parameters:
a : number of subproblems
b : factor by which input size shrinks
d : need to do n^d work to create all the subproblems and combine their solutions.

Requires all subproblems to be the same size!

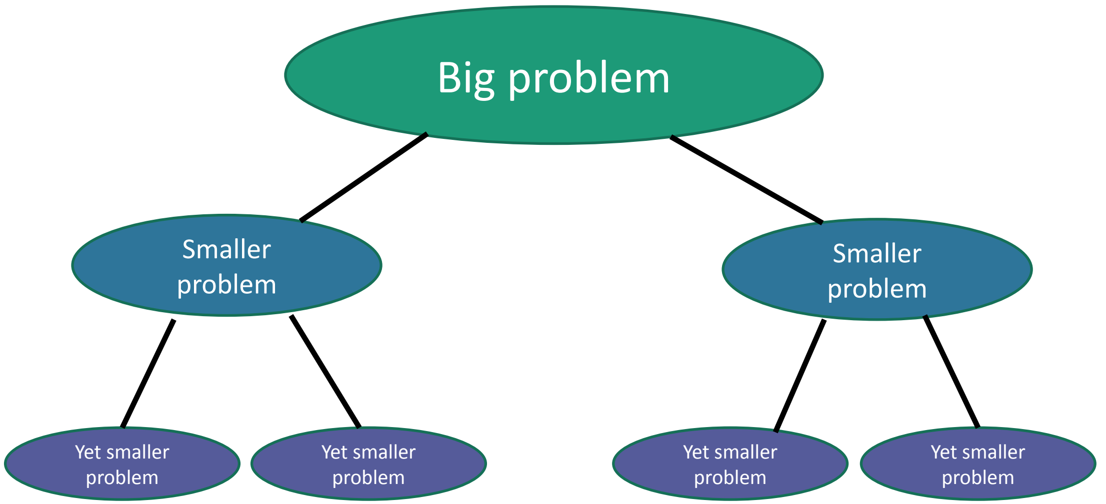
The Substitution Method

1. Guess what the answer is.
    - Try a few levels of recursion and see if you spot a pattern
2. Formally prove that that's what the answer is.
    - Often using induction. You may have to leave some constants unspecified till the end – then see what they need to be for the proof to work
3. Profit
    - Pretend you didn't do Steps 1 and 2 and write down a nice proof.

k-SELECT

Main idea: Divide and Conquer

- **Merge sort** – divide and conquer algorithm for sorting an array
- **SELECT** – divide and conquer algorithm for finding the *k*th smallest element of an array



SELECT

- `getPivot(A)` returns some pivot for us.
- `Partition(A,p)` splits up A into L, A[p], R.

- `Select`(A,k):
  - **If** len(A) <= 50:
    - A = `MergeSort`(A)
    - **Return** A[k-1]
  - p = `getPivot`(A)
  - L, pivotVal, R = `Partition`(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return `Select`(L, k)
  - **Else if** len(L) < k-1:
    - return `Select`(R, k – len(L) – 1)

Base Case: If the len(A) = O(1), then any sorting algorithm runs in time O(1).

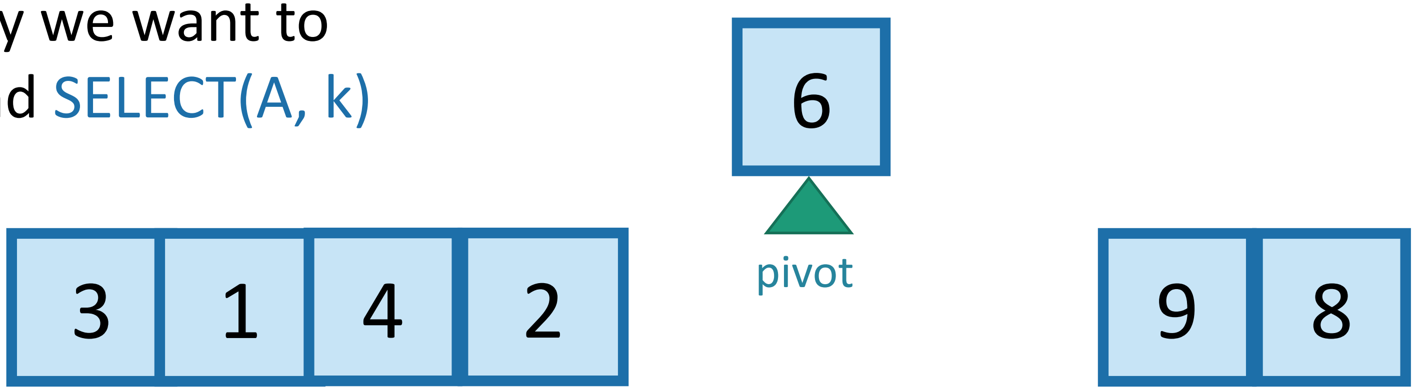
Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

Partitioning

Say we want to find SELECT(A, k)



L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

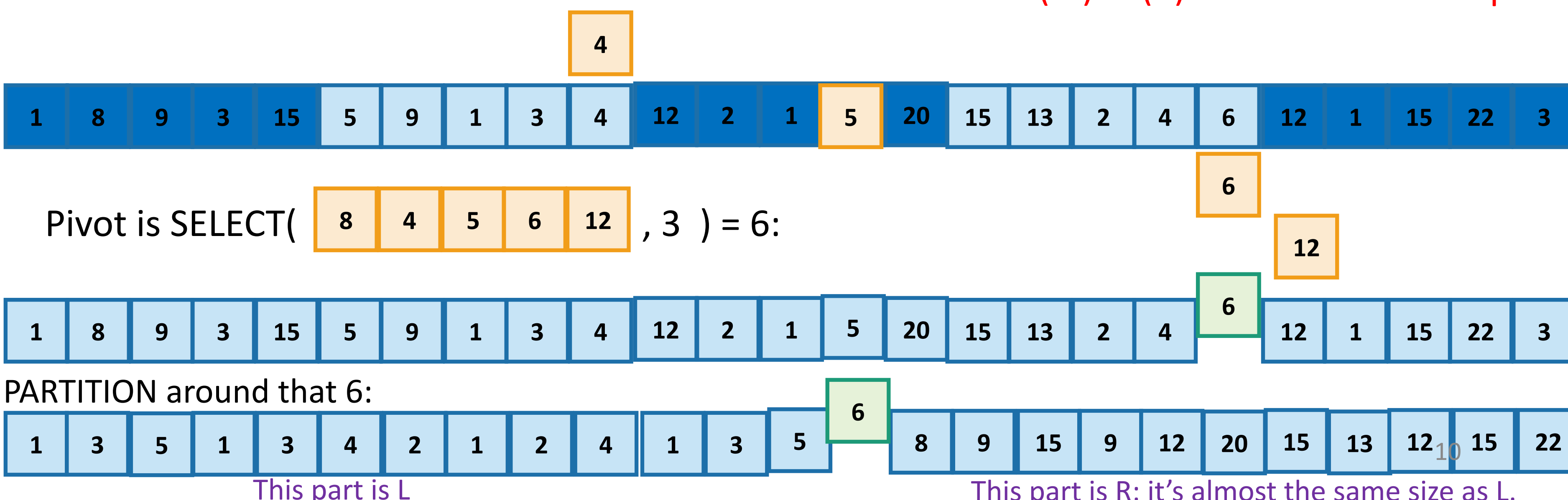
- If $k = 5 = \text{len}(L) + 1$:
  - We should return A[pivot]
- If $k < 5$:
  - We should return SELECT(L, k)
- If $k > 5$:
  - We should return SELECT(R, k – 5)

Partitioning like this takes time O(n) since we don't care about sorting each half.

Choosing the pivot

- **CHOOSEPIVOT(A):**
  - Split A into m = $\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** p

This takes time O(1), for each group, since each group has size 5. So that's O(m)=O(n) total in the for loop.



SELECT

- **`getPivot(A)`** returns some pivot for us.
  - How?? Median of sub-medians!
- **`Partition(A,p)`** splits up A into L, A[p], R.

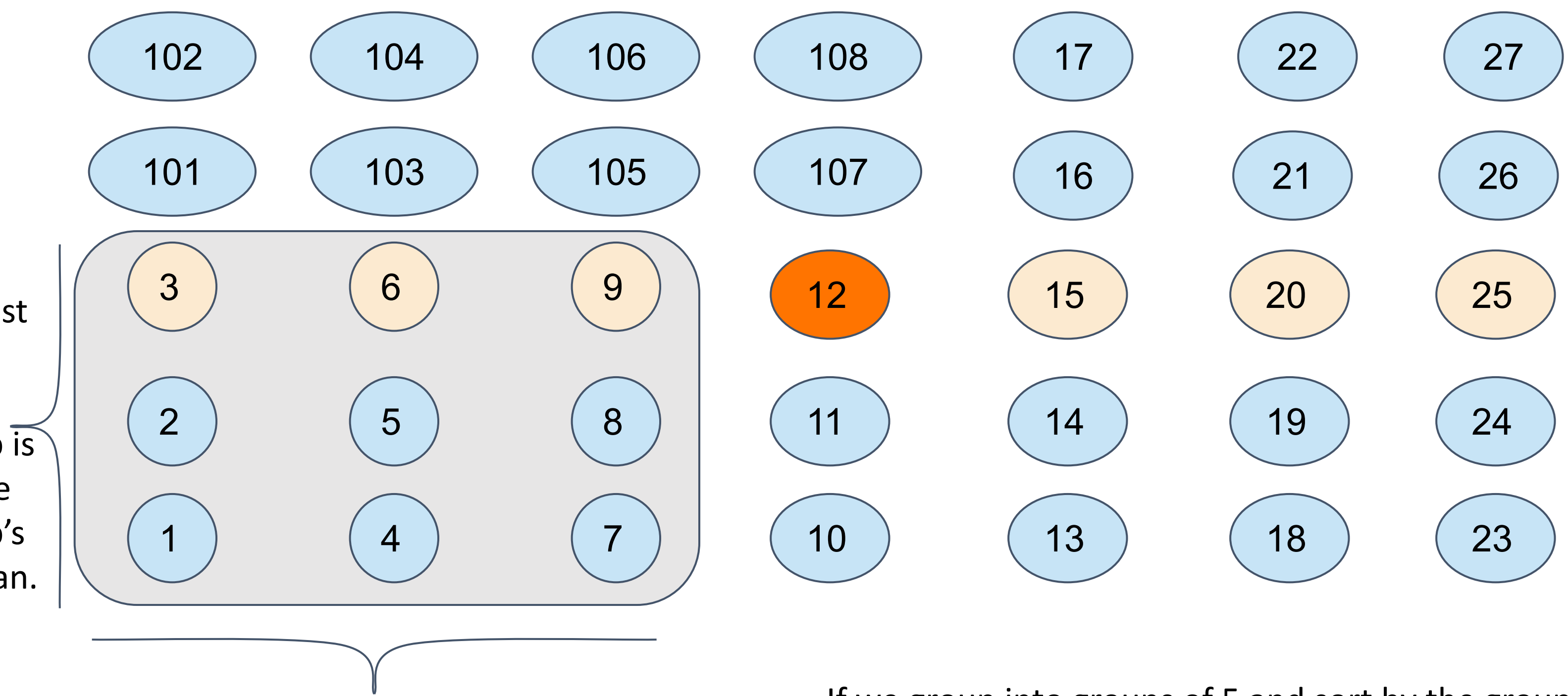
- **`Select`**(A,k):
  - **If** len(A) <= 50:
    - A = **`MergeSort`**(A)
    - **Return** A[k-1]
  - p = **`getPivot`**(A)
  - L, pivotVal, R = **`Partition`**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **`Select`**(L, k)
  - **Else if** len(L) < k-1:
    - return **`Select`**(R, k – len(L) – 1)

Running time:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \Theta(n)$$

Why 70%/30% split worst case?

The most lopsided split that can happen after partitioning around the median of medians is 70/30.



At least ⅗ of each group is <= the group's median.

At least ½ of the groups have medians <= pivot

If we group into groups of 5 and sort by the groups' medians, the gray stuff (at least ⅗ * ½ = 3/10) all lies in one side of the partition.

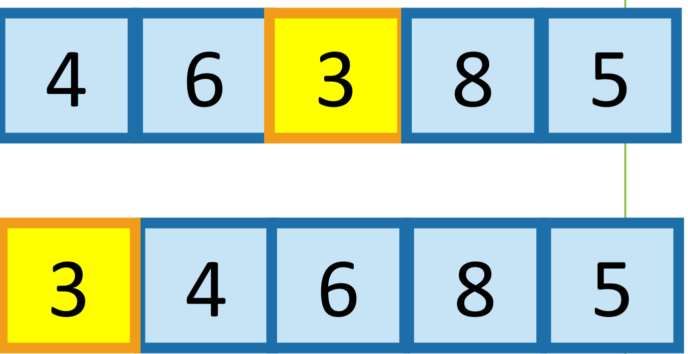
Space Complexity

Space complexity of an algorithm

- Definition: the **space complexity** of an algorithm is how much memory the algorithm needs to run, excluding the input and output.

- Expressed as a function of input size
    - Could vary based on language, compiler, etc. -> Big-O notation!

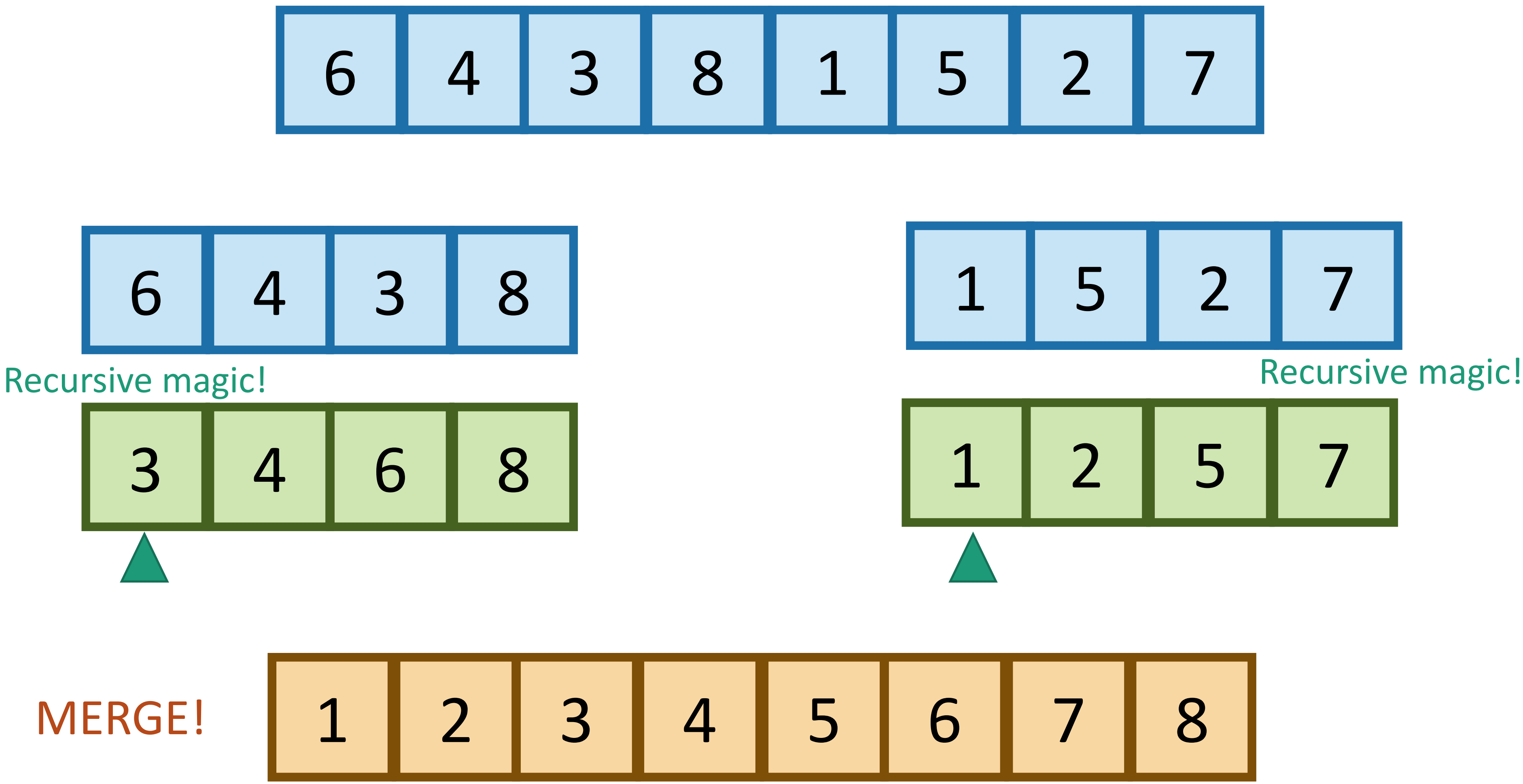
Example: Insertion Sort




```
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

- Input: array of size n
- All operations are done in-place -> no extra space needed
- Space complexity = O(1)

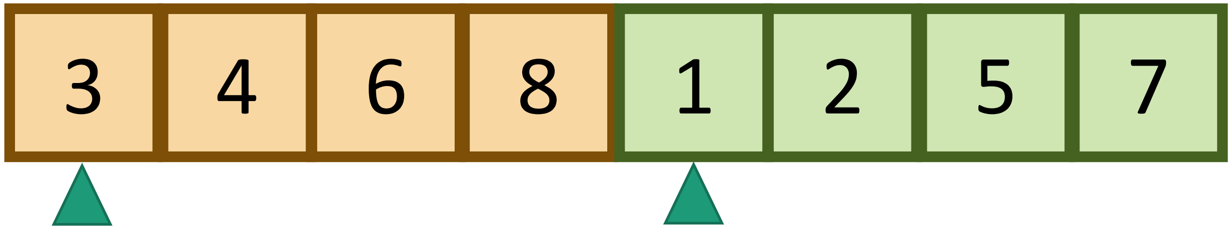
Example: Merge sort



Example: Merge sort

- Merging two arrays of size k/2 into a new array of size k requires extra space of size k
- The top level of merge sort needs space n, so merge sort has space complexity O(n)
  - Merge sort has log(n) levels of merges, why is it not n log(n)?
- **Can we do better?**

In-place merging



- If the left element is smaller, move the left pointer to the right.
- If the right element is smaller, move it to the position of the left element and shift everything in between to the right. Then move both pointers to the right.
- Now requires no extra space -> space complexity is O(1)!

In-place merging

- What happened to time complexity?
  - “Shift everything to the right” is O(n), in the worst case we need to do it O(n) times
- This merge takes time $O(n^2)$!
- Often there is a trade-off between time and space complexity.
  - In what situations is having a small space complexity more important?

Quick Sort

QuickSort

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick pivot x with **pivot**.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Rearrange A as [L, x, R]
  - QuickSort(L)
  - QuickSort(R)

Running time:

$$T(n) = T(|L|) + T(|R|) + \Theta(n)$$

$$\boldsymbol{T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)}$$

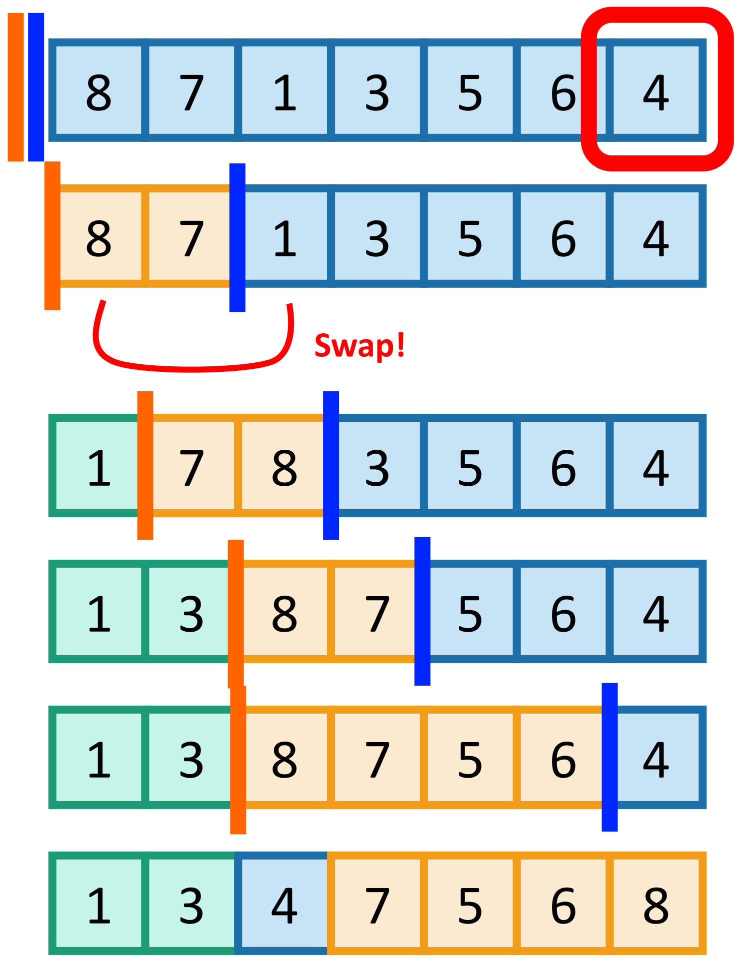
$$\boxed{\boldsymbol{T(n) = O(n\, log(n))}}$$

In-Place [O(1) memory!] Quick Sort

- Recall the Naïve memory complexity of Quick Sort is O(n logn)
  - Why? Think about storing an ordering of n elements for log(n) levels
- We can improve it to O(n)
  - Why? Can use a single array to represent the ordering and update at each level
- Can we do even better?
  - Let these happy Hungarians show you the answer!

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

A better way to do Partition



Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize and

Step forward.

When sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

Quick Sort vs Merge Sort

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	• Worst-case: $O(n^2)$ • Expected: $O(n \log(n))$	• Worst-case: $O(n \log(n))$
In-Place? (With $O(\log(n))$ extra memory)	Yes, can be implemented in-place (relatively) easily	Not as easily since you'd have to sacrifice stability and runtime, but it can be done
Stable?	No	Yes

stable sorting algorithms sort identical elements in the same order as they appear in the input

Quick Sort vs Merge Sort

Which one would you use for a small array?

Given the small size it mostly does not matter. At this scale the difference will be in the order of *ns* or *μs*, so the only way to be sure which is better is to write implementations and test it in practice. In fact, insertion sort can often be the fastest for *very* small arrays.

Which one would you use for an array with millions of elements?

Because for large *n* the Law of Large Numbers kicks in, we can reasonably expect both algorithms to run in O(n log n). It then becomes a choice between stability and memory overhead.

Which one would you use in a security-critical situation?

We value the predictability and consistency of a deterministic algorithm in these situations, so merge sort would be preferred. Some pitfalls of quicksort are:

- Randomized algorithms are harder to log, debug, and reproduce than deterministic algorithms.
- If an adversary can guess which seed you start with, they may be able to craft a worst-case n^2 input!

Thank You!