Lecture 6

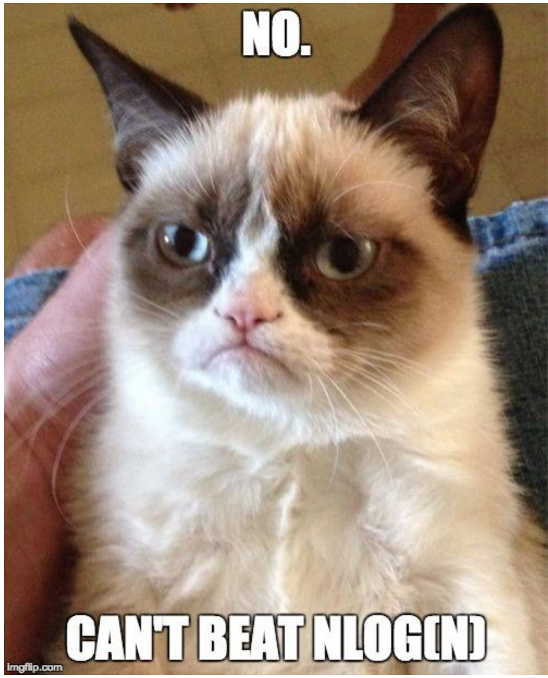
Sorting lower bounds and O(n)-time sorting

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))

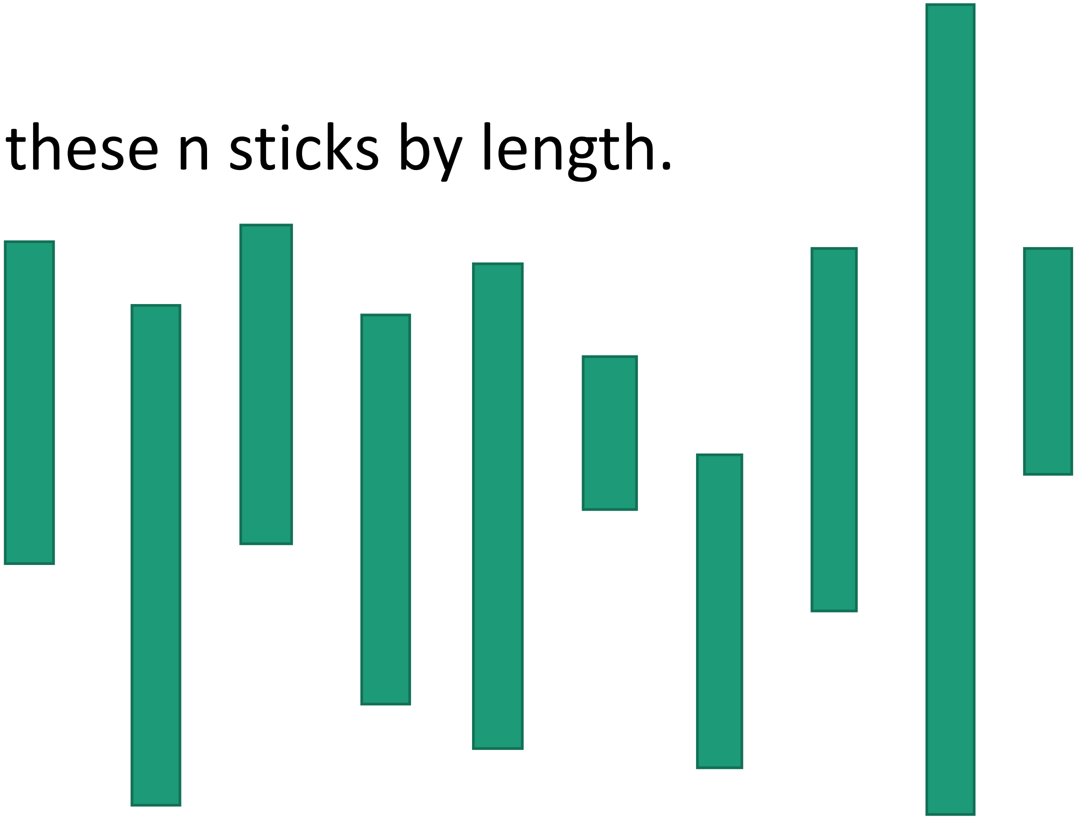
Can we do better?

Depends on who you ask…



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

- Problem: sort these n sticks by length.



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

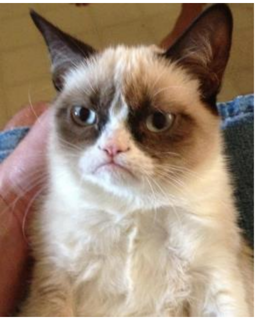
That may have been unsatisfying

- 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?

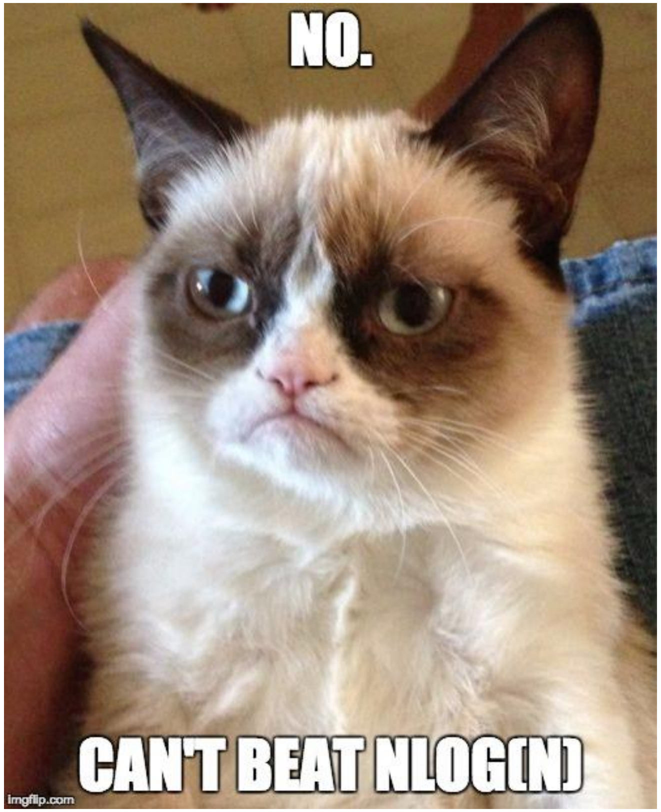
Today: two (more) models



- 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…)
  - 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

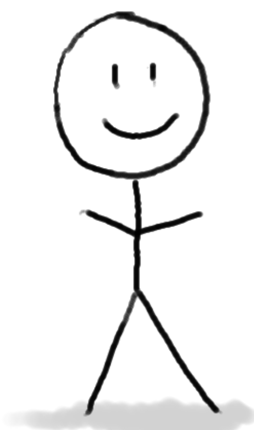
 is shorthand for

"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.

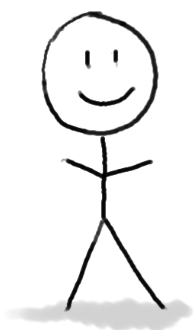
The genie can answer YES/NO questions of the form:
is [this] bigger than [that]?

All the sorting algorithms we have seen work like this.

eg, QuickSort:

Pivot!

7	6	3	5	1	4	2



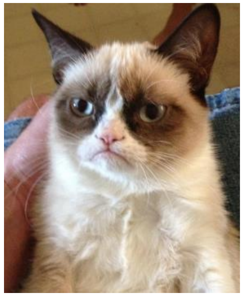
Is 7 bigger than 5 ? **YES**

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!!!

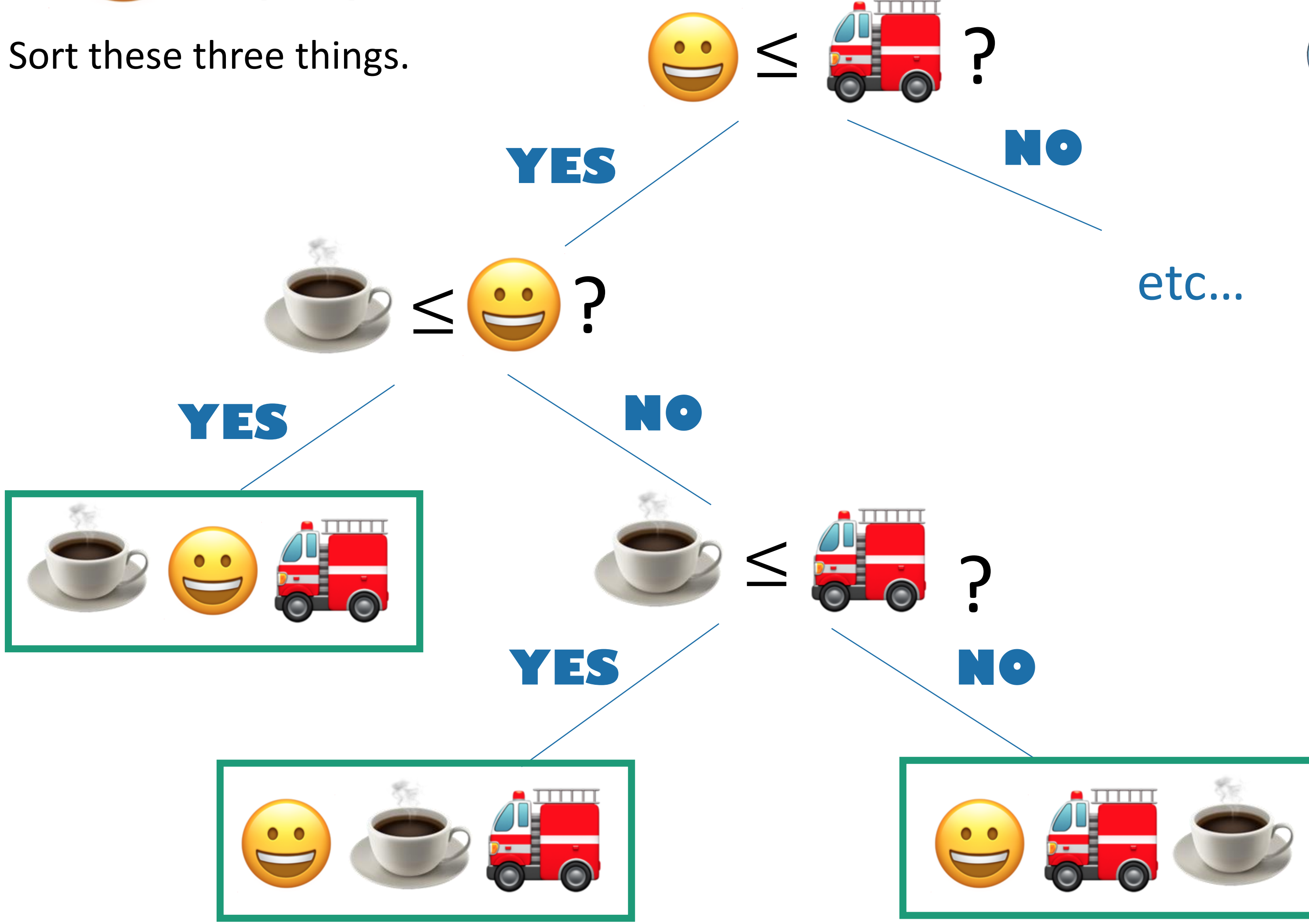
- How might we prove this?

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.

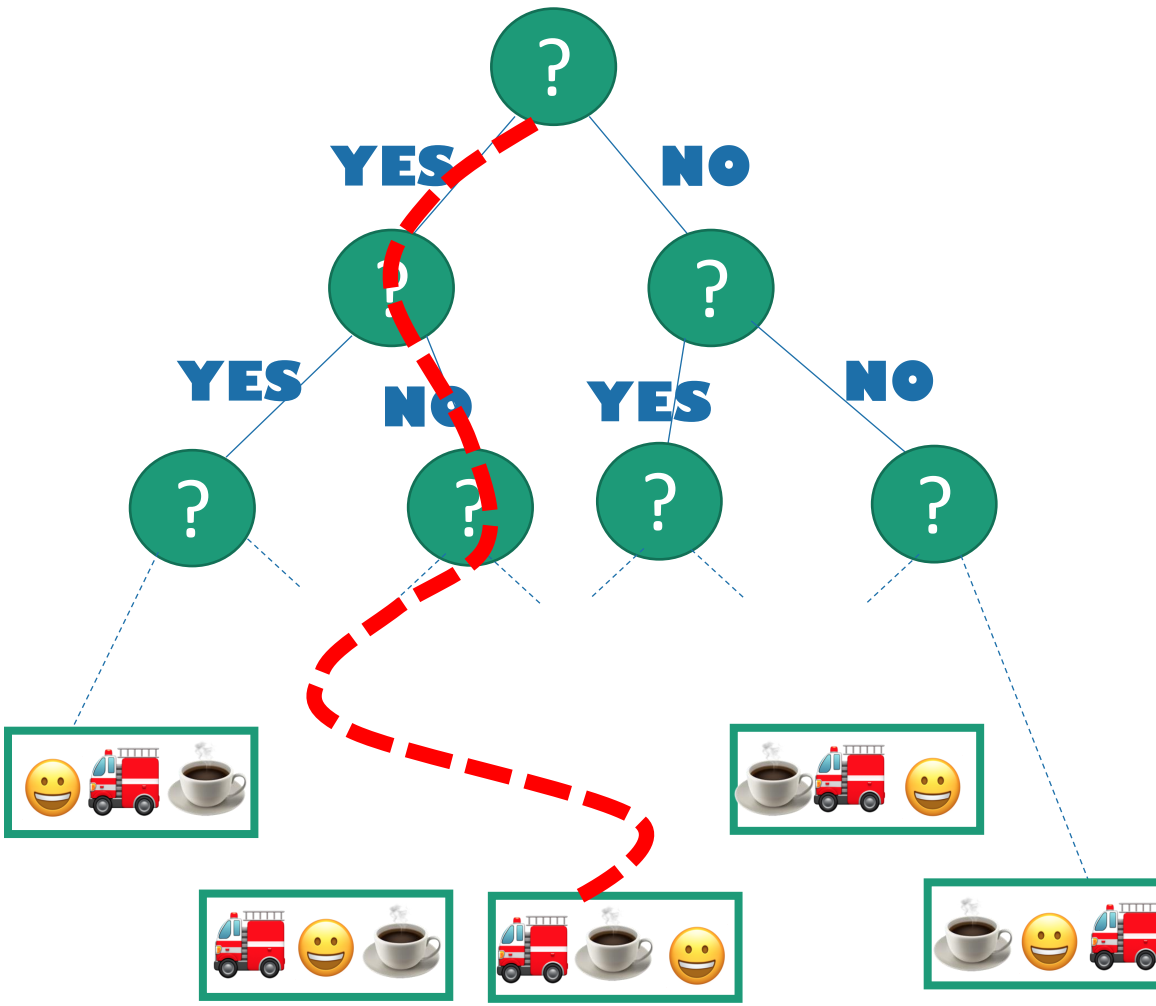
Decision trees

Sort these three things.

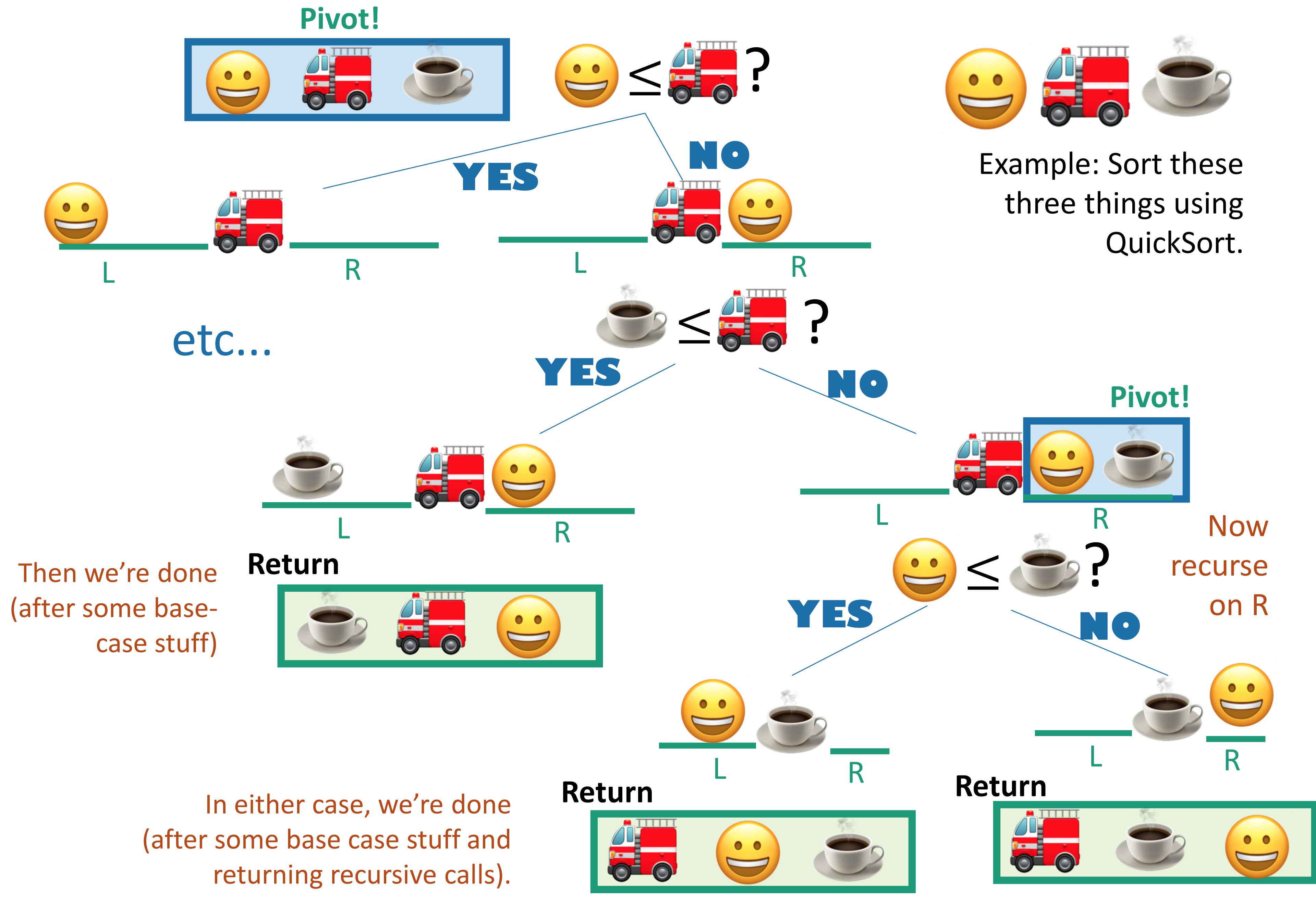


Decision trees

- Internal nodes correspond to yes/no questions.
- Each internal node has two children, one for “yes” and one for “no.”
- Leaf nodes correspond to outputs.
  - In this case, all possible orderings of the items.
- Running an algorithm on a particular input corresponds to a particular path through the tree.



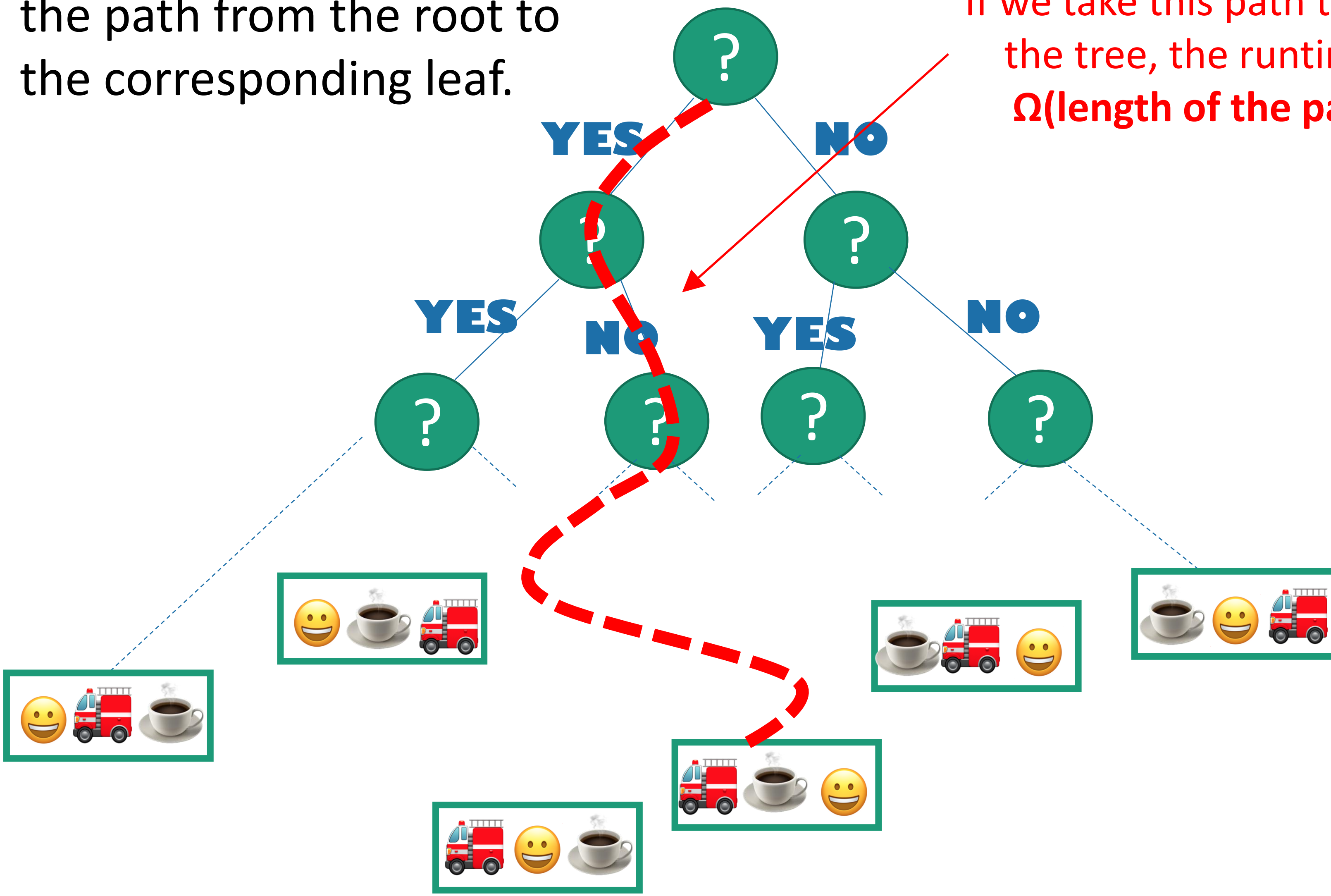
Comparison-based algorithms look like decision trees.



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

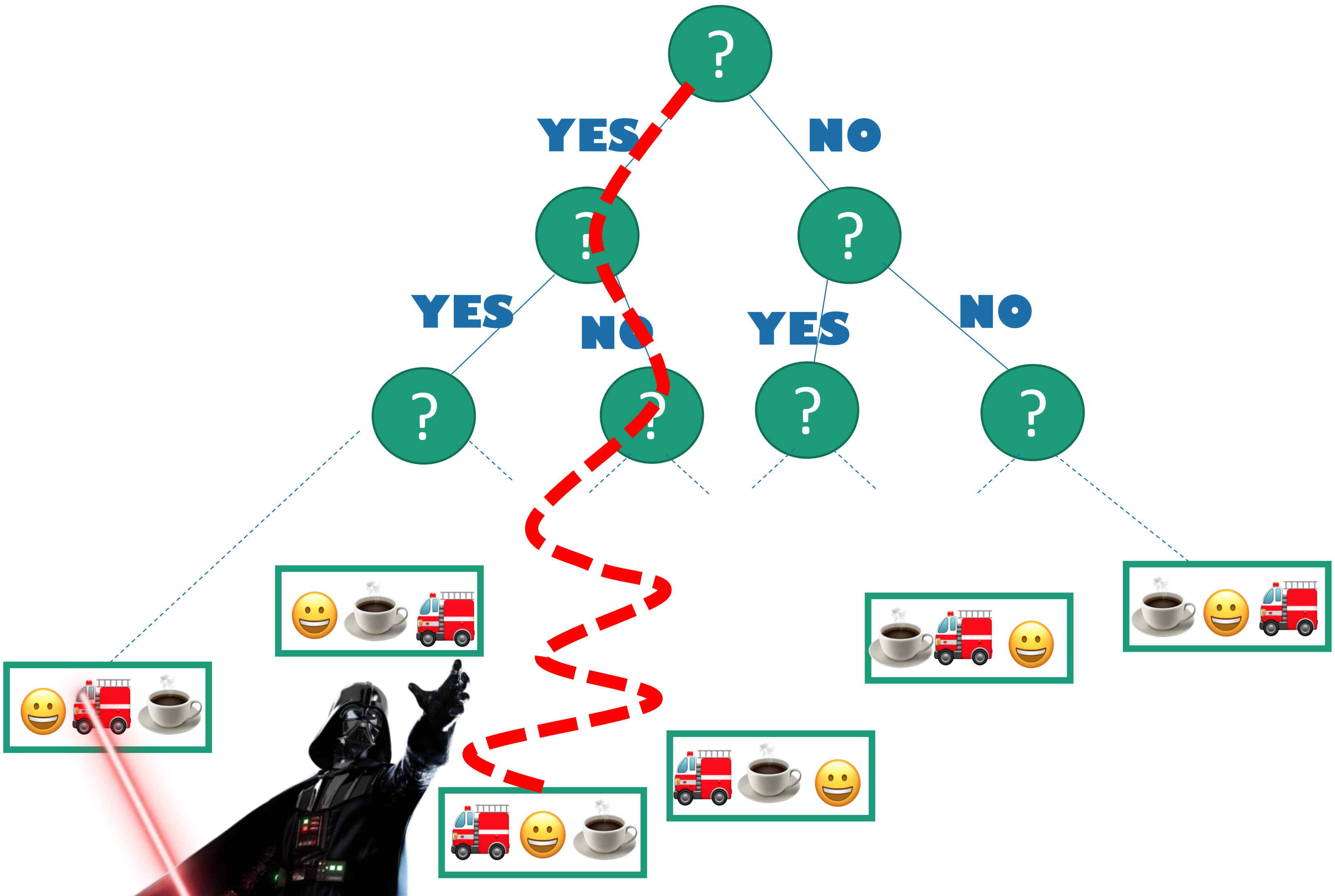
A: At least the length of the path from the root to the corresponding leaf.

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



Q: What's the worst-case runtime?

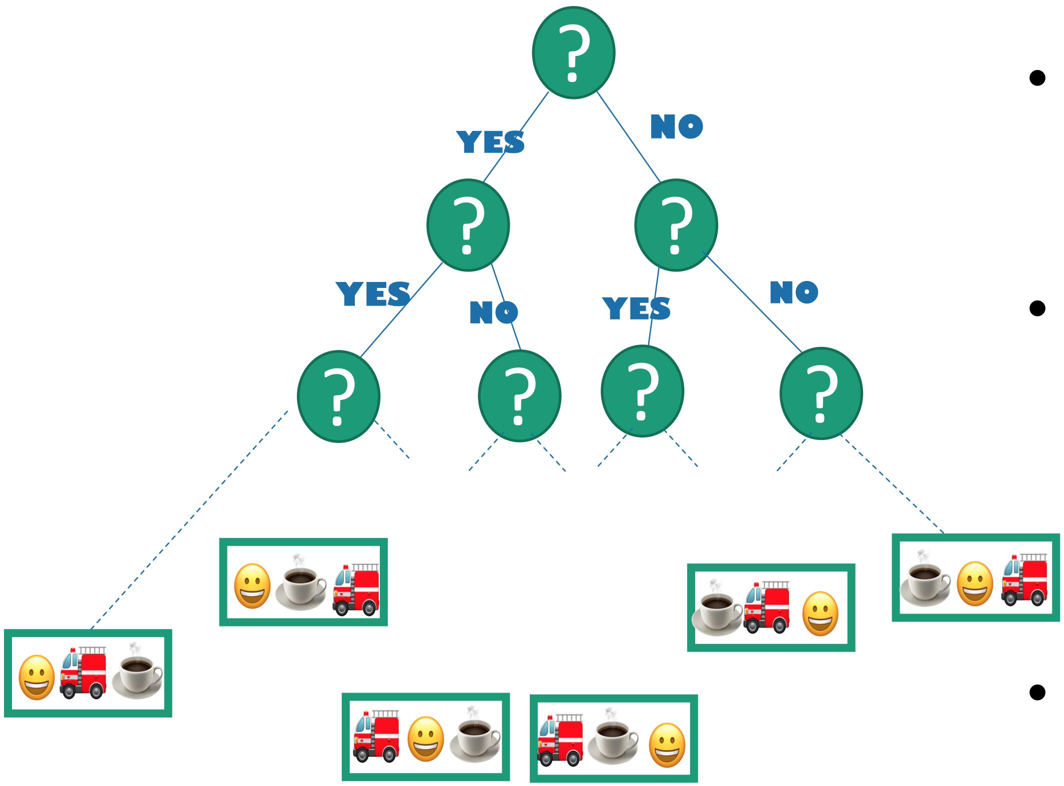
A: At least Ω(length of the longest path).



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) = Ω(n log(n)).

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

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take Ω(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 Ω(n log(n)).
  - So any comparison-based sorting algorithm must have worst-case running time at least Ω(n log(n)).

Aside:
What about randomized algorithms?

- For example, QuickSort?
- Theorem:

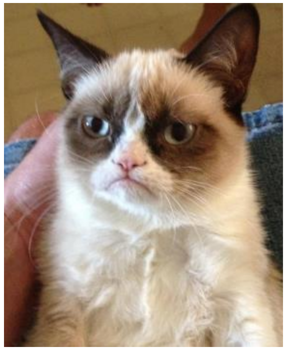
  - Any randomized comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps in expectation.
- Proof:
  - (same ideas as deterministic case)
  - (you are not responsible for this proof in this class)

Try to prove this yourself!

Ollie the over-achieving ostrich

\end{Aside}

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 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.

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

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 nlog(n)?

Beyond comparison-based sorting algorithms

Another model of computation

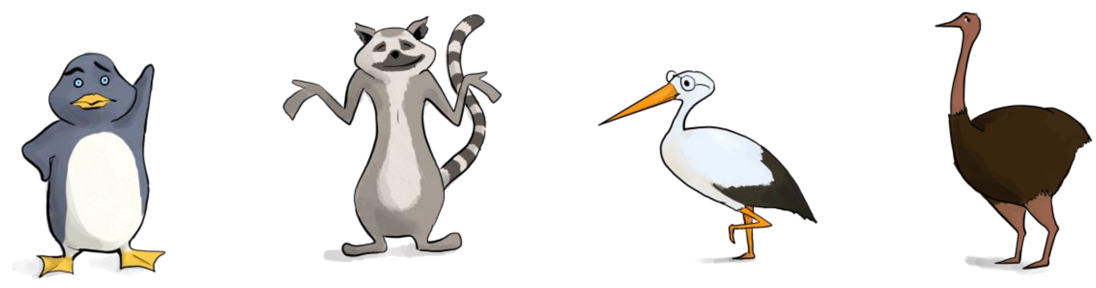
- The items you are sorting have meaningful values.

9	6	3	5	2	1	2

instead of

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.

9	6	3	5	2	1	2

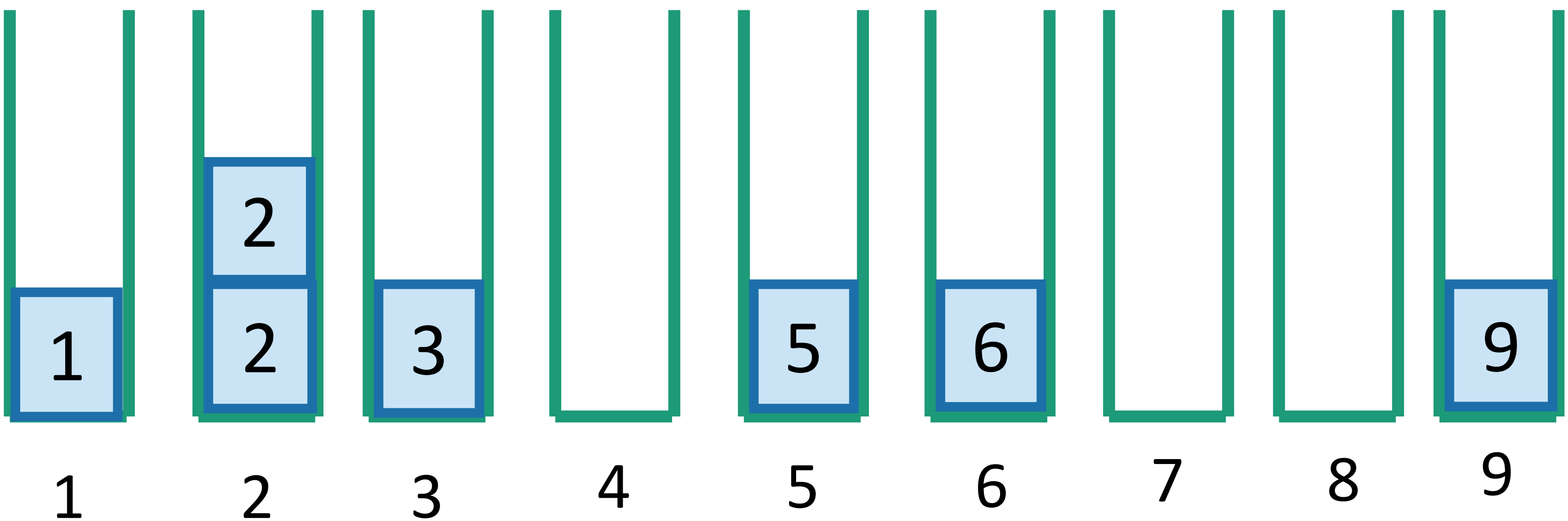
instead of

Why might this help?

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

CountingSort:

9	6	3	5	2	1	2



Concatenate the buckets!

SORTED!

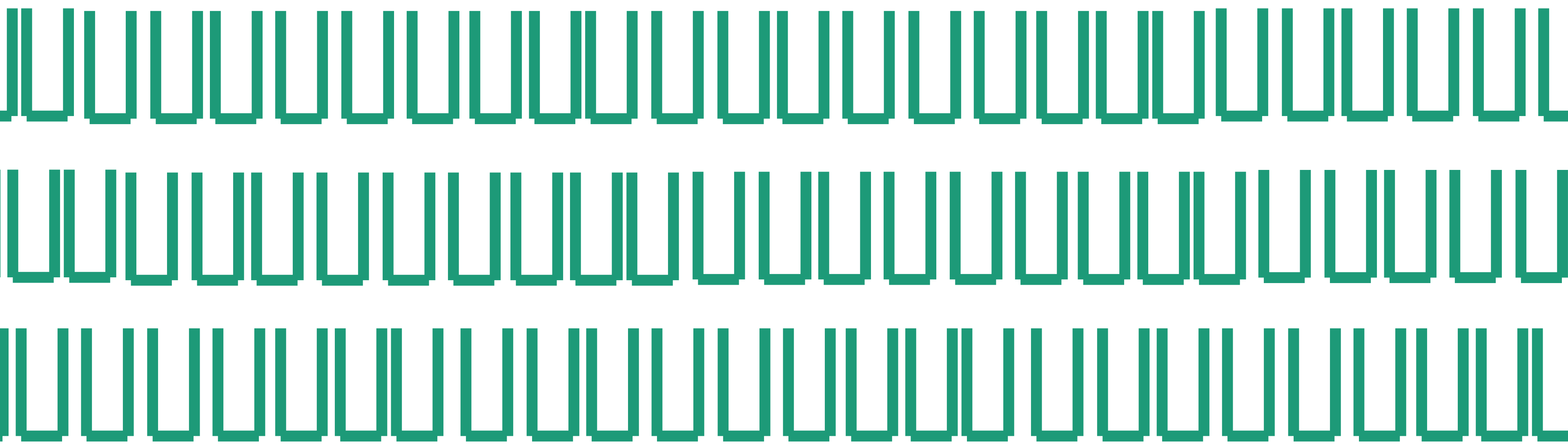
In time O(n).

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.

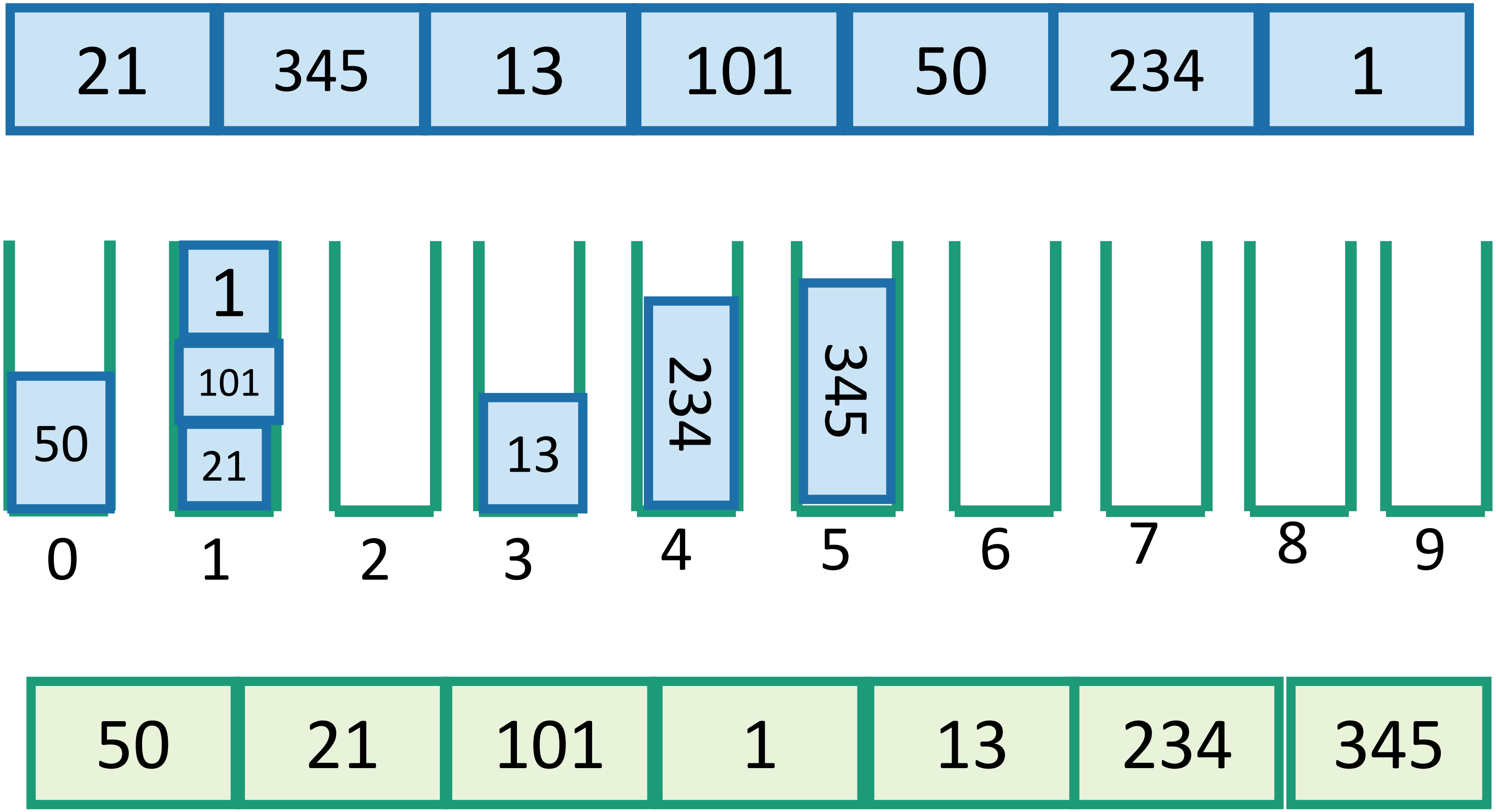


RadixSort

- For sorting integers up to size M
  - or more generally for lexicographically sorting strings
- Can use less space than CountingSort

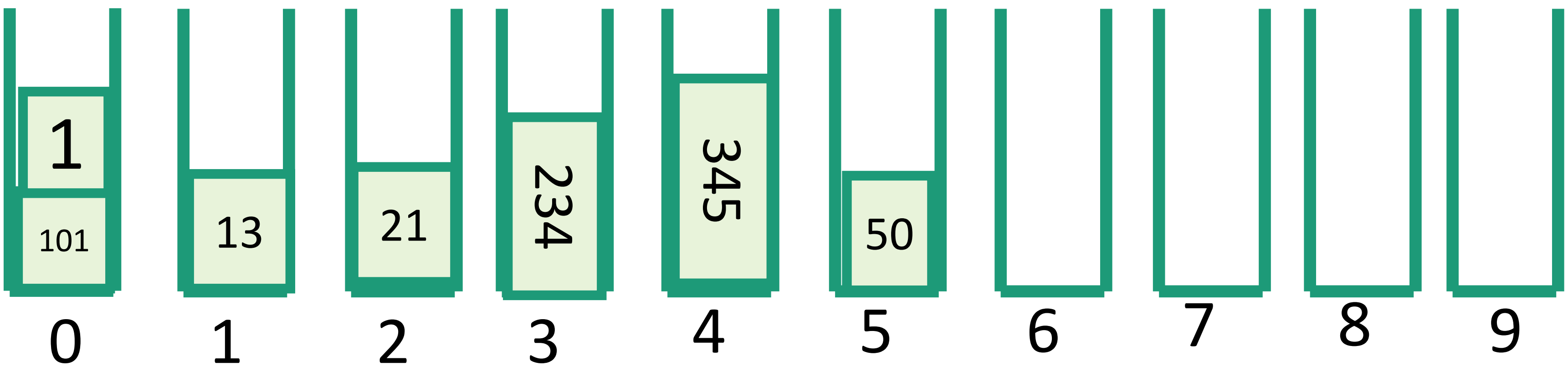
- Idea: CountingSort on the least-significant digit first, then the next least-significant, and so on.

Step 1: CountingSort on least significant digit



Step 2: CountingSort on the 2nd least sig. digit

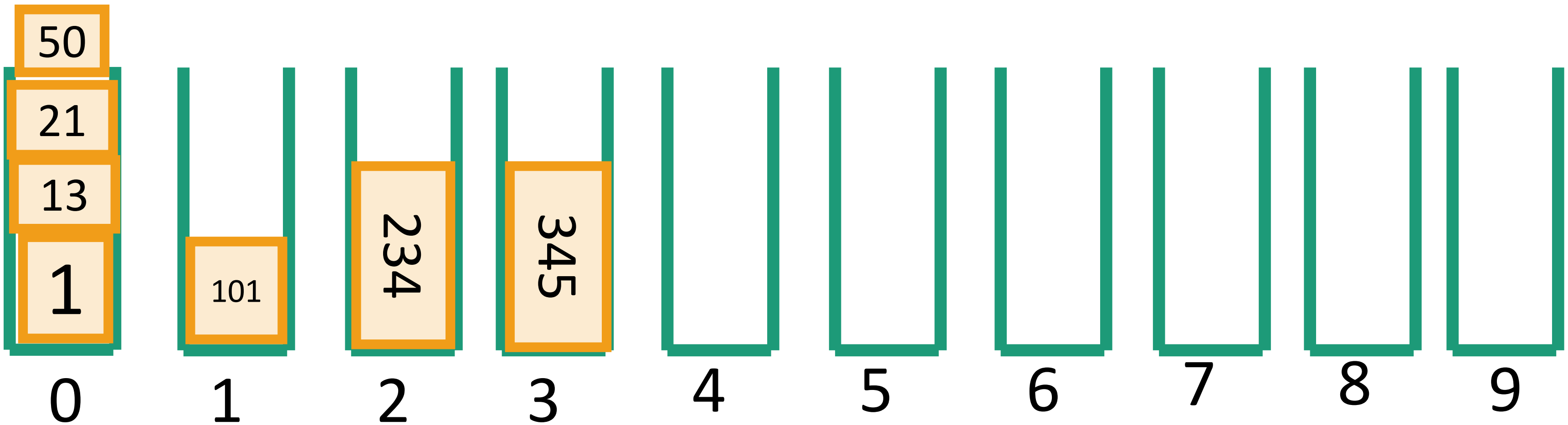
50	21	101	1	13	234	345



101	1	13	21	234	345	50

Step 3: CountingSort on the 3rd least sig. digit

101	1	13	21	234	345	50



1	13	21	50	101	234	345

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

What is the running time?

for RadixSorting numbers base-10.

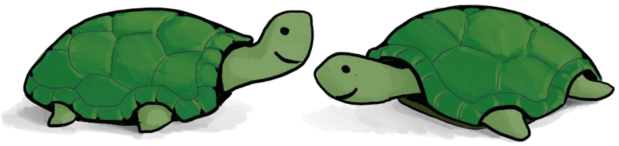
- Suppose we are sorting n d-digit numbers (in base 10).

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
Think: 3 minutes
Pair and share: 2 minutes

What is the running time?

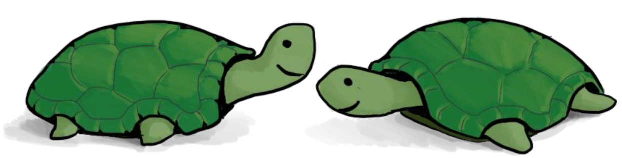
for RadixSorting numbers base-10.

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

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

021	345	013	101	050	234	001

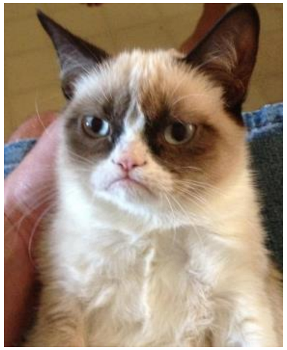
1. How many iterations are there?
   - d iterations
2. How long does each iteration take?
   - Time to initialize 10 buckets, plus time to put n numbers in 10 buckets. O(n).
3. What is the total running time?
   - O(nd)



Think-Pair-Share Terrapins

This doesn't seem so great

- To sort n integers, each of which is in {1,2,…,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 $x = [x_d x_{d-1} \ldots x_1]$ base 10, that means:
$$x = x_1 + x_2 \cdot 10 + \cdots + x_{d-1} \cdot 10^{d-2} + x_d \cdot 10^{d-1}$$
where $x_i \in \{0,1,\ldots,9\}$
- Suppose that $x_d \neq 0$. Then we have
  - $x \geq x_d \cdot 10^{d-1}$
  - $\log_{10}(x) + 1 - \log_{10}(x_d) \geq d$
  - $\log_{10}(x) + 1 > d$
  - $\lfloor \log_{10}(n) \rfloor + 1 \geq d$

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

(take logs of both sides and rearrange)

$\log_{10}(x_d) > 0$ since $x_d > 0$

Since d is an integer

- On the other hand, we also have
  - $x < (x_d+1) \cdot 10^{d-1}$
  - $\log_{10}(x) + 1 - \log_{10}(x_d + 1) < d$
  - $\log_{10}(x) < d$
  - $\lfloor \log_{10}(n) \rfloor + 1 \leq d$

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

(take logs of both sides and rearrange)

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

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)

Example: base 100

Original array:

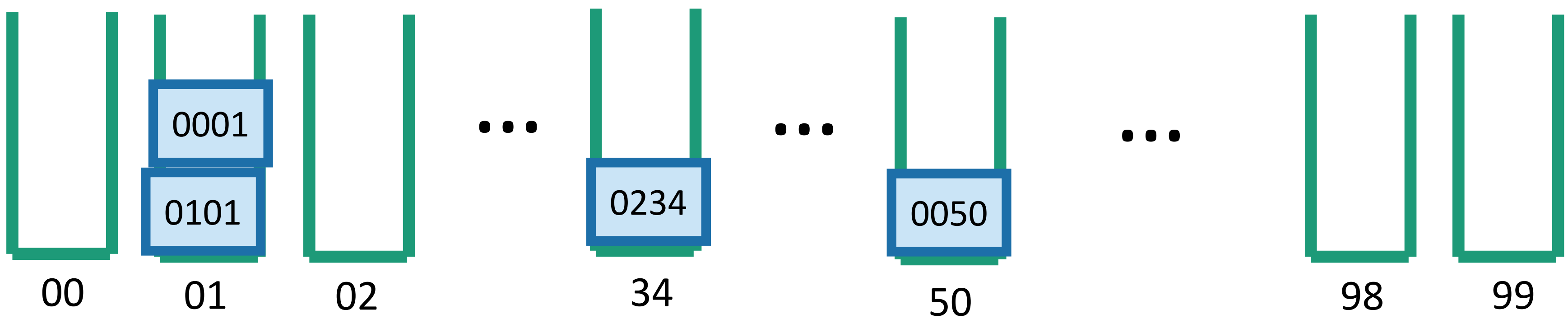
21	345	13	101	50	234	1

Example: base 100

Original array:

0021	0345	0013	0101	0050	0234	0001

100 buckets:

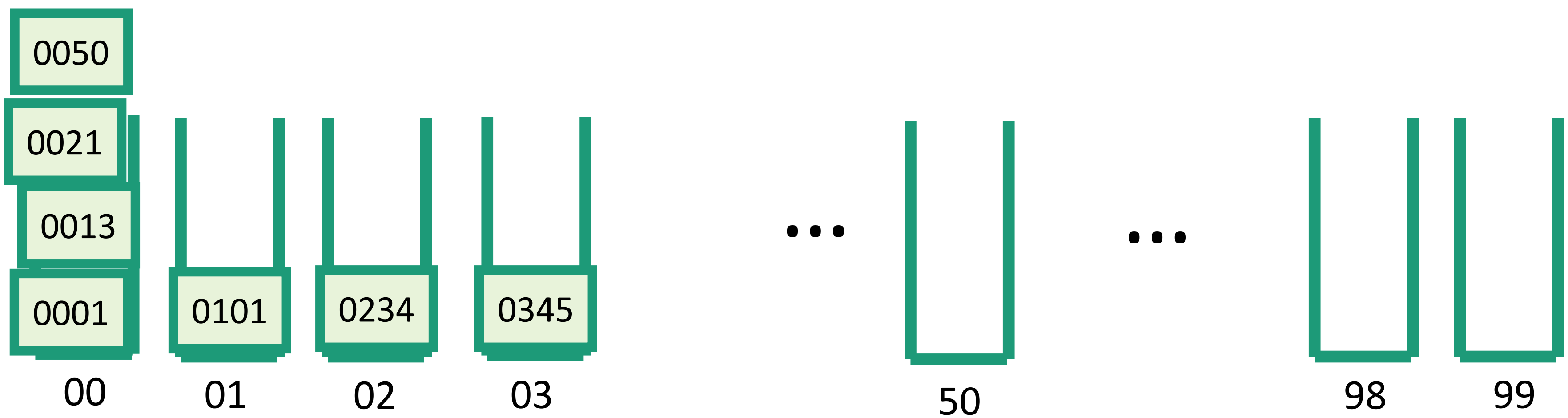


0101	0001	0013	0021	0234	0345	0050

Example: base 100

0101	0001	0013	0021	0234	0345	0050

100 buckets:



0001	0013	0021	0050	0101	0234	0345

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

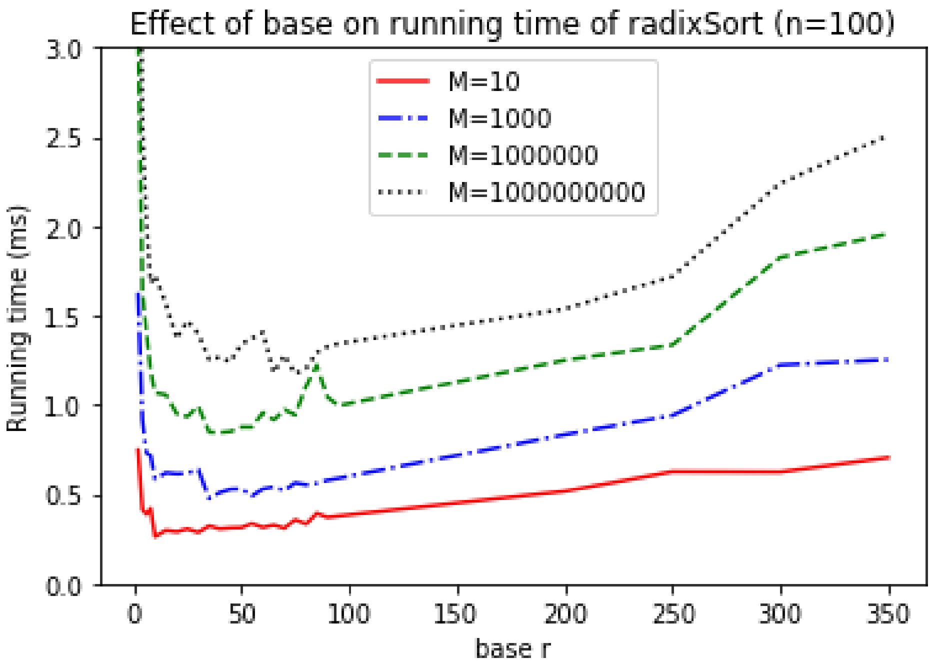
- Say we want to sort:
  - n integers,
  - maximum size M,
  - in base r.
- Number of iterations of RadixSort:
  - Same as number of digits, base r, of an integer x of max size M.
  - That is $d = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
  - Initialize r buckets, put n items into them
  -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

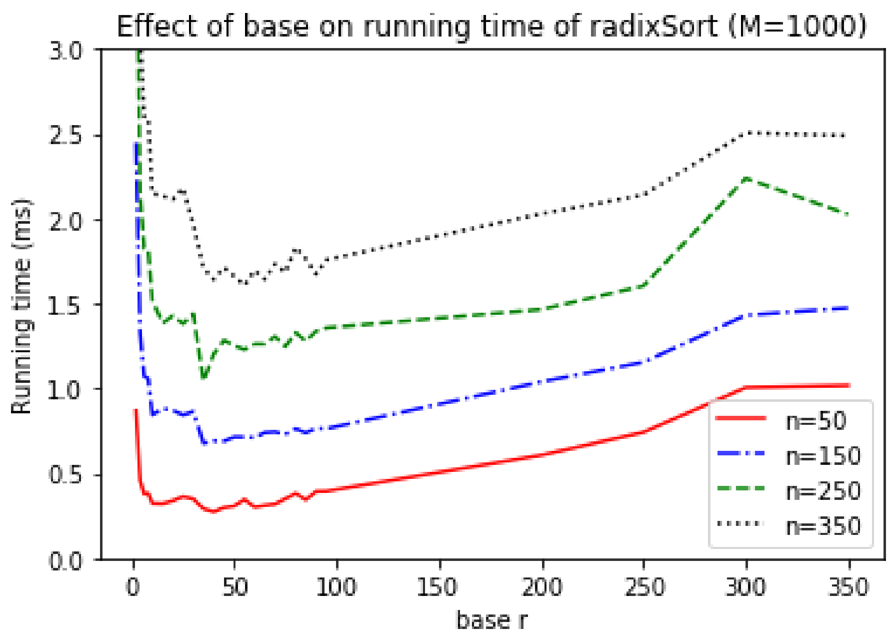
Convince yourself that this is the right formula for d.

Running time: $O\left(\left(\lfloor \log_r(M) \rfloor + 1\right) \cdot (n + r)\right)$

Trade-offs

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





A reasonable choice: r=n

- Running time:

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

Intuition: balance n and r here.

- Choose n=r:

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

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.

Next time

- Binary search trees!
- Balanced binary search trees!