

# Lecture 6

Sorting lower bounds and  $O(n)$ -time sorting

# Announcements

- HW2 due **Friday**
- HW3 posted **Friday**
- Please send any OAE letters to Jessica Su (jtysu) ASAP.

# Sorting

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

*Can we do better?*

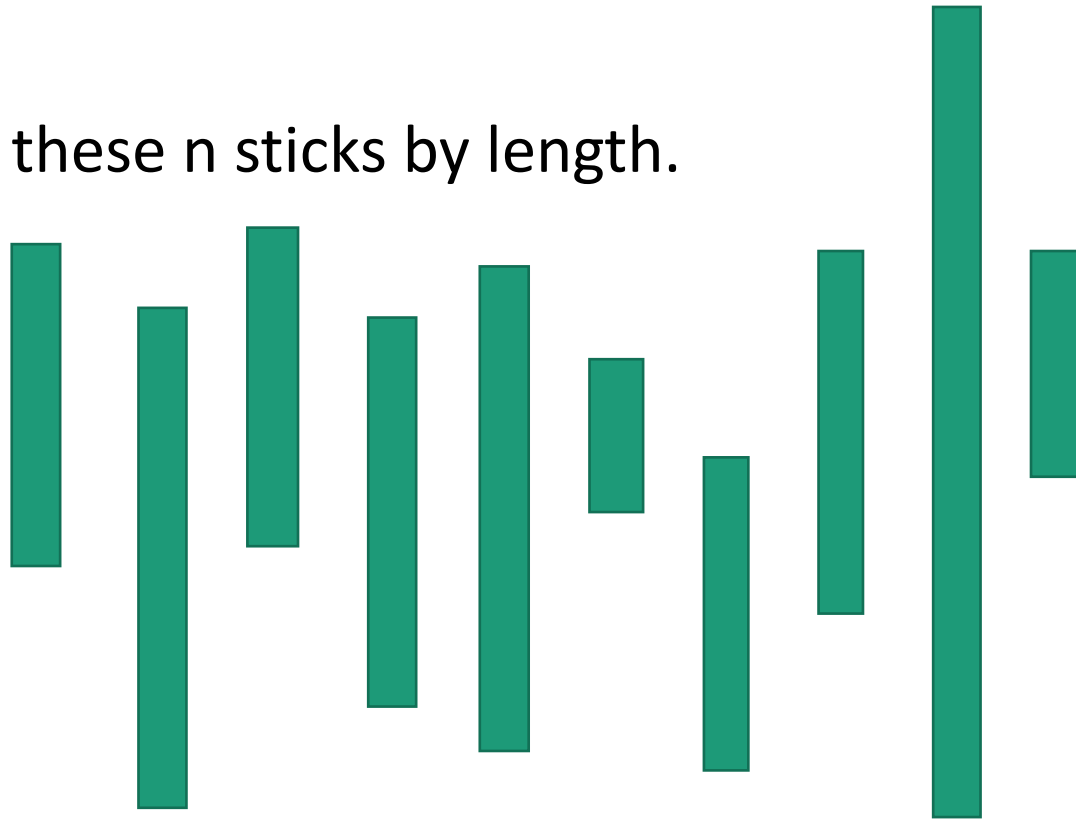
Depends on who  
you ask...





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

- Problem: sort these  $n$  sticks by length.



- Now they are sorted this way.

- Algorithm:
  - ↓ Drop them on a table.



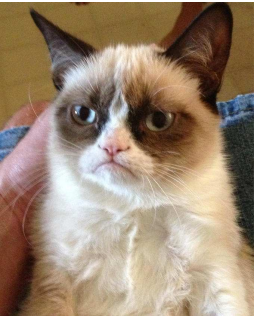
# That may have been unsatisfying

- 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

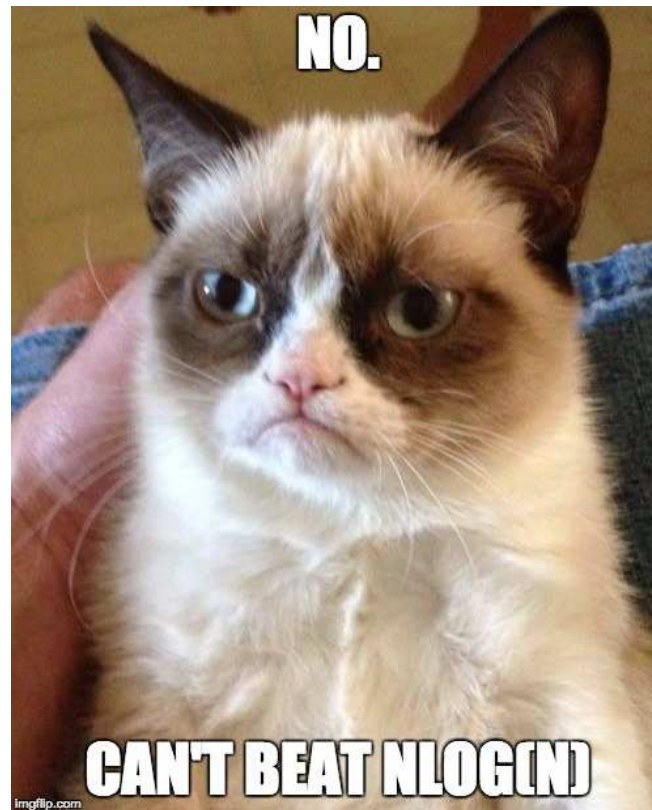


- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - We'll see that any algorithm in this model must take at least  $\Omega(n \log(n))$  steps.




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

# Comparison-based sorting



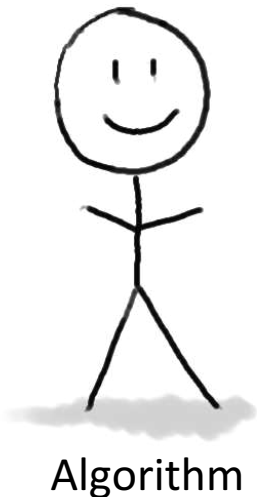
# Comparison-based sorting algorithms



 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  ?



**YES**

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



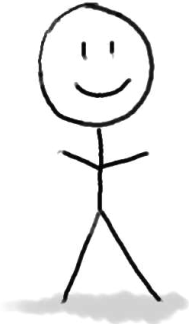
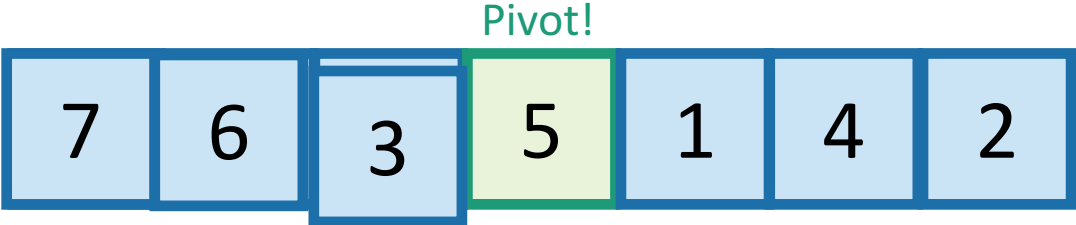
There is a **genie** who knows what  
the right order is.

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.

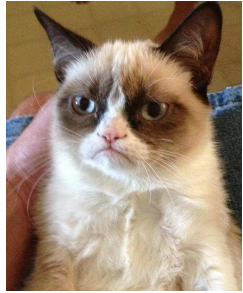
eg, QuickSort:



- Is **7** bigger than **5** ? **YES**
- Is **6** bigger than **5** ? **YES**
- Is **3** bigger than **5** ? **NO**



etc.



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

- Theorem:

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

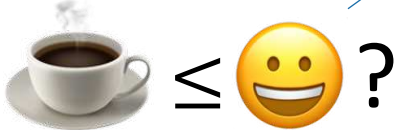


Sort these three things.



**YES**

**NO**



**YES**

**NO**



**YES**

**NO**



etc...

# All comparison-based algorithms look like this

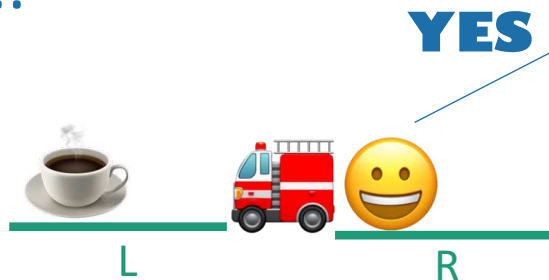
Pivot!



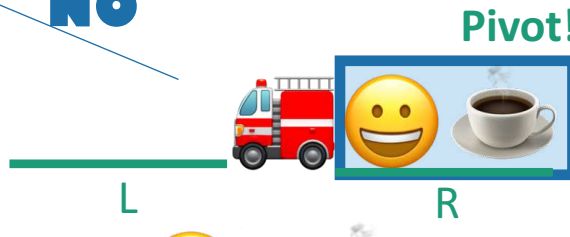
Example: Sort these three things using QuickSort.



etc...



NO



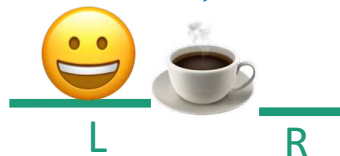
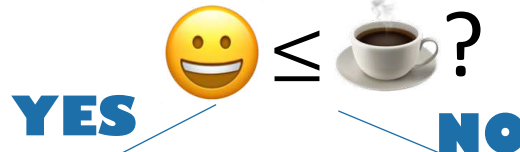
Pivot!

Then we're done (after some base-case stuff)

Return



Now recurse on R



Return



Return

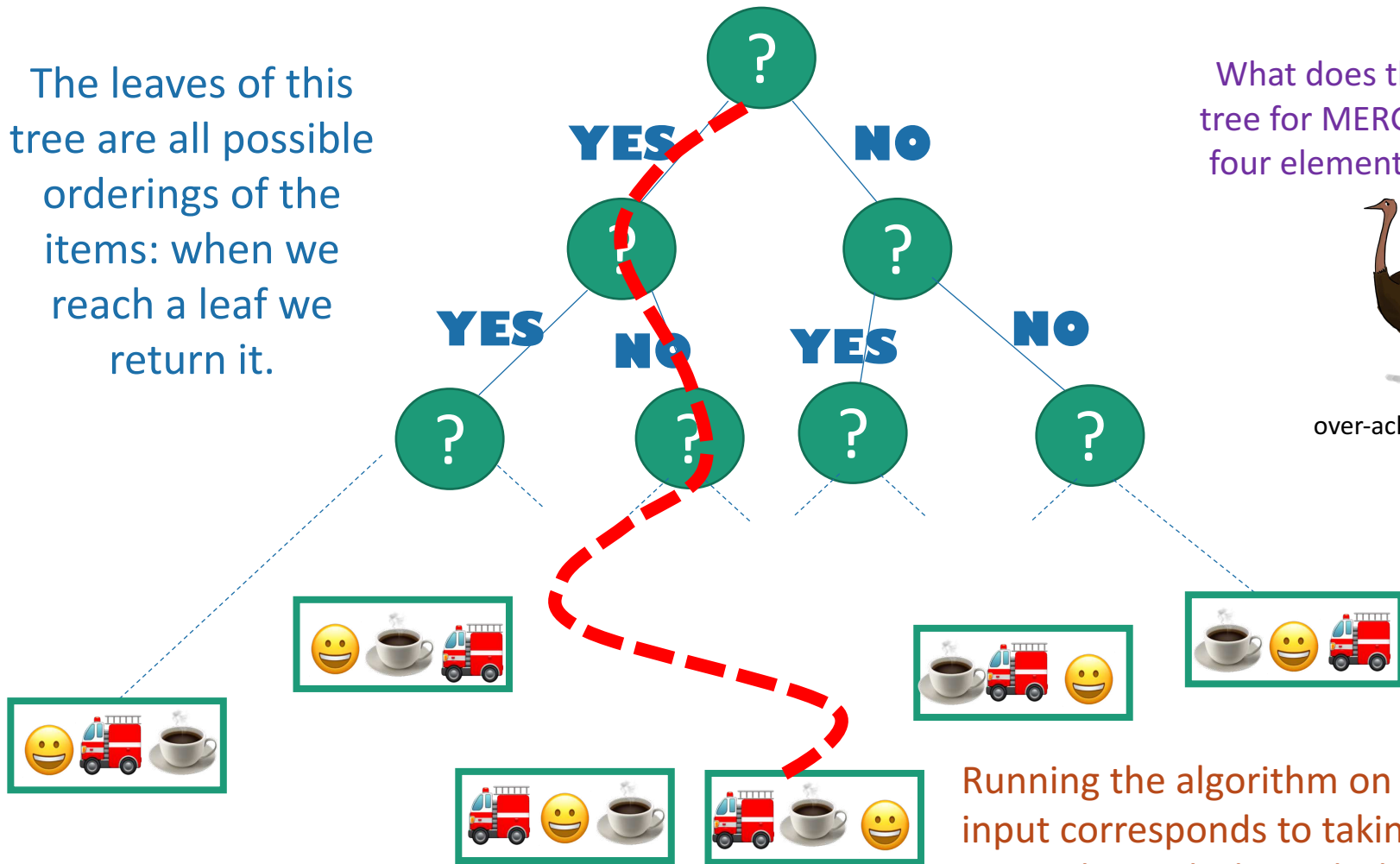


In either case, we're done (after some base case stuff and returning recursive calls).

# 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?

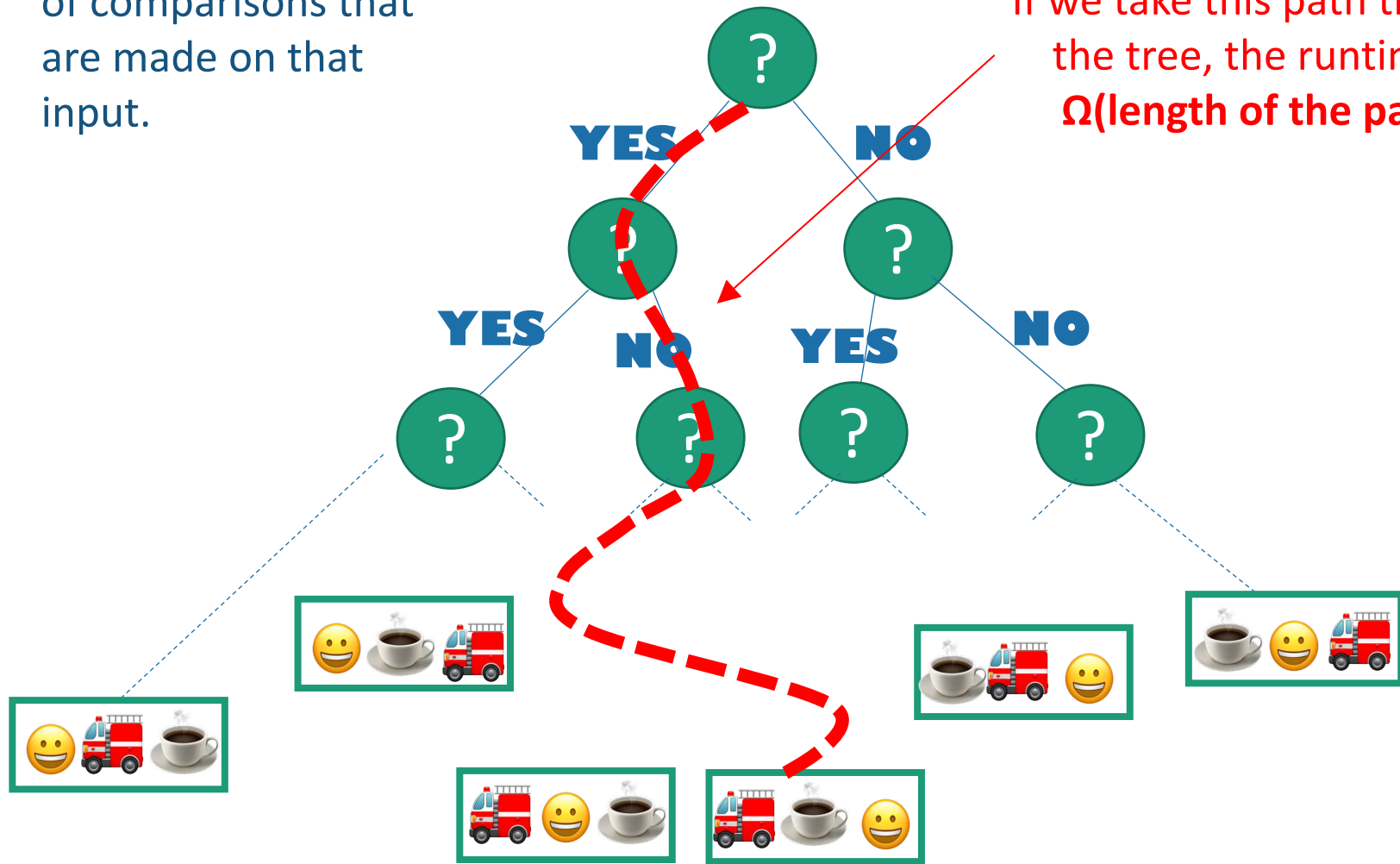


Ollie the over-achieving ostrich

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

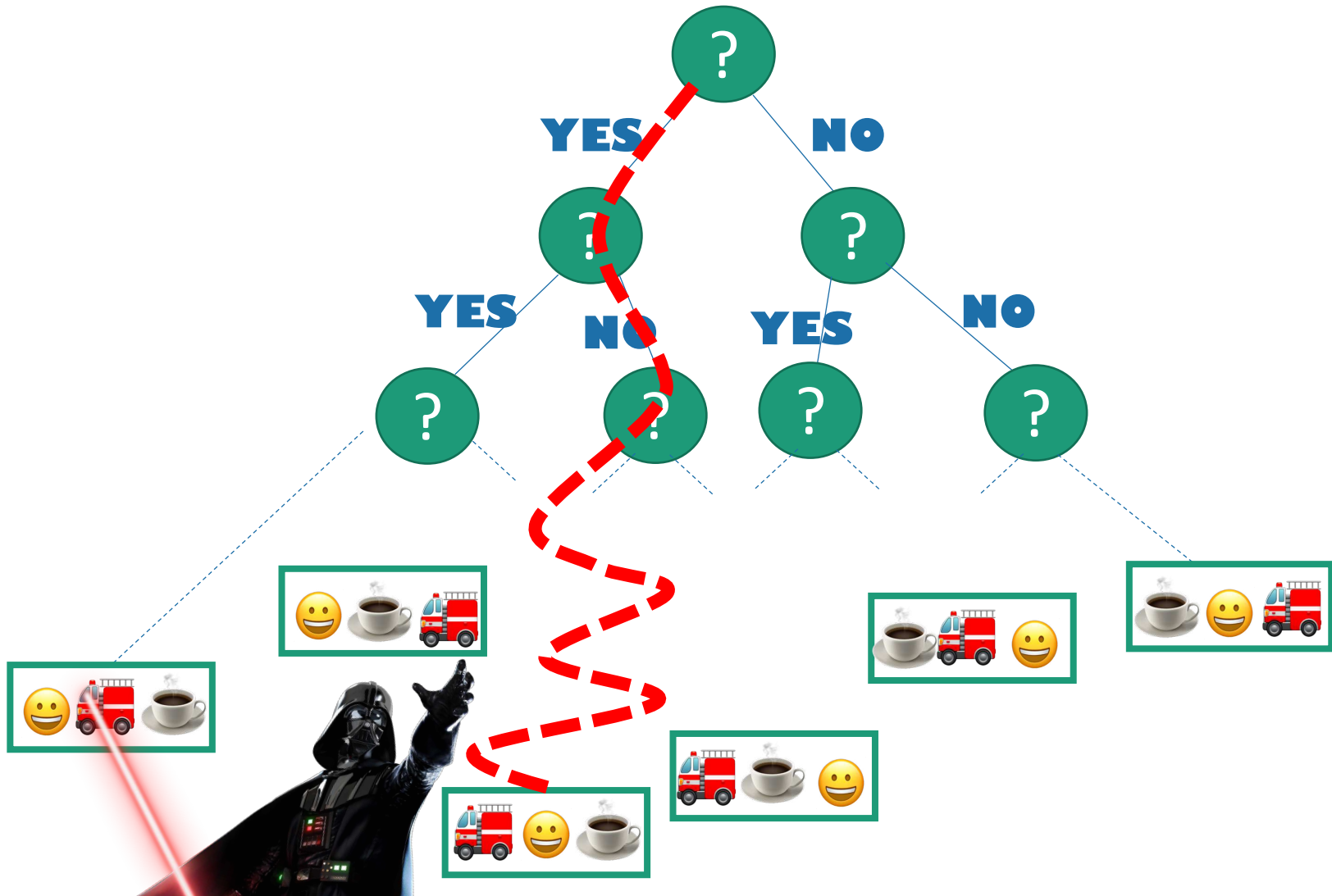
# What's the runtime on a particular input?

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



# What's the worst-case runtime?

At least  $\Omega(\text{length of the longest path})$ .

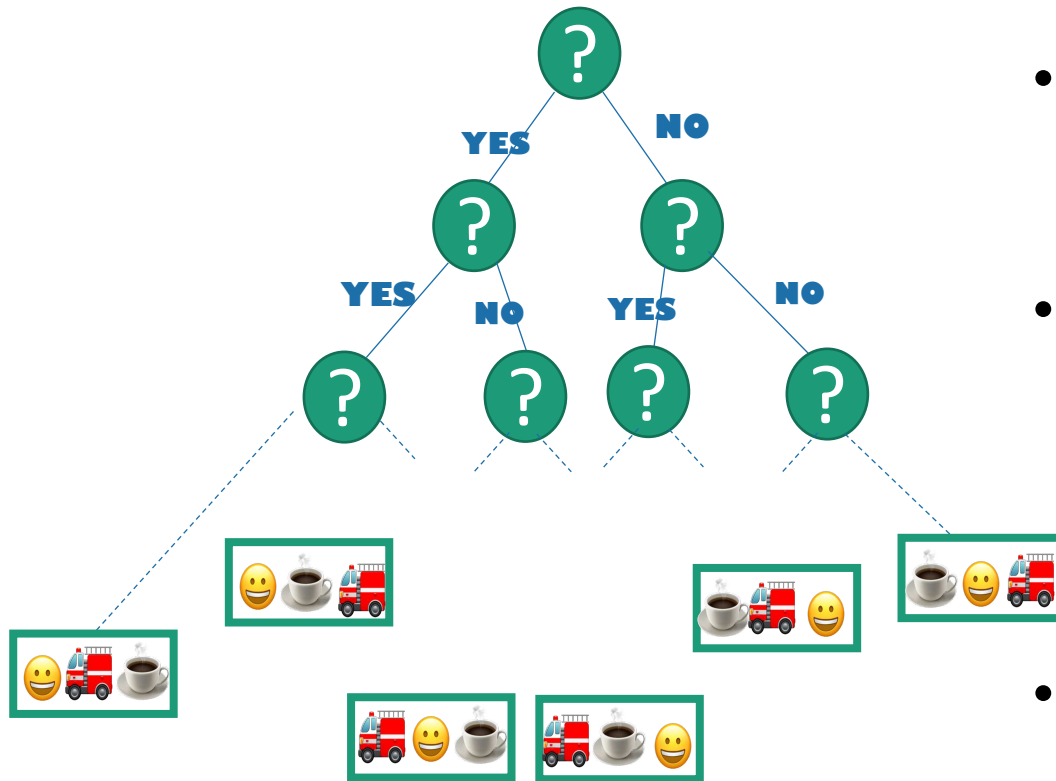




being sloppy about floors and ceilings!

# How long is the longest path?

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



- This is a binary tree with at least  $n!$  leaves.
- 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  $\log(n!)$ .

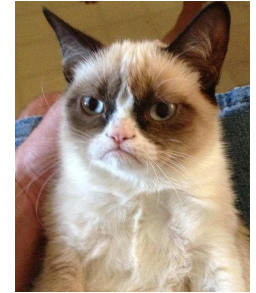
- $n!$  is about  $(n/e)^n$  (Stirling's approx.\*).
- $\log(n!)$  is about  $n \log(n/e) = \Omega(n \log(n))$ .

**Conclusion:** the longest path has length at least  $\Omega(n \log(n))$ .

\*Stirling's approximation is a bit more complicated than this, but this is good enough for the asymptotic result we want.



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

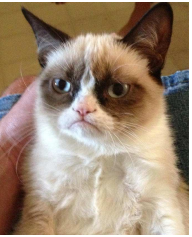


- Theorem:
  - Any deterministic comparison-based sorting algorithm must take  $\Omega(n \log(n))$  steps.
- Proof recap:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with  $n!$  leaves.
  - The worst-case running time is at least the depth of the decision tree.
  - 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:
  - see lecture notes
  - (same ideas as deterministic case)



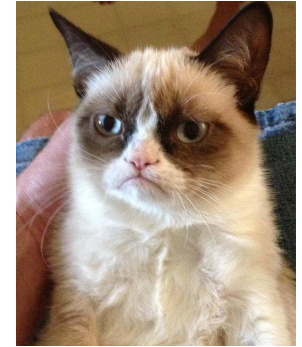
Try to prove this yourself!



`\end{Aside}`

Ollie the over-achieving ostrich

# So that's bad news.



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

But look on the bright side!

# 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.**

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

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



# Beyond comparison-based sorting algorithms



# Another model of computation

- The items you are sorting have **meaningful values**.



instead of



# Pre-lecture exercise

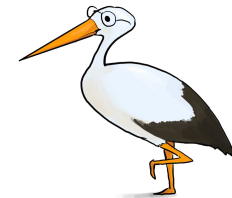
- Sorting CS161 students by their month of birth.
  - *[Discussion on board]*



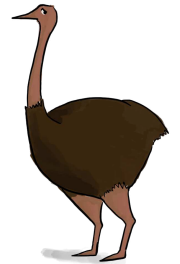
1



1



4



5

# Another model of computation

- The items you are sorting have **meaningful values**.



instead of





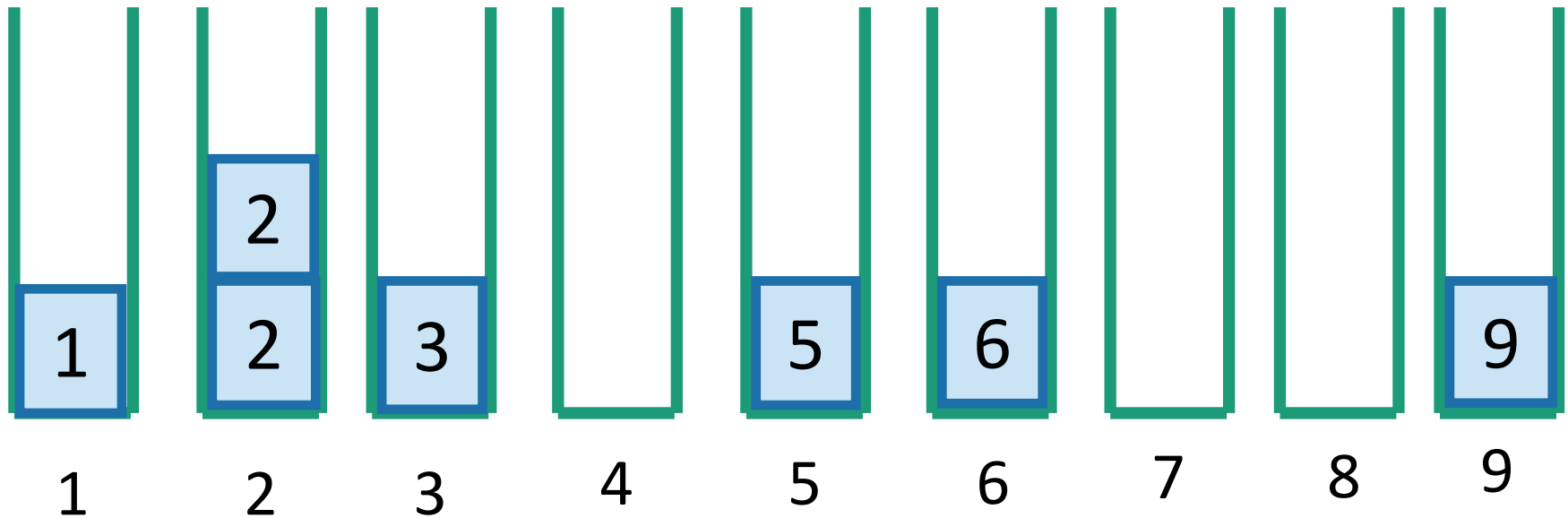
# Why might this help?



Implement the **buckets** as linked lists. They are first-in, first-out. This will be useful later.

## BucketSort:

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

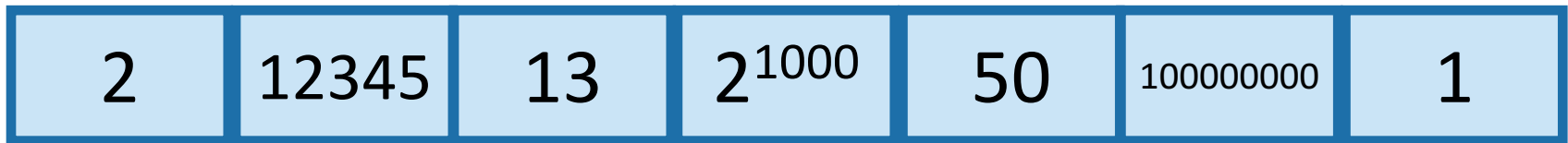


Concatenate the buckets!

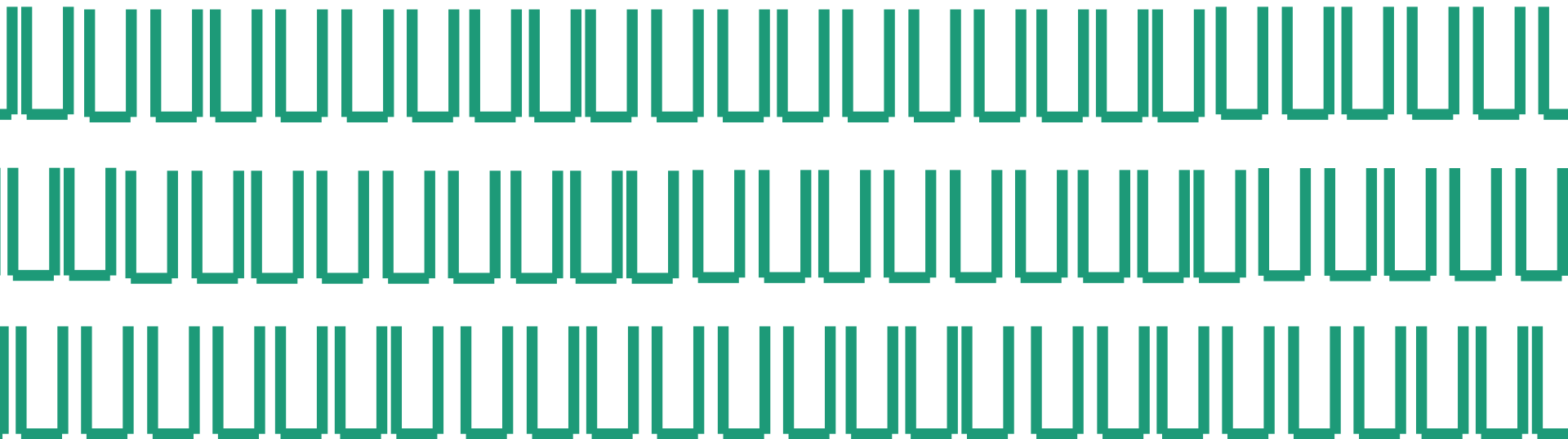
**SORTED!**  
In time  $O(n)$ .

# Issues

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



- Space...

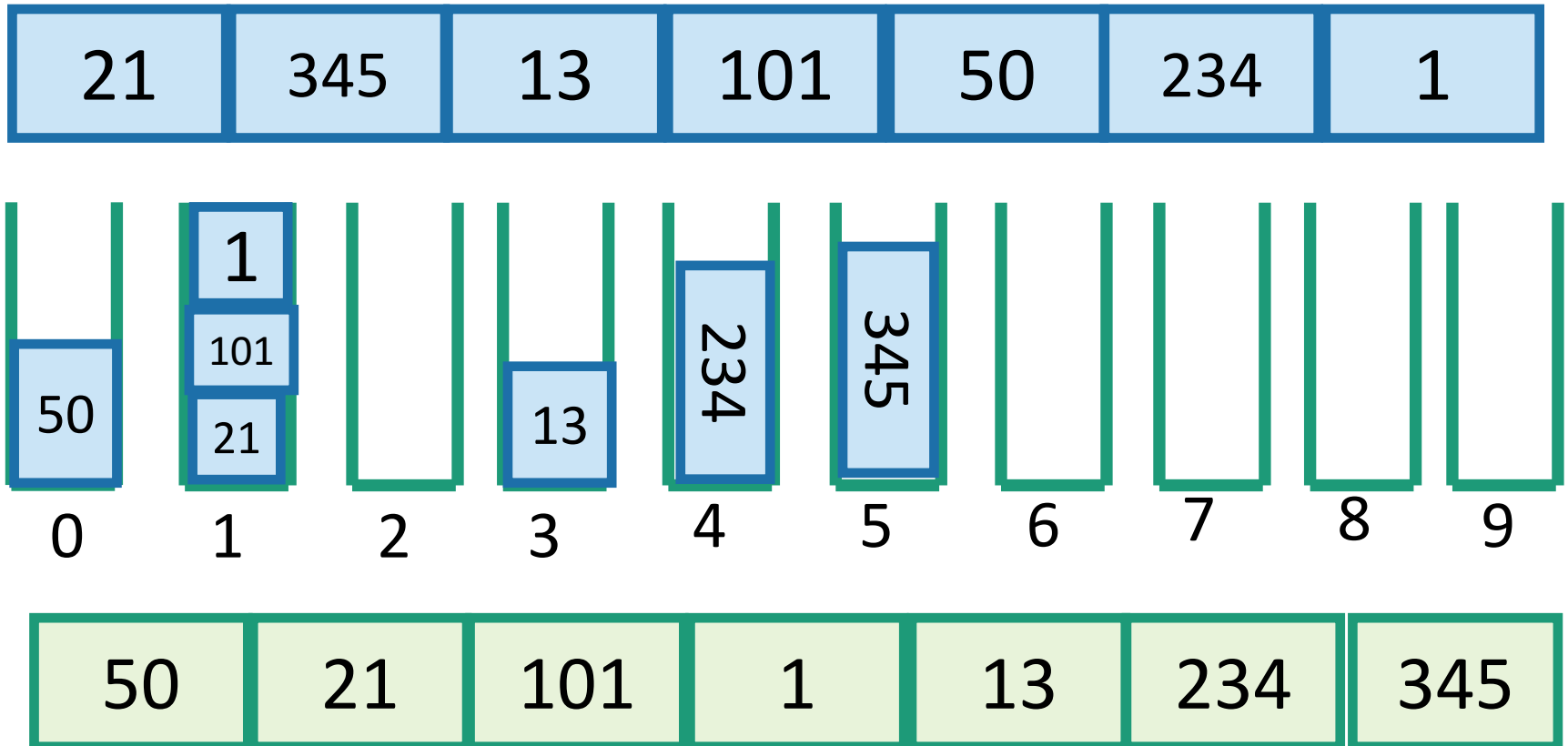


# One solution: RadixSort

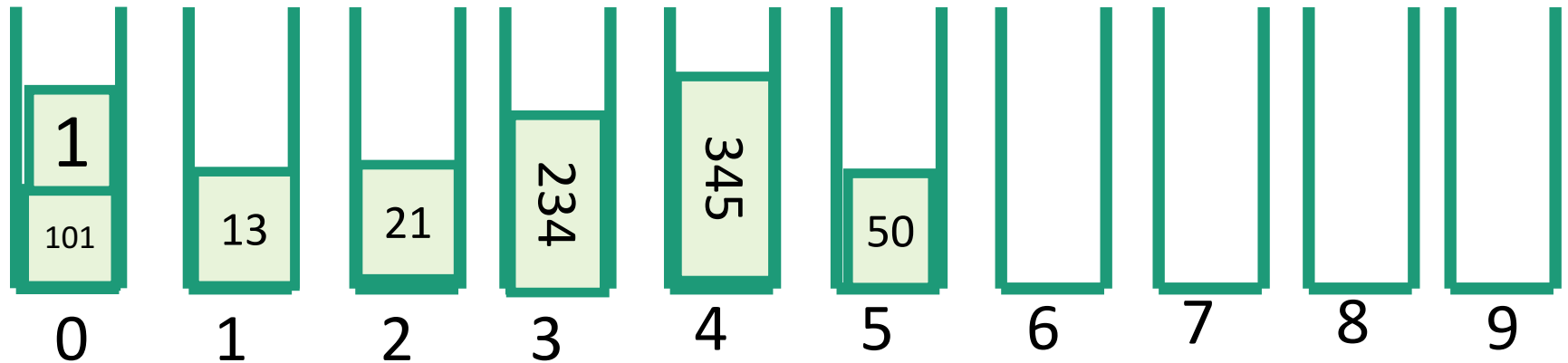
Say we're sorting integers.

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

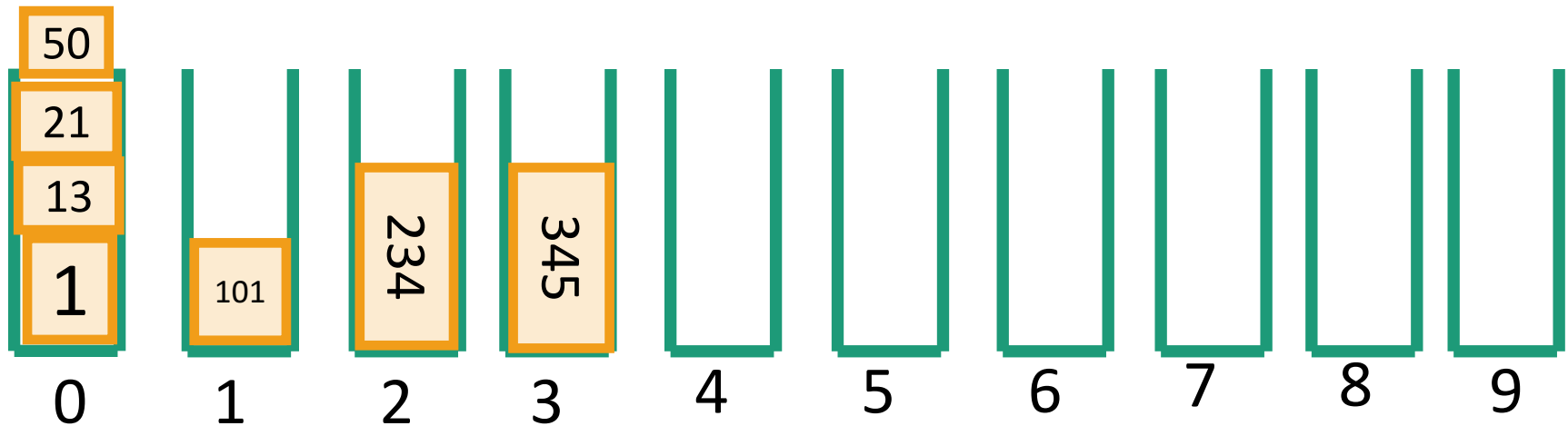
Step 1: BucketSort on LSB:



# Step 2: BucketSort on the 2<sup>nd</sup> digit



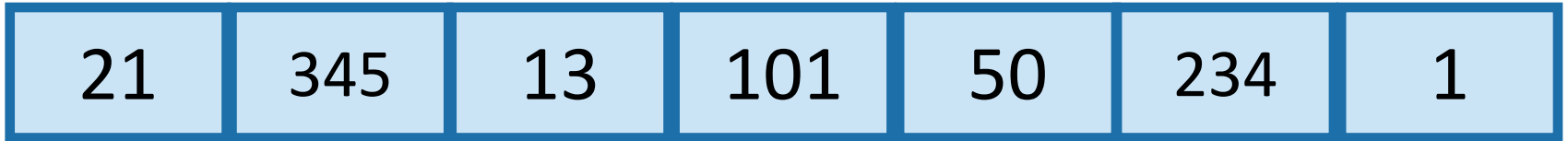
# Step 3: BucketSort on the 3<sup>rd</sup> digit



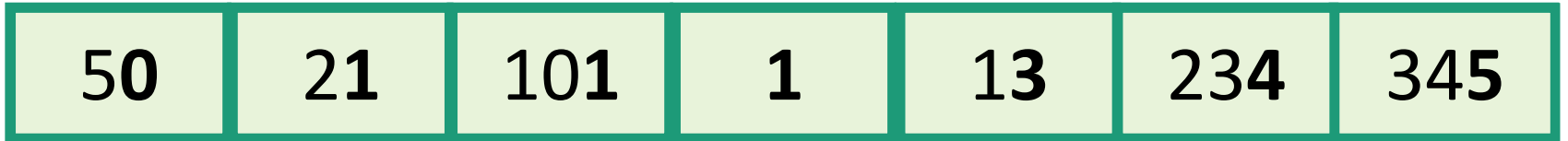
It worked!!

# 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...



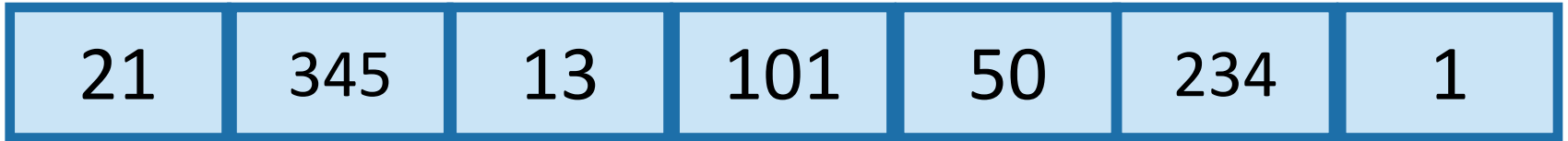
Or at least a  
little formally!

Lucky the lackadaisical lemur

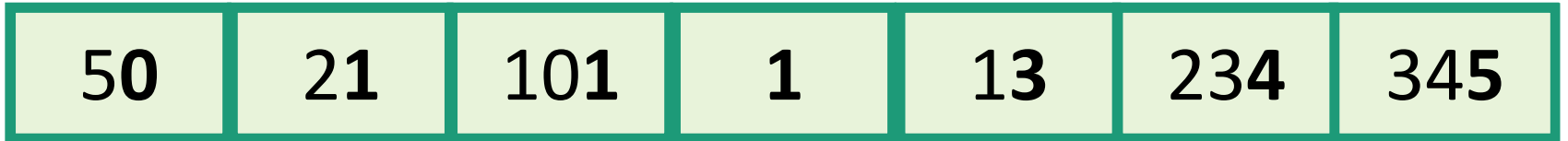
- Argue by induction.
- Inductive hypothesis:

# 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...



Or at least a little formally!

Lucky the lackadaisical lemur

- Argue by induction.
- Inductive hypothesis:
  - 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:
  - (See lecture notes or CLRS)
- Conclusion:
  - 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.\*



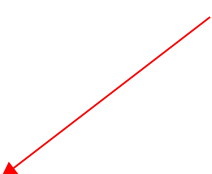
Plucky the pedantic penguin

\*the buzzword here is that bucketSort is *stable*.

# What is the running time?

- Say they are **d-digit** numbers.
  - There are  $d$  iterations.
  - Each iteration takes time  $O(n + 10) = O(n)$
- Total time:  $O(nd)$ .
- Say the biggest integer is  $M$ . What is  $d$ ?
  - $d = \lfloor \log_{10}(M) \rfloor + 1$
  - so  $O(nd) = O(n \log_{10}(M))$ .

The "10" is because we are working base 10.



Can we do better?  
what if  $M = n$ ?

# Trade-offs...

- RadixSort works with any base.
- Before we did it base **r=10**.
- But we could do it base **r=2** or **r=20** just as easily.
  - *[On board]*
  
- Running time for general r and M?
  - *[On board]*

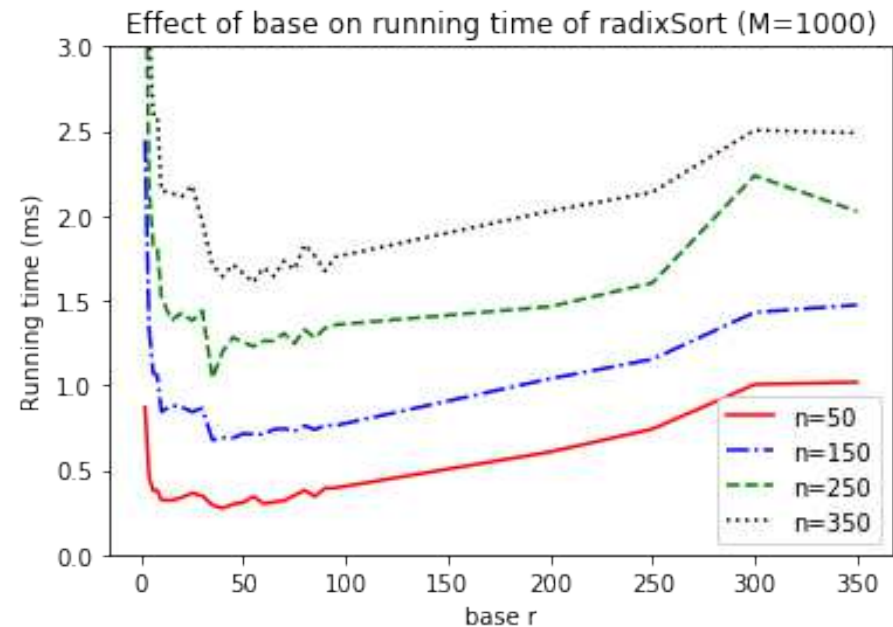
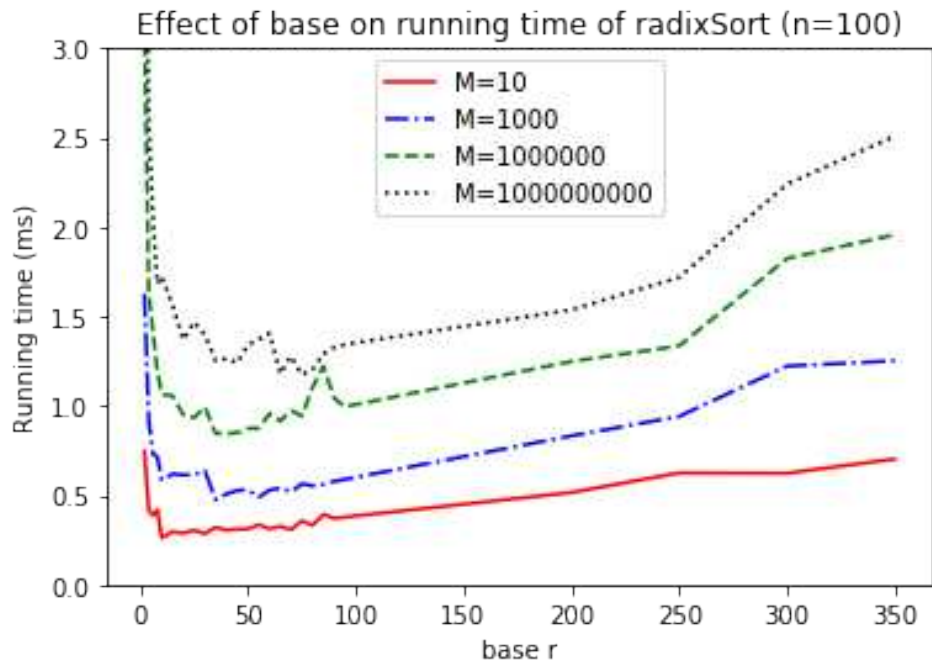


Running time:  $O((n + r) \cdot \lceil \log_r(M) \rceil)$

## Trade-offs ctd...

- There are  $n$  numbers, biggest one is  $M$ .
- What should we choose for  $r$  (in terms of  $M, n$ )?

*There's some sweet spot...* (and maybe it's growing with  $M$  and  $n$ ?)



# We get...

- *[Discussion on board...]*

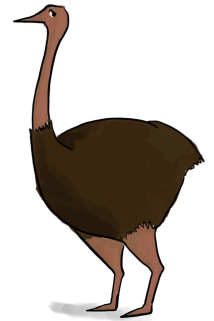


- If we choose  $r = n$ , running time is

$$T(n) = O(n \cdot \lceil \log_n(M) \rceil)$$

- If  $M = O(n)$ ,  $T(n) = O(n)$ . Awesome!
- If  $M = \Omega(n^n)$ ,  $T(n) = O(n^2)$ ...

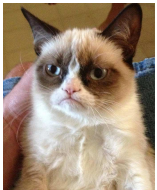
Choosing  $r = n$   
is pretty good.  
What's the *optimal*  
choice of  $r$ ?



Ollie the over-achieving ostrich

# The story so far

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



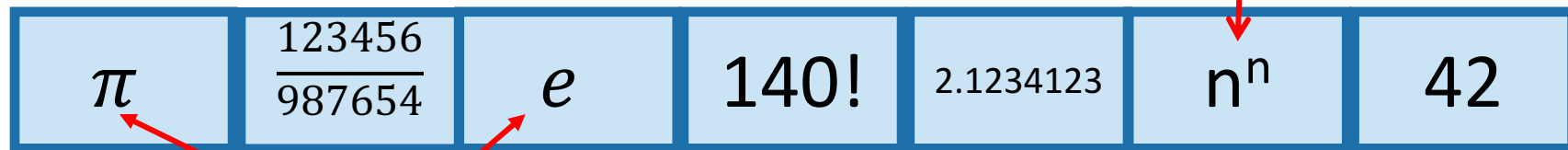
- If we assume a bit of structure on the values, we have an  $O(n)$ -time sorting algorithm.



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

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

- Lots of precision...



Even with integers, if the biggest one is really big, RadixSort is slow.

- We can compare these pretty quickly (just look at the most-significant digit):
  - $\pi = 3.14\dots$
  - $e = 2.78\dots$
- 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.

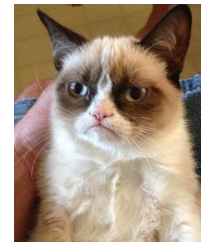


# Recap

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

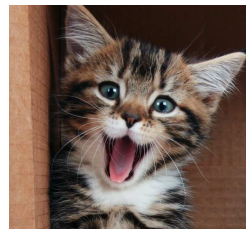
- Comparison-based sorting model

- This includes MergeSort, QuickSort, InsertionSort
- Any algorithm in this model must use at least  $\Omega(n \log(n))$  operations.



- But if we are sorting small integers (or other reasonable data):

- BucketSort and RadixSort
- Both run in time  $O(n)$





# Next time

- Binary search trees!
- Balanced binary search trees!
- Special guest lecturer: *Sam Kim!*

## Before next time

- Pre-lecture exercise for Lecture 7
  - Remember binary search trees?

**CHUCK NORRIS  
QUICKSORTS STICKS**



**IN TIME 0(1)**