CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- **Welcome prospective students!**
- **Lab 7 posted (due next Friday)**
- **Video on (if your internet will accommodate that)**
- **Let me know if you will be missing class or lab**
- **Lab this week: same plan (will start at 9am)**
  - Remember to sign-in on the google sheet

Remind me to RECORD

REVISED TA/OFFICE HOURS

Sunday 7-9pm (Juvia)

Monday 8-midnight (Steve)

Tuesday 11:30-12:30pm (Lizzie)

Tuesday 4:30-6pm (Sara)

Wednesday 8-midnight (Steve)

Thursday 11:30-12:30pm (Lizzie)

Thursday 9-11pm (Will)

Friday 8-10pm (Gareth)

Today/Tomorrow

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

FINAL PROJECT

More details to come, but loosely a "choose your own adventure" style

1) Finish Lab 7 (with a few additional analyses)
2) Huffman encoding part of Lab 6 (worth small amount of extra credit)
3) Quadtree graphics application (worth a slightly larger amount of extra credit)

APR 16 OUTLINE

- Quick sort
- Radix sort
- Merge sort

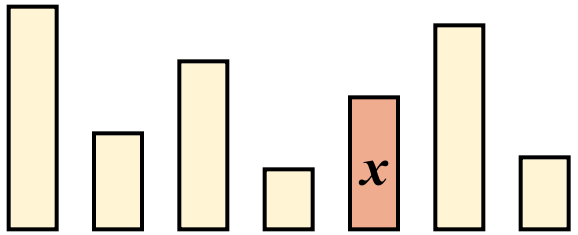
APR 16 OUTLINE

- Quick sort
- Radix sort
- Merge sort

QUICK SORT: HIGH LEVEL

A randomized sorting algorithm based on divide-and-conquer

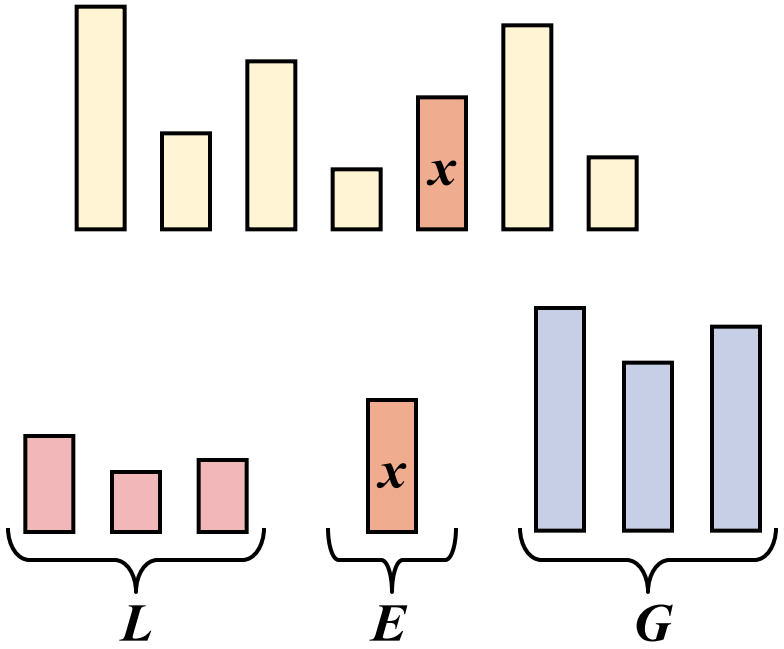
- divide: pick a random element x (pivot)



QUICK SORT: HIGH LEVEL

A randomized sorting algorithm based on divide-and-conquer

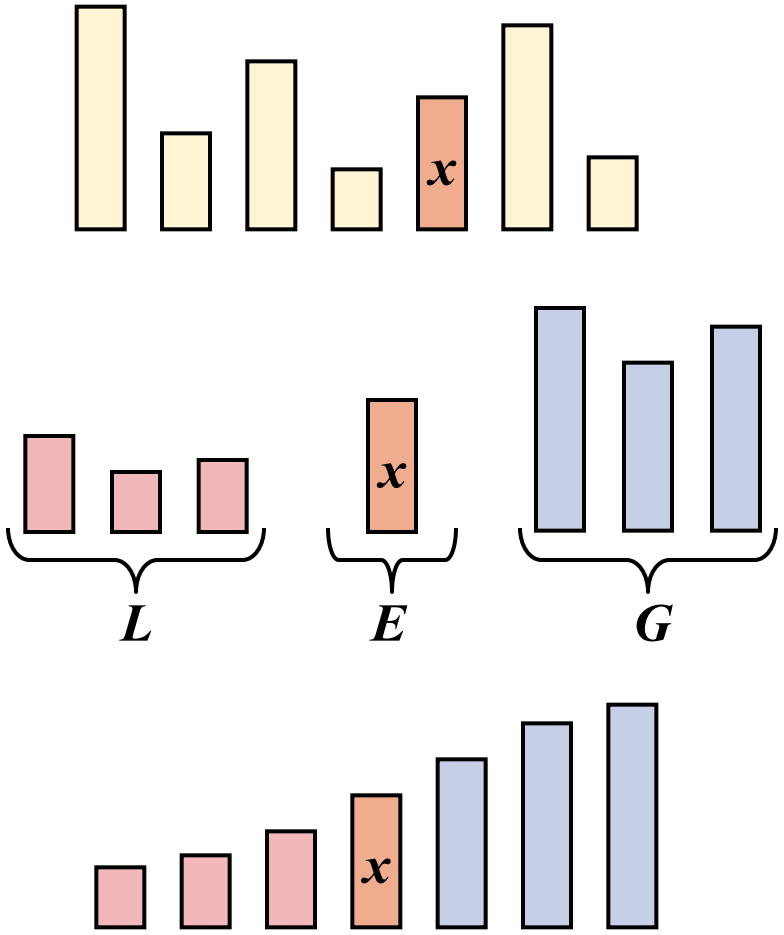
- divide: pick a random element x (pivot) and partition into
  - L: $<x$
  - E: $=x$
  - G: $>x$



QUICK SORT: HIGH LEVEL

A randomized sorting algorithm based on divide-and-conquer

- divide: pick a random element x (pivot) and partition into
  - L: $<x$
  - E: $=x$
  - G: $>x$
- conquer: sort L and G
- combine: join L, E and G



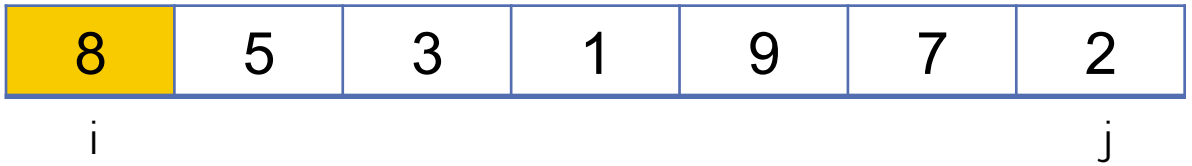
QUICK SORT: HOW TO CHOOSE THE PIVOT?

- First element
- Last element
- Random element
- Median of 3 random elements

Regardless of what we do, it is convenient to swap the pivot to the beginning or end so all elements can be moved relative to the pivot.

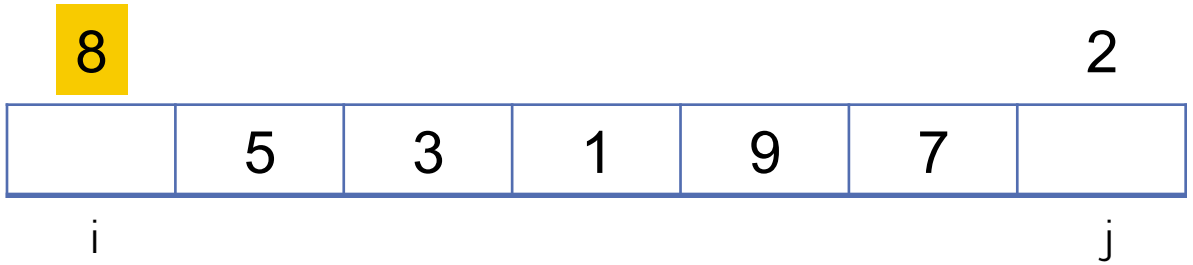
QUICKSORT EXAMPLE

8 < 2 ✖



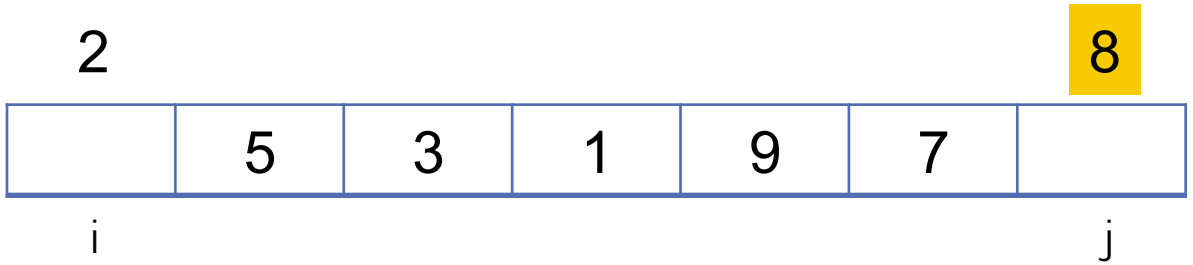
QUICKSORT EXAMPLE

8 < 2 ✖



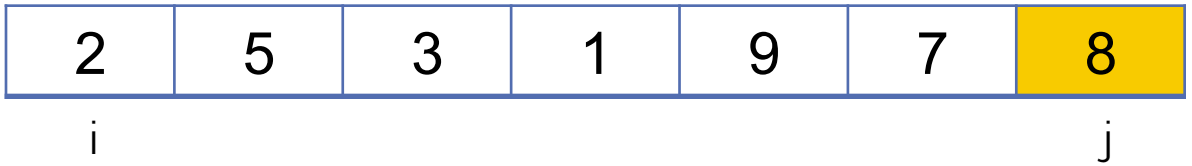
QUICKSORT EXAMPLE

8 < 2 ✖



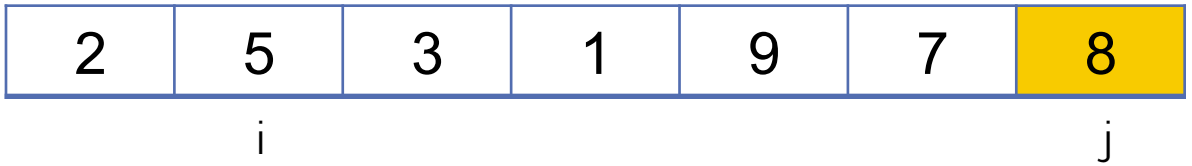
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓



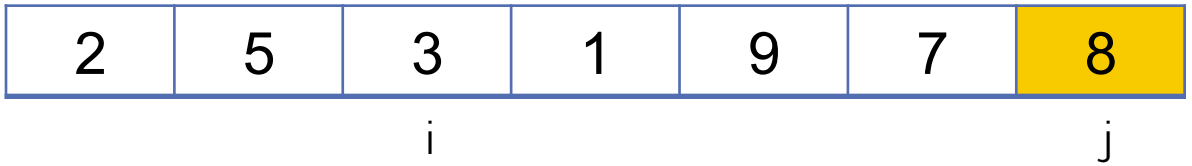
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓



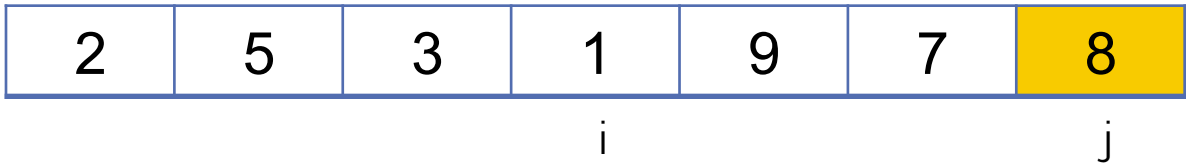
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓



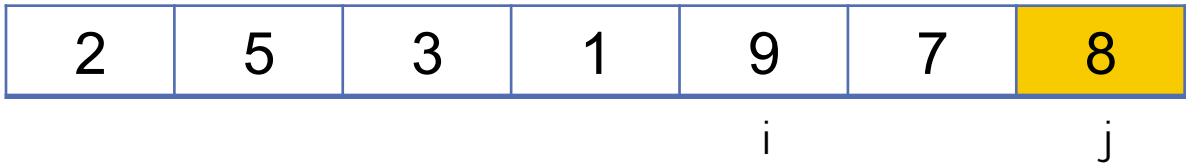
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓



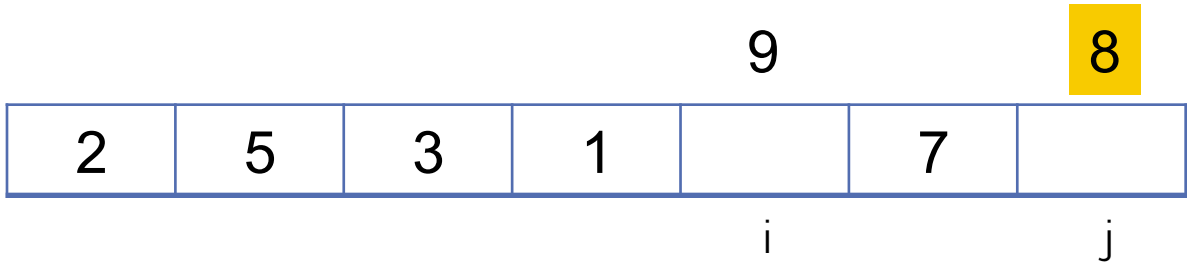
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗



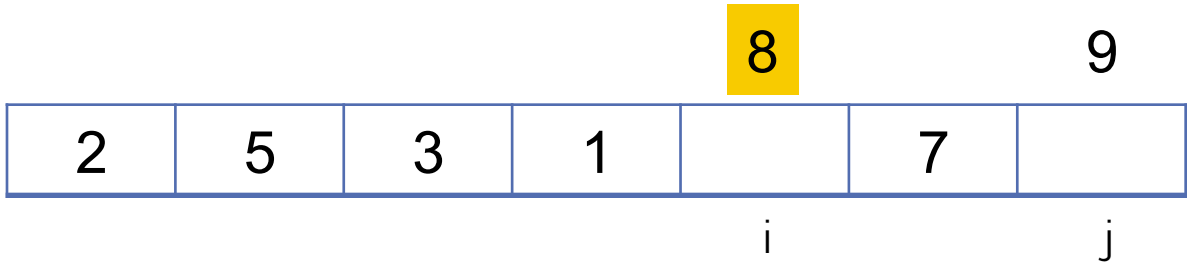
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗



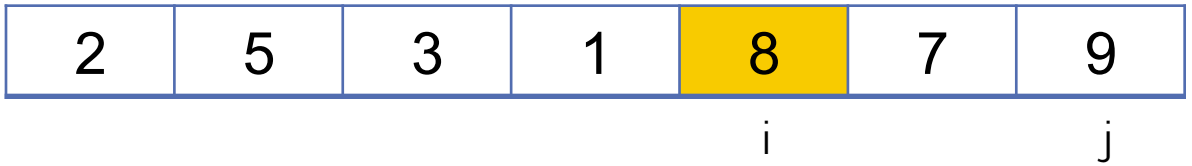
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗



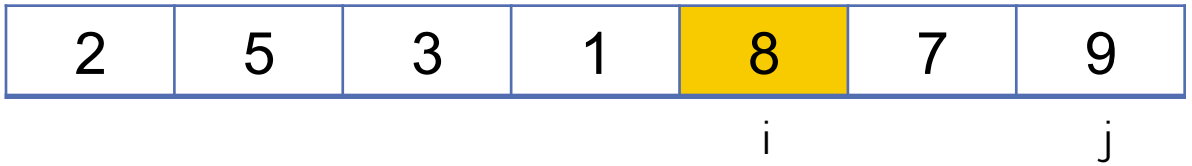
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗



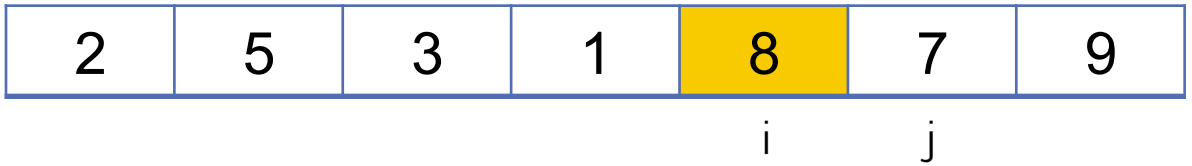
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗
8 < 9 ✓



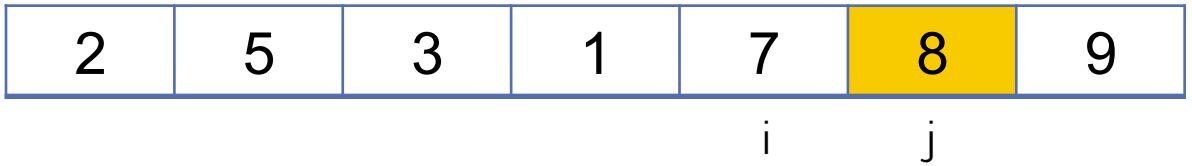
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗
8 < 9 ✓
8 < 7 ✗



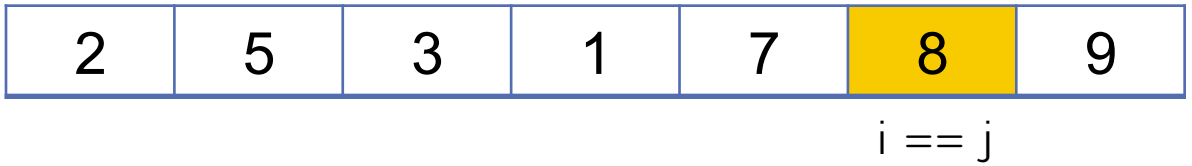
QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗
8 < 9 ✓
8 < 7 ✗

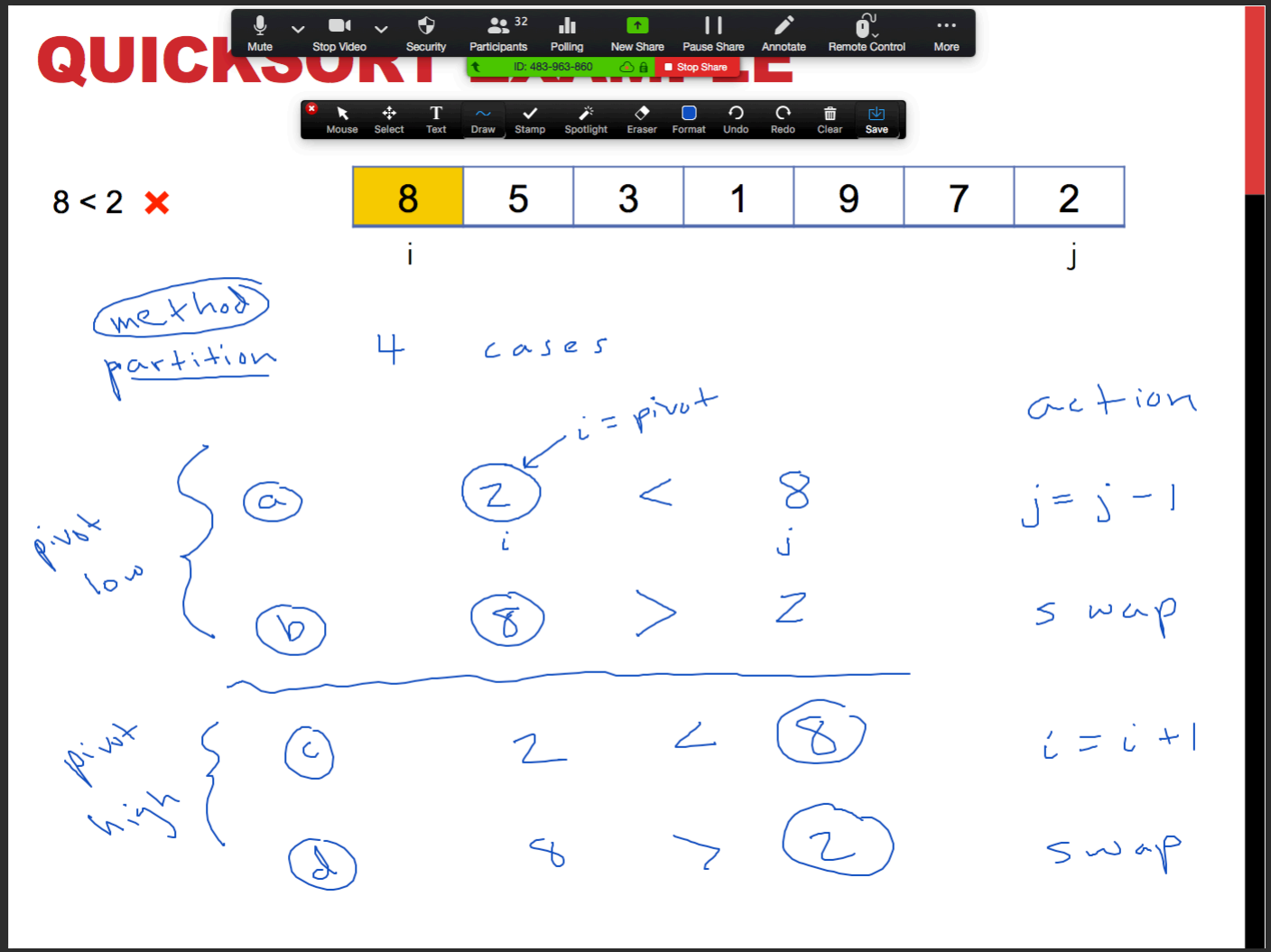


QUICKSORT EXAMPLE

8 < 2 ✗
2 < 8 ✓
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 ✗
8 < 9 ✓
8 < 7 ✗
7 < 8 ✓



IMPLEMENTING PARTITION



QUICKSORT (LAST TIME)

1. Using the first number in each (sub)array as a pivot, quicksort the following array:

$$A = \{6, 3, 5, 2, 7, 1, 9\}$$

2. Based on the example above, are some pivots better than others?

   Yes! 1 ends up being a bad pivot (in the left recursive call) because it doesn't end up in the middle.

3. Can quicksort be implemented in-place? Why or why not?

   Yes! We only rely on swaps (in-place operation) and keeping track of high and low indices.

QUICKSORT (GROUP EXERCISE)

4. Pretend we have a function called `partition(A, low, hi)` that returns the index where the pivot should go in the subarray from `A[low:hi]` (inclusive). The pivot should begin at `A[low]`. Fill in the following method (pseudocode):

```
def quicksort(A, low, hi):


    _____ = partition(            )

    quicksort(             )

    quicksort(             )
```

5. Now write pseudocode for `partition`, our helper method. What is the return type?

```
def partition(A, low, hi):
```

6. What is the runtime of quicksort, in terms of the array length n?

QUICKSORT (GROUP EXERCISE)

4. Pretend we have a function called `partition(A, low, hi)` that returns the index where the pivot should go in the subarray from `A[low:hi]` (inclusive). The pivot should begin at `A[low]`. Fill in the following method (pseudocode):

```
def quicksort(A, low, hi):
    if ??
        p = partition(A, low, hi)
        quicksort(A, low, p-1)
        quicksort(A, p+1, high)
```

pivot = 17

p = 5

low + high

meet

5. Now write pseudocode for `partition`, our helper method. What is the return type?

```
def partition(A, low, hi):
```

i, j

pivot = A[low]

main:

quicksort(A, 0, size-1)

if -

loop

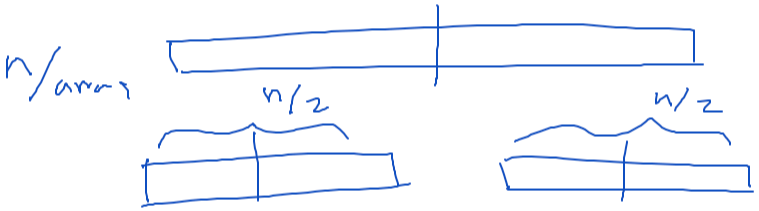
6. What is the runtime of quicksort, in terms of the array length n?

EXPECTED RUNTIME

QUICKSORT (GROUP EXERCISE)

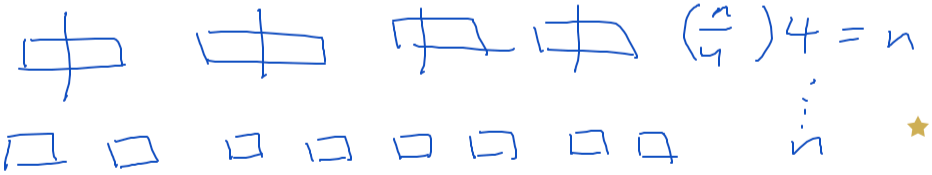
6. Runtime : $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$

best / expected



partition
$O(n)$

$\frac{n}{2} + \frac{n}{2} = n$



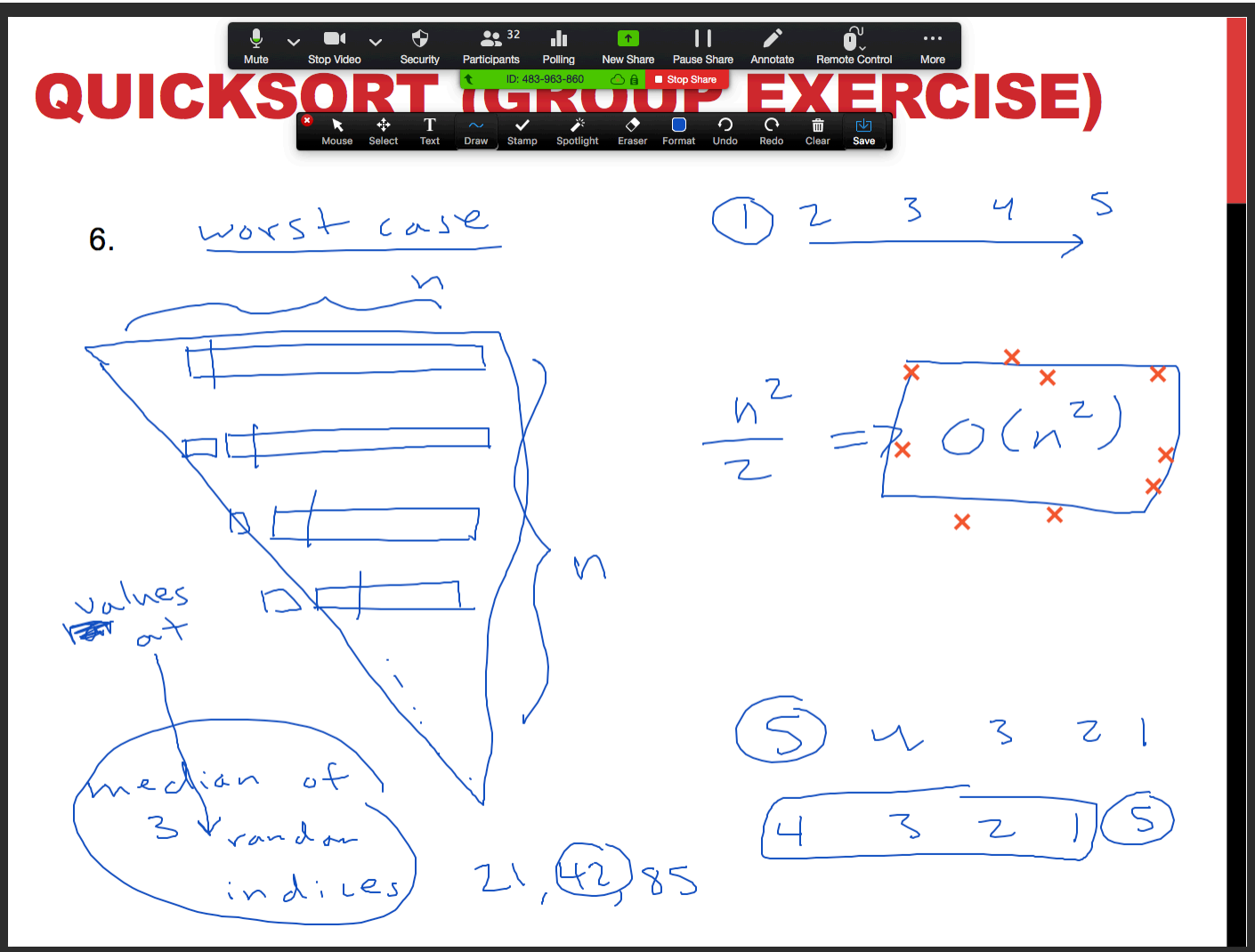
overall
$n \log(n)$

levels
x

$\frac{n}{2^x} = 1 \Rightarrow x = \log(n)$

WORST-CASE RUNTIME

QUICKSORT (GROUP EXERCISE)



QUICKSORT – WORST CASE COMPLEXITY

What's the worst case for quicksort? I.e., when will it be forced to do a lot of comparisons?

When the pivot item picked in each round is the smallest (or largest) item, then we have to do n (n – 1) / 2 comparisons.

The good news is: if the pivot is picked randomly, this is unlikely to happen!

The expected case analysis of this algorithm turns out to be O(n log n) !

DIVIDE-AND-CONQUER

Divide **– the problem (input) into smaller pieces**

Conquer **– solve each piece individually, usually recursively**

Combine **– the piecewise solutions into a global solution**

Usually involves recursion

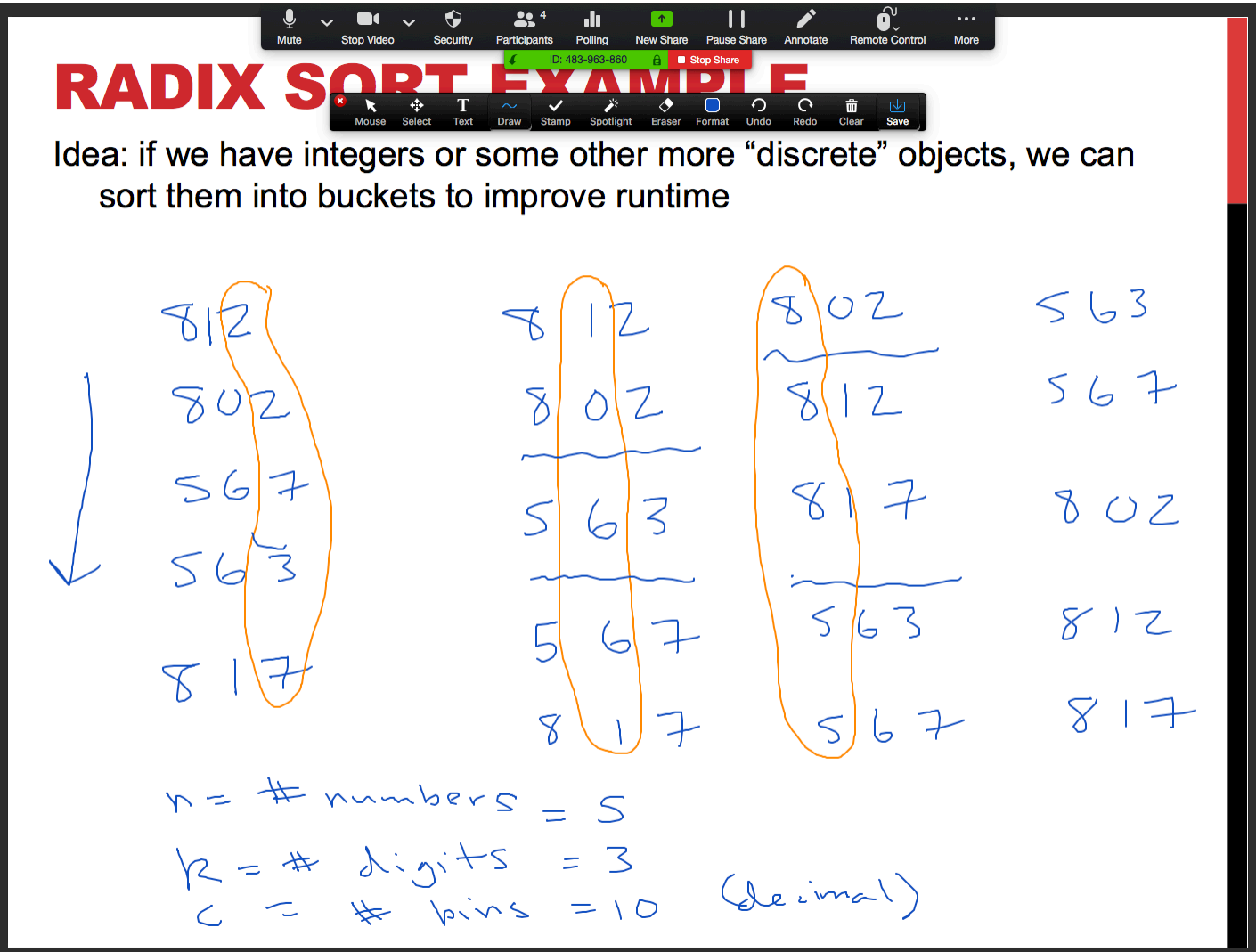
Analysis usually involves solving recurrence relations

APR 16 OUTLINE

- Quick sort
- **Radix sort**
- Merge sort

RADIX SORT EXAMPLE

Idea: if we have integers or some other more “discrete” objects, we can sort them into buckets to improve runtime



Radix sort columns (high to low here)

1	2	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
0	1	0	0	0

Radix sort columns (high to low here)

1	2	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
0	1	0	0	0

2	1	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
1	0	0	0	0

Radix sort columns (high to low here)

1	2	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
0	1	0	0	0

2	1	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
1	0	0	0	0

3	4	2	1	5
0	0	1	1	0
1	0	0	0	0
0	0	1	1	1
1	1	0	0	0
0	0	1	0	0

Radix sort columns (high to low here)

1	2	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
0	1	0	0	0

2	1	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
1	0	0	0	0

3	4	2	1	5
0	0	1	1	0
1	0	0	0	0
0	0	1	1	1
1	1	0	0	0
0	0	1	0	0

2	1	5	3	4
1	1	0	0	0
0	0	0	1	0
1	1	1	0	0
0	0	0	1	1
1	0	0	0	0

Radix sort columns (high to low here)

1	2	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
0	1	0	0	0

2	1	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
1	0	0	0	0

3	4	2	1	5
0	0	1	1	0
1	0	0	0	0
0	0	1	1	1
1	1	0	0	0
0	0	1	0	0

2	1	5	3	4
1	1	0	0	0
0	0	0	1	0
1	1	1	0	0
0	0	0	1	1
1	0	0	0	0

3	2	1	5	4
0	1	1	0	0
1	0	0	0	0
0	1	1	1	0
1	0	0	0	1
0	1	0	0	0

Radix sort columns (high to low here)

1	2	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
0	1	0	0	0

2	1	3	4	5
1	1	0	0	0
0	0	1	0	0
1	1	0	0	1
0	0	1	1	0
1	0	0	0	0

3	4	2	1	5
0	0	1	1	0
1	0	0	0	0
0	0	1	1	1
1	1	0	0	0
0	0	1	0	0

2	1	5	3	4
1	1	0	0	0
0	0	0	1	0
1	1	1	0	0
0	0	0	1	1
1	0	0	0	0

3	2	1	5	4
0	1	1	0	0
1	0	0	0	0
0	1	1	1	0
1	0	0	0	1
0	1	0	0	0

2	1	3	5	4
1	1	0	0	0
0	0	1	0	0
1	1	0	1	0
0	0	1	0	1
1	0	0	0	0

RADIX SORT RUNTIME (EXERCISE)

n = number of numbers we want to sort

k = number of digits in each number

c = number of bins (2 for binary, 10 for decimal, etc)

APR 16 OUTLINE

- Quick sort
- Radix sort
- **Merge sort**

Next time!