

BDMI-课程编号-01510243

大数据与机器智能

排序算法

清华大学计算机科学与技术系人工智能所

黄世宇

huangsy1314@163.com

[CC BY-NC-SA](#)

目录:

- 快速排序

- 桶排序

快速排序

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
 - **BogoSort**
 - QuickSort
- **BogoSort** is a pedagogical tool.
- **QuickSort** is important to know. (in contrast with BogoSort...)



Assume A has
distinct entries

From your pre-lecture exercise: BogoSort

- **BogoSort(A)**
 - **While** true:
 - Randomly permute A.
 - Check if A is sorted.
 - **If** A is sorted, **return** A.

Suppose that you can draw a
random integer in $\{1, \dots, n\}$ in time
 $O(1)$. How would you randomly
permute an array in-place in time
 $O(n)$?



Ollie the over-achieving ostrich

- Let $X_i = \begin{cases} 1 & \text{if A is sorted after iteration i} \\ 0 & \text{otherwise} \end{cases}$
- $E[X_i] = \frac{1}{n!}$
- $E[\text{number of iterations until A is sorted}] = n!$

Expected Running time of BogoSort

This isn't random, so we can pull it out of the expectation.

$$\begin{aligned} &E[\text{running time on a list of length } n] \\ &= E[(\text{number of iterations}) * (\text{time per iteration})] \end{aligned}$$

$$= (\text{time per iteration}) * E[\text{number of iterations}]$$

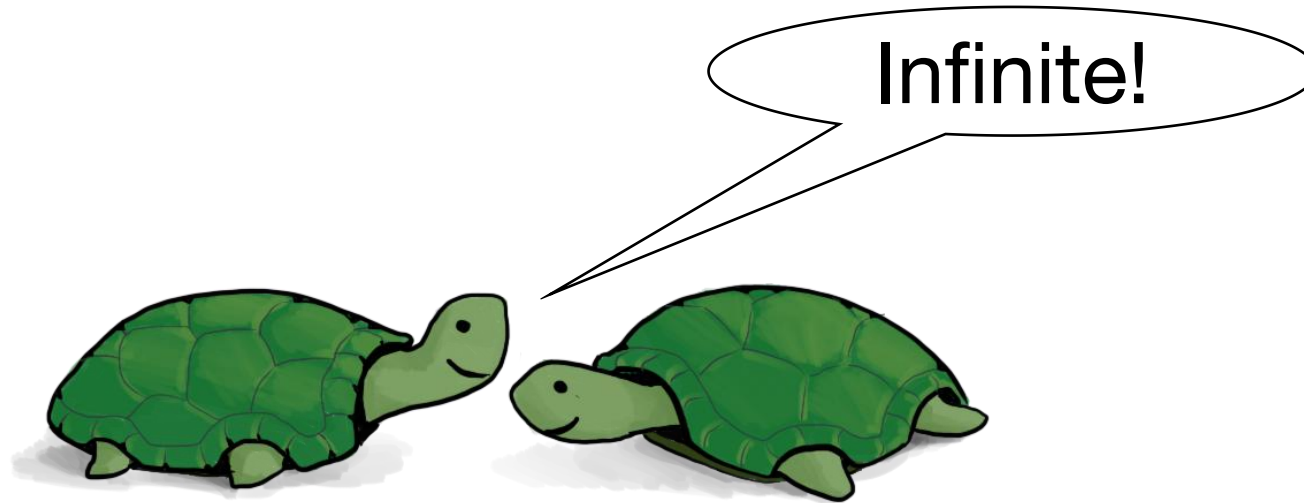
$$= O(n \cdot n!)$$

This is $O(n)$ (to permute and then check if sorted)

We just computed this. It's $n!$.

= REALLY REALLY BIG.

Worst-case running time of BogoSort?



Think-Pair-Share
Terrapins!

- **BogoSort(A)**
 - **While true:**
 - Randomly permute A.
 - Check if A is sorted.
 - If A is sorted, **return** A.



What have we learned?

- Expected running time:
 1. You publish your randomized algorithm.
 2. Bad guy picks an input.
 3. You get to roll the dice.
- Worst-case running time:
 1. You publish your randomized algorithm.
 2. Bad guy picks an input.
 3. Bad guy gets to “roll” the dice.
- Don't use bogoSort.

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.

- BogoSort

- QuickSort



- BogoSort is a pedagogical tool.
 - QuickSort is important to know. (in contrast with BogoSort...)

a better randomized algorithm:

QuickSort

- Expected runtime $O(n \log(n))$.
- Worst-case runtime $O(n^2)$.
- In practice works great!
 - (More later)

Quicksort

We want to sort this array.

For the rest of the lecture,
assume all elements of A are
distinct.

First, pick a
“pivot.”

Do it at random.

Next, partition the array into
“bigger than 5” or “less than
5”

Arrange
them like
so:

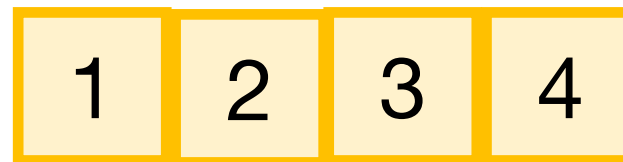
Recurse on
L and R:



This PARTITION
step takes time
 $O(n)$. (Notice that
we don't sort each
half).
[same as in
SELECT]

L = array with things
smaller than A[pivot]

R = array with things
larger than A[pivot]



PseudoPseudoCode for what we just saw

IPython Lecture
5 notebook for
actual code.

- QuickSort(A):
 - If $\text{len}(A) \leq 1$:
 - **return**
 - Pick some $x = A[i]$ at random. Call this the **pivot**.
 - **PARTITION** the rest of A into:
 - L (less than x) and
 - R (greater than x)
 - Replace A with [L, x, R] (that is, rearrange A in this order)
 - QuickSort(L)
 - QuickSort(R)

Assume that all elements
of A are distinct. How
would you change this if
that's not the case?



How would you do all this in-place?
Without hurting the running time?
(We'll see later...)



Running time?

- $T(n) = T(|L|) + T(|R|) + O(n)$
- In an ideal world...
 - if the pivot splits the array exactly in half...

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

- We've seen that a bunch:

$$T(n) = O(n \log(n)).$$



The expected running time of QuickSort is $O(n\log(n))$.

Proof:

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.

Remember, we are
assuming all elements of A
are distinct

Aside

why is $E[|L|] = \frac{n-1}{2}$?

- $E[|L|] = E[|R|]$
 - by symmetry
- $E[|L| + |R|] = n - 1$
 - because L and R make up everything except the pivot.
- $E[|L|] + E[|R|] = n - 1$
 - By linearity of expectation
- $2E[|L|] = n - 1$
 - Plugging in the first bullet point.
- $E[|L|] = \frac{n-1}{2}$
 - Solving for $E[|L|]$.

The expected running time of QuickSort is $O(n \log(n))$.

Proof:

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
 - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n \log(n))$.

*Disclaimer: this proof is wrong.



Red flag

We can use the same argument to prove something false.

- **Slow** sort(A):
 - If $\text{len}(A) \leq 1$:
 - return

- Pick the pivot x to be either $\max(A)$ or $\min(A)$, randomly

- \\ We can find the max and min in $O(n)$ time

- L (less than x) and

- R (greater than x)

- Replace A with [L, x, R] (that is, rearrange A in this order)

- **Slow** L)

- **Slow** R)

- Same recurrence relation:

$$T(n) = T(|L|) + T(|R|) + O(n)$$

- We still have $E[|L|] = E[|R|] = \frac{n-1}{2}$

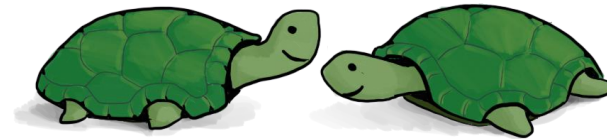
- But now, one of $|L|$ or $|R|$ is always $n-1$.

- You check: Running time is $\Theta(n^2)$, with probability 1.

The expected running time of SlowSort is $O(n \log(n))$.

Proof:

What's wrong???



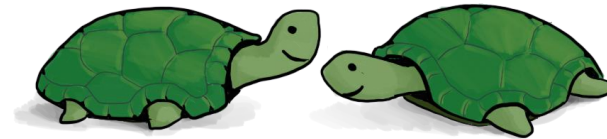
- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
 - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n \log(n))$.

***Disclaimer: this proof is wrong.**

The expected running time of SlowSort is $O(n \log(n))$.

Proof:

What's wrong???



- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
 - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n \log(n))$.

***Disclaimer: this proof is wrong.**

What's wrong?

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
 - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n \log(n))$.

This argument says:

That's not how
expectations work!



Plucky the Pedantic
Penguin

$T(n)$	=	some function of $ L $ and $ R $	✓
$E[T(n)]$	=	$E[\text{some function of } L \text{ and } R]$	✓
$E[T(n)]$	=	some function of $E L $ and $E R $	✗

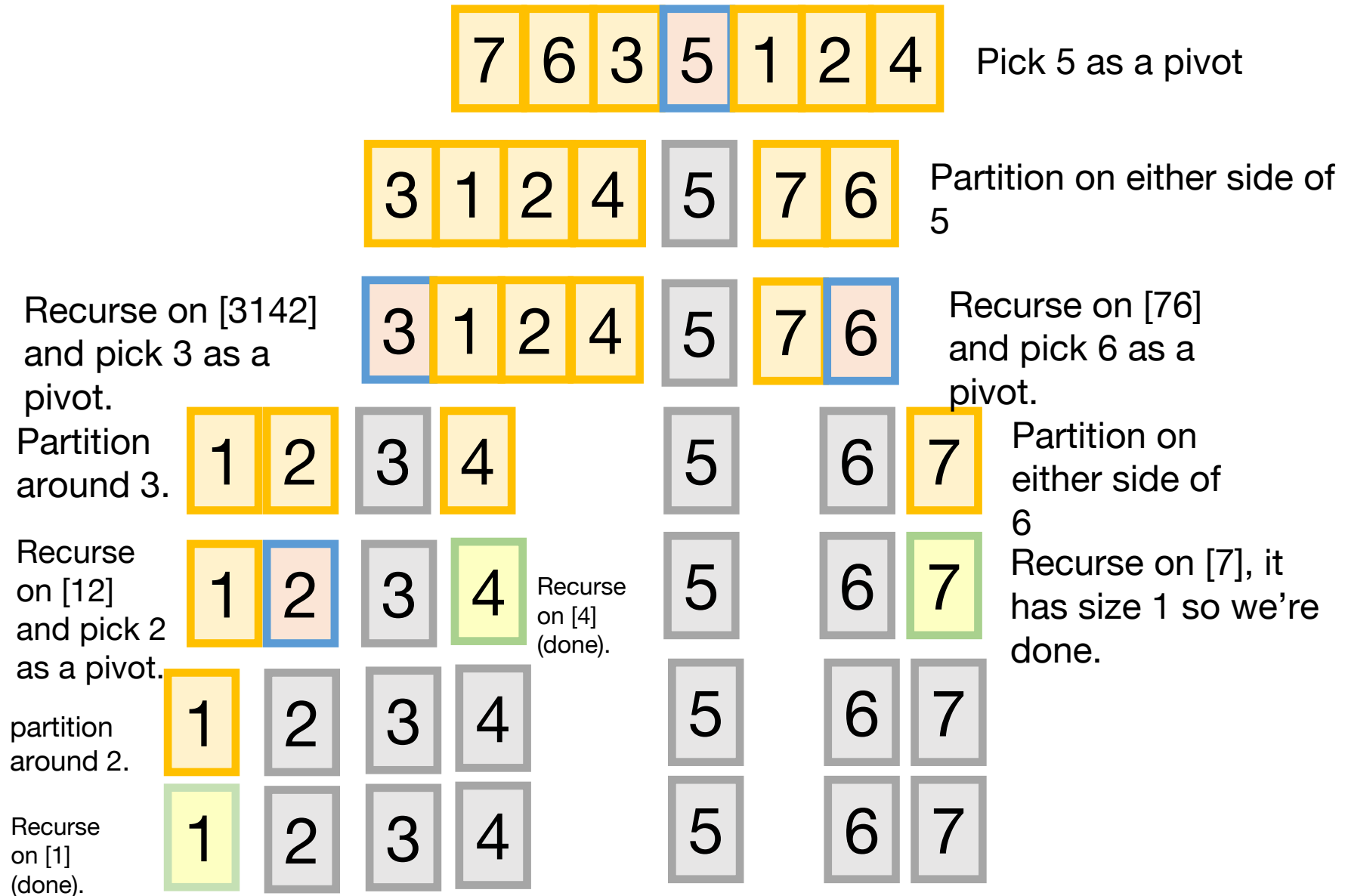
Instead

- We'll have to think a little harder about how the algorithm works.

Next goal:

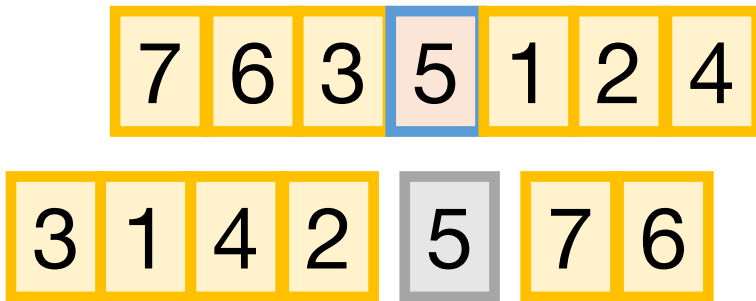
- Get the same conclusion, correctly!

Example of recursive calls

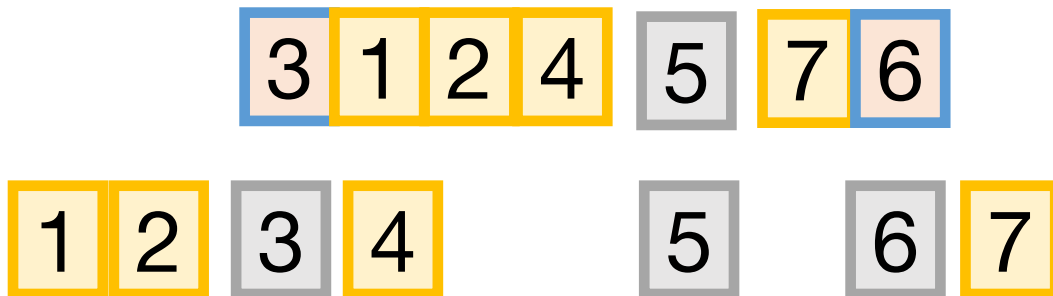


How long does this take to run?

- We will count the number of **comparisons** that the algorithm does.
 - This turns out to give us a good idea of the runtime. (Not obvious).
- How many times are any two items compared?



In the example before, everything was compared to 5 once in the first step....and never again.



But not everything was compared to 3.
5 was, and so were 1,2 and 4.
But not 6 or 7.

Each pair of items is compared either 0 or 1 times. Which is it?

7	6	3	5	1	2	4
---	---	---	---	---	---	---

Let's assume that the numbers in the array are actually the numbers 1,...,n

Of course this doesn't have to be the case! It's a good exercise to convince yourself that the analysis will still go through without this assumption. (Or see CLRS)



- **Whether or not a, b are compared** is a random variable, that depends on the choice of pivots. Let's say

$$X_{a,b} = \begin{cases} 1 & \text{if } a \text{ and } b \text{ are ever compared} \\ 0 & \text{if } a \text{ and } b \text{ are never compared} \end{cases}$$

- In the previous example $X_{1,5} = 1$, because item 1 and item 5 were compared.
- But $X_{3,6} = 0$, because item 3 and item 6 were NOT compared.

Counting comparisons

- The number of comparisons total during the algorithm is

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^n X_{a,b}$$

- The expected number of comparisons is

$$E \left[\sum_{a=1}^{n-1} \sum_{b=a+1}^n X_{a,b} \right] = \sum_{a=1}^{n-1} \sum_{b=a+1}^n E[X_{a,b}]$$

using linearity of expectations.

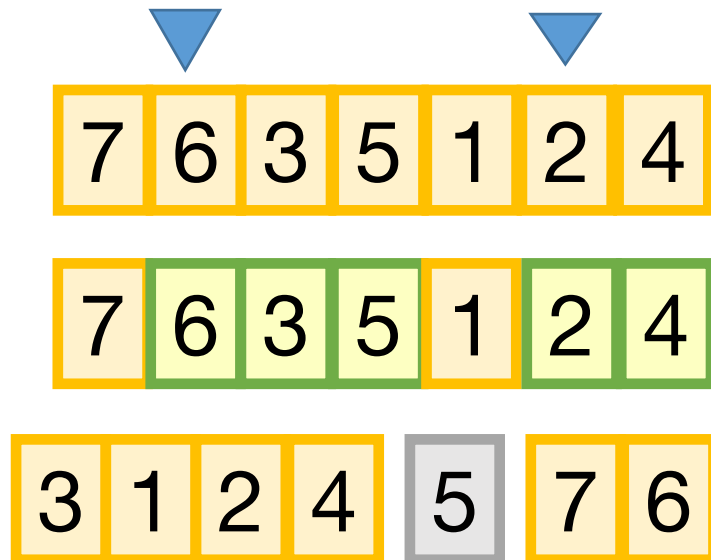
Counting comparisons

expected number of comparisons:

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^n E[X_{a,b}]$$

- So we just need to figure out $E[X_{a,b}]$
- $E[X_{a,b}] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$
 - (using definition of expectation)
- So we need to figure out:

$P(X_{a,b} = 1)$ = the probability that a and b are ever compared.



Say that $a = 2$ and $b = 6$. What is the probability that 2 and 6 are ever compared?

This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.

If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.

Counting comparisons

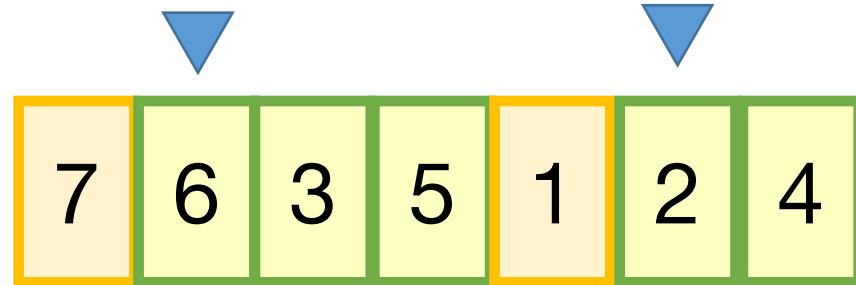
$$P(X_{a,b} = 1)$$

= probability a,b are ever compared

= probability that one of a,b are picked first out of all of the $b - a + 1$ numbers between them.

2 choices out of $b-a+1$...

$$= \frac{2}{b - a + 1}$$



Aside:

Why don't we care about 1 and 7?

In a bit more detail:

- Let $S = \{a, a+1, \dots, b\}$
- $P\{a, b \text{ are ever compared}\}$
 $= \sum_{\text{stuff}} P\{a \text{ or } b \text{ picked first out of } S \mid \text{stuff}\} \cdot P\{\text{stuff}\}$

where the sum is over all the stuff that does not involve S .

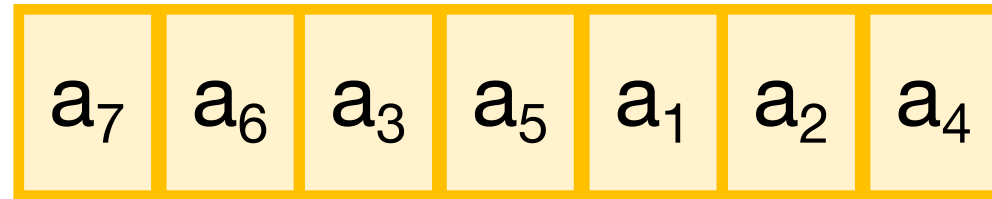
- But since that stuff is independent of what happens with S , this is equal to:

$$\begin{aligned} &= \sum_{\text{stuff}} P\{a \text{ or } b \text{ picked first out of } S\} \cdot P\{\text{stuff}\} \\ &= P\{a \text{ or } b \text{ picked first out of } S\} \cdot \sum_{\text{stuff}} P\{\text{stuff}\} \\ &= P\{a \text{ or } b \text{ picked first out of } S\} \\ &= 2/|S| \end{aligned}$$

Aside:

Why can we assume that the elements of the array are $\{1, 2, \dots, n\}$?

- More generally, say the elements of the array are $a_1 < a_2 < \dots < a_n$, so the array looks like:



- Then we'd do exactly the same thing, except we'd focus on the subscripts instead of the values. For example, the probability that a_2 and a_6 are ever compared is the probability that a_2 or a_6 are picked as a pivot before a_3, a_4 , or a_5 are.

All together now...

Expected number of comparisons

$$\begin{aligned} & \bullet E \left[\sum_{a=1}^{n-1} \sum_{b=a+1}^n X_{a,b} \right] \\ & \bullet = \sum_{a=1}^{n-1} \sum_{b=a+1}^n E[X_{a,b}] \\ & \bullet = \sum_{a=1}^{n-1} \sum_{b=a+1}^n P(X_{a,b} = 1) \\ & \bullet = \sum_{a=1}^{n-1} \sum_{b=a+1}^n \frac{2}{b-a+1} \end{aligned}$$

This is the expected number of comparisons throughout the algorithm

linearity of expectation

definition of expectation

the reasoning we just did

- This is a big nasty sum, but we can do it.
- We get that this is less than $2n \ln(n)$.

Do this sum!



Ollie the over-achieving ostrich

Almost done

- We saw that $E[\text{number of comparisons}] = O(n \log(n))$
- Is that the same as $E[\text{running time}]$?
- In this case, **yes**.
- We need to argue that the running time is dominated by the time to do comparisons.
- (See CLRS for details).
- QuickSort(A):
 - If $\text{len}(A) \leq 1$:
 - **return**
 - Pick some $x = A[i]$ at random. Call this the **pivot**.
 - **PARTITION** the rest of A into:
 - L (less than x) and
 - R (greater than x)
 - Replace A with [L, x, R] (that is, rearrange A in this order)
 - QuickSort(L)
 - QuickSort(R)

What have we learned?

- The expected running time of QuickSort is $O(n \log(n))$

Worst-case running time

- Suppose that an adversary is choosing the “random” pivots for you.
- Then the running time might be $O(n^2)$
 - Eg, they'd choose to implement SlowSort
 - In practice, this doesn't usually happen.



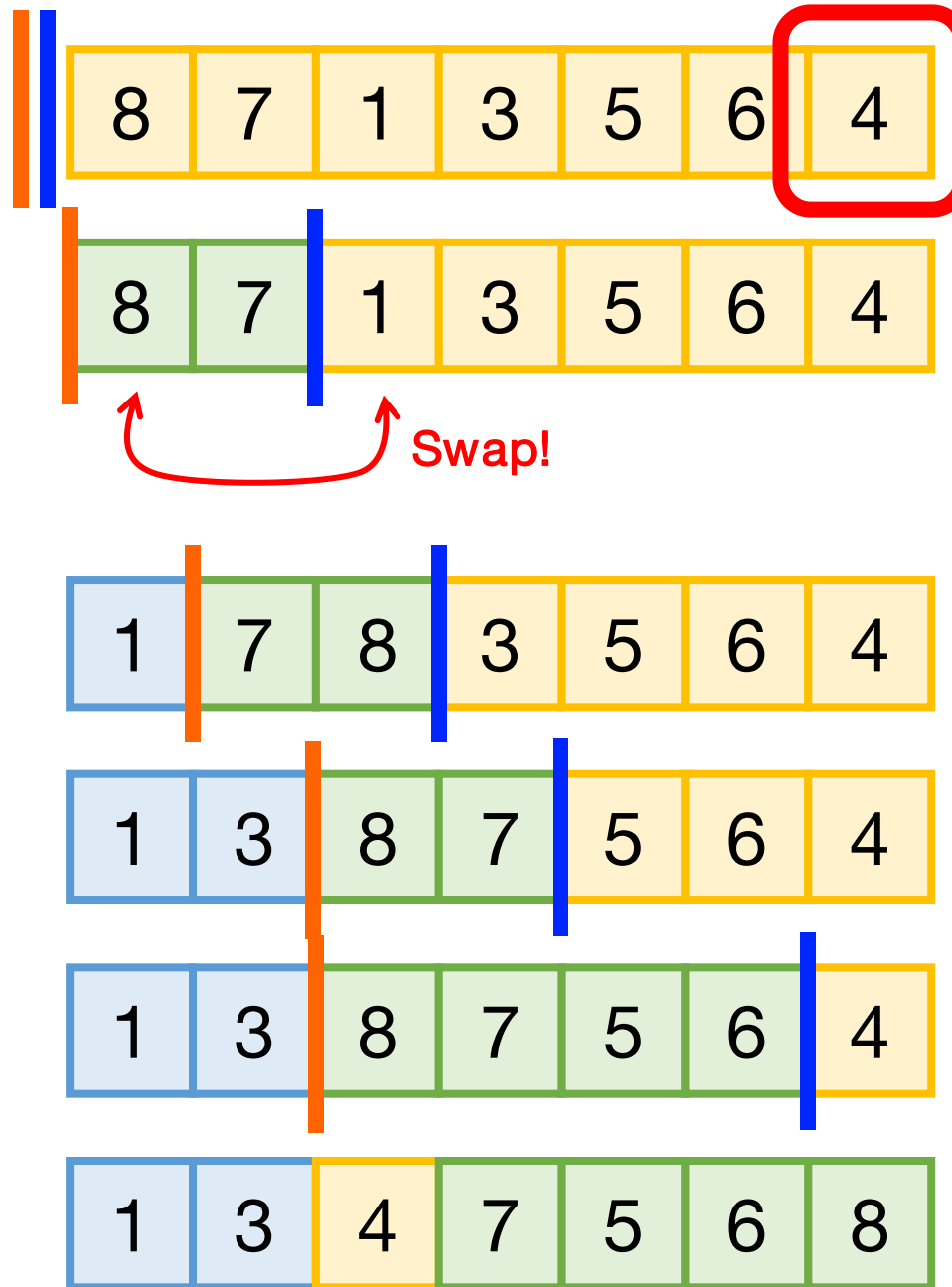
A note on implementation

- Our pseudocode is easy to understand and analyze, but is not a good way to implement this algorithm.

```
• QuickSort(A):  
  • If len(A) <= 1:  
    • return  
  • Pick some x = A[i] at random. Call this the pivot.  
  • PARTITION the rest of A into:  
    • L (less than x) and  
    • R (greater than x)  
  • Replace A with [L, x, R] (that is, rearrange A in this order)  
  • QuickSort(L)  
  • QuickSort(R)
```



- Instead, implement it **in-place** (without separate L and R)
 - You may have seen this in 106b.
 - Here are some Hungarian Folk Dancers showing you how it's done:
<https://www.youtube.com/watch?v=ywWBy6J5gz8>
 - Check out IPython notebook for Lecture 5 for two different ways.

A better way to do Partition




Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize  and 

Step  forward.

When  sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars. Repeat till the end, then put the pivot in the right place.

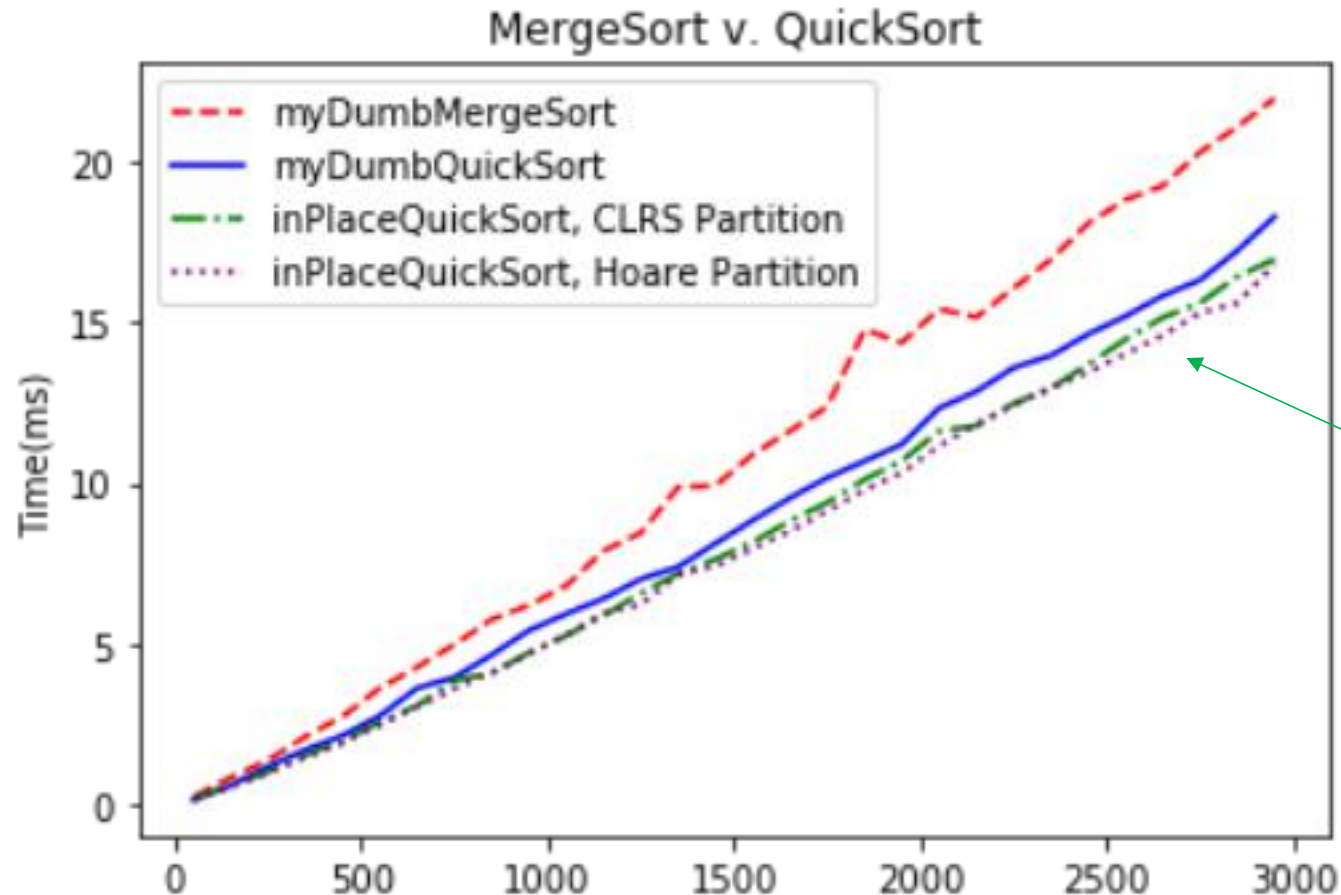
See CLRS or Lecture 5 IPython notebook for pseudocode/real

QuickSort vs. smarter QuickSort vs Mergesort?



See IPython notebook for Lecture
5

- All seem pretty comparable...



Hoare Partition is a different way of doing it (c.f. CLRS Problem 7-1), which you might have seen elsewhere.

You are not responsible for knowing it for this class.

The slicker in-place ones use less space, and also are a smidge faster on my system.

QuickSort vs MergeSort

*What if you want $O(n \log(n))$ worst-case runtime and stability?
Check out “Block Sort” on Wikipedia!

	QuickSort (random pivot)	MergeSort (deterministic)	Understand this
Running time	<ul style="list-style-type: none">Worst-case: $O(n^2)$Expected: $O(n \log(n))$	Worst-case: $O(n \log(n))$	
Used by	<ul style="list-style-type: none">Java for primitive typesC qsortUnixg++	<ul style="list-style-type: none">Java for objectsPerl	These are just for fun. (Not on exam).
In-Place? (With $O(\log(n))$ extra memory)	Yes, pretty easily	Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice runtime).	
Stable?	No	Yes	
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists	

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
 - **BogoSort**
 - **QuickSort**



- **BogoSort** is a pedagogical tool.
- **QuickSort** is important to know. (in contrast with BogoSort...)

Recap



Recap

- How do we measure the runtime of a randomized algorithm?
 - Expected runtime
 - Worst-case runtime
- **QuickSort** (with a random pivot) is a randomized sorting algorithm.
 - In many situations, QuickSort is nicer than MergeSort.
 - In many situations, MergeSort is nicer than QuickSort.



Code up QuickSort and MergeSort in a few different languages, with a few different implementations of lists A (array vs linked list, etc).

What's faster?

(This is an exercise best done in C where you have a bit more control than in Python).



Ollie the over-achieving

Next time

- Can we sort faster than $\Theta(n \log(n))$??

Before next time

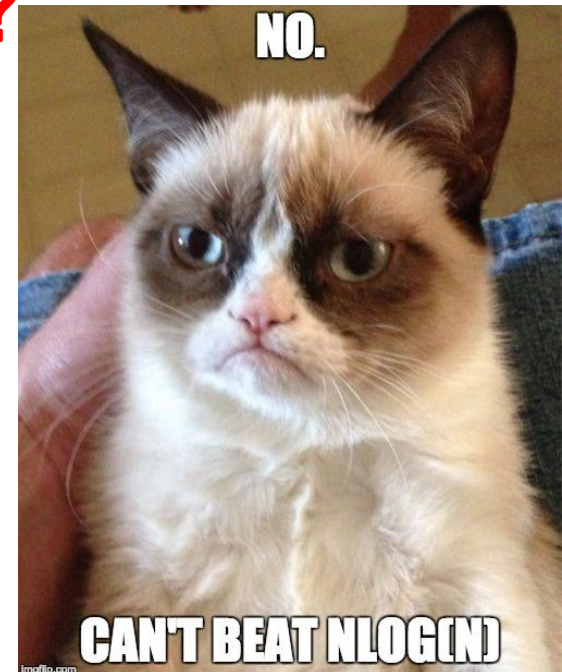
- ***Pre-lecture exercise*** for Lecture 6.
 - Can we sort even faster than QuickSort/MergeSort?

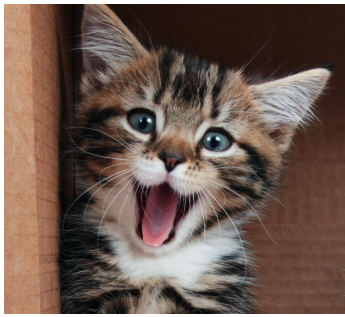
Sorting

- 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?

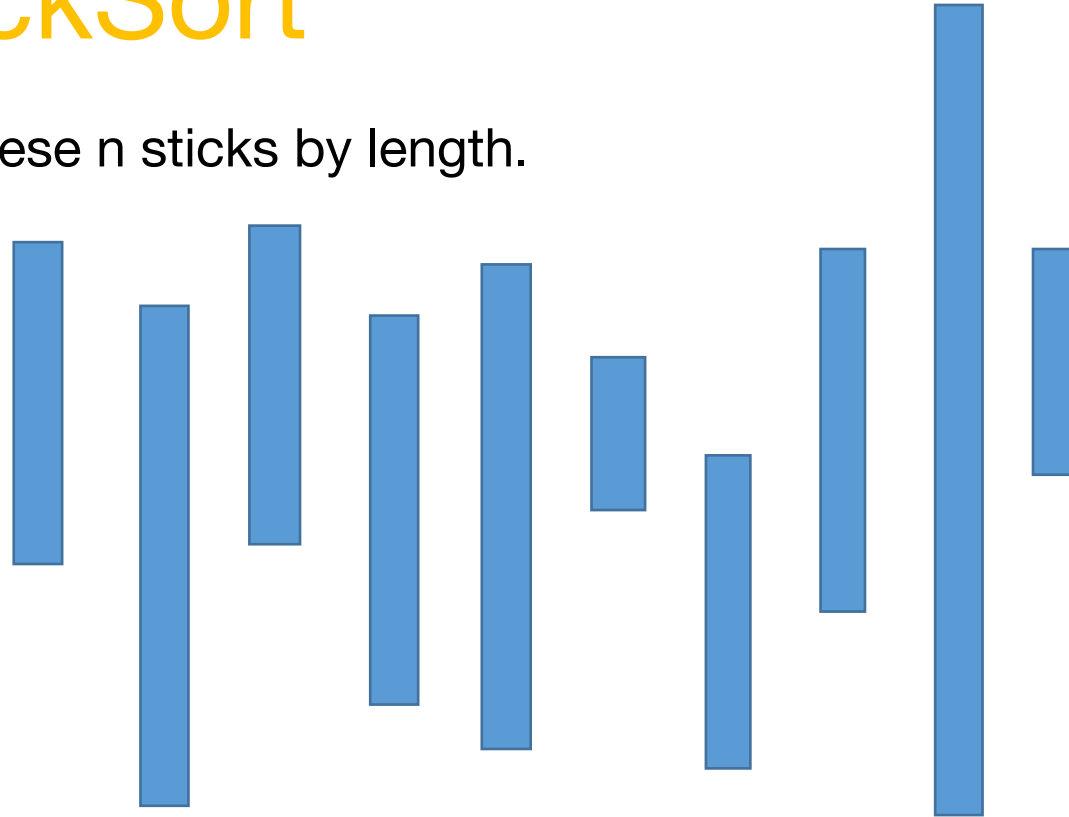
Depends on
who you ask...





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?

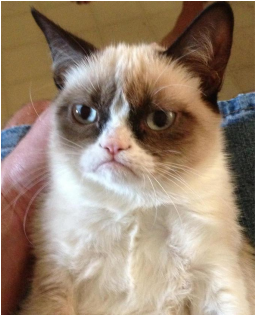
- Input: array
- Output: sorted array
- Operations allowed: comparisons

-vs-

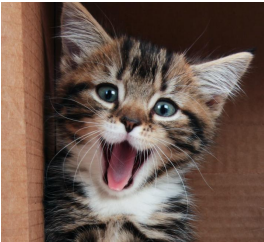
- Input: sticks
- Output: sorted sticks in vertical order
- Operations allowed: dropping on tables

- What are reasonable models of computation?

Today: two (more) models

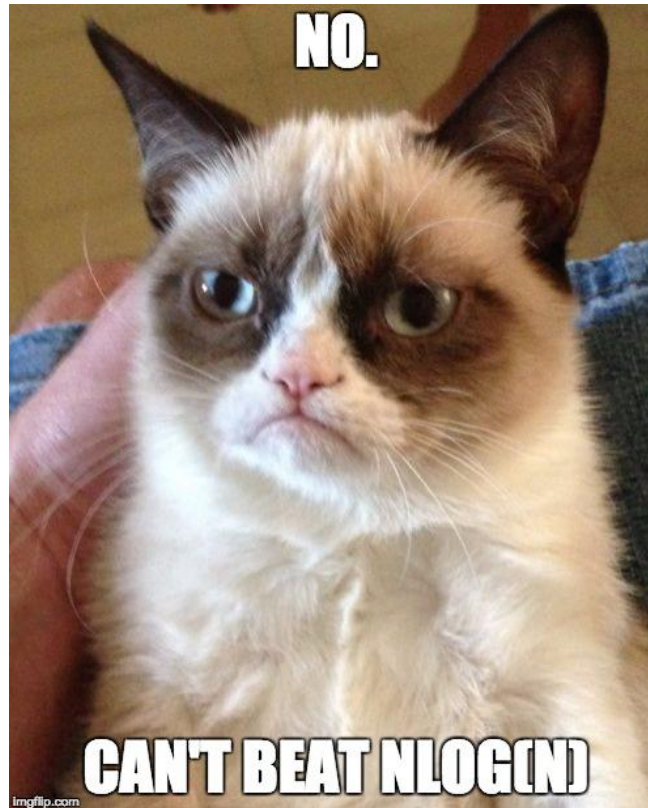


- Comparison-based sorting model
 - This includes MergeSort, QuickSort, InsertionSort
 - We'll see that any algorithm in this model must take at least $\Omega(n \log(n))$ steps.



- 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”

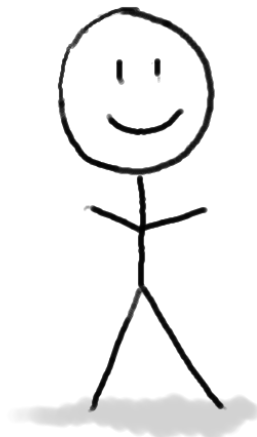
Want to sort these items.
There's some ordering on them, but we don't know what it is.

Is  bigger than  ?



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

YES



Algorithm

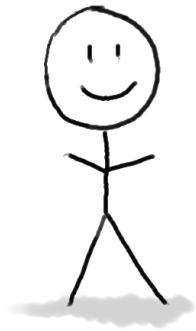
The algorithm's job is
to output a correctly
sorted list of all the
objects.

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:

	7	6	3	5	1	4	2
				Pivot!			



Is

7

bigger than

5

?

YES

Is

6

bigger than

5

?

YES

Is

3

bigger than

5

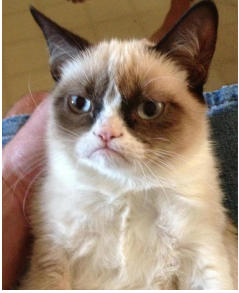
?

NO

5



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

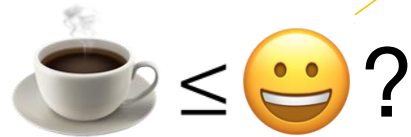


Sort these three things.



YES

NO



YES

NO



YES

NO

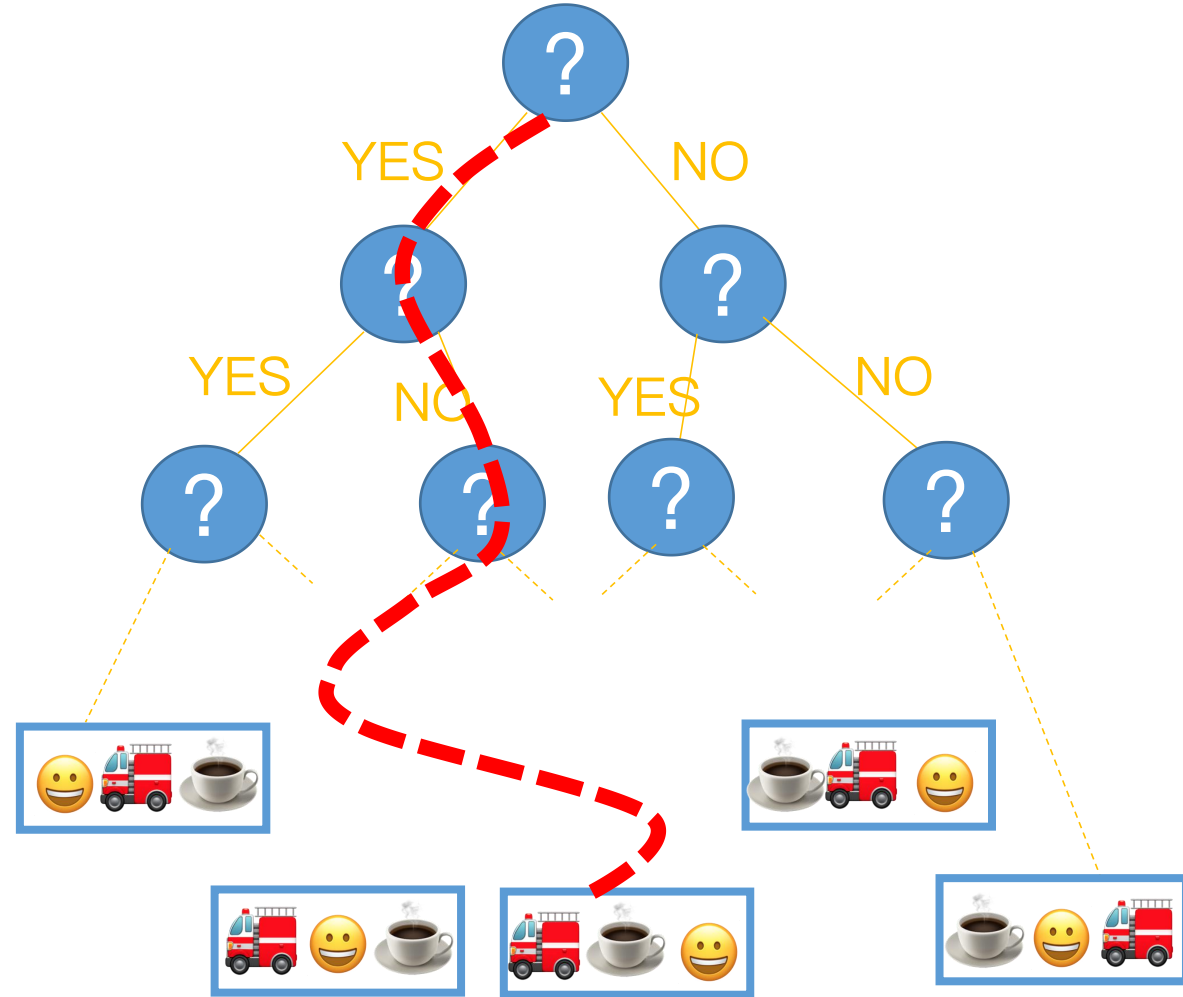


etc...

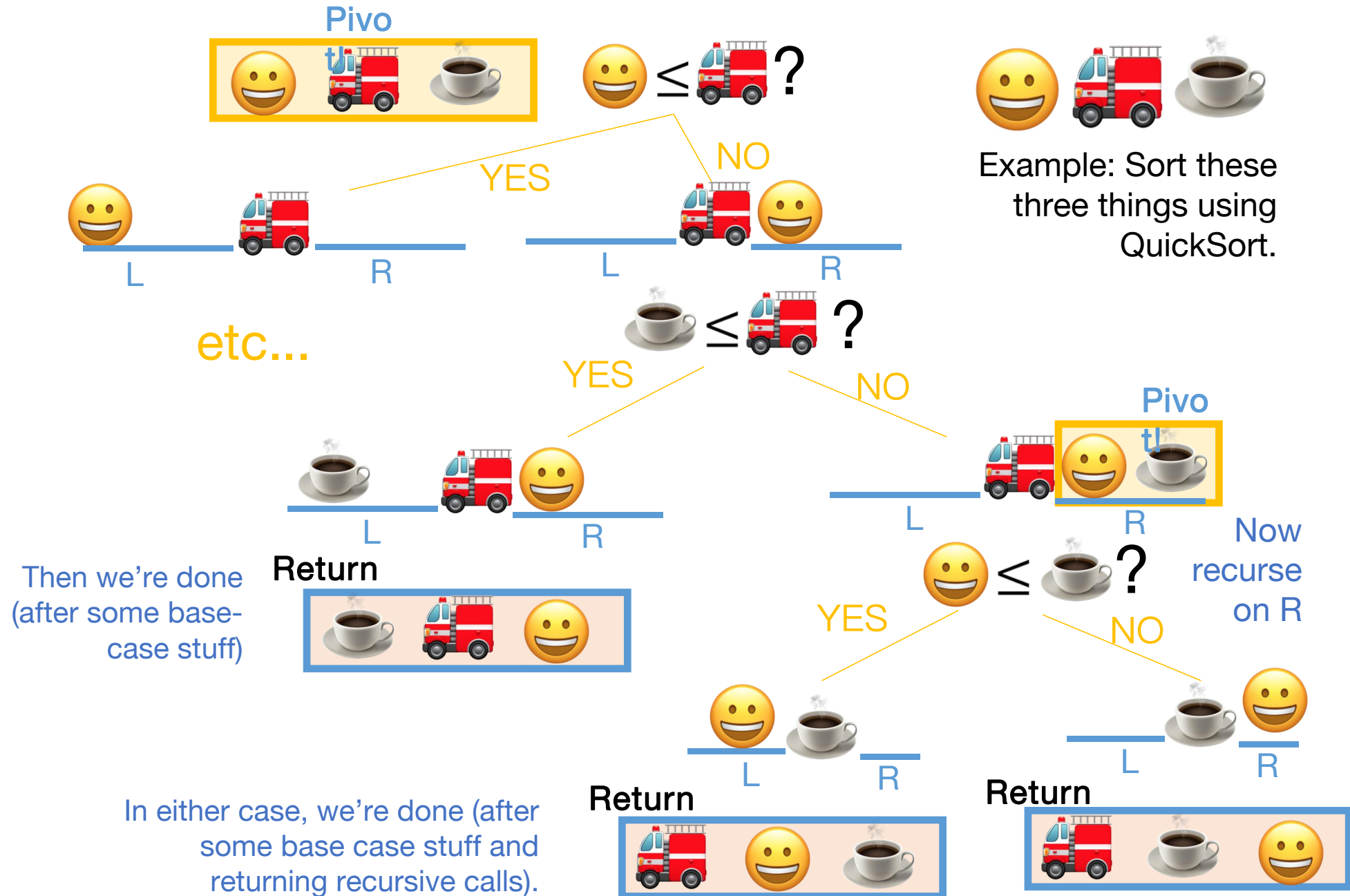


Decision trees

- 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.

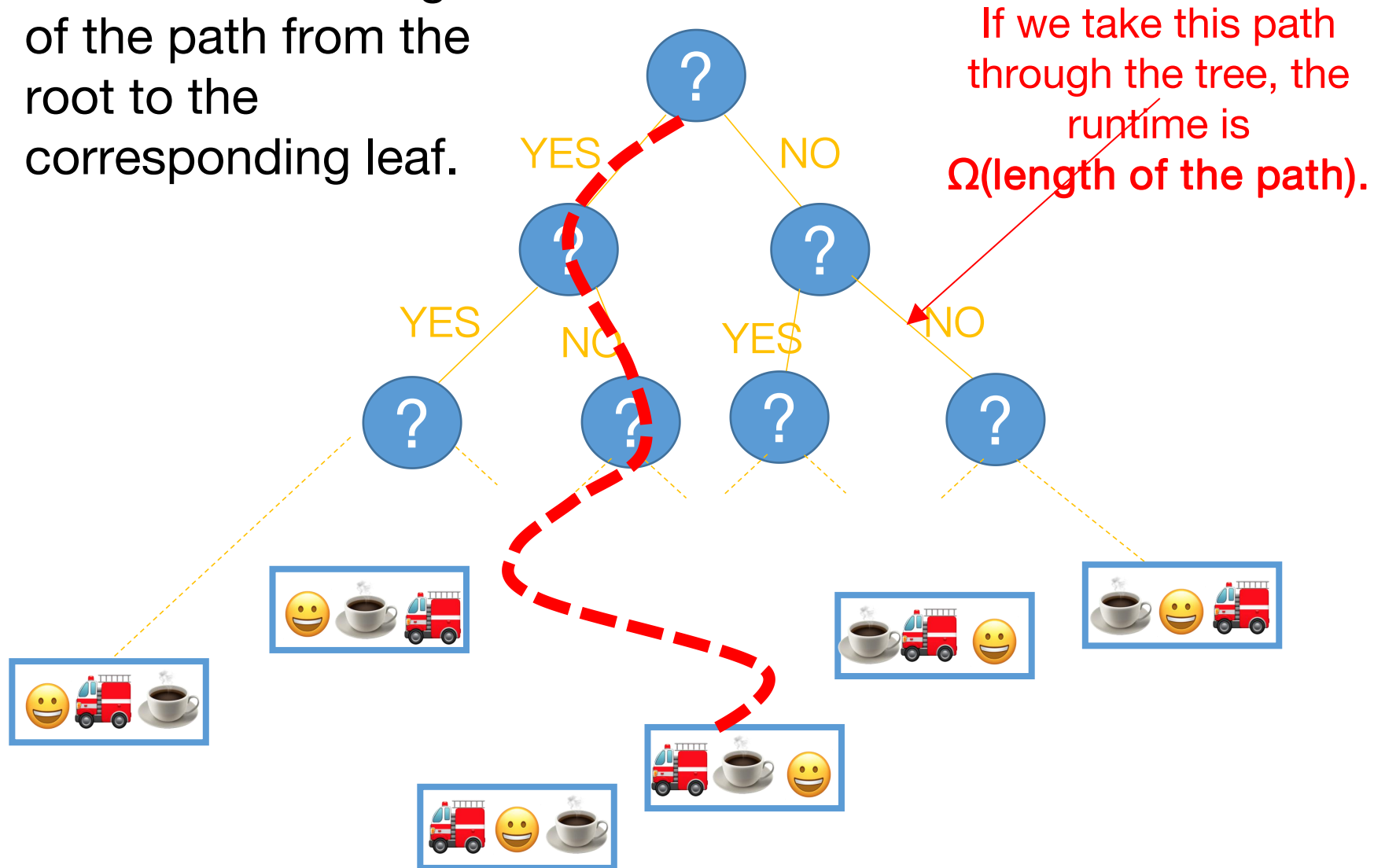


Comparison-based algorithms look like decision trees.



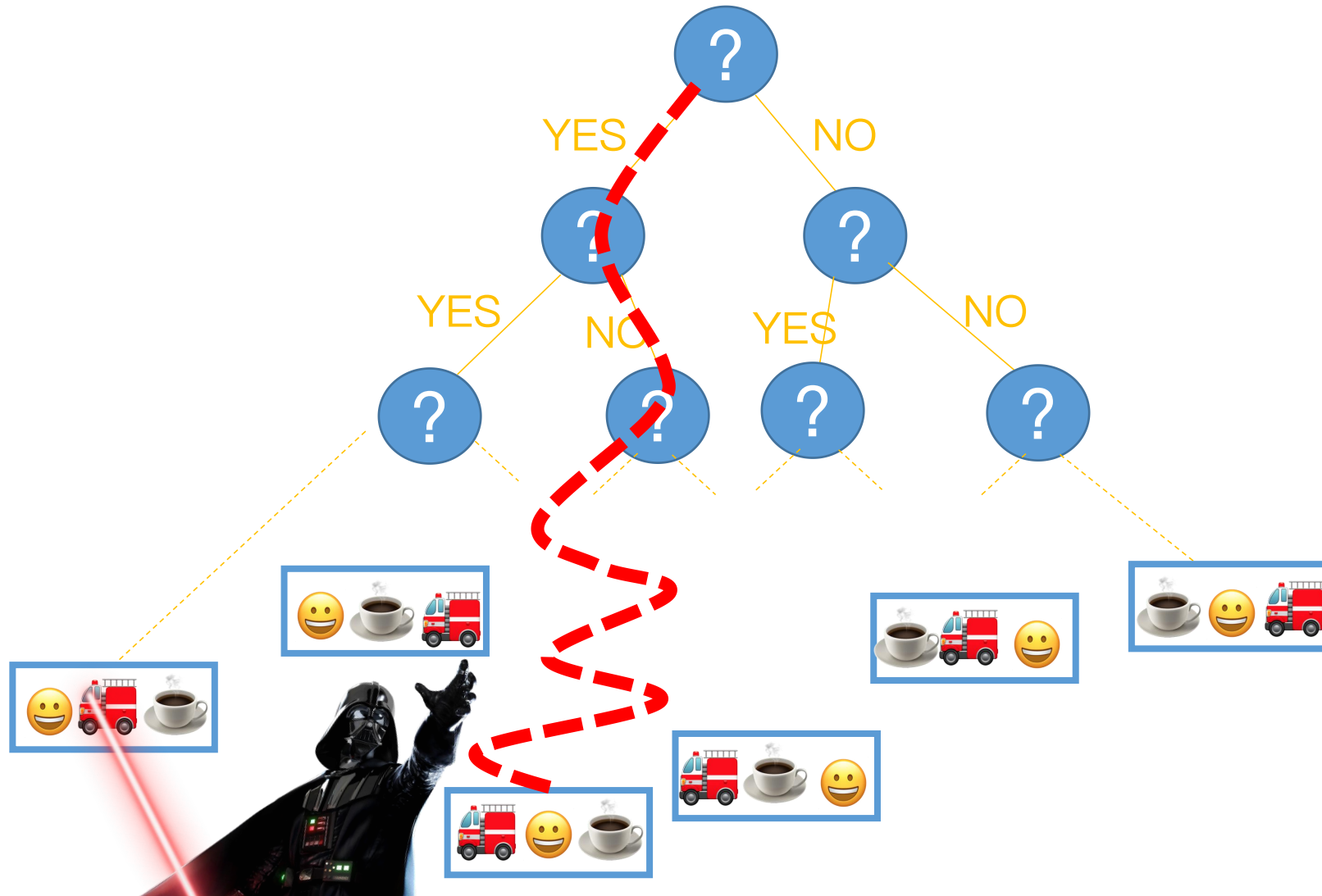
Q: What's the runtime on a particular input?

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



Q: What's the worst-case runtime?

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

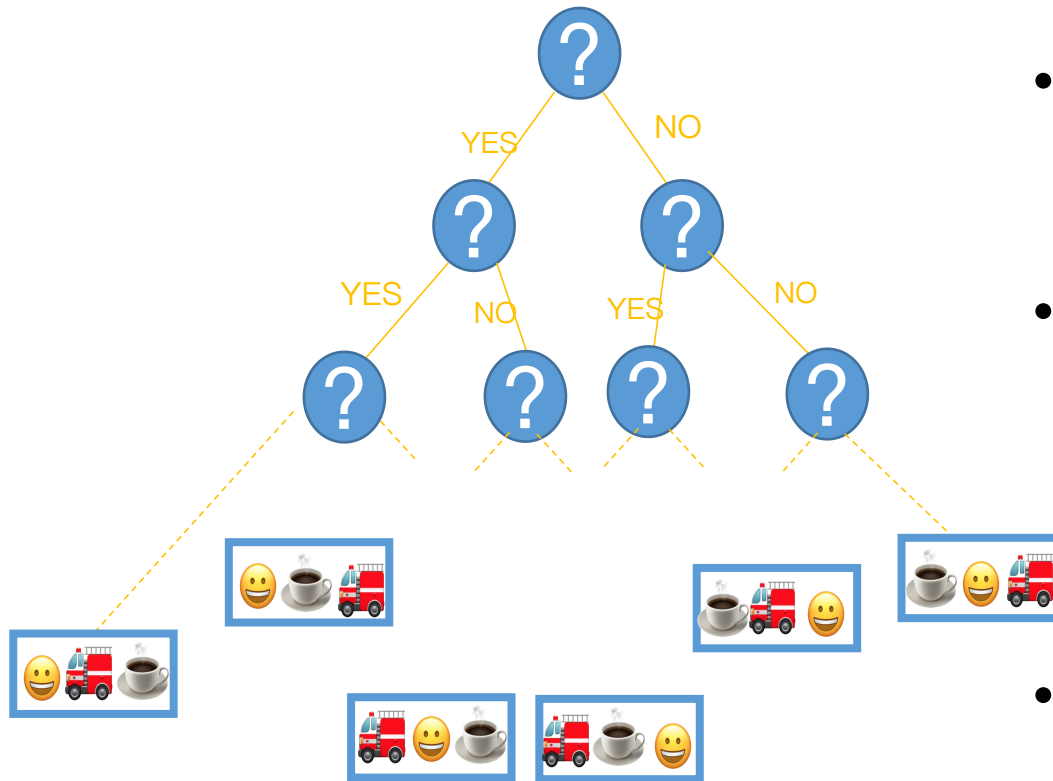


How long is the longest path?



being sloppy
about floors and
ceilings!

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



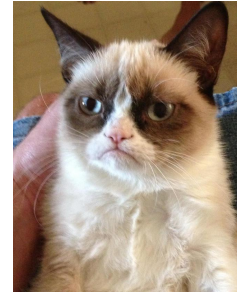
- This is a binary tree with at least $n!$ leaves.
- The shallowest tree with $n!$ leaves is the completely balanced one, which has depth $\log(n!)$.
- So in all such trees, the longest path is at least $\log(n!)$.

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



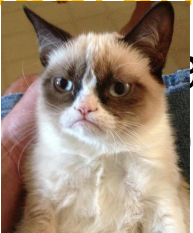
- **Theorem:**
 - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.
- **Proof recap:**
 - Any deterministic comparison-based algorithm can be represented as a decision tree with $n!$ leaves.
 - The worst-case running time is at least the depth of the decision tree.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

-  Randomized comparison-based sorting algorithm must take $\Omega(n^2)$ steps in expectation.

- Proof:

- see reading posted on website
 - (Avrim Blum's notes)
- (same ideas as deterministic case)

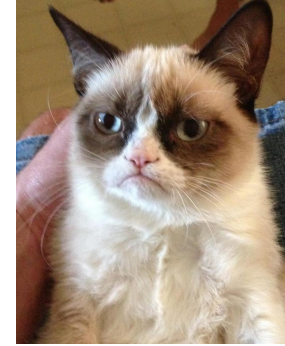
Try to prove this
yourself!



\end{Aside}

Ollie the over-achieving

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



桶排序

Another model of computation

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



instead of



Pre-lecture exercise

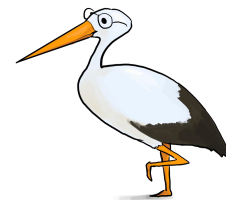
- How long does it take to sort n people by their month of birth?
- [discussion]



1 (Jan)



1 (Jan)



4 (Apr)



5 (May)

Another model of computation

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



instead of



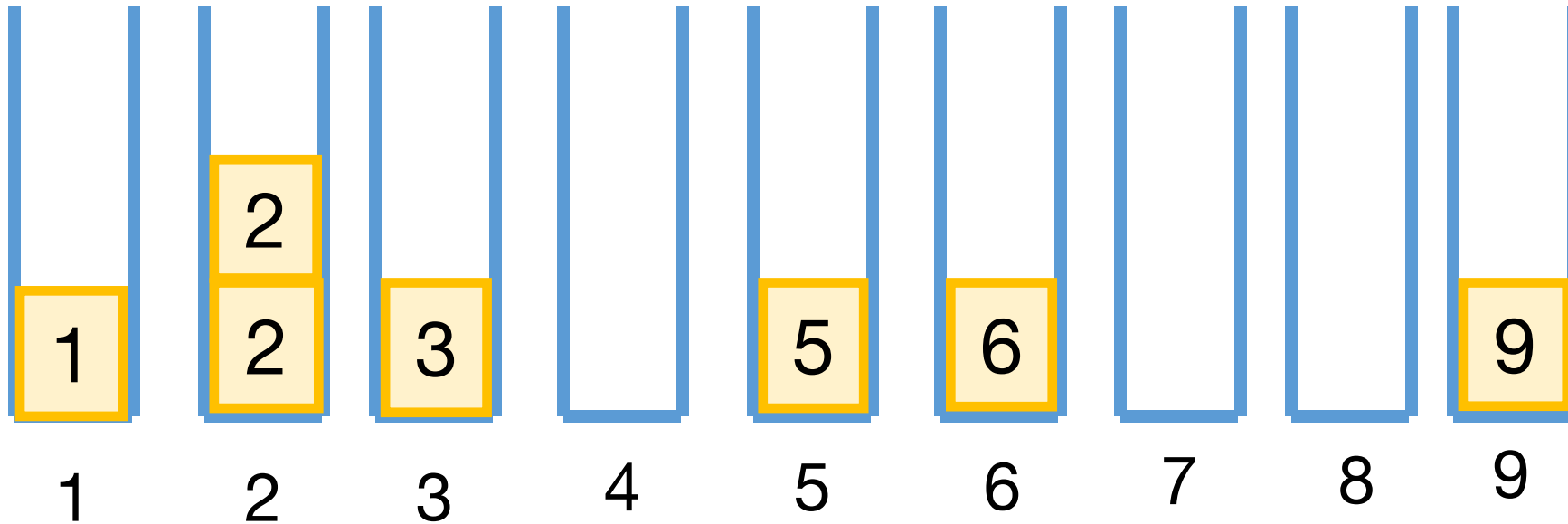
Why might this help?



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

BucketSort:

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



Concatenate
the buckets!

SORTED!

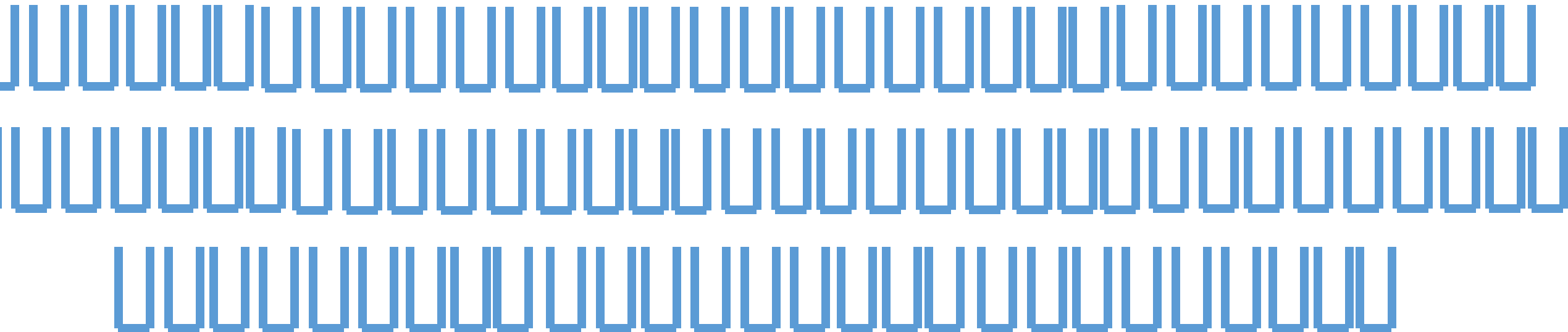
In time $O(n)$.

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	1234 5	13	2^{1000}	50	10000000 0	1
---	-------------	----	------------	----	-----------------	---

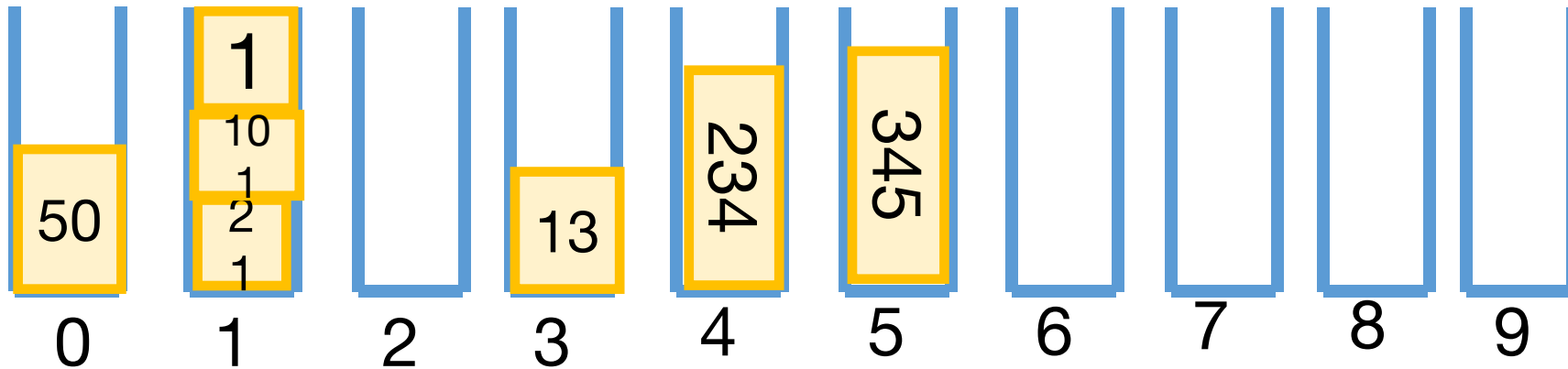
- Need to assume there are not too many such values.



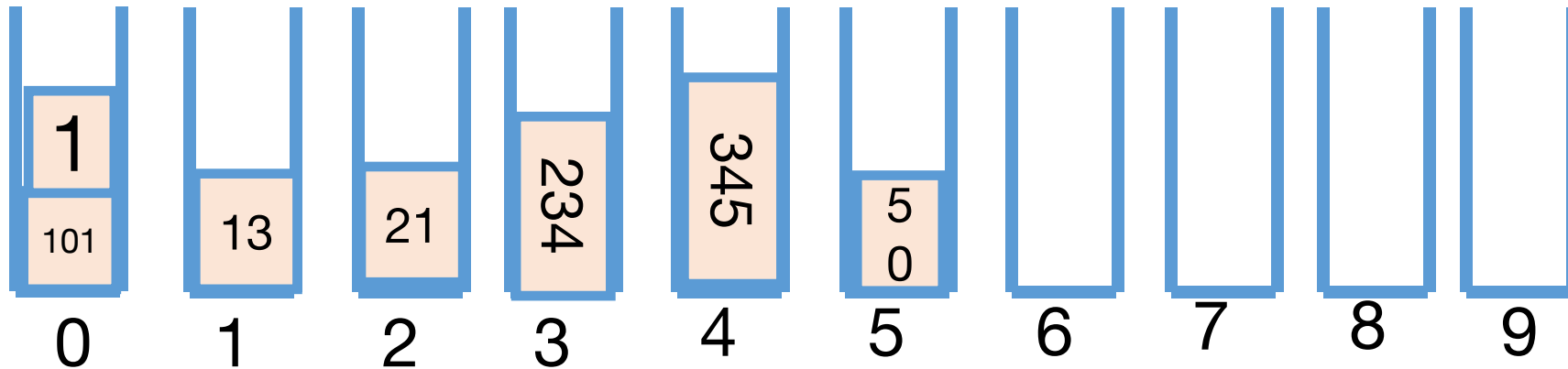
RadixSort

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

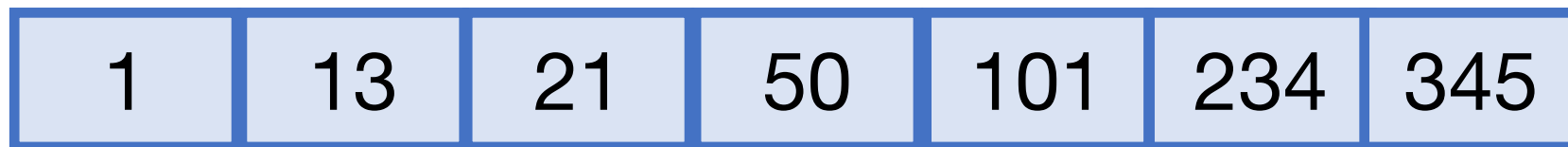
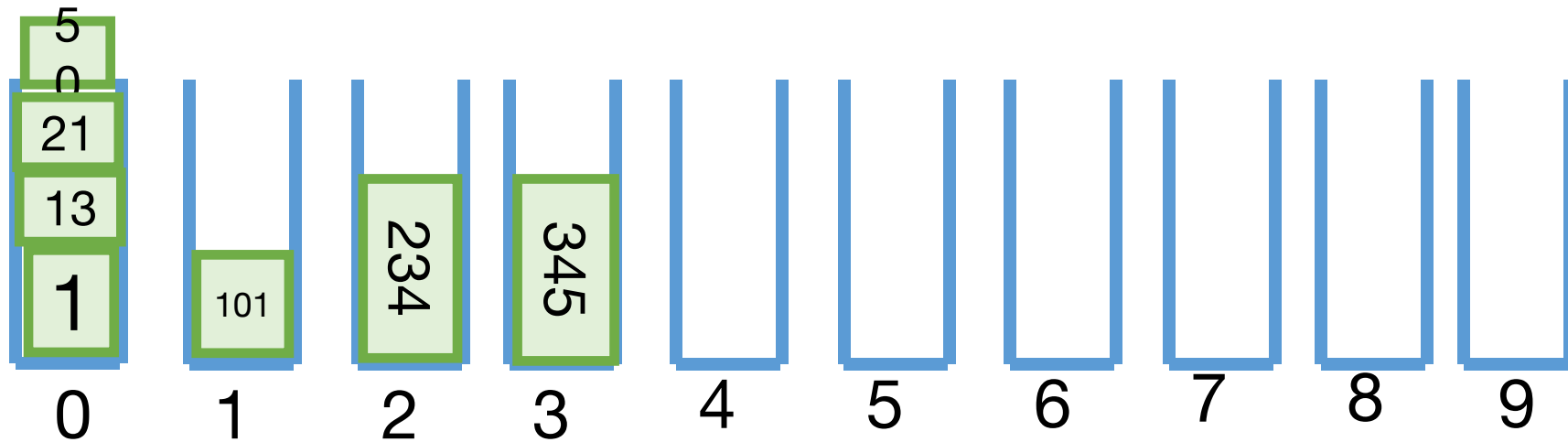
Step 1: BucketSort on least significant digit



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



Step 3: BucketSort on the 3rd 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
di

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

Next array is sorted by the first two digits.

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

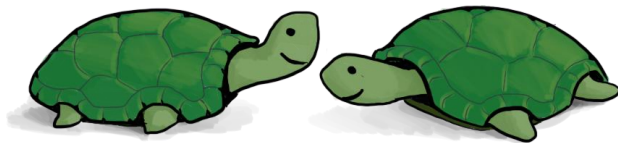
Next array is sorted by all three digits.

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

Sorted array

To prove this is correct...

- What is the inductive hypothesis?



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

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!

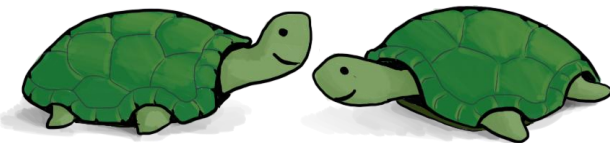
What is the running time?

for RadixSorting
numbers base-10.

- Suppose we are sorting n d -digit numbers (in base 10), $d=3$:
e.g., $n=7$.

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

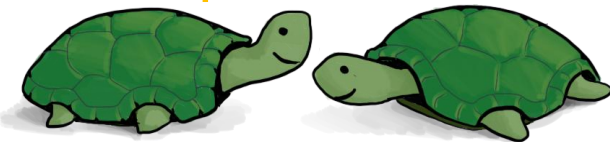
What is the running time?

for RadixSorting
numbers base-10.

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

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

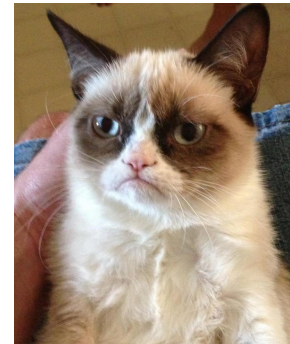
1. How many iterations are there?
 - d iterations
2. How long does each iteration take?
 - Time to initialize 10 buckets, plus time to put n numbers in 10 buckets. $O(n)$.
3. What is the total running time?
 - $O(nd)$



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!



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)
- We will see 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
----	-----	----	-----	----	-----	---

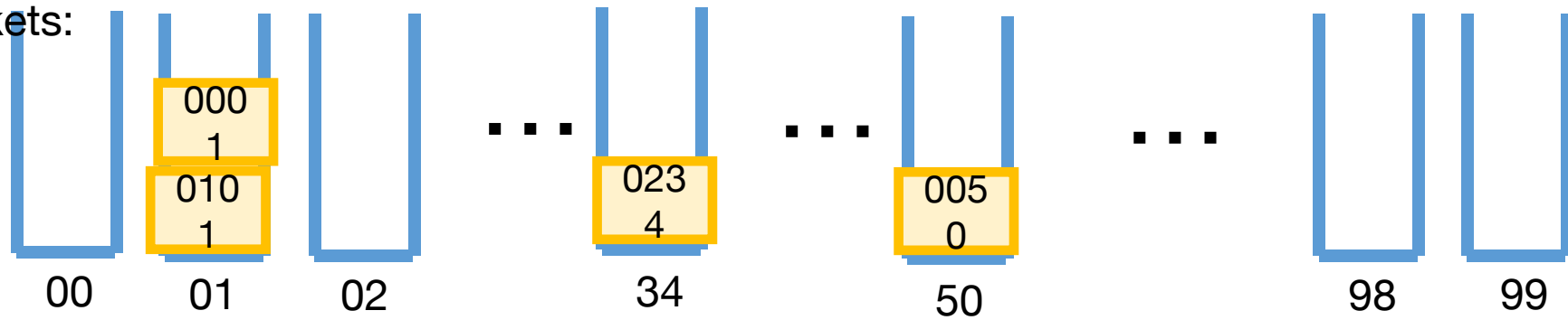
Example: base 100

Original array:

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

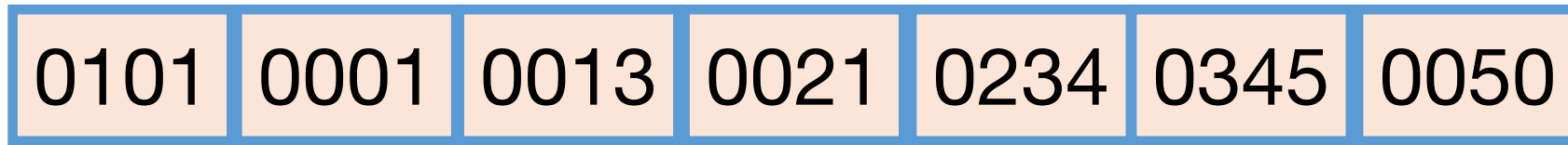
100

buckets:

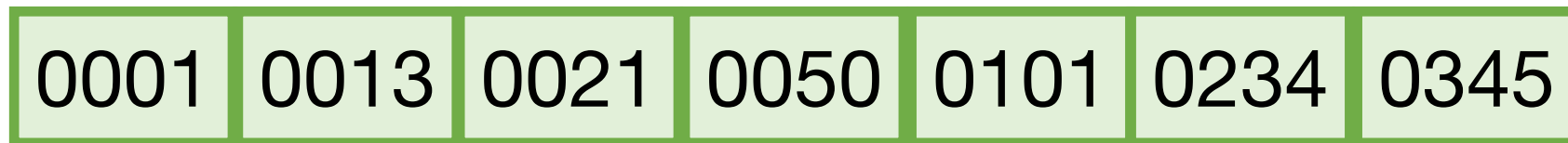
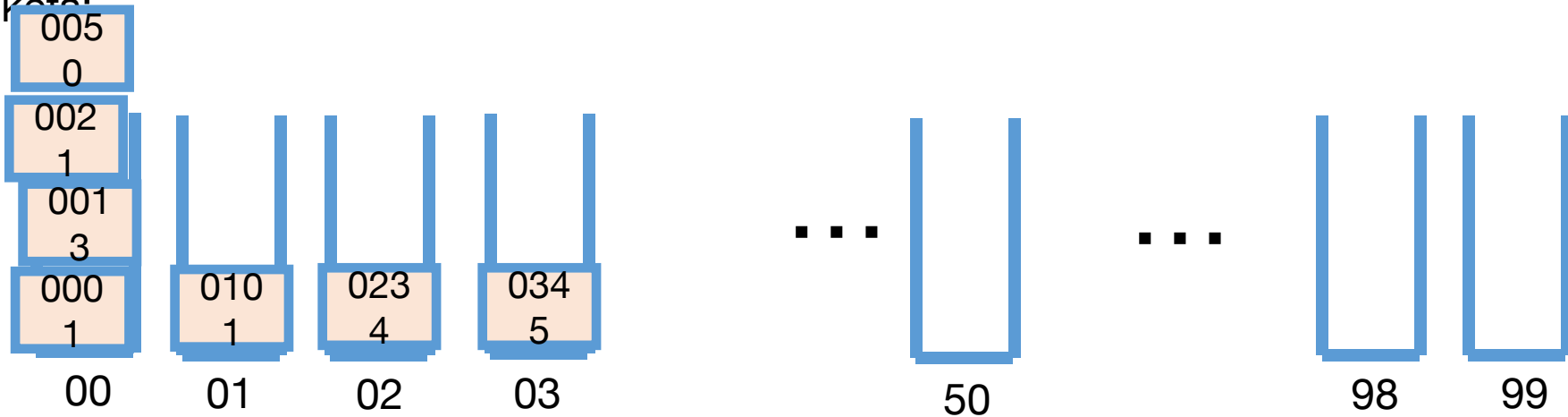


0101	0001	0013	0021	0234	0345	0050
------	------	------	------	------	------	------

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:

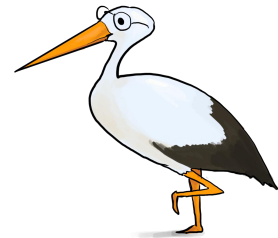
- $d=3$, so 3 iterations.
- 10 buckets

Bigger base means more buckets but fewer iterations.

General running time of RadixSort

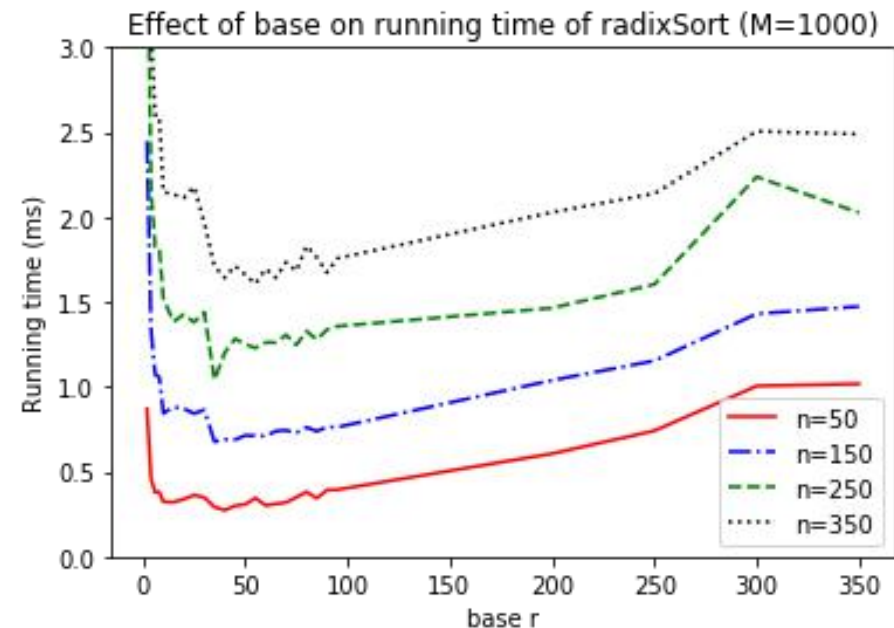
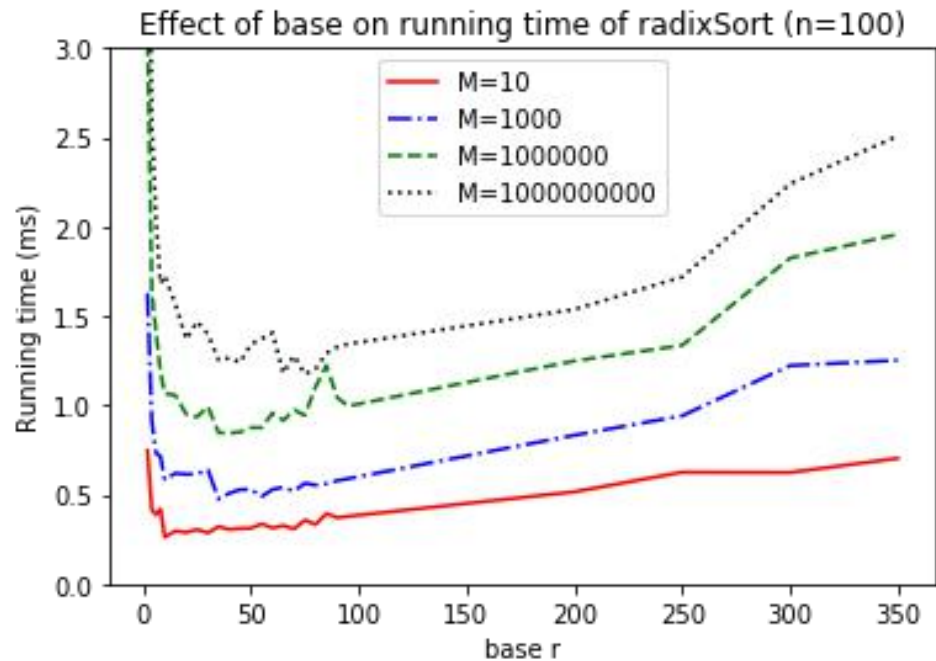
- Say we want to sort:
 - n integers,
 - maximum size M ,
 - in base r .
- Number of iterations of RadixSort:
 - Same as number of digits, base r , of an integer x of max size M .
 - That is $d = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
 - Initialize r buckets, put n items into them
 - $O(n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

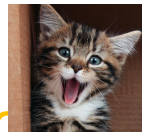
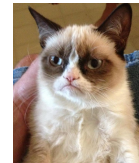
You can put any
constant here
instead of 100.

- 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):
 - BucketSort and RadixSort
 - Both run in time $O(n)$ 😊
 - Might take more space and/or be slower if integers get too big 😞



Thanks~

Questions?