BDMI-01510243

BDMI

Sorting Algorithms

Shiyu Huang

huangsy1314@163.com

Dept. of Comp. Sci. Tech., BNRist Center, Institute for AI, THBI Lab, Tsinghua University

目录:

- 快速排序
- 桶排序

快速排序

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.

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

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

Ollie the over-achieving ostrich

Expected Running time of BogoSort

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

E[running time on a list of length n]

= E[(number of iterations) * (time per iteration)]

= (time per iteration) * E[number of iterations]

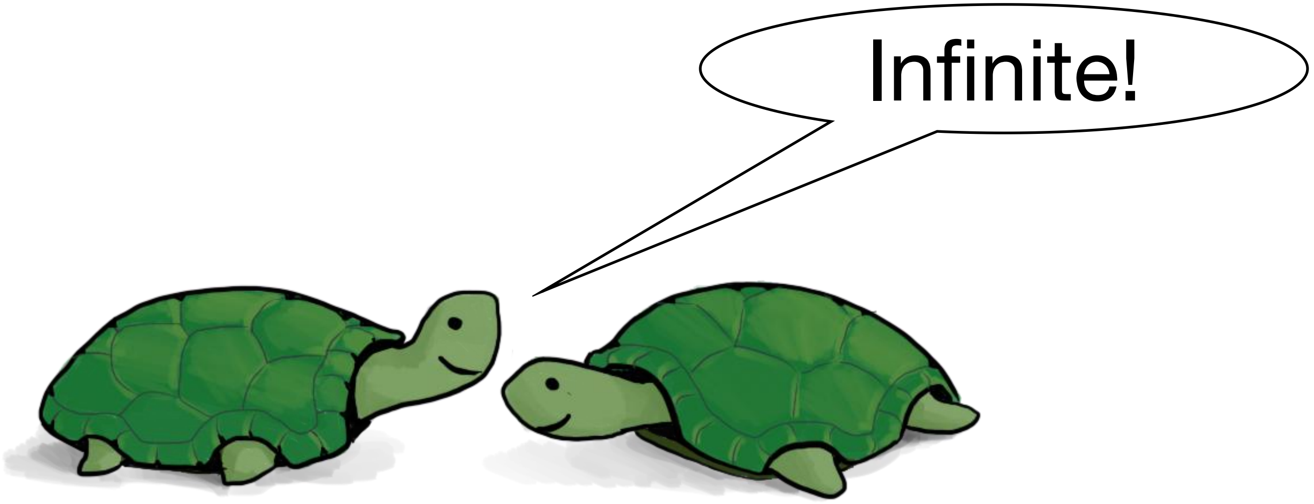
This is O(n) (to permute and then check if sorted)

We just computed this. It's n!.

$= O(n \cdot n!)$

= 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.

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.

7	6	3	5	1	2	4

random pivot!

Next, partition the array into "bigger than 5" or "less than 5"

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

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:

1	2	3	4	5	6	7

PseudoPseudoCode for what we just saw

IPython Lecture 5 notebook for actual code.

- 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)

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?

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

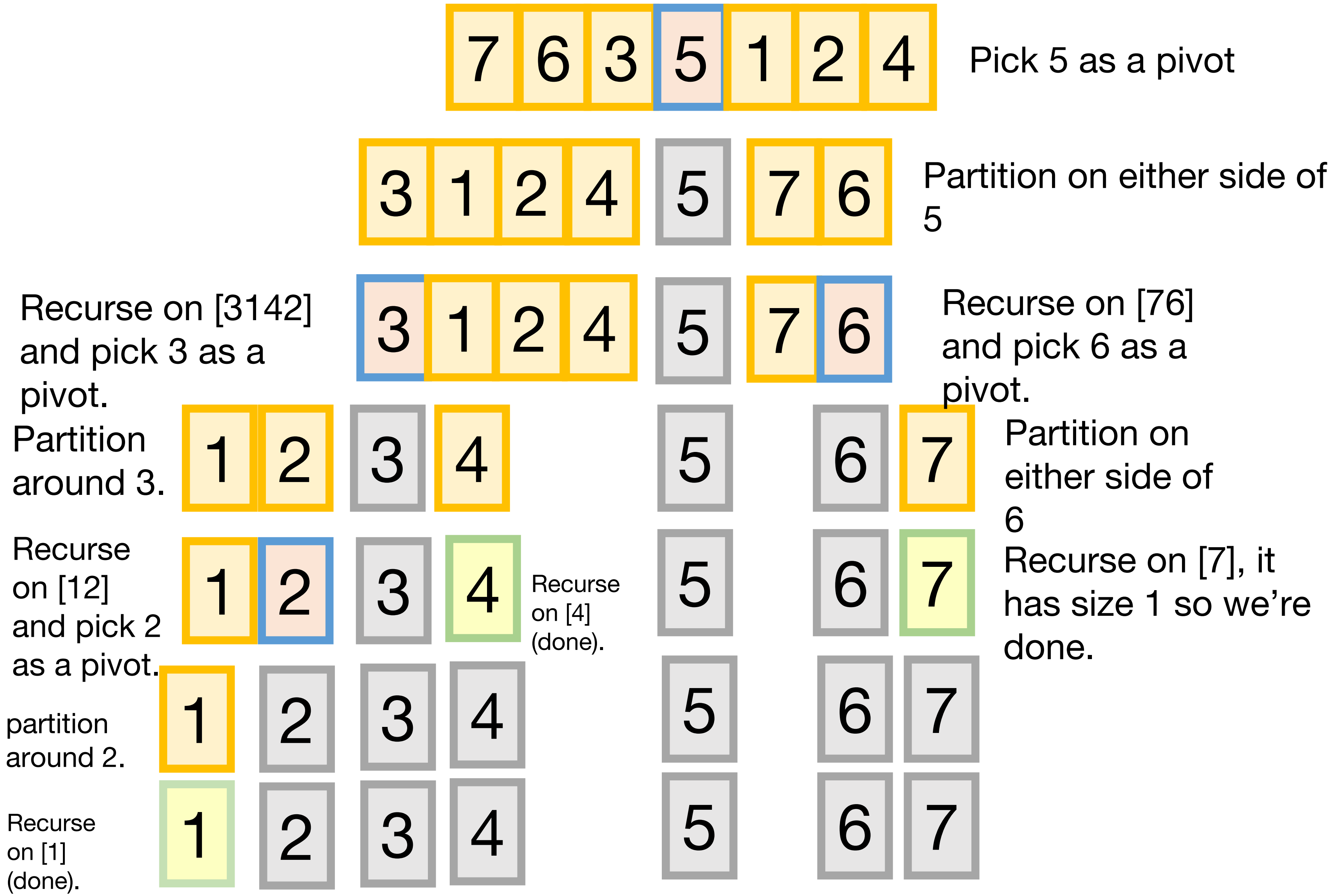
- In an ideal world…
  - if the pivot splits the array exactly in half…

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

- We've seen that a bunch:

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

Example of recursive calls



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.

QuickSort vs MergeSort

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

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	• Worst-case: $O(n^2)$ • Expected: O(n log(n))	Worst-case: O(n log(n))
Used by	• Java for primitive types • C qsort • Unix • g++	• Java for objects • Perl
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

Understand this

These are just for fun. (Not on exam).

Next

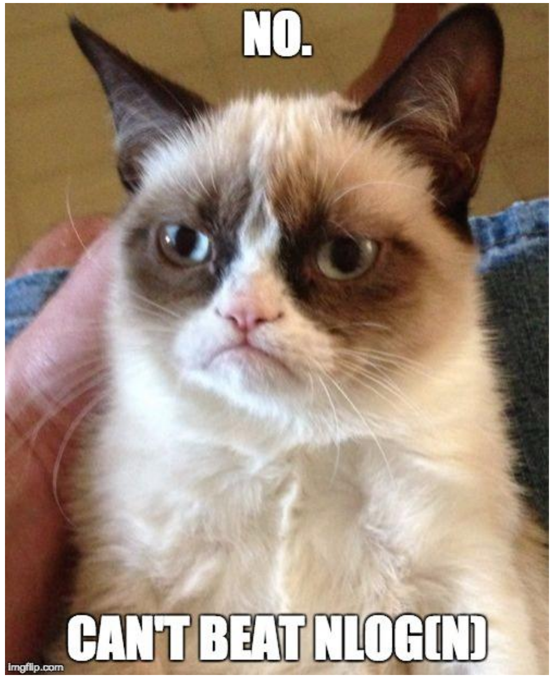
- Can we sort faster than $\Theta(n\log(n))$??

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))

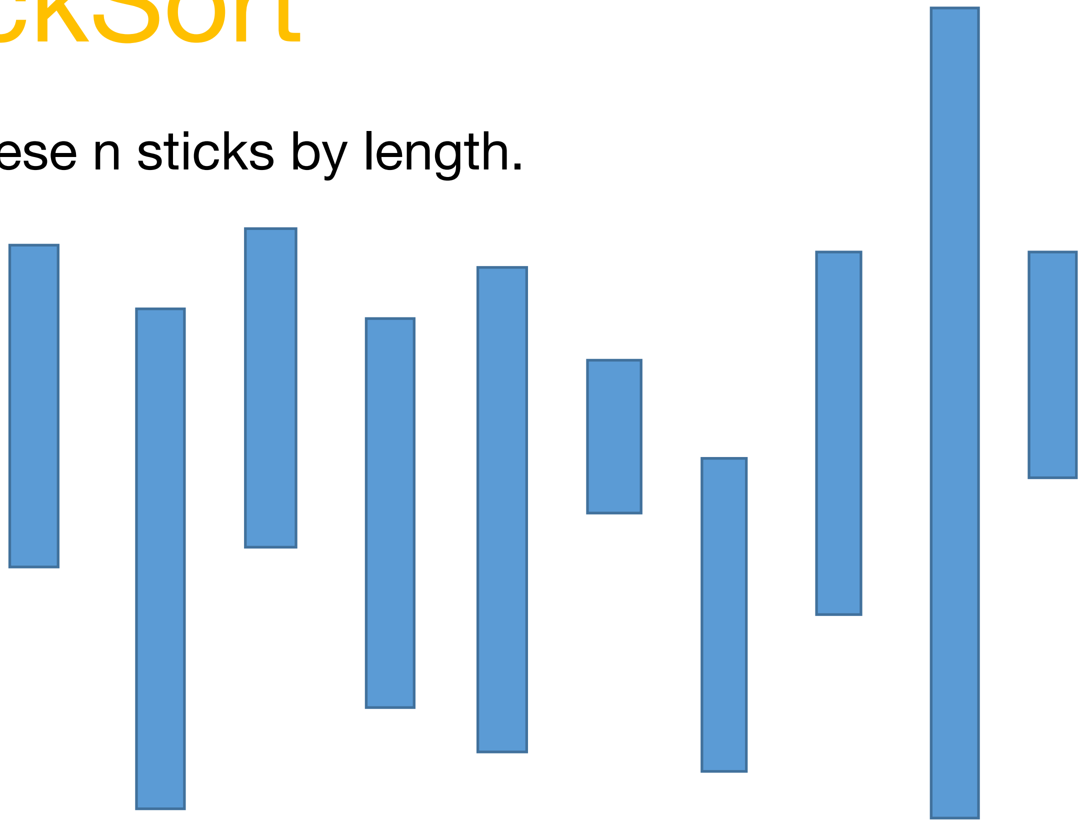
Can we do better?

Depends on
who you ask…



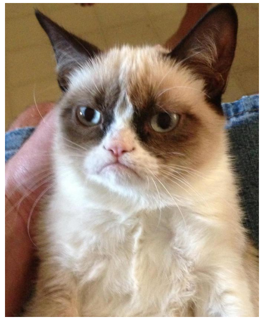
An O(1)-time algorithm for sorting: StickSort

- Problem: sort these n sticks by length.



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

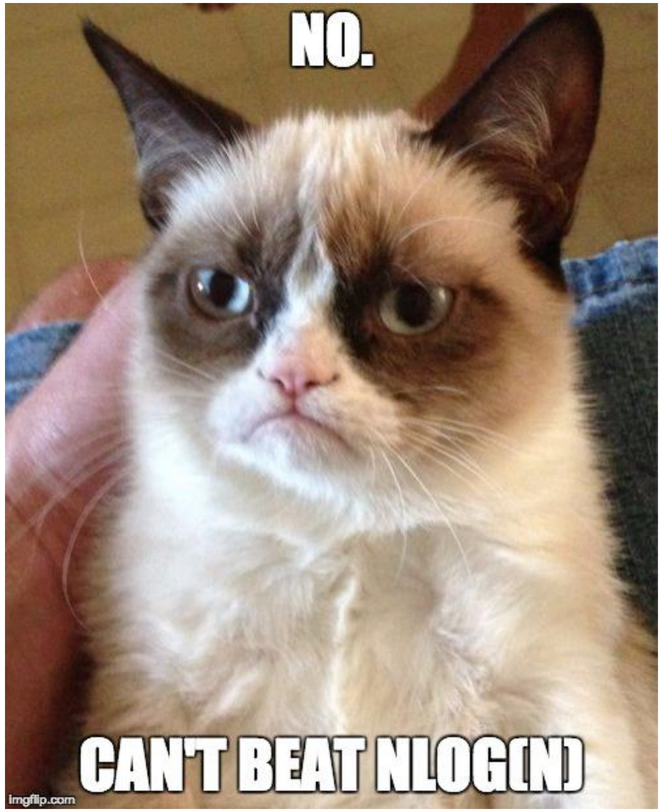
Today: two (more) models



- 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

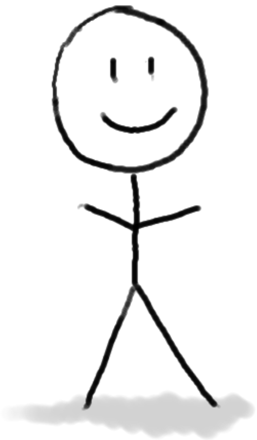
 is shorthand for

"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



Algorithm

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

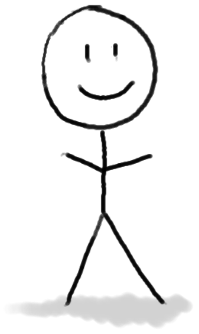
There is a genie who knows what the right order is.

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: 7 6 3 5 1 4 2

Pivo
t!



Is 7 bigger tha 5 ? YES

Is 6 bigger tha 5 ? YES

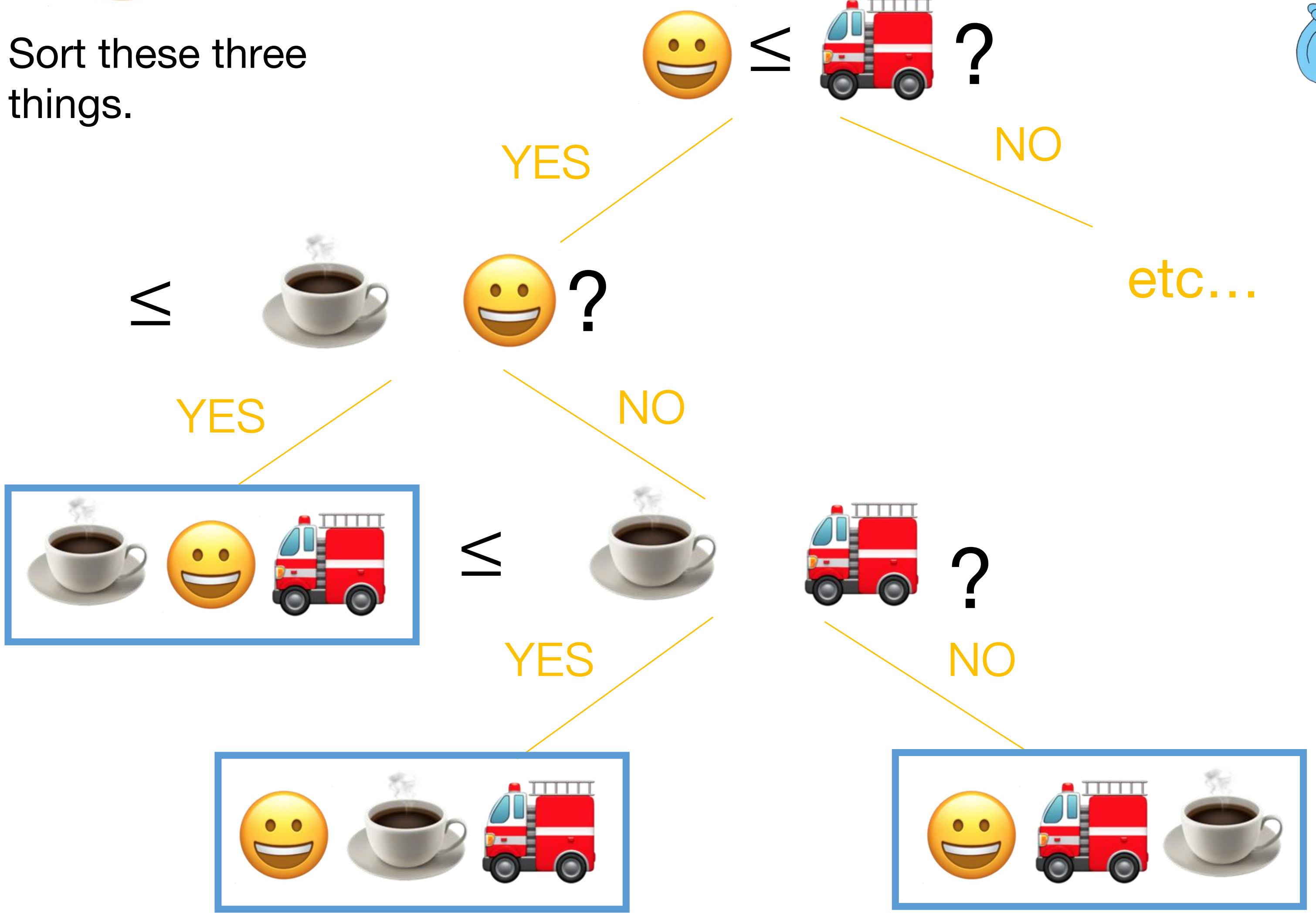
Is 3 bigger tha 5 ? NO

5

etc.

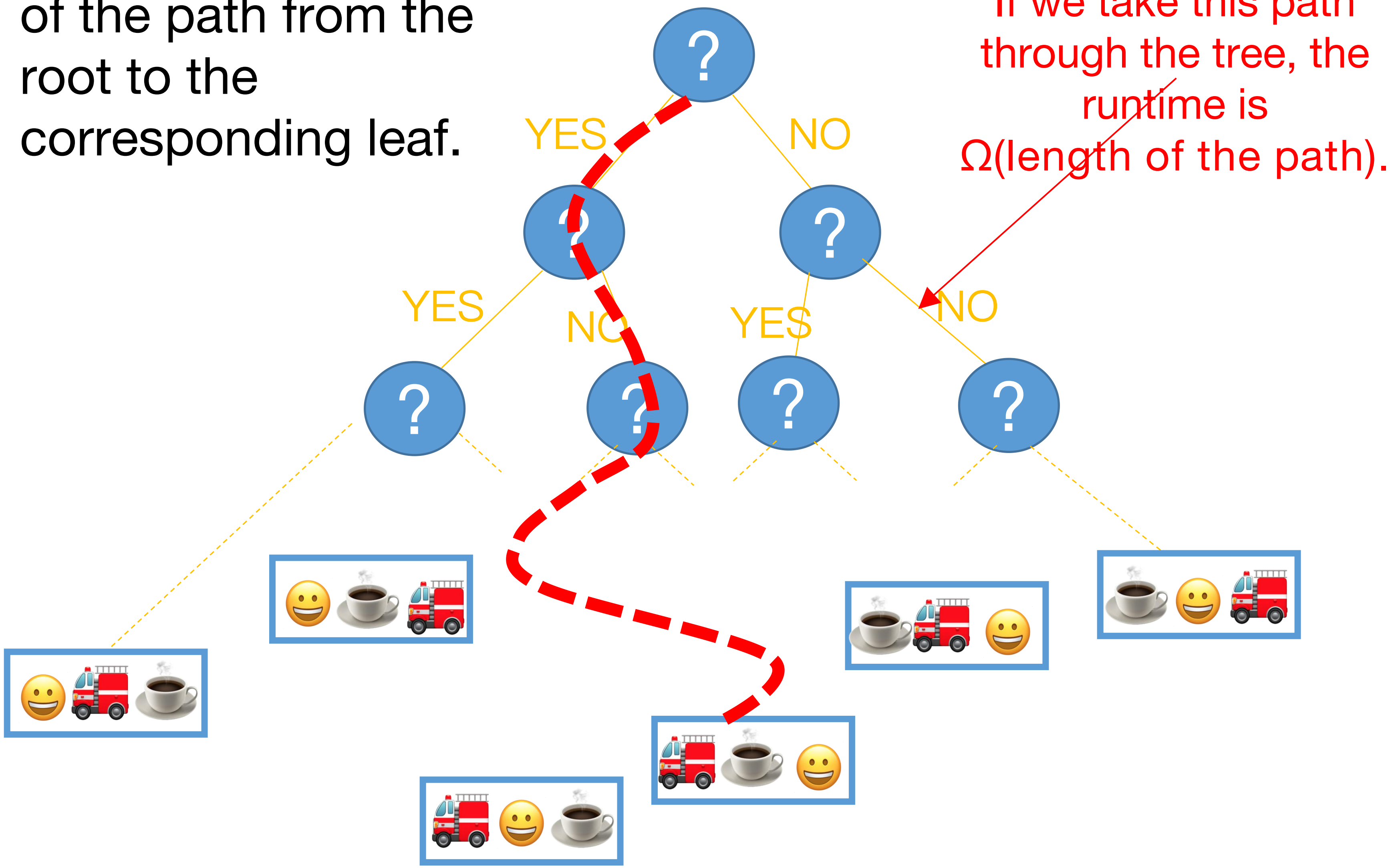
Decision trees

Sort these three things.

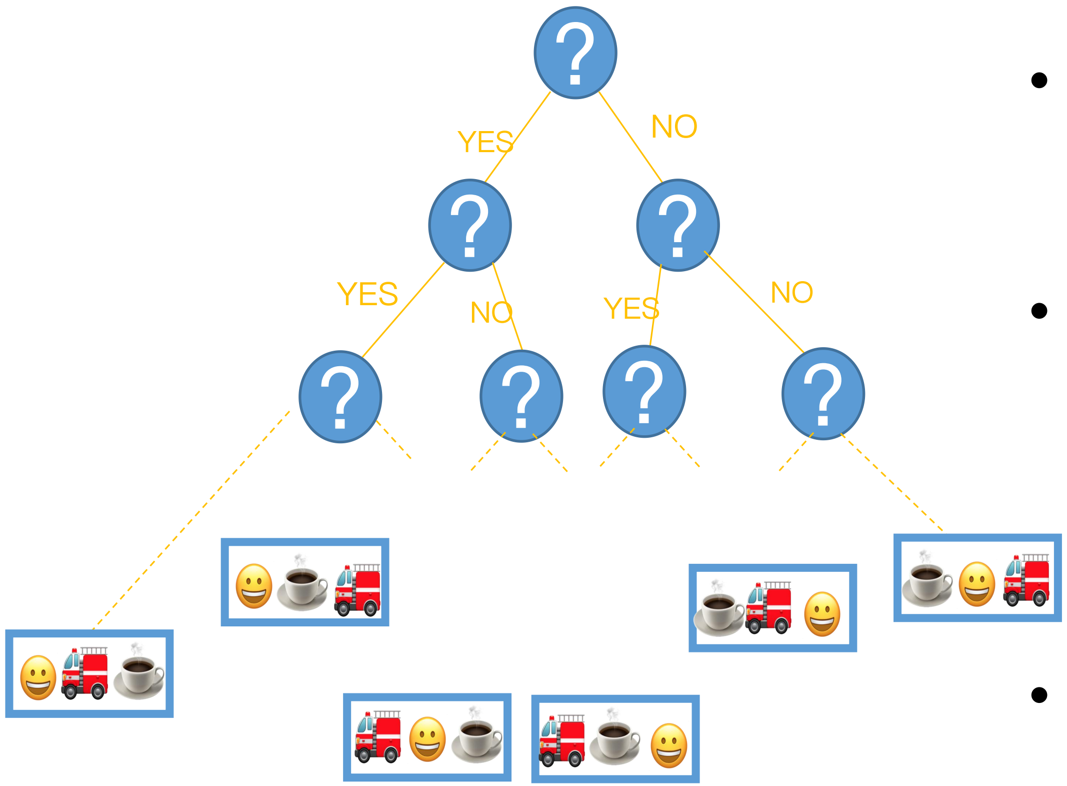


Q: What's the runtime on a particular input?

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



How long is the longest path?



- 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!).

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

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

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!

Beyond comparison-based sorting algorithms

桶排序

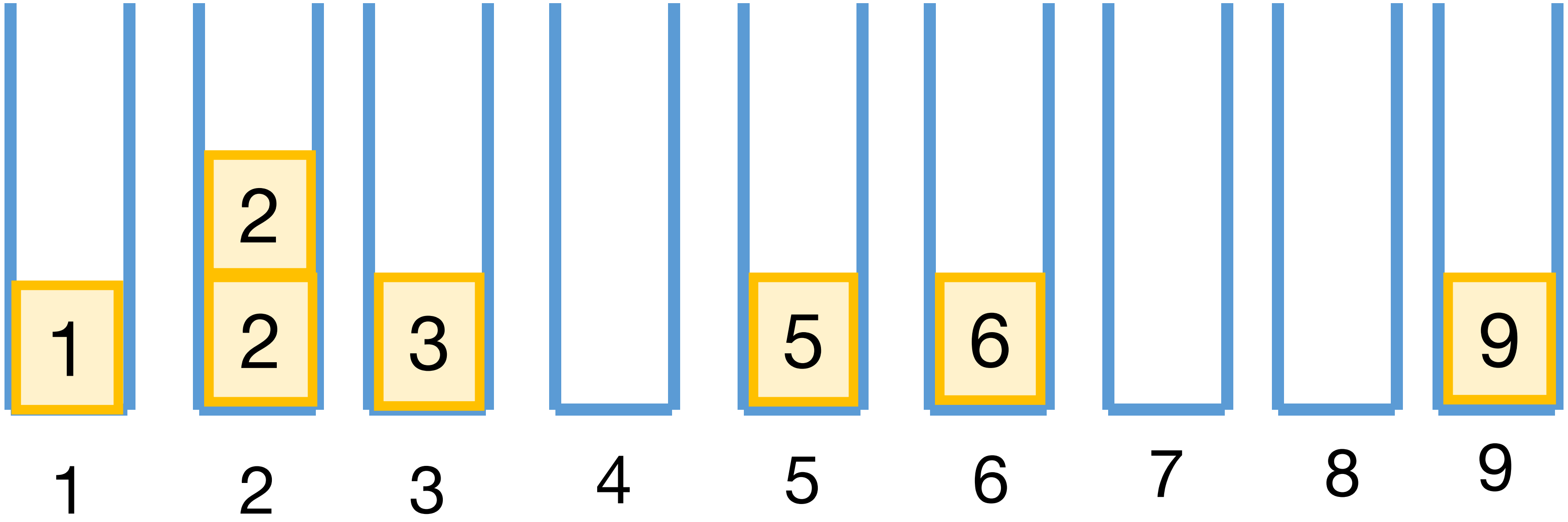
Why might this help?

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

BucketSort:

9	6	3	5	2	1	2

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



Concatenate the buckets!

SORTED!

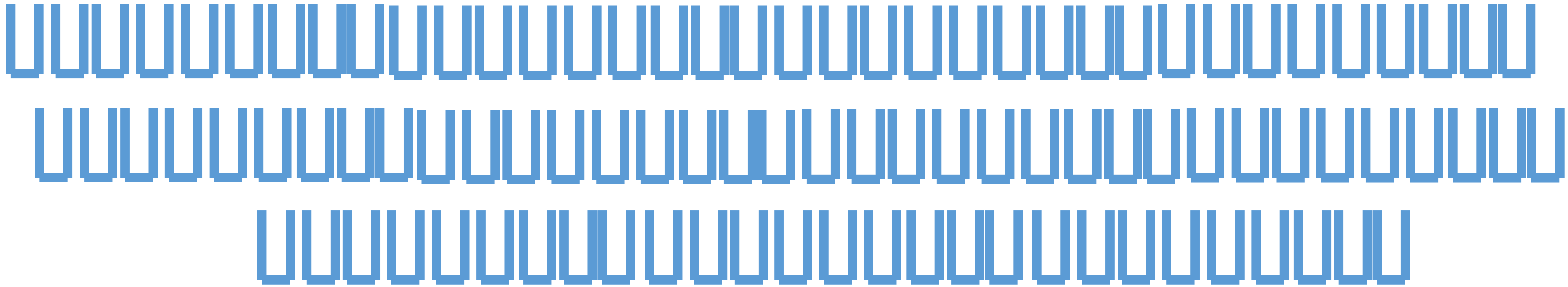
In time O(n).

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	12345	13	2^{1000}	50	100000000	1

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

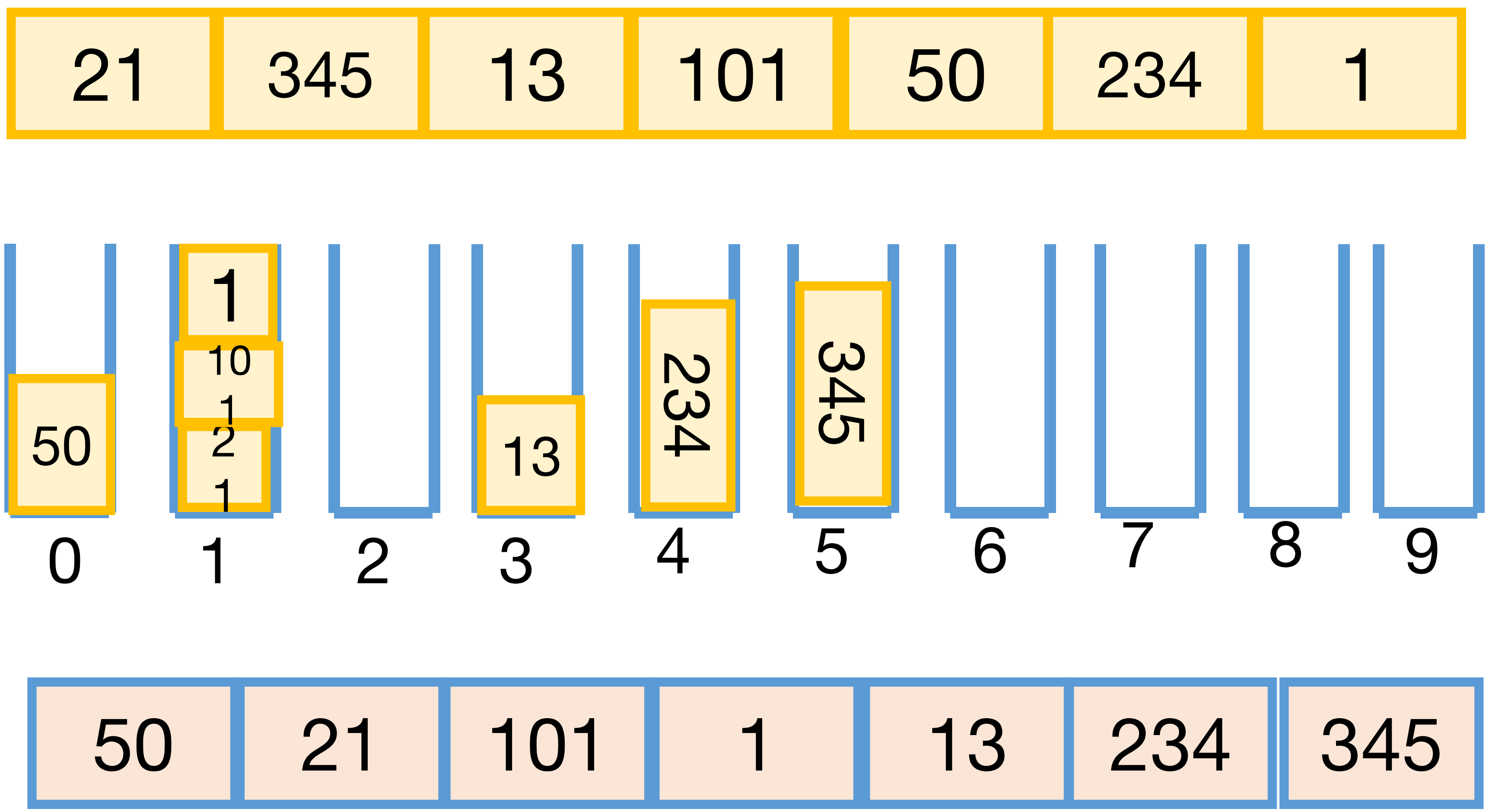


RadixSort

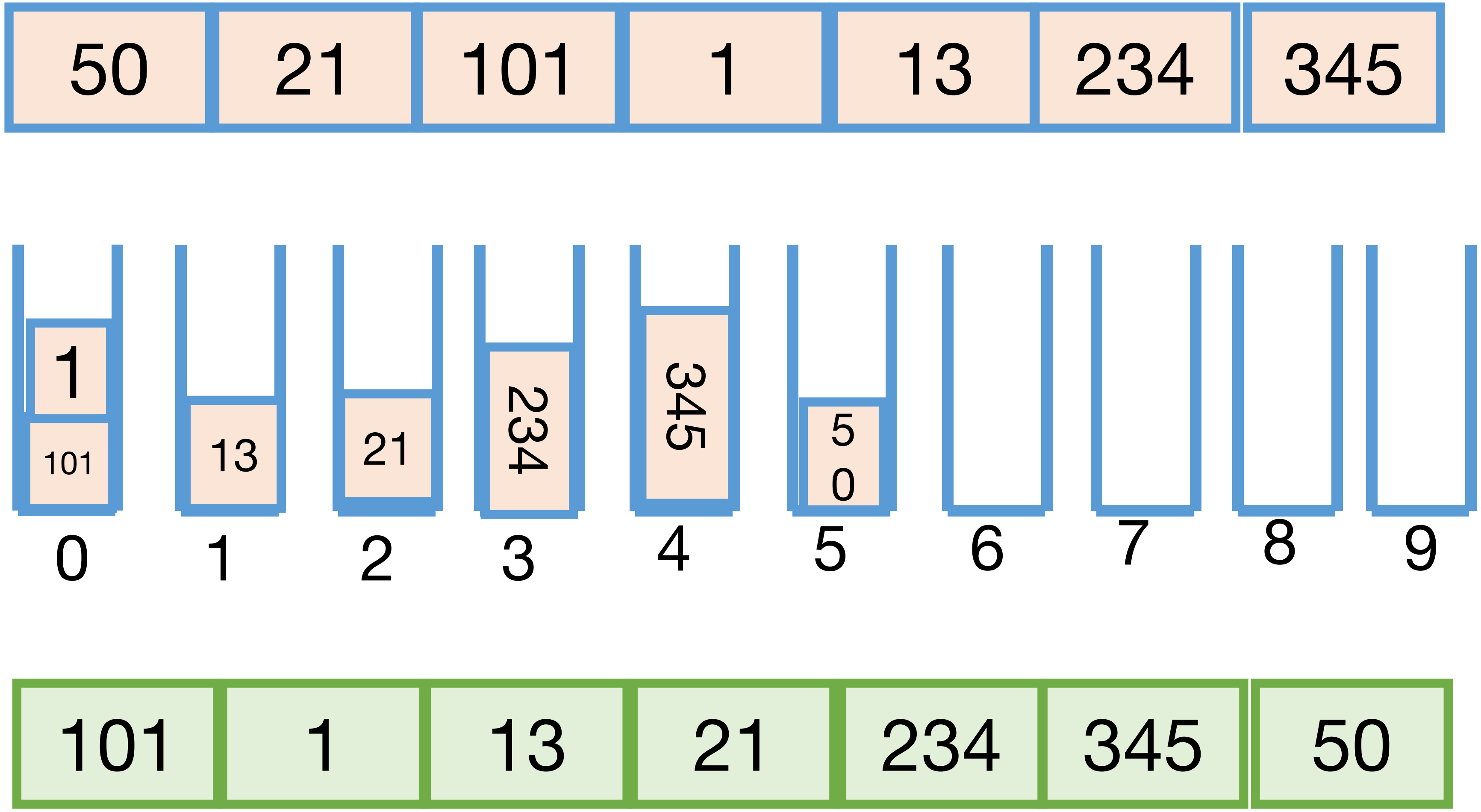
- For sorting integers up to size M
  - or more generally for lexicographically sorting strings
- Can use less space than BucketSort

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

Step 1: BucketSort on least significant digit

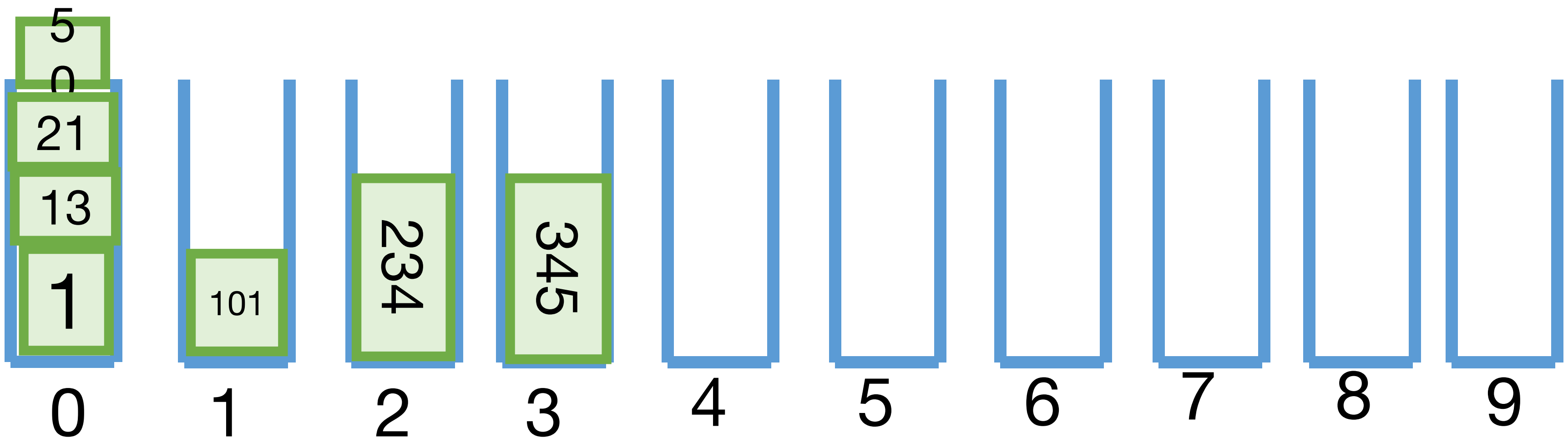


Step 2: BucketSort on the 2nd least sig. digit



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

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
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

What is the running time?

for RadixSorting numbers base-10.

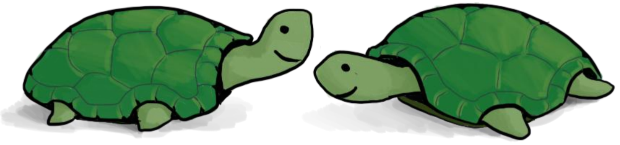
- Suppose we are sorting n d-digit numbers (in base 10).

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

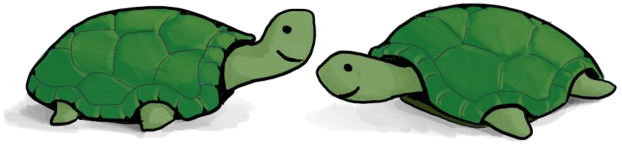
What is the running time?

for RadixSorting numbers base-10.

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

021	345	013	101	050	234	001

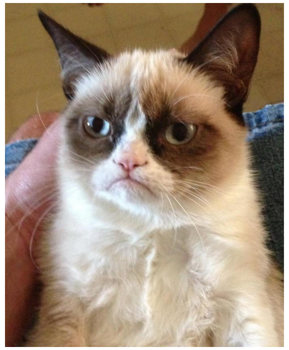
1. How many iterations are there?
   - d iterations
2. How long does each iteration take?
   - Time to initialize 10 buckets, plus time to put n numbers in 10 buckets. O(n).
3. What is the total running time?
   - O(nd)



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!



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

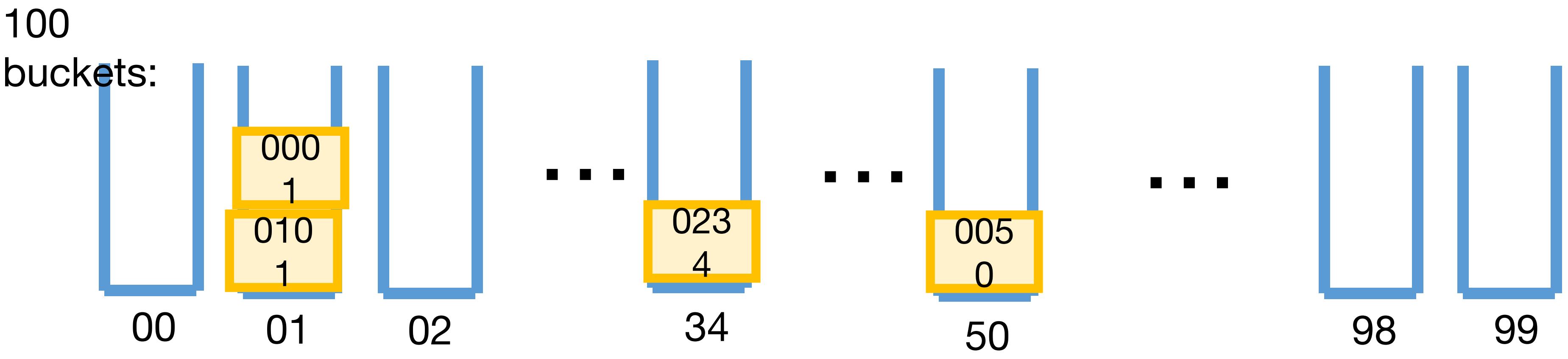
Original array:

21	345	13	101	50	234	1

Example: base 100

Original array:

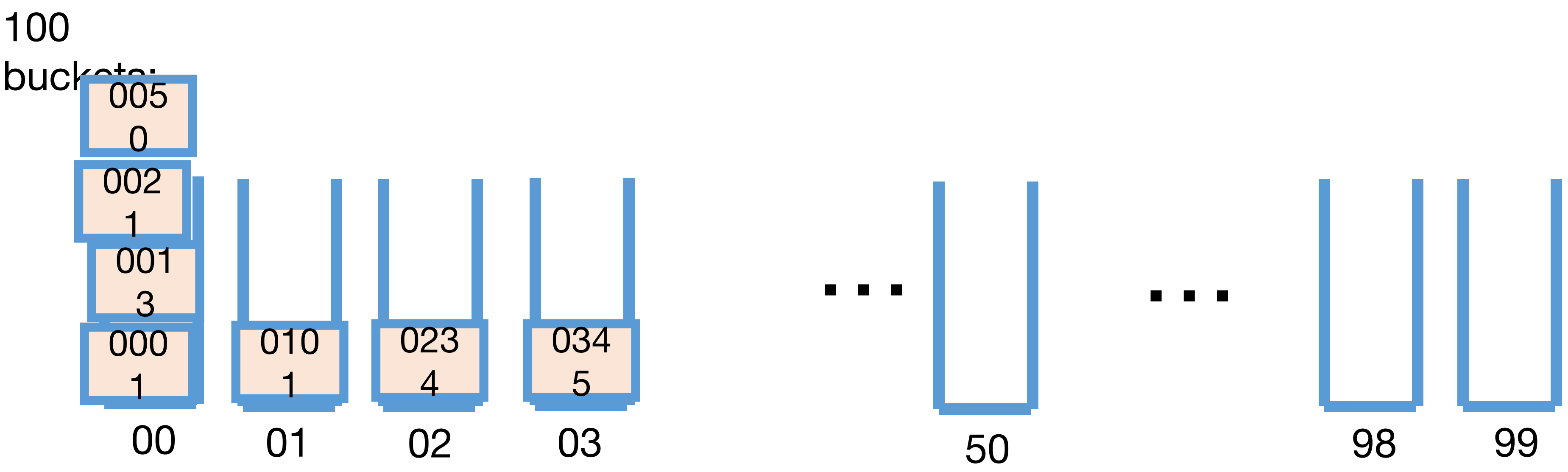
0021	0345	0013	0101	0050	0234	0001



0101	0001	0013	0021	0234	0345	0050

Example: base 100

0101	0001	0013	0021	0234	0345	0050



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.

vs.

Base 10:

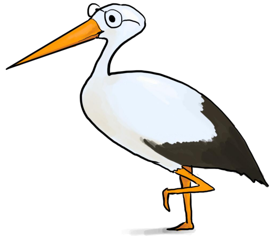
- d=3, so 3 iterations.
- 10 buckets

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.



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

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

- 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?

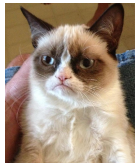
You can put any constant here instead of 100.

- 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.

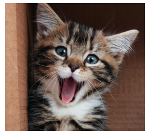
Recap

- 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?