CSE100: Design and Analysis of Algorithms
Lecture 12 – Randomized Algorithms (wrap up) and Sorting Lower Bounds

Mar 01st 2022

Randomized Algorithms,QuickSort, Sorting lower bounds and O(n)-time sorting

PseudoPseudoCode for QuickSort (review)

Lab 04-2 asks for an implementation of 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)

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

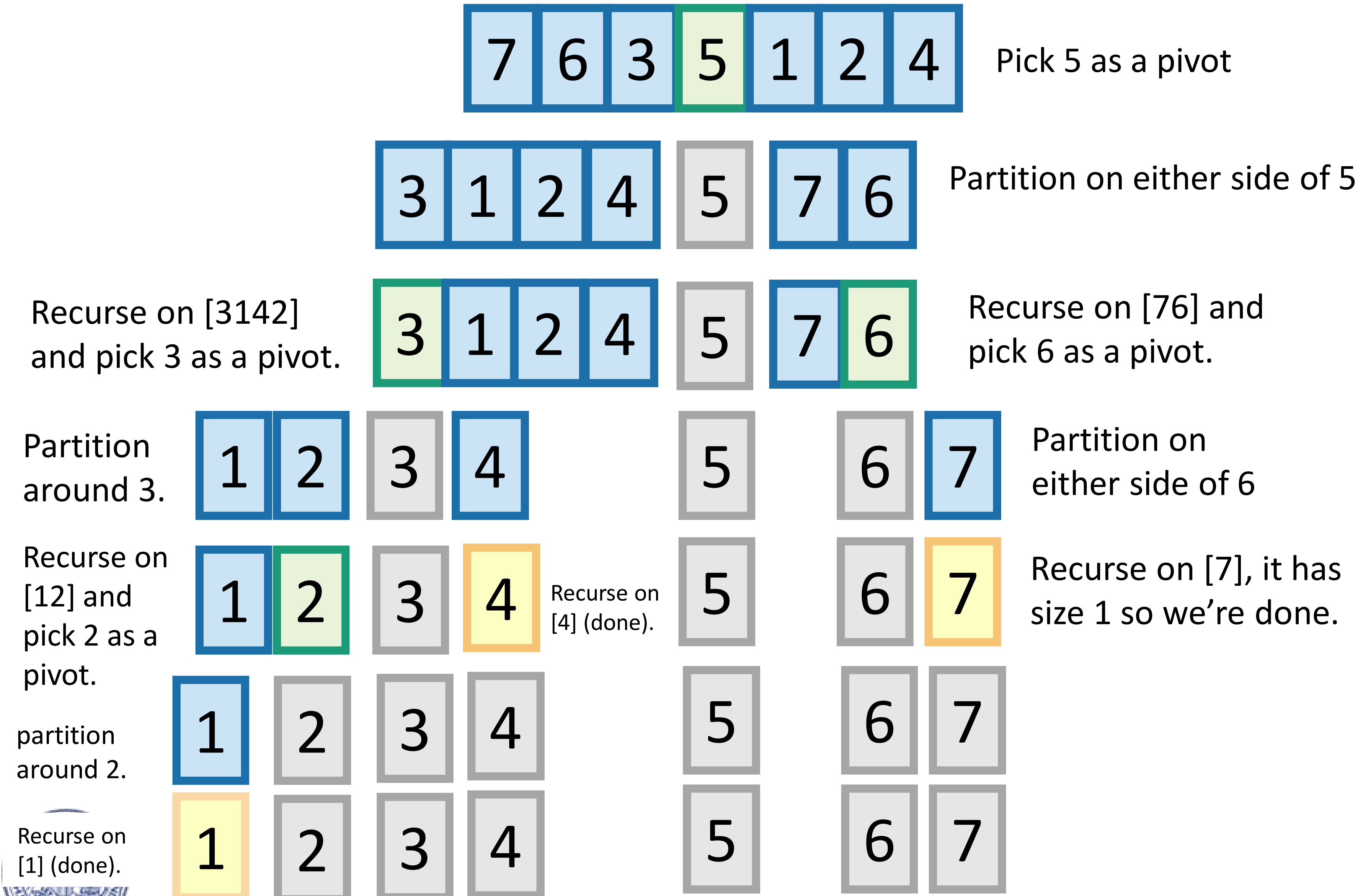
How would you do all this in-place? Without hurting the running time? (We'll see later…)

Today

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

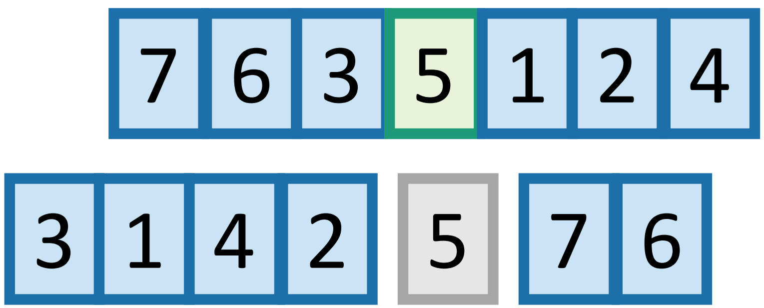
- BogoSort is a pedagogical tool.
- QuickSort is important to know. (in contrast with BogoSort…)

Example of recursive calls

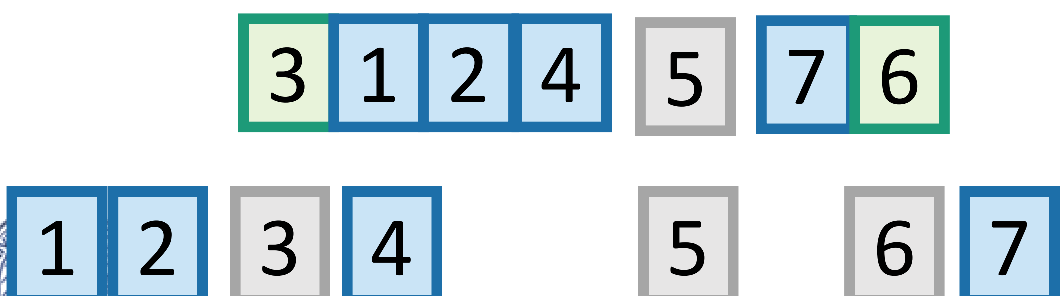


How long does this take to run?

- 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.

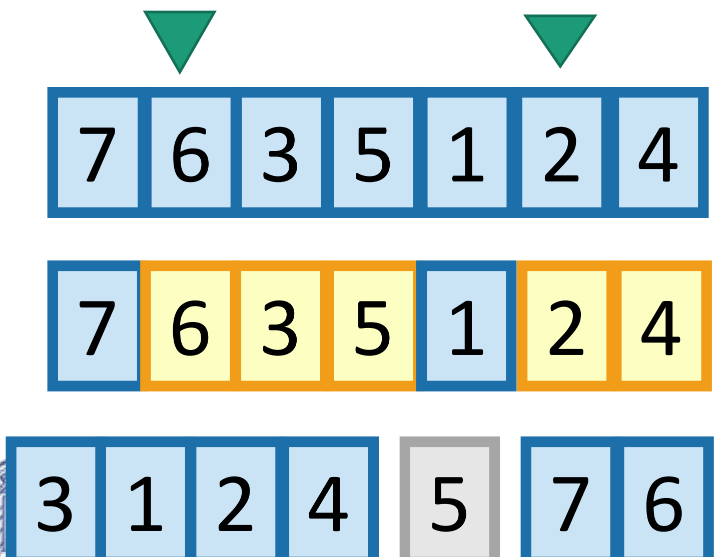
Counting comparisons

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

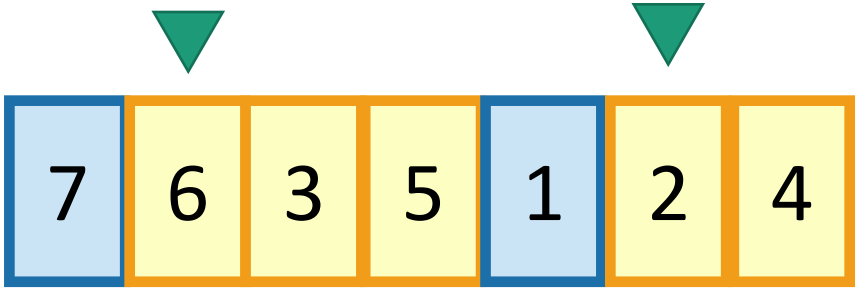
$P(\ X_{a,b} = 1\ )$

= 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,…,b}
- $P\{a, b \text{ are ever compared}\}$

  $= \sum_{\text{stuff}} P\{\text{a or b picked first out of S}|\text{ stuff}\} \cdot P\{\text{stuff}\}$

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:

$= \sum_{\text{stuff}} P\{\text{a or b picked first out of S}\} \cdot P\{\text{stuff}\}$

$= P\{\text{a or b picked first out of S}\} \cdot \sum_{\text{stuff}} P\{\text{stuff}\}$

$= P\{\text{a or b picked first out of S}\}$

$= 2/|S|$

Aside:

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

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

a_7	a_6	a_3	a_5	a_1	a_2	a_4

- 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

- $E\left[\sum_{a=1}^{n-1}\sum_{b=a+1}^{n} X_{a,b}\right]$ — This is the expected number of comparisons throughout the algorithm
- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} E[\, X_{a,b}]$ — linearity of expectation
- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} P(\, X_{a,b} = 1)$ — definition of expectation
- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} \frac{2}{b-a+1}$ — 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[number of comparisons] = O(n log(n))
- Is that the same as E[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** 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)

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.

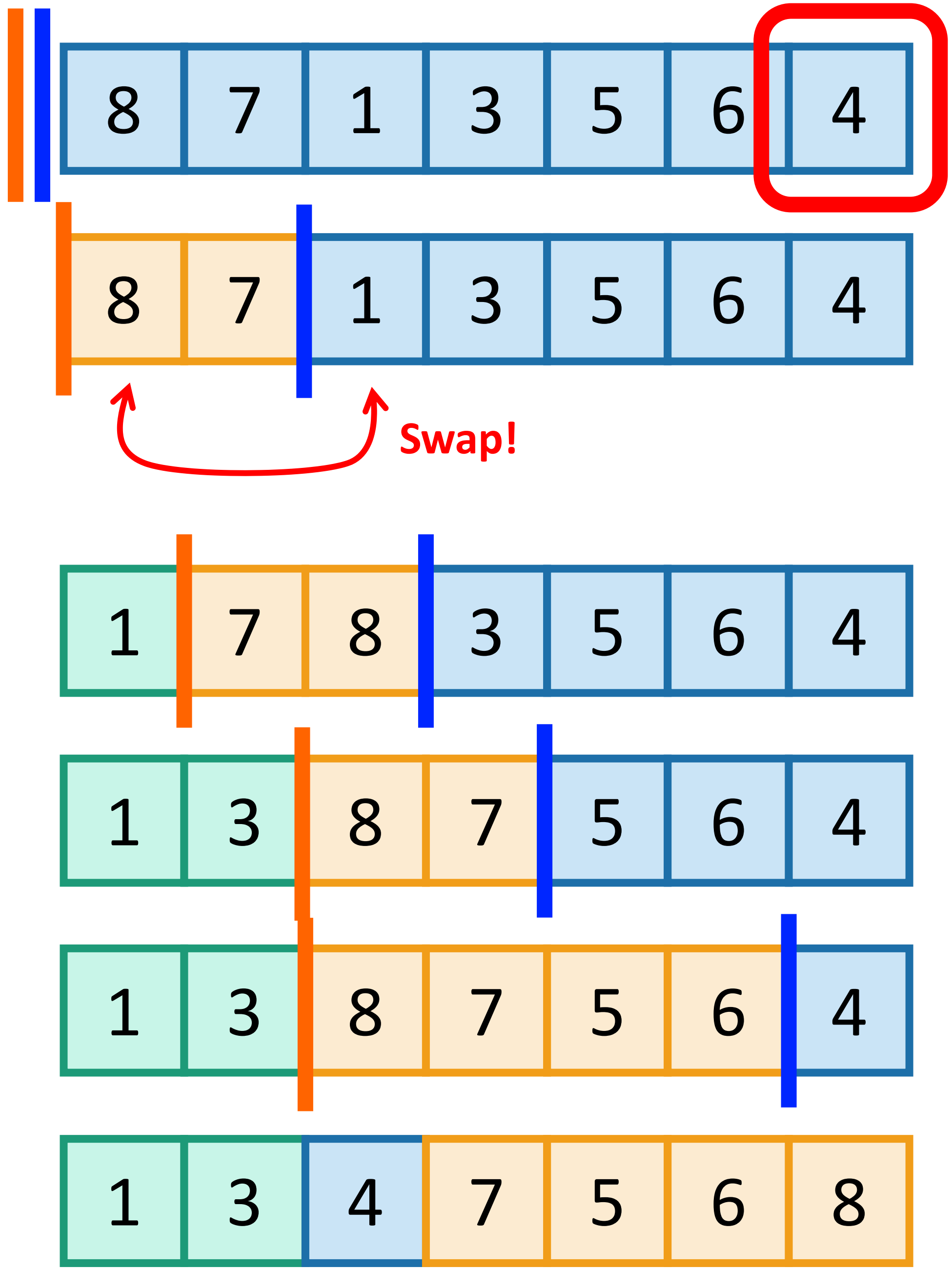
A note on implementation

- 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)
  - Here are some Hungarian Folk Dancers showing you how it's done: https://www.youtube.com/watch?v=ywWBy6J5gz8.

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.

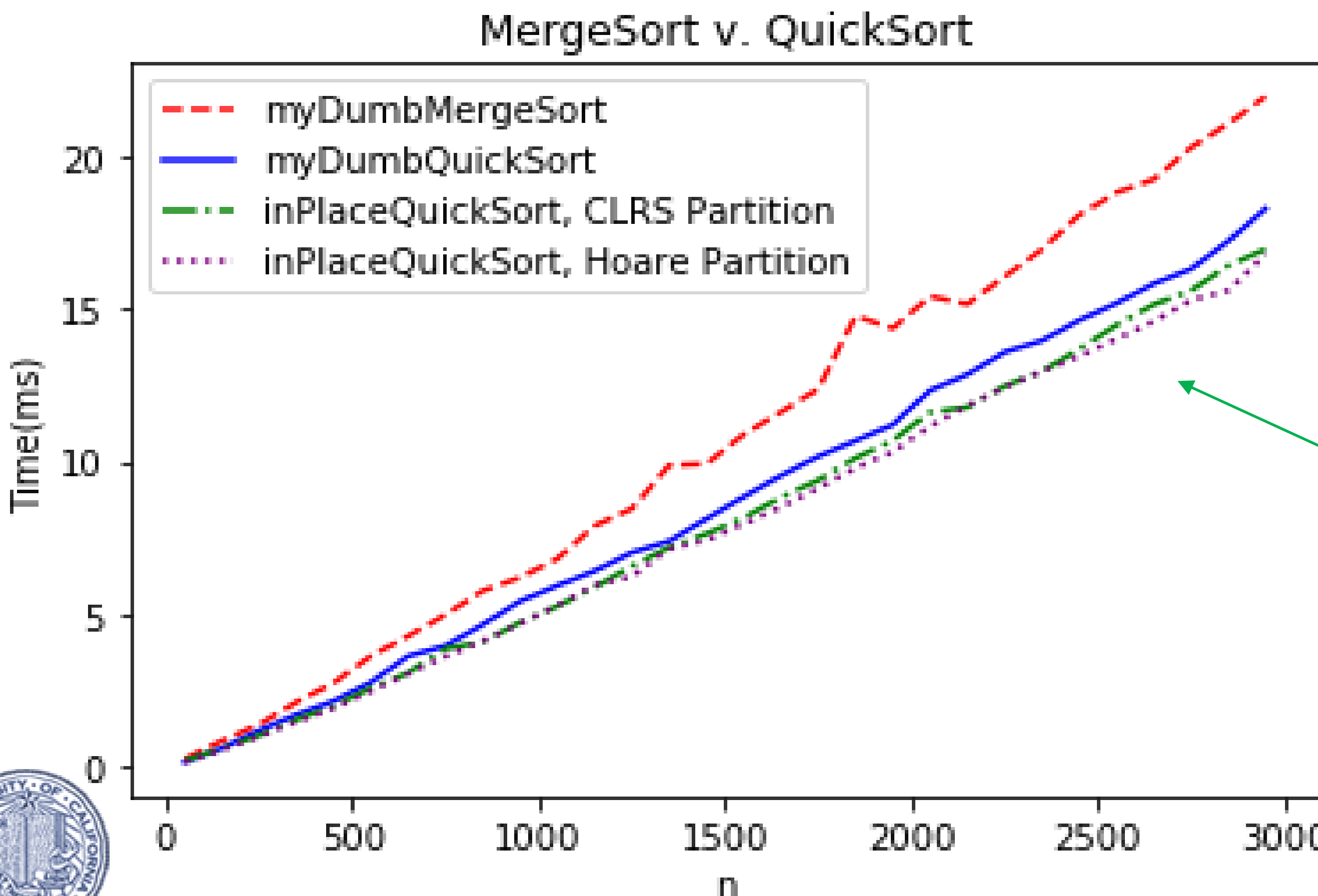
Repeat till the end, then put the pivot in the right place.

See CLRS for pseudocode.

QuickSort vs. smarter QuickSort vs. Mergesort?

python™

- 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(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).

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).

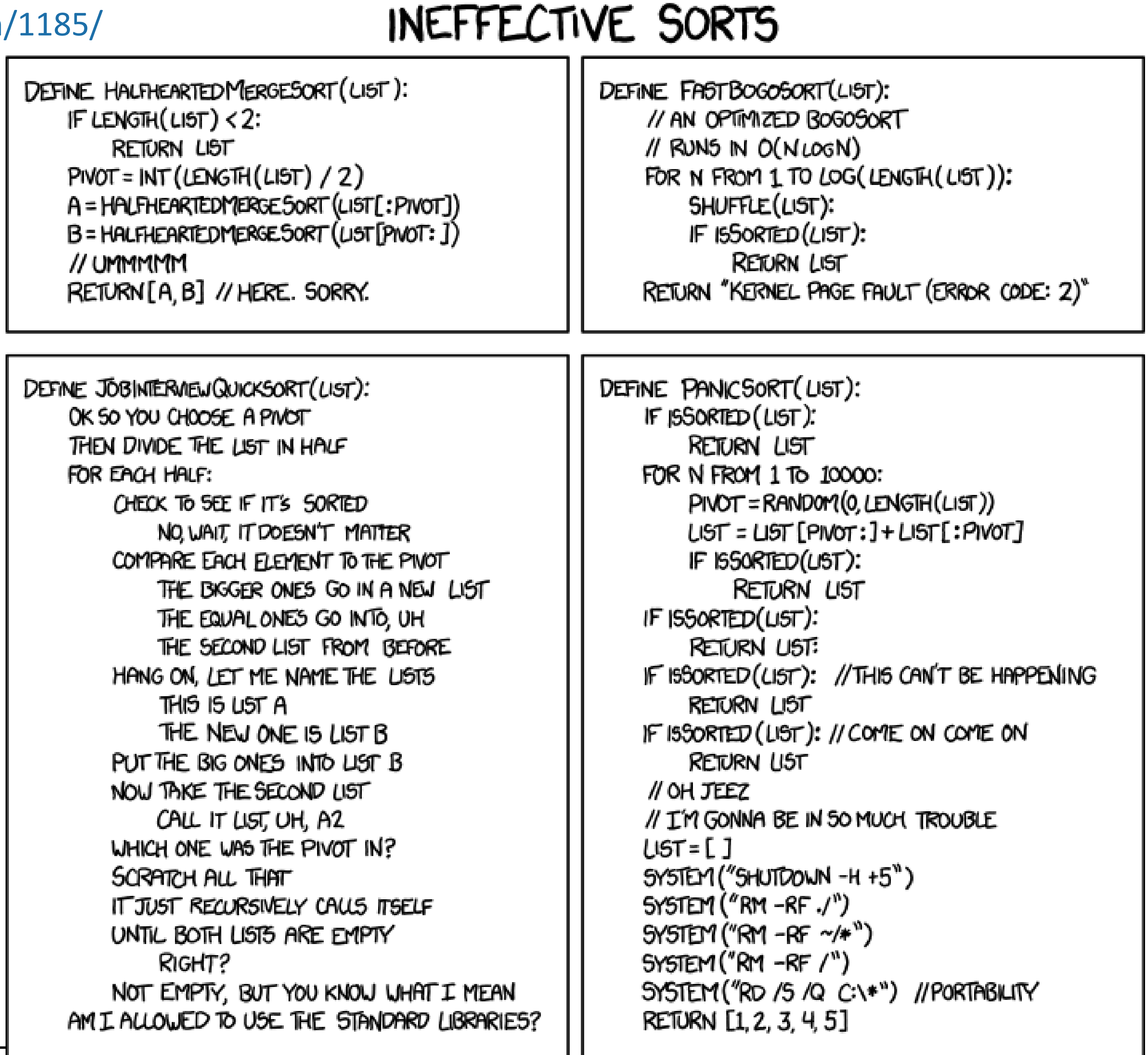
Ollie the over-achieving ostrich

Next Part

- Can we sort even faster than QuickSort/MergeSort?
- Can we sort faster than $\Theta(n\log(n))$??

https://xkcd.com/1185/

(h/t Dana)

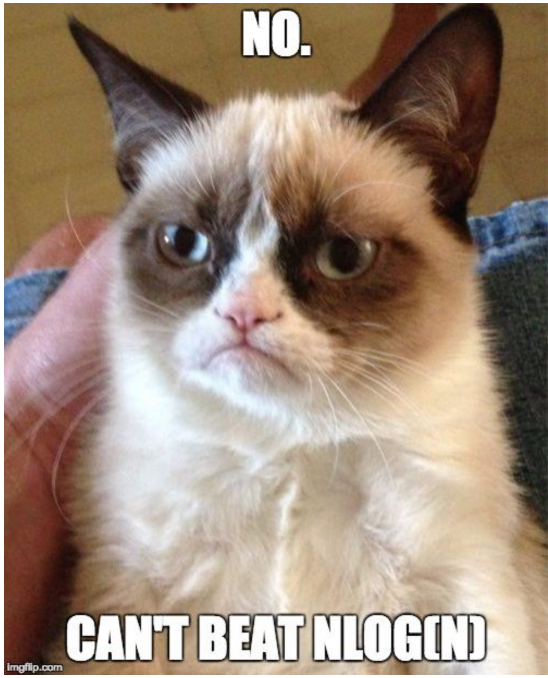


Sorting

- 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?

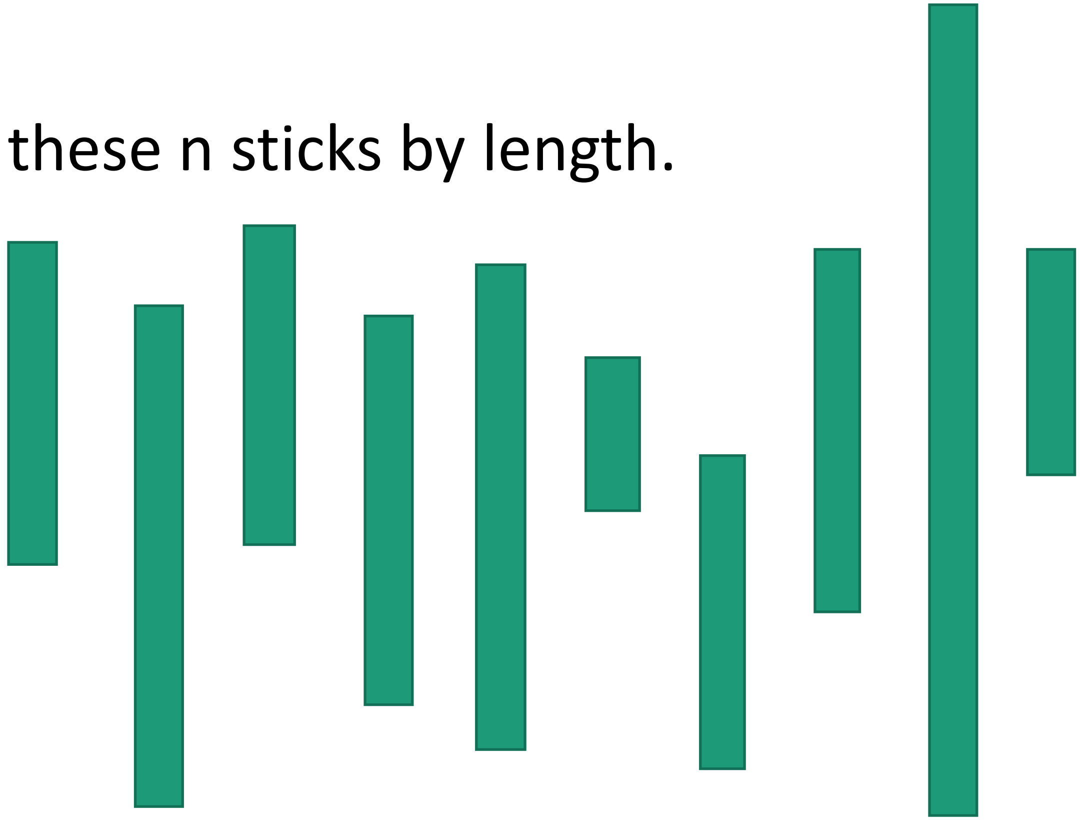
Depends on who you ask…



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

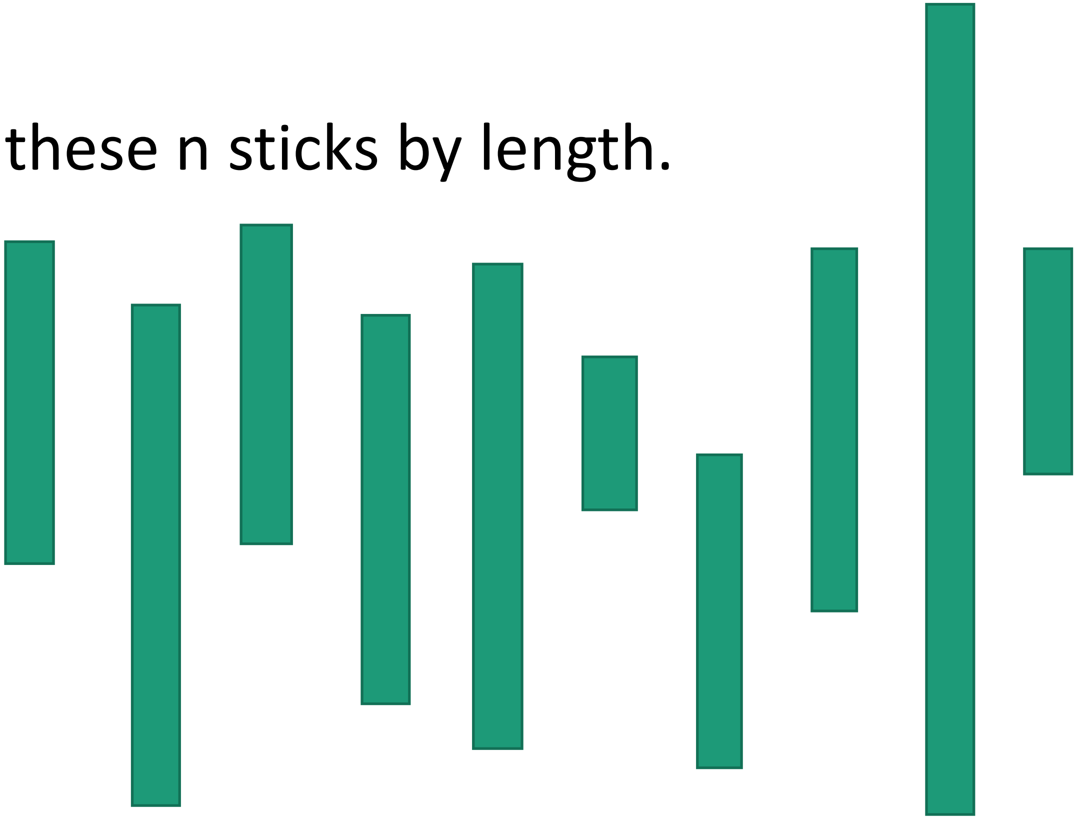
An O(1)-time algorithm for sorting: StickSort

- Problem: sort these n sticks by length.



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

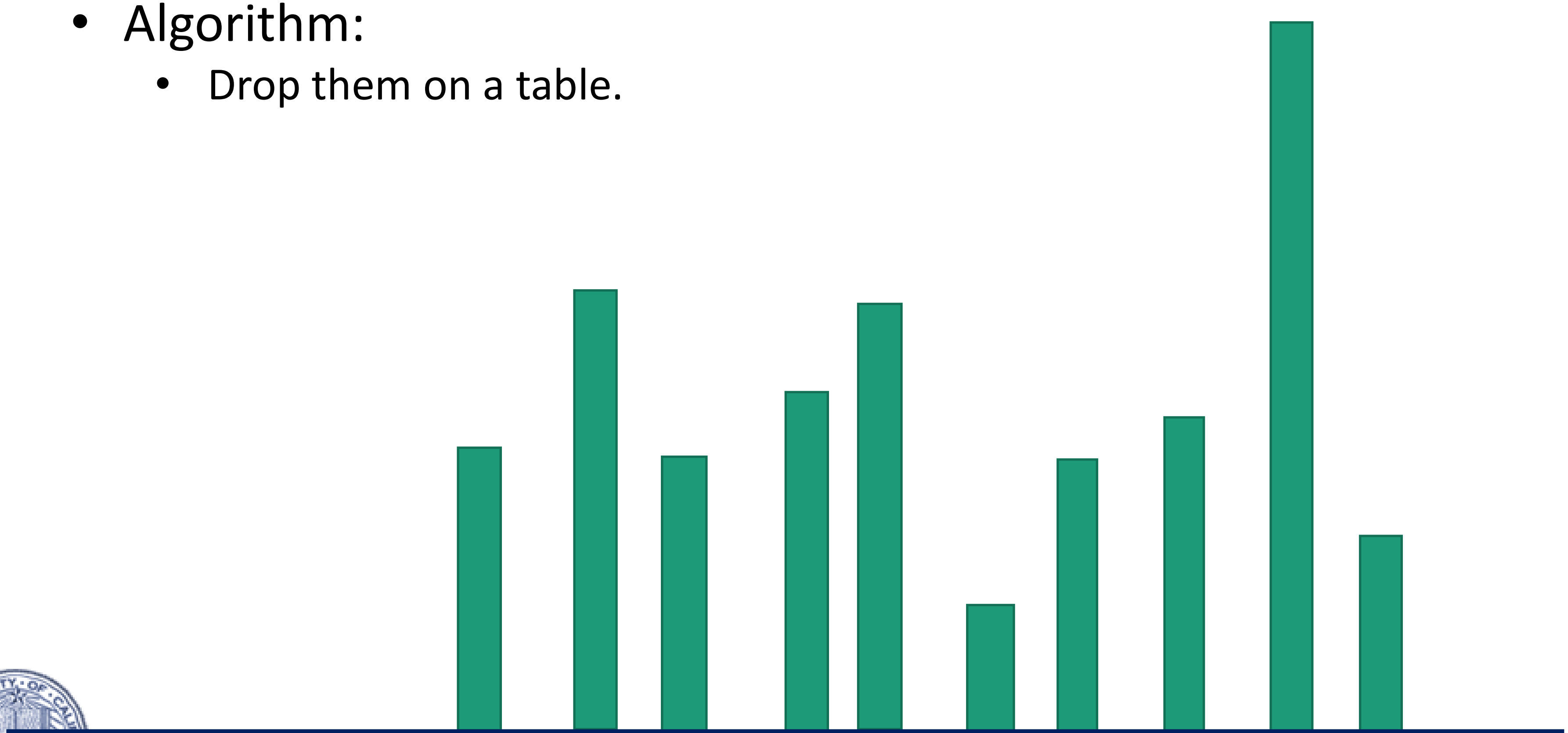
- Problem: sort these n sticks by length.



- Algorithm:
  - Drop them on a table.

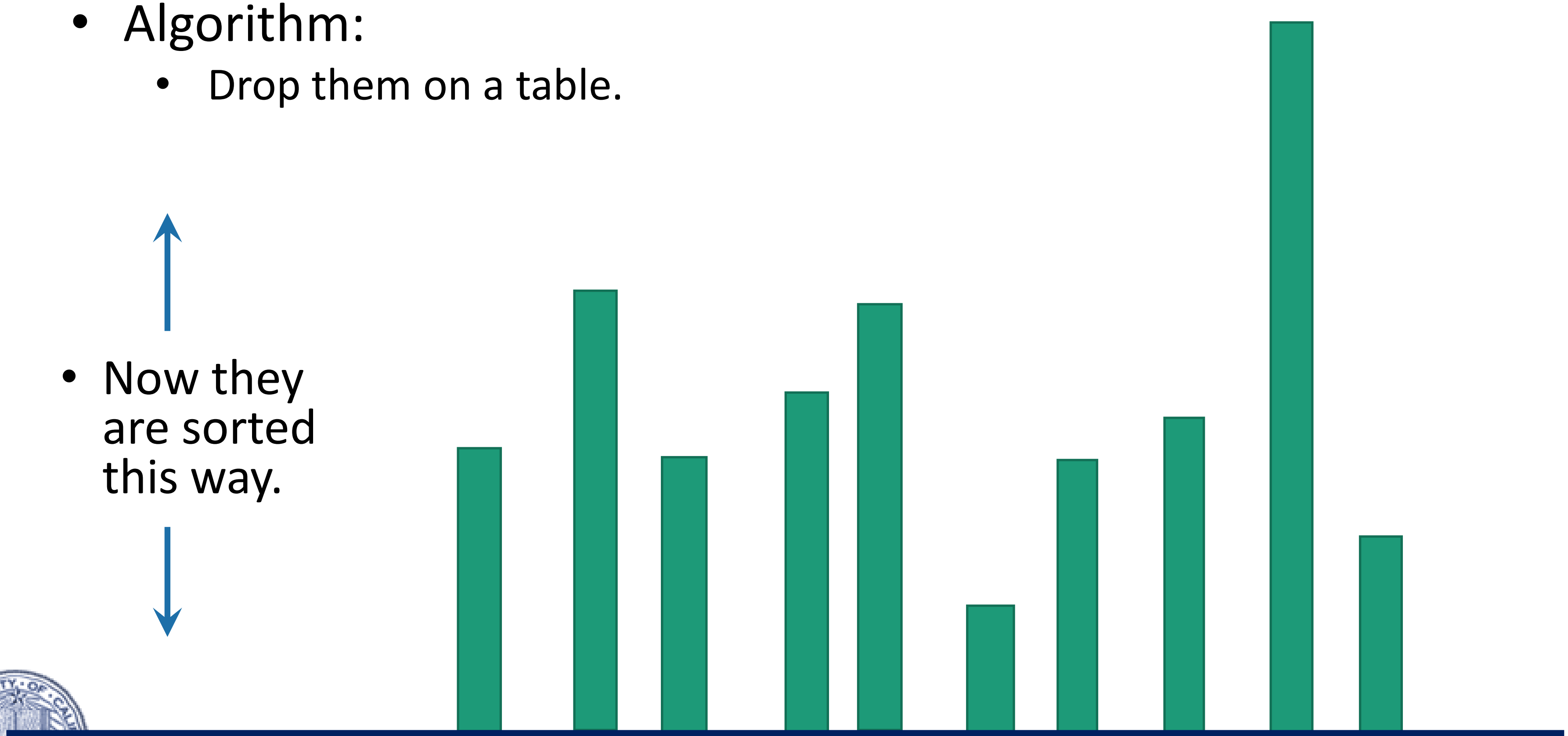
An O(1)-time algorithm for sorting: StickSort

- Problem: sort these n sticks by length.
- Algorithm:
  - Drop them on a table.



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

- Problem: sort these n sticks by length.
- Algorithm:
  - Drop them on a table.
- Now they are sorted this way.



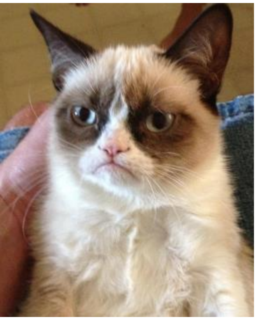
That may have been unsatisfying

- 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?

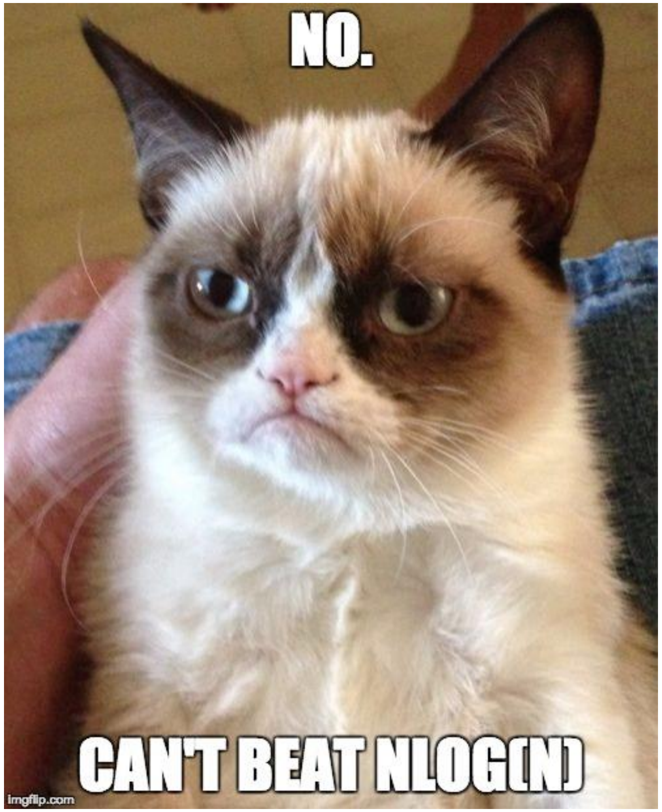
Today: two (more) models



- 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



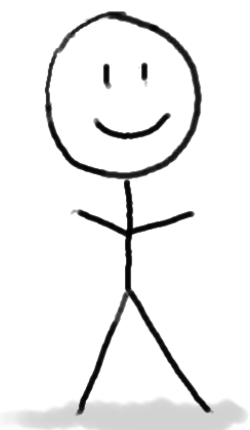
Comparison-based sorting algorithms

 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.



Algorithm

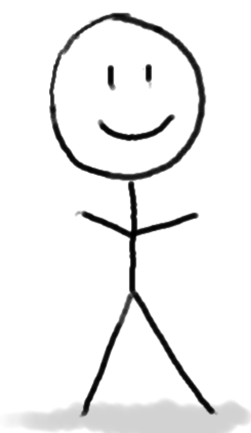
Comparison-based sorting algorithms

 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.

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



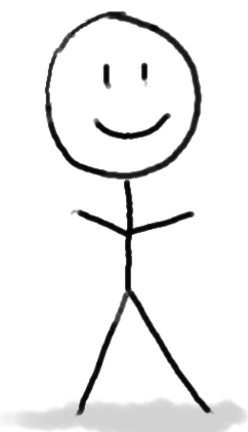
Algorithm

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.



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]?

Algorithm

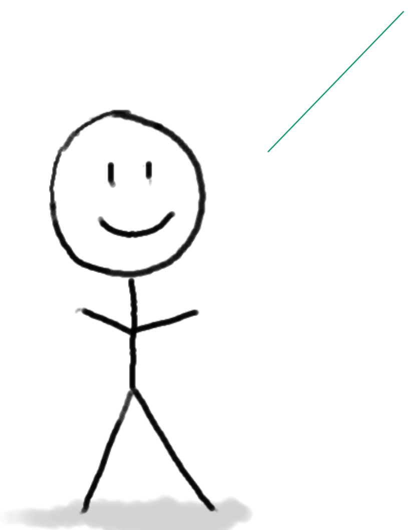
Comparison-based sorting algorithms

 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 igger than 



Algorithm

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]?

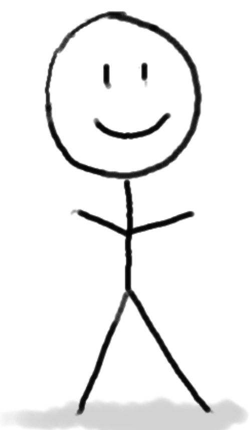
Comparison-based sorting algorithms

 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 igger than 



YES

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]?

Algorithm

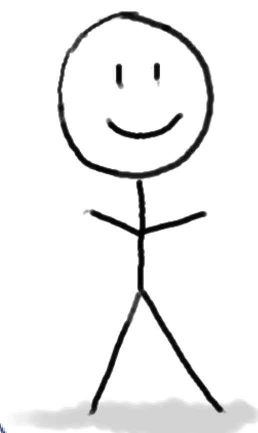
Comparison-based sorting algorithms

 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 igger than 



YES

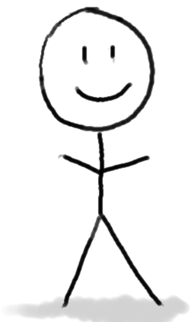
There is a genie who knows what the right order is.

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

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

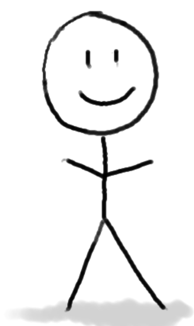
Algorithm

All the sorting algorithms we have seen work like this.



All the sorting algorithms we have seen work like this.

eg, QuickSort:

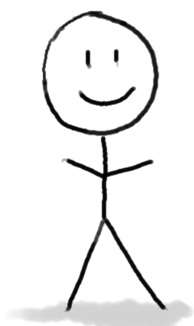


All the sorting algorithms we have seen work like this.

eg, QuickSort:

Pivot!

7	6	3	5	1	4	2

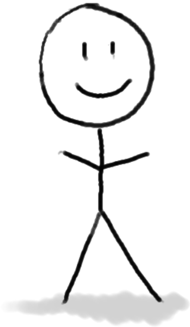


All the sorting algorithms we have seen work like this.

Pivot!

eg, QuickSort:

7	6	3	5	1	4	2

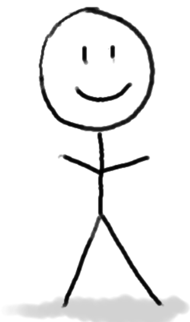


All the sorting algorithms we have seen work like this.

Pivot!

eg, QuickSort:

7	6	3	5	1	4	2



5

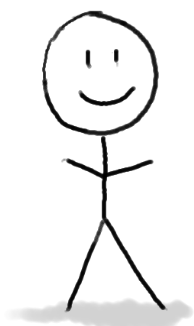
All the sorting algorithms we have seen work like this.

eg, QuickSort:

Pivot!

7	6	3	5	1	4	2

Is 7 bigger than 5 ?



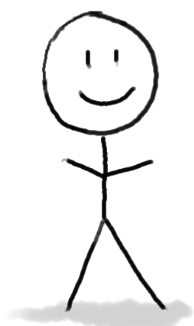
5

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

5

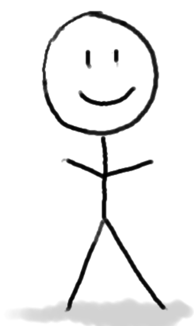
All the sorting algorithms we have seen work like this.

Pivot!

eg, QuickSort: | 7 | 6 | 3 | 5 | 1 | 4 | 2 |

Is 7 bigger than 5 ?

YES



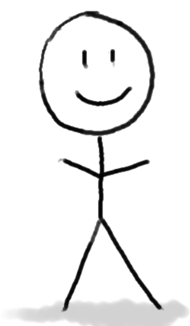
5 7

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

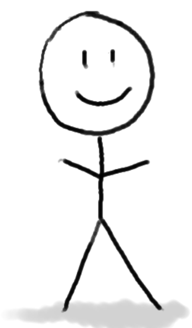
Is 6 bigger than 5 ?

5 7

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**

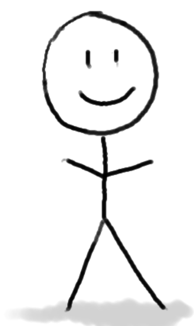
5 7

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

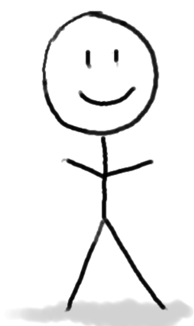
5 7 6

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 ?

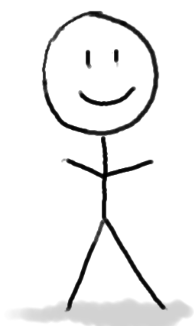
5 7 6

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**

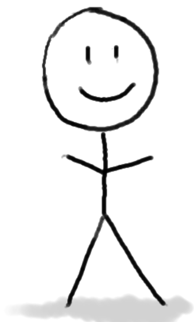
5 7 6

All the sorting algorithms we have seen work like this.

Pivot!

eg, QuickSort:

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**

3

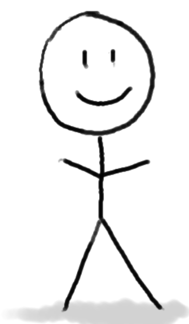
5 7 6

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**

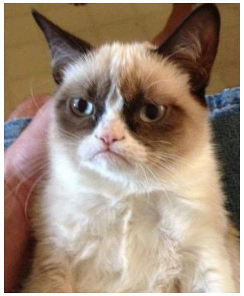
Is 6 bigger than 5 ? **YES**

Is 3 bigger than 5 ? **NO**

3

5 7 6

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.

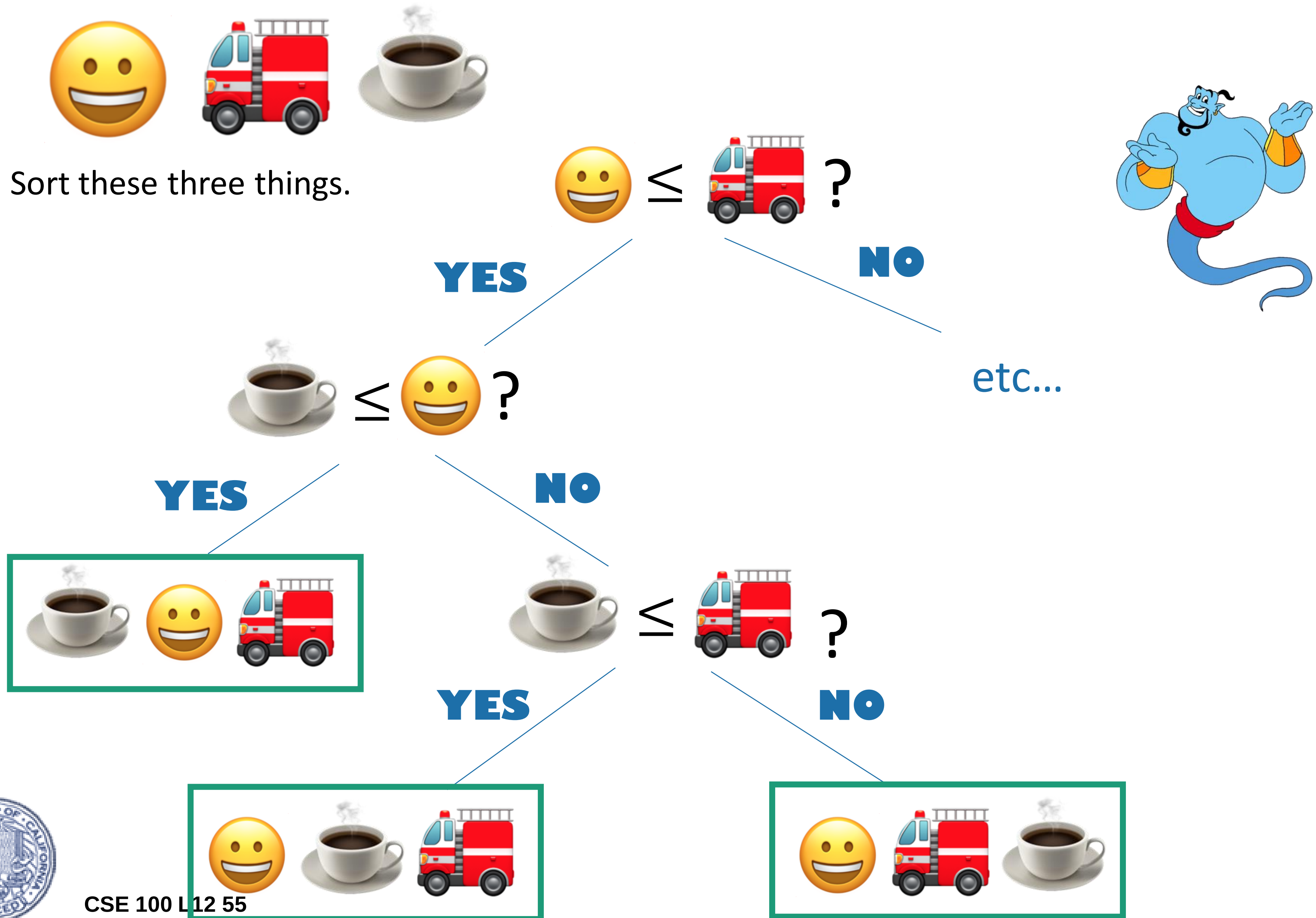
This covers all the sorting algorithms we know!!!

- How might we prove this?

1. Consider all comparison-based algorithms, one-by-one, and analyze them.
2. Don't do that.

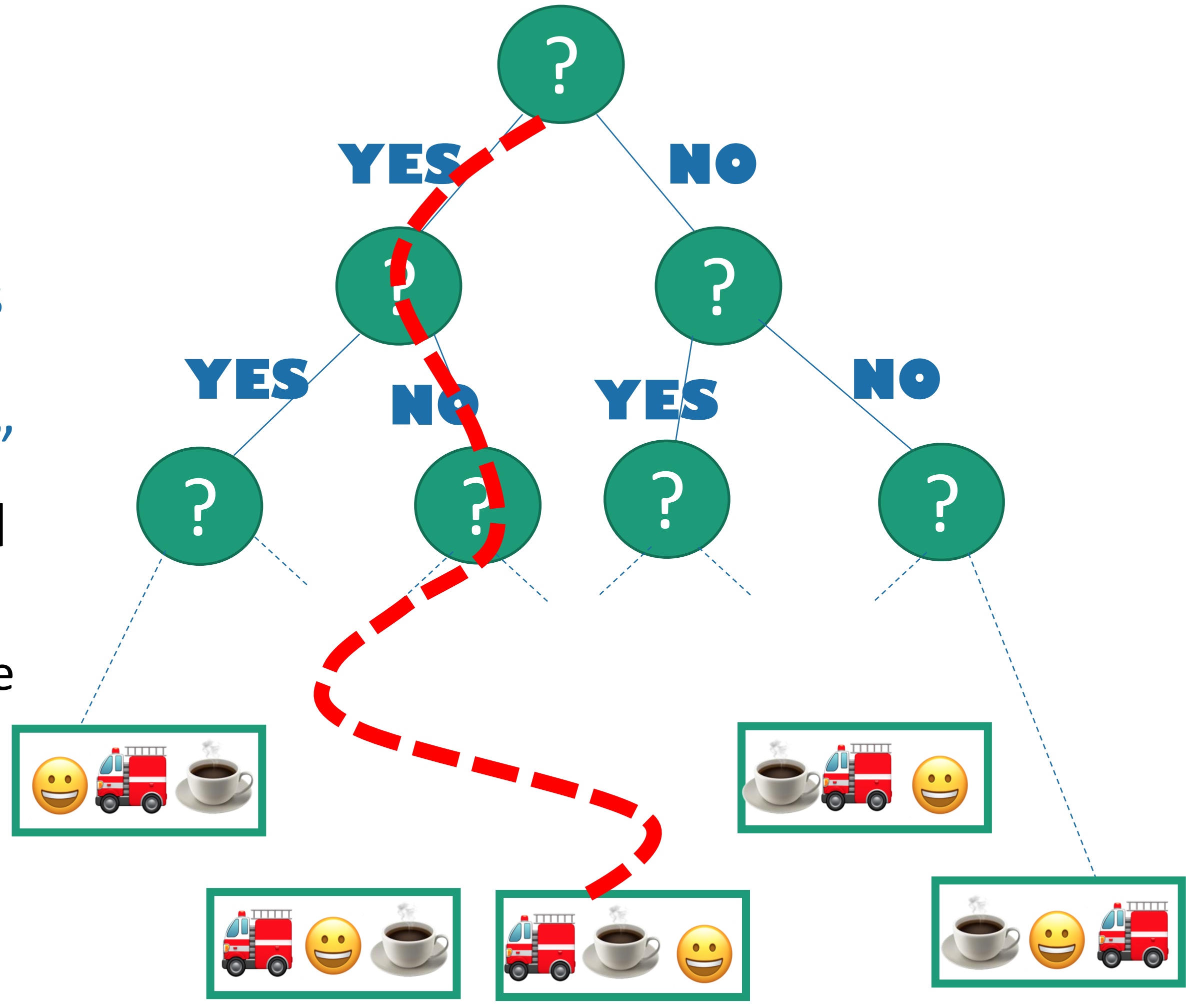
Instead, argue that all comparison-based sorting algorithms give rise to a **decision tree**.
Then analyze decision trees.

Decision trees



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.

Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.

Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.

Pivot!

😀 🚒 ☕

😀 🚒 ☕

Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.

Pivot!

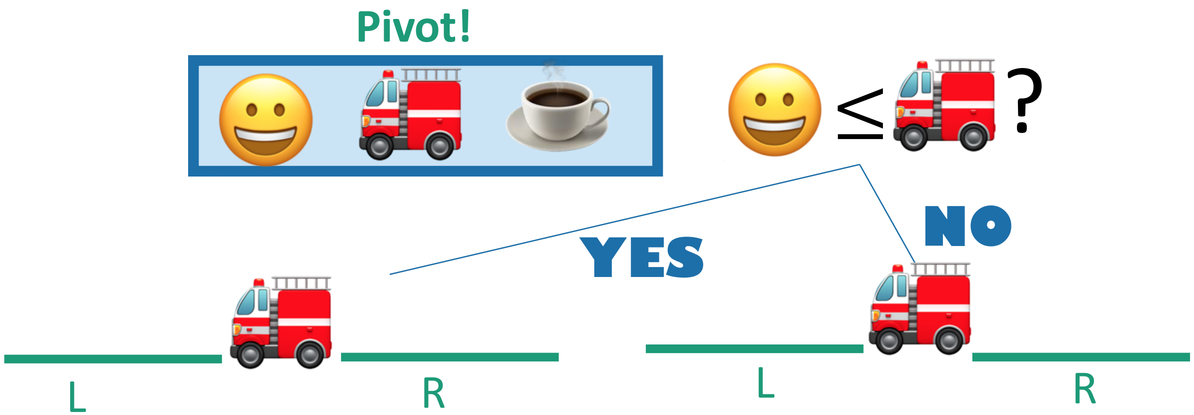
😀 🚒 ☕

😀 ≤ 🚒 ?

😀 🚒 ☕

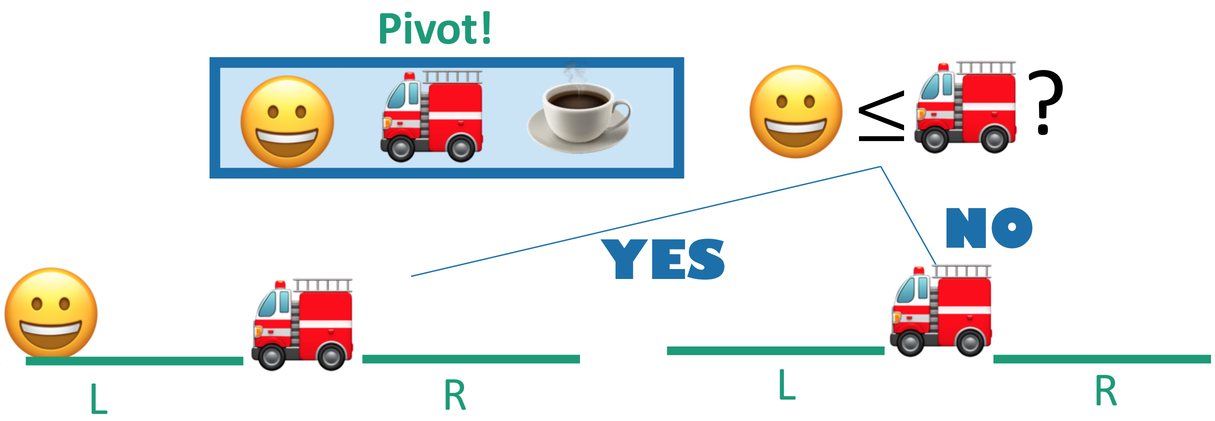
Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.



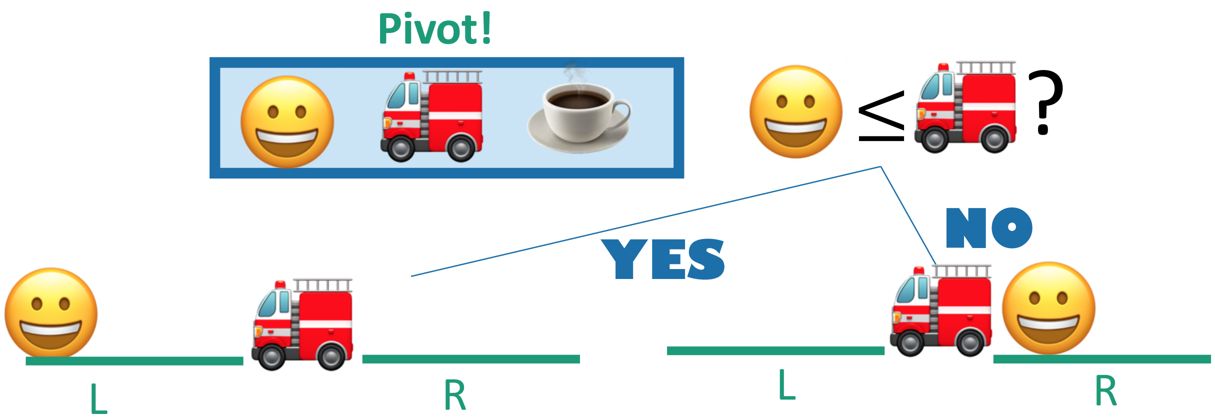
Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.



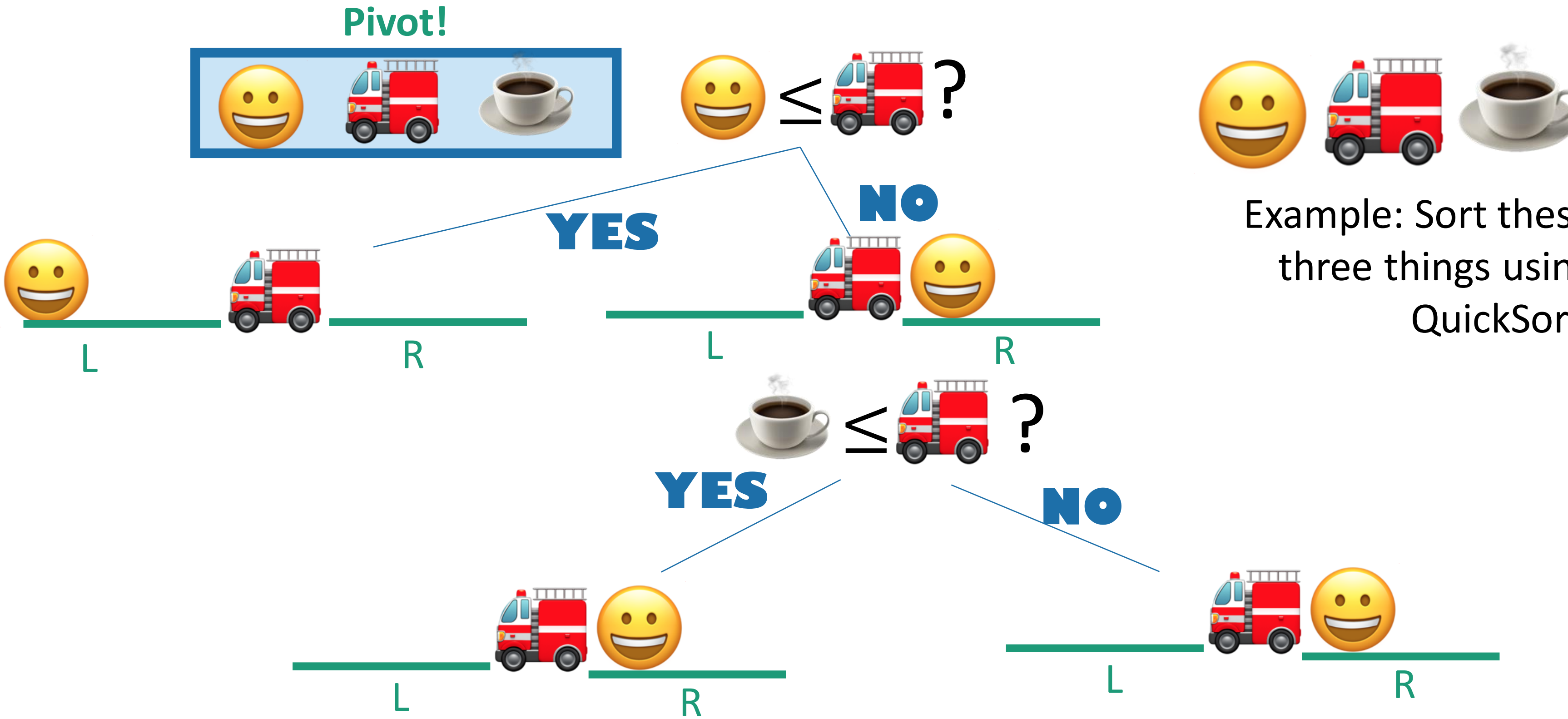
Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.



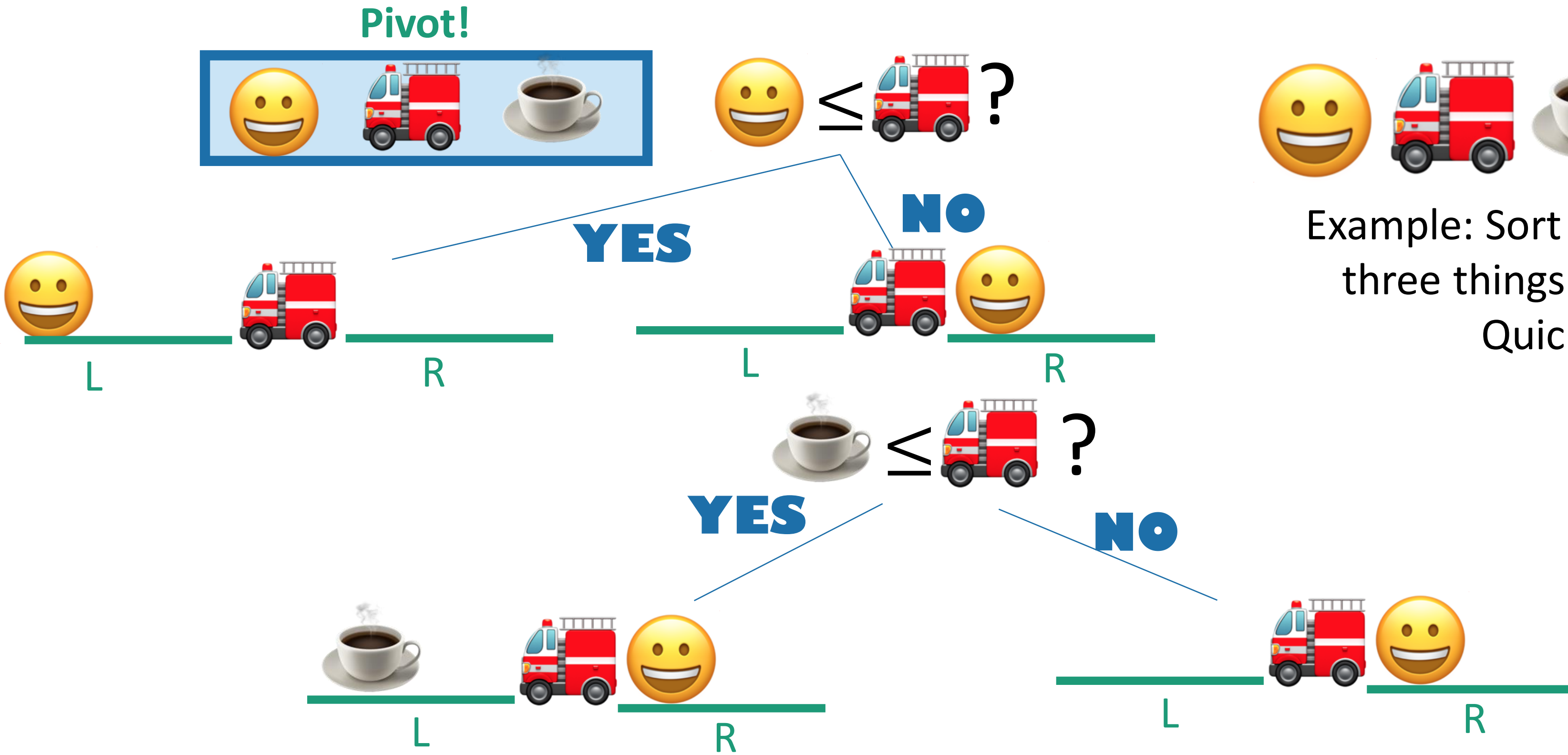
Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.



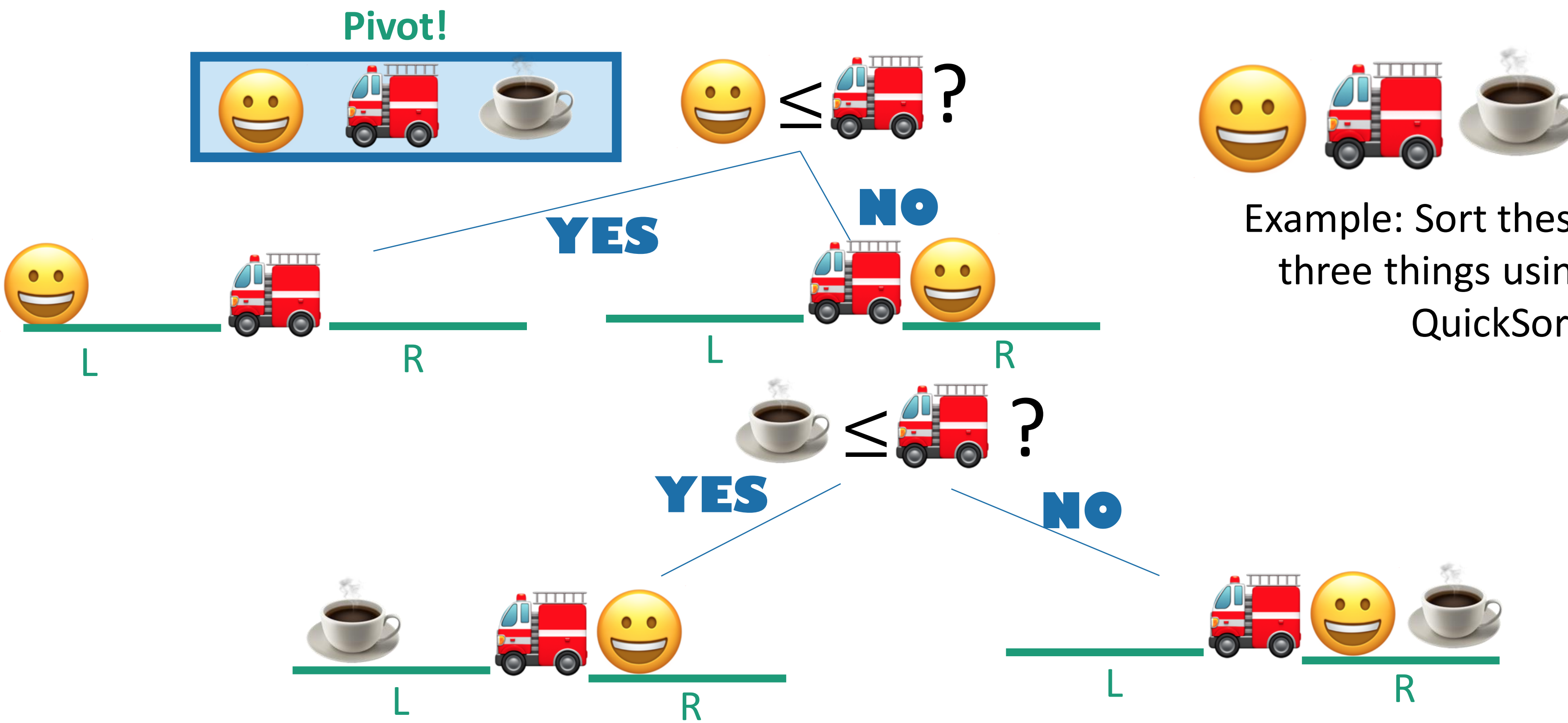
Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.



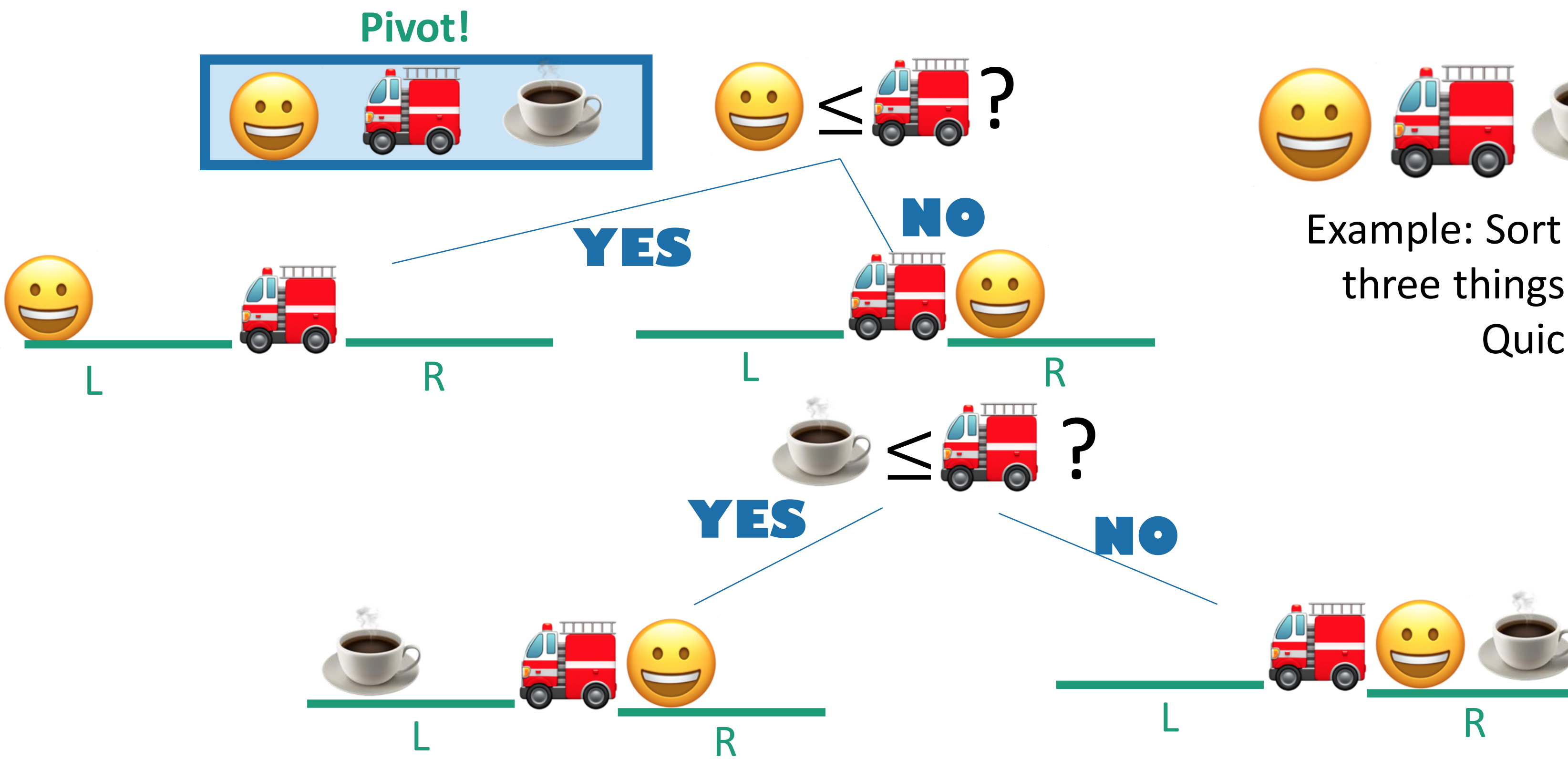
Example: Sort these three things using QuickSort.

Comparison-based algorithms look like decision trees.



Example: Sort these three things using QuickSort.

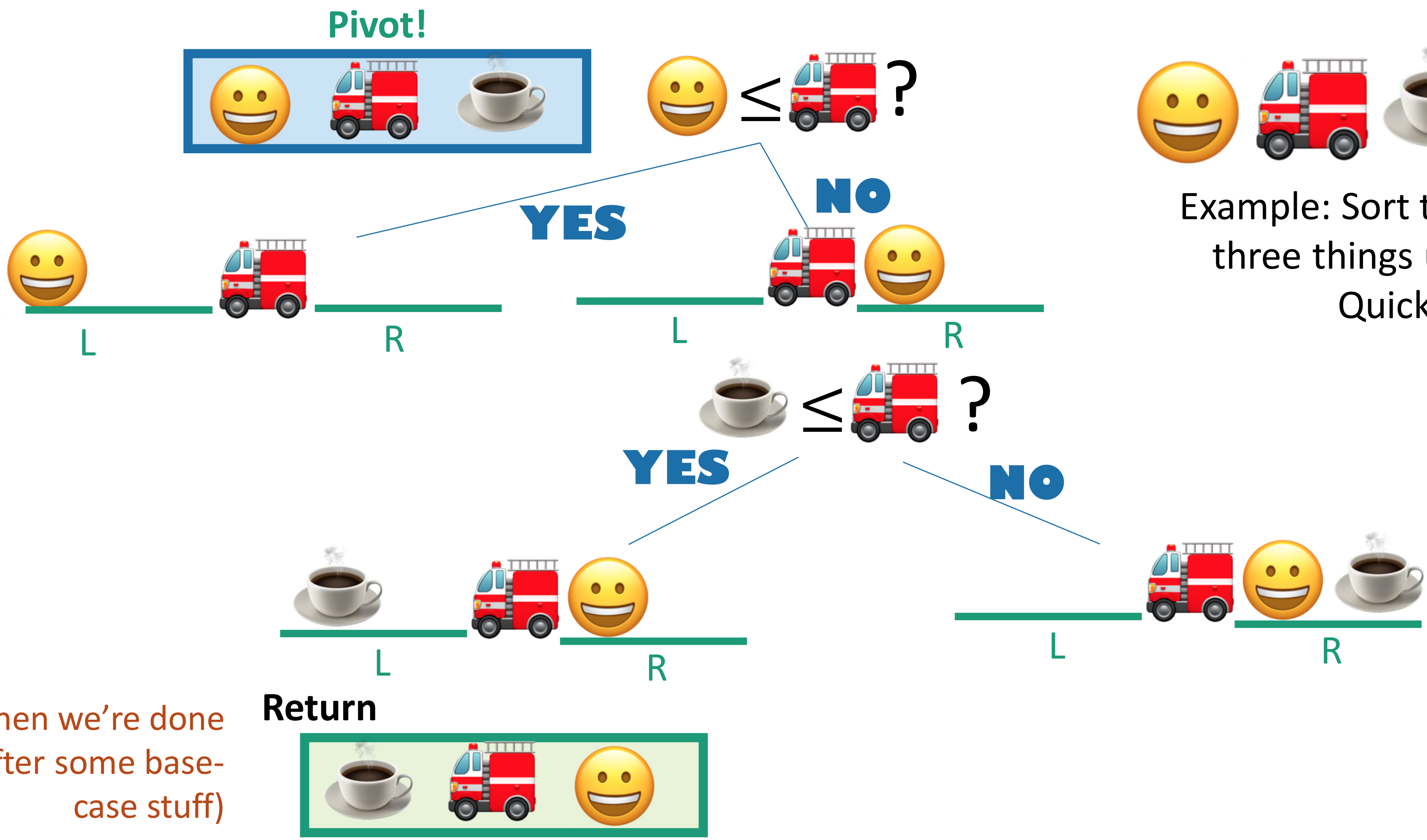
Comparison-based algorithms look like decision trees.



Example: Sort these three things using QuickSort.

Then we're done (after some base-case stuff)

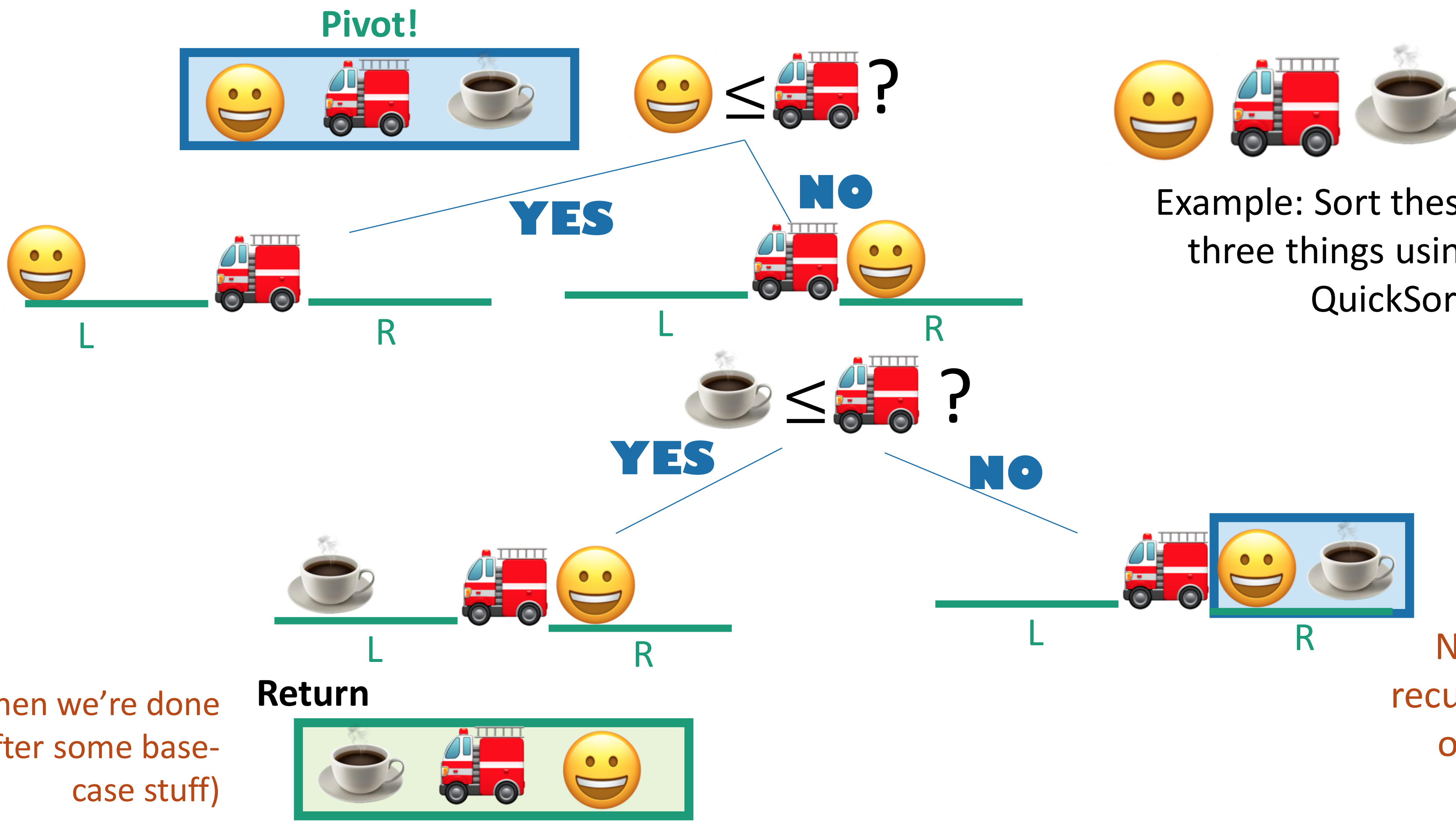
Comparison-based algorithms look like decision trees.



Example: Sort these three things using QuickSort.

Then we're done (after some base-case stuff)

Comparison-based algorithms look like decision trees.

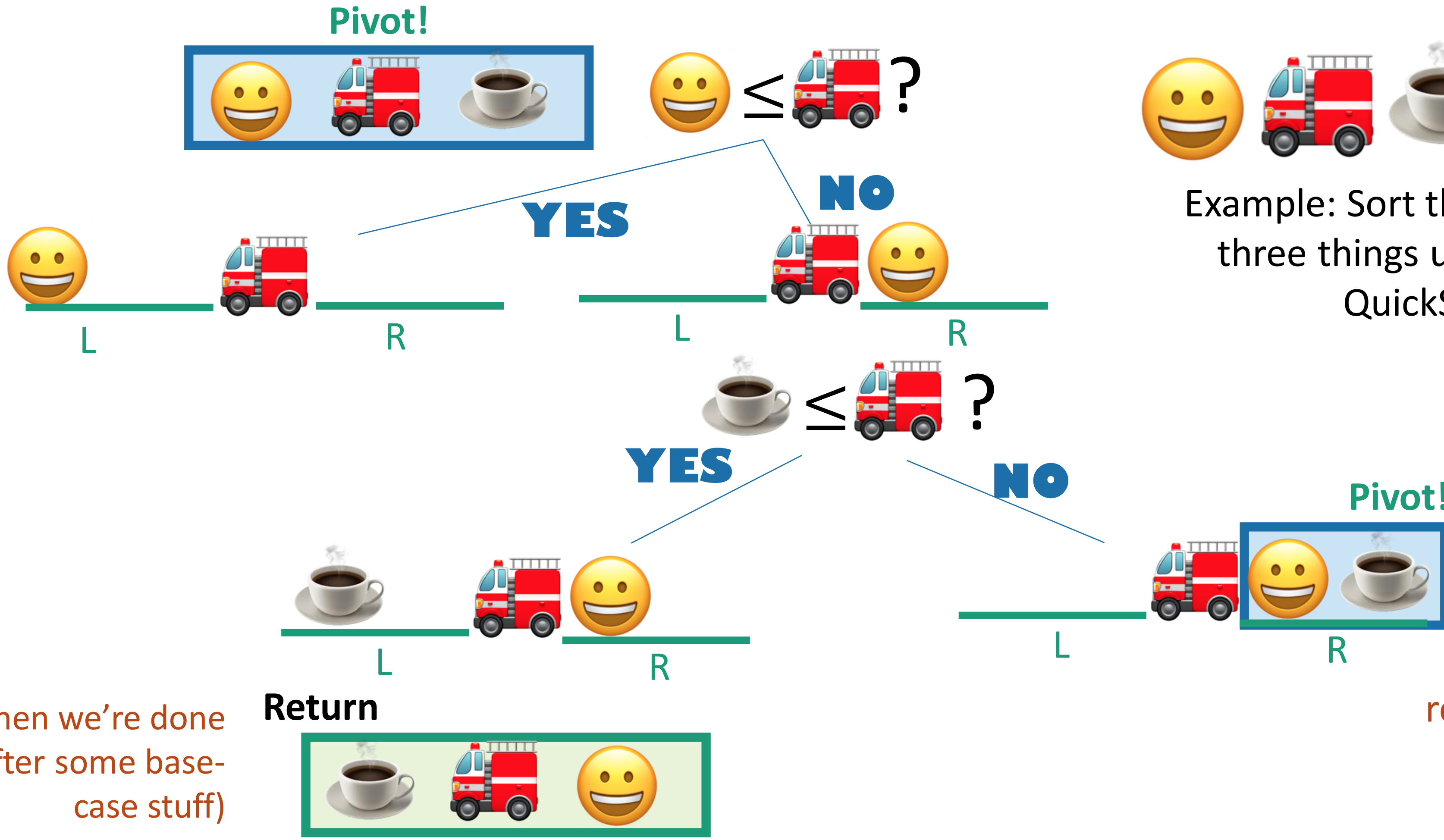


Example: Sort these three things using QuickSort.

Now recurse on R

Then we're done (after some base-case stuff)

Comparison-based algorithms look like decision trees.

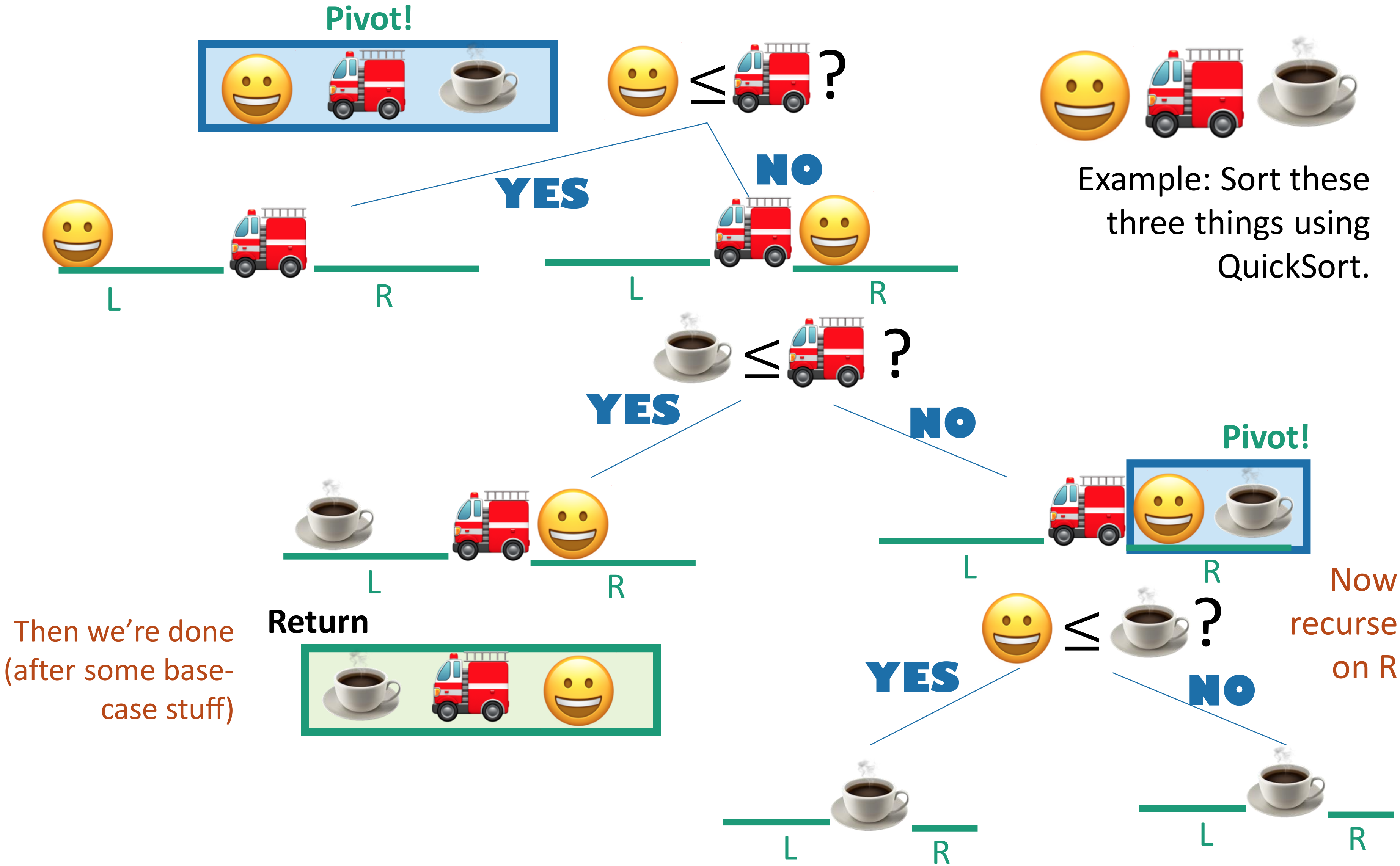


Example: Sort these three things using QuickSort.

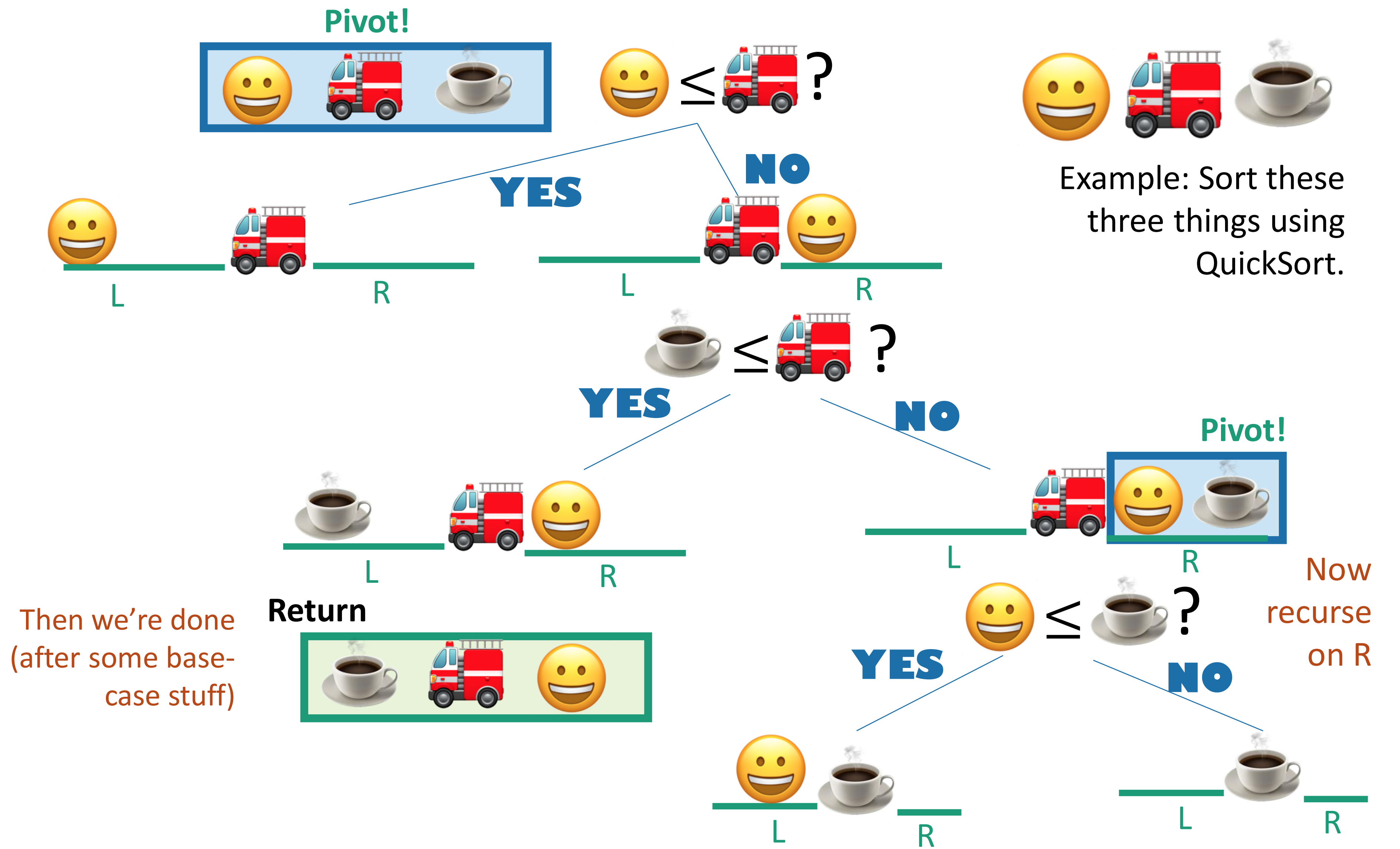
Then we're done (after some base-case stuff)

Now recurse on R

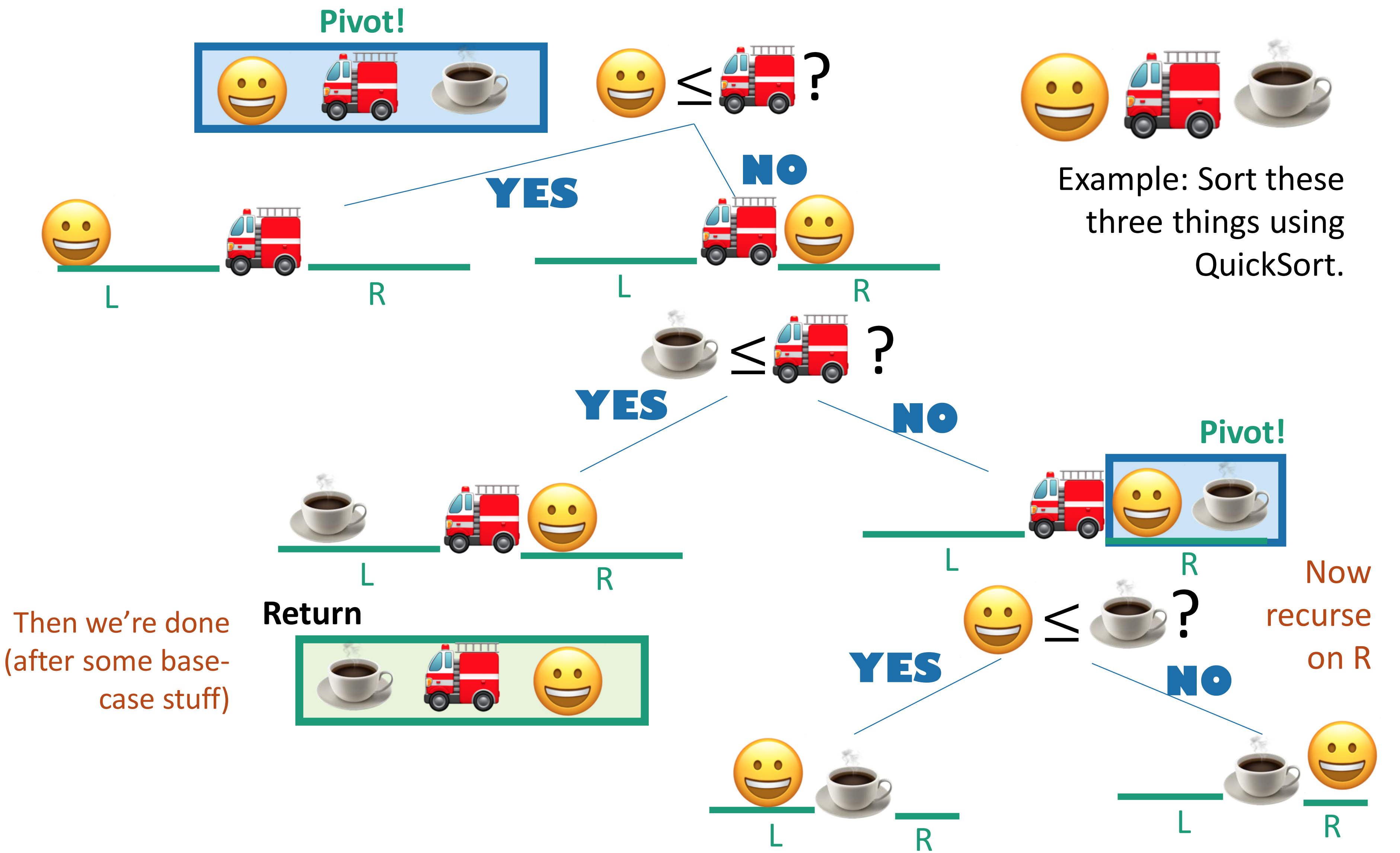
Comparison-based algorithms look like decision trees.



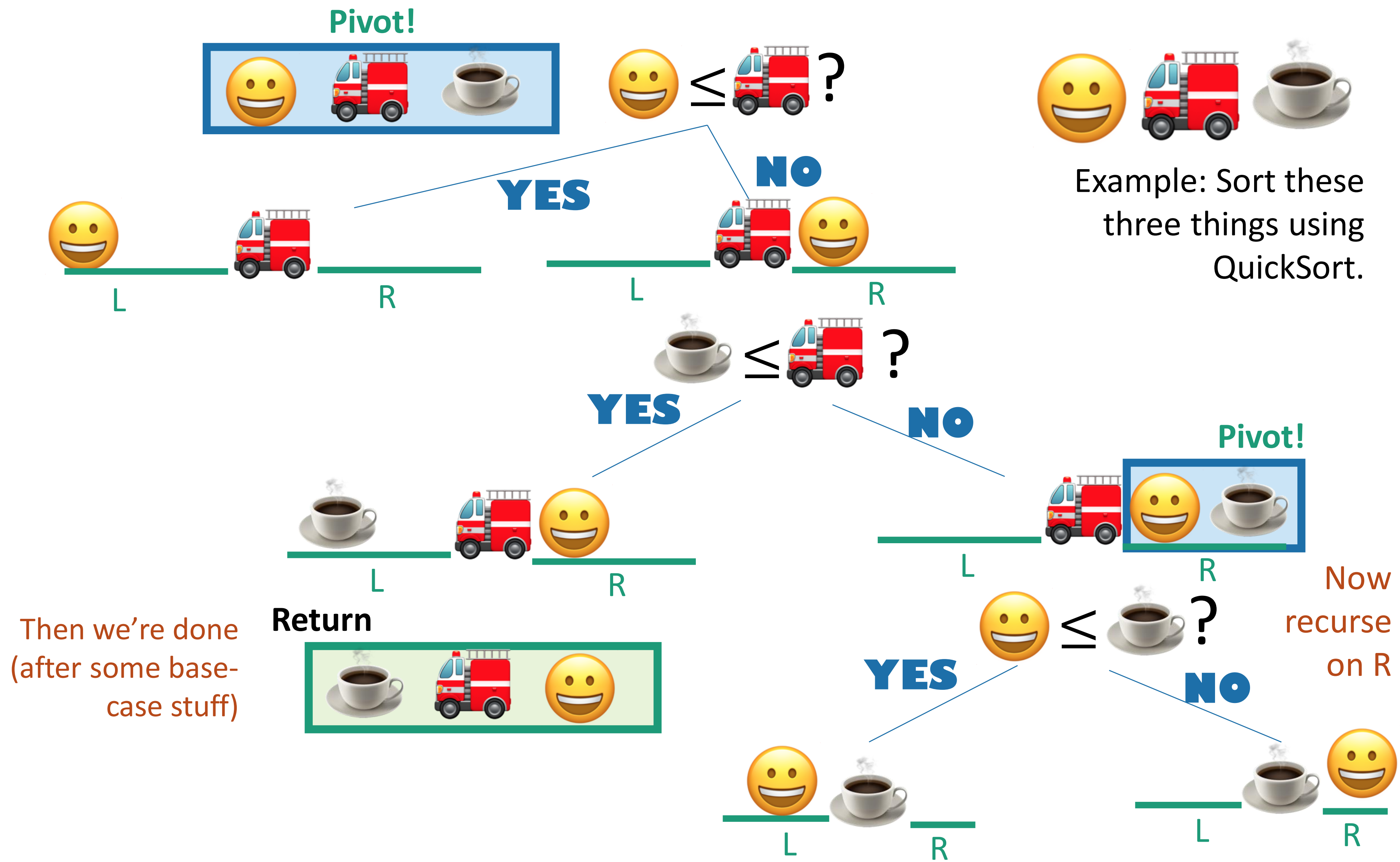
Comparison-based algorithms look like decision trees.



Comparison-based algorithms look like decision trees.

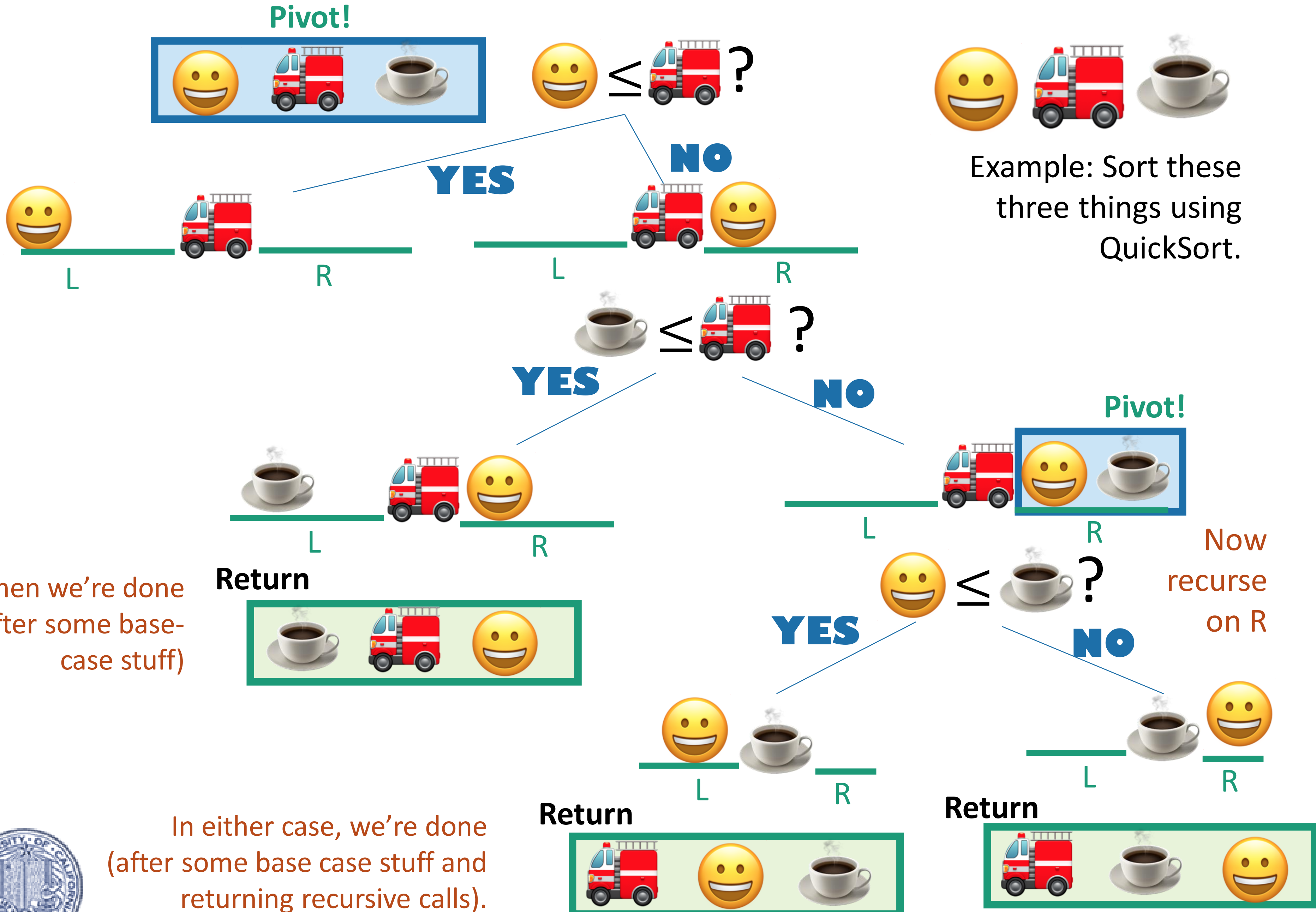


Comparison-based algorithms look like decision trees.



In either case, we're done (after some base case stuff and returning recursive calls).

Comparison-based algorithms look like decision trees.



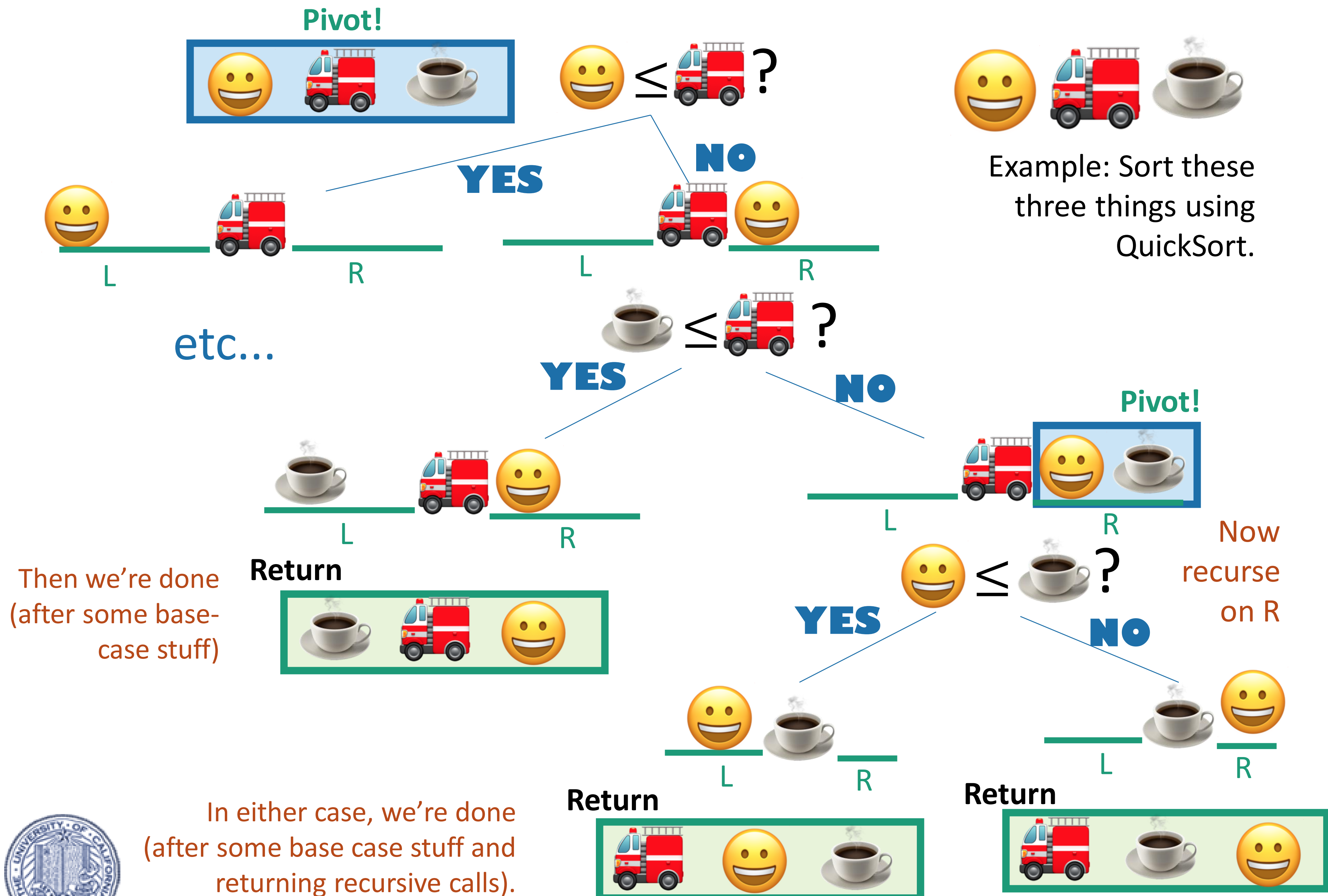
Example: Sort these three things using QuickSort.

Then we're done (after some base-case stuff)

Now recurse on R

In either case, we're done (after some base case stuff and returning recursive calls).

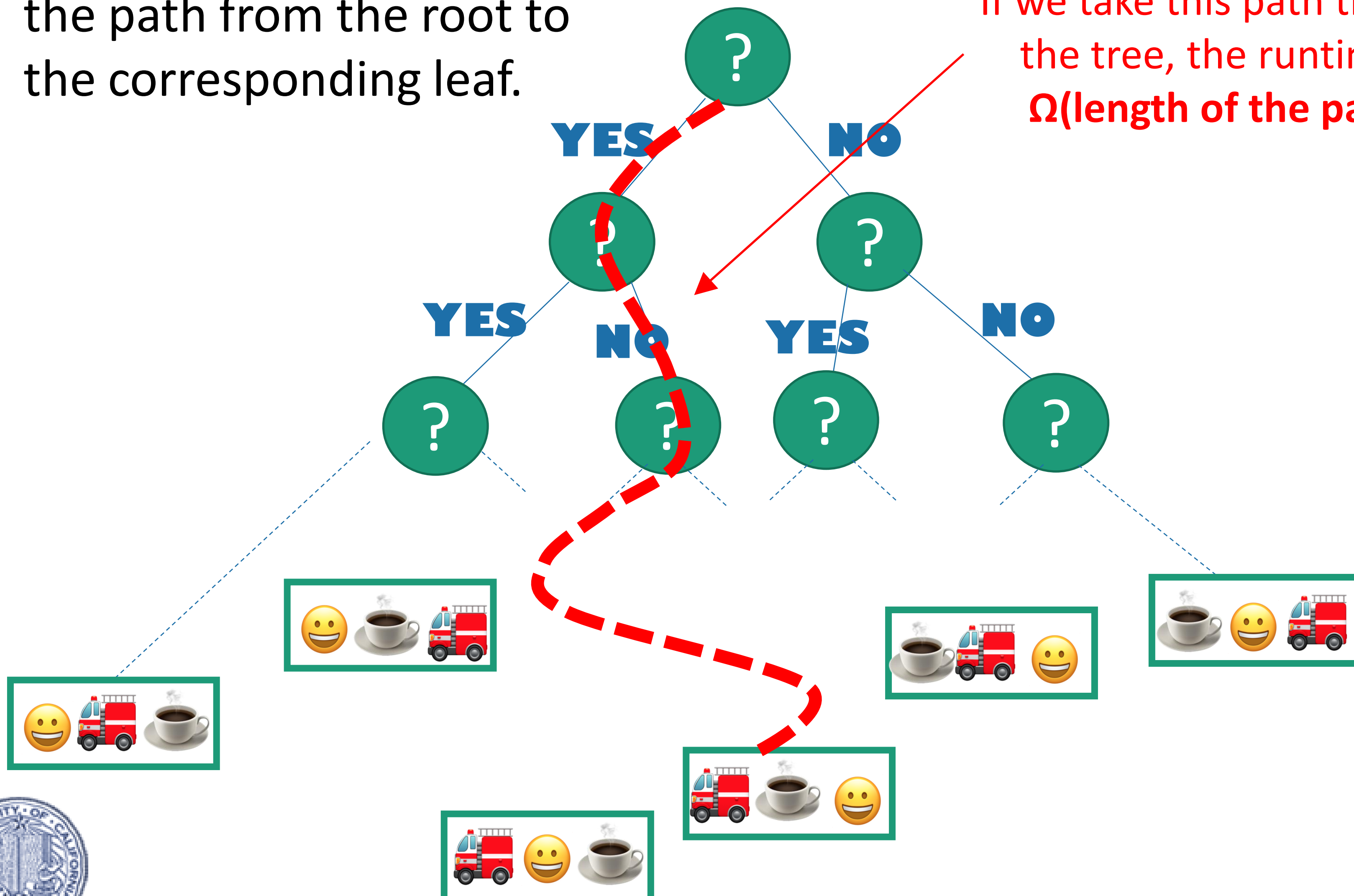
Comparison-based algorithms look like decision trees.



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

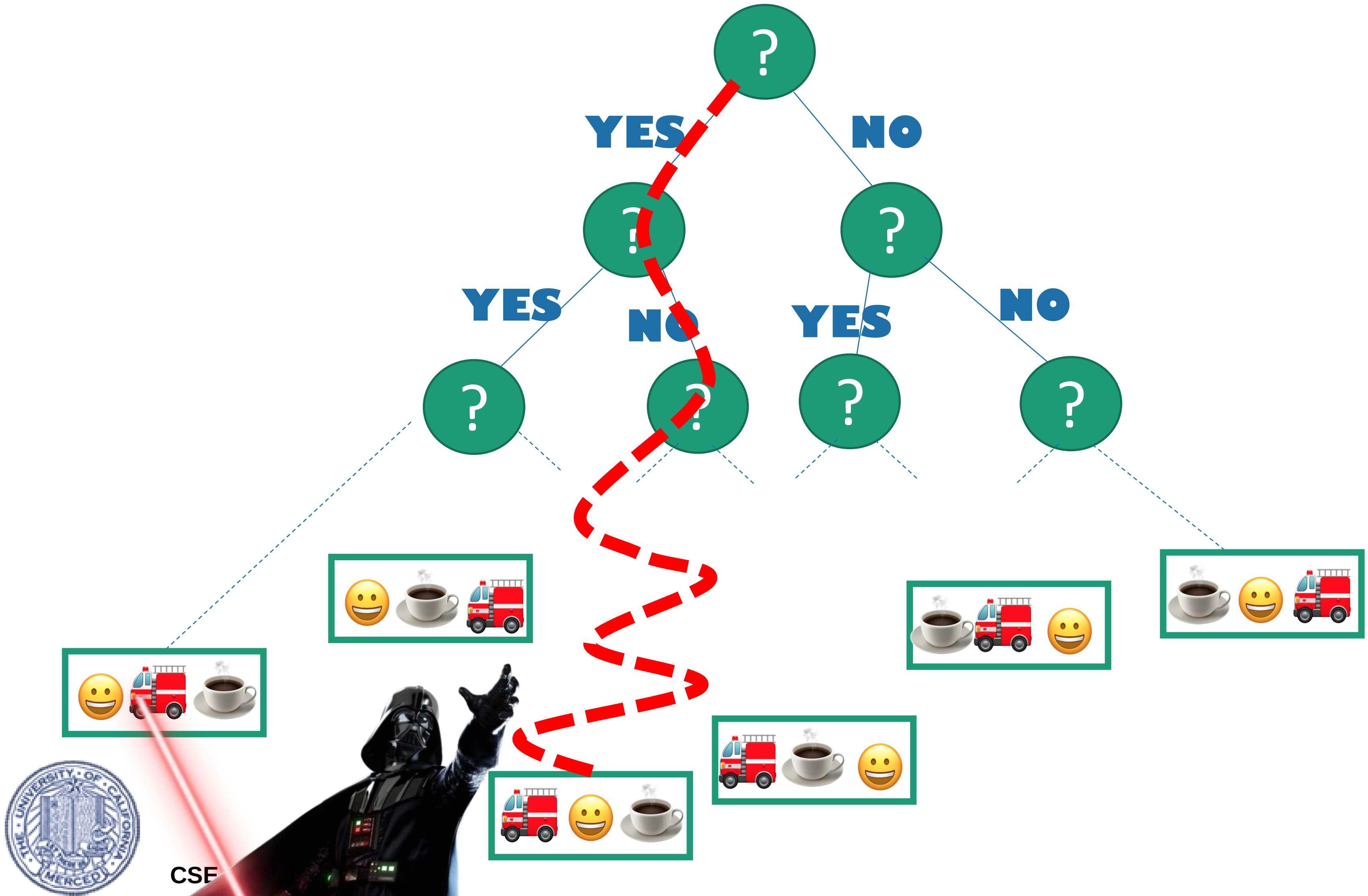
A: At least the length of the path from the root to the corresponding leaf.

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



Q: What's the worst-case runtime?

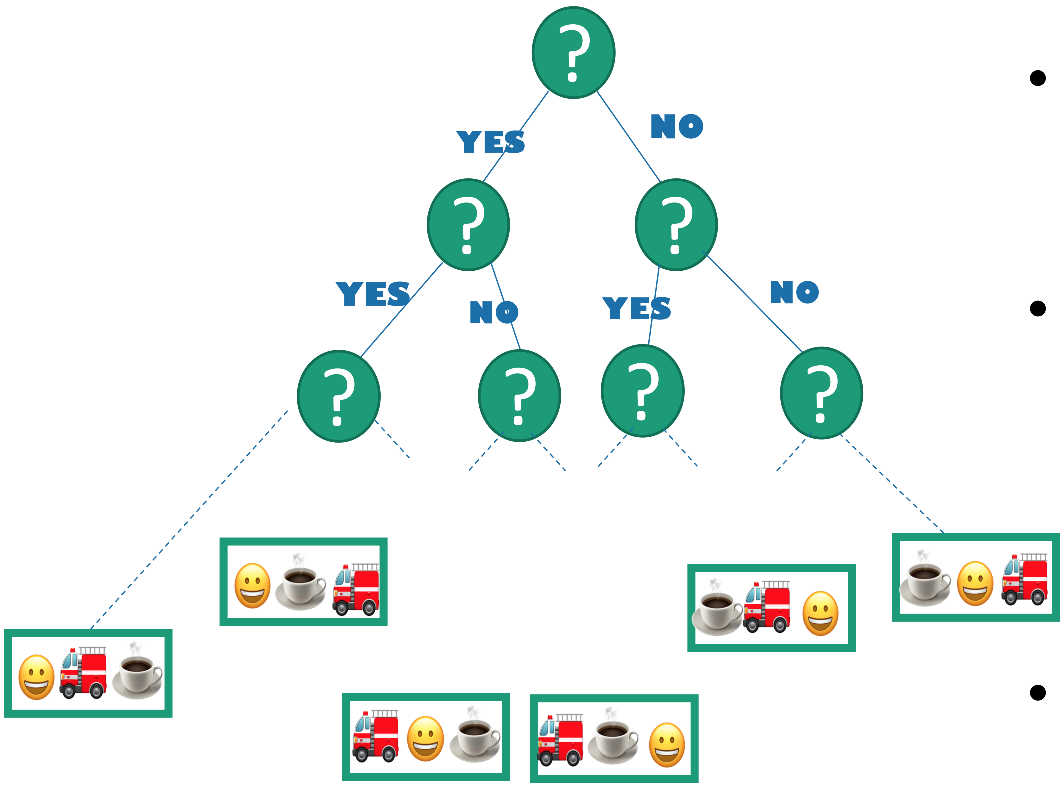
A: At least Ω(length of the longest path).



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 ______



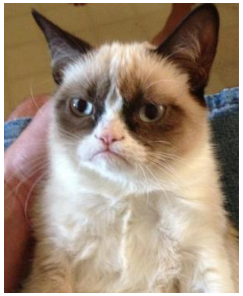
- This is a binary tree with at least __n!__ leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth _log(n!)_.
- So in all such trees, the longest path is at least log(n!).

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



- Theorem:
  - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

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

Aside:
What about randomized algorithms?

- For example, QuickSort?
- Theorem:

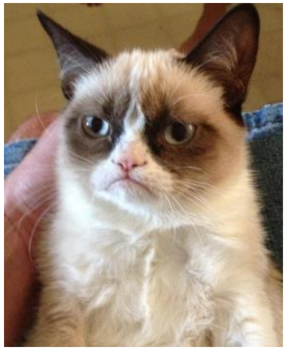
  - Any randomized comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps in expectation.
- Proof:
  - see reading posted on website
    - (Avrim Blum's notes)
  - (same ideas as deterministic case)

Try to prove this yourself!

Ollie the over-achieving ostrich

\end{Aside}

So that's bad news

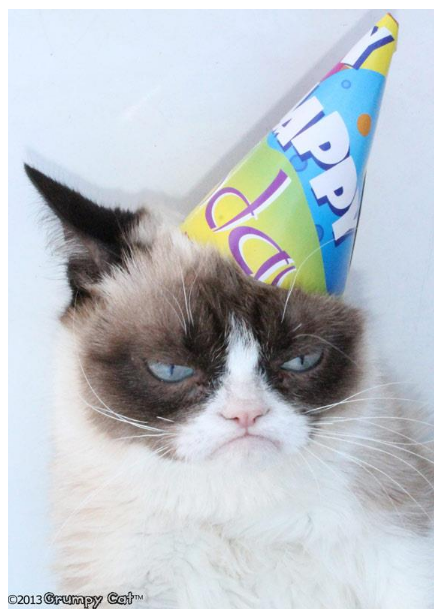


- Theorem:
  - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

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

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)?

Beyond comparison-based sorting algorithms

Another model of computation

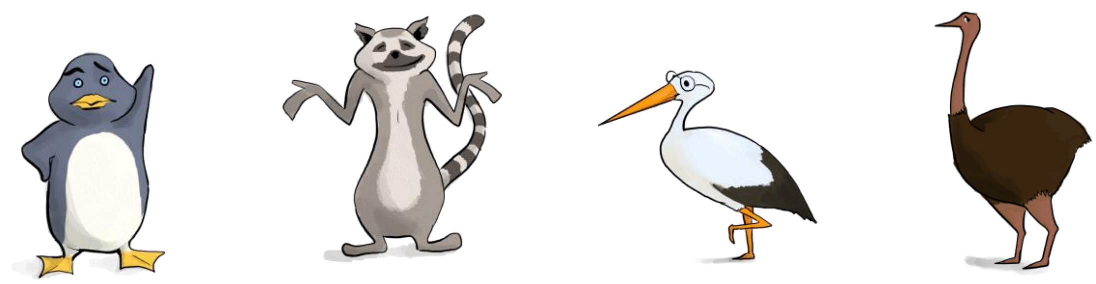
- The items you are sorting have meaningful values.

9	6	3	5	2	1	2

instead of

Practice 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?

BucketSort:

Note: this is a simplification of
what CLRS calls “BucketSort”

Why might this help?

BucketSort:

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

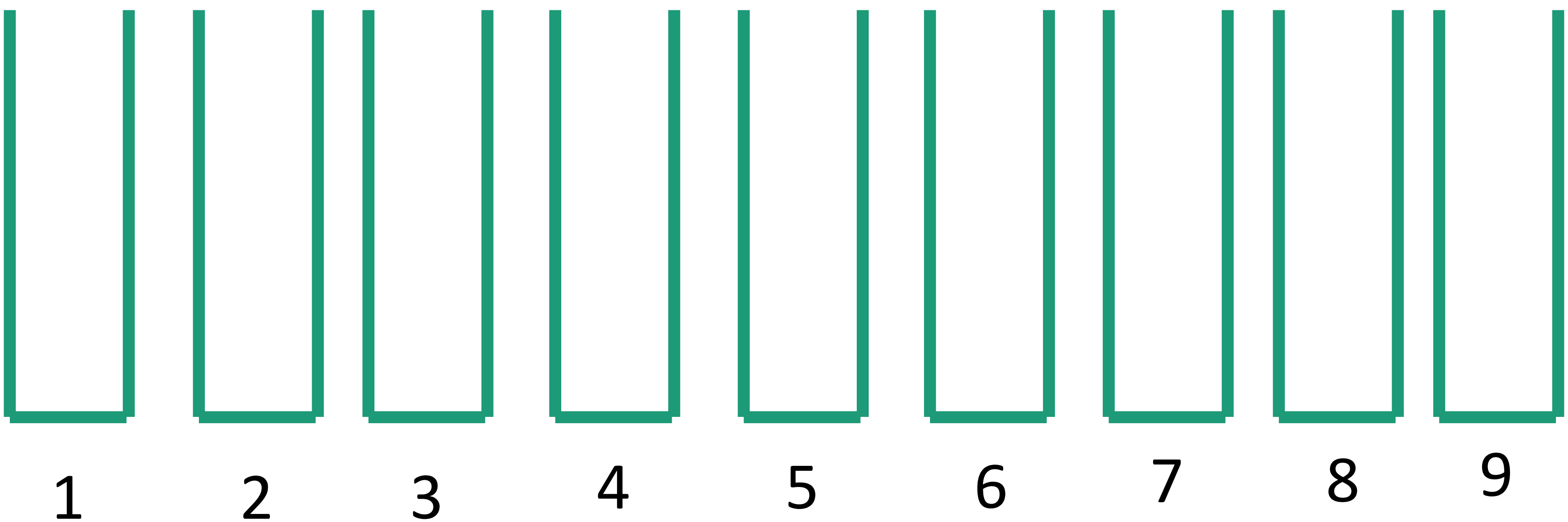
9	6	3	5	2	1	2

Why might this help?

BucketSort:

Note: this is a simplification of what CLRS calls “BucketSort”

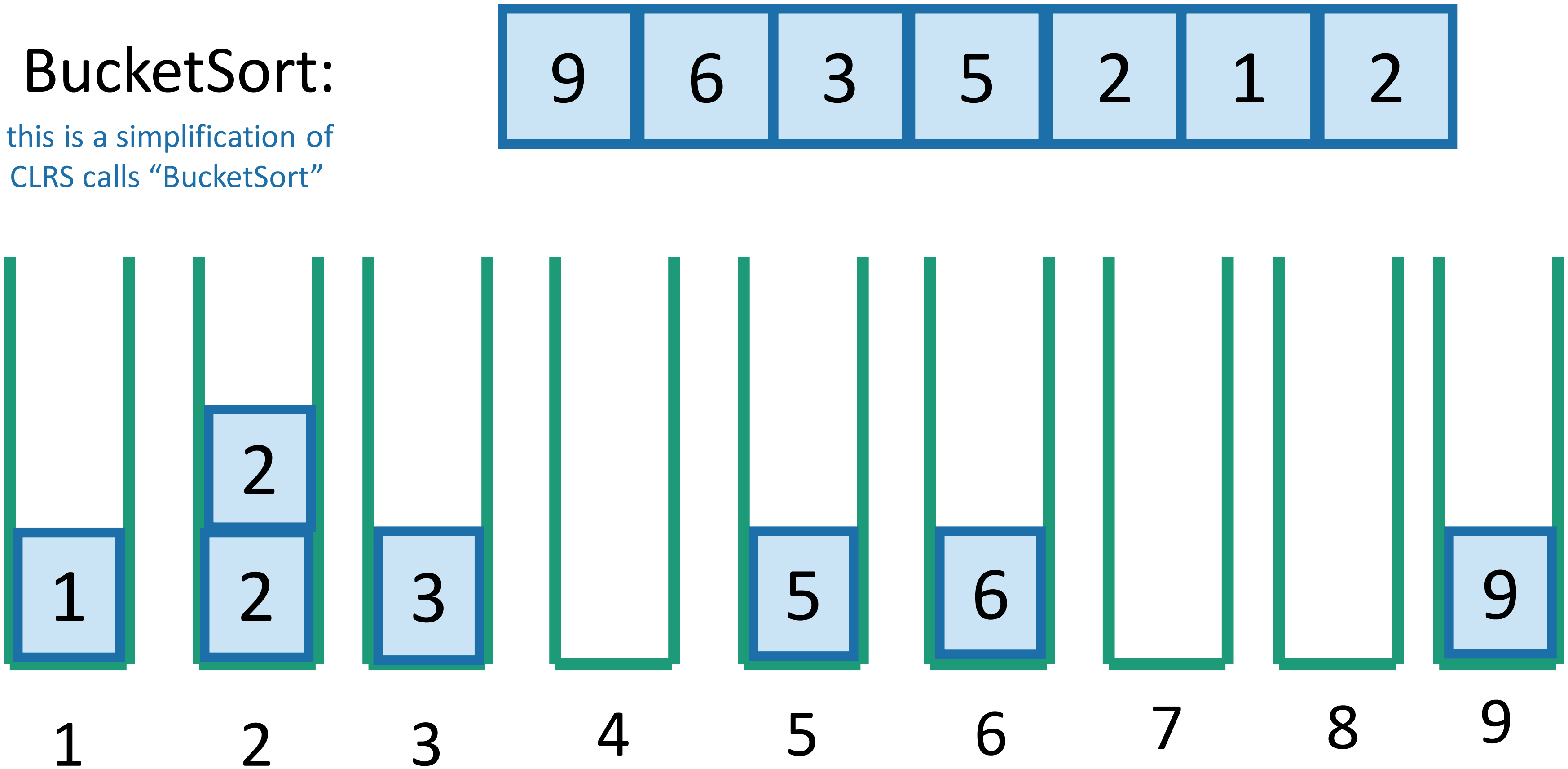
9	6	3	5	2	1	2



Why might this help?

BucketSort:

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



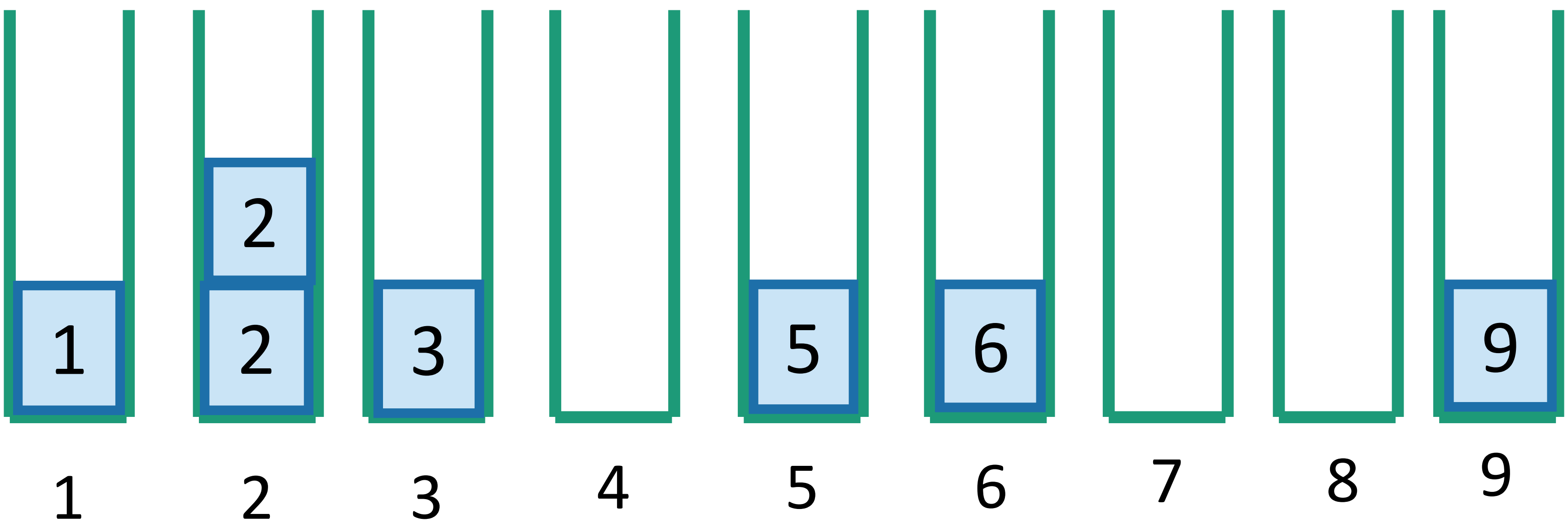
Why might this help?

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"

9	6	3	5	2	1	2

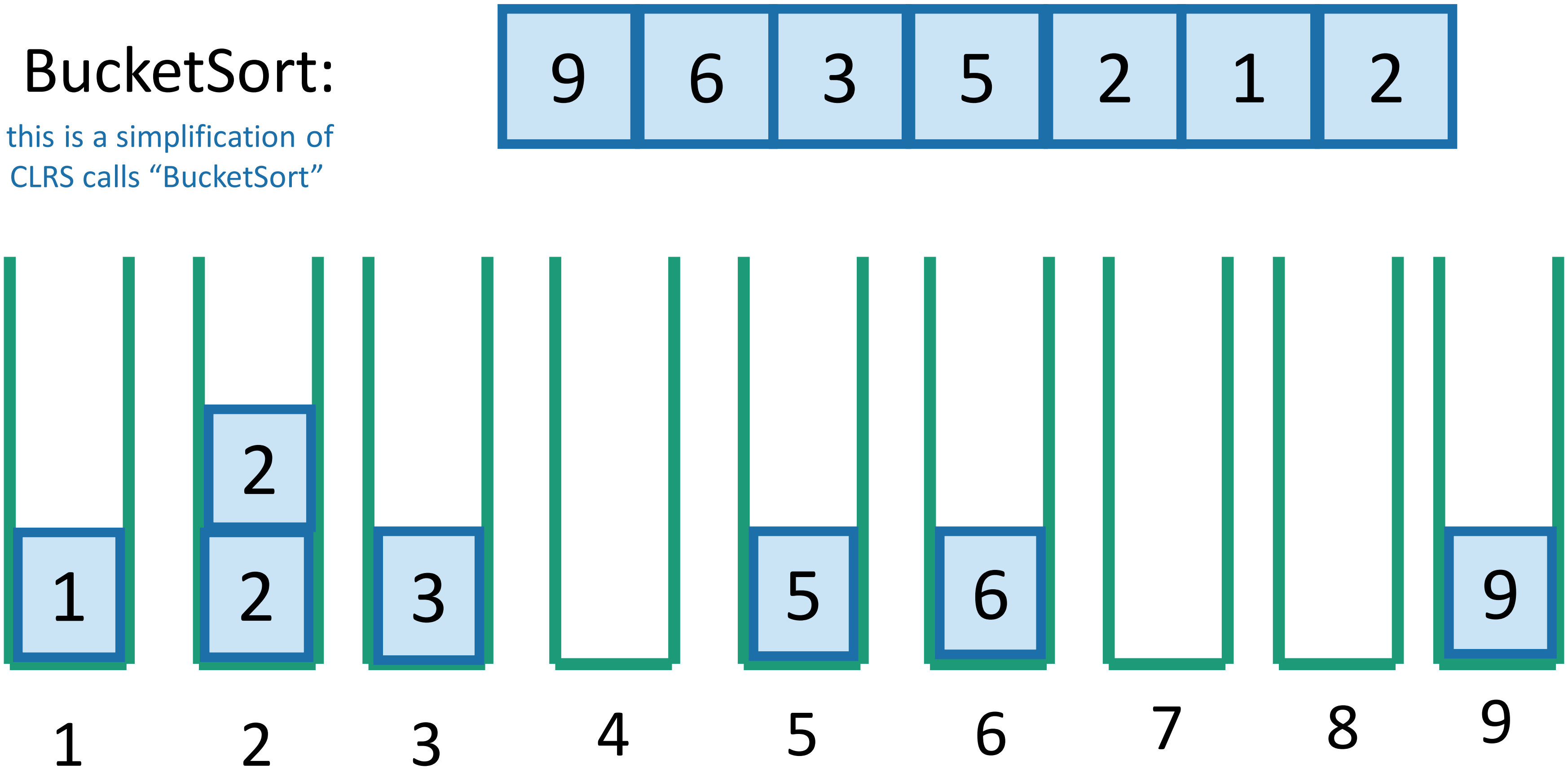


Why might this help?

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"

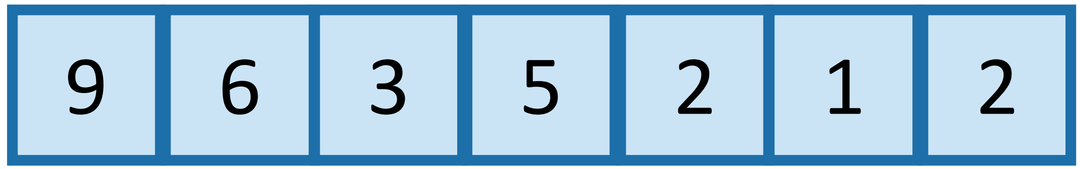


Concatenate the buckets!

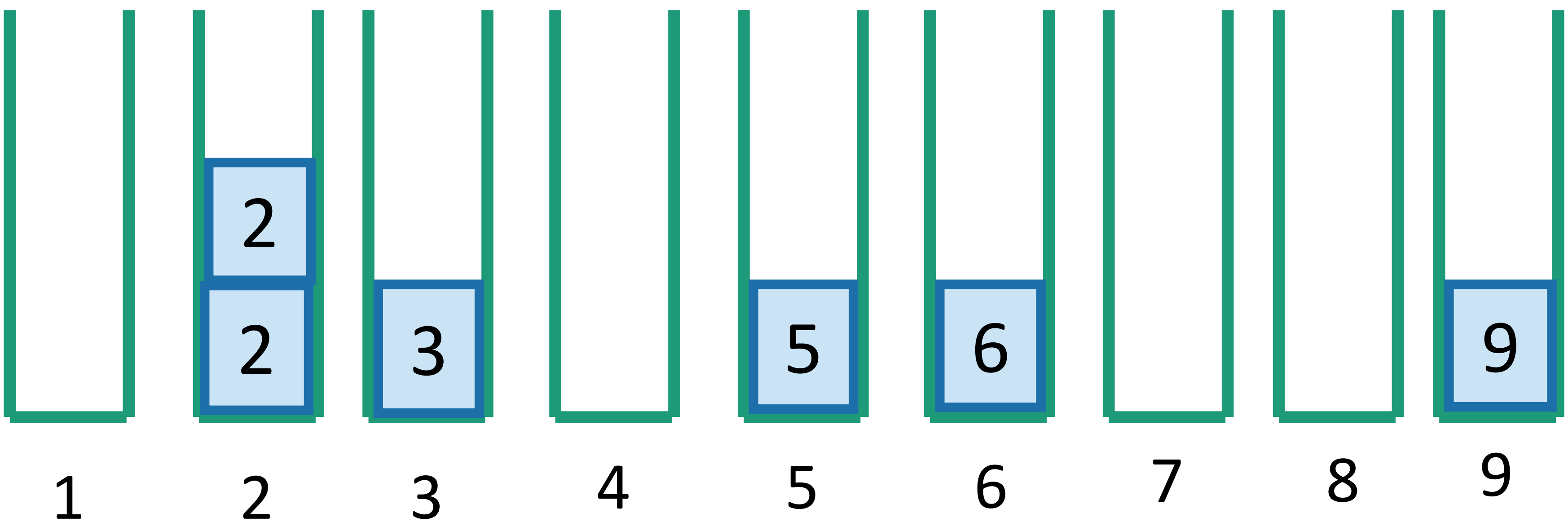
Why might this help?

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

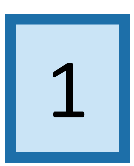
BucketSort:



Note: this is a simplification of what CLRS calls “BucketSort”



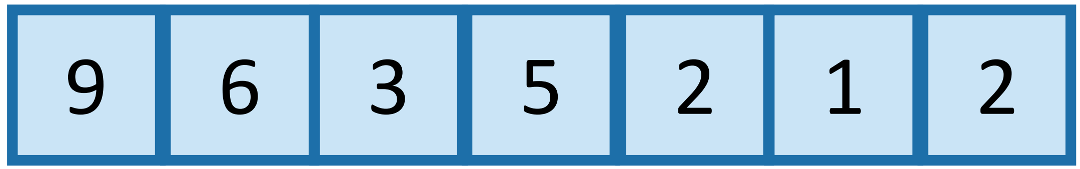
Concatenate the buckets!



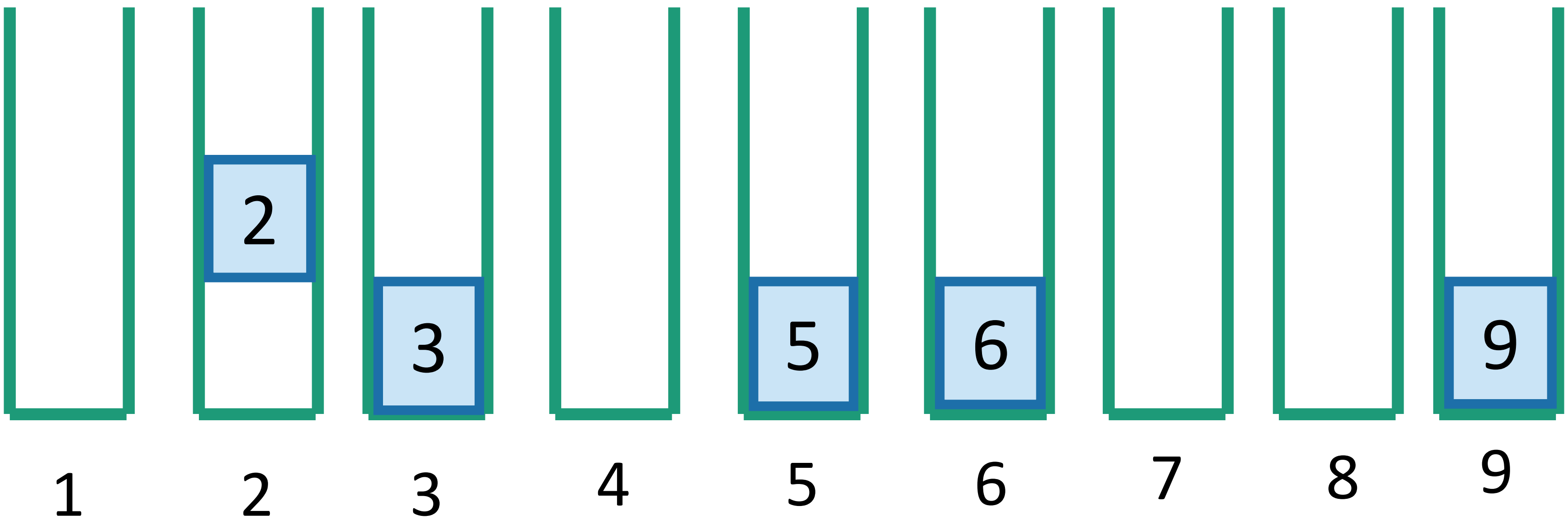
Why might this help?

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

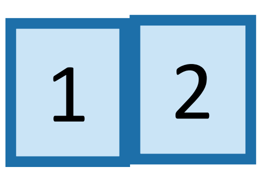
BucketSort:



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



Concatenate the buckets!



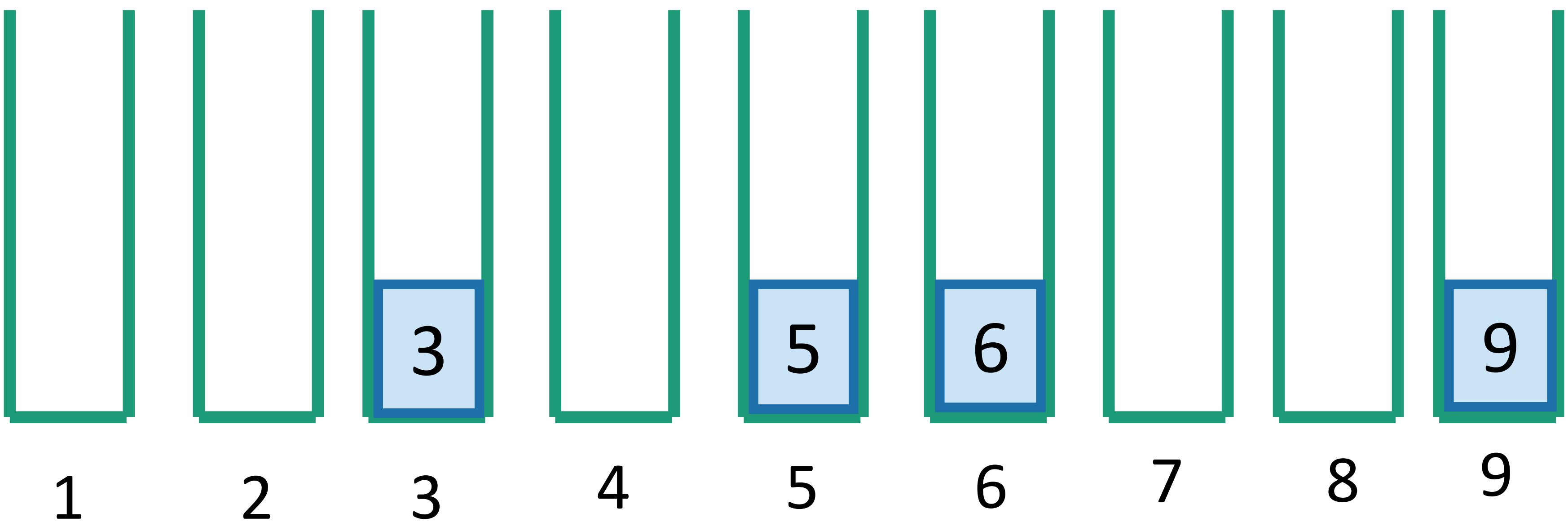
Why might this help?

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!

1	2	2

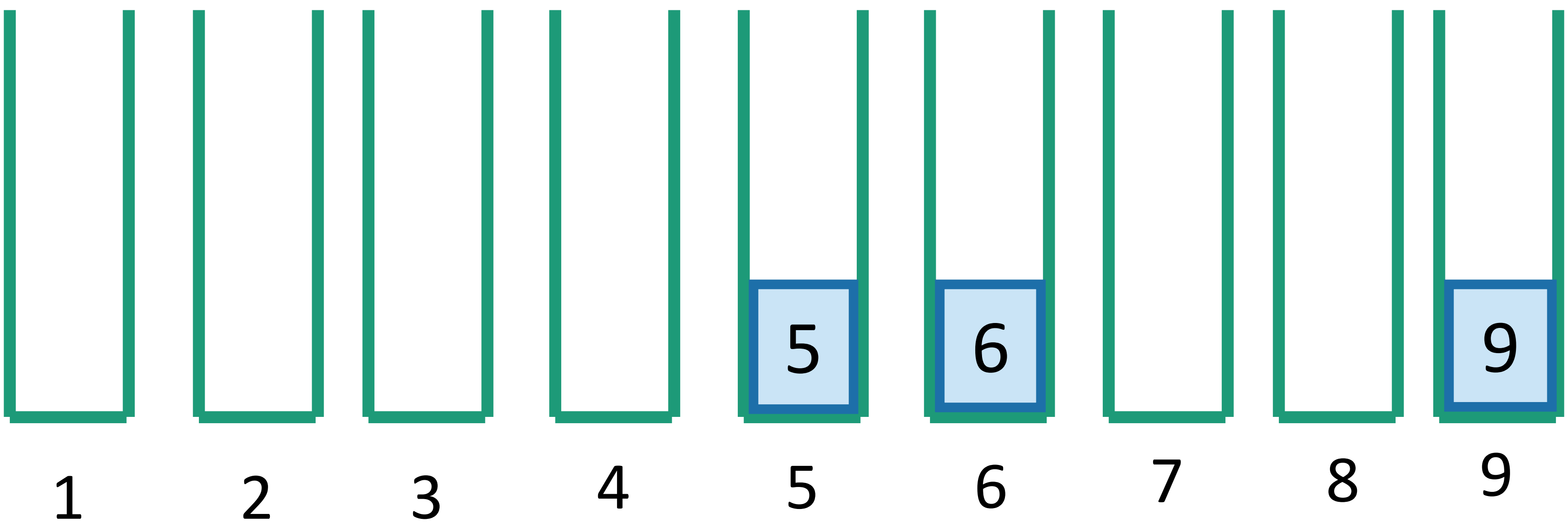
Why might this help?

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!

1	2	2	3

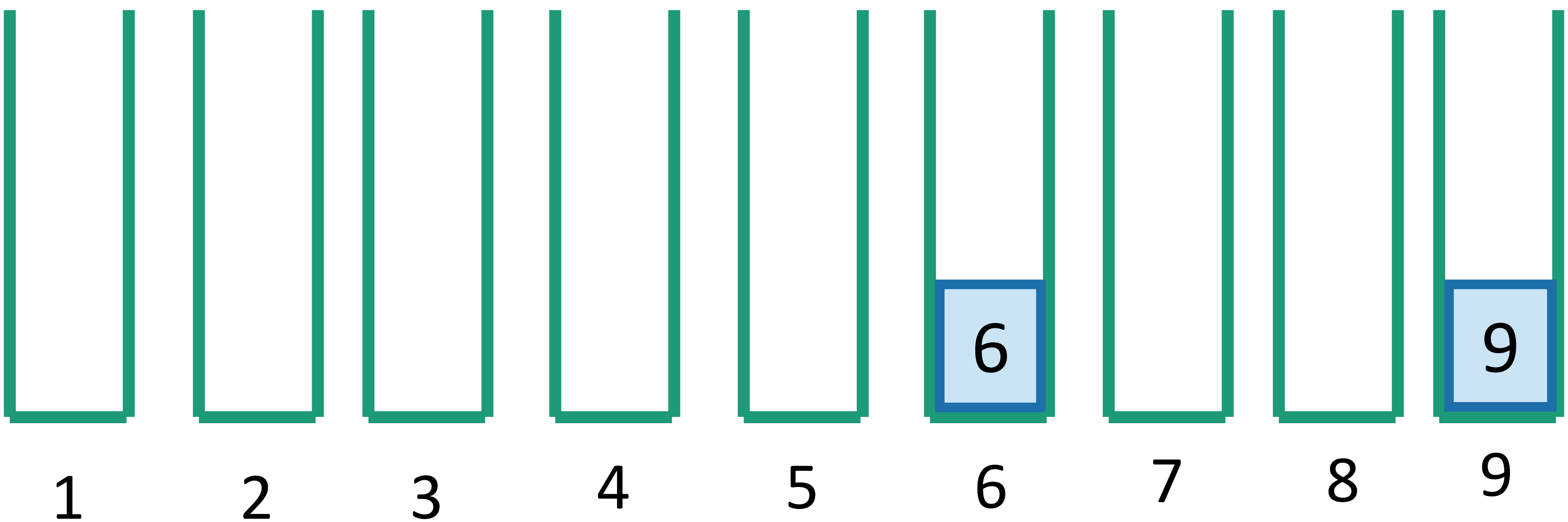
Why might this help?

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"

9	6	3	5	2	1	2



Concatenate the buckets!

1	2	2	3	5

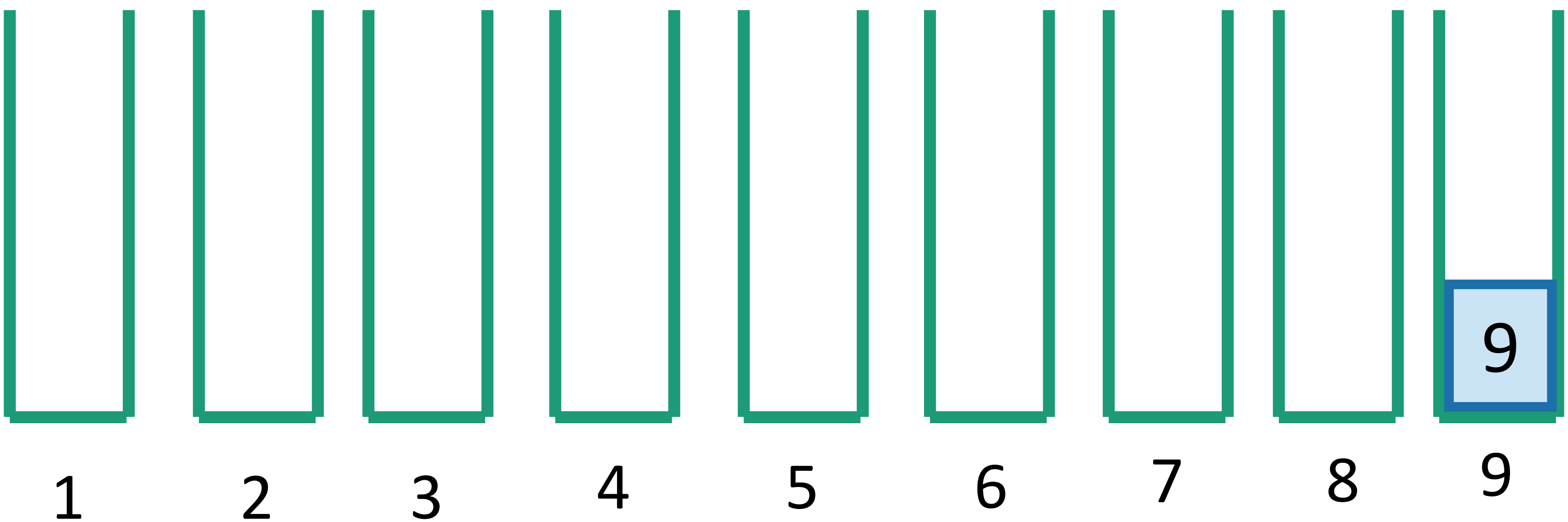
Why might this help?

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”

9	6	3	5	2	1	2



Concatenate the buckets!

1	2	2	3	5	6

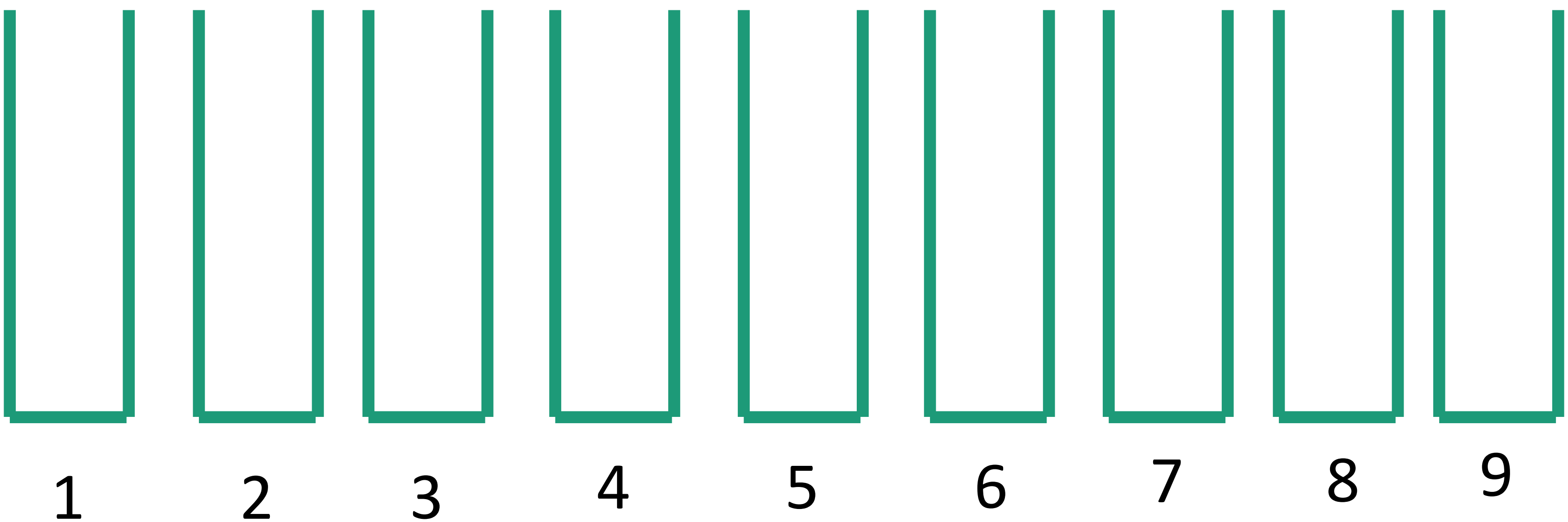
Why might this help?

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”

9	6	3	5	2	1	2



Concatenate the buckets!

1	2	2	3	5	6	9

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"

9	6	3	5	2	1	2

1 2 3 4 5 6 7 8 9

Concatenate the buckets!

1	2	2	3	5	6	9

SORTED!

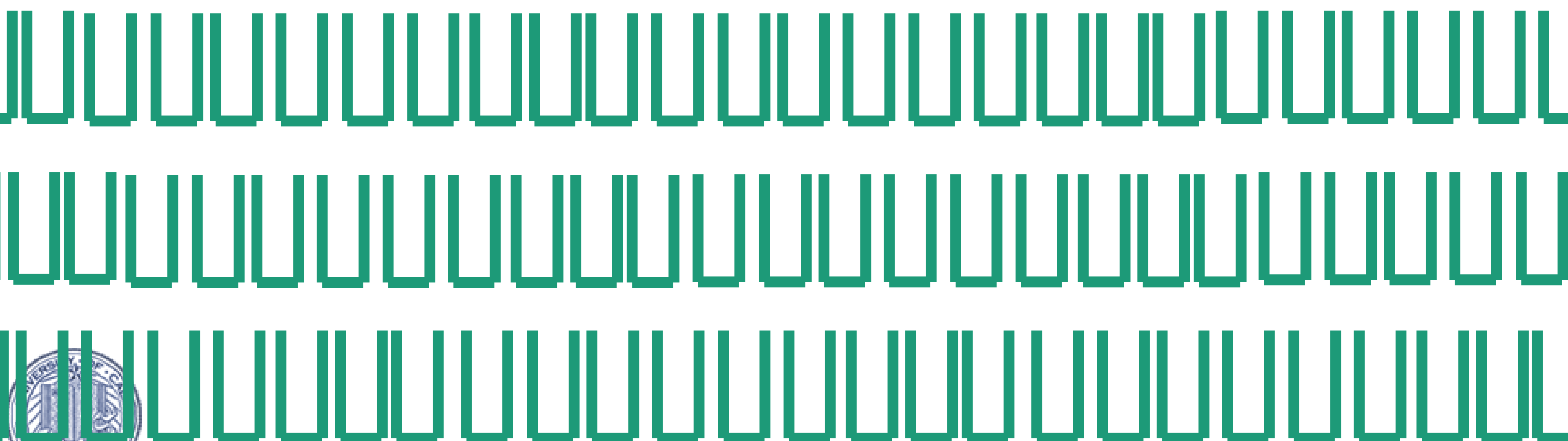
In time O(n).

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.

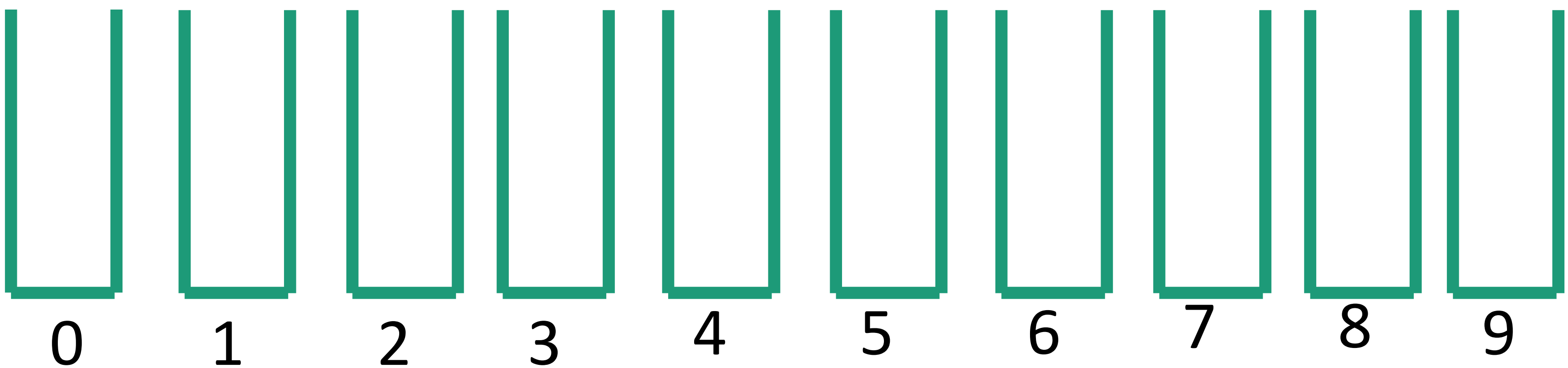
Step 1: BucketSort on least significant digit

Step 1: BucketSort on least significant digit

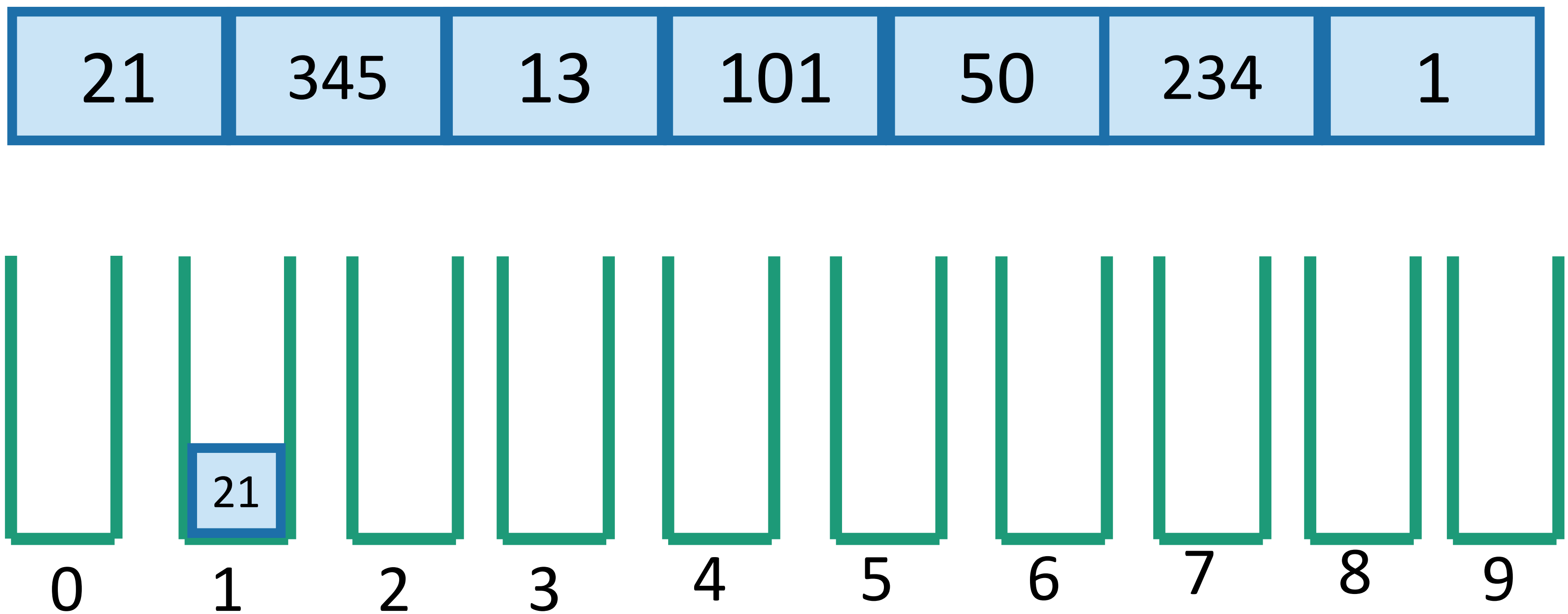
21	345	13	101	50	234	1

Step 1: BucketSort on least significant digit

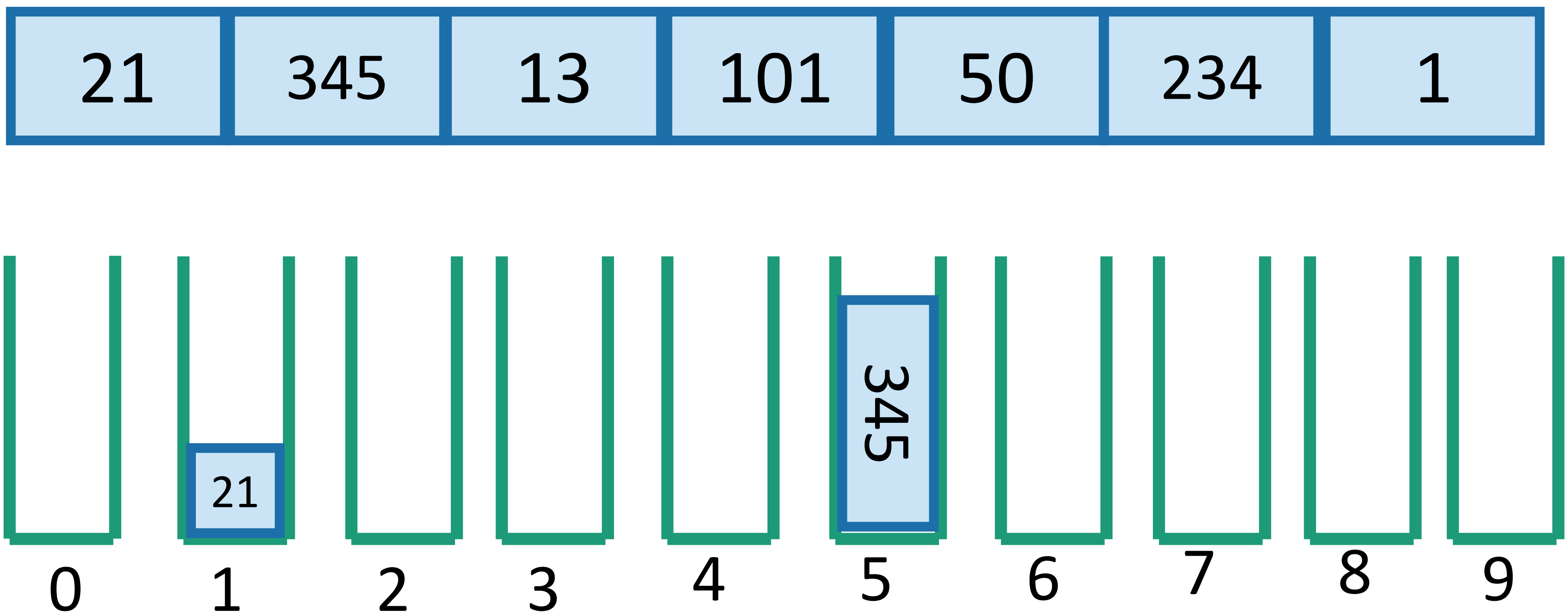
21	345	13	101	50	234	1



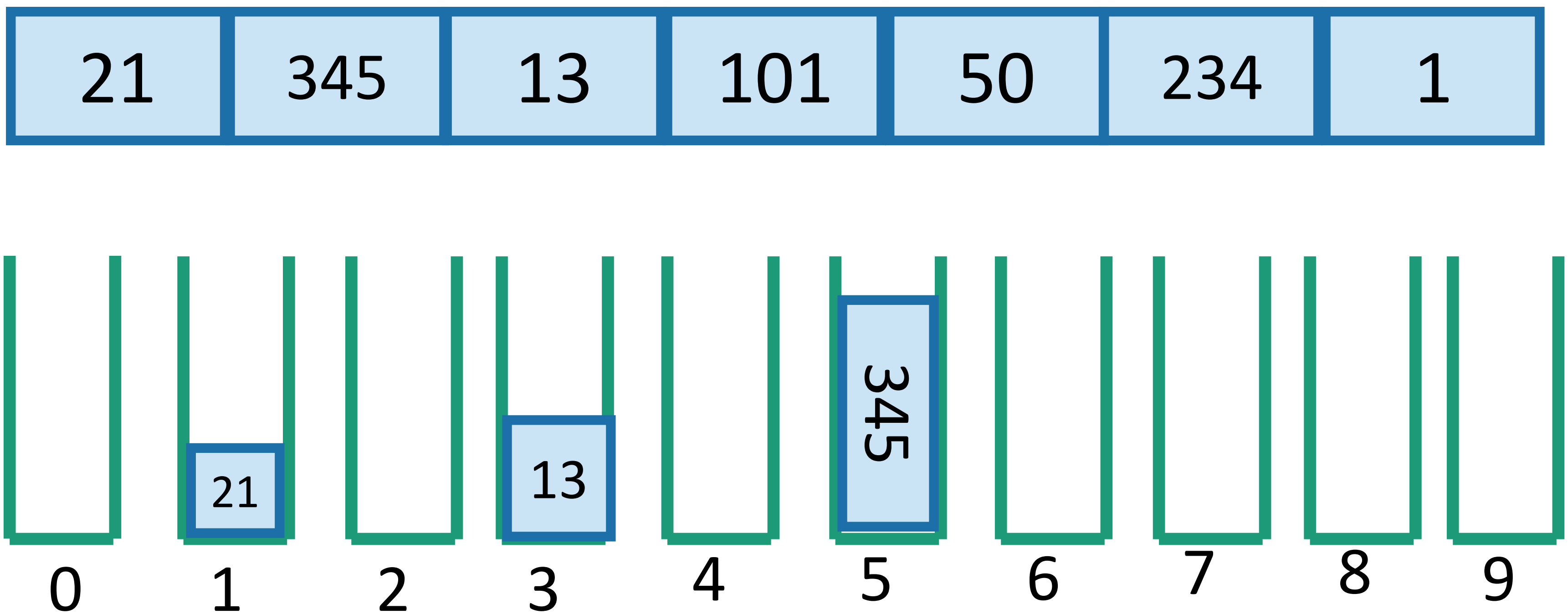
Step 1: BucketSort on least significant digit



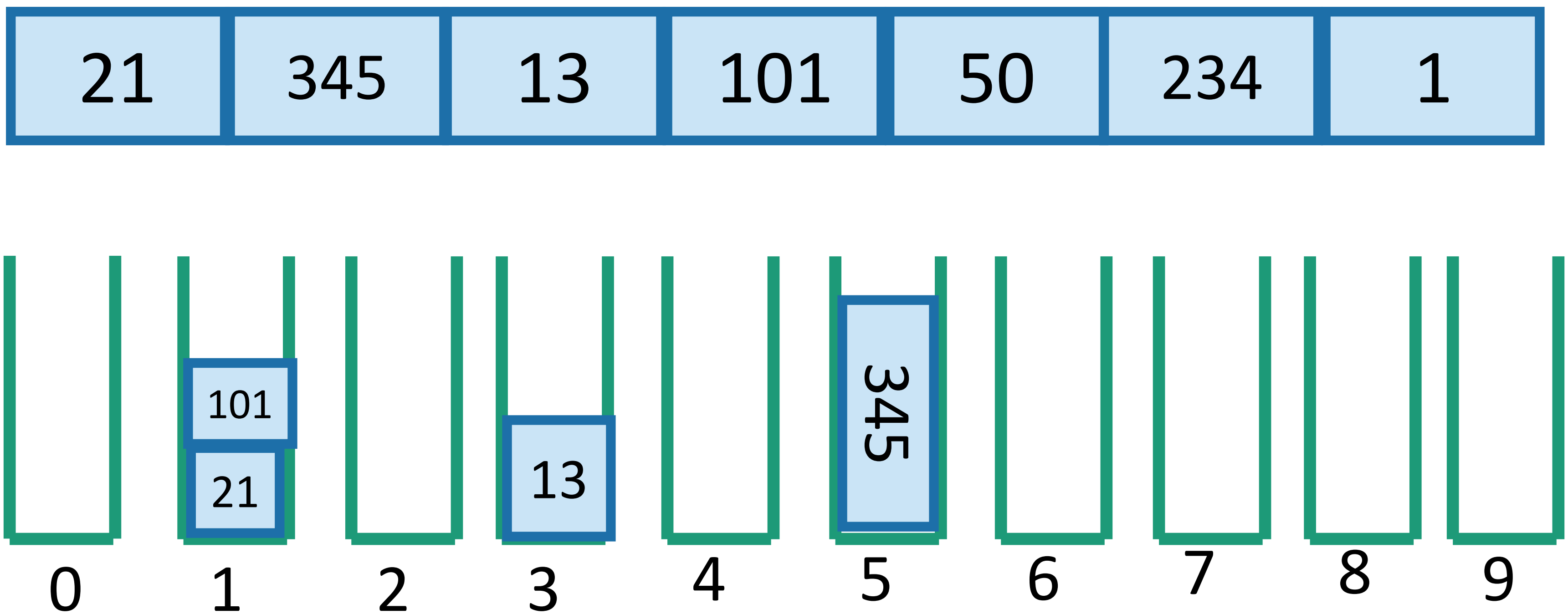
Step 1: BucketSort on least significant digit



Step 1: BucketSort on least significant digit

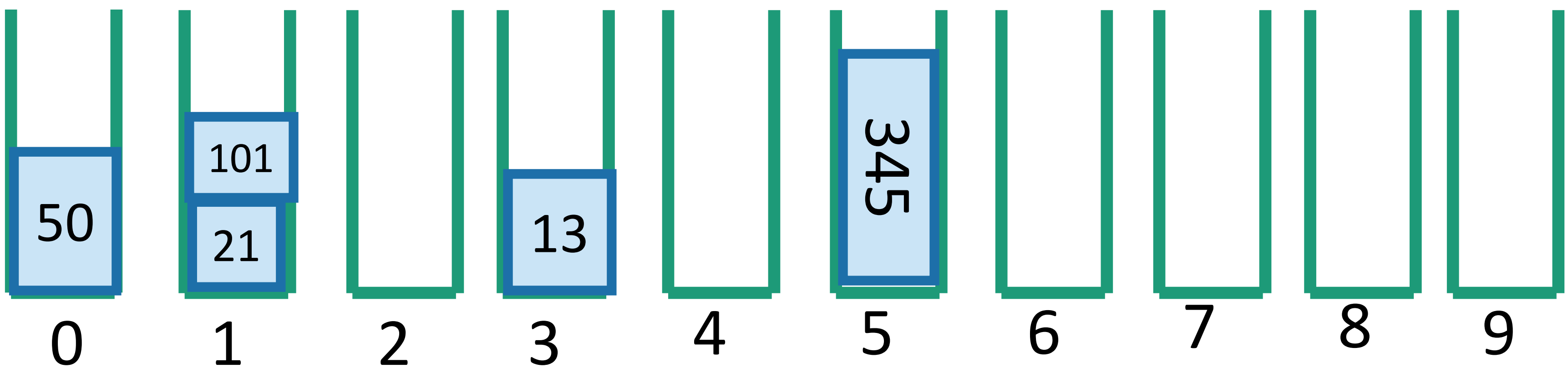


Step 1: BucketSort on least significant digit

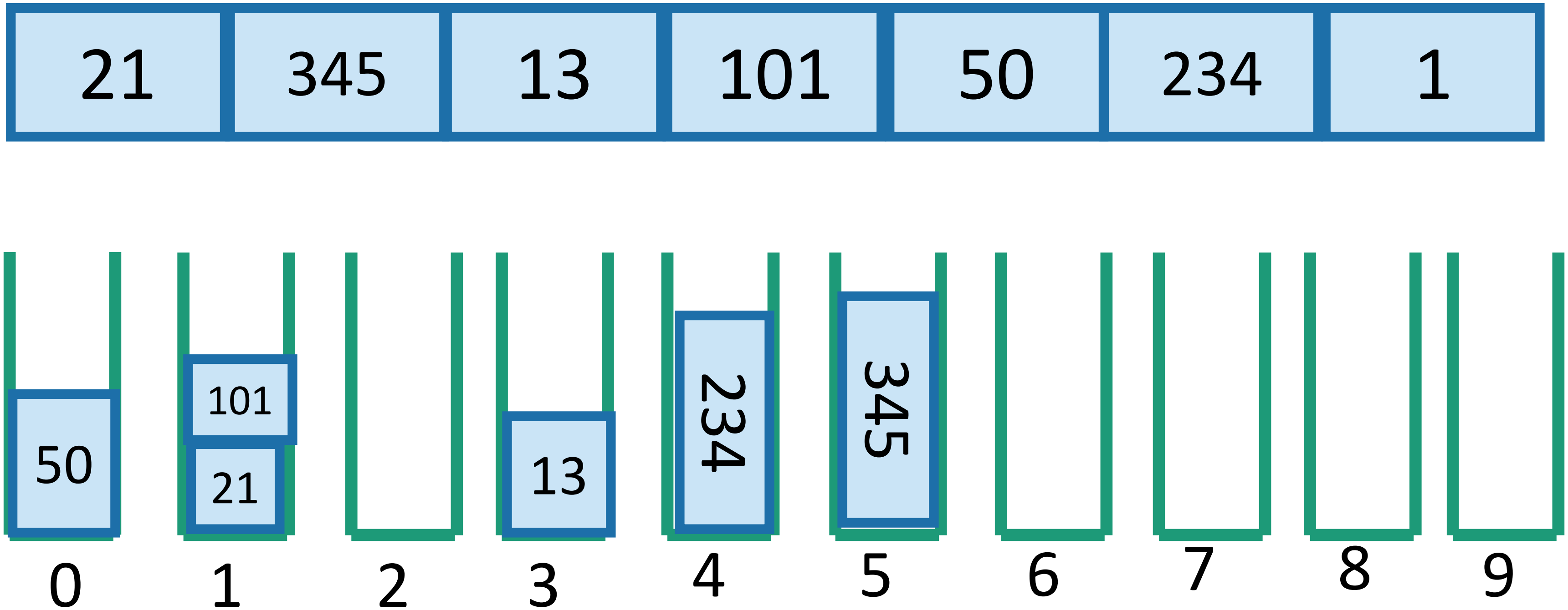


Step 1: BucketSort on least significant digit

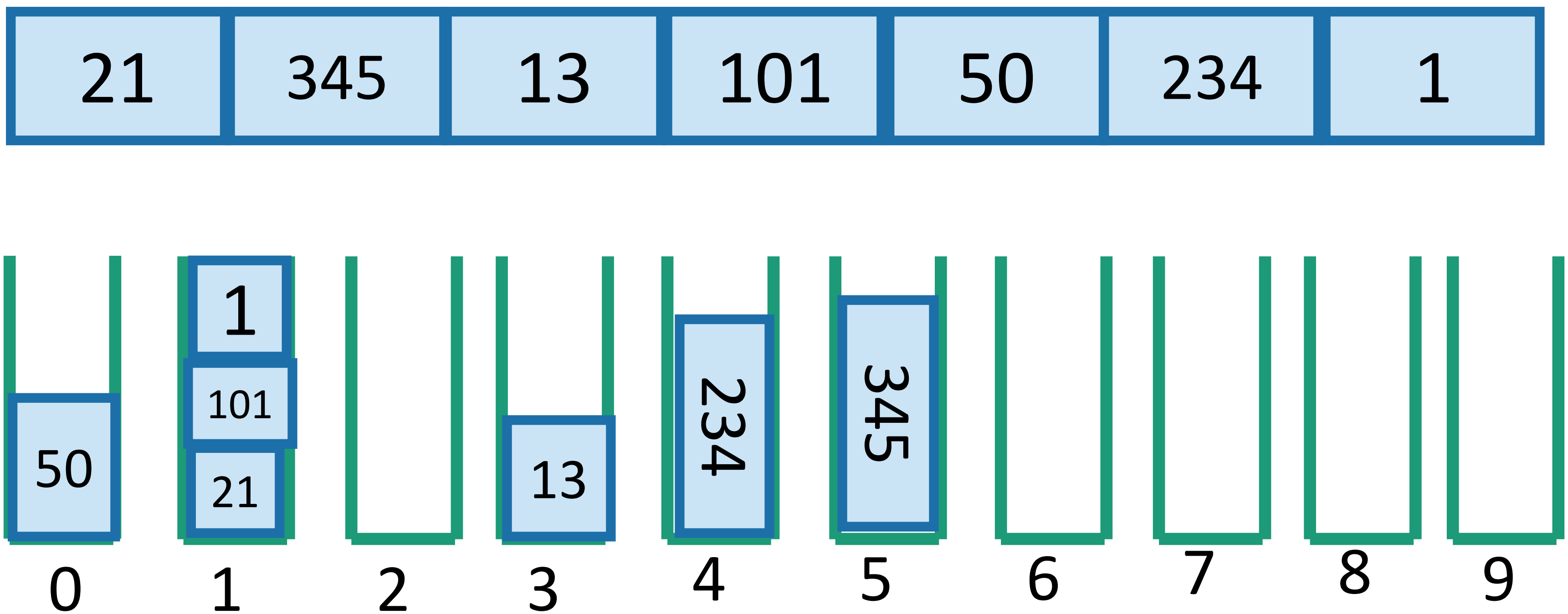
21	345	13	101	50	234	1



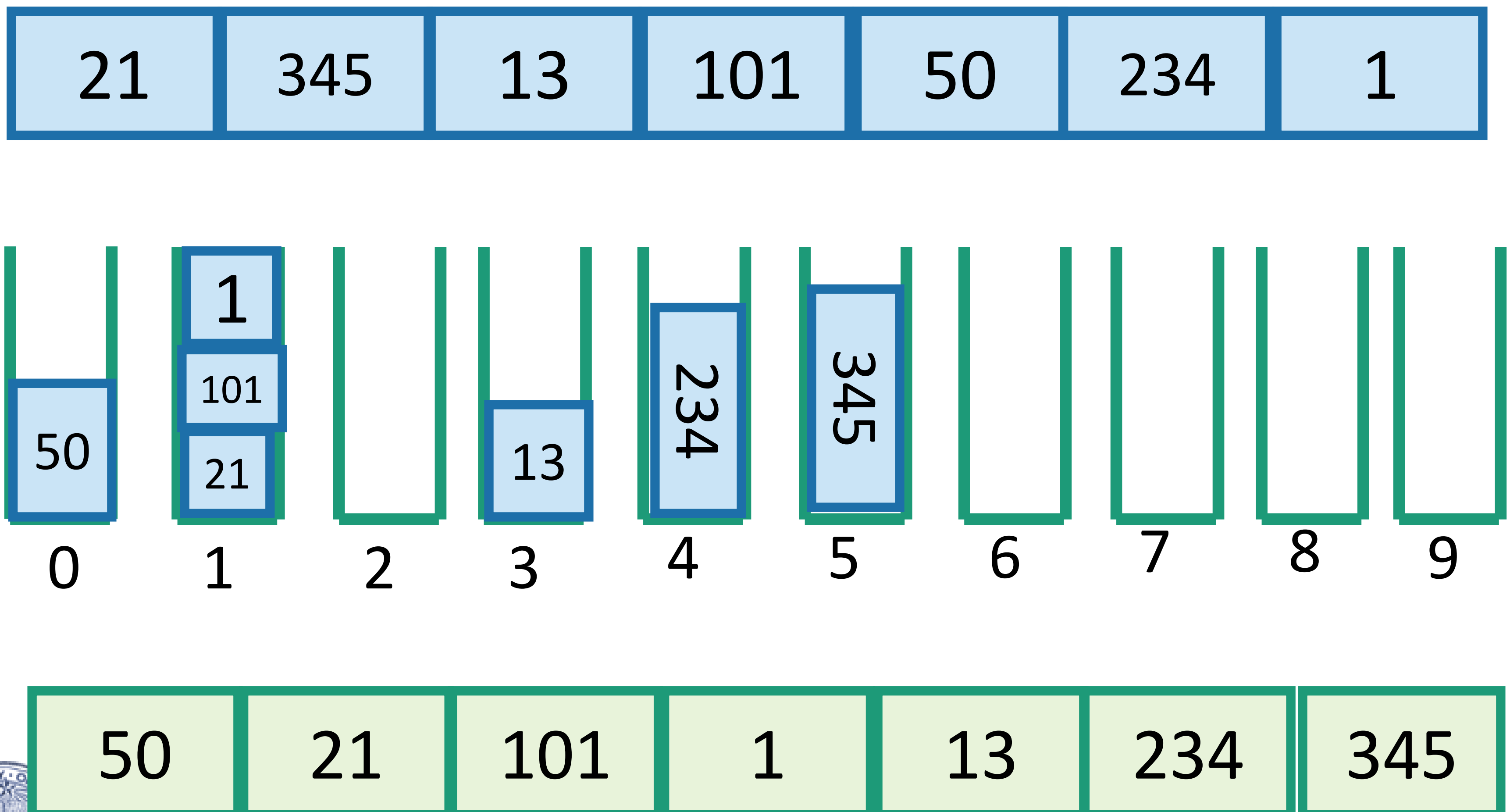
Step 1: BucketSort on least significant digit



Step 1: BucketSort on least significant digit

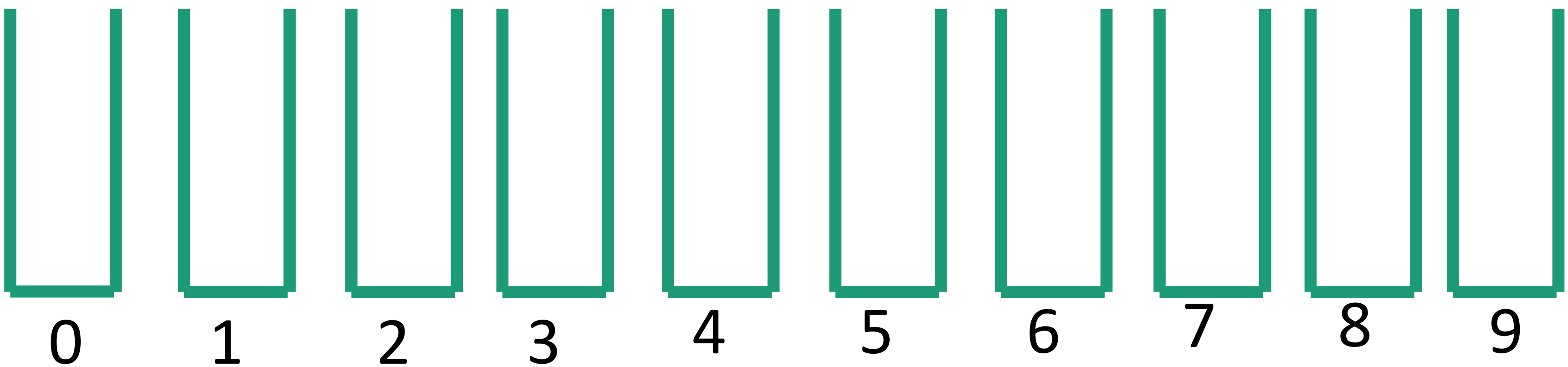


Step 1: BucketSort on least significant digit

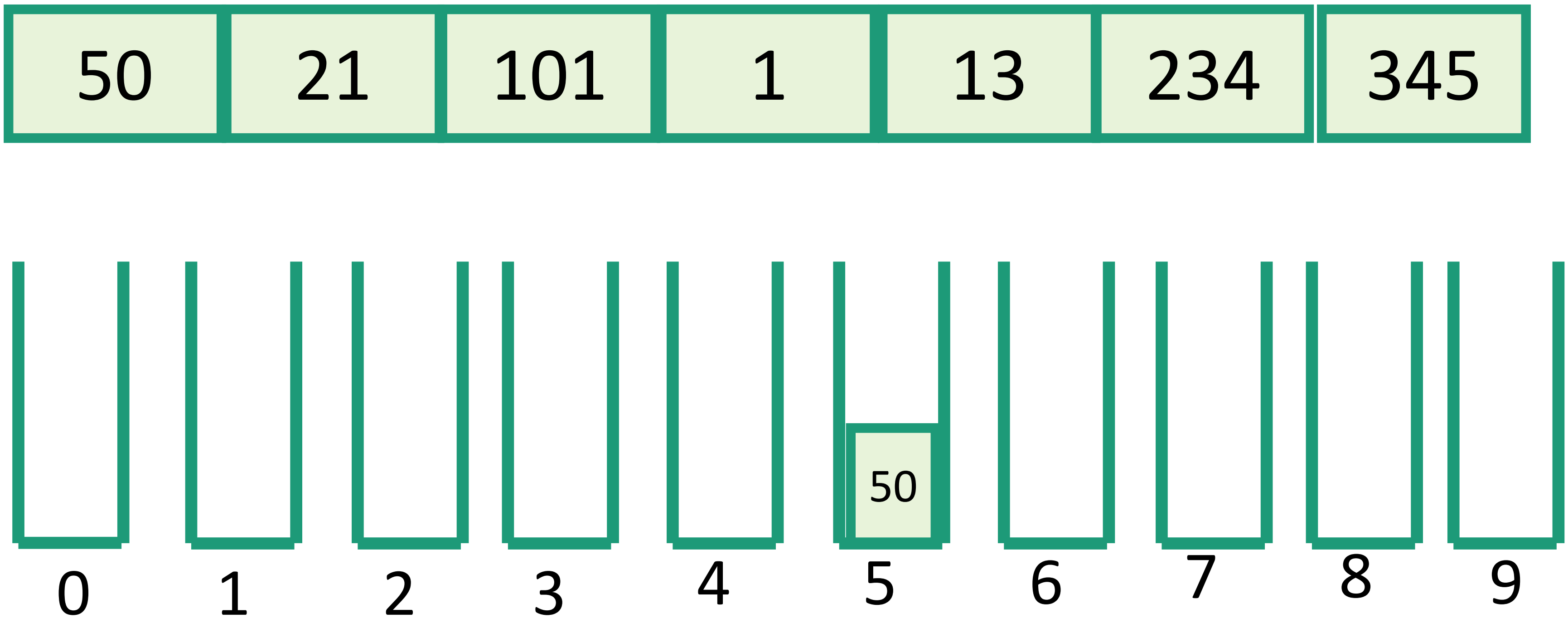


Step 2: BucketSort on the 2^{nd} least sig. digit

50	21	101	1	13	234	345

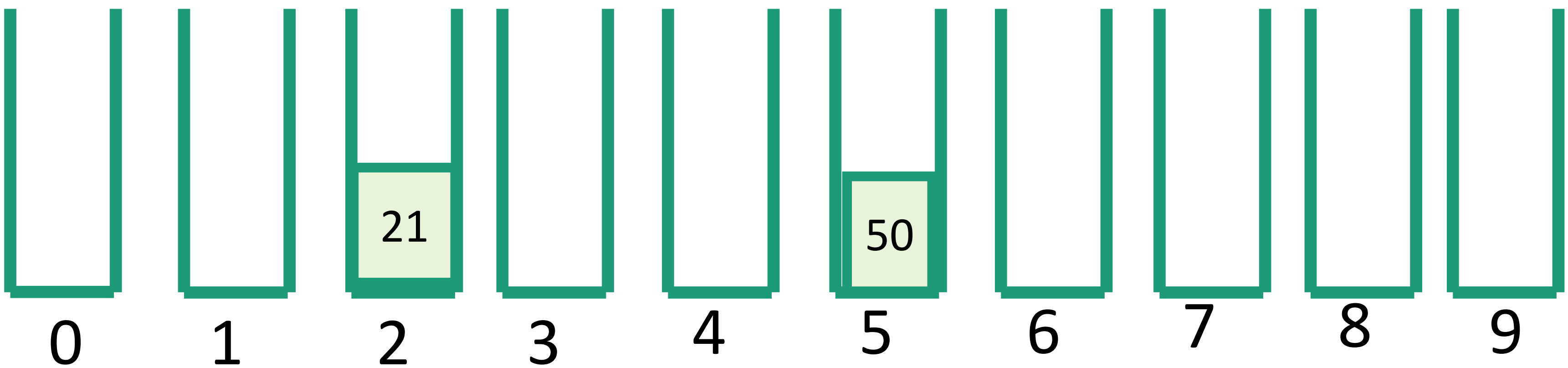


Step 2: BucketSort on the 2nd least sig. digit



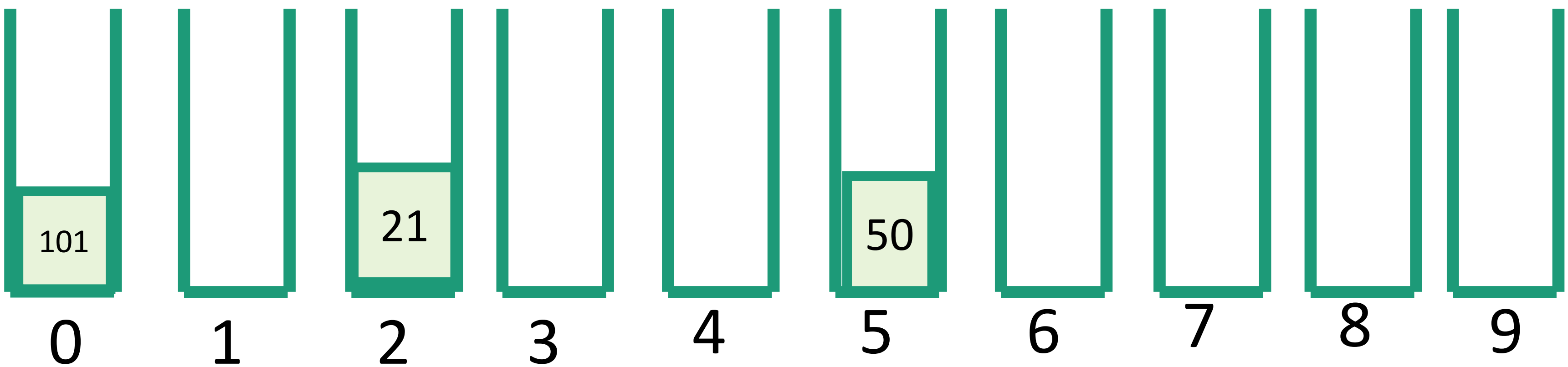
Step 2: BucketSort on the 2nd least sig. digit

50	21	101	1	13	234	345

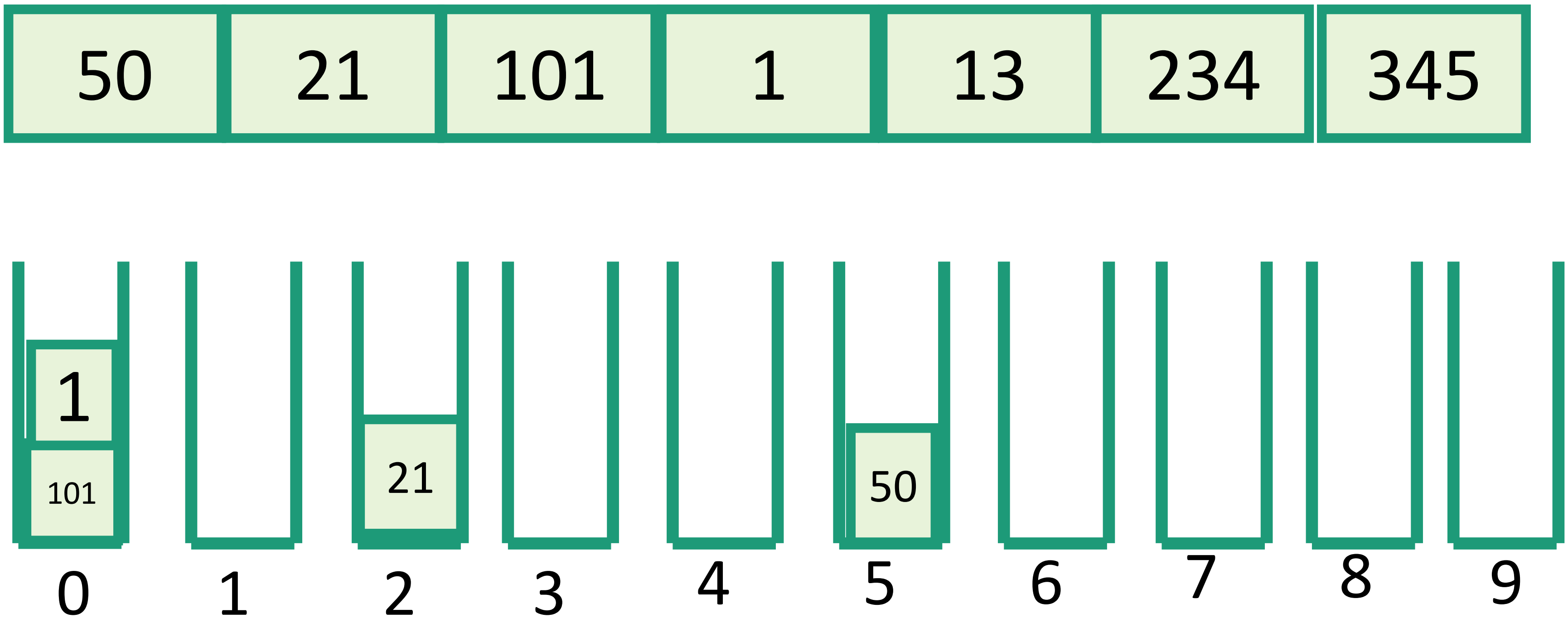


Step 2: BucketSort on the 2nd least sig. digit

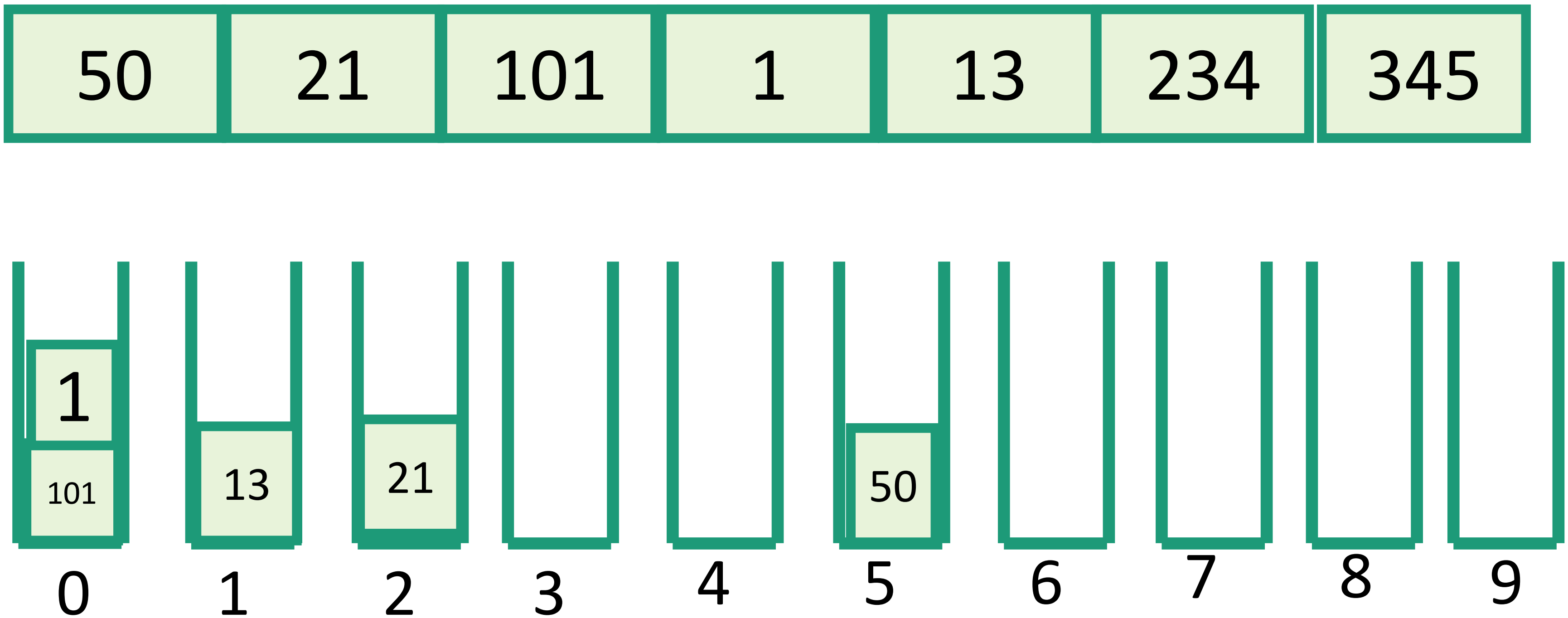
50	21	101	1	13	234	345



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

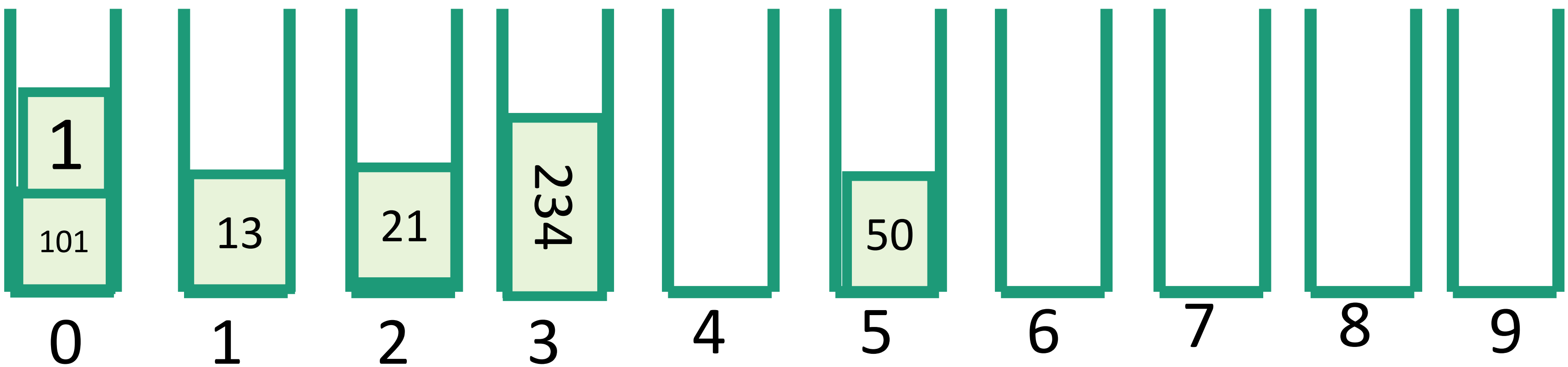


Step 2: BucketSort on the 2nd least sig. digit

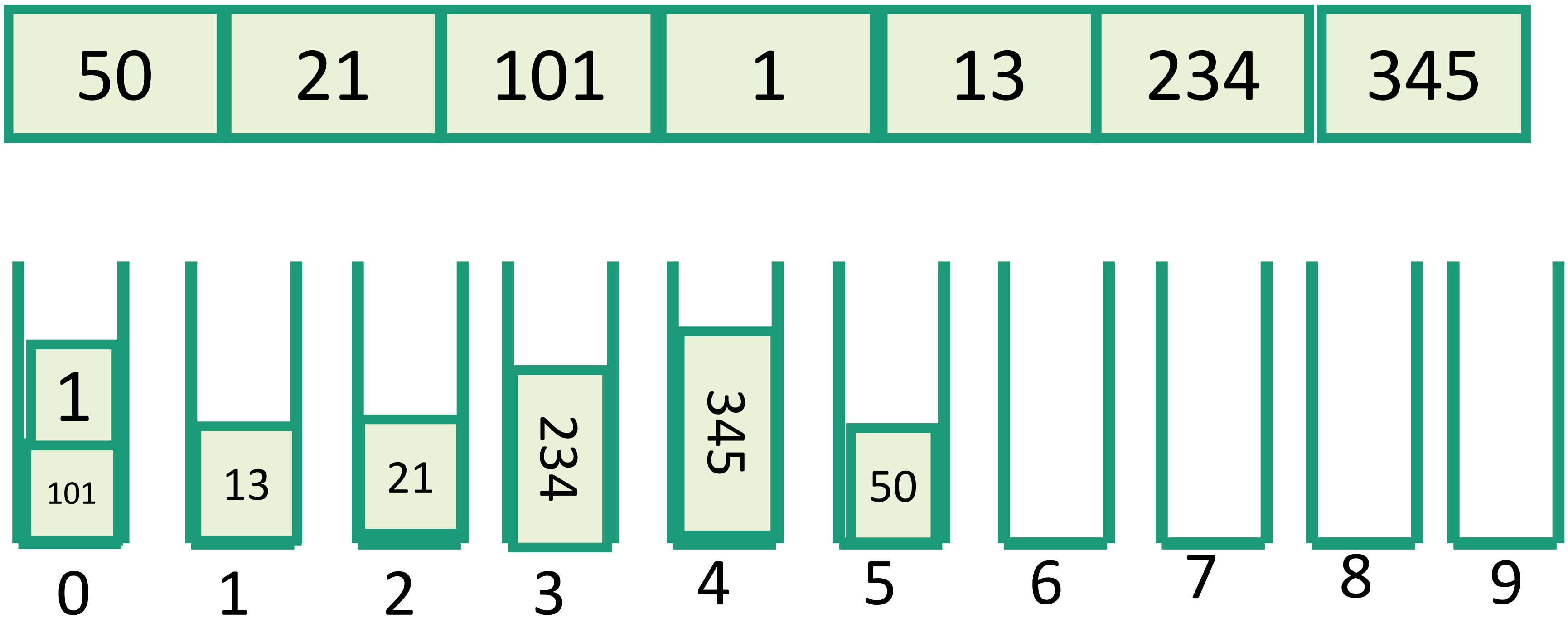


Step 2: BucketSort on the 2nd least sig. digit

50	21	101	1	13	234	345

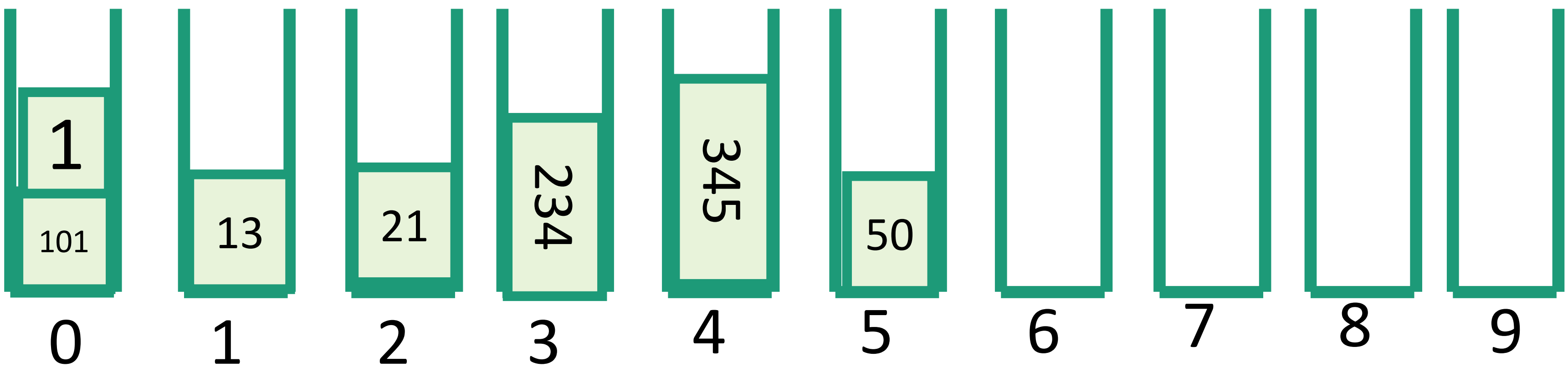


Step 2: BucketSort on the 2^{nd} least sig. digit



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

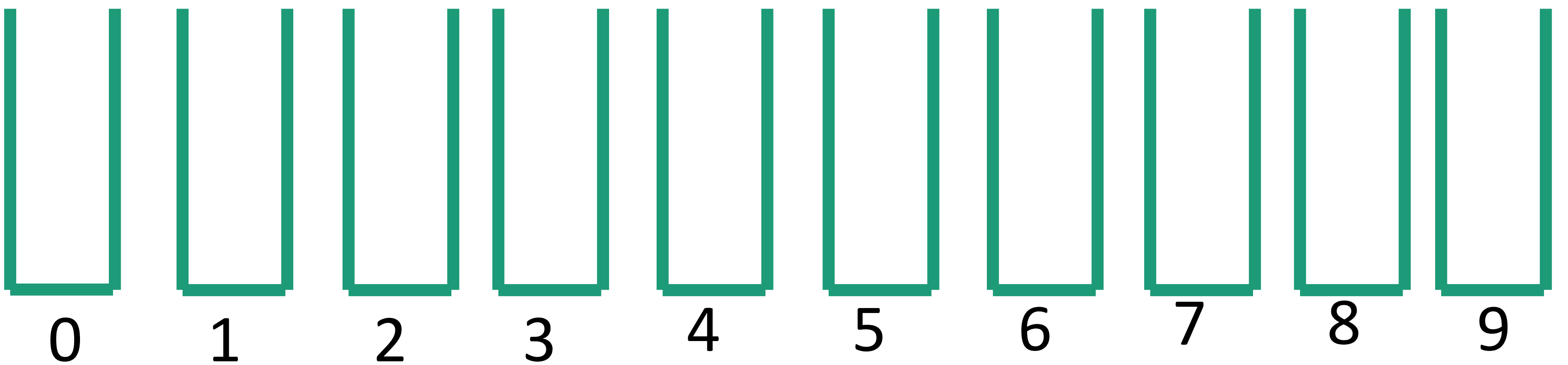
50	21	101	1	13	234	345



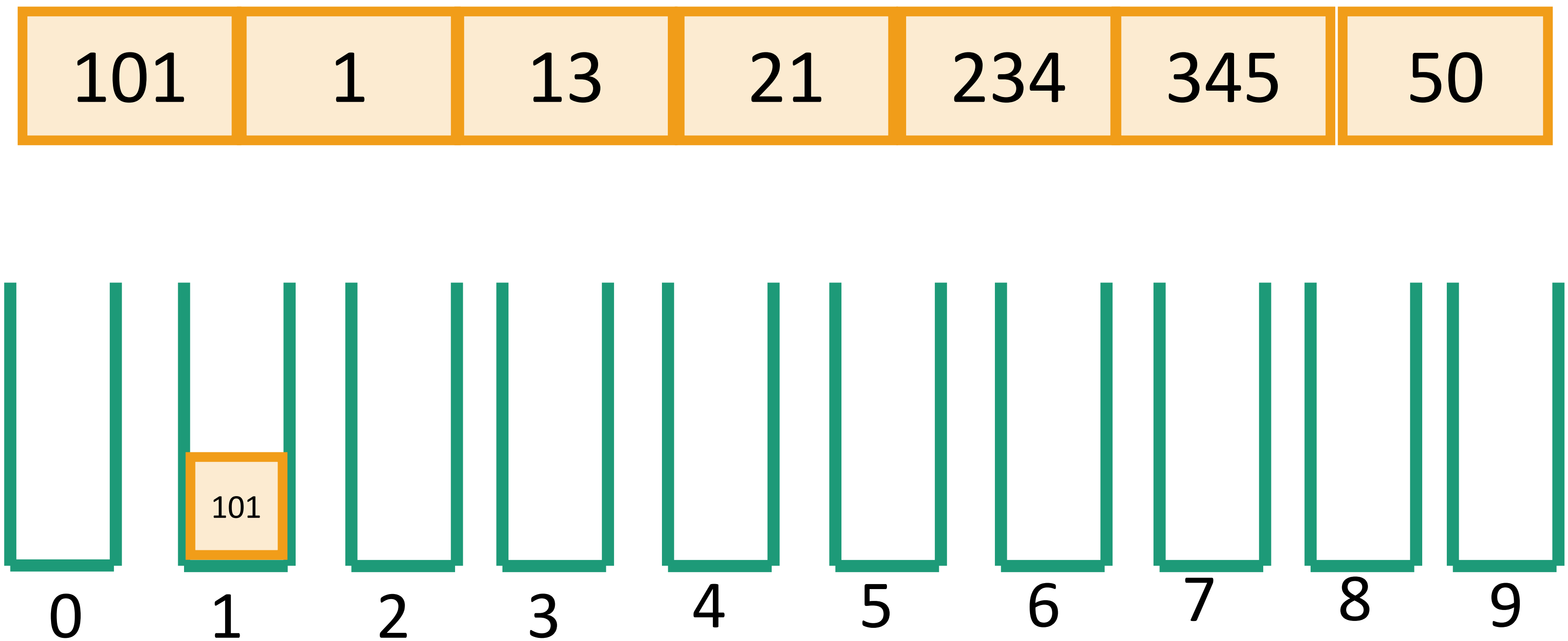
101	1	13	21	234	345	50

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

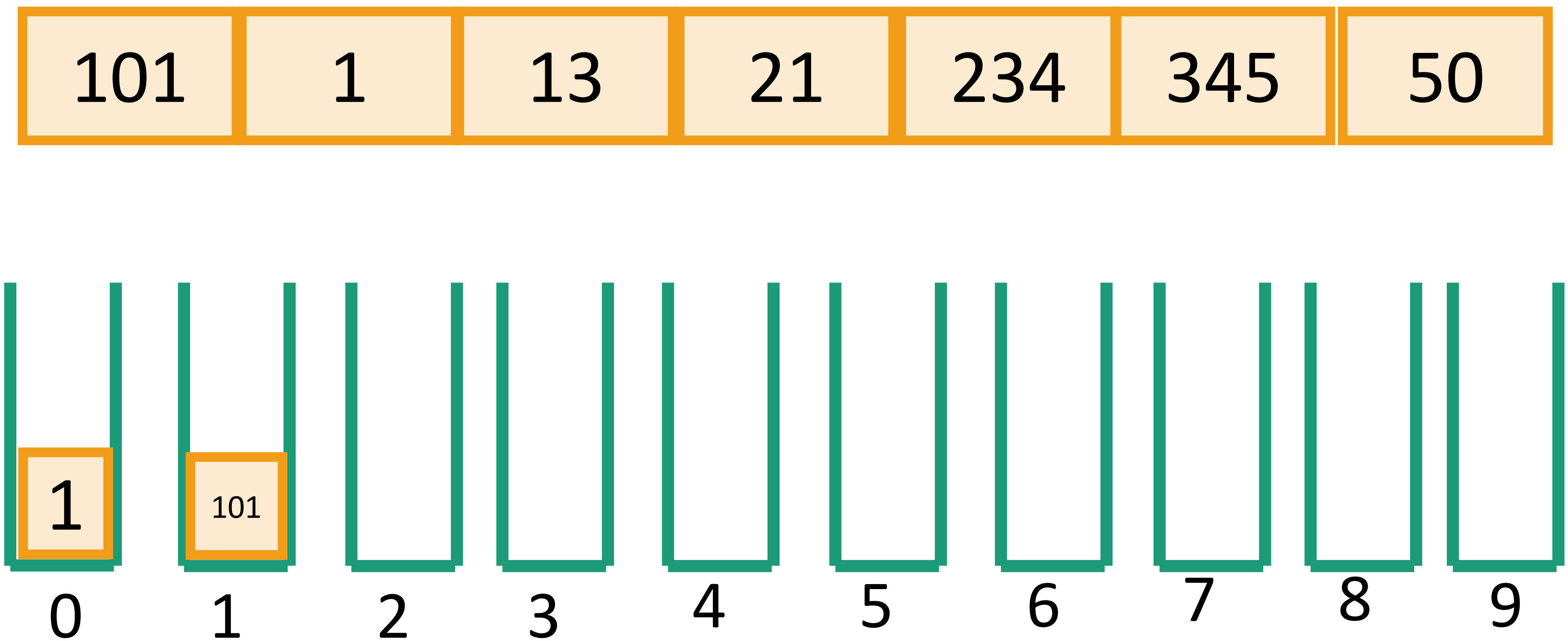
101	1	13	21	234	345	50



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

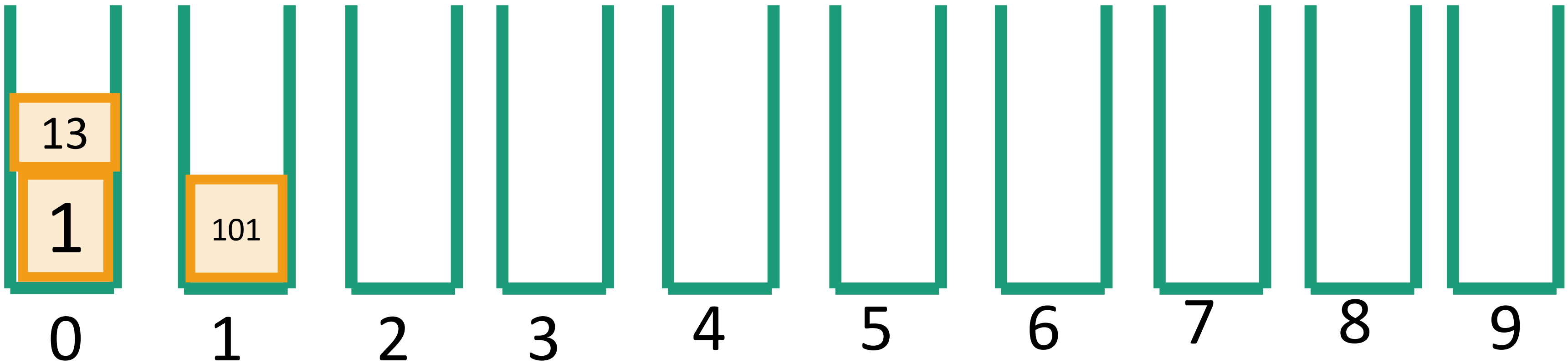


Step 3: BucketSort on the 3rd least sig. digit



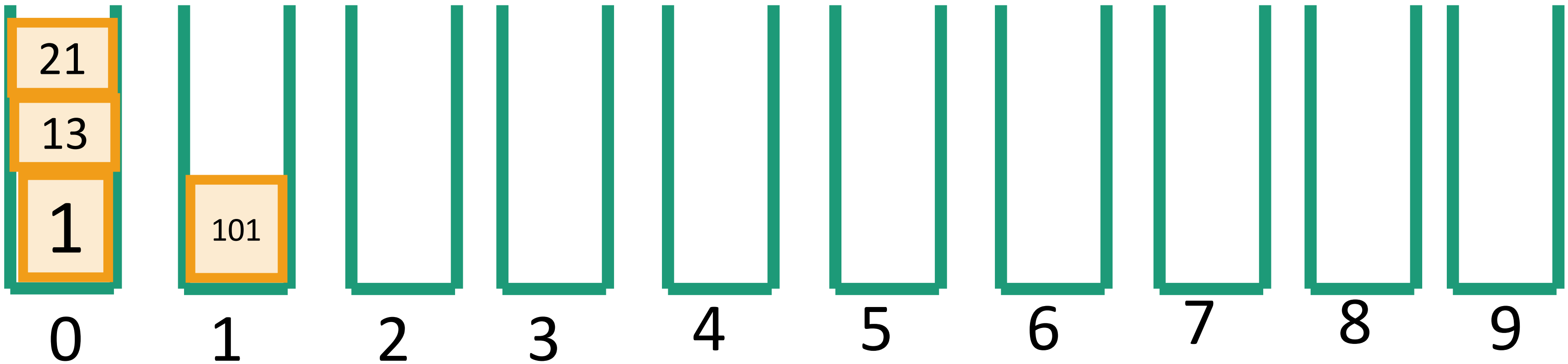
Step 3: BucketSort on the 3rd least sig. digit

101	1	13	21	234	345	50



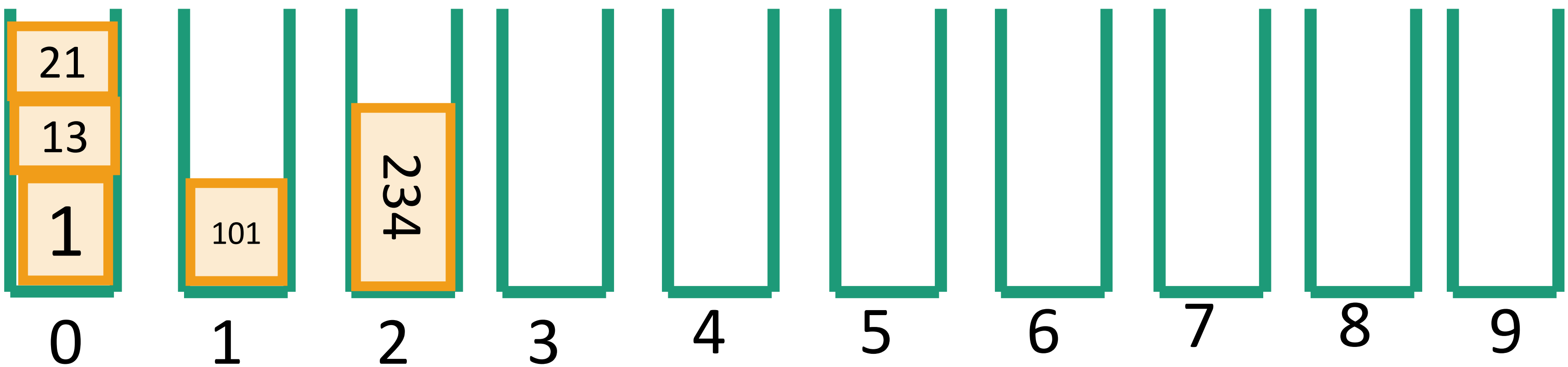
Step 3: BucketSort on the 3rd least sig. digit

101	1	13	21	234	345	50



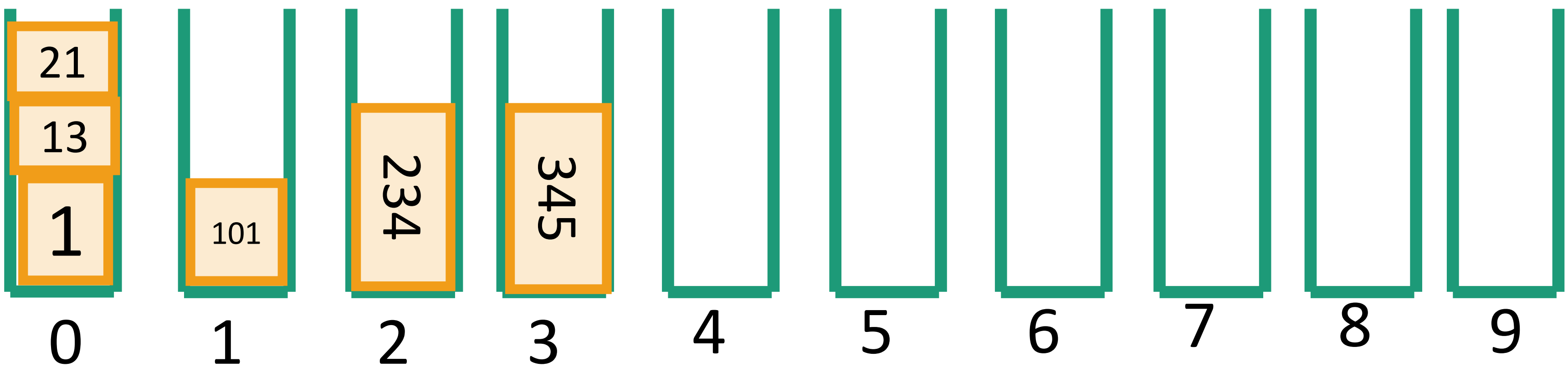
Step 3: BucketSort on the 3rd least sig. digit

101	1	13	21	234	345	50



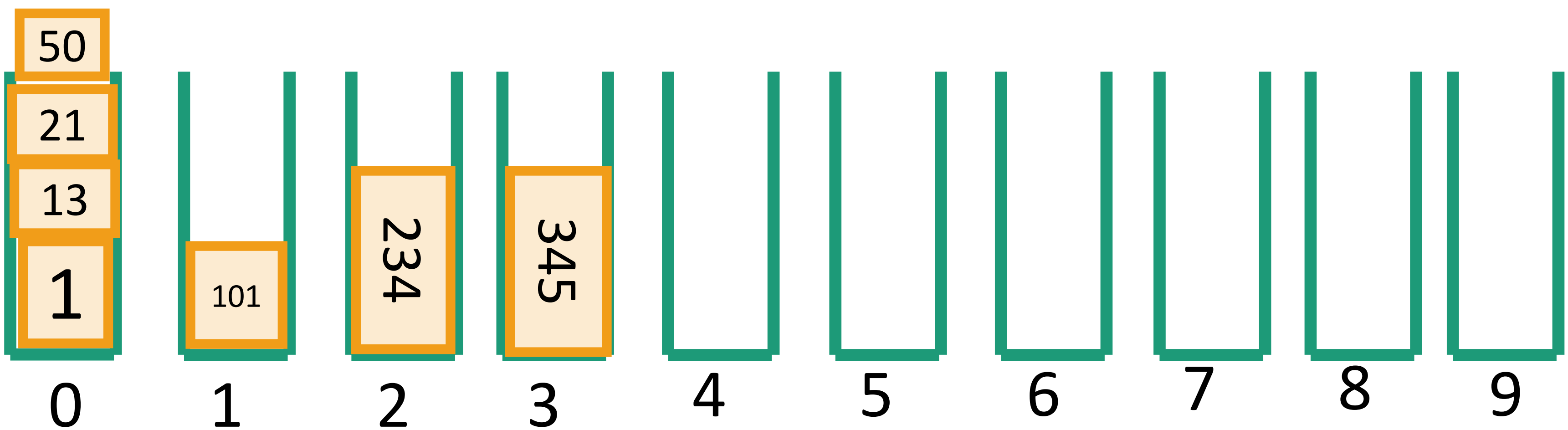
Step 3: BucketSort on the 3rd least sig. digit

101	1	13	21	234	345	50



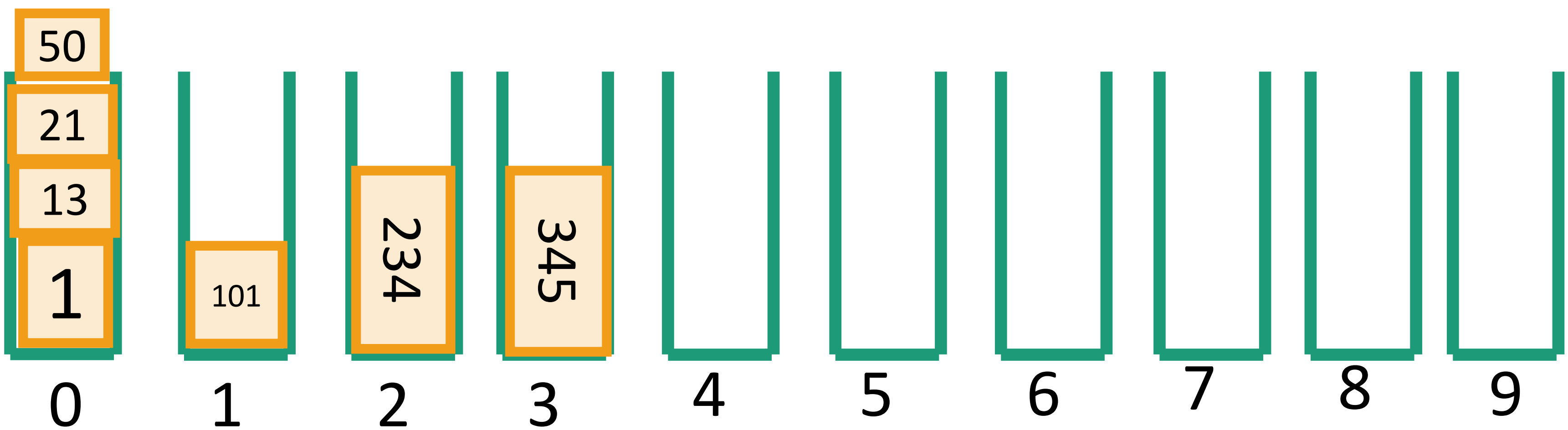
Step 3: BucketSort on the 3rd least sig. digit

101	1	13	21	234	345	50



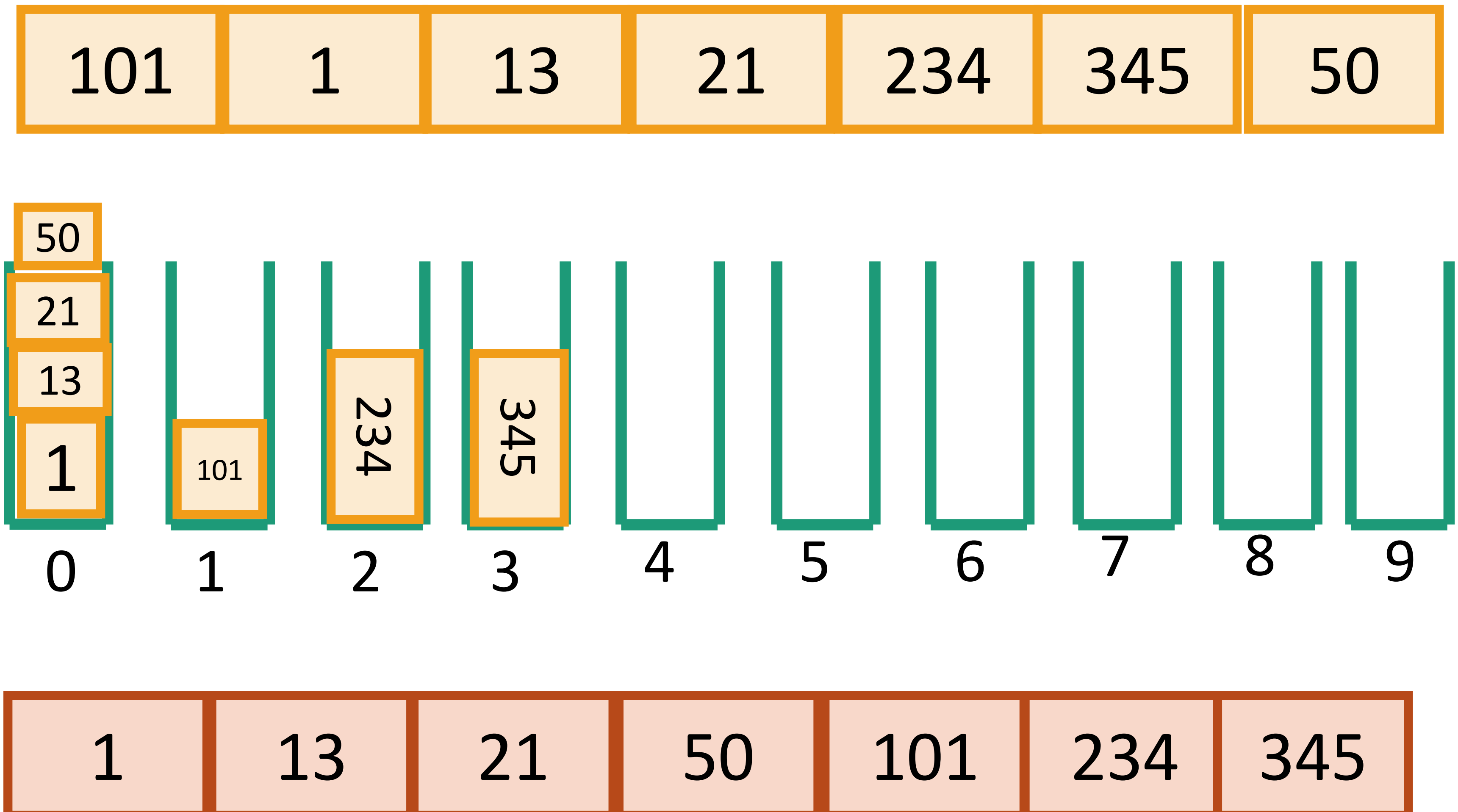
Step 3: BucketSort on the 3rd least sig. digit

101	1	13	21	234	345	50



1	13	21	50	101	234	345

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



It worked!!

Why does this work?

Original array:

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

Sorted array

Why does this work?

Original array:

21	345	13	101	50	234	1

Next array is sorted by the first digit.

5**0**	2**1**	10**1**	**1**	1**3**	23**4**	34**5**

101	1	13	21	234	345	50

1	13	21	50	101	234	345

Sorted array

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

1	13	21	50	101	234	345

Sorted array

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