Linear Time Sorting Algorithms

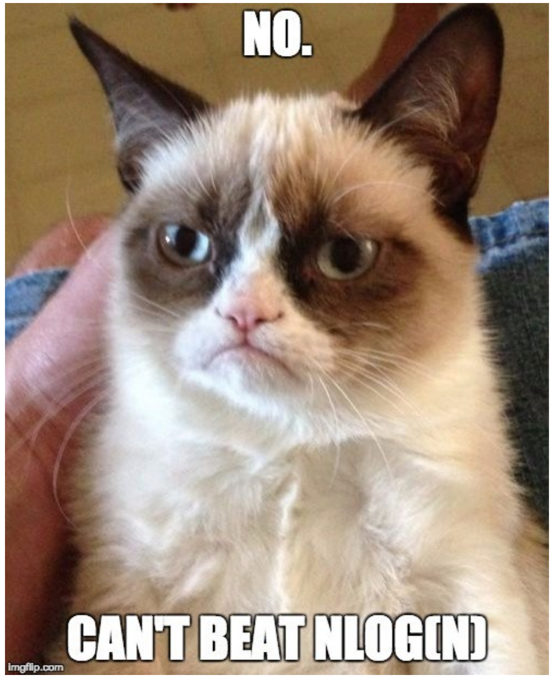
Sorting lower bounds and O(n)-time sorting

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))

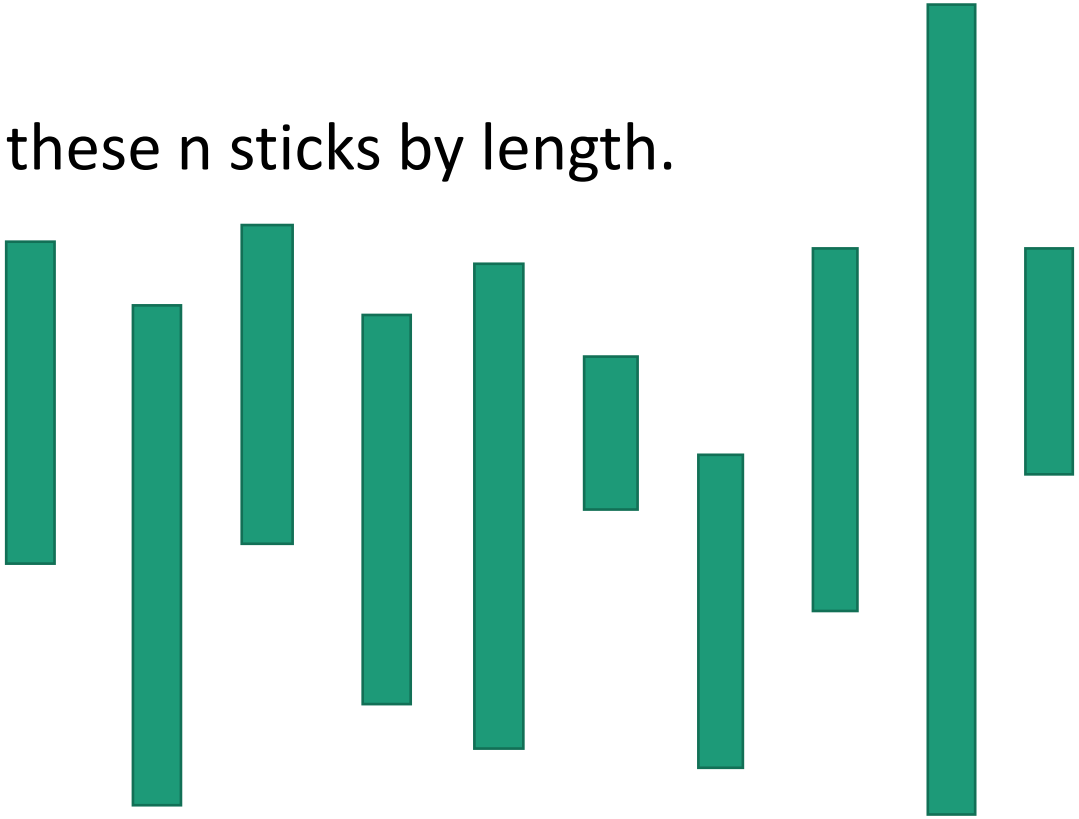
Can we do better?

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

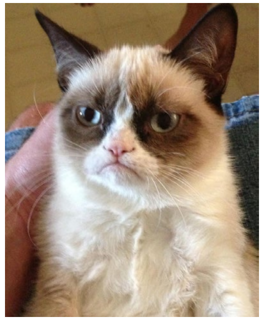
- But StickSort does raise some important questions:
  - What is our model of computation?
    - Input: array
    - Output: sorted array
    - Operations allowed: comparisons

    -vs-

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

  - What are reasonable models of computation?

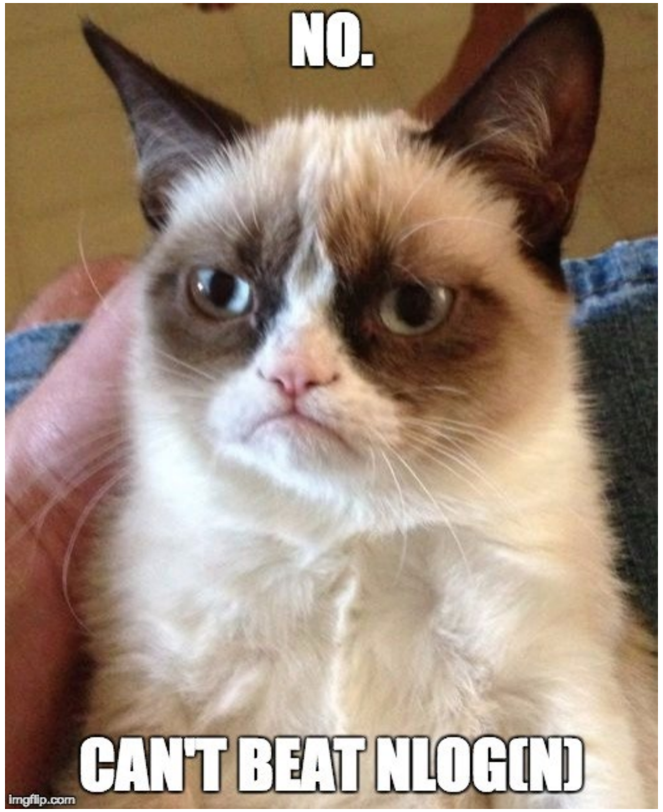
Today: two (more) models



- 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

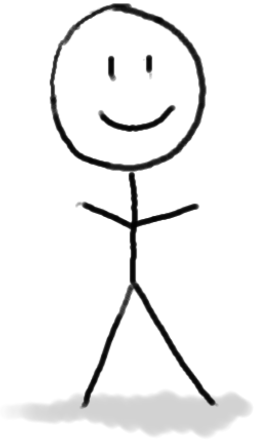
- You want to sort an array of items.
- You can't access the items' values directly: you can only compare two items and find out which is bigger or smaller.

Comparison-based sorting algorithms

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

 is shorthand for
"the first thing in the input list"

Is  bigger than  ?



Algorithm

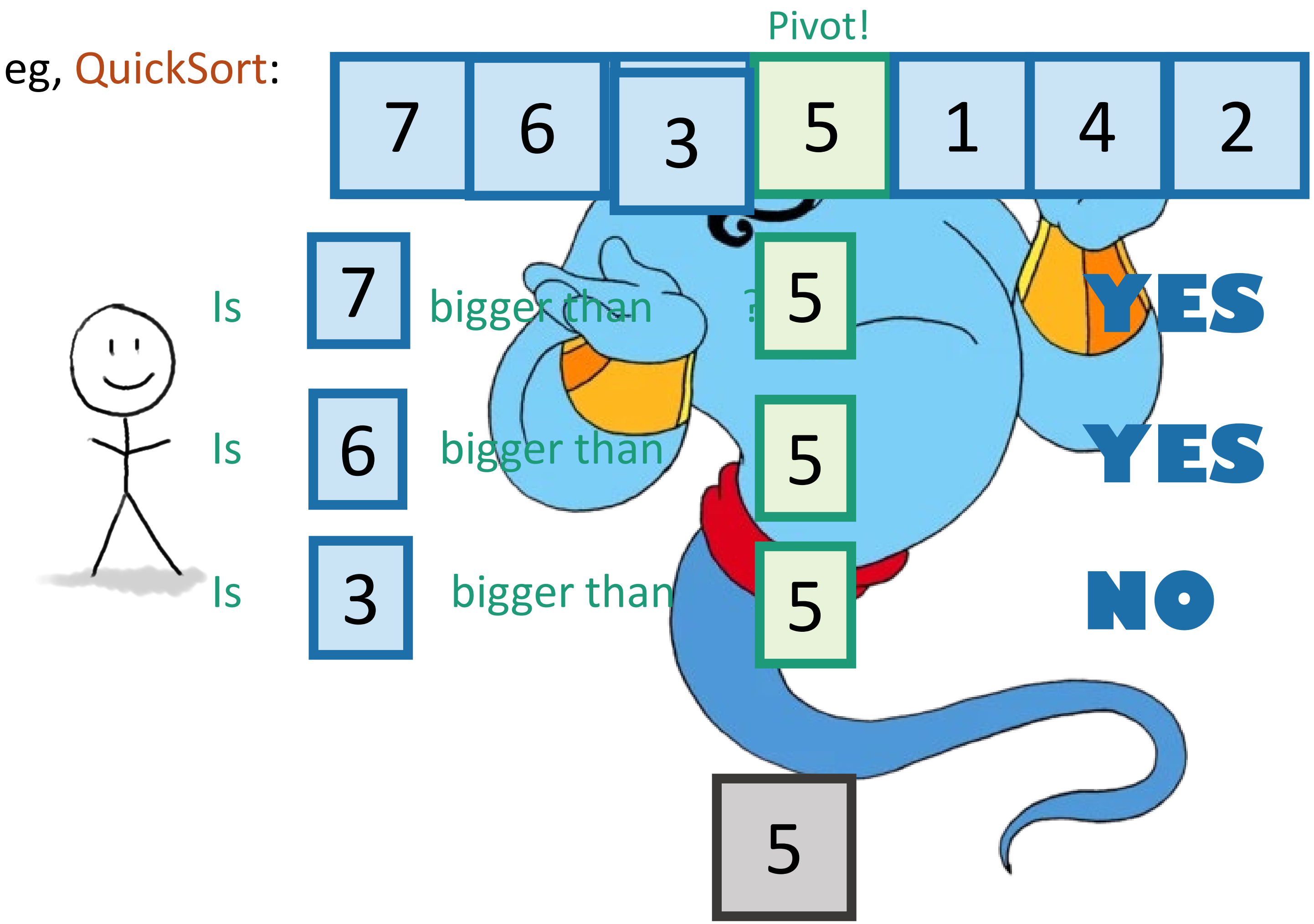
YES

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

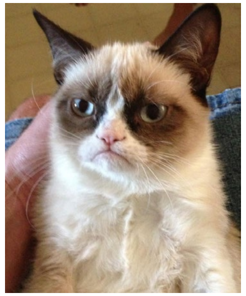
The algorithm's job is to output a correctly sorted list of all the objects.

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.



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.

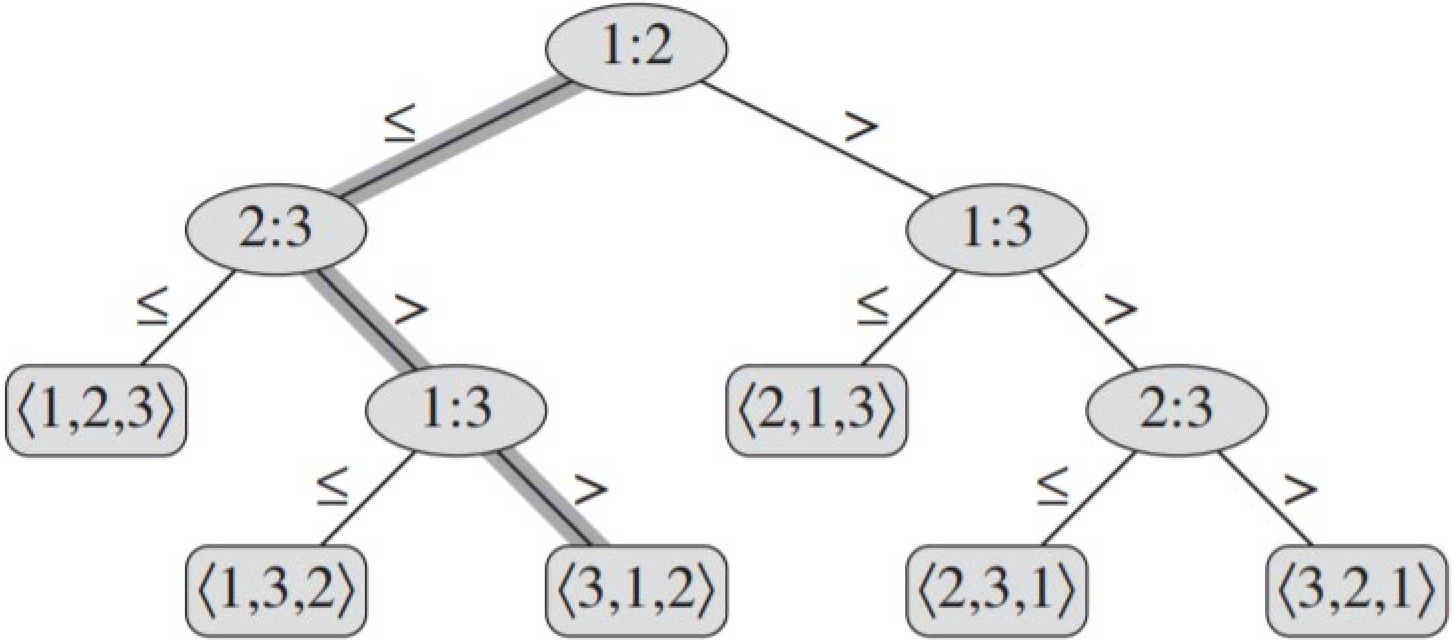
This covers all the sorting algorithms we know!!!

- How might we prove this?

1. Consider all comparison-based algorithms, one-by-one, and analyze them.

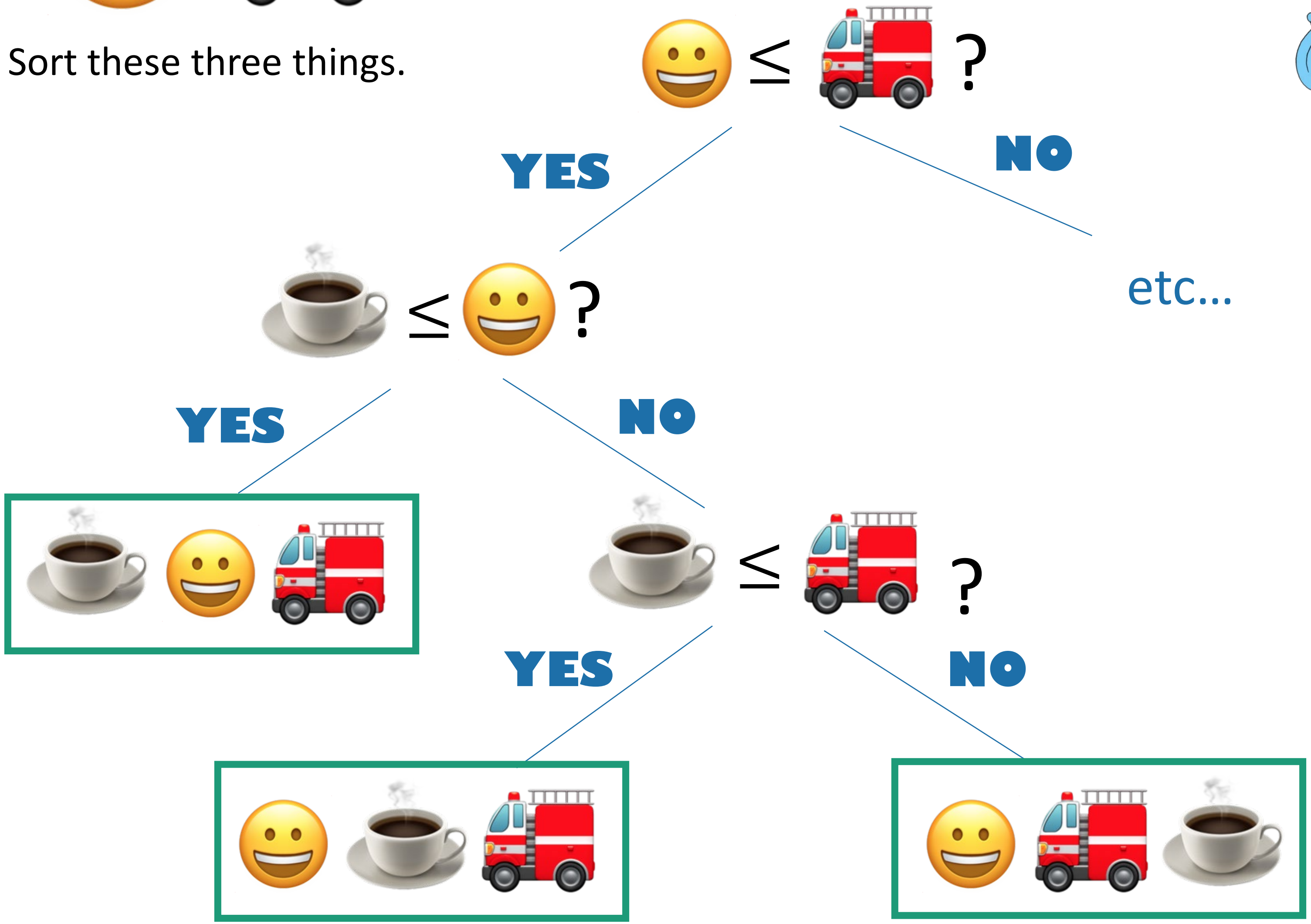
2. Don't do that.

Instead, argue that all comparison-based sorting algorithms give rise to a **decision tree**.
Then analyze decision trees.


1:2
≤
>
2:3
1:3
≤
>
≤
>
⟨1,2,3⟩
1:3
⟨2,1,3⟩
2:3
≤
>
≤
>
⟨1,3,2⟩
⟨3,1,2⟩
⟨2,3,1⟩
⟨3,2,1⟩

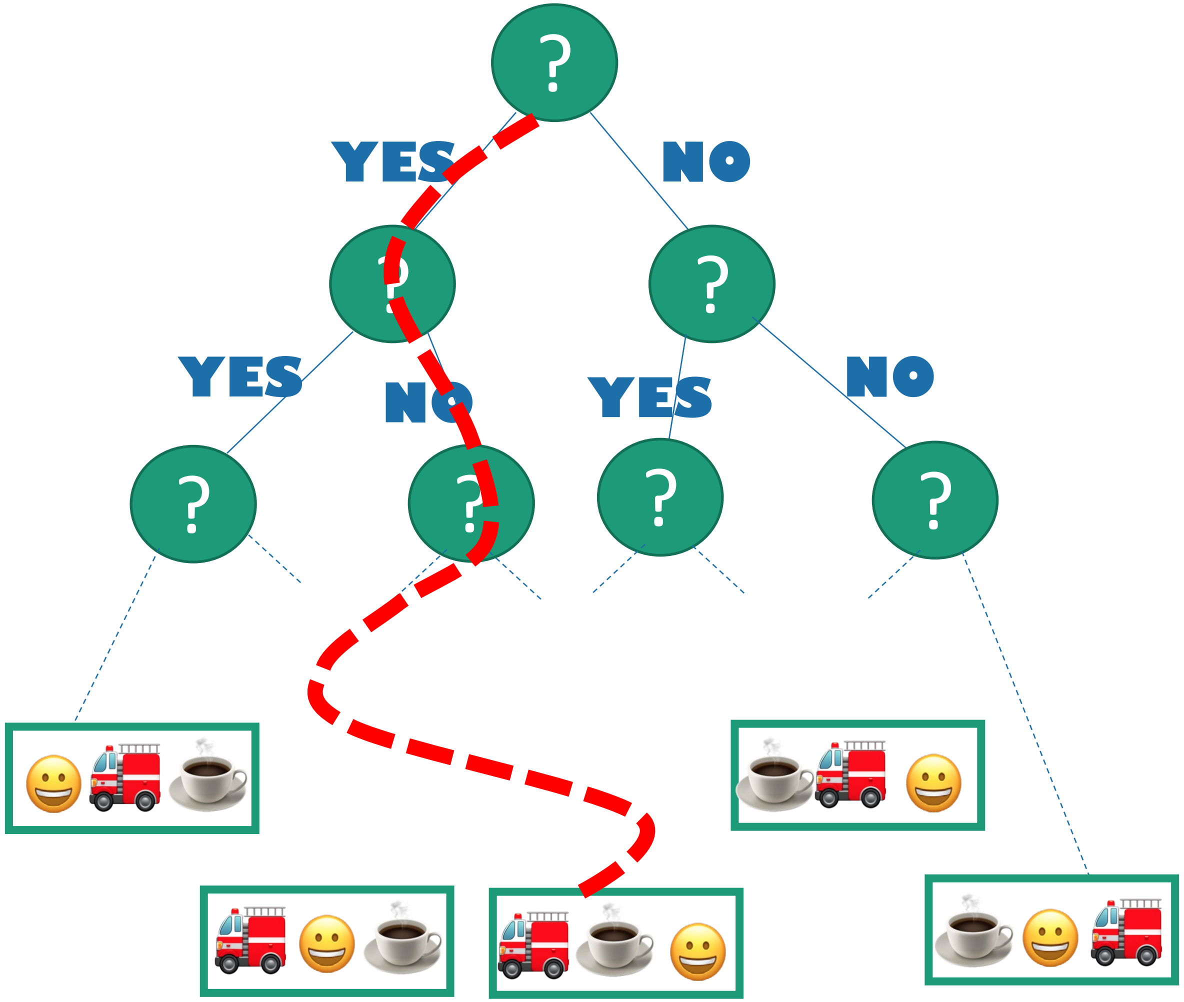
Decision trees

Sort these three things.

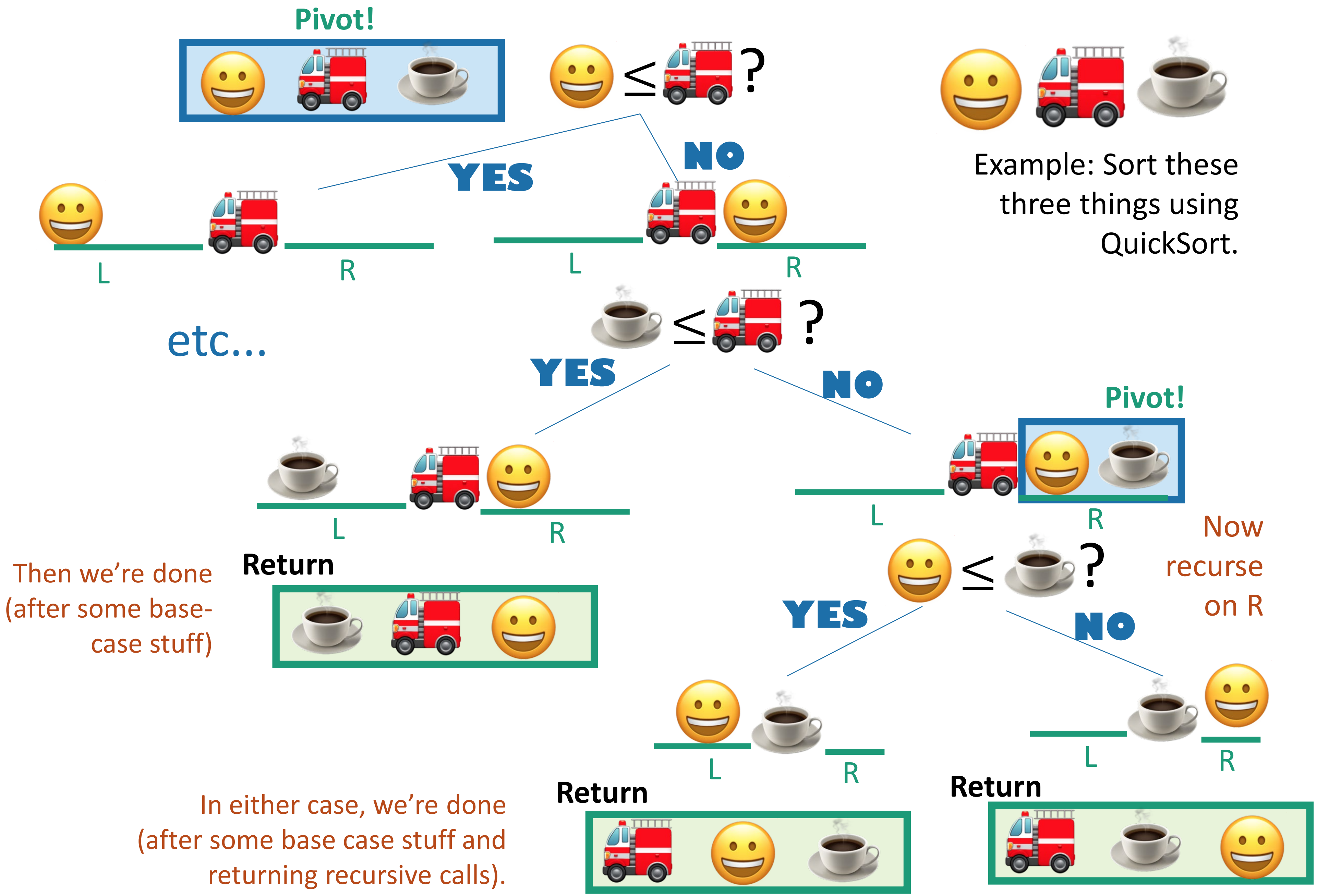


Decision trees

- 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.

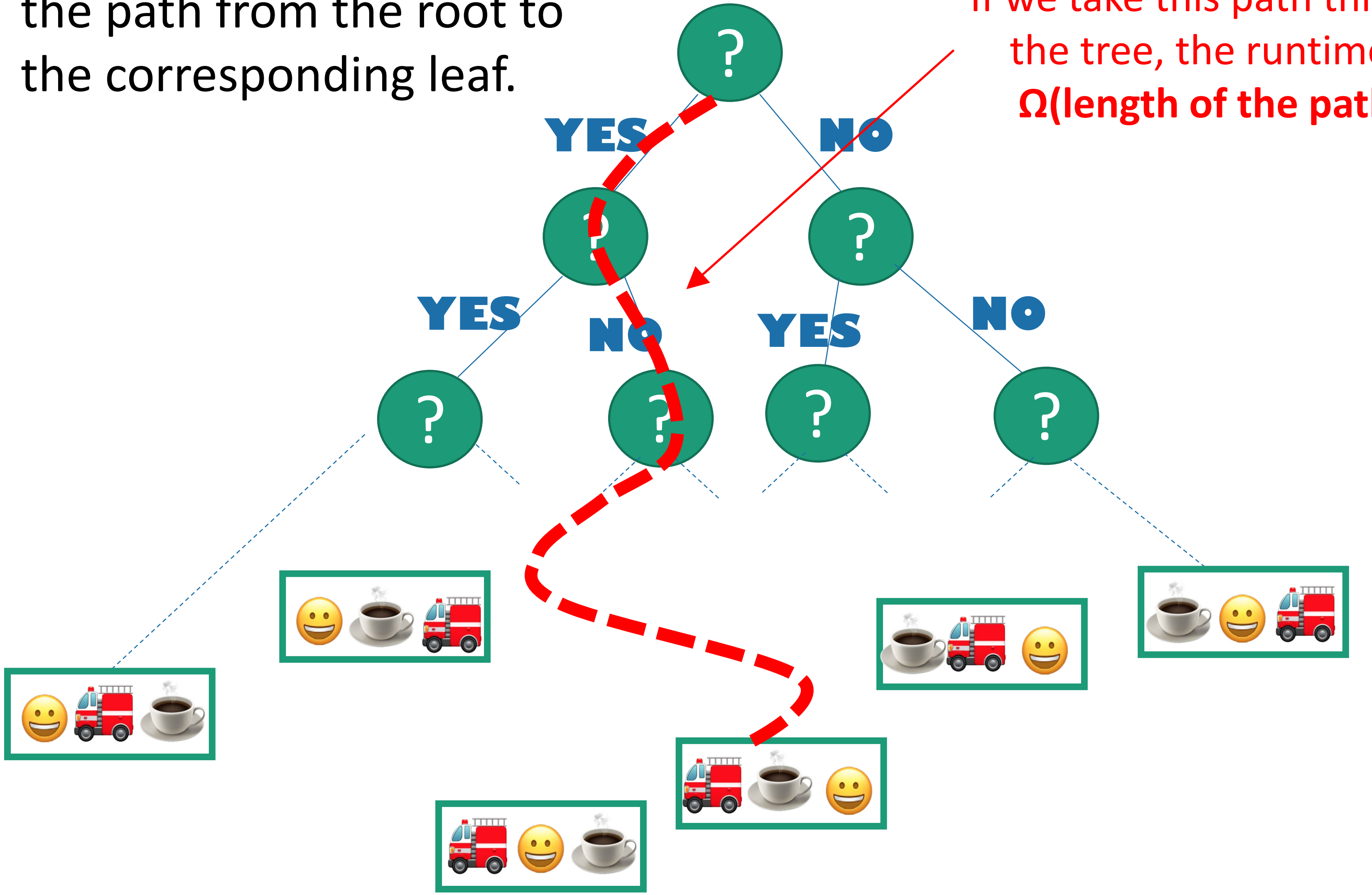


Comparison-based algorithms look like decision trees.



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

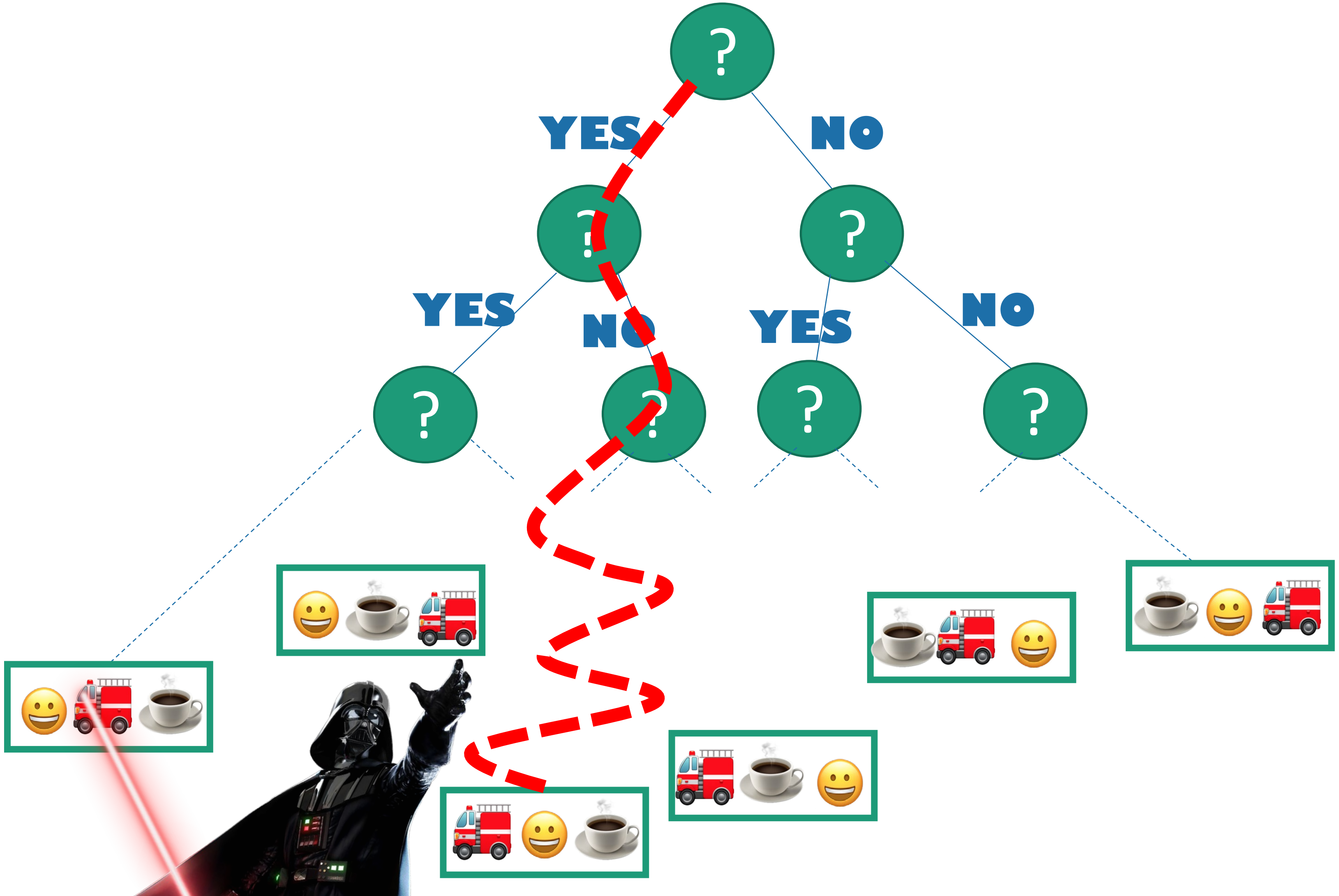
A: At least the length of the path from the root to the corresponding leaf.



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

Q: What's the worst-case runtime?

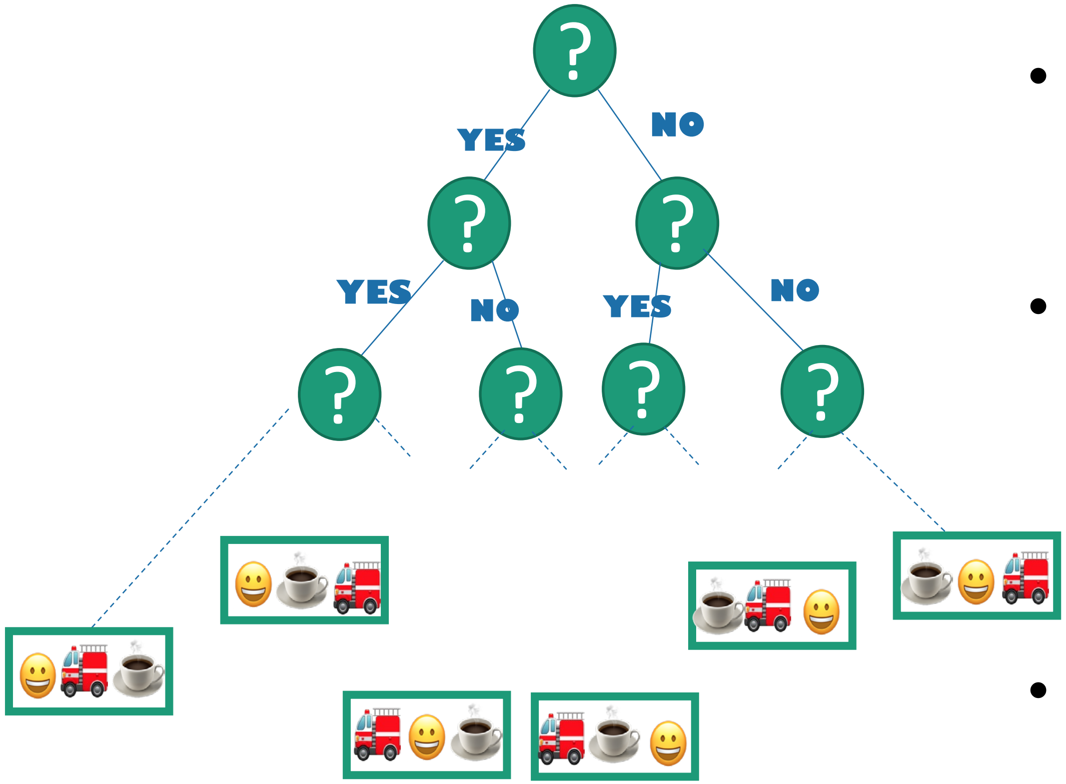
A: At least Ω(length of the longest path).



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 _____



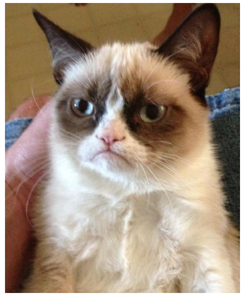
- This is a binary tree with at least __n!__ leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth _log(n!)_.
- So in all such trees, the longest path is at least log(n!).

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

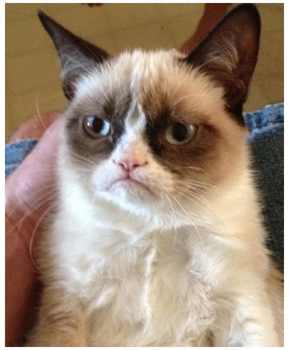
Lower bound of $\Omega(n \log(n))$.



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

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.

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)?

Counting Sort

Sorting in Linear Time

Beyond comparison-based sorting algorithms

Counting Sort

Suppose, we want to sort this array.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 1: Find out the maximum element from the given array.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

Although, we are doing comparisons to find the max element. So, the joke is on them

Counting Sort: Working

Step 2: Initialize another array of length max+1 with all elements as 0. This array will be used for storing the count of the elements in the array.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

1

Auxiliary array/data structure is a fancy way of saying helper array/data structure

Counting Sort: Working

Step 3: Store the count of each element at their respective index in the auxiliary count array

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8

Auxiliary[1] array/data structure is a fancy way of saying helper array/data structure

Counting Sort: Working

Step 3: Store the count of each element at their respective index in the auxiliary count array

For example: the count of element 1 is 3 therefore, 3 is stored on the 1st index of count array.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	3	2	1	0	2	0	1	1
	0	1	2	3	4	5	6	7	8

1

Auxiliary array/data structure is a fancy way of saying helper array/data structure

Counting Sort: Working

Step 4: Store the cumulative sum of the elements of the count array.

$$count[i] = \sum count[x]; \quad 0 <= x <= i$$

count

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

0	3	2	1	0	2	0	1	1
0	1	2	3	4	5	6	7	8

Counting Sort: Working

Step 4: Store the cumulative sum of the elements of the count array.

count[i] = ∑ count[x];   0 <= x <= i

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

+

count	0	3	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

Counting Sort: Working

- Step 5: Declare another Output Array.

1	3	2	8	5	1	5	1	2	7

- 0 1 2 3 4 5 6 7 8 9

- count

0	3	5	6	6	8	8	9	10

- 0 1 2 3 4 5 6 7 8

	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 6: Find the index of each element of the original array in the count array. Place the element at the index in output array by subtracting -1

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

- count

0	3	5	6	6	8	8	9	10
0	1	2	3	4	5	6	7	8

	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 6: Find the index of each element of the original array in the count array. Place the element at the index in output array by subtracting -1

1	3	2	8	5	1	5	1	2	7

- 0 1 2 3 4 5 6 7 8 9

- count

0	3	5	6	6	8	8	9	10

- 0 1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8 9

Counting Sort: Working

Step 6: Find the index of each element of the original array in the count array. Place the element at the index in output array by subtracting -1

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

- count

0	3	5	6	6	8	8	9	10
0	1	2	3	4	5	6	7	8

	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 6: Find the index of each element of the original array in the count array. Place the element at the index in output array by subtracting -1

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	3	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8
		3-1							

	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 6: Find the index of each element of the original array in the count array. Place the element at the index in output array by subtracting -1

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	3	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

3-1

		1							
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 7: After placing each element at its correct position in output array, decrease its count by one in count array.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count									
	0	3	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

3-1

		1							
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 7: After placing each element at its correct position in output array, decrease its count by one in count array.

0	1	2	3	4	5	6	7	8	9
1	3	2	8	5	1	5	1	2	7

count

0	1	2	3	4	5	6	7	8
0	2	5	6	6	8	8	9	10

0	1	2	3	4	5	6	7	8	9
		1							

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

		1							
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

6-1

		1							
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	$\boxed{3}$	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	5	6	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

6-1

		1			$\boxed{3}$				
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	$\boxed{3}$	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	5	5	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

6-1

		1			$\boxed{3}$				
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	5	5	6	8	8	9	10
	0	1	2	3	4	5	6	7	8

5-1

		1		2	3				
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	$\boxed{2}$	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count

0	2	4	5	6	8	8	9	10
0	1	2	3	4	5	6	7	8

5-1

		1		$\boxed{2}$	3				
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

	1	3	2	8	5	1	5	1	2	7
	0	1	2	3	4	5	6	7	8	9

count	0	2	4	5	6	8	8	9	9
	0	1	2	3	4	5	6	7	8

8-1

		1		2	3		5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	4	5	6	7	8	9	9
	0	1	2	3	4	5	6	7	8

8-1

		1		2	3		5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	2	4	5	6	7	8	9	9
	0	1	2	3	4	5	6	7	8

2-1

	$\boxed{1}$	1		2	3		5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	$\boxed{1}$	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	1	4	5	6	7	8	9	9
	0	1	2	3	4	5	6	7	8

	2-1								
	$\boxed{1}$	1		2	3		5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	$\boxed{5}$	1	2	7
0	1	2	3	4	5	6	7	8	9

count

0	1	4	5	6	7	8	9	9
0	1	2	3	4	5	6	7	8

7-1

	1	1		2	3	$\boxed{5}$	5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

0	1	2	3	4	5	6	7	8	9
1	3	2	8	5	1	$\boxed{5}$	1	2	7

count

0	1	2	3	4	5	6	7	8
0	1	4	5	6	6	8	9	9

7-1

0	1	2	3	4	5	6	7	8	9
	1	1		2	3	$\boxed{5}$	5		8

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	1	4	5	6	6	8	9	9
	0	1	2	3	4	5	6	7	8

1-1

1	1	1		2	3	5	5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	0	4	5	6	6	8	9	9
	0	1	2	3	4	5	6	7	8

1-1

1	1	1		2	3	5	5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count

0	0	4	5	6	6	8	9	9
0	1	2	3	4	5	6	7	8

			4-1						
1	1	1	2	2	3	5	5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	$\boxed{2}$	7
0	1	2	3	4	5	6	7	8	9

count

0	0	3	5	6	6	8	9	9
0	1	2	3	4	5	6	7	8

			4-1						
1	1	1	$\boxed{2}$	2	3	5	5		8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	$\boxed{7}$
0	1	2	3	4	5	6	7	8	9

count	0	0	3	5	6	6	8	9	9
	0	1	2	3	4	5	6	7	8

9-1

1	1	1	2	2	3	5	5	$\boxed{7}$	8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Step 8: Repeat the same process for all the elements.

1	3	2	8	5	1	5	1	2	7
0	1	2	3	4	5	6	7	8	9

count	0	0	3	5	6	6	8	8	9
	0	1	2	3	4	5	6	7	8

9-1

1	1	1	2	2	3	5	5	7	8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Working

Now, the data in the output array is sorted.

1	1	1	2	2	3	5	5	7	8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Stable or Unstable

Now, the data in the output array is sorted.
Was it a Stable Sort or Unstable Sort?

1	1	1	2	2	3	5	5	7	8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Stable or Unstable

Now, the data in the output array is sorted.
Was it a Stable Sort or Unstable Sort?

UNSTABLE

1	1	1	2	2	3	5	5	7	8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Stable or Unstable

Now, the data in the output array is sorted.
Can we make it a Stable Sort?

1	1	1	2	2	3	5	5	7	8
0	1	2	3	4	5	6	7	8	9

Counting Sort: Stable

Now, the data in the output array is sorted.
Start iterating from the end of the array.

1	1	1	2	2	3	5	5	7	8
0	1	2	3	4	5	6	7	8	9

Another model of computation

- The items you are sorting have meaningful values.

9	6	3	5	2	1	2

instead of

Another model of computation

- The items you are sorting have meaningful values.

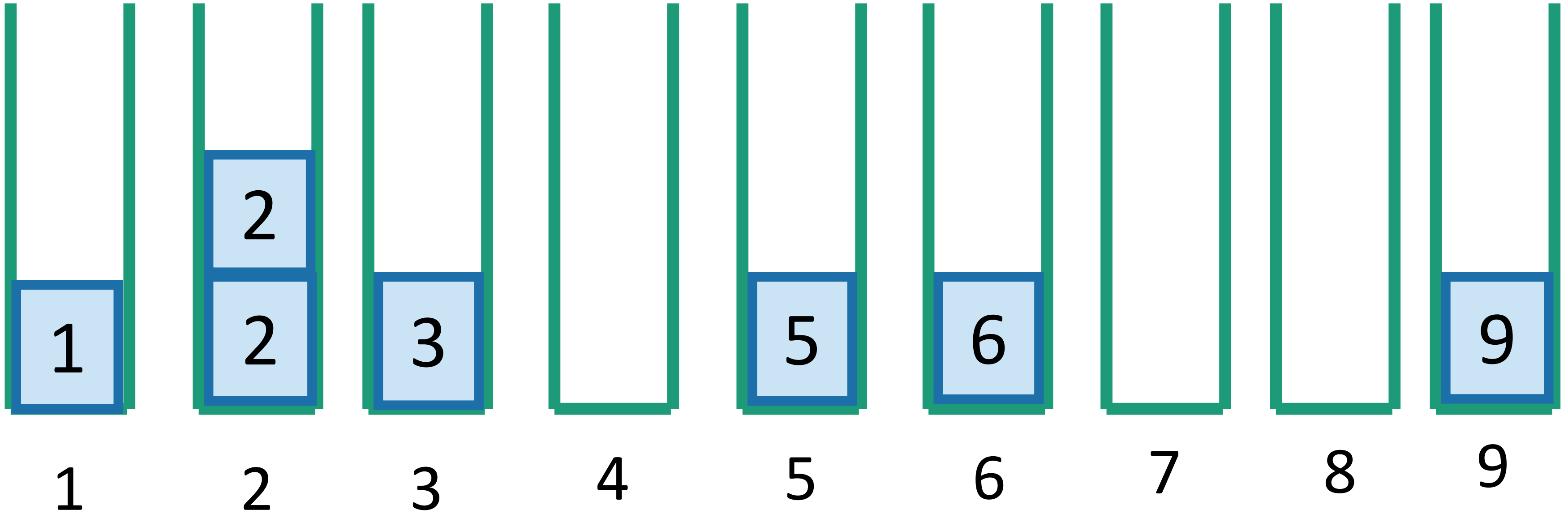
9	6	3	5	2	1	2

instead of

Why might this help?

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

CountingSort:



Concatenate
the buckets!

SORTED!

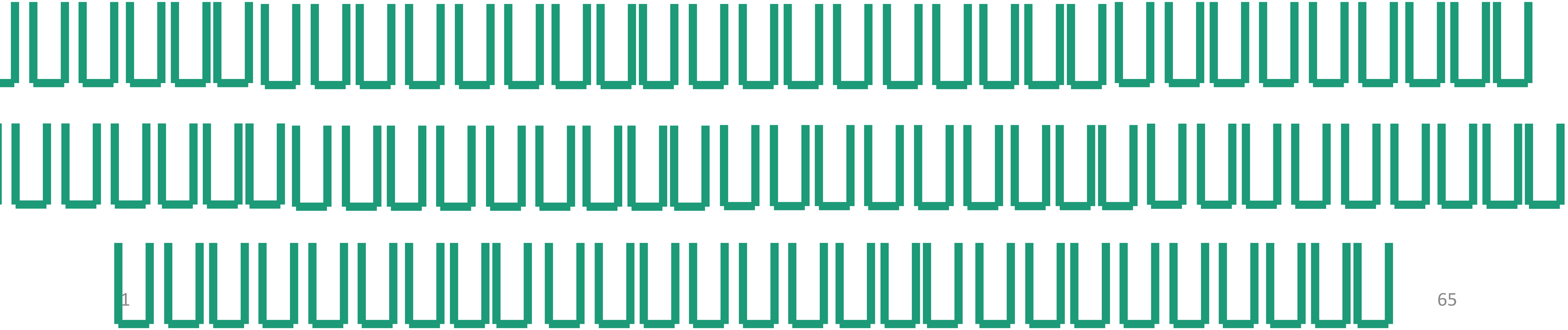
In time O(n).

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.

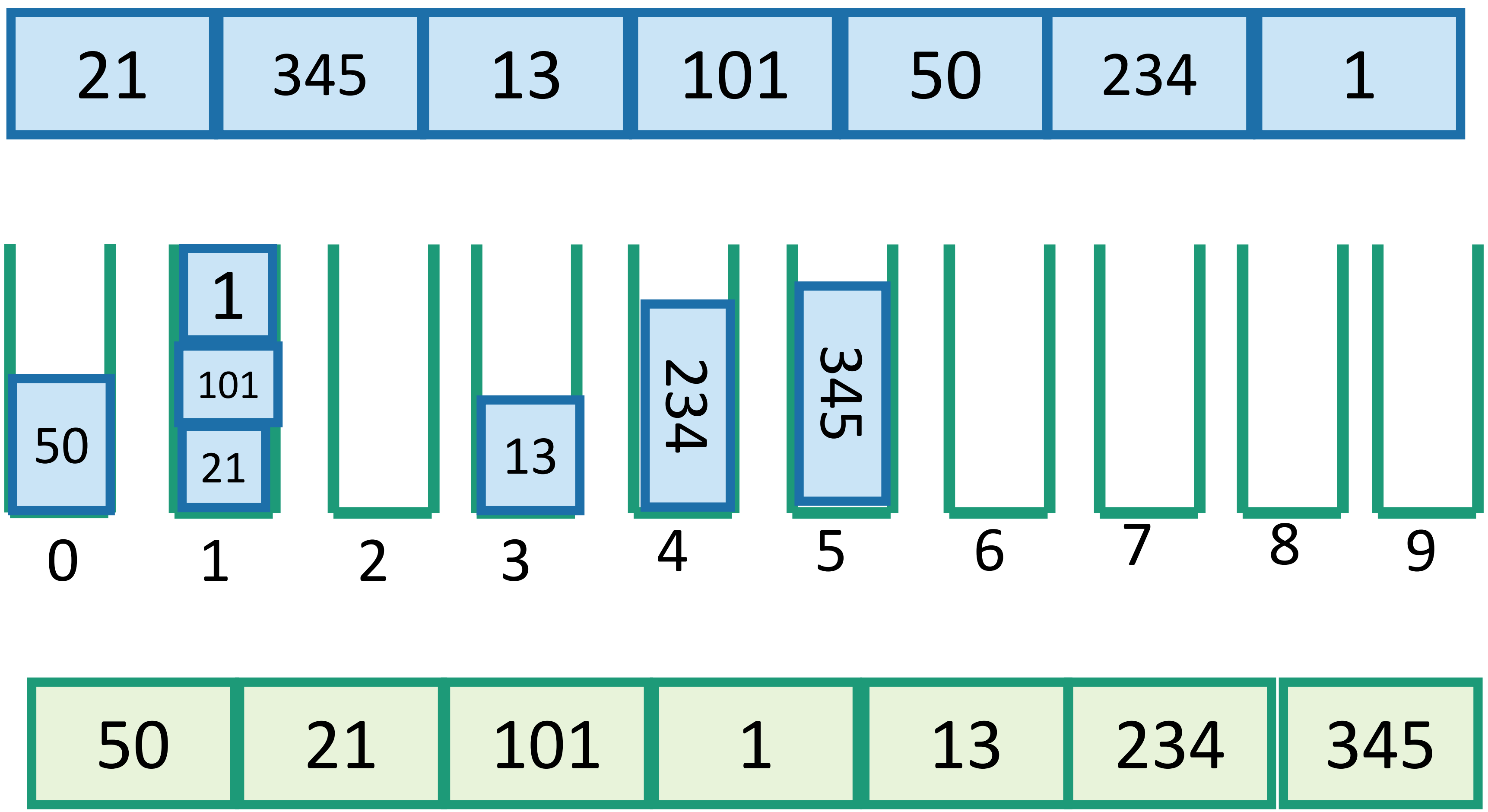


Radix Sort

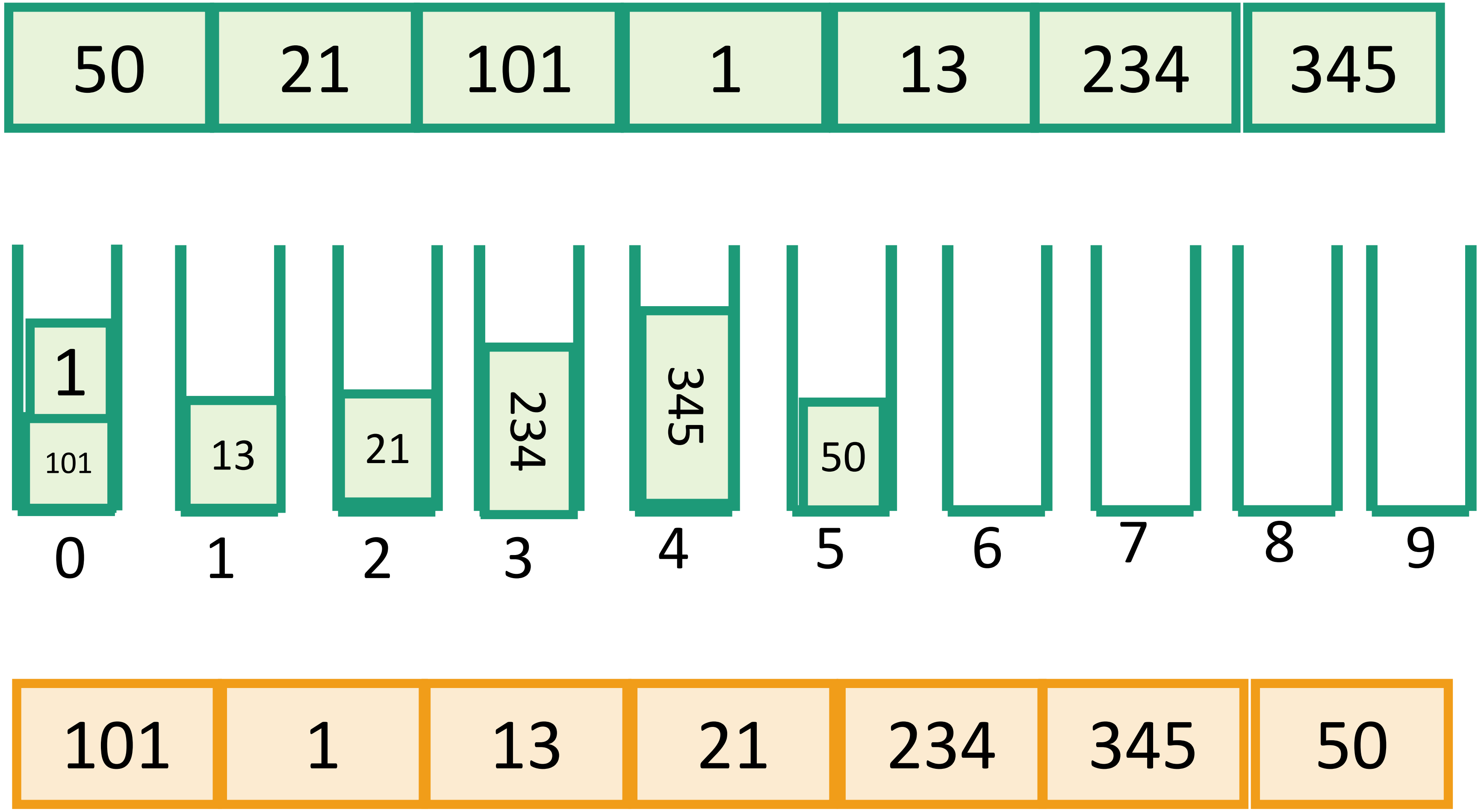
RadixSort

- 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

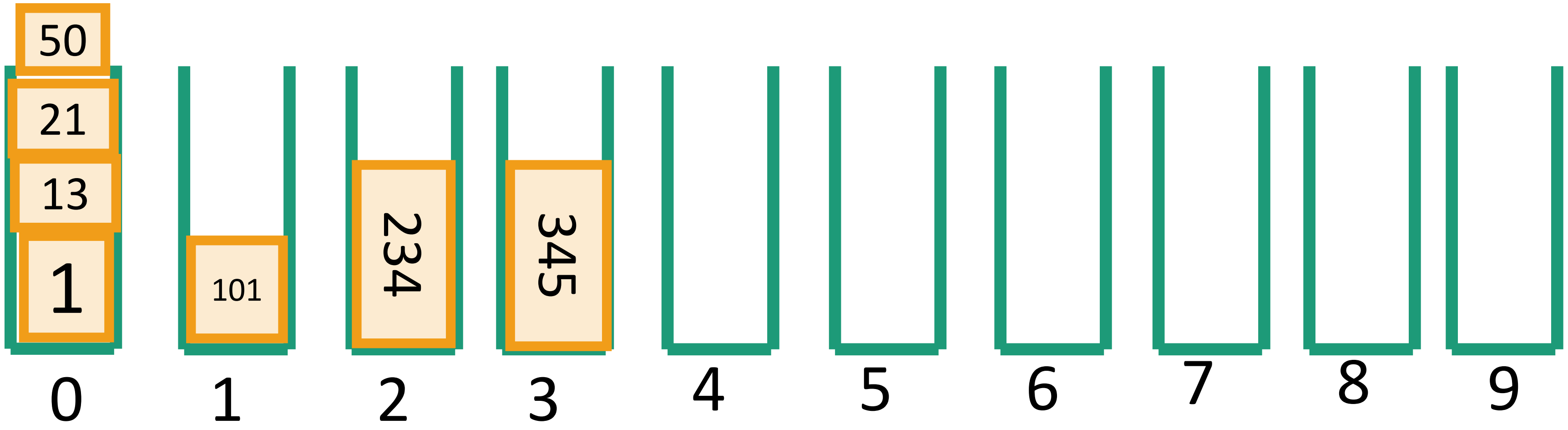


Step 2: CountingSort on the 2nd least sig. digit



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

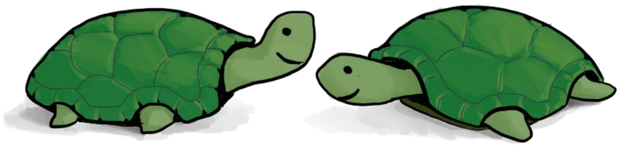
Next array is sorted by all three digits.

001	013	021	050	101	234	345

Sorted array

To prove this is correct…

- What is the inductive hypothesis?



Think-Pair-Share Terrapins
Think: 1 min
Pair + Share: 1 min

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!

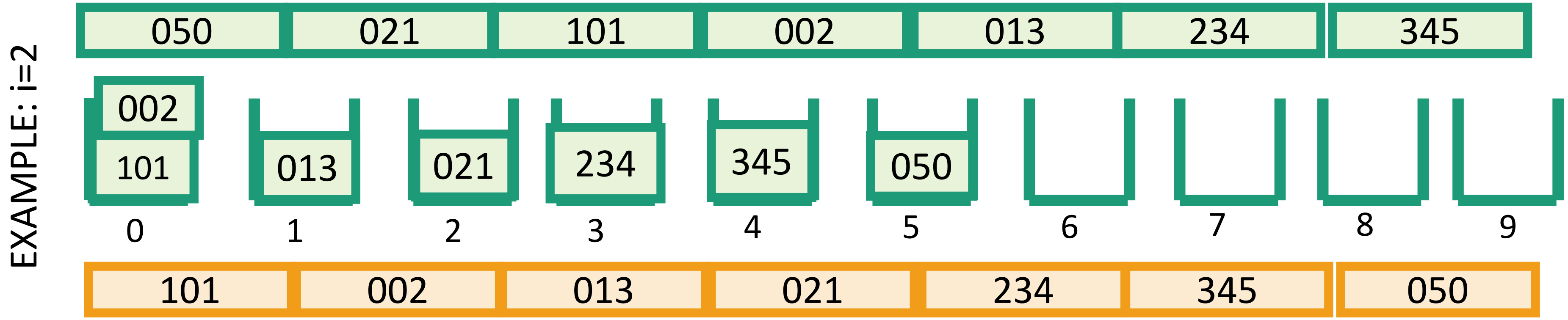
Inductive step

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.

IH: this array is sorted by first digit.



Want to show: this array is sorted by 1^{st} and 2^{nd} digits.

Proof sketch…

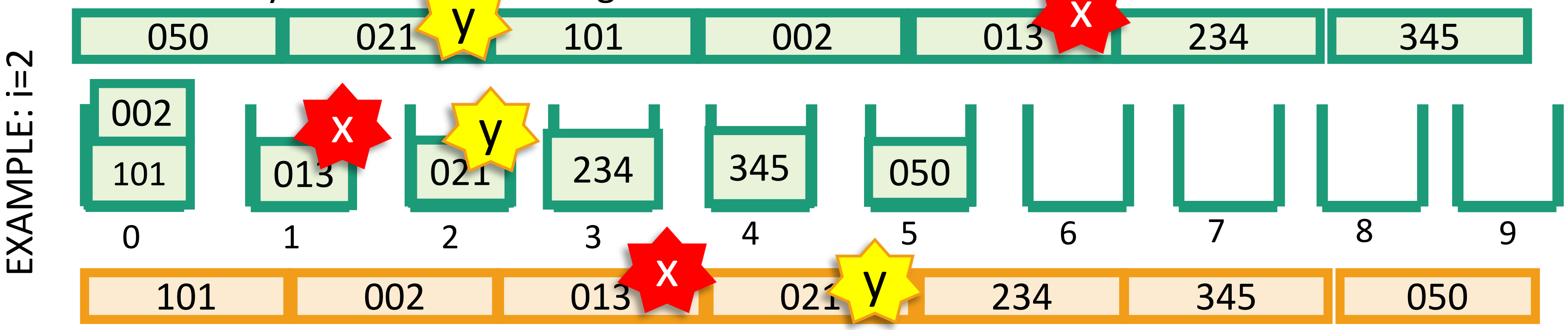
proof on next (skipped) slide

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

- Let $x=[x_d x_{d-1} \ldots x_2 x_1]$ and $y=[y_d y_{d-1} \ldots y_2 y_1]$ be any x,y.
- Suppose $[x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1]$.
- 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.

Aka, we want to show that for any x and y so that x belongs before y, we put x before y.

IH: this array is sorted by first digit.



Want to show: this array is sorted by 1^{st} and 2^{nd} digits.

Proof sketch…

proof on next (skipped) slide

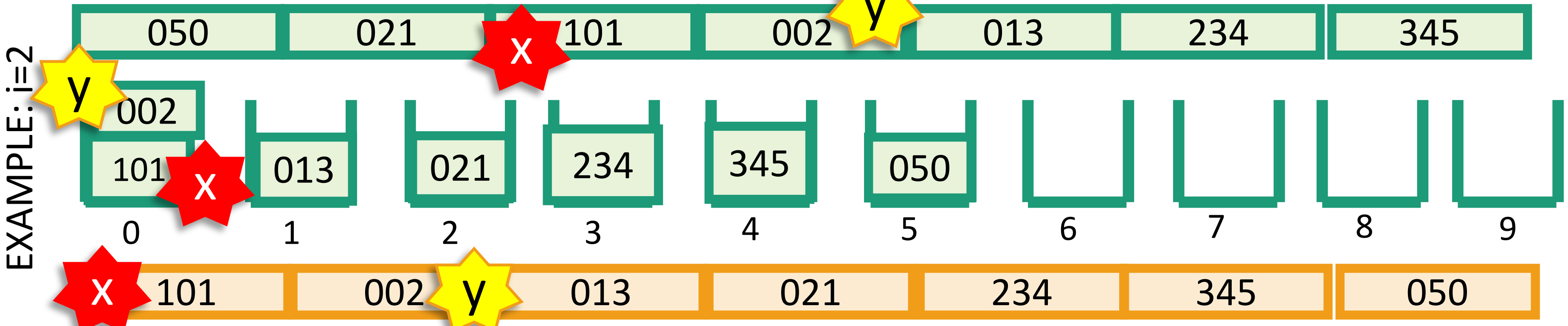
Want to show: after the i'th iteration, the array is sorted by the first i least-significant digits.

- Let $x=[x_d x_{d-1} \ldots x_2 x_1]$ and $y=[y_d y_{d-1} \ldots y_2 y_1]$ be any x,y.
- Suppose $[x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1]$.
- Want to show that x appears before y at end of i'th iteration.

Aka, we want to show that for any x and y so that x belongs before y, we put x before y.

- 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.

IH: this array is sorted by first digit.



Want to show: this array is sorted by 1^{st} and 2^{nd} digits.

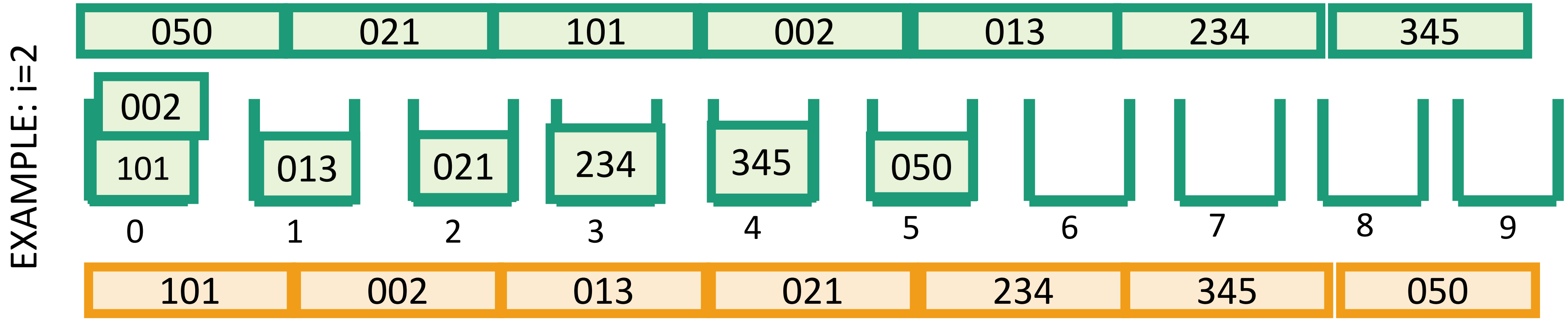
Inductive step

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 1st and 2nd 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).

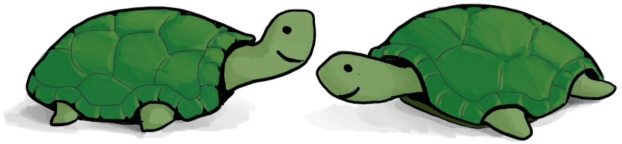
e.g., n=7, d=3:

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
Think: 3 minutes
Pair and share: 2 minutes

What is the running time?

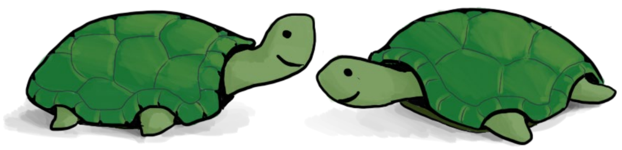
for RadixSorting numbers base-10.

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

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

021	345	013	101	050	234	001

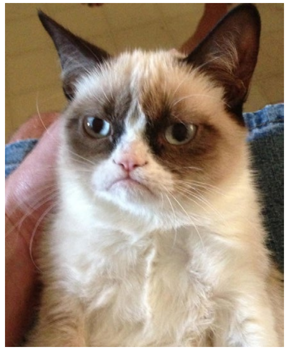
1. How many iterations are there?
   - d iterations
2. How long does each iteration take?
   - Time to initialize 10 buckets, plus time to put n numbers in 10 buckets. O(n).
3. What is the total running time?
   - O(nd)



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
    - $\lfloor \log_{10}(1234) \rfloor + 1 = 4$
- 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:

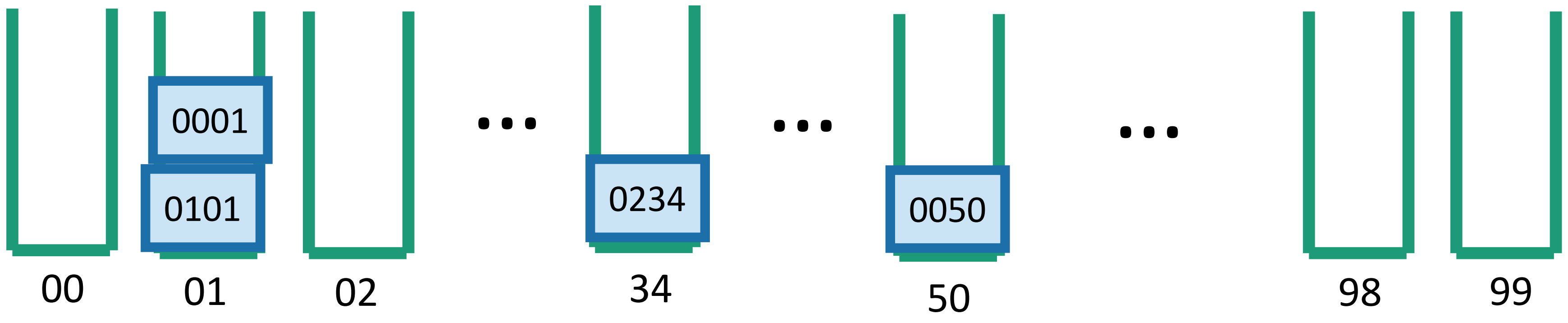
21	345	13	101	50	234	1

Example: base 100

Original array:

0021	0345	0013	0101	0050	0234	0001

100 buckets:

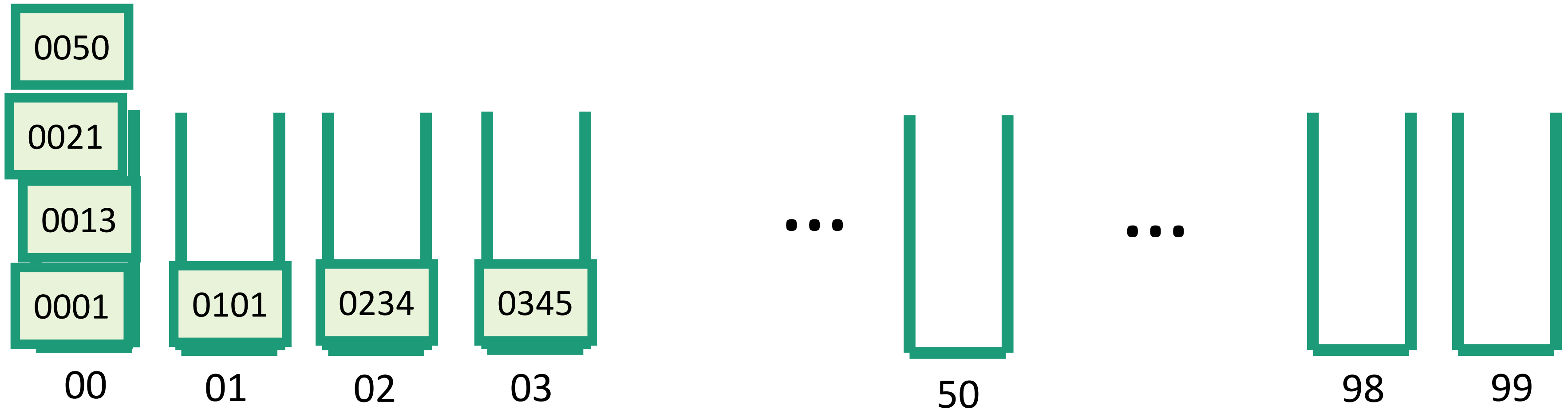


0101	0001	0013	0021	0234	0345	0050

Example: base 100

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.

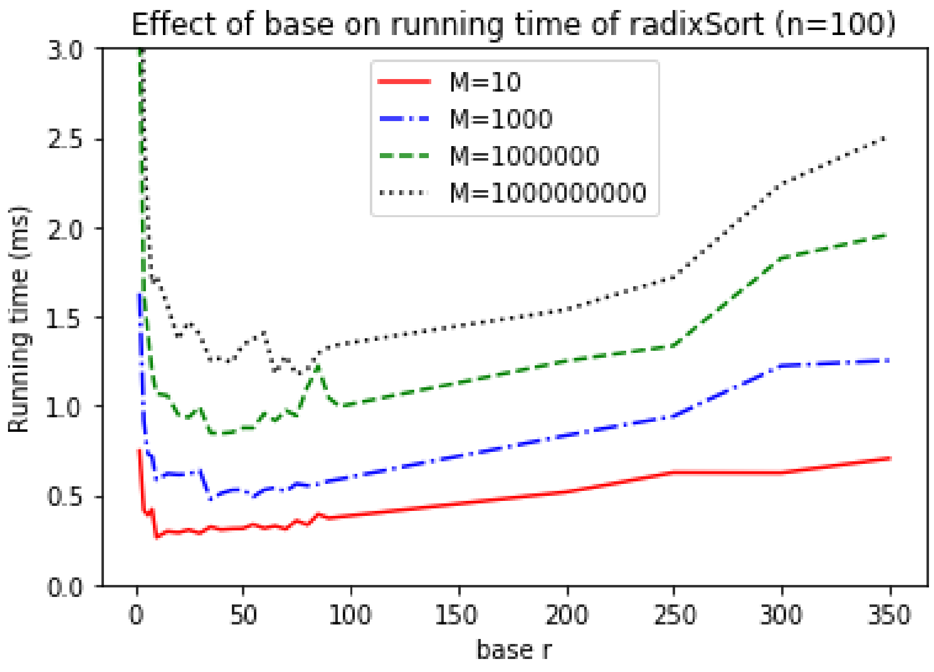
General running time of RadixSort

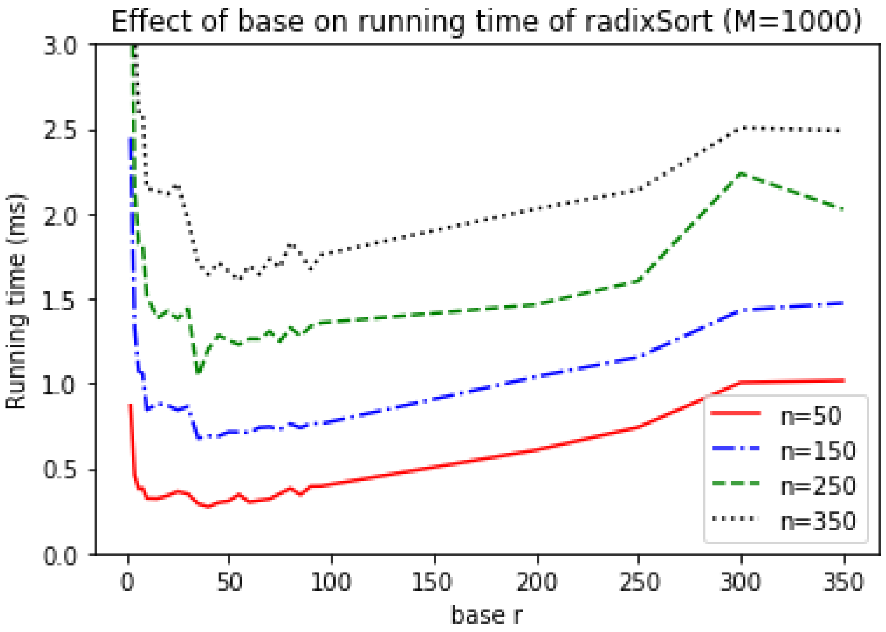
- Say we want to sort:
  - n integers,
  - maximum size M,
  - in base r.
- Number of iterations of RadixSort:
  - Same as number of digits, base r, of an integer x of max size M.
  - That is $d = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
  - Initialize r buckets, put n items into them
  -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

Trade-offs

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





A reasonable choice: r=n

- Running time:

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

Intuition: balance n and r here.

- Choose n=r:

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

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?

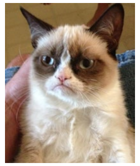
You can put any constant here instead of 100.

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

Bucket Sort

Sorting in Linear Time

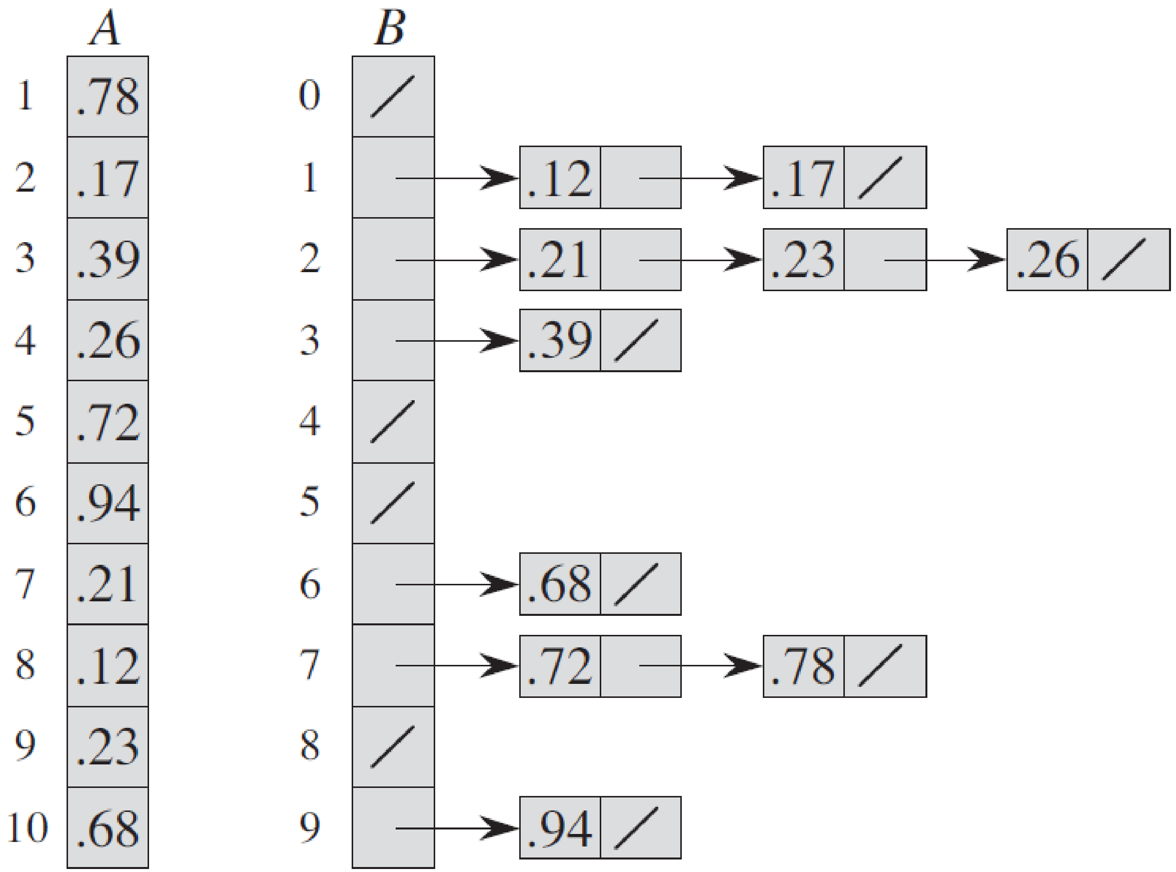
What is Bucket Sort?

- Input is generated from the uniform distribution in the interval [0,1)
- Interval is divided into n equal sized sub-intervals
- Buckets contains the input values
- For each bucket, run a sorting algorithm.

BUCKET-SORT(A)

```
let B[0 .. n − 1] be a new array
n = A.length
for i = 0 to n − 1
        make B[i] an empty list
for i = 1 to n
        insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
        sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Bucket Sort in Action



Time Complexity

BUCKET-SORT(A)

let $B[0\,..\,n-1]$ be a new array
$n = A.length$
for $i = 0$ **to** $n - 1$
    make $B[i]$ an empty list
for $i = 1$ **to** n
    insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
for $i = 0$ **to** $n - 1$
    sort list $B[i]$ with insertion sort
concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order

Review of Sorting Algorithms

Comparison Based Algorithms

- Merge Sort
- Selection Sort
- Bubble Sort
- Quick Sort
- Insertion Sort
- Any other Two Algorithms

Linear Time Sorting Algorithms

- Counting Sort
- Radix Sort
- Bucket Sort
- Any other two Algorithms

Components to Complete

- Description of Algorithm in your own words
- Pseudo code of algorithm
- Time Complexity Analysis
- Three Strengths
- Three Weakness
- Dry run on small input

Thank You