Lecture 6

Sorting lower bounds on O(n)-time sorting

Announcements

- HW2 due Friday

- Please send any OAE letters to Luna Frank-Fischer (luna16@stanford.edu) by April 28.

Sorting

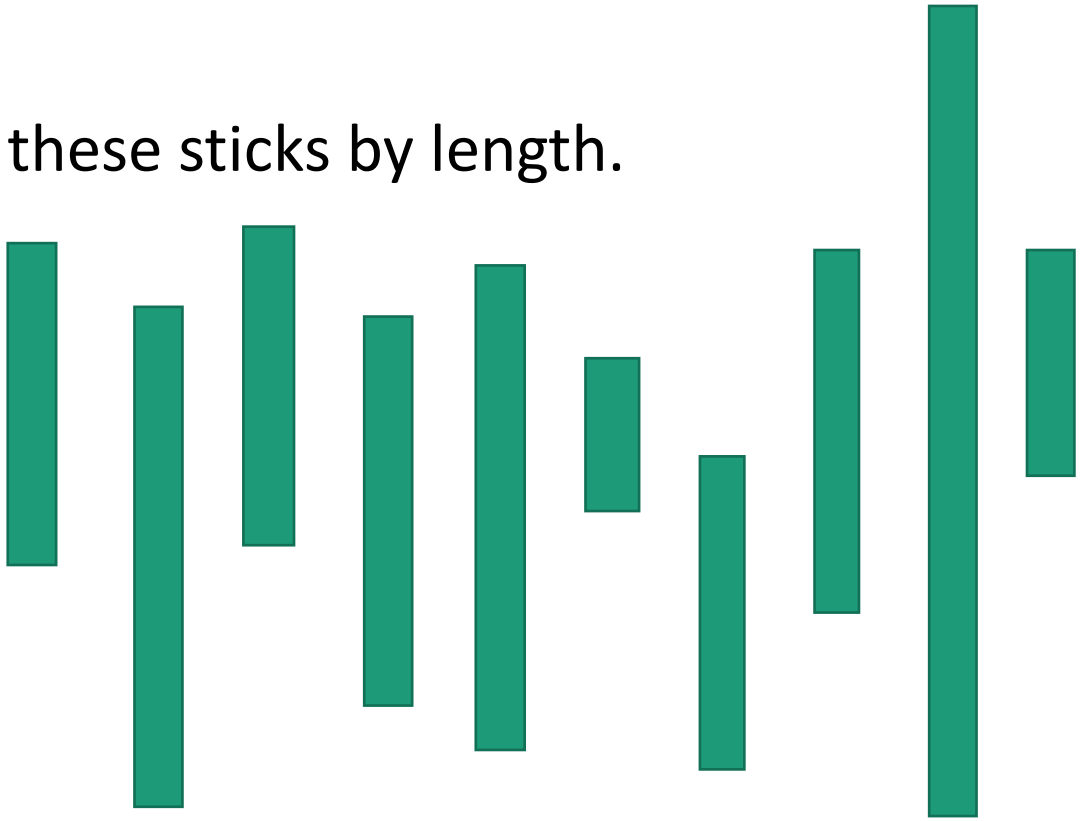
- We've seen a few O(n log(n))-time algorithms.
  - MERGESORT has worst-case running time O(nlog(n))
  - QUICKSORT has expected running time O(nlog(n))

Can we do better?

Depends on who you ask…

An O(1)-time algorithm for sorting: StickSort

- Problem: sort these sticks by length.



- Now they are sorted this way.
- Algorithm:
  - Drop them on a table.

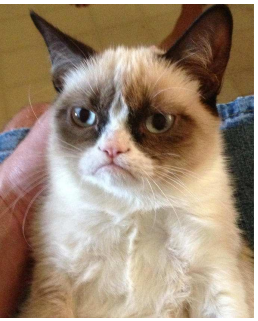
That may have been unsatisfying

- But StickSort does raise some important questions:
  - What is our model of computation?
    - Input: array
    - Output: sorted array
    - Operations allowed: comparisons

    -vs-

    - Input: sticks
    - Output: sorted sticks in vertical order
    - Operations allowed: dropping on tables
  - What are reasonable models of computation?

Today: two (more) models



- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - We'll see that any algorithm in this model must take at least $\Omega(n \log(n))$ steps.

- Another model (more reasonable than the stick model…)
  - BucketSort and RadixSort
  - Both run in time $O(n)$

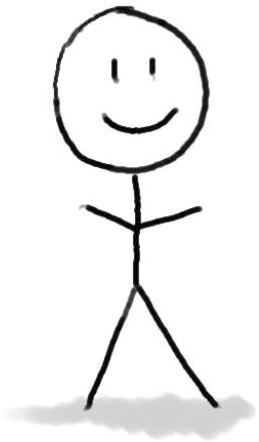
Comparison-based sorting

Comparison-based sorting algorithms

😀 is shorthand for "the first thing in the input list"

Want to sort these items.
There's some ordering on them, but we don't know what it is.

Is  bigger than  ?



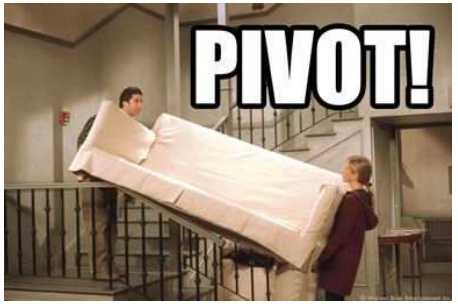
Algorithm

YES

There is a genie who knows what the right order is.

The algorithm's job is to output a correctly sorted list of all the objects.

The genie can answer YES/NO questions of the form:
is [this] bigger than [that]?

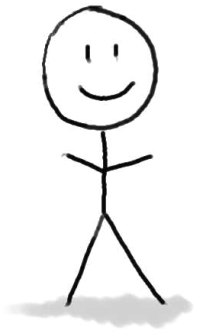


All the sorting algorithms we have seen work like this.

eg, QuickSort:

Pivot!

7	6	3	5	1	4	2



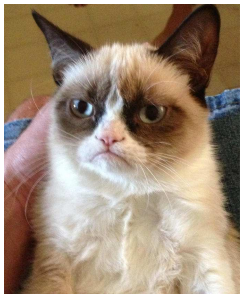
Is 7 bigger than 5 ? **YES**

Is 6 bigger than 5 ? **YES**

Is 3 bigger than 5 ? **NO**

5

etc.



Lower bound of Ω(n log(n)).

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.
  - Any randomized comparison-based sorting algorithm must take Ω(n log(n)) steps in expectation.

This covers all the sorting algorithms we know!!!

- How might we prove this?

1. Consider all comparison-based algorithms, one-by-one, and analyze them.

2. Don't do that.

Instead, argue that all comparison-based sorting algorithms give rise to a **decision tree**.
Then analyze decision trees.

Decision trees

Sort these three things.

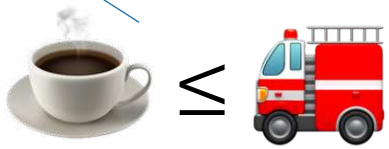
YES

NO

etc…

YES

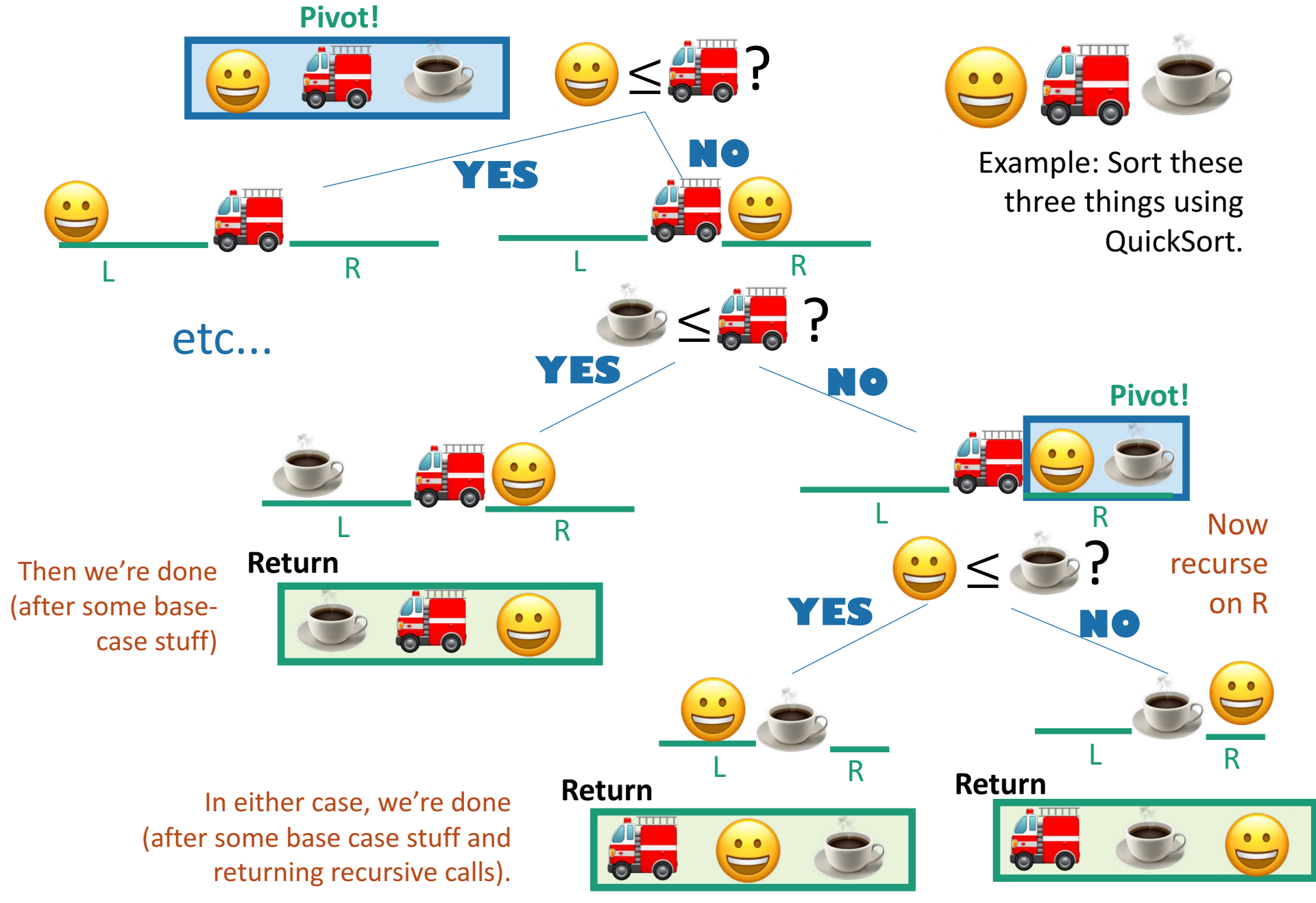
NO



YES

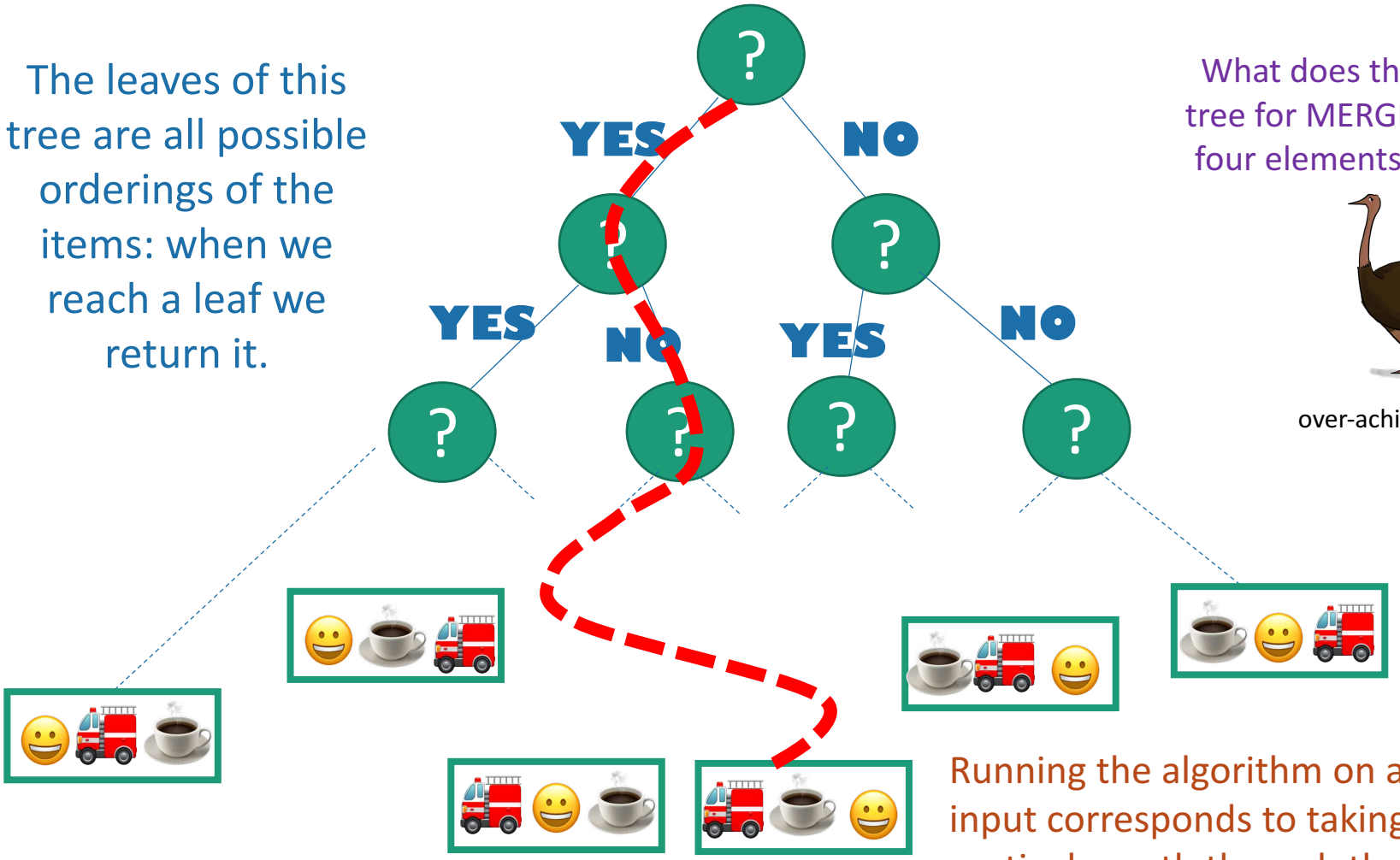
NO

All comparison-based algorithms look like this



All comparison-based algorithms have an associated decision tree.

The leaves of this tree are all possible orderings of the items: when we reach a leaf we return it.



What does the decision tree for MERGESORTING four elements look like?

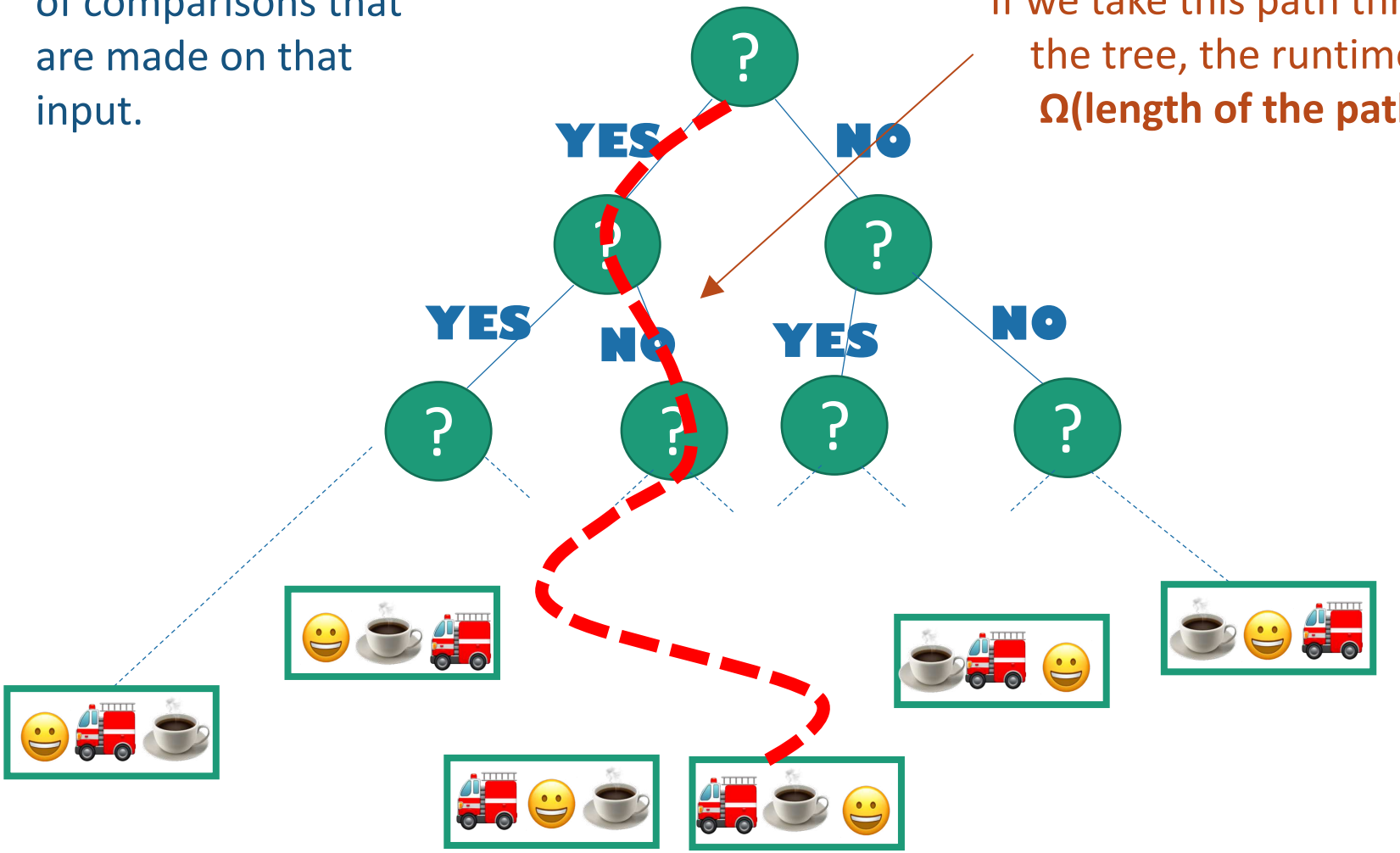
Ollie the over-achieving ostrich

Running the algorithm on a given input corresponds to taking a particular path through the tree.

What's the runtime on a particular input?

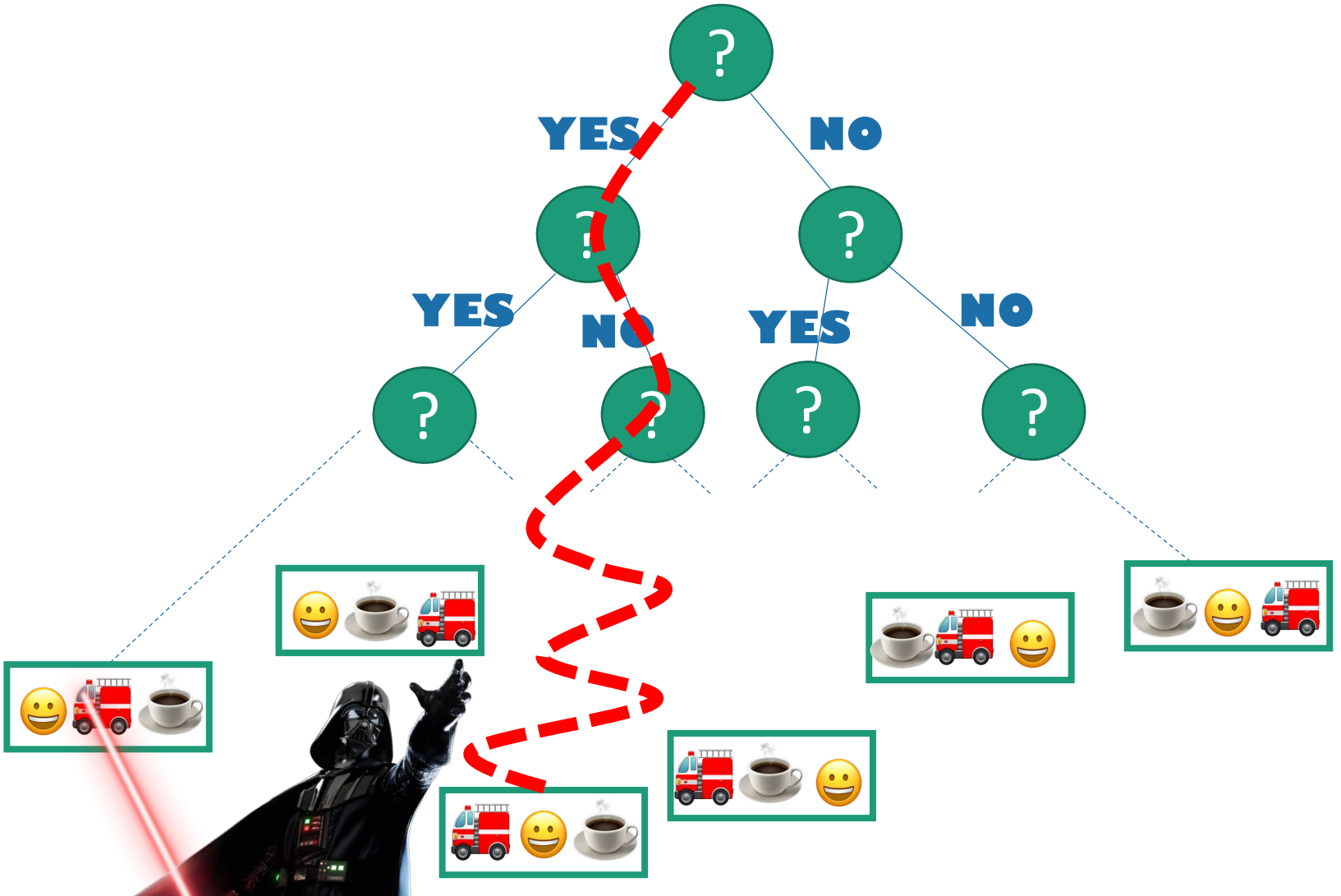
At least the number of comparisons that are made on that input.

If we take this path through the tree, the runtime is **Ω(length of the path).**



What's the worst-case runtime?

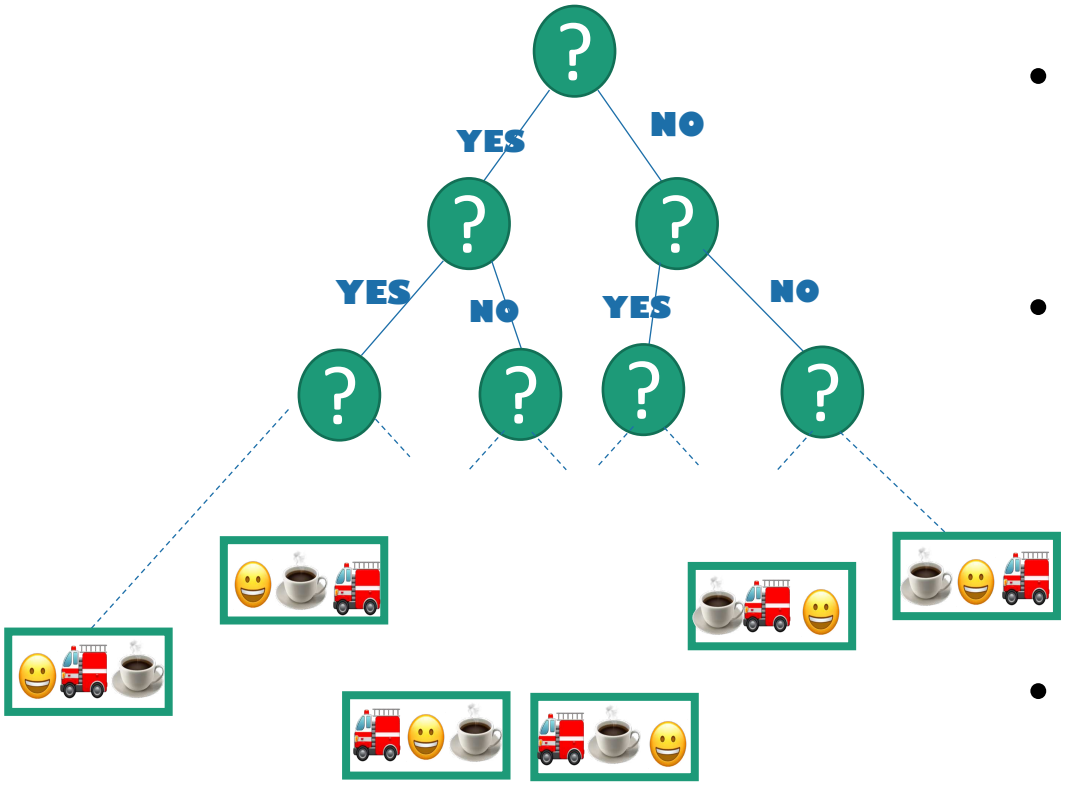
At least Ω(length of the longest path).



How long is the longest path?

being sloppy about floors and ceilings!

We want a statement: in all such trees, the longest path is at least _____



- This is a binary tree with at least _____ leaves.

n!

- The shallowest tree with n! leaves is the completely balanced one, which has depth _____.

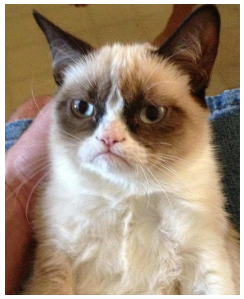
log(n!)

- So in all such trees, the longest path is at least log(n!).

- n! is about $(n/e)^n$ (Stirling's formula).
- log(n!) is about n log(n/e) = Ω(n log(n)).

Conclusion: the longest path has length at least Ω(n log(n)).

Lower bound of $\Omega(n \log(n))$.

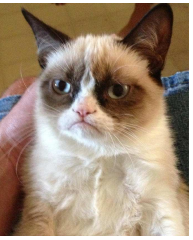


- Theorem:
  - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

- Proof:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.
  - The worst-case running time is at least the depth of the decision tree.
  - All decision trees with n! leaves have depth $\Omega(n \log(n))$.
  - So any comparison-based sorting algorithm must have worst-case running time at least $\Omega(n \log(n))$.

Aside:
What about randomized algorithms?

- For example, QuickSort?
- Theorem:
  - Any randomized comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps in expectation.



- Proof:
  - at the end of today if time
  - otherwise see lecture notes
  - (same ideas as deterministic case)

Try to prove this yourself!
We'll see this at the end of today's lecture if there's time.

Ollie the over-achieving ostrich

\end{Aside}

So, MergeSort is optimal!

- This is one of the cool things about lower bounds like this: we know when we can declare victory!

But what about StickSort?

- StickSort can't be implemented as a comparison-based sorting algorithm. So these lower bounds don't apply.
- But StickSort was kind of dumb.

Especially if I have to spend time cutting all those sticks to be the right size!

But might there be another model of computation that's less dumb, in which we can sort faster?

Beyond comparison-based sorting algorithms

Another model of computation

- The items you are sorting have meaningful values.

9	6	3	5	2	1	2

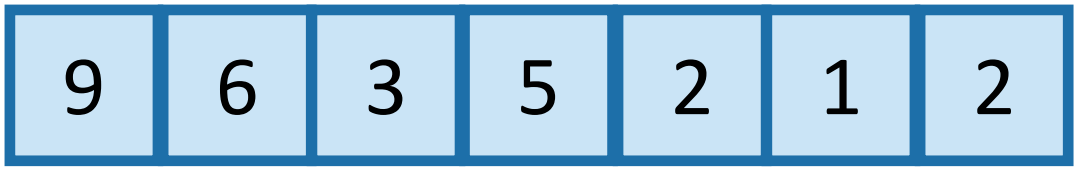
instead of

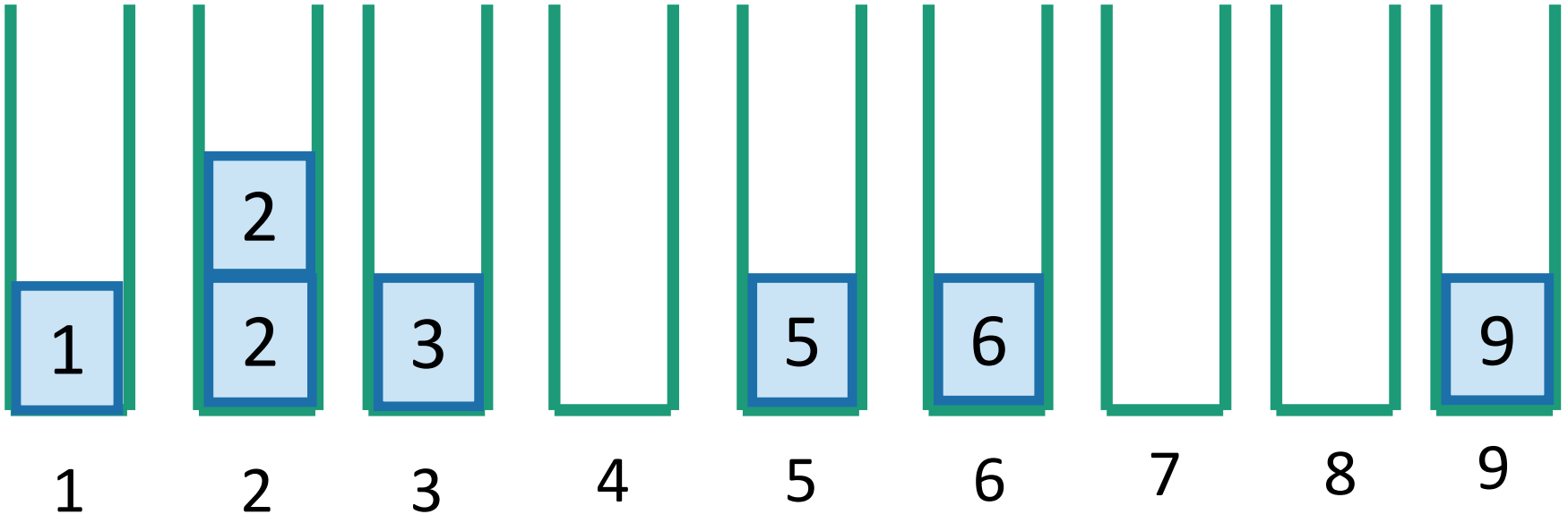
Why might this help?

Implement the buckets as linked lists. They are first-in, first-out.

BucketSort:

Note: this is a simplification of what CLRS calls "BucketSort"





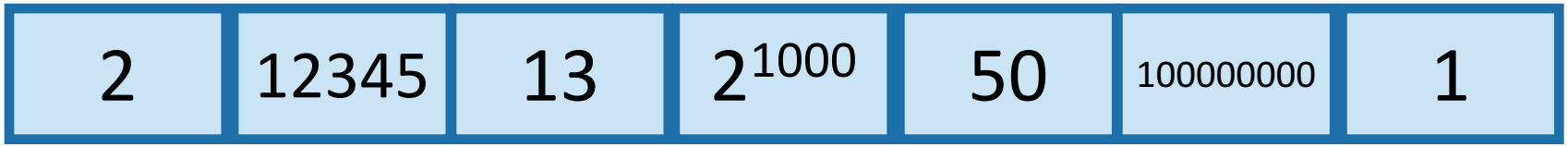
Concatenate the buckets!

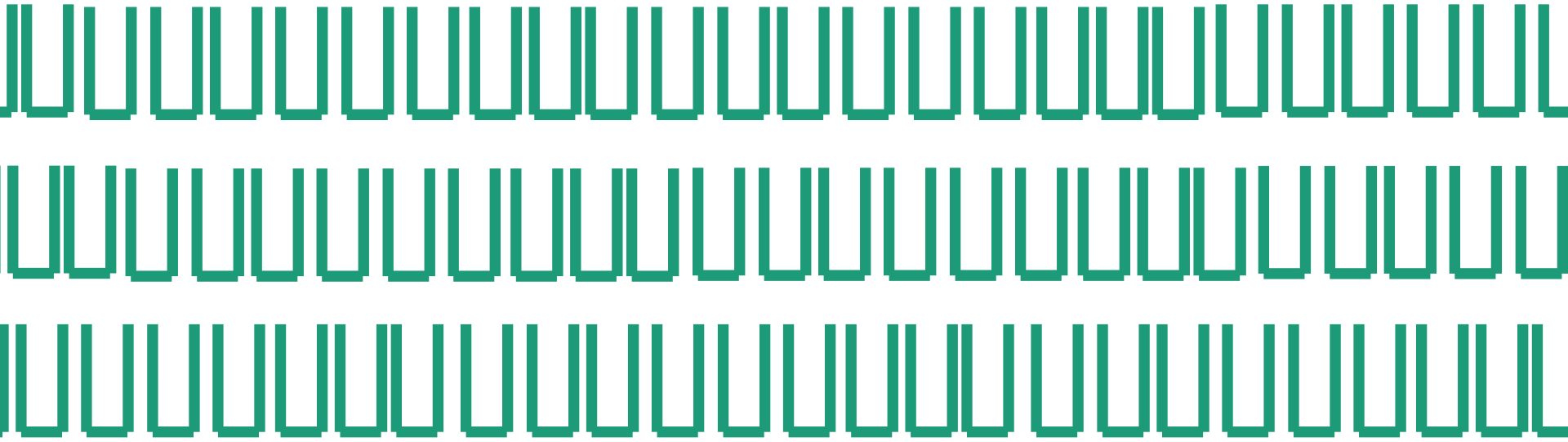
SORTED!

In time O(n).

Issues

- Need to be able to know what bucket to put something in.
  - That's okay for now: it's part of the model.
- Need to know what values might show up ahead of time.

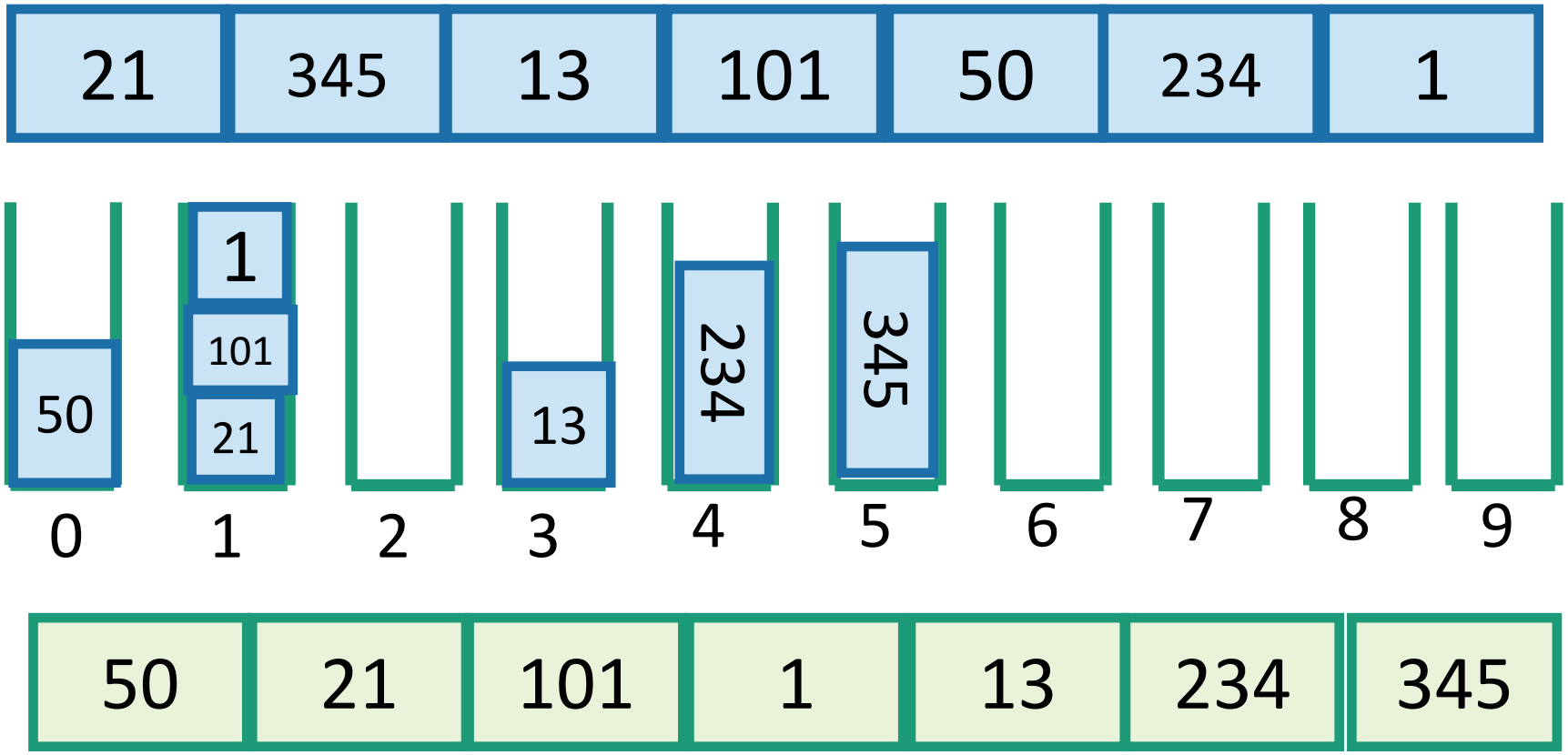




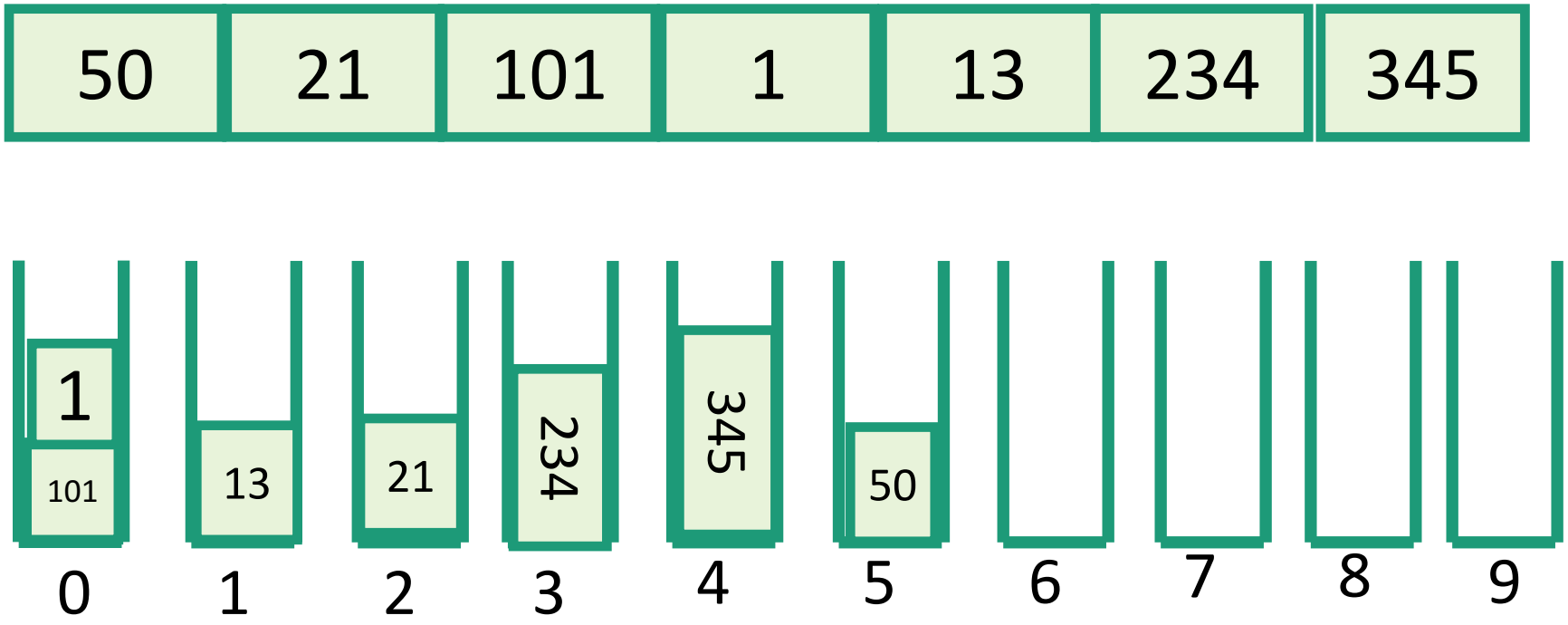
One solution: RadixSort

- Idea: BucketSort on the least-significant digit first, then the next least-significant, and so on.

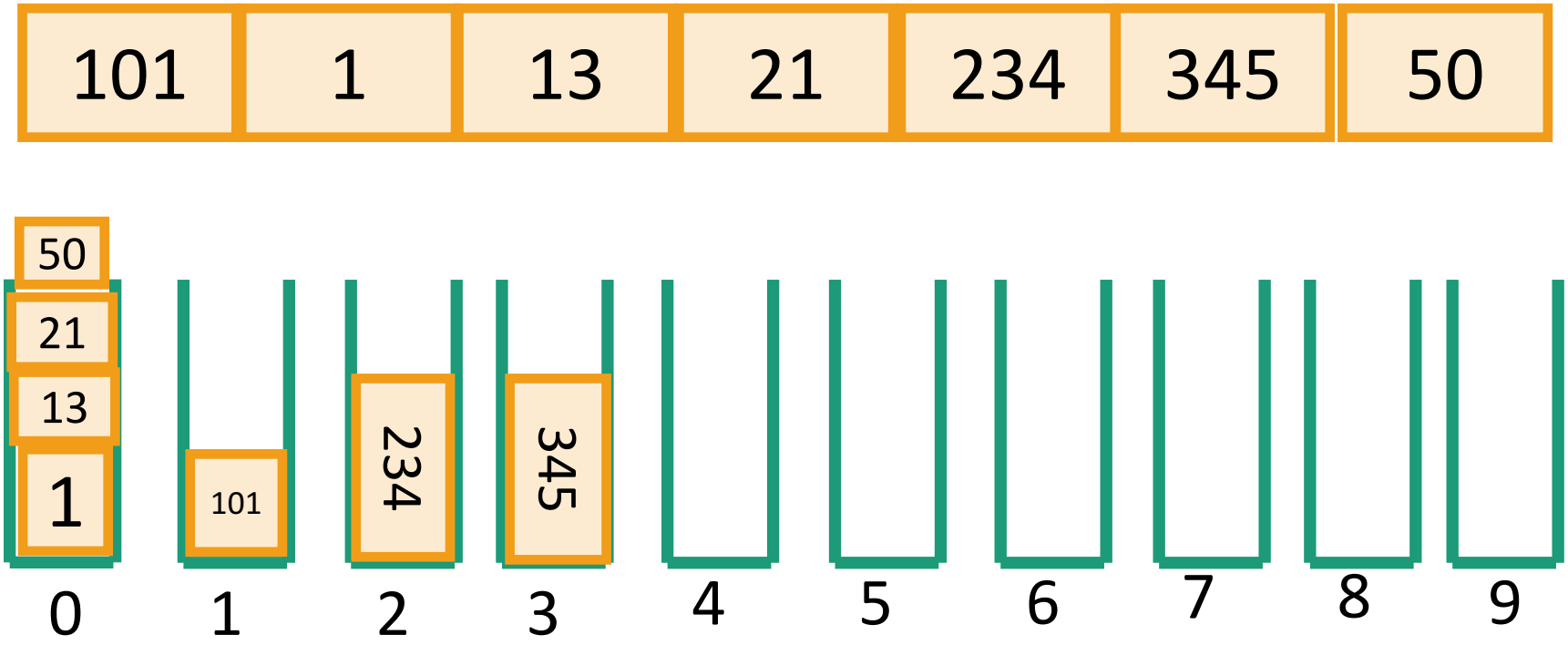
Step 1: BucketSort on LSB:



Step 2: BucketSort on the 2nd digit



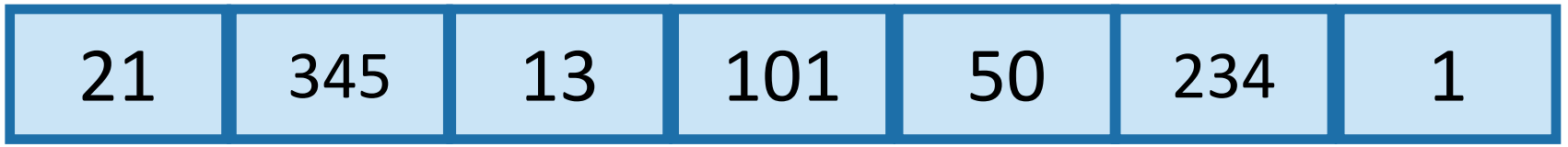
Step 3: BucketSort on the 3rd digit



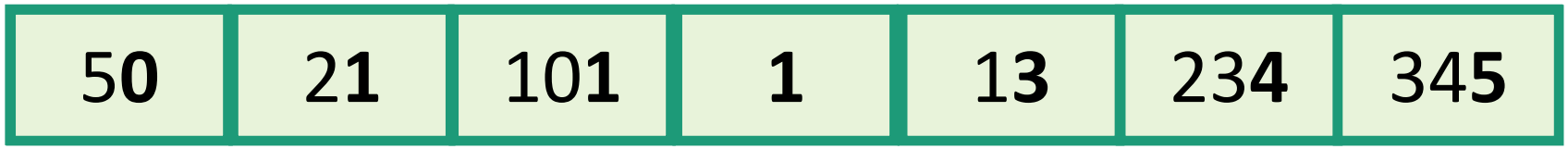
It worked!!

Why does this work?

Original array:



Next array is sorted by the first digit.



Next array is sorted by the first two digits.

Next array is sorted by all three digits.

Sorted array

Formally…

Lucky the lackadaisical lemur

This is the **outline** of a proof, not a formal proof.

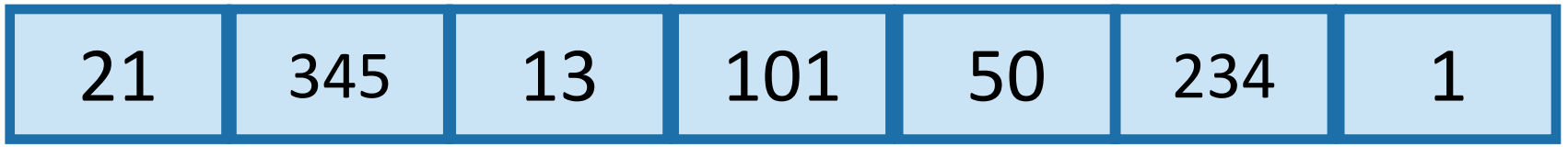
Make this formal! (or see lecture notes).

Ollie the over-achieving ostrich

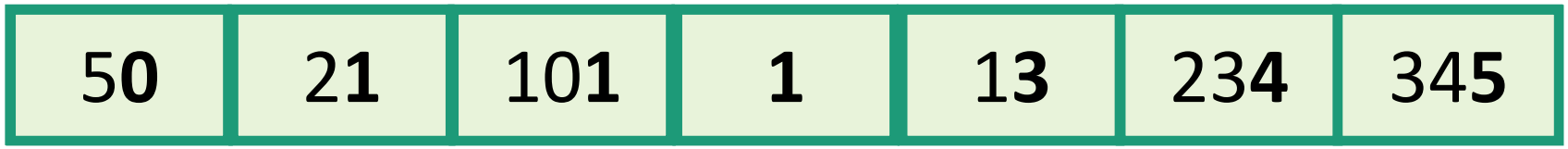
- Argument via loop invariant (aka induction).
- Loop Invariant:

Why does this work?

Original array:



Next array is sorted by the first digit.



Next array is sorted by the first two digits.

Next array is sorted by all three digits.

Sorted array

Formally…

This is the **outline** of a proof, not a formal proof.

Lucky the lackadaisical lemur

Make this formal! (or see lecture notes).

Ollie the over-achieving ostrich

- Argument via loop invariant (aka induction).
- Loop Invariant:
  - After the k'th iteration, the array is sorted by the first k least-significant digits.
- Base case:
  - "Sorted by 0 least-significant digits" means not sorted.
- Inductive step:
  - (You fill in…)
- Termination:
  - After the d'th iteration, the array is sorted by the d least-significant digits. Aka, it's sorted.

This needs to use: (1) bucket sort works, and (2) we treat each bucket as a FIFO queue.

Plucky the pedantic penguin

What is the running time?

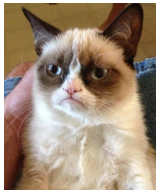
- Depends on how many digits the biggest number has.
  - Say d-digit numbers.
- There are d iterations
- Each iteration takes time $O(n + 10)$
  - We can change the 10 into an "r:" this is the "radix"
  - Example: if r = 2, we write everything in binary and only have two buckets.
  - Example: If r = 10000000, we write everything base-10000000 and have 10000000 buckets.
  - Example: if r = n, we write everything in base-n and have n buckets.
- Time is $O(d(n+r))$.
- If $d = O(1)$ and $r = O(n)$, running time $O(n)$.

How big can the biggest number be if d = O(1) and r = n?

So this is a O(n)-time sorting algorithm!

The story so far

- If we use a comparison-based sorting algorithm, it MUST run in time $\Omega(n\log(n))$.



- If we assume that we can do a little more than compare the values, we have an $O(n)$-time sorting algorithm.

Why would we ever use a comparison-based sorting algorithm??

Why would we ever use a comparison-based sorting algorithm?

- d might not be "constant." (aka, it might be big)

π	$\frac{123456}{987654}$	e	140!	2.1234123	2^n	42

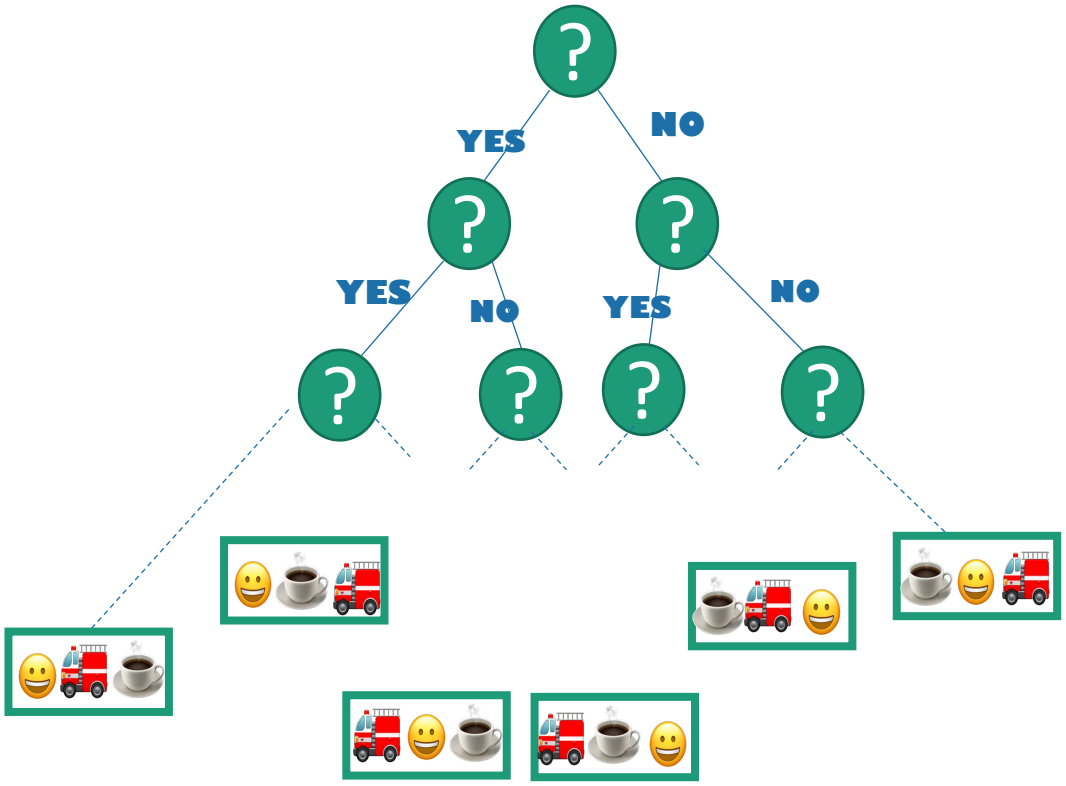
  - We can compare these pretty quickly (just look at the most-significant digit):
    - π = 3.14….
    - e = 2.78….
  - But to do RadixSort we'd have to look at every digit.
  - This is especially problematic since both of these have infinitely many digits...

- RadixSort needs extra memory for the buckets.
  - Not in-place
- I want to sort emoji by talking to a genie.
  - RadixSort makes more assumptions on the input.

Do we have time for the lower bound on randomized algorithms?

If so…

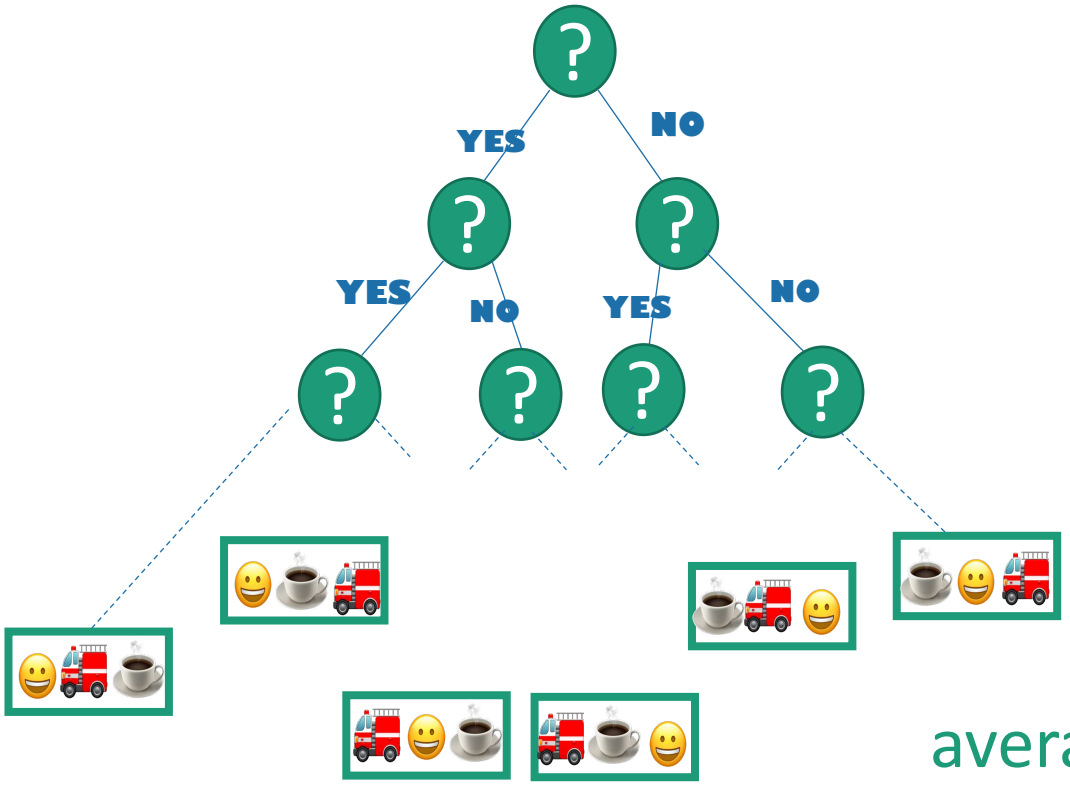
- Recall the lower bound for a deterministic algorithm.



- The longest path in this tree has length $\Omega(n\log(n))$.
- The running time of the algorithm is at least the length of this path.

A different model

- How about a deterministic algorithm on a random input?
- Not worst-case model.
- Not our randomized algorithm model either.



average

- The ~~longest~~ path in this tree has length $\Omega(n\log(n))$.

average

- The running time of the algorithm is at least the average length of this path.

So a deterministic algorithm must take time $\Omega(n\log(n))$ **even on random inputs.**

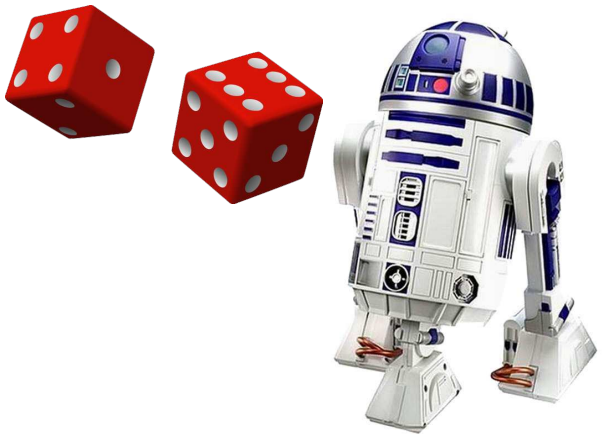
This is a pretty strong statement!

- Before:

  If an adversary gets to pick the input, we need time Ω(nlog(n)).

- Now:

  If the input is chosen randomly, we **still** need time Ω(nlog(n)).



But what does that model have to do with anything?

The argument here is pretty subtle! Understand why it makes sense!

It turns out that in this case,

Deterministic algorithm on random input

does at least as well as

Randomized algorithm on random input

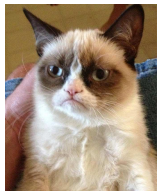
does at least as well as

Randomized algorithm on worst-case input

And we just showed that this didn't do very well.

A deterministic algorithm must take time $\Omega(n\log(n))$ **even on random inputs.**

A randomized algorithm must take time $\Omega(n\log(n))$ **on worst-case inputs.**



This is what we wanted.

Recap

- How difficult a problem is depends on the model of computation.
- How reasonable a model of computation is is up for debate.

- StickSort can sort sticks in O(1) time.
- RadixSort can sort smallish integers in O(n) time.
- If we want to sort emoji (or arbitrary-precision numbers), we require Ω(nlog(n)) time (like MergeSort).

Next Time

- Binary search trees