Randomized Algorithms

CSE 312 Spring 21
Lecture 28

Announcements

25

Three Concept checks this week (to get to a total of 30)

Concept Check 28 is a “standard” one for today’s lecture

Concept Check 29 asks you to fill out the official UW course evals.

Please fill these out! It’s super useful to know what you found valuable, what you found frustrating, what took too long,…we are always working to improve the course.

Concept Check 30 asks you to fill out a different anonymous form with specific feedback about the real world assignments.

They were new this quarter – asking about logistics (e.g., would you rather have had these as part of regular homeworks like the programming or as separate work like we did?), for any examples to add to suggested lists (like in real world 1/2), etc.

Start thinking of questions to ask in section (or on Friday)

Final Logistics

What’s Fair Game?

All the content/theorems we learned.

Principles behind the programming questions, but not the code itself.

Applications lectures won’t be tested directly (that is to say we won’t give you a problem that would only make sense if you followed the application lectures.

But the principles behind the applications are absolutely fair game.

And we reserve the right to give a problem inspired by the applications lectures, as long as you could do it even if you didn’t see the applications.

E.g. we could ask a question about covariance (because you know the formula from main lectures) but we wouldn’t ask you to think about the ML implications of a picture of covariances.

What’s a randomized algorithm?

A randomized algorithm is an algorithm that uses randomness in the computation.

Well, ok.

Let’s get a little more specific.

Two common types of algorithms

Las Vegas Algorithm

Always tells you the right answer

Takes varying amounts of time.

Monte Carlo Algorithm

Usually tells you the right answer.

A classic Las Vegas Algorithm

Remember Quick Sort?

Pick a "pivot" element

Move all the elements smaller than the pivot to the left subarray (in no particular order)

Move all elements greater than the pivot element to the right subarray (in no particular order)

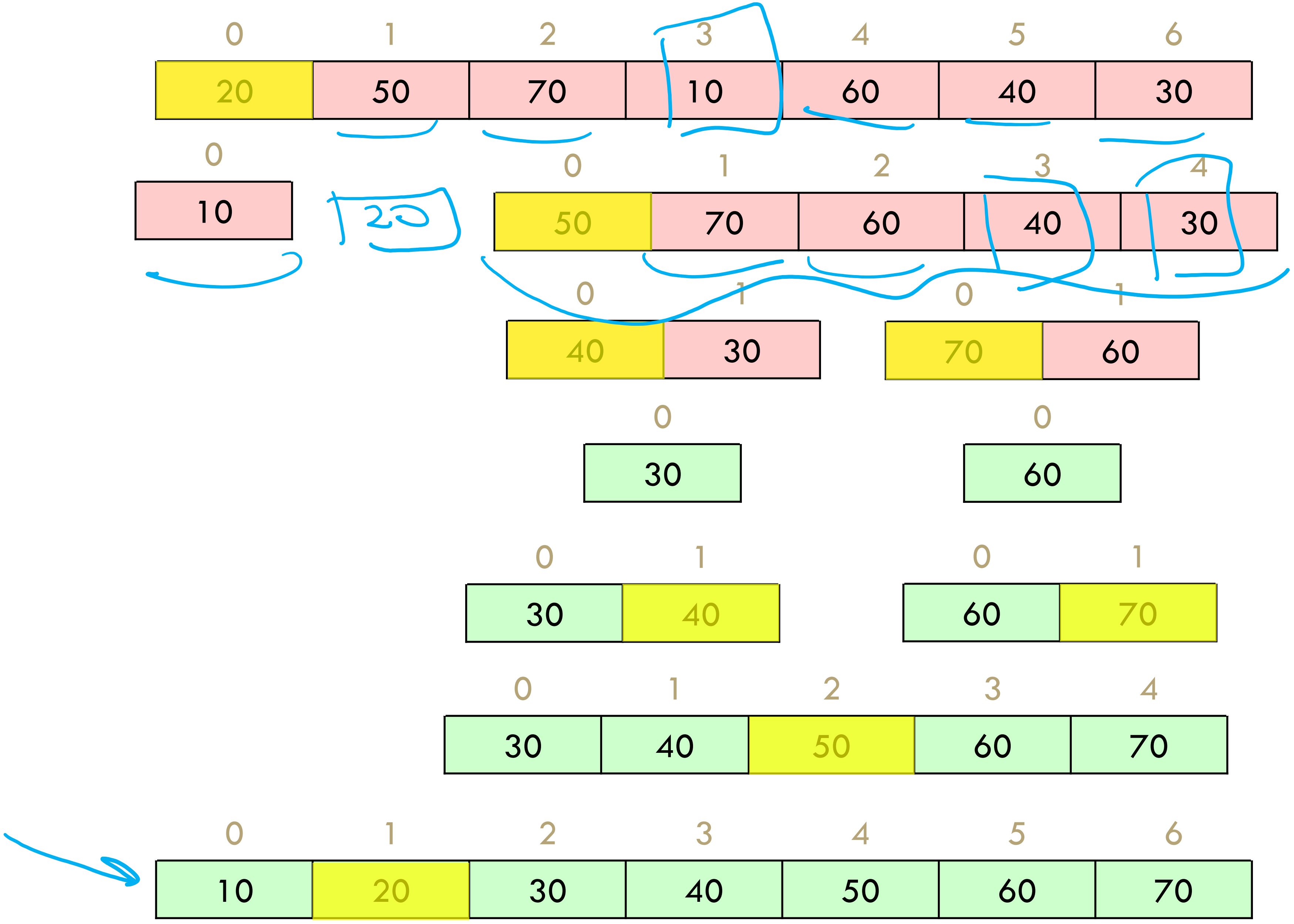
Make a recursive call

It's sometimes implemented as a Las Vegas Algorithm.

That is, you'll always get the same answer (there's only one sorted array) but the time can vary.

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

20

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

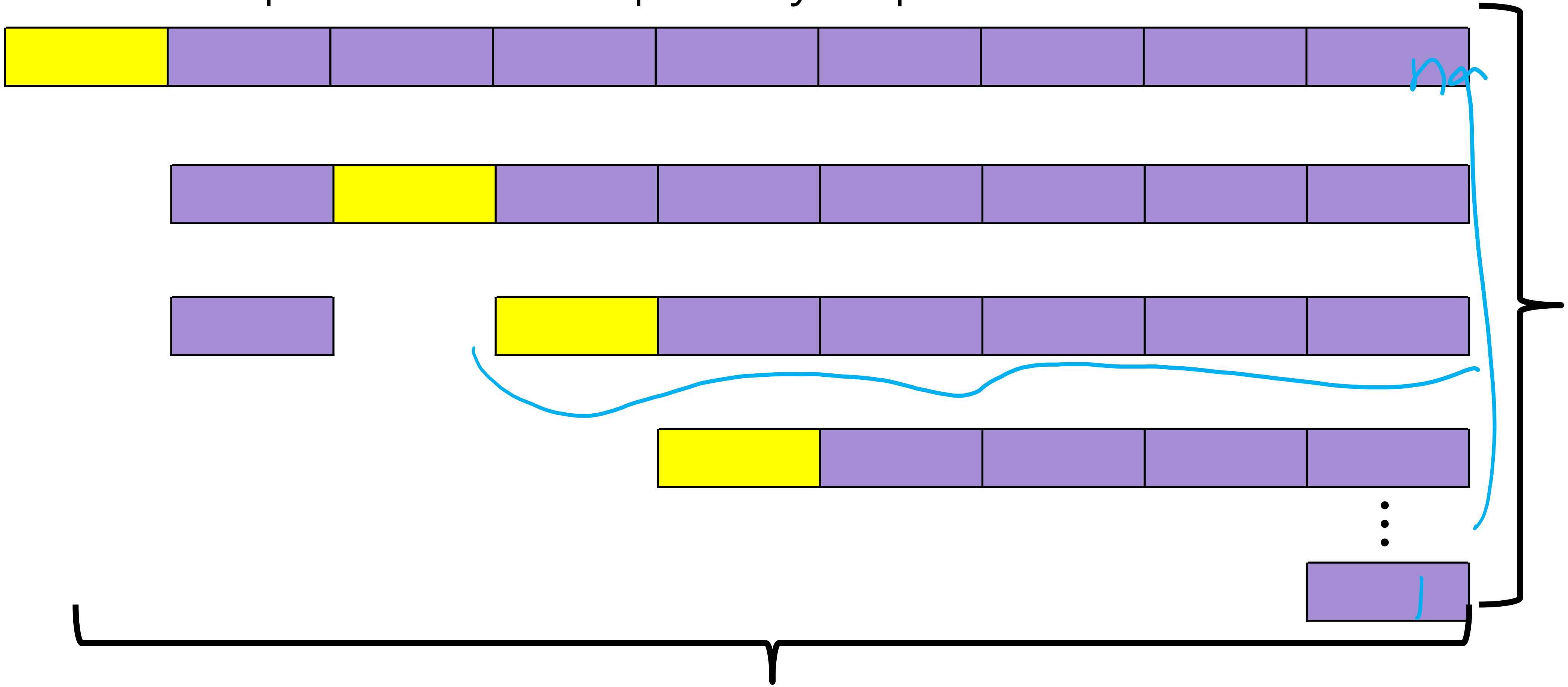
0	1	2	3	4	5	6
10	20	30	40	50	60	70

How long does it take?

Total time:

$$\sum_{i=1}^{n} i = O(n^2)$$

Well…it depends on what pivots you pick.



About n levels

$O(i)$ work when i elements remaining.

For Simplicity

We’ll talk about how quicksort is really done at the end.

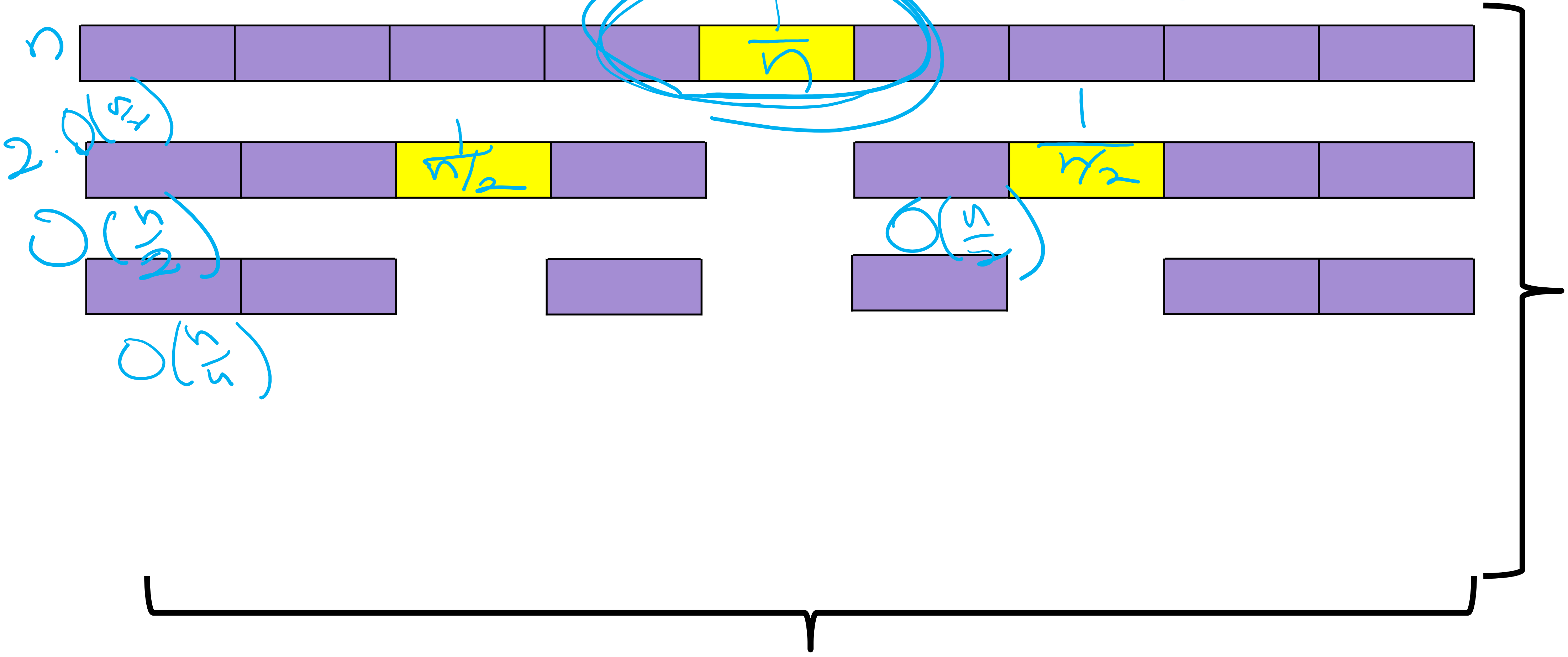
For now an easier-to-analyze version:

```
if(elements remaining > 1)
pick a pivot uniformly at random
split based on pivot
sortedLeft = QuickSort(left half)
sortedRight = QuickSort(right half)
return (sortedLeft pivot sortedRight)
```

What leads to a good time?

$$\frac{n}{2^i} < 1 \qquad i \approx \log(n)$$

Pivots closer to the middle would be better.



Cut in half is ideal. $O(\log n)$ levels.

$O(i)$ work when i elements remaining. -- $O(n)$ per level

Work at each level

How much work do those splits do?

Each call choose a pivot ($O(n)$ total per level)

Each element is compared to the pivot and possibly swapped ($O(1)$ per element so $O(n)$ per level)

So as long as we need at most $O(\log n)$ levels, we'll get the $O(n \log n)$ running time we need.

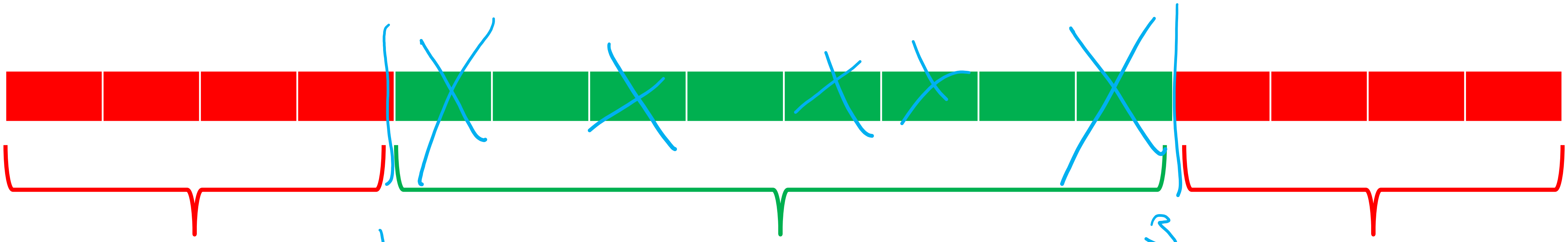
We only get the perfect pivot with probability $\frac{1}{n}$. That's not very likely…maybe we can settle for something more likely.

Focus on an element

Let's focus on one element of the array x_i.

The recursion will stop when every element is all alone in their own subarray.

Call an iteration "good for x_i" if the array containing x_i in the next step is at most $\frac{3}{4}$ the size it was in the current step.



Pivot here: might leave x_i in a big subarray (if x_i is big)

Pivot here: both subarrays $\leq 3/4$ size. Must be good for x_i.

Pivot here: might leave x_i in a big subarray (if x_i is small)

Good for x_i

At least half of the potential pivots guarantee x_i ends up with a good iteration. So we'll use $\underline{\mathbb{P}(x_i \textbf{ good iteration}) \geq 1/2}$

It's actually quite a bit more than half for large arrays – one of the two red subarrays **might** be good for x_i (just bad for the others in the array)

x_i might be our pivot, in which case it's totally done.

To avoid any tedious special cases for small arrays, just say at least ½.

How many levels?

How many levels do we need to go?

Once x_i is in a size 1 subarray, it's done. How many iterations does it take?

If we only had good iterations, we'd need

CLT;

approximately

$$\left(\frac{3}{4}\right)^k n \leq 1 \Rightarrow n \leq \left(\frac{4}{3}\right)^k \Rightarrow k \geq \log_{4/3} n.$$

I want (at the end of our process) to say with probability at least <blah> the running time is at most $O(n \log n)$.
What's the probability of getting a lot of good iterations…what's the tool we should use?

Fill out the poll everywhere so Robbie knows how long to explain
Go to pollev.com/cse312

Needed iterations

x_i is done after $\log_{4/3} n = \frac{\ln(n)}{\ln\left(\frac{4}{3}\right)} \leq 4 \ln n$ good for x_i iterations.

Let's imagine we do $24 \ln(n)$ iterations. Let X be the number of good for x_i iterations. Let $Y \sim Bin\left(24 \ln(n), \frac{1}{2}\right)$

$$\mathbb{P}(X \leq 4 \ln n) \leq \mathbb{P}(Y \leq 4 \ln n)$$

Set up for Chernoff

$$\mathbb{P}\left(Y \leq [1 - \delta] \cdot \frac{24 \ln(n)}{2}\right) \leq \exp\left(-\frac{\delta^2 \mu}{2}\right)$$

$$1 - \delta = 1/3 \Rightarrow \delta = 2/3$$

Applying Chernoff

$$\mathbb{P}\left(Y \le [1-\delta] \cdot \frac{24\ln(n)}{2}\right) \le \exp\left(-\frac{\delta^2 \mu}{2}\right) \le \exp\left(-\frac{\frac{1}{3^2} \cdot 12\ln(n)}{2}\right) = \underline{e^{-\frac{8}{3} \cdot \ln(n)}}$$

So, the probability that x_i is not done after $24\ln(n)$ iterations is at most $e^{-8\ln(n)/3} \underline{= n^{-8/3}}$

Finishing The bound

So x_i is done with probability at least $1 - n^{-8/3}$

But x_i being done doesn't mean the whole algorithm is done…

This argument so far does apply to any other x_j -- but they aren't independent, so….union bound!

$$\mathbb{P}(\text{algorithm not done}) \leq \sum \mathbb{P}(x_i \overset{\text{not}}{\text{done}}) = n\mathbb{P}(x_i \overset{\text{not}}{\text{done}}) = n \cdot n^{-\frac{8}{3}} = n^{-5/3}$$

$$\mathbb{P}(\text{algorithm done}) > 1 - n^{-5/3}.$$

$$> 1 - \frac{1}{5}$$

The Theorem

Quicksort

With probability at least $1 - \frac{1}{n}$, Quicksort runs in time $\boldsymbol{O(n \cdot log\ n)}$

This kind of bound (with probability → 1 as $n \to \infty$ is called a “high probability bound” we say quicksort needs $O(n \log n)$ time “with high probability”

Better than finding a bound on the expected running time!

Want a different bound?

Want an even better probability? You just have to tweak the constant factors!

Be more careful in defining a "good iteration" or just change $24\ln(n)$ to $48\ln(n)$ or $100\ln(n)$.

It all ends up hidden in the big-O anyway.

That's the power of concentration – the constant coefficient affects the exponent of the probability.

Common Quicksort Implementations

A common strategy in practice is the “median of three” rule.

Choose three elements (either at random or from specific spots). Take the median of those for your pivot

Guarantees you don’t have the worst possible pivot.

Only a small constant number of extra steps beyond the fixed pivot (find the median of three numbers is just a few comparisons).

Another strategy: find the true median (very fancy, very impractical: take 421)

Algorithms with some probability of failure

There are also algorithms that sometimes give us the wrong answer. (Monte Carlo Algorithms)

Wait why would we accept a probability of failure?

Suppose your algorithm **succeeds** with probability only $1/n$.

But given two runs of the algorithm, you can tell which is better.

E.g. “find the biggest <blah>” – whichever is bigger is the better one.

How many independent runs of the algorithm do we need to get the right answer with high probability?

Small Probability of Failure

How many independent runs of the algorithm do we need to get the right answer with high probability?

Probability of failure

$$\left(1-\frac{1}{n}\right)^{k\cdot n} \leq e^{-k}$$

Choose $k \approx \ln(n)$, and we get high probability of success.

So $n \cdot \ln(n)$ (for example) independent runs gives you the right answer with high probability.

Even with very small chance of success, a moderately larger number of iterations gives high probability of success. Not a guarantee, but close enough to a guarantee for most purposes.