

# Lecture 6

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

# Announcements

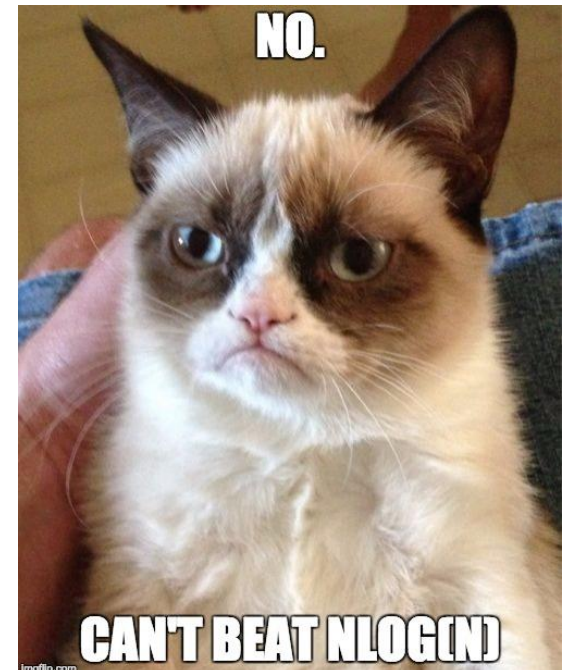
- Add/Drop deadline Friday (tomorrow).
- HW2 due Friday (also tomorrow).
  - Reminder: HW solutions must be typed
  - Reminder: ***Please*** tag problems on Gradescope!
- Exam logistics!
  - Exam I is 9-10:20am, Thursday 10/16
  - See Ed post (by Anisha) for locations and other info
  - Highlights:
    - Remember to bring your ID to the exam!
    - Please show up promptly at 9am! (If not earlier)
    - The actual “test-taking time” is **60 minutes**, to allow for exam distribution and collection.
    - Info about a review session coming soon

# 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?***

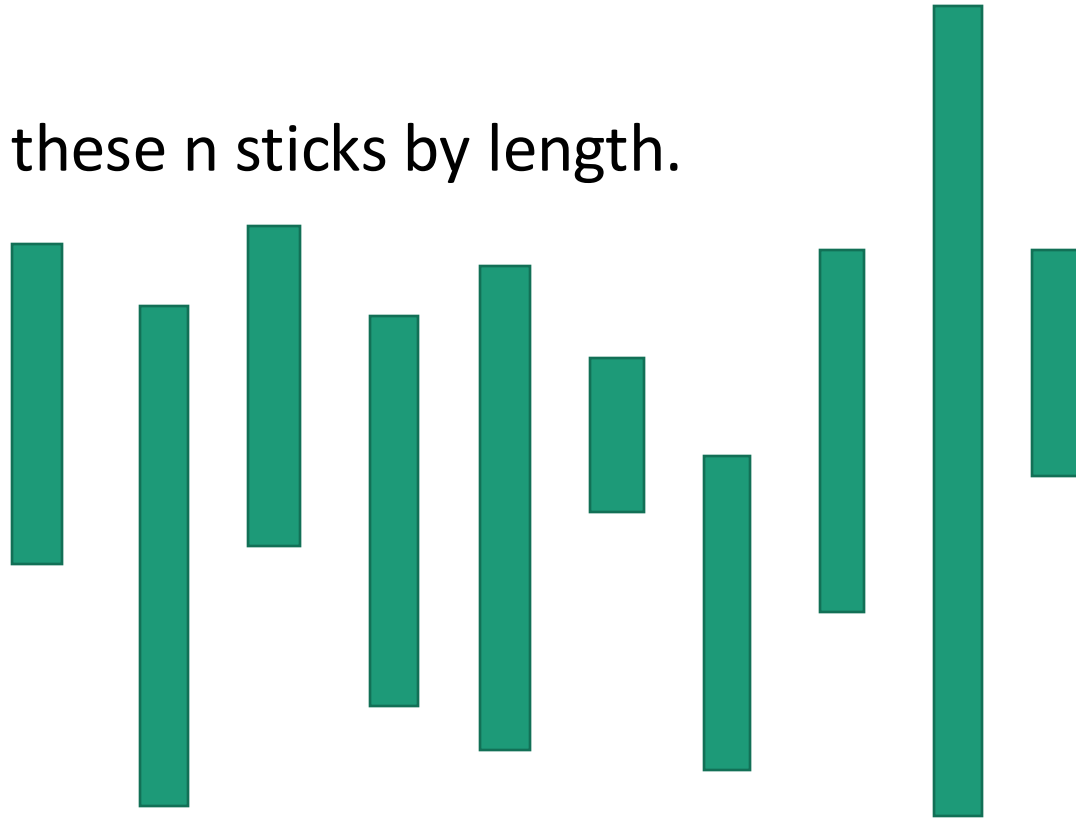
Depends on who  
you ask...





# 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: random access into the array; comparing two array elements

-VS-

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

# Today: two (more) models

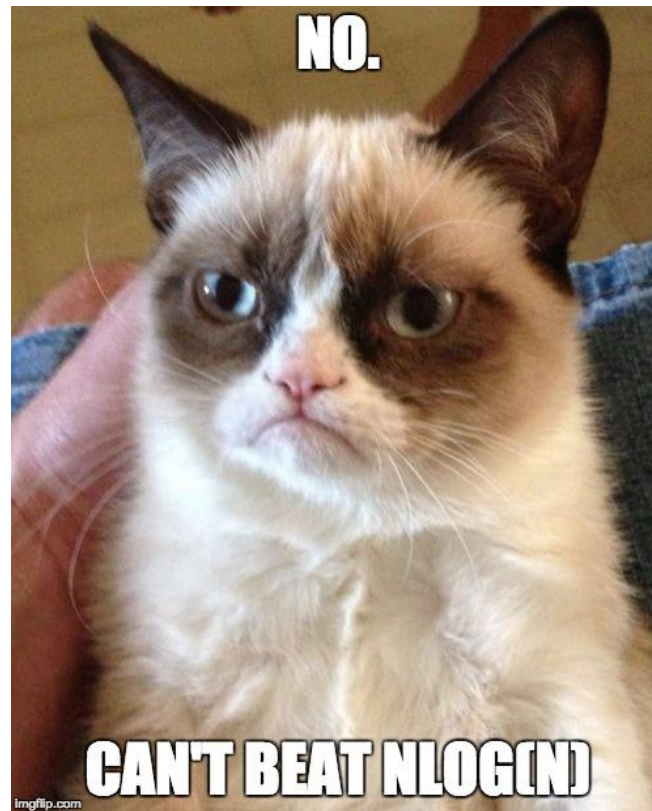


- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - We'll see that any algorithm in this model must take at least  $\Omega(n \log(n))$  steps.



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

# Comparison-based sorting



# Comparison-based sorting algorithms

- You want to sort an array of items.
- You can't access the items' values directly: you can only compare two items and find out which is bigger or smaller.



# Comparison-based sorting algorithms



is shorthand for  
“the first thing in the input list”

Want to sort these items.

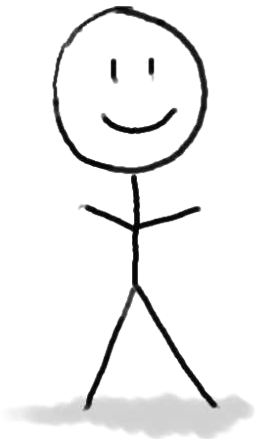
There's some ordering on them, but we don't know what it is.

Is  bigger than

?



**YES**



Algorithm

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

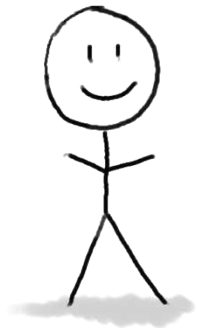
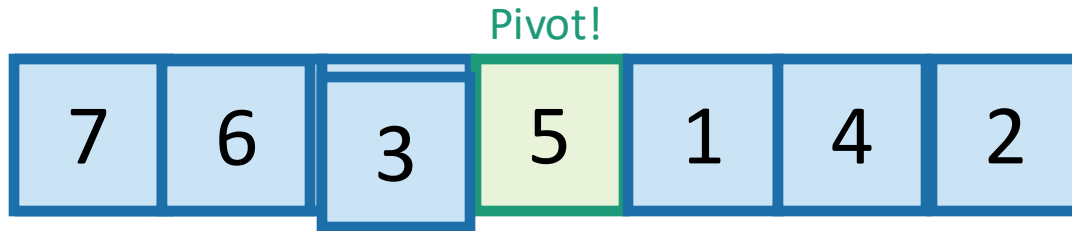


There is a **genie** who knows what  
the right order is.

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

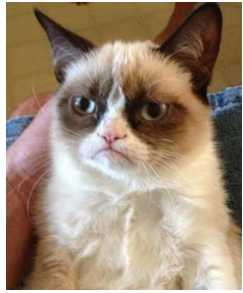
?

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**

etc...



**YES**

**NO**



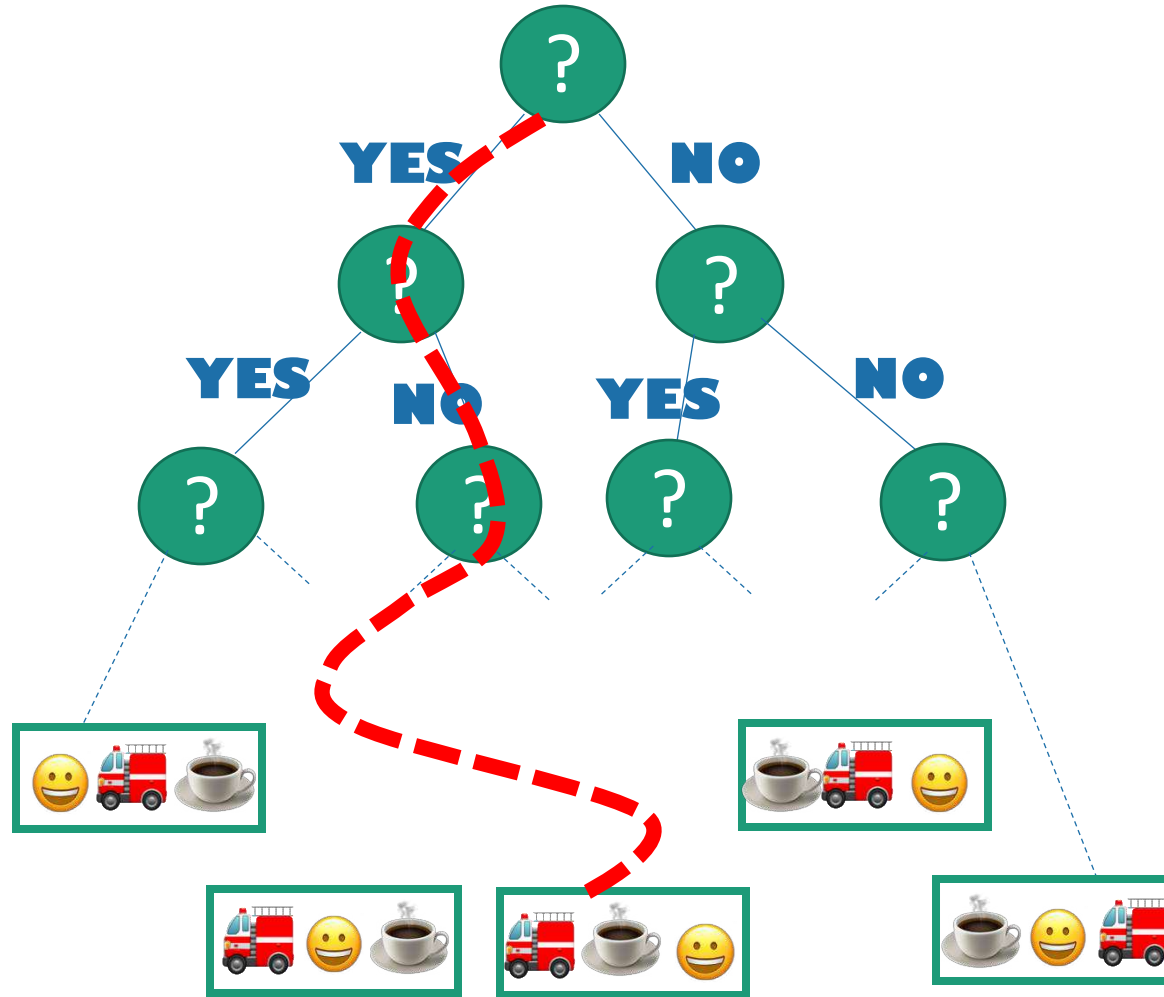
**YES**

**NO**



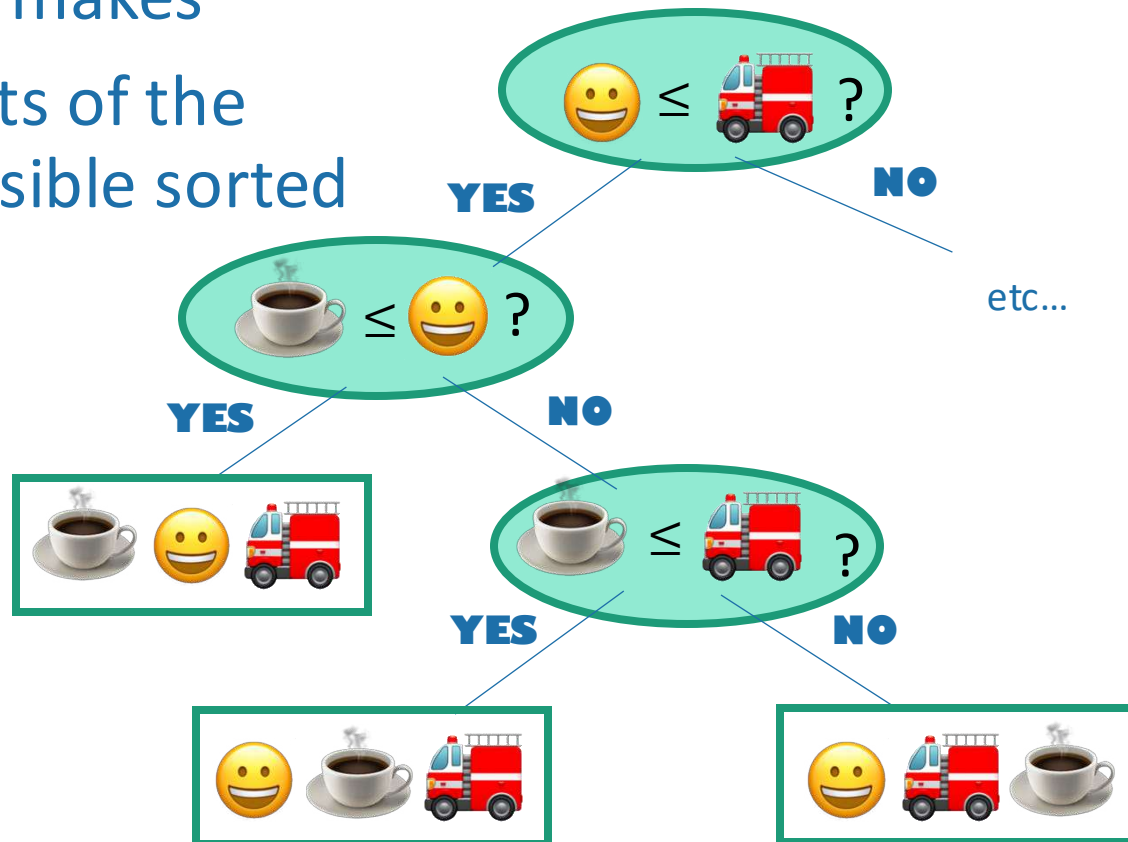
# Decision trees

- Internal nodes correspond to yes/no questions that the algorithm makes.
- Each internal node has two children, one for “yes” and one for “no.”
- Leaf nodes correspond to outputs of the algorithm.
- Running the algorithm on a particular input corresponds to a particular path through the tree.



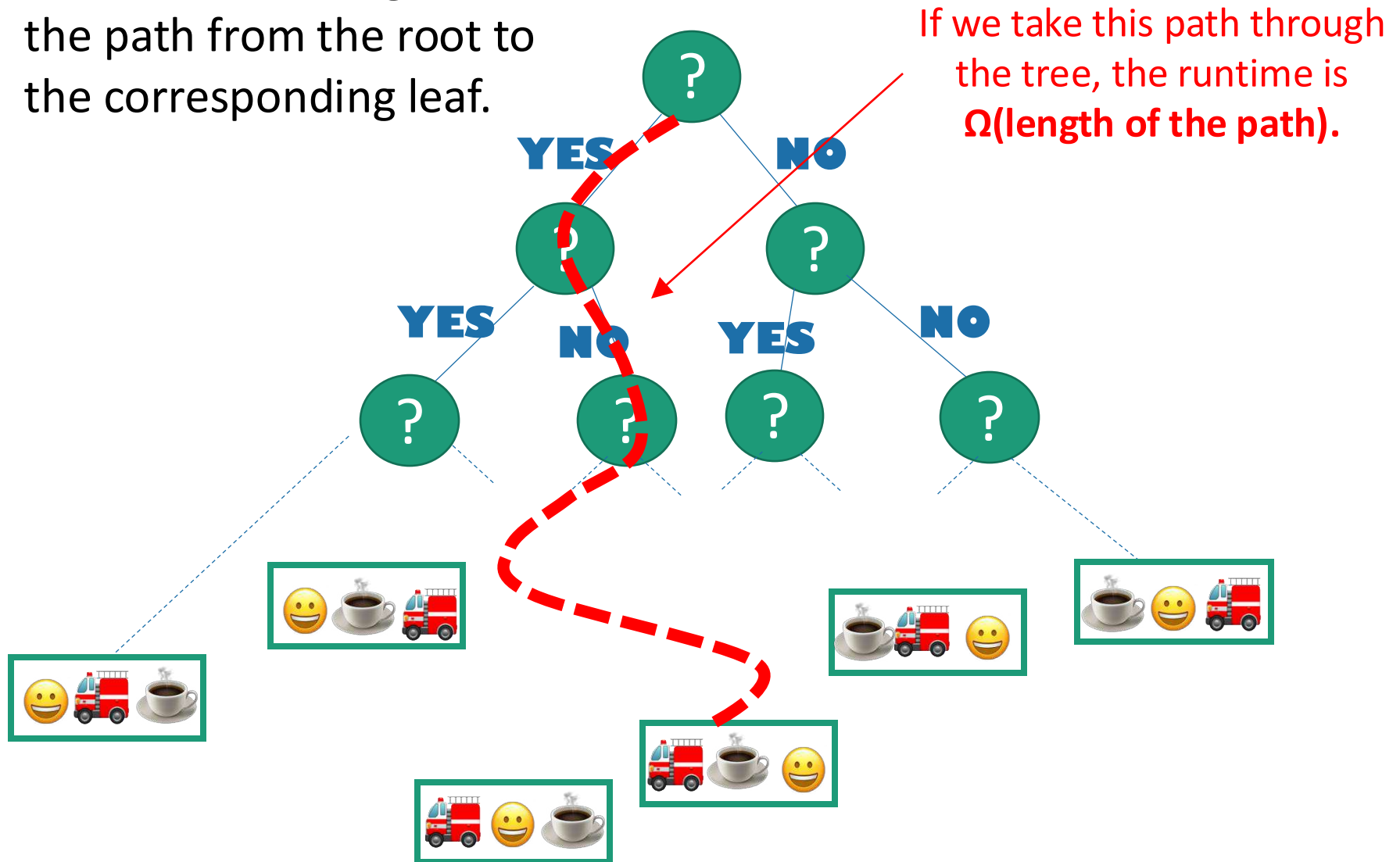
# Any comparison-based sorting algorithm gives a decision tree

- **Internal nodes:** comparisons that the algorithm makes
- **Leaf nodes:** outputs of the algorithm, aka possible sorted orderings



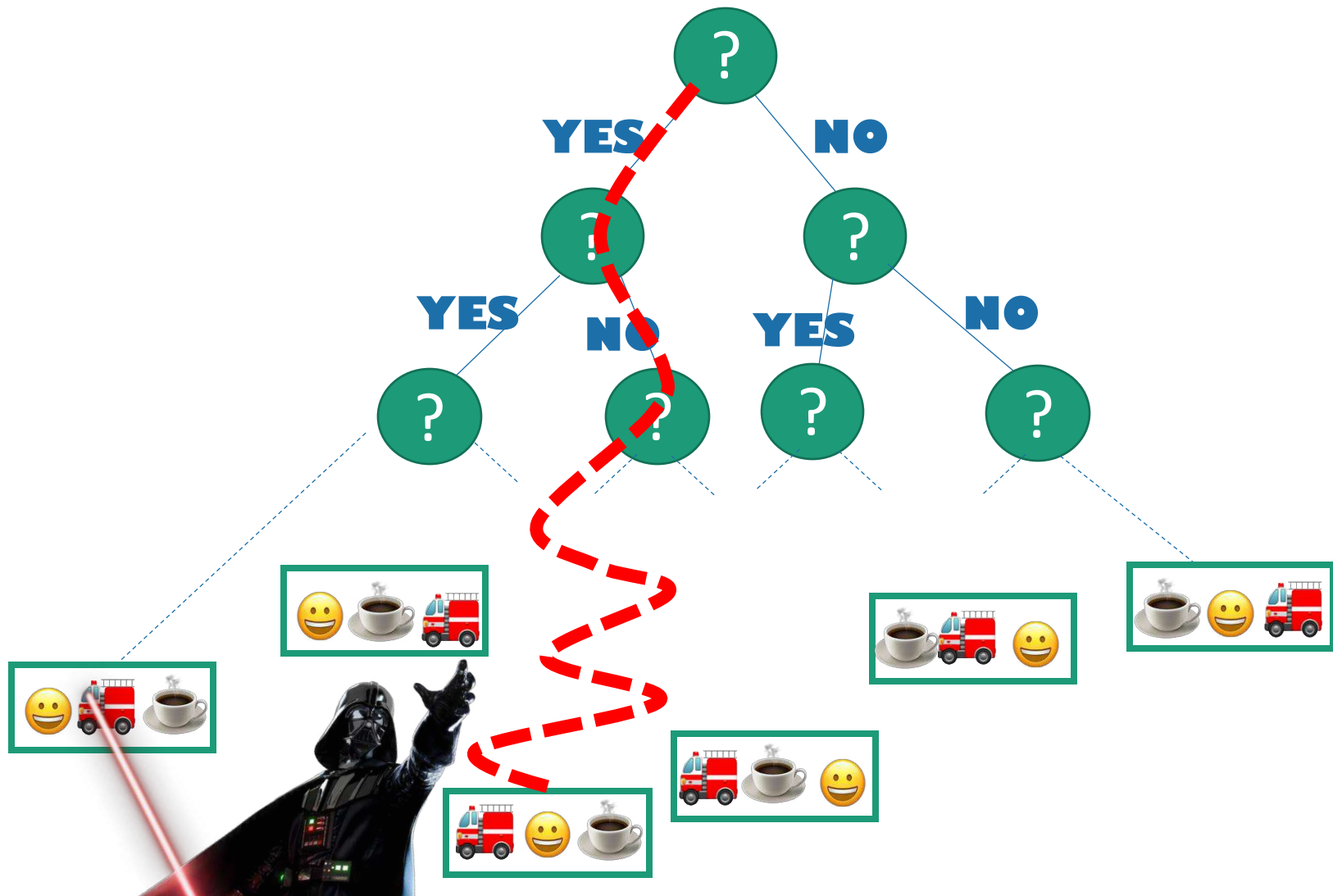
Q: What's the runtime on a particular input?

A: At least the length of the path from the root to the corresponding leaf.



Q: What's the worst-case runtime?

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



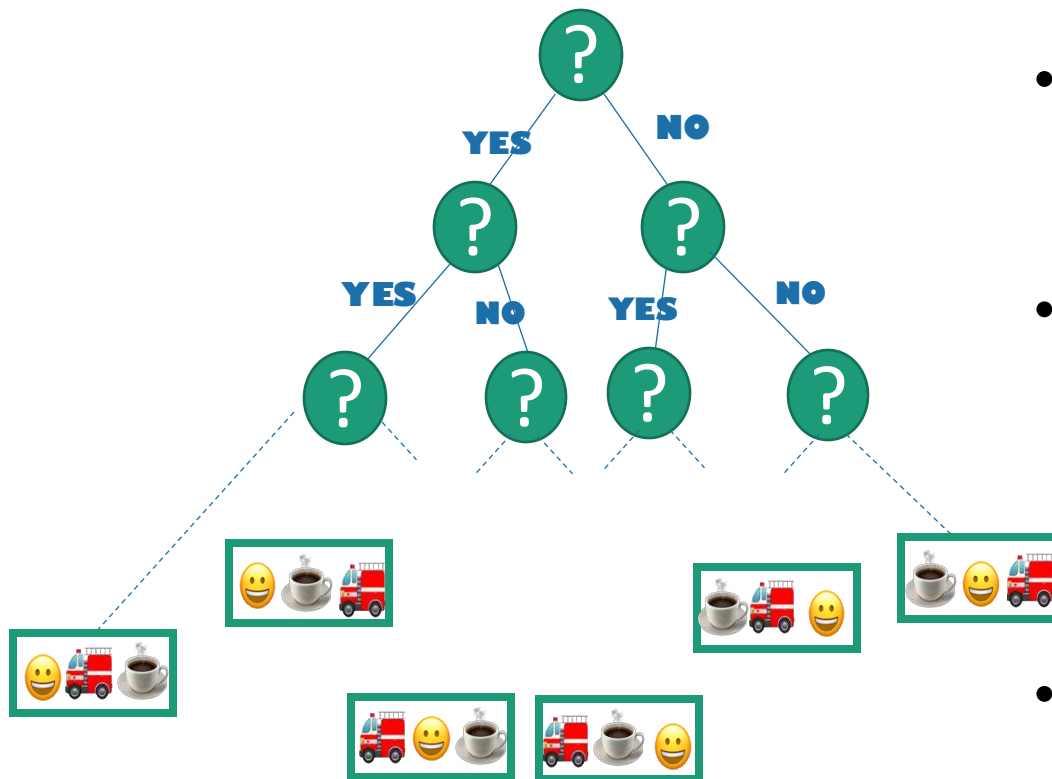


# How long is the longest path?



being sloppy about  
floors and ceilings!

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



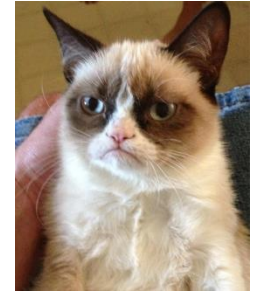
- This is a binary tree with at least  $n!$  leaves.
- The shallowest tree with  $n!$  leaves is the completely balanced one, which has depth  $\log(n!)$ .
- So in all such trees, the longest path is at least  $\log(n!)$ .

- $n!$  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  $\Omega(n \log(n))$ .

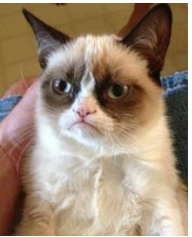
# Aside:

## What about randomized algorithms?

- For example, QuickSort?

- Theorem:

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



- Proof:

- (same ideas as deterministic case)
- (you are not responsible for this proof in this class)

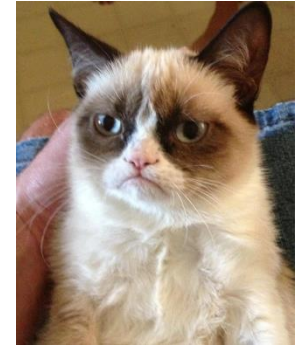
# \end{Aside}

Try to prove this  
yourself!



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.

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 silly.

## Can we do better?

- Is there be another model of computation that's **less silly** than the StickSort model, in which we can **sort faster** than  $n\log(n)$ ?

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

- How long does it take to sort  $n$  people by their month of birth?
- [discussion]



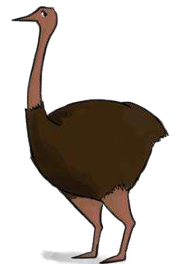
1 (Jan)



1 (Jan)



4 (Apr)



5 (May)

# Another model of computation

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



instead of

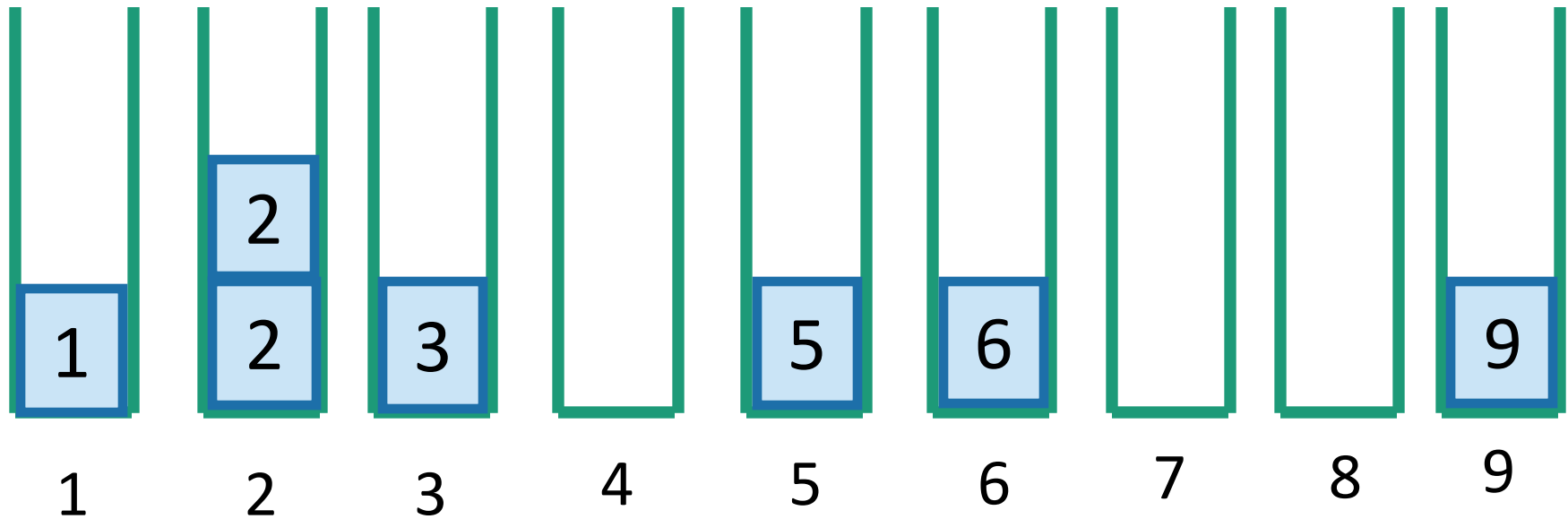


# Why might this help?



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

CountingSort:



Concatenate  
the buckets!

**SORTED!**

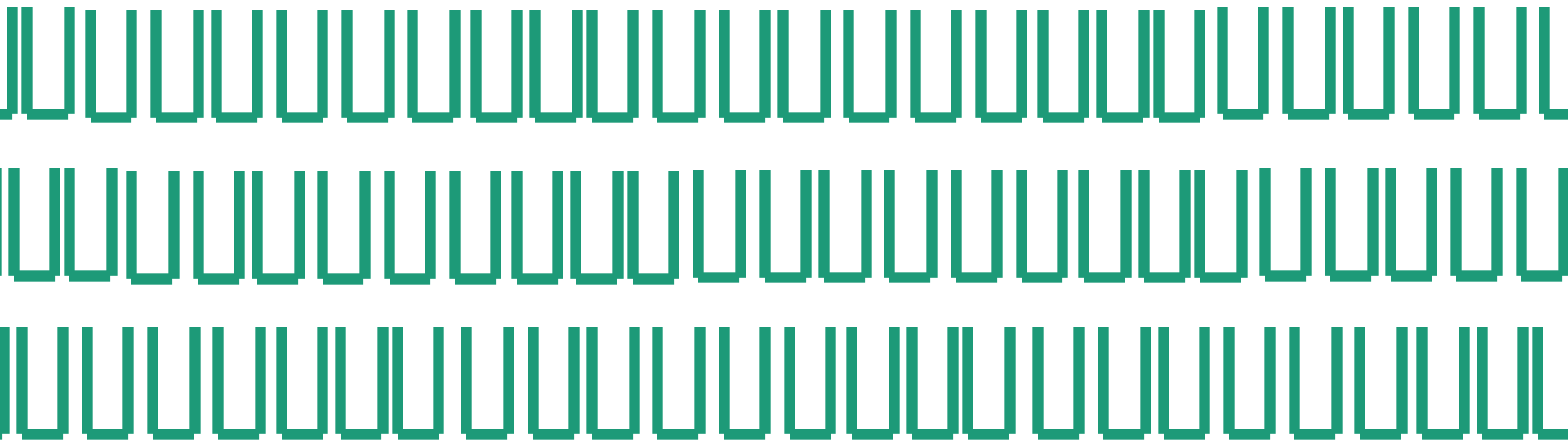
In time  $O(n)$ .

# Assumptions

- Need to be able to know what bucket to put something in.
  - We assume we can evaluate the items directly, not just by comparison
- Need to know what values might show up ahead of time.

2	12345	13	$2^{1000}$	50	100000000	1
---	-------	----	------------	----	-----------	---

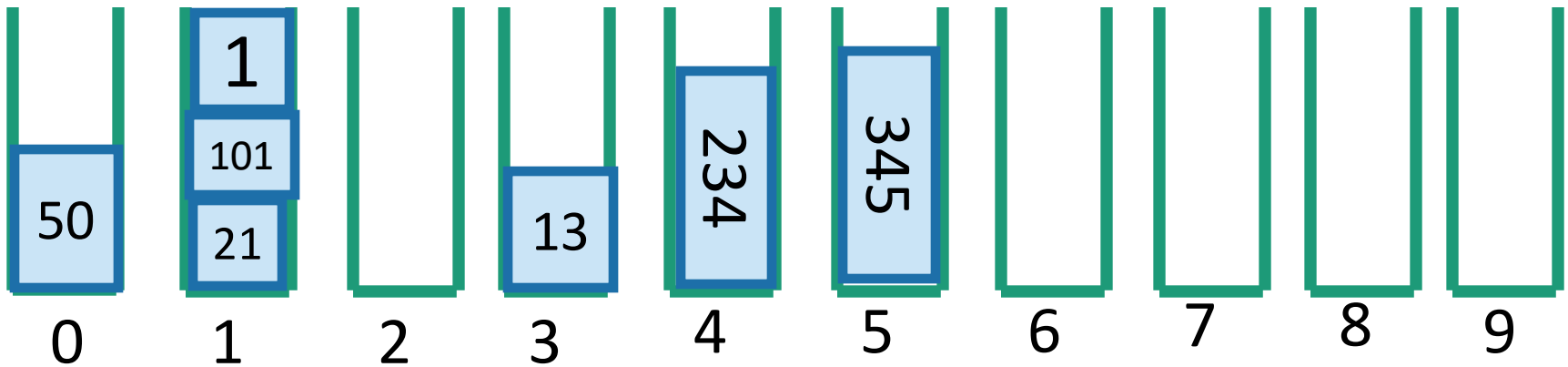
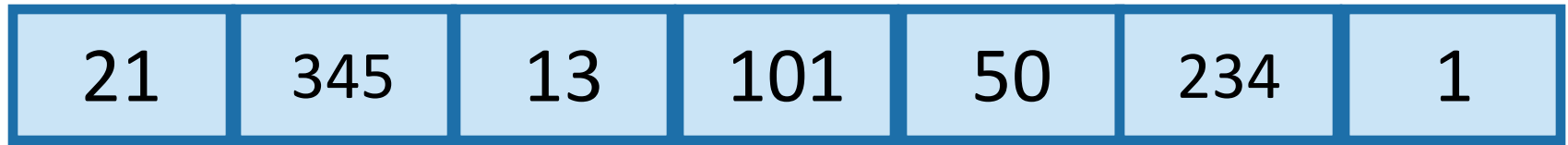
- Need to assume there are not too many such values.



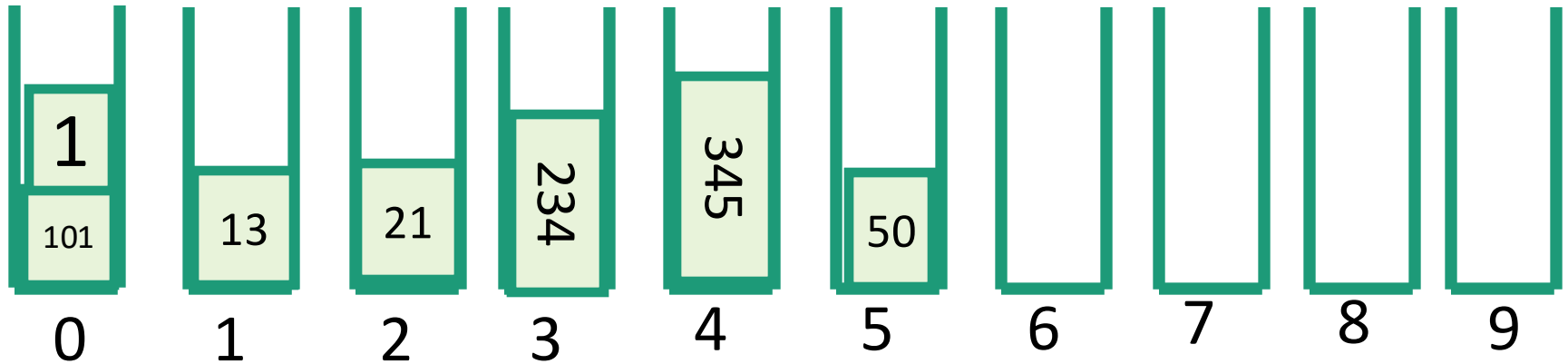
# RadixSort

- For sorting integers up to size  $M$ 
  - or more generally for lexicographically sorting strings
- Can use less space than CountingSort
- Idea: CountingSort on the least-significant digit first, then the next least-significant, and so on.

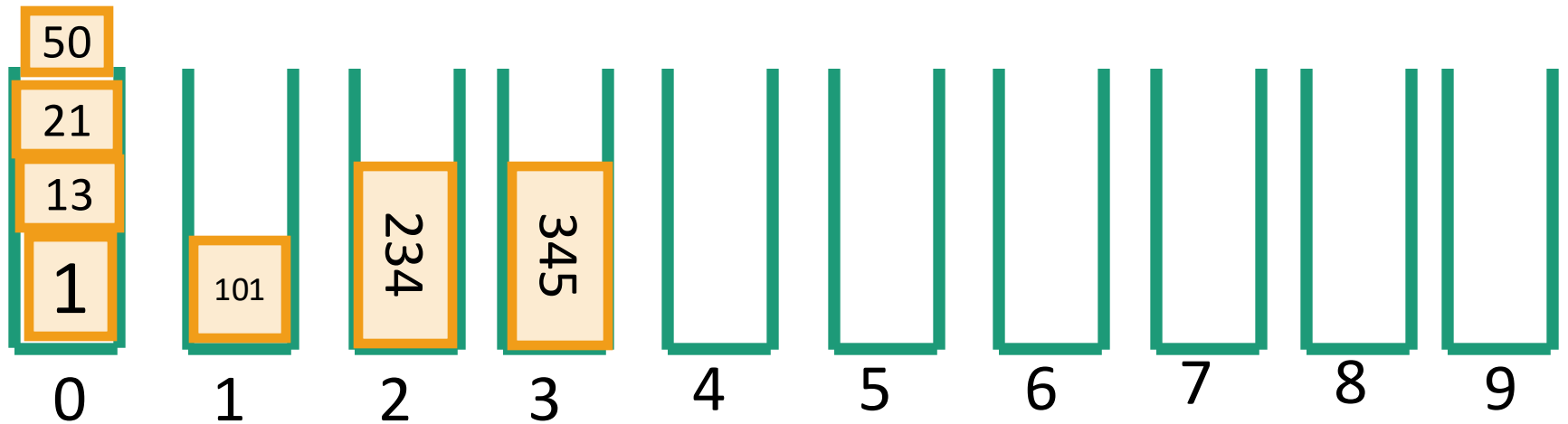
# Step 1: CountingSort on least significant digit



Step 2: CountingSort on the 2<sup>nd</sup> least sig. digit



# Step 3: CountingSort on the 3<sup>rd</sup> least sig. digit



It worked!!



# Why does this work?

Original array:

21	345	13	101	50	234	1
----	-----	----	-----	----	-----	---

Next array is sorted by the first digit.

50	21	101	1	13	234	345
----	----	-----	---	----	-----	-----

Next array is sorted by the first two digits.

101	01	13	21	234	345	50
-----	----	----	----	-----	-----	----

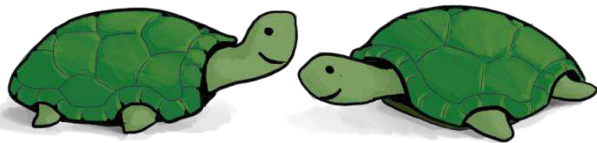
Next array is sorted by all three digits.

001	013	021	050	101	234	345
-----	-----	-----	-----	-----	-----	-----

Sorted array

# To prove this is correct...

- What is the inductive hypothesis?
- (And if you have time, try to think about how the other steps would go...)



Think-Pair-Share Terrapins

Original array:

21	345	13	101	50	234	1
----	-----	----	-----	----	-----	---

Next array is sorted by the first digit.

50	21	101	1	13	234	345
----	----	-----	---	----	-----	-----

Next array is sorted by the first two digits.

101	01	13	21	234	345	50
-----	----	----	----	-----	-----	----

Next array is sorted by all three digits.

001	013	021	050	101	234	345
-----	-----	-----	-----	-----	-----	-----

Sorted array

# RadixSort is correct

- 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, so the IH holds for  $k=0$ .
- Inductive step:
  - TO DO
- Conclusion:
  - The inductive hypothesis holds for all  $k$ , so after the last iteration, the array is sorted by all the digits. Hence, it's sorted!

# Inductive step

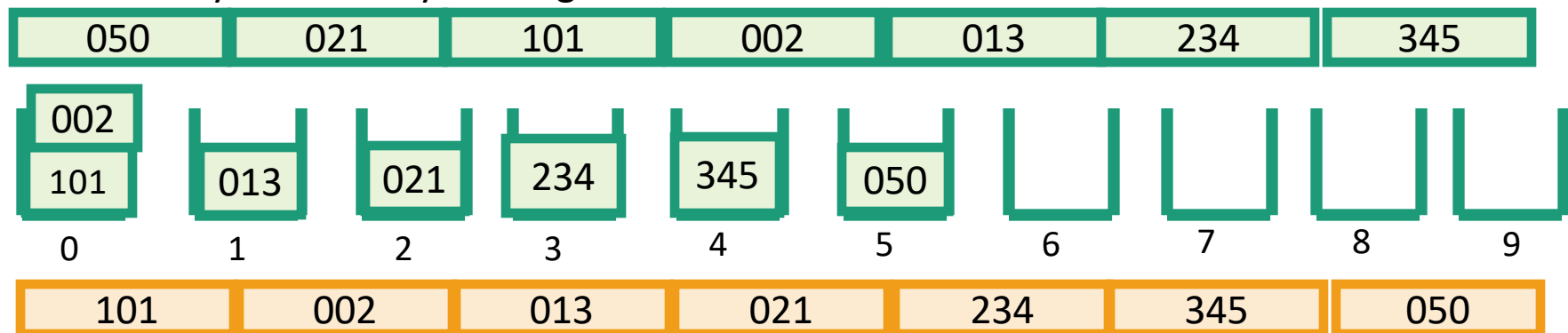
Inductive hypothesis:

After the  $k$ 'th iteration, the array is sorted by the first  $k$  least-significant digits.

- Need to show: if IH holds for  $k=i-1$ , then it holds for  $k=i$ .
  - Suppose that after the  $i-1$ 'st iteration, the array is sorted by the first  $i-1$  least-significant digits.
  - Need to show that after the  $i$ 'th iteration, the array is sorted by the first  $i$  least-significant digits.

EXAMPLE:  $i=2$

IH: this array is sorted by first digit.



Want to show: this array is sorted by 1<sup>st</sup> and 2<sup>nd</sup> digits.

# Proof sketch...

proof on next (skipped) slide

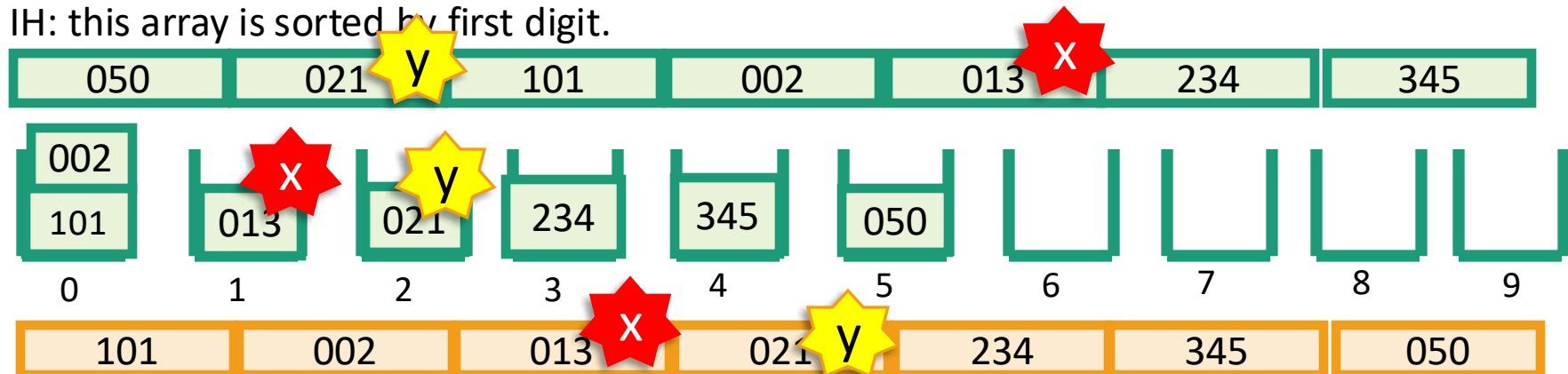
Want to show: after the  $i$ 'th iteration, the array is sorted by the first  $i$  least-significant digits.

- Let  $x=[x_dx_{d-1}...x_2x_1]$  and  $y=[y_dy_{d-1}...y_2y_1]$  be any  $x,y$ .
- Suppose  $[x_ix_{i-1}...x_2x_1] < [y_iy_{i-1}...y_2y_1]$ .
- Want to show that  $x$  appears before  $y$  at end of  $i$ 'th iteration.
- CASE 1:  $x_i < y_i$ 
  - $x$  is in an earlier bucket than  $y$ .



EXAMPLE:  $i=2$

IH: this array is sorted by first digit.



Want to show: this array is sorted by 1<sup>st</sup> and 2<sup>nd</sup> digits.

# Proof sketch...

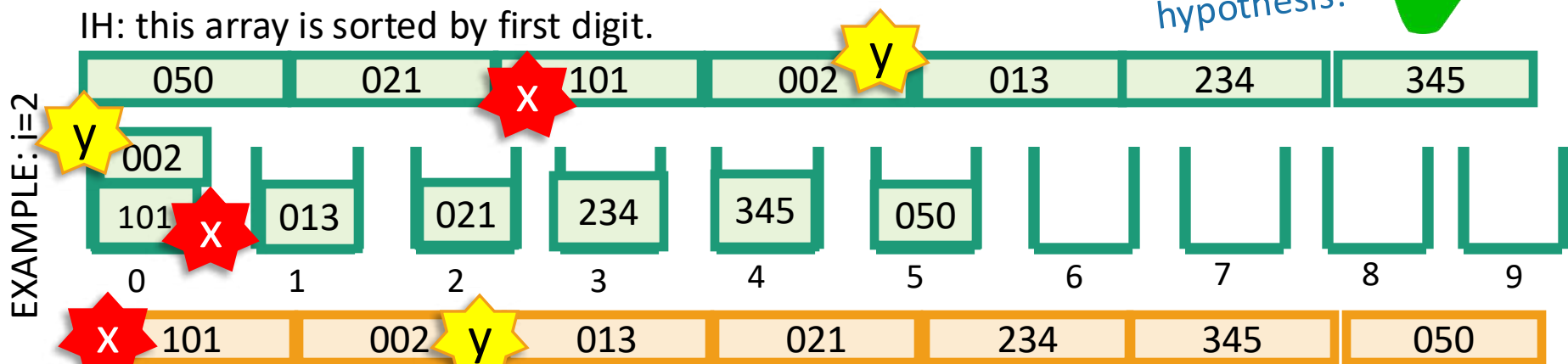
proof on next (skipped) slide

Want to show: after the  $i$ 'th iteration, the array is sorted by the first  $i$  least-significant digits.

- Let  $x=[x_dx_{d-1}...x_2x_1]$  and  $y=[y_dy_{d-1}...y_2y_1]$  be any  $x,y$ .
- Suppose  $[x_ix_{i-1}...x_2x_1] < [y_iy_{i-1}...y_2y_1]$ .
- Want to show that  $x$  appears before  $y$  at end of  $i$ 'th iteration.
- CASE 1:  $x_i < y_i$ 
  - $x$  is in an earlier bucket than  $y$ .
- CASE 2:  $x_i = y_i$ 
  - $x$  and  $y$  in same bucket, but  $x$  was put in the bucket first.



By the inductive hypothesis!



Want to show: this array is sorted by 1<sup>st</sup> and 2<sup>nd</sup> digits.

Want to show: after the  $i$ 'th iteration, the array is sorted by the first  $i$  least-significant digits.

SLIDE SKIPPED  
IN CLASS. Here  
for reference.

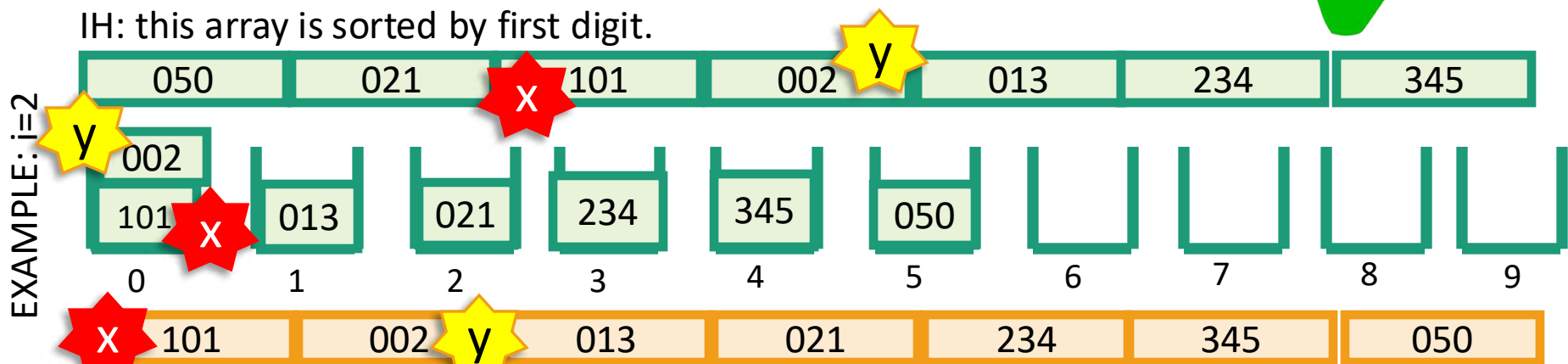
- Let  $x=[x_dx_{d-1}\dots x_2x_1]$  and  $y=[y_dy_{d-1}\dots y_2y_1]$  be any  $x,y$ .
- Suppose  $[x_ix_{i-1}\dots x_2x_1] < [y_iy_{i-1}\dots y_2y_1]$ .
- Want to show that  $x$  appears before  $y$  at end of  $i$ 'th iteration.
- CASE 1:  $x_i < y_i$ .
  - $x$  appears in an earlier bucket than  $y$ , so  $x$  appears before  $y$  after the  $i$ 'th iteration.
- CASE 2:  $x_i = y_i$ .
  - $x$  and  $y$  end up in the same bucket.
  - In this case,  $[x_{i-1}\dots x_2x_1] < [y_{i-1}\dots y_2y_1]$ , so by the inductive hypothesis,  $x$  appeared before  $y$  after  $i-1$ 'st iteration.
  - Then  $x$  was placed into the bucket before  $y$  was, so it also comes out of the bucket before  $y$  does.
    - Recall that the buckets are FIFO queues.
  - So  $x$  appears before  $y$  in the  $i$ 'th iteration.

# Proof sketch...

proof on next (skipped) slide

Want to show: after the  $i$ 'th iteration, the array is sorted by the first  $i$  least-significant digits.

- Let  $x=[x_dx_{d-1}...x_2x_1]$  and  $y=[y_dy_{d-1}...y_2y_1]$  be any  $x,y$ .
- Suppose  $[x_ix_{i-1}...x_2x_1] < [y_iy_{i-1}...y_2y_1]$ .
- Want to show that  $x$  appears before  $y$  at end of  $i$ 'th iteration.
- CASE 1:  $x_i < y_i$ 
  - $x$  is in an earlier bucket than  $y$ .
- CASE 2:  $x_i = y_i$ 
  - $x$  and  $y$  in same bucket, but  $x$  was put in the bucket first.



Want to show: this array is sorted by 1<sup>st</sup> and 2<sup>nd</sup> digits.



# Inductive step

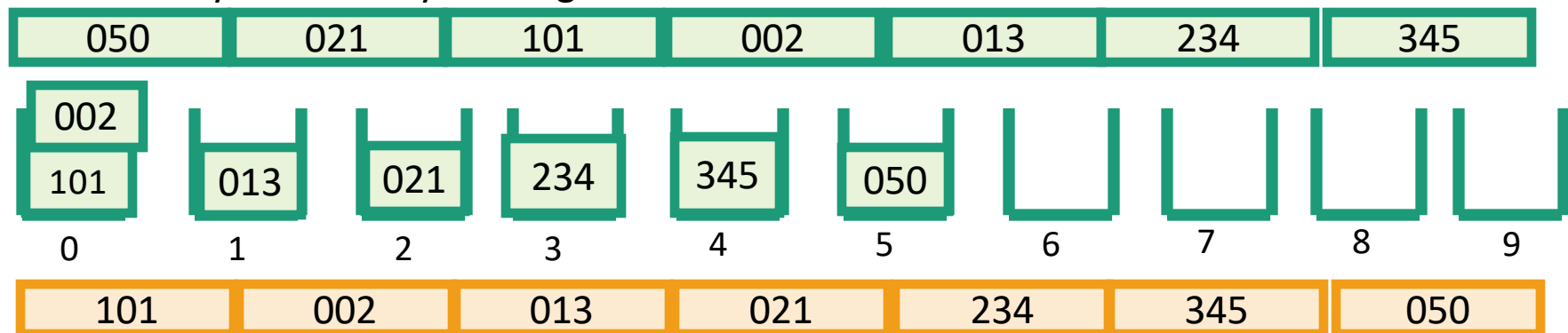
Inductive hypothesis:

After the  $k$ 'th iteration, the array is sorted by the first  $k$  least-significant digits.

- Need to show: if IH holds for  $k=i-1$ , then it holds for  $k=i$ .
  - Suppose that after the  $i-1$ 'st iteration, the array is sorted by the first  $i-1$  least-significant digits.
  - Need to show that after the  $i$ 'th iteration, the array is sorted by the first  $i$  least-significant digits.




IH: this array is sorted by first digit.



Want to show: this array is sorted by 1<sup>st</sup> and 2<sup>nd</sup> digits.

# RadixSort is correct

- 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, so the IH holds for  $k=0$ .
- Inductive step:
  - TO DO 
- Conclusion:
  - The inductive hypothesis holds for all  $k$ , so after the last iteration, the array is sorted by all the digits. Hence, it's sorted!

# What is the running time?

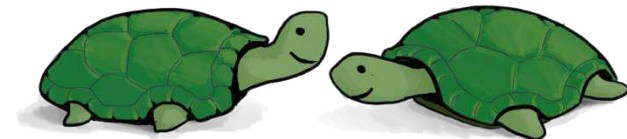
for RadixSorting  
numbers base-10.

- Suppose we are sorting  $n$   $d$ -digit numbers (in base 10).

e.g.,  $n=7$ ,  $d=3$ :

021	345	013	101	050	234	001
-----	-----	-----	-----	-----	-----	-----

1. How many iterations are there?
2. How long does each iteration take?
3. What is the total running time?



Think-Pair-Share Terrapins

# What is the running time?

for RadixSorting  
numbers base-10.

- Suppose we are sorting  $n$   $d$ -digit numbers (in base 10).

e.g.,  $n=7$ ,  $d=3$ :

021	345	013	101	050	234	001
-----	-----	-----	-----	-----	-----	-----

1. How many iterations are there?

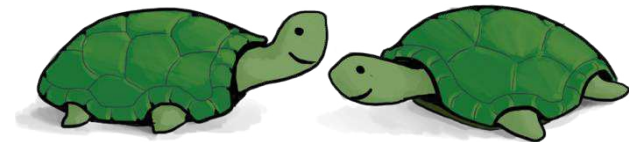
- $d$  iterations

2. How long does each iteration take?

- Time to initialize 10 buckets, plus time to put  $n$  numbers in 10 buckets.  $O(n)$ .

3. What is the total running time?

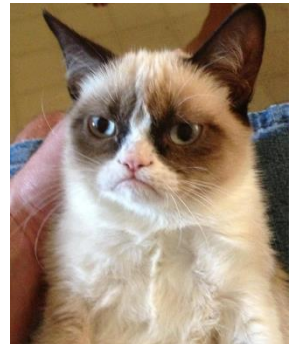
- $O(nd)$



Think-Pair-Share Terrapins

# This doesn't seem so great

- To sort  $n$  integers, each of which is in  $\{1, 2, \dots, n\}$ ...
- $d = \lfloor \log_{10}(n) \rfloor + 1$ 
  - For example:
    - $n = 1234$
    - $\lfloor \log_{10}(1234) \rfloor + 1 = 4$
    - More explanation on next (skipped) slide.
- Time =  $O(nd) = O(n \log(n))$ .
  - Same as MergeSort!



# Aside: why $d = \lfloor \log_{10}(n) \rfloor + 1$ ?

Slide  
skipped  
in class

- When we write a number  $x = [x_d x_{d-1} \dots x_1]$  base 10, that means:  
$$x = x_1 + x_2 \cdot 10 + \dots + x_{d-1} \cdot 10^{d-2} + x_d \cdot 10^{d-1}$$

where  $x_i \in \{0, 1, \dots, 9\}$

- Suppose that  $x_d \neq 0$ . Then we have

- $x \geq x_d \cdot 10^{d-1}$

Since  $x$  is bigger than just the last term in that sum!

- $\log_{10}(x) + 1 - \log_{10}(x_d) \geq d$

(take logs of both sides and rearrange)

- $\log_{10}(x) + 1 \geq d$

$\log_{10}(x_d) \geq 0$  since  $x_d \geq 1$

- $\lfloor \log_{10}(n) \rfloor + 1 \geq d$

Since  $d$  is an integer

- On the other hand, we also have

- $x < (x_d + 1) \cdot 10^{d-1}$

Since if  $x \geq (x_d + 1) \cdot 10^{d-1}$  then the  $d$ 'th digit would have been  $x_d + 1$  instead of  $x_d$

- $\log_{10}(x) + 1 - \log_{10}(x_d + 1) < d$

(take logs of both sides and rearrange)

- $\log_{10}(x) < d$

$\log_{10}(x_d + 1) \leq 1$  since  $x_d < 10$

- $\lfloor \log_{10}(n) \rfloor + 1 \leq d$

Since  $d$  is an integer

# Can we do better?

- RadixSort base 10 doesn't seem to be such a good idea...
- But what if we change the base? (Let's say base  $r$ )
- There's a trade-off:
  - Bigger  $r$  means more buckets
  - Bigger  $r$  means fewer digits



# Example: base 100

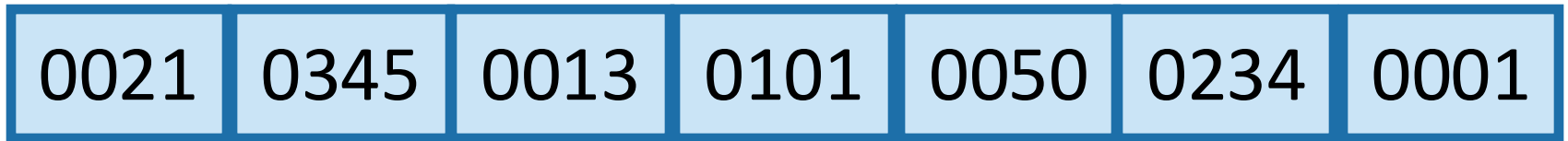
Original array:

21	345	13	101	50	234	1
----	-----	----	-----	----	-----	---

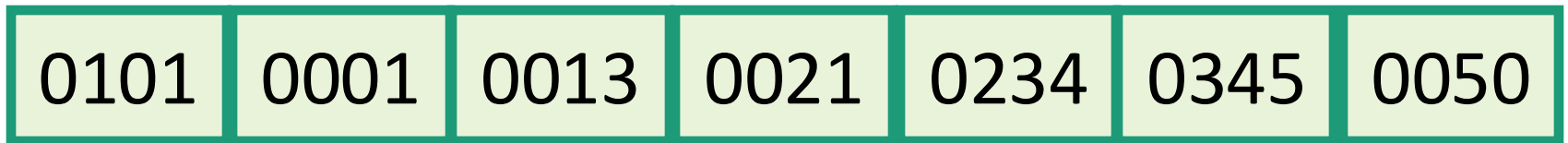
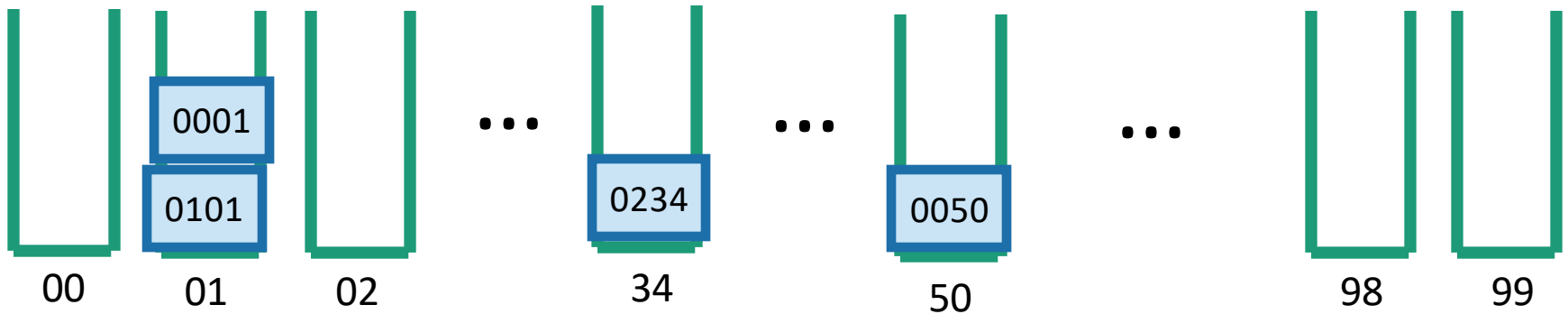


# Example: base 100

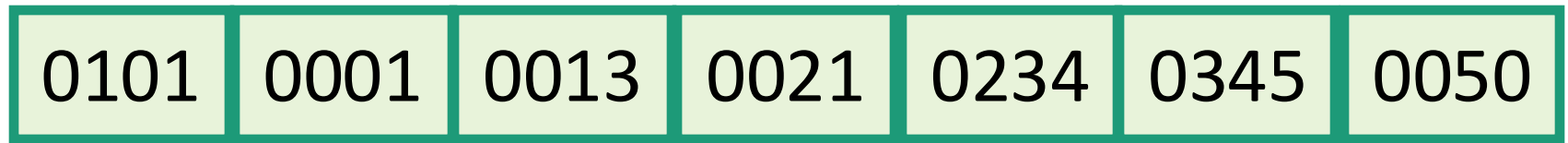
Original array:



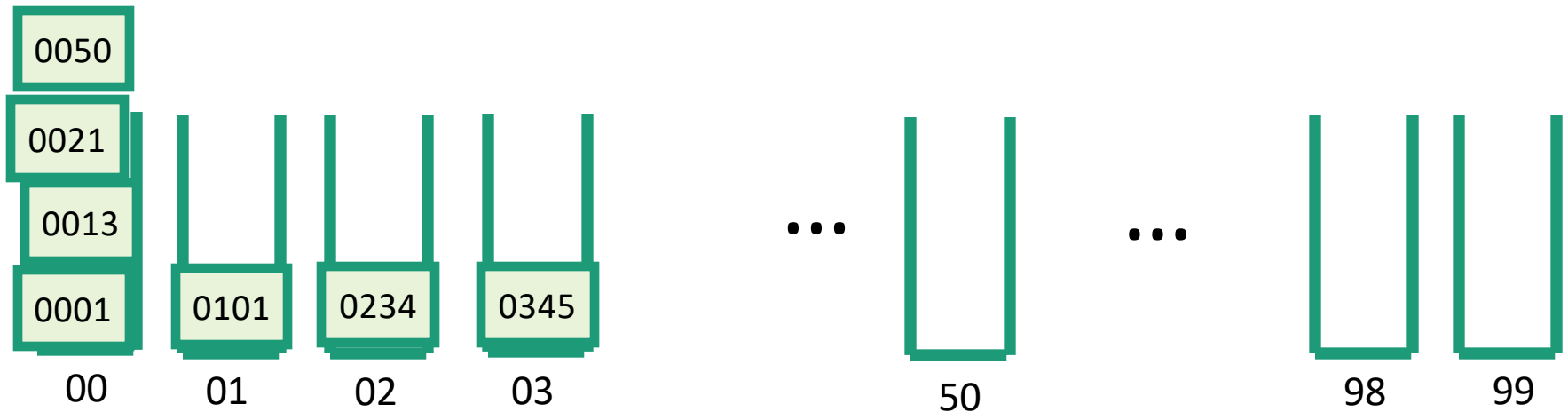
100 buckets:



# Example: base 100



100 buckets:



Sorted!

# Example: base 100

Original array

0021	0345	0013	0101	0050	0234	0001
------	------	------	------	------	------	------

0101	0001	0013	0021	0234	0345	0050
------	------	------	------	------	------	------

0001	0013	0021	0050	0101	0234	0345
------	------	------	------	------	------	------

Sorted array

Base 100:

- $d=2$ , so only 2 iterations.
- 100 buckets

vs.

Base 10:

- $d=3$ , so 3 iterations.
- 10 buckets

Bigger base means more buckets but fewer iterations.

# General running time of RadixSort

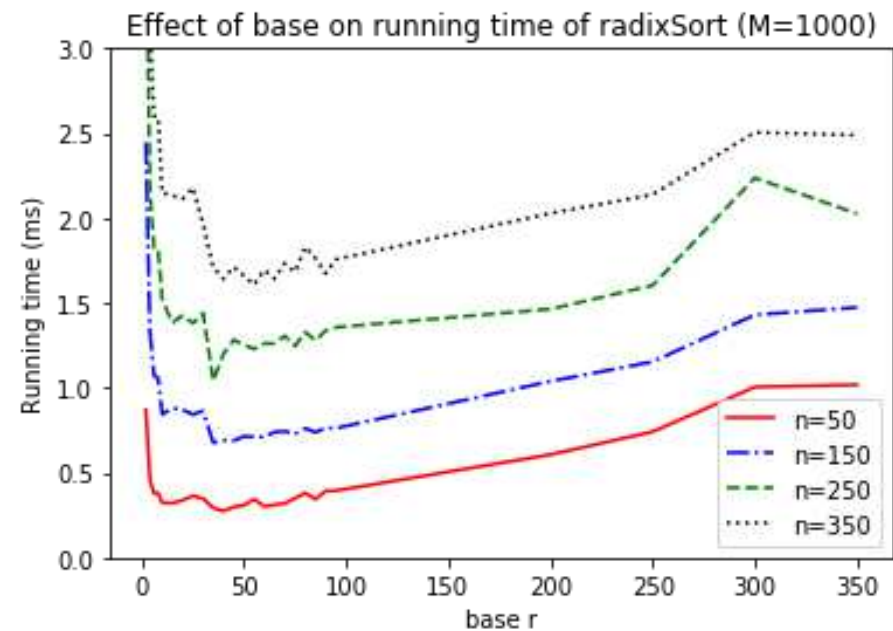
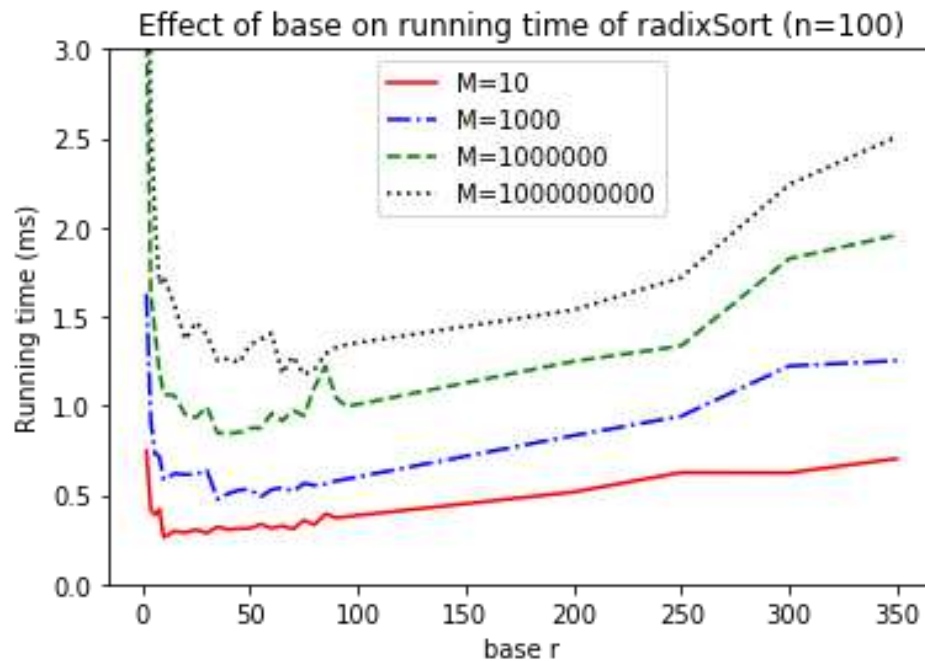
- Say we want to sort:
  - $n$  integers,
  - maximum size  $M$ ,
  - in base  $r$ .
- Number of iterations of RadixSort:
  - Same as number of digits, base  $r$ , of an integer  $x$  of max size  $M$ .
  - That is  $d = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
  - Initialize  $r$  buckets, put  $n$  items into them
  - $O(n + r)$  total time.
- Total time:
  - $O(d \cdot (n + r)) = O((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r))$

Convince yourself that  
this is the right formula  
for  $d$ .



# Trade-offs

- Given  $n$ ,  $M$ , how should we choose  $r$ ?
- Looks like there's some sweet spot:



# A reasonable choice: $r=n$

- Running time:

$$O((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r))$$

Intuition: balance  $n$  and  $r$  here.

- Choose  $n=r$ :

$$O(n \cdot (\lfloor \log_n(M) \rfloor + 1))$$

Choosing  $r = n$  is pretty good. What choice of  $r$  optimizes the asymptotic running time? What if I also care about space?



Ollie the over-achieving ostrich

# Running time of RadixSort with $r=n$

- To sort  $n$  integers of size at most  $M$ , time is

$$O\left(n \cdot (\lfloor \log_n(M) \rfloor + 1)\right)$$

- So the running time (in terms of  $n$ ) depends on how big  $M$  is in terms of  $n$ :
  - If  $M \leq n^c$  for some constant  $c$ , then this is  $O(n)$ .
  - If  $M = 2^n$ , then this is  $O\left(\frac{n^2}{\log(n)}\right)$
- The number of buckets needed is  $r=n$ .

# What have we learned?


You can put any  
constant here  
instead of 100.

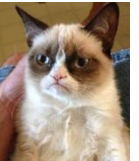
- RadixSort can sort  $n$  integers of size at most  $n^{100}$  in time  $O(n)$ , and needs enough space to store  $O(n)$  integers.
- If your integers have size much much bigger than  $n$  (like  $2^n$ ), maybe you shouldn't use RadixSort.
- It matters how we pick the base.





# Recap

- How difficult sorting 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 it can handle arbitrary comparable objects. 😊
- If we are sorting small integers (or other reasonable data):
  - CountingSort and RadixSort 
  - Both run in time  $O(n)$  😊
  - Might take more space and/or be slower if integers get too big 😞



# Next time

- Binary search trees!
- Balanced binary search trees!

## Before next time

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