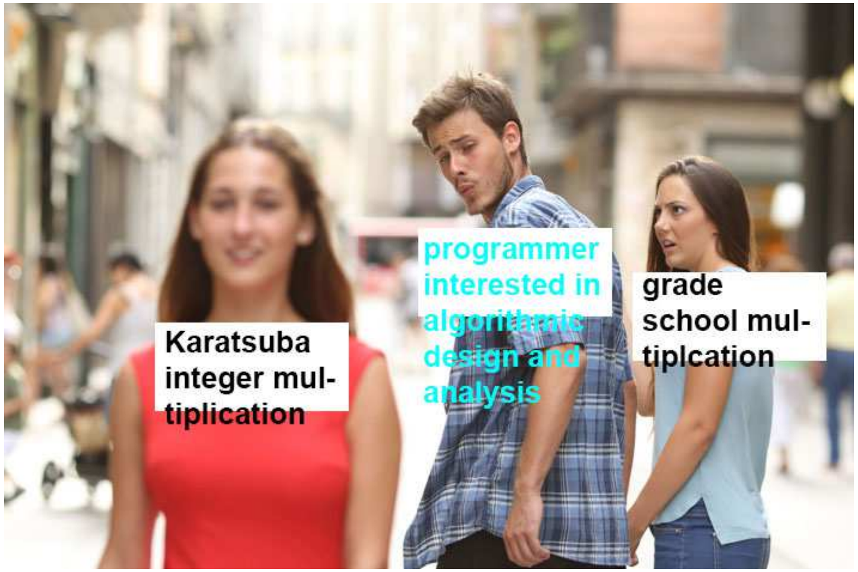
Lecture 5

Randomized algorithms and QuickSort

Announcements

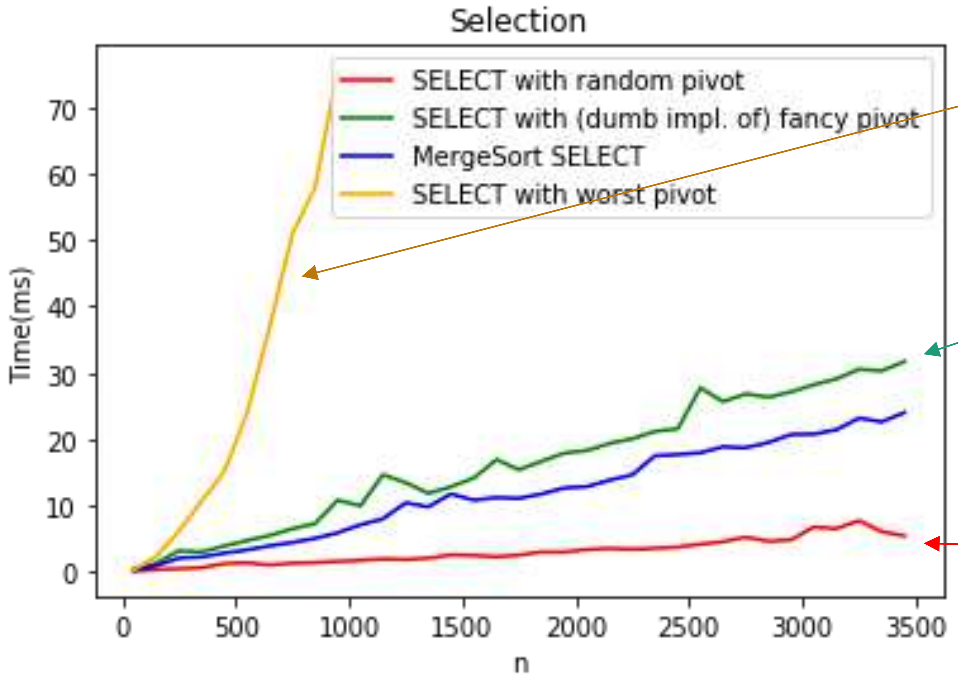
- HW1 is graded! Thanks TAs for **super**-fast turnaround!!
- HW2 is posted! Due Friday.
- Please send any OAE letters to Jessica Su (stysu@stanford.edu) by Friday.
- Garrick attempts to make my cultural references more up-to-date:



Thanks Garrick!

Last time

- We saw a divide-and-conquer algorithm to solve the **`Select`** problem in time O(n) in the worst-case.
- It all came down to picking the pivot…



We choose a pivot **randomly** and then a bad guy gets to decide what the array was.

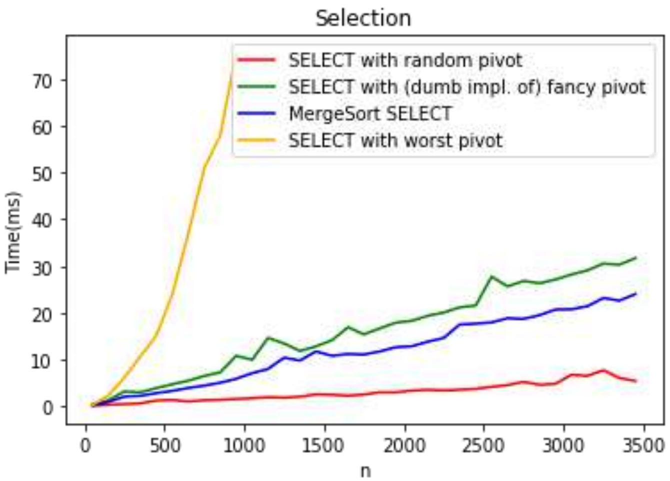
We choose a pivot **cleverly** and then a bad guy gets to decide what the array was.

The bad guy gets to decide what the array was and *then* we choose a pivot **randomly.**

Randomized algorithms

- We make some random choices during the algorithm.
- We hope the algorithm works.
- We hope the algorithm is fast.

e.g., **`Select`** with a random pivot is a randomized algorithm.



It was actually always correct

Looks like it's probably fast but not always.

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - BogoSort
  - QuickSort

- BogoSort is a pedagogical tool.
- QuickSort is important to know. (in contrast with BogoSort…)

How do we measure the runtime of a randomized algorithm?

Scenario 1

1. Bad guy picks the input.
2. You run your randomized algorithm.

Scenario 2

1. Bad guy picks the input.
2. Bad guy chooses the randomness (fixes the dice)

- In **Scenario 1**, the running time is a **random variable.**
  - It makes sense to talk about **expected running time**.
- In **Scenario 2**, the running time is **not random**.
  - We call this the **worst-case running time** of the randomized algorithm.

Today

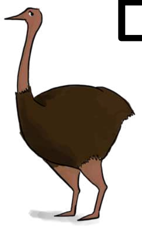
- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - BogoSort
  - QuickSort

- BogoSort is a pedagogical tool.
- QuickSort is important to know. (in contrast with BogoSort…)

Example

- BogoSort(A):
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

Suppose that you can draw a random integer in {1,…,n} in time O(1). How would you randomly permute an array in-place in time O(n)?



Ollie the over-achieving ostrich

- What is the expected running time?
  - You analyzed this in your pre-lecture exercise *[also on board now]*

- What is the worst-case running time?
  - *[on board]*

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - BogoSort
  - QuickSort

- BogoSort is a pedagogical tool.
- QuickSort is important to know. (in contrast with BogoSort…)

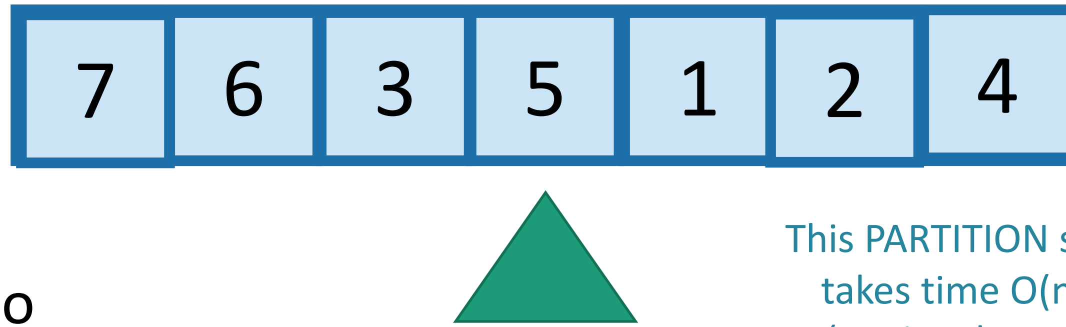
a better randomized algorithm: QuickSort

- Runs in expected time $O(n\log(n))$.
- Worst-case runtime $O(n^2)$.
- In practice often more desirable.
  - (More later)

Quicksort

We want to sort this array.

First, pick a "pivot."
Do it at random.



This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

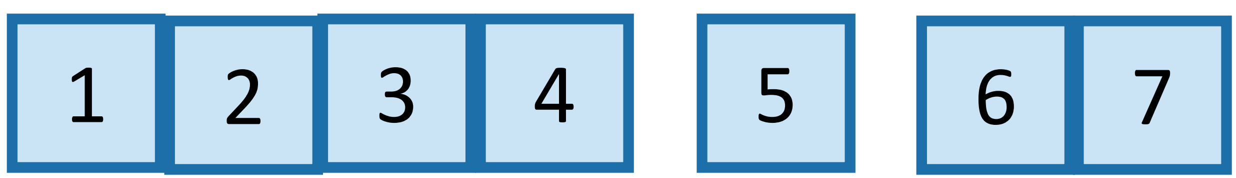
Next, partition the array into "bigger than 5" or "less than 5"

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:



PseudoPseudoCode for what we just saw

IPython Lecture 5 notebook for actual code.

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] at random. Call this the **pivot**.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - QuickSort(L)
  - QuickSort(R)

Assume that all elements of A are distinct. How would you change this if that's not the case?

How would you do all this in-place? Without hurting the running time? (We'll see later…)

Running time?

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

- In an ideal world…
  - if the pivot splits the array exactly in half…

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

- We've seen that a bunch:

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

The expected running time of QuickSort is O(nlog(n)).

Proof:*

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs,

  the running time is $T(n) = O(n\log(n))$.
- Therefore,

  the expected running time is $O(n\log(n))$.

***Disclaimer: this proof is wrong.**

Red flag

We can use the same argument to prove something false.

- **Slow** Sort(A):
  - **If** len(A) <= 1:
    - **return**
  - **Pick the pivot x to be either max(A) or min(A), randomly**
    - \\ We can find the max and min in O(n) time
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - **Slow** Sort(L)
  - **Slow** Sort(R)

- Same recurrence relation:

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

- But now, one of |L| or |R| is n-1.
- Running time is $O(n^2)$, with probability 1.

The expected running time of SlowSort is O(nlog(n)).

Proof:*

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs,

  the running time is $T(n) = O(n\log(n))$.
- Therefore,

  the expected running time is $O(n\log(n))$.

***Disclaimer: this proof is wrong.**

What's wrong?

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs,

  the running time is $T(n) = O(n \log(n))$.
- Therefore,

  the expected running time is $O(n \log(n))$.

This argument says:

$$T(n) = \text{some function of } |L| \text{ and } |R|$$ ✓

$$\mathbb{E}[T(n)] = \mathbb{E}[\text{some function of } |L| \text{ and } |R|]$$ ✓

$$\mathbb{E}[T(n)] = \text{some function of } \mathbb{E}|L| \text{ and } \mathbb{E}|R|$$ ✗

That's not how expectations work!

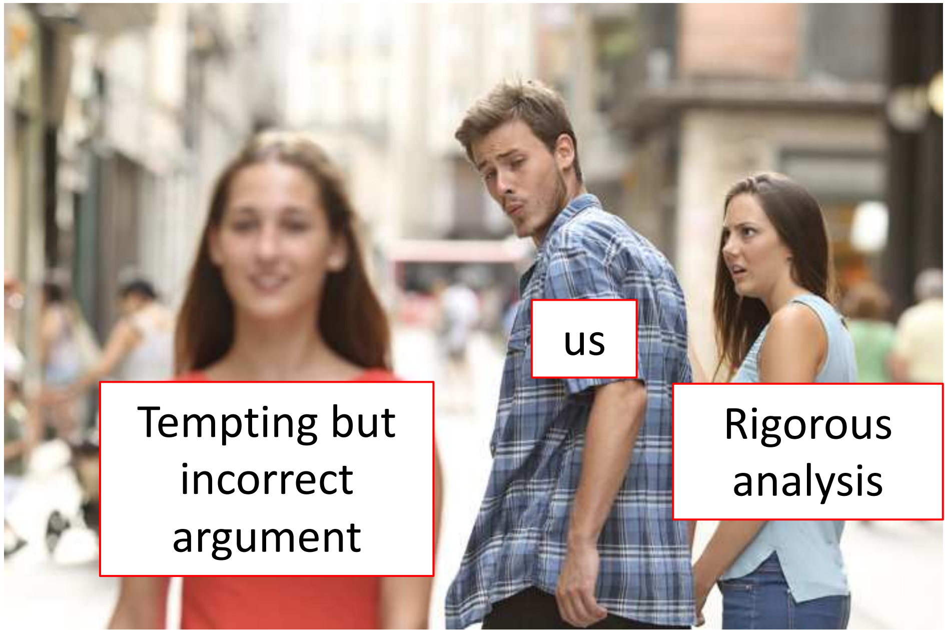
Plucky the Pedantic Penguin

Instead

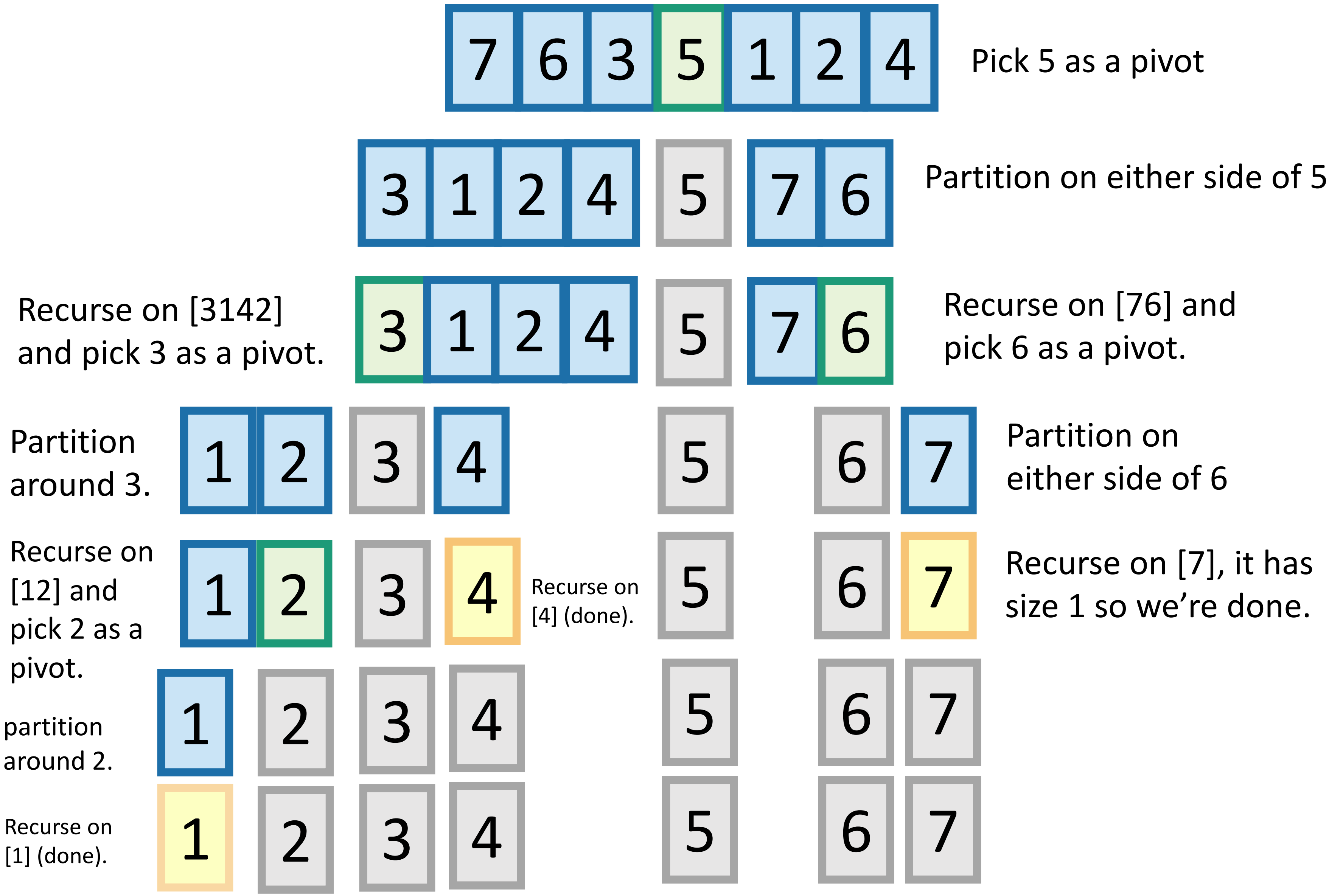
- We'll have to think a little harder about how the algorithm works.

Next goal:

- Get the same conclusion, correctly!

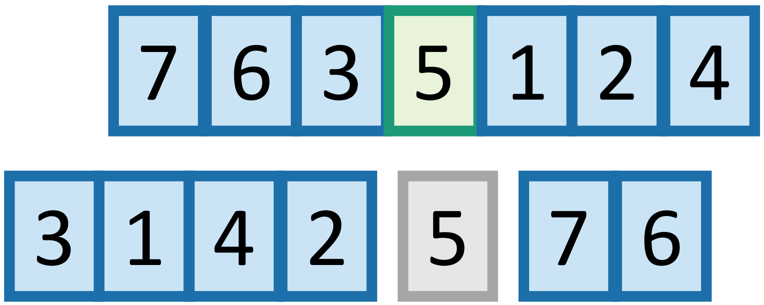


Example of recursive calls

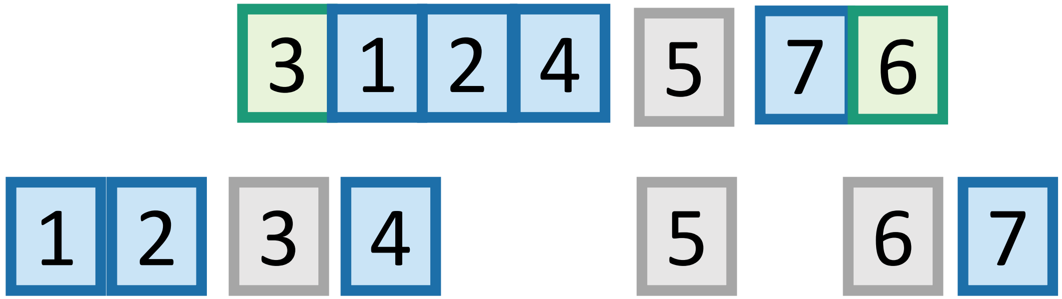


How long does this take to run?

- We will count the number of comparisons that the algorithm does.
  - This turns out to give us a good idea of the runtime. (Not obvious).
- How many times are any two items compared?



In the example before, everything was compared to 5 once in the first step....and never again.



But not everything was compared to 3.
5 was, and so were 1,2 and 4.
But not 6 or 7.

Each pair of items is compared either 0 or 1 times. Which is it?

7	6	3	5	1	2	4

Let's assume that the numbers in the array are actually the numbers 1,…,n

Of course this doesn't have to be the case! It's a good exercise to convince yourself that the analysis will still go through without this assumption. (Or see CLRS)

- Whether or not a,b are compared is a random variable, that depends on the choice of pivots. Let's say

$$X_{a,b} = \begin{cases} 1 & \textit{if a and b are ever compared} \\ 0 & \textit{if a and b are never compared} \end{cases}$$

- In the previous example $\mathbf{X_{1,5} = 1}$, because item 1 and item 5 were compared.
- But $\mathbf{X_{3,6} = 0}$, because item 3 and item 6 were NOT compared.
- Both of these depended on our random choice of pivot!

Counting comparisons

- The number of comparisons total during the algorithm is

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b}$$

- The expected number of comparisons is

$$E\left[\sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b}\right] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}]$$

using linearity of expectations.

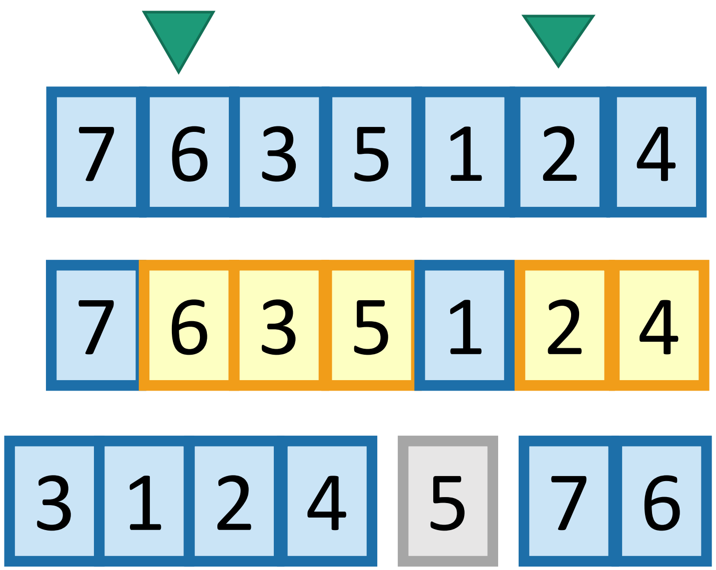
Counting comparisons

expected number of comparisons:

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} E[\,X_{a,b}]$$

- So we just need to figure out $E[\ X_{a,b}\ ]$
- $E[\ X_{a,b}\ ] = P(\ X_{a,b} = 1\ ) \cdot 1 + P(\ X_{a,b} = 0\ ) \cdot 0 = P(X_{a,b} = 1)$
  - (using definition of expectation)
- So we need to figure out

$P(X_{a,b} = 1)$ = the probability that a and b are ever compared.



Say that a = 2 and b = 6. What is the probability that 2 and 6 are ever compared?

This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.

If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.

Counting comparisons

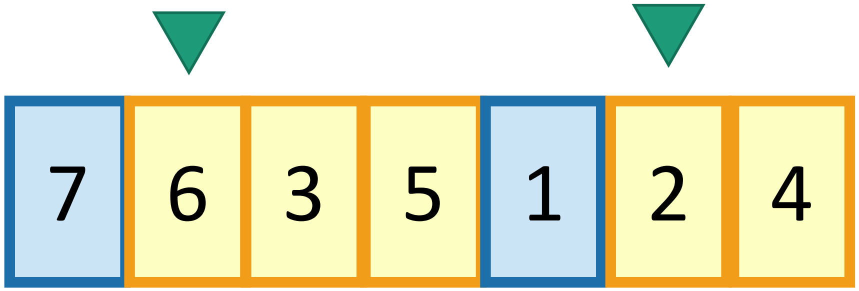
$P(\,X_{a,b} = 1\,)$

= probability a,b are ever compared

= probability that one of a,b are picked first out of all of the b – a +1 numbers between them.

2 choices out of b-a+1…

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



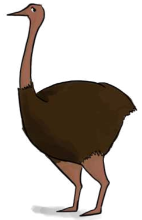
All together now…

Expected number of comparisons

- $E\left[\sum_{a=1}^{n}\sum_{b=a+1}^{n} X_{a,b}\right]$ — This is the expected number of comparisons throughout the algorithm
- $= \sum_{a=1}^{n}\sum_{b=a+1}^{n} E[\, X_{a,b}]$ — linearity of expectation
- $= \sum_{a=1}^{n}\sum_{b=a+1}^{n} P(\, X_{a,b} = 1)$ — definition of expectation
- $= \sum_{a=1}^{n}\sum_{b=a+1}^{n} \frac{2}{b-a+1}$ — the reasoning we just did

- This is a big nasty sum, but we can do it.
- We get that this is less than 2n ln(n).

Do this sum!



Ollie the over-achieving ostrich

Almost done

- We saw that E[number of comparisons] = O(n log(n))
- Is that the same as E[running time]?

- In this case, **yes**.
- We need to argue that the running time is dominated by the time to do comparisons.
- (See CLRS for details).

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] at random. Call this the pivot.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - QuickSort(L)
  - QuickSort(R)

Conclusion

- Expected running time of QuickSort is O(nlog(n))

Bonus material in the lecture notes: a second way to show this!

Worst-case running time

- Suppose that an adversary is choosing the "random" pivots for you.
- Then the running time might be $O(n^2)$
  - Eg, they'd choose to implement SlowSort
  - In practice, this doesn't usually happen.

A note on implementation

- This pseudocode is easy to understand and analyze, but is not a good way to implement this algorithm.

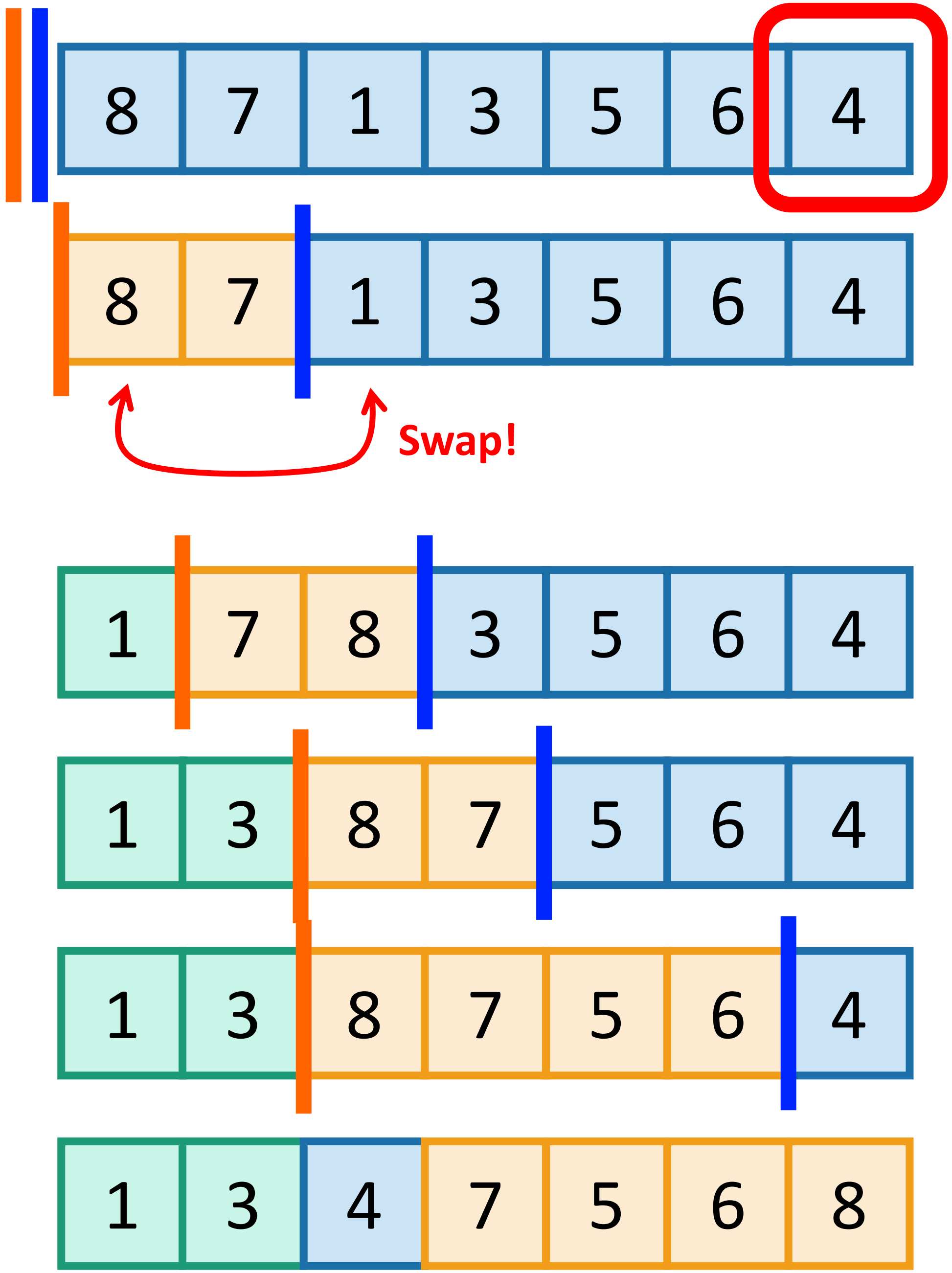
> - QuickSort(A):
>   - **If** len(A) <= 1:
>   - **return**
>   - Pick some x = A[i] at random. Call this the pivot.
>   - PARTITION the rest of A into:
>   - L (less than x) and
>   - R (greater than x)
>   - Replace A with [L, x, R] (that is, rearrange A in this order)
>   - QuickSort(L)
>   - QuickSort(R)

- Instead, implement it **in-place** (without separate L and R)
    - You may have seen this in 106b.
    - Here are some Hungarian Folk Dancers showing you how it's done: https://www.youtube.com/watch?v=ywWBy6J5gz8
    - Check out IPython notebook for Lecture 5 for two different ways.

A better way to do Partition

Pivot

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



Initialize [orange bar] and [blue bar]

Step [blue bar] forward.

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

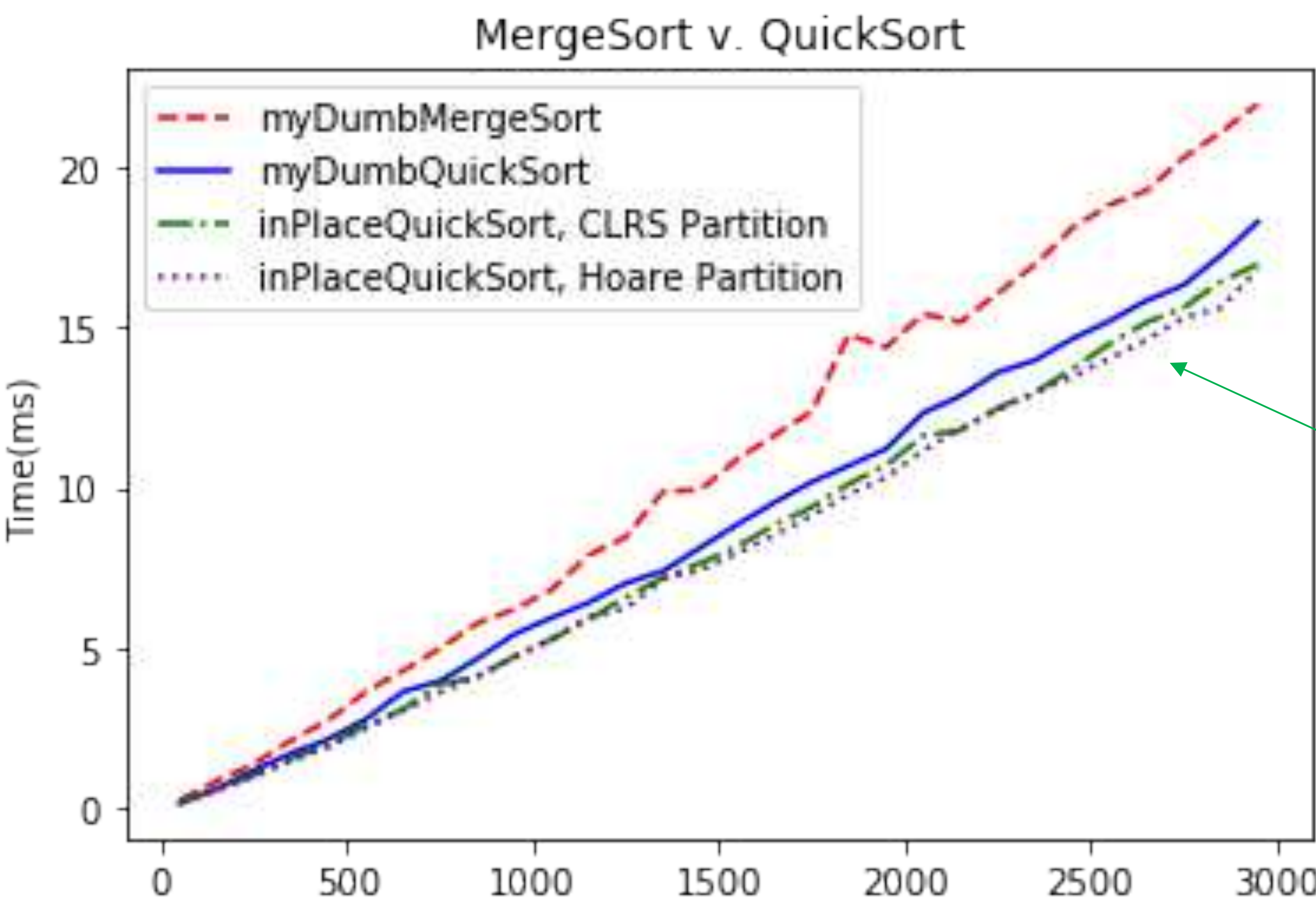
Repeat till the end, then put the pivot in the right place.

See CLRS or Lecture 5 IPython notebook for pseudocode/real code.

QuickSort vs. smarter QuickSort vs. Mergesort?

See IPython notebook for Lecture 5

- All seem pretty comparable…



Hoare Partition is a different way of doing it (c.f. CLRS Problem 7-1), which you might have seen elsewhere. You are not responsible for knowing it for this class.

The slicker in-place ones use less space, and also are a smidge faster on my system.

QuickSort vs MergeSort

*In fact, I don't know how to do this if you want O(nlog(n)) worst-case runtime and stability.

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	• Worst-case: $O(n^2)$ • Expected: O(n log(n))	Worst-case: O(n log(n))
Used by	• Java for primitive types • C qsort • Unix • g++	• Java for objects • Perl
In-Place? (With O(log(n)) extra memory)	Yes, pretty easily	Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice runtime).
Stable?	No	Yes
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists

Understand this

These are just for fun. (Not on exam).

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - BogoSort
  - QuickSort

- BogoSort is a pedagogical tool.
- QuickSort is important to know. (in contrast with BogoSort…)

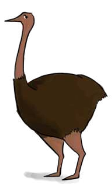
Recap

Recap

- How do we measure the runtime of a randomized algorithm?
  - Expected runtime
  - Worst-case runtime

- QuickSort (with a random pivot) is a randomized sorting algorithm.
  - In many situations, QuickSort is nicer than MergeSort.
  - In many situations, MergeSort is nicer than QuickSort.

Code up QuickSort and MergeSort in a few different languages, with a few different implementations of lists A (array vs linked list, etc). What's faster?
(This is an exercise best done in C where you have a bit more control than in Python).



Ollie the over-achieving ostrich

Next time

- Can we sort faster than $\Theta(n\log(n))$??

Before next time

- ***Pre-lecture exercise*** for Lecture 6.
  - Can we sort even faster than QuickSort/MergeSort?

https://xkcd.com/1185/

(h/t Dana)