Lecture 4:
Randomized Algs. & QuickSort

July 1, 2020

ANNOUNCEMENTS

- Homework 1 was due ~30 minutes ago!
  - Don’t forget that we have a late policy!
- Homework 2 is out & due next Wednesday 1pm!
- We’ve got another CA: Bryce Cai!
- We’re upgrading our Homework Parties with Nooks!

NOOKS

- A video conferencing platform (built by a Stanford CS undergrad!)
- Great for simulating large group discussion
- We’ll have an overall “room”, separate “rooms” for each homework problem (think of them as different tables you could sit at if we were meeting IRL), and probably some miscellaneous rooms too
- You can see who’s in each room, which rooms have a CA in it, chat with the people in the room, “whisper” to a friend in the room (i.e. private voice channel), & more!
- Our CS 161 Nooks will always be available, so you can join outside of Homework Parties (organize with your peers on Slack)! Our scheduled Homework Parties are times when it’s guaranteed that your peers and a CA will be present!

LAST TIME

- The Substitution Method!
- A linear-time algorithm for SELECT
- All purely deterministic (i.e. no randomness involved) algorithms & analysis!

WHAT WE’LL COVER TODAY

- Quick highlight reel from yesterday’s SELECT!
- What is a Randomized Algorithm? How could we analyze them?
- Examples of randomized algorithms and analyzing them!
    - BogoSort & **QuickSort**!

HIGHLIGHTS OF SELECT

We covered a lot of details - here are the big picture takeaways.

LINEAR SELECTION: THE BIG IDEA

Select a pivot: **Median of Medians**

Partition around pivot

Recurse!

LINEAR SELECTION: RUNTIME

Select a pivot: **Median of (sub)Medians**

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

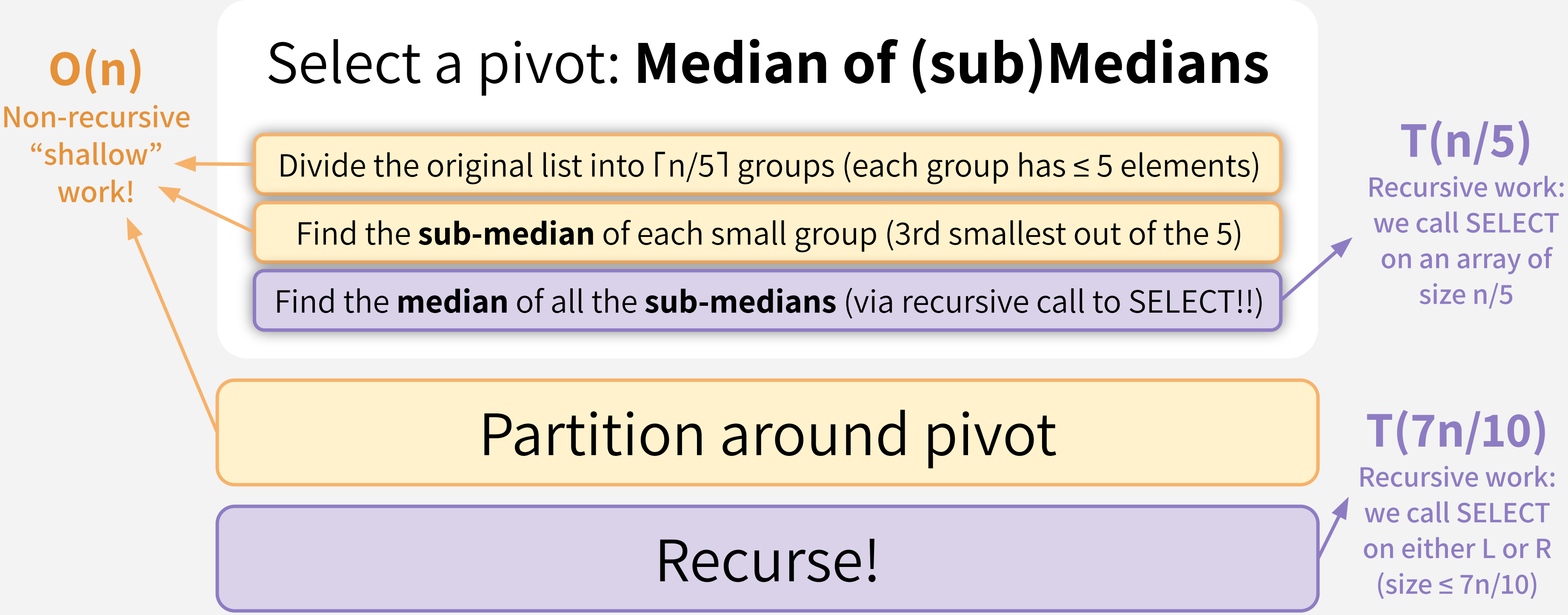
Find the **sub-median** of each small group (3rd smallest out of the 5)

Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

Partition around pivot

Recurse!

LINEAR SELECTION: RUNTIME



Select a pivot: **Median of (sub)Medians**

Divide the original list into $\lceil n/5 \rceil$ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

Partition around pivot

Recurse!

O(n)

Non-recursive "shallow" work!

T(n/5)

Recursive work: we call SELECT on an array of size n/5

T(7n/10)

Recursive work: we call SELECT on either L or R (size ≤ 7n/10)

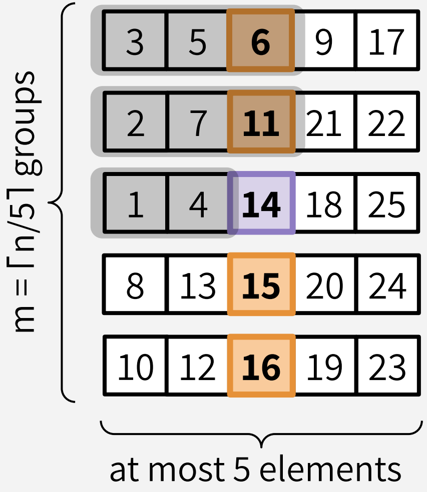
WAIT: WHERE DID WE GET 7n/10?

At the end of last lecture, we proved this claim:

$$3n/10 - 6 \le \texttt{len(L)} \le 7n/10 + 5$$

$$3n/10 - 6 \le \texttt{len(R)} \le 7n/10 + 5$$

this is because
`len(L)` + `len(R)` = n-1,
so if
3n/10 - 6 ≤ `len(L)`
then
`len(R)` **≤ 7n/10 + 5**



3	5	**6**	9	17
2	7	**11**	21	22
1	4	**14**	18	25
8	13	**15**	20	24
10	12	**16**	19	23

We asked ourselves:
At least how many elements are guaranteed to be smaller than the median of medians?

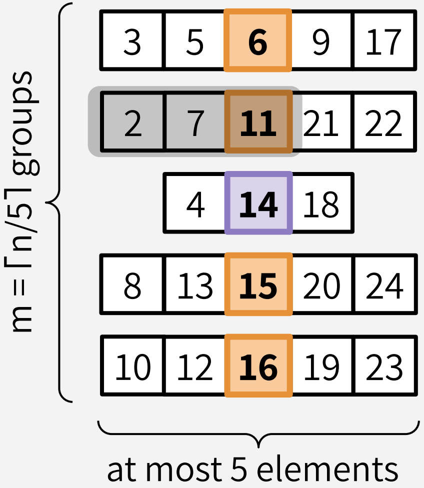
The shaded region denotes the only elements that are *guaranteed* to be smaller than **14** (the median of medians). We counted that up, took care of some off-by-one errors just to be safe (i.e. just to make sure we're underestimating), and we got **3n/10 - 6**!

(DETAILS IF YOU'RE CURIOUS)

At the end of last lecture, we proved this claim:

$$3n/10 - 6 \leq \texttt{len(L)} \leq 7n/10 + 5$$

$$3n/10 - 6 \leq \texttt{len(R)} \leq 7n/10 + 5$$



3	5	**6**	9	17
2	7	**11**	21	22
	4	**14**	18	
8	13	**15**	20	24
10	12	**16**	19	23

3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians** + ~~2 elements from the group containing the **median of medians**~~

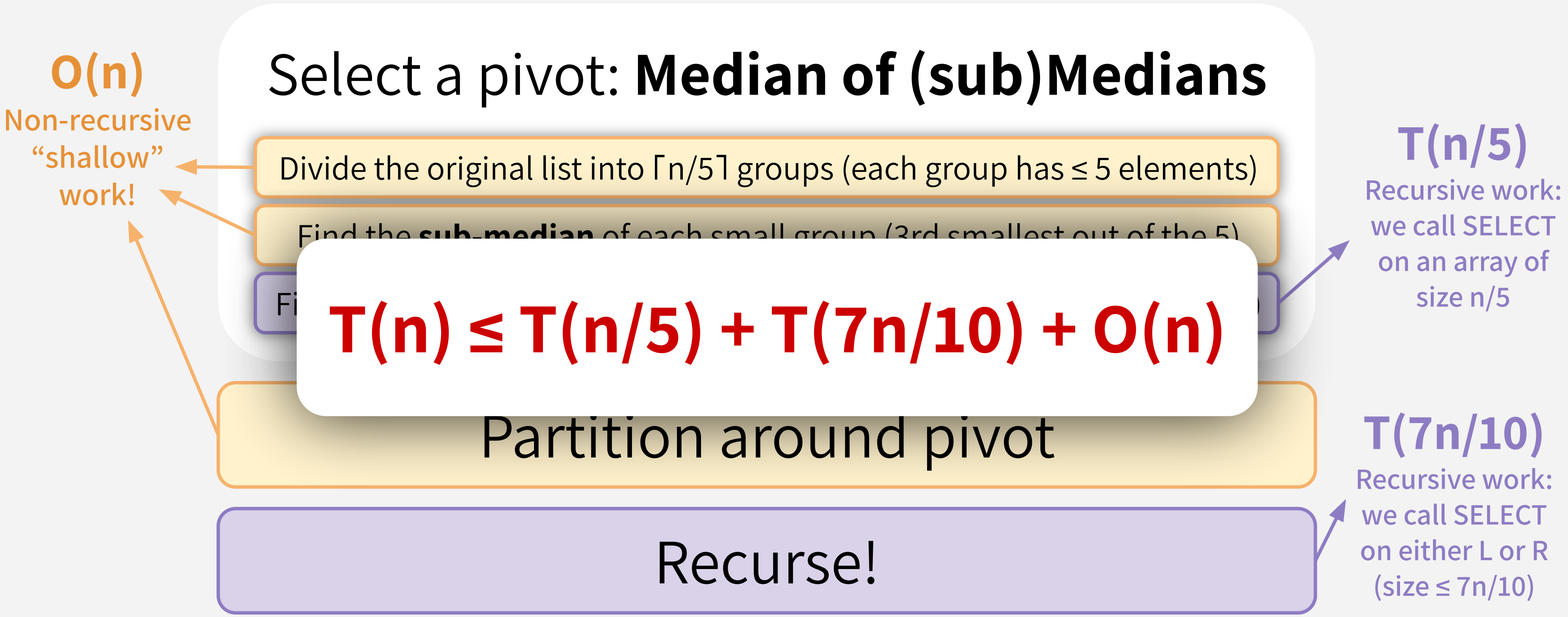
$$3 \cdot (\lceil m/2 \rceil - 1 - 1) \cancel{+ 2}$$

To exclude the group with the **median of medians**

To exclude any of those groups that might be a "leftover" group!

The group with the **median of medians** might be a "leftover" group! Might as well just get rid of the +2 to be safe

LINEAR SELECTION: RUNTIME



O(n)

Non-recursive "shallow" work!

Select a pivot: **Median of (sub)Medians**

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Fi

$$T(n) \le T(n/5) + T(7n/10) + O(n)$$

Partition around pivot

Recurse!

T(n/5)

Recursive work: we call SELECT on an array of size n/5

T(7n/10)

Recursive work: we call SELECT on either L or R (size ≤ 7n/10)

LINEAR SELECTION: RUNTIME

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

We solved this recurrence using the Substitution Method at the start of last class!

⬇

O(n)

Worst-case Runtime!

PSEUDOCODE & RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else if len(L) < k-1:
        return SELECT(R, k-len(L)-1)
```

O(n) work outside of recursive calls

(base case, set-up within MEDIAN_OF_MEDIANS, partitioning)

T(n/5) work hidden in this recursive call

(remember, MEDIAN_OF_MEDIANS calls SELECT on ⌈n/5⌉-size array)

T(7n/10) work hidden in this recursive call

7n/10 is the maximum size of either L or R (this is what the median-of-medians technique guarantees us)!

LINEAR SELECTION: THE BIG IDEA

Select a pivot: **Median of Medians**

Partition around pivot

Recurse!

Median of Medians is really cool! The math was a little detailed, but worth the time to digest so that you're 110% convinced that the technique does give a ~7n/10 bound on the max size of either L or R. Solving the recurrence can be done via Substitution Method. SELECT as a whole is an amazing display of Divide-and-Conquer!

RANDOMIZED ALGORITHMS

What are randomized algorithms? And how are they analyzed?

WHAT IS A RANDOMIZED ALGORITHM?

- An algorithm that incorporates randomness as part of its operation.
- Basically, we’ll make random choices during the algorithm:
  - Sometimes, we’ll just hope that it works!
  - Other times, we’ll just hope that our algorithm is fast!
- Let’s formalize this...

LAS VEGAS vs. MONTE CARLO

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

We'll focus on these algorithms today
(BogoSort, QuickSort, QuickSelect)

MONTE CARLO ALGORITHMS

Correctness is a random variable.
(i.e. there's a chance the output is wrong)

But the runtime is guaranteed!

We'll see some examples of these later in the quarter!

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input,
then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input,
then *the bad guy chooses the randomness* (“fixes the dice”) in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

~~“Expected value over possible inputs”~~

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

In both cases, we are still thinking about the *WORST-CASE INPUT*

~~“The worst possible input”~~

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* (“fixes the dice”) in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

“Expected value over *dice outcomes*” ✓

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and <u>a bad guy picks the input</u>, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

In both cases, we are still thinking about the *WORST-CASE INPUT*

“The worst possible *dice outcomes*” ✓

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and <u>a bad guy picks the input</u>, then *the bad guy chooses the randomness* (“fixes the dice”) in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

“Expected
val

EXPECTED RUNNING TIME

Don’t get confused!!!

Even with randomized algorithms, we are still considering the *WORST CASE INPUT*, regardless of whether we’re computing expected or worst-case runtime.

Expected runtime ***IS NOT*** runtime when given an expected input! We are taking the expectation over the random choices that our algorithm would make, ***NOT*** an expectation over the distribution of possible inputs.

p

make our algorithm suffer the most), so we can reason about the ***worst-case running time***

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

$$\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

By linearity of expectation: $\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{n}{100}$

c. Suppose I draw independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, and I stop when I see the first "**1**". Let N be the last index that we draw. What is the expected value of N?

N is a *geometric random variable* (from CS109). We can use the formula: $\mathbb{E}[N] = \frac{1}{p} = \frac{1}{1/100} = 100$

GEOMETRIC RANDOM VARIABLE

If **N** represents "number of trials/attempts",
and **p** is the probability of "success" on each trial, then:

$$\mathbb{E}[N] = \frac{1}{p}$$

$$\begin{aligned}\mathbb{E}[N] &= 1(p) + (1 + \mathbb{E}[N])(1-p) \\ &= p + (1-p) + (1-p)\mathbb{E}[N] \\ &= 1 + (1-p)\mathbb{E}[N]\end{aligned}$$

$$\begin{aligned}\mathbb{E}[N](1-(1-p)) &= 1 \\ \mathbb{E}[N](p) &= 1 \\ \mathbb{E}[N] &= \frac{1}{p}\end{aligned}$$

BOGOSORT

A bit silly, but a great pedagogical tool!

BOGOSORT

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
            if A[i] > A[i+1]:
                sorted = False
        if sorted:
            return A
```

This randomly permutes A
(assume it takes O(n) time)

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

Probability that $\mathbf{X_i = 1}$ (A is sorted) = **1/n!**

since there are n! possible orderings of A and only one is sorted (assume A has distinct elements) ⇒ $E[X_i] = 1/n!$

E[# of iterations/trials] = 1/(prob. of success on each trial)

= 1/(1/n!) = **n!**

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

E[runtime on a list of length n]

= **E**[(# of iterations) * (time per iteration)]

= (time per iteration) * **E**[# of iterations]

= O(n) * **E**[# of iterations]

= O(n) * (n!)

= O(n * n!)

= ***REALLY REALLY BIG***

BOGOSORT: WORST-CASE RUNTIME?

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

Worst-case runtime =

$$\infty$$

This is as if the “bad guy” chooses all the randomness in the algorithm, so each shuffle could be unlucky… forever...

WHAT HAVE WE LEARNED?

EXPECTED RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. You get to roll the dice (leave it up to randomness)

WORST-CASE RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. Bad guy “rolls” the dice (will choose the randomness in the worst way possible)

Don’t use BogoSort.

5-MINUTE BREAK

Stay hydrated, stretch, ask questions, etc.

QUICKSORT

A much better randomized algorithm

QUICKSORT OVERVIEW

EXPECTED RUNNING TIME

O (n log n)

WORST-CASE RUNNING TIME

O (n^2)

In practice, it works great! It's competitive with MergeSort (& often better in some contexts!), and it runs *in place* (no need for lots of additional memory)

QUICKSORT: THE IDEA

Let's use DIVIDE-and-CONQUER again!

Select a pivot *at random*

Partition around it

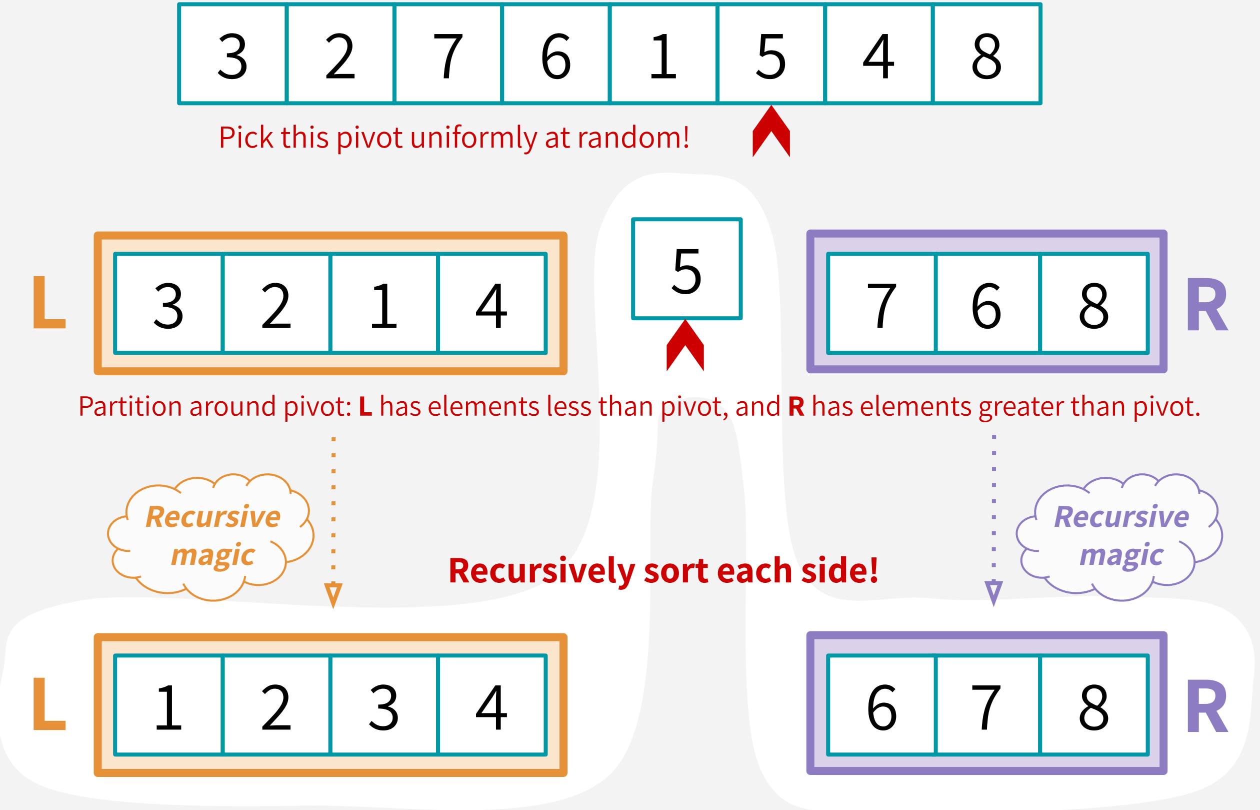
Recursively sort L and R!

QUICKSORT: THE IDEA

Select a pivot

Partition around it

Recurse!



3	2	7	6	1	5	4	8

Pick this pivot uniformly at random!

L

3	2	1	4

5

7	6	8

R

Partition around pivot: **L** has elements less than pivot, and **R** has elements greater than pivot.

Recursively sort each side!

L

1	2	3	4

6	7	8

R

QUICKSORT: PSEUDO-PSEUDOCODE

Here’s the high level outline:

(I’ve posted an IPython Notebook on the course website with actual code for QuickSort)

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

IDEAL RUNTIME?

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$

$$T(0) = T(1) = O(1)$$

In an ideal world, the pivot would split the array exactly in half, and we'd get:

$$T(n) = T(n/2) + T(n/2) + O(n)$$

IDEAL RUNTIME?

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random
    PARTITION A in
        L (less th
        R (greater
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$) + T(|R|) + O(n)$

$T(1) = O(1)$

, the pivot would split the array exactly in half, and we'd get:

$$T(n) = T(n/2) + T(n/2) + O(n)$$

In an ideal world:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

$$T(n) = \mathbf{O(n \log n)}$$

WORST-CASE RUNTIME

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$

$$T(0) = T(1) = O(1)$$

With the unluckiest randomness, the pivot would be either min(A) or max(A):

$$T(n) = T(0) + T(n-1) + O(n)$$

WORST-CASE RUNTIME

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = ra
    PARTITION 
        L (less
        R (grea
    Replace A w
    QUICKSORT(L)
    QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$T(|R|) + O(n)$

$) = O(1)$

ndomness, the pivot

min(A) or max(A):

$$T(n) = T(0) + T(n-1) + O(n)$$

With the worst “randomness”

$$T(n) = T(n-1) + O(n)$$

$$T(n) = \mathbf{O(n^2)}$$

(recursion tree/table or substitution method!)

EXPECTED RUNTIME = O(n log n)

AN **INCORRECT** PROOF:

- $E[\,|L|\,] = E[\,|R|\,] = (n - 1)/2$

AN ASIDE: why is E[|L|] = (n-1)/2?

$$E[\,|L|\,] = E[\,|R|\,]$$

(by symmetry)

$$E[\,|L| + |R|\,] = n - 1$$

(because L and R make up everything except the pivot)

$$E[\,|L|\,] + E[\,|R|\,] = n - 1$$

(by linearity of expectation)

$$2 \cdot E[\,|L|\,] = n - 1$$

(plugging the first line)

$$E[\,|L|\,] = (n - 1)/2$$

(Solving for E[|L|])

EXPECTED RUNTIME = O(n log n)

AN **INCORRECT** PROOF:

- $E[\,|L|\,] = E[\,|R|\,] = (n - 1)/2$
- If this occurs, then $T(n) = T(\,|L|\,) + T(\,|R|\,) + O(n)$ could be written as $T(n) = 2T(n/2) + O(n)$.
- Therefore, the expected running time is $O(n \log n)$!

Why is this wrong?

Well, for starters, we can use the exact same argument to prove something false...

SLOWSORT

```
SLOWSORT(A):
    if len(A) <= 1:
        return                randomly choose either!
    pivot = either max(A) or min(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    SLOWSORT(L)
    SLOWSORT(R)
```

Recurrence Relation for SLOW SORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

Same recurrence relation!

We also still have:

$E[|L|] = E[|R|] = (n-1)/2$

But now, one of **$|L|$** or **$|R|$** is *always* n-1
& the runtime is **$\Theta(n^2)$**, with probability 1

SLOWSORT

```
SLOWSORT(A):
    if len(A)
        retur
    pivot = e
    PARTITION
        L (les
        R (gre
    Replace A
    SLOWSORT(
    SLOWSORT(R)
```

Recurrence Relation for

SORT

T(|R|) + O(n)

) = O(1)

nce relation!

ll have:

] = (n-1)/2

|L| or **|R|** is *always* n-1

& the runtime is **Θ(n²)**, with probability 1

RED FLAG:

We could use the exact same (incorrect) proof to prove that **SLOWSort** has expected runtime **O(n log n)**, when it actually has expected runtime of **Θ(n²)**...

EXPECTED RUNTIME = O(n log n)

AN **INCORRECT** PROOF:

- E[|L|] = E[|R|] = (n - 1)/2
- If this occurs, then T(n) = T(|L|) + T(|R|) + O(n) could be written as T(n) = 2T(n/2) + O(n).
- Therefore, the expected running time is O(n log n)!

Why is this wrong?

EXPECTED RUNTIME = O(n log n)

AN

Basically:

E[f(x)] is *not necessarily* the same as f(**E[**x**]**)

e.g. $E[X^2]$ is not the same as $(E[X])^2$

We were reasoning about T(**E[**x**]**) instead of **E[**T(x)**]**

Why is this wrong?

EXPECTED RUNTIME = O(n log n)

Instead, to prove that the expected runtime of QuickSort is O(n log n), we're going to count the **number of comparisons** that this algorithm performs, and take the expectation of that!

How many times are any two items compared?

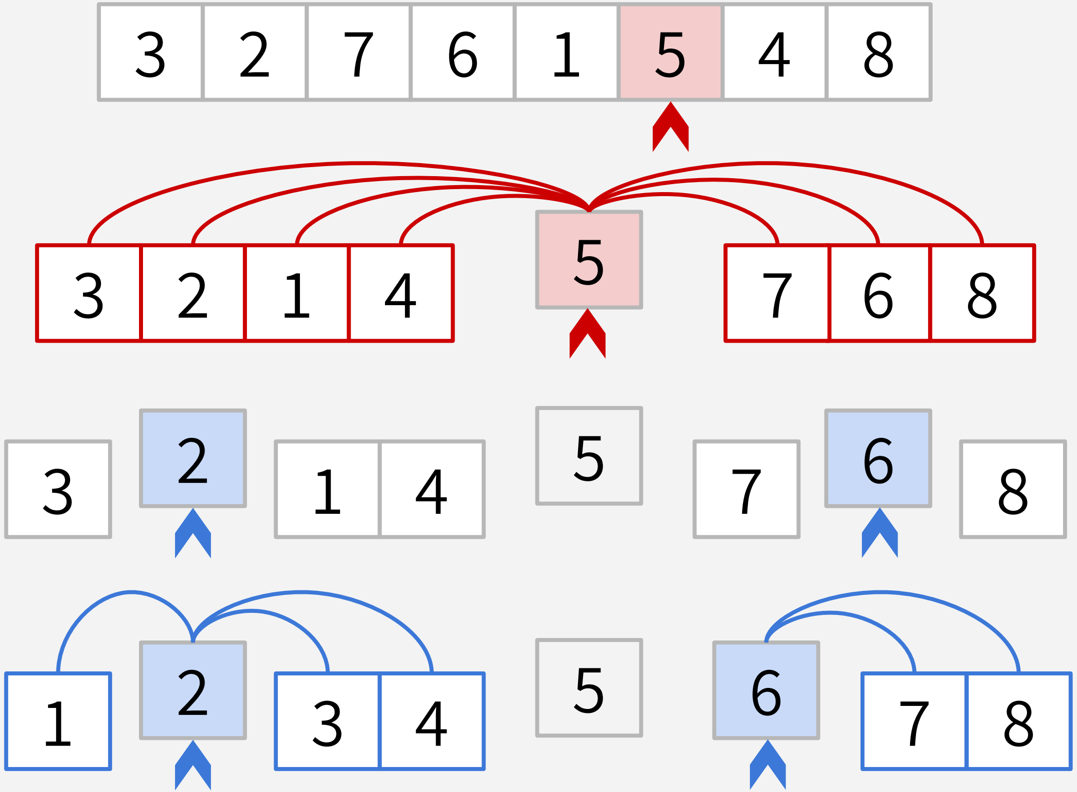
3-MINUTE BREAK

Stay hydrated, stretch, ask questions, etc.

QUICKSORT O(n log n) EXPECTED RUNTIME

The correct way to prove this expected runtime:
How many times are any two items compared, in expectation?

HOW MANY COMPARISONS?



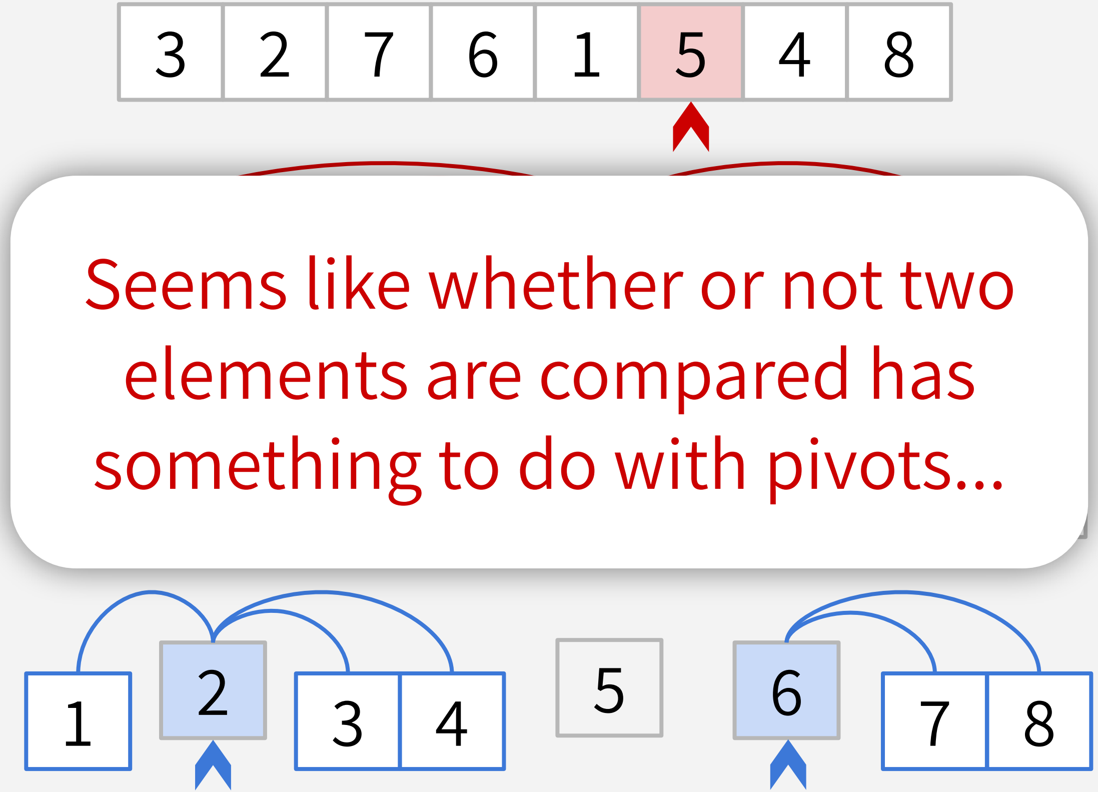
Everything is compared to 5 once in this first step… and then never again with **5**.

Only 1, 3, & 4 are compared to **2**.

And only 7 & 8 are compared with **6**.

No comparisons ever happen between two numbers on opposite sides of 5.

HOW MANY COMPARISONS?



Seems like whether or not two elements are compared has something to do with pivots...

Everything is compared to 5 once in this first step… and then never again with **5**.

Only 1, 3, & 4 are compared to **2**.

And only 7 & 8 are compared with **6**.

No comparisons ever happen between two numbers on opposite sides of 5.

HOW MANY COMPARISONS?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b} = 1}$ if **a** and **b** are compared

$\mathbf{X_{a,b} = 0}$ otherwise

In our example, $\mathbf{X_{2,5}}$ took on the value **1** since **2** and **5** were compared.
On the other hand, $\mathbf{X_{3,7}}$ took on the value **0** since **3** and **7** are *not* compared.

Total number of comparisons =

$$\mathbb{E}\left[\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} X_{a,b}\right] \underset{\text{by linearity of expectation!}}{=} \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} \mathbb{E}\left[X_{a,b}\right]$$

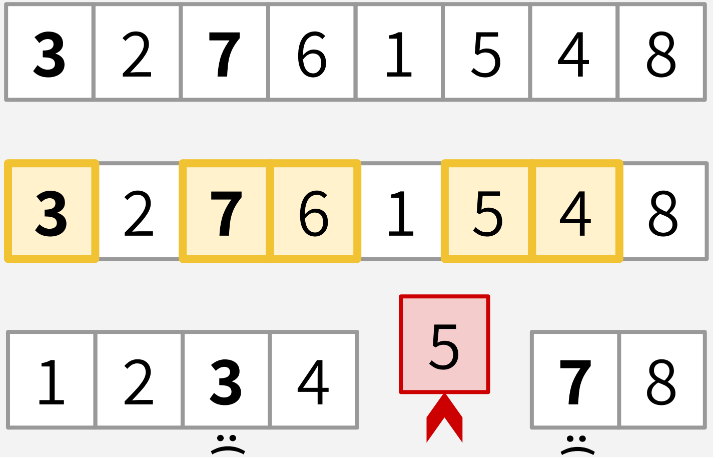
We need to figure out this value!

HOW MANY COMPARISONS?

So, what's $\mathbf{E[X_{a,b}]}$?

$$E[X_{a,b}] = 1 \cdot P(X_{a,b} = 1) + 0 \cdot P(X_{a,b} = 0) = P(X_{a,b} = 1)$$

So, what's $\mathbf{P(X_{a,b} = 1)}$? It's the probability that **a** and **b** are compared. Consider this example:



3	2	**7**	6	1	5	4	8

3	2	**7**	6	1	5	4	8

1	2	**3**	4

7	8

$\mathbf{P(X_{3,7} = 1)}$ is the probability that **3** and **7** are compared.

This is exactly the probability that either 3 or 7 is first picked to be a pivot out of the highlighted entries.

If 4, 5, or 6 get picked as a pivot first, then 3 and 7 would be separated and never see each other again.

HOW MANY COMPARISONS?

So, what's $\mathbf{E}[X_{a,b}]$?

$P(X_{a,b} = 1)$ *aka probability that **a** & **b** are compared*

=

probability that either **a** or **b** are selected as a pivot before elements between **a** and **b**.

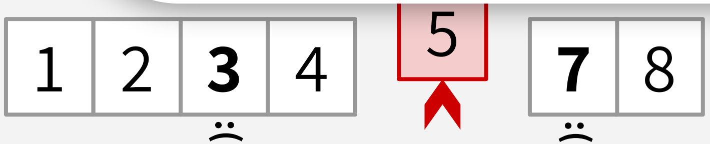
=

$$\frac{2}{(\text{\# elements from } \mathbf{a} \text{ to } \mathbf{b}\text{, inclusive})}$$

So … nple:

3 … ed.

3 … **first** … **es.**



If 4, 5, or 6 get picked as a pivot first, then 3 and 7 would be separated and never see each other again.

HOW MANY COMPARISONS?

So, what's **E[X$_{a,b}$]**?

P(X$_{a,b}$ = 1) *aka probability that **a** & **b** are compared*

=

probability that either **a** or **b** are selected as a pivot before elements between **a** and **b**.

=

$$\frac{2}{\mathbf{b} - \mathbf{a} + 1}$$

So

nple:

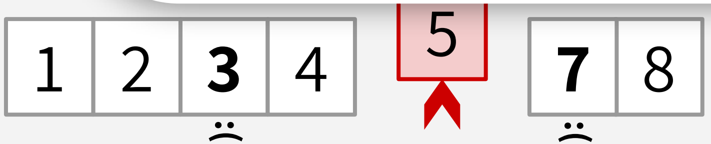
3

ed.

3

first

es.



If 4, 5, or 6 get picked as a pivot first, then 3 and 7 would be separated and never see each other again.

QUICKSORT EXPECTED RUNTIME

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed $E[X_{a,b}] = P(X_{a,b,} = 1)$

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

$$= 2n\sum_{c=1}^{n-1}\frac{1}{c+1}$$

Nothing in the summation depends on a, so pull 2 out

$$\leq 2n\sum_{c=1}^{n-1}\frac{1}{c}$$

decrease each denominator ⟶ we get the harmonic series!

$$= O(n \log n)$$

If E[# comparisons] = O(n log n), does this mean E[running time] is also O(n log n)?

YES! Intuitively, the runtime is dominated by comparisons. See Lemma 5.2 in Section 5.5 of AI if you're curious.

QUICKSORT

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Worst case runtime:
$O(n^2)$

Expected runtime:
$O(n \log n)$

3-MINUTE BREAK

Stay hydrated, stretch, ask questions, etc.

QUICKSORT IN PRACTICE

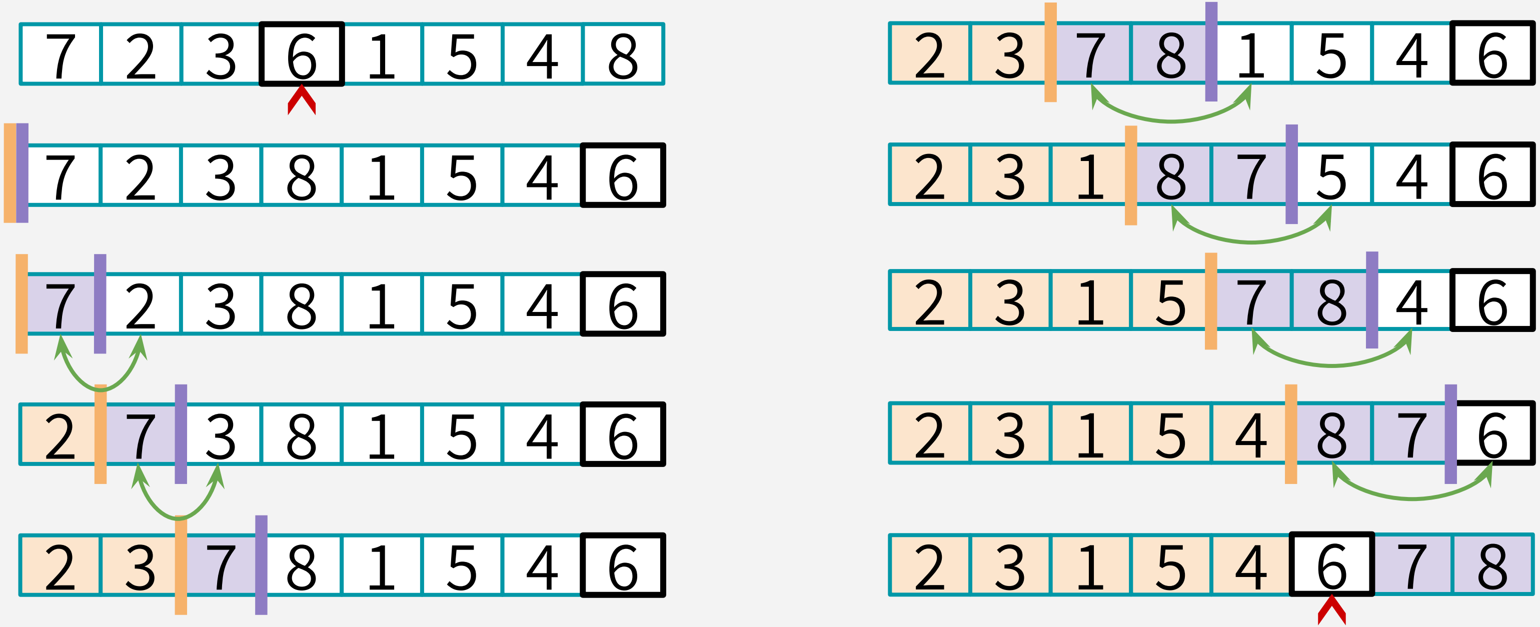
How is it implemented? Do people use it?

IMPLEMENTING QUICKSORT

In practice, a more clever approach is used to implement PARTITION, so that the entire QuickSort algorithm can be implemented “in-place”

(i.e. via swaps, rather than constructing separate L or R subarrays)

AN EXAMPLE IN-PLACE PARTITION



7 2 3 6 1 5 4 8

7 2 3 8 1 5 4 6

7 2 3 8 1 5 4 6

2 7 3 8 1 5 4 6

2 3 7 8 1 5 4 6

2 3 7 8 1 5 4 6

2 3 1 8 7 5 4 6

2 3 1 5 7 8 4 6

2 3 1 5 4 8 7 6

2 3 1 5 4 6 7 8

Choose pivot & swap with last element so pivot is at the end.

⇨

Initialize 🟧 and 🟪

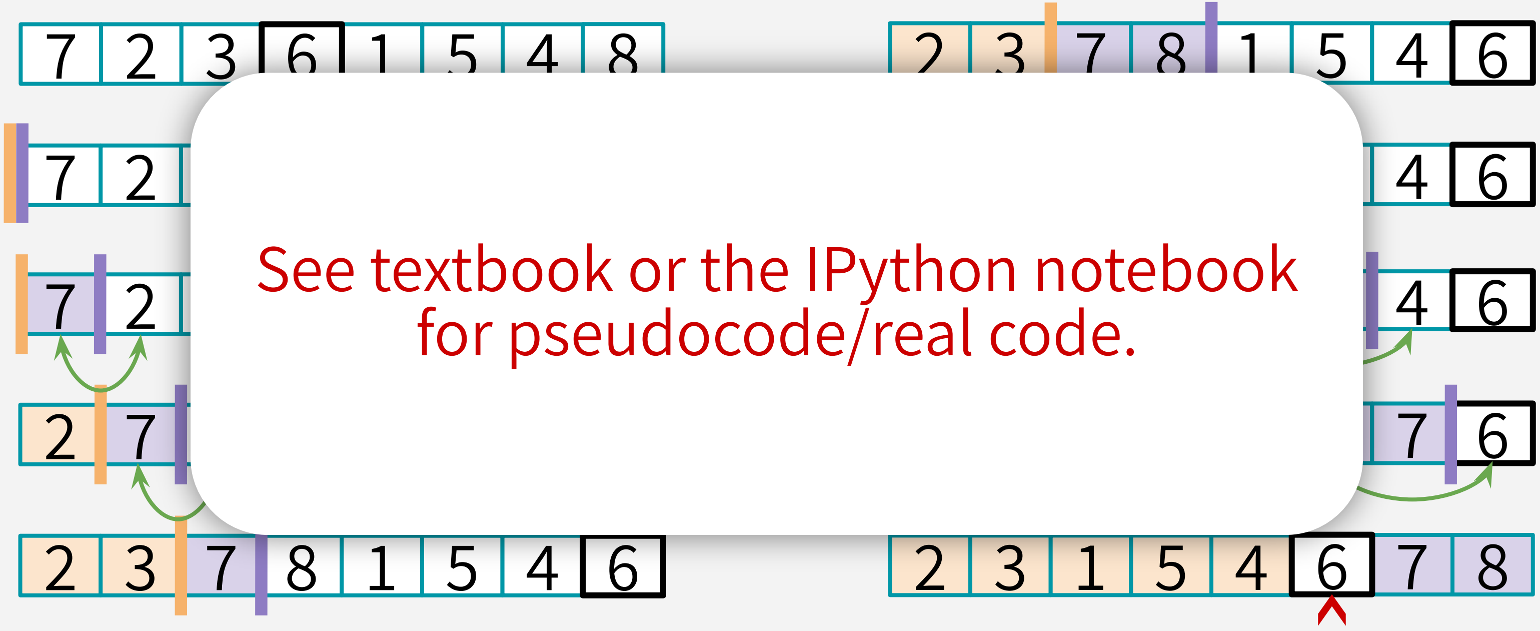
⇨

Increment 🟪 until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars

⇨

Repeat until the 🟪 bar reaches the end, then swap the pivot into the right place.

AN EXAMPLE IN-PLACE PARTITION



See textbook or the IPython notebook for pseudocode/real code.

Choose pivot & swap with last element so pivot is at the end. ⇨ Initialize ▮ and ▮ ⇨ Increment ▮ until it sees something smaller than pivot, **swap** the things ahead of the bars & increment both bars ⇨ Repeat until the ▮ bar reaches the end, then swap the pivot into the right place.

IMPLEMENTING QUICKSORT

There's another in-place partition algorithm called Hoare Partition that's even more efficient as it performs less swaps.

(you're not responsible for knowing it in this class)

Check out these Hungarian Folk Dancers showing you how it's done!
(and see the IPython notebook for details)

QUICKSORT vs. MERGESORT

	QuickSort (random pivot)	**MergeSort** (deterministic)
Runtime	**Worst-case: $O(n^2)$** **Expected: O(n log n)**	**Worst-case: O(n log n)**
Used by	Java (primitive types), C (qsort), Unix, gcc…	Java for objects, perl
In-place? **(i.e. with O(log n) extra memory)**	Yes, pretty easily!	Easy if you sacrifice runtime (O(nlogn) MERGE runtime). Not so easy if you want to keep runtime & stability.
Stable?	No	Yes
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists

You do not need to understand
any of this stuff

RECAP

- Runtimes of **randomized algorithms** can be measured in two main ways:
    - Expected runtime (you roll the dice)
    - Worst-case runtime (the bad guy gets to fix the dice)
- **QUICKSORT!**
    - Another *DIVIDE and CONQUER* sorting algorithm that employs randomness
    - Elegant, structurally simple, and actually used in practice!

NEXT TIME

- Can we sort faster than Θ(n log n)???