CS 161 - Section 3

CA : [Name of CA]

Section 3 agenda

- Recap:
  - QuickSort
  - Runtime and Randomness
  - Lower Bounds for Sorting Algorithms
  - RadixSort, BucketSort
- Practice Problems!

Quick Sort

QuickSort

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick pivot x with **pivot**.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Rearrange A as [L, x, R]
  - QuickSort(L)
  - QuickSort(R)

Running time:

$T(n) = T(|L|) + T(|R|) + \Theta(n)$

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

$$\boxed{\boldsymbol{T(n) = O(n\ log(n))}}$$

In-Place [O(1) memory!] Quick Sort

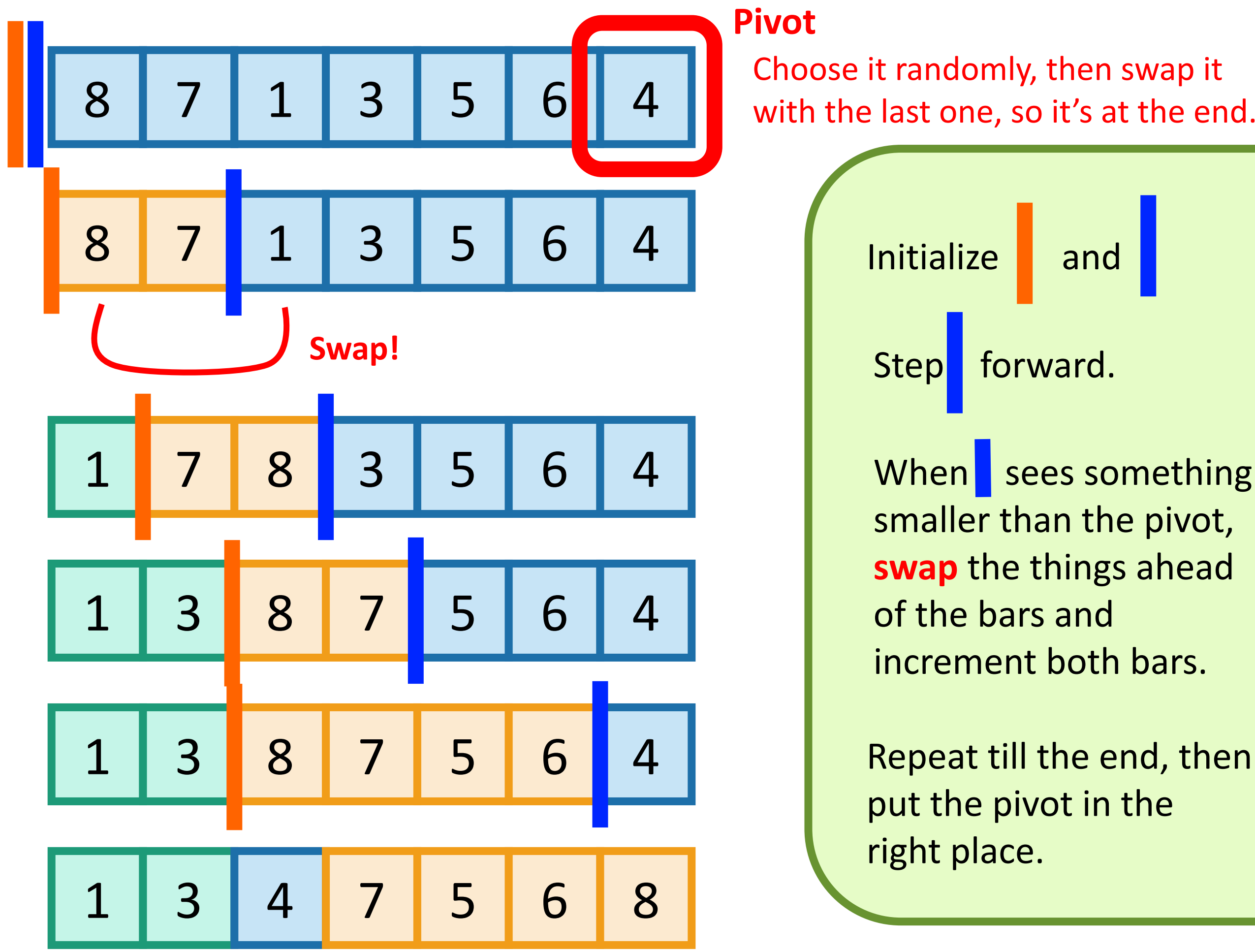
- Recall the Naïve memory complexity of Quick Sort is O(n log n)
  - Why? Think about storing an ordering of n elements for log(n) levels

- We can improve it to O(n)
  - Why? Can use a single array to represent the ordering and update at each level

- Can we do even better?
  - Let these happy Hungarians show you the answer!

https://www.youtube.com/watch?v=ywWBy6J5gz8&ab_channel=AlgoRythmics

A better way to do Partition



Initialize [orange bar] and [blue bar]

Step [blue bar] forward.

When [blue bar] 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.

Quick Sort vs Merge Sort

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	• Worst-case: $O(n^2)$ • Expected: $O(n \log(n))$	• Worst-case: $O(n \log(n))$
In-Place? (With $O(\log(n))$ extra memory)	Yes, can be implemented in-place (relatively) easily	Not as easily since you'd have to sacrifice stability and runtime, but it can be done
Stable?	No	Yes

stable sorting algorithms sort identical elements in the same order as they appear in the input

Quick Sort vs Merge Sort

Which one would you use for a small array?

Given the small size it mostly does not matter. You could still argue for Quick sort as the *expected* runtime is O(n logn) but we can get away with faster than that, and even if we miss, the worst case is O(n^2). But at this scale the difference will be in the order of *ms*

Which one would you use for an array with millions of elements?

Because for large *n* the Law of Large Numbers kicks in, we can reasonably expect both algorithms to run in O(n logn). It then becomes a priority choice between stability and memory-space

Which one would you use for an array of unknown size?

This might be a little of a personal choice, but *usually* for unknown inputs (assuming you don’t even know the expected range of sizes) we value the predictability and consistency of a deterministic algorithm, so Merge sort would be preferred. Randomized algorithms can be quite challenging to debug.

Runtime and Randomness

Runtime for Randomized Algorithms

Expected Runtime

- Measures what happens **in expectation** → computing the average case in the face over randomness in the algorithm
- QuickSort → O(nlogn)
- Remember that this is still a big O runtime!
  - Upper bound holds when the adversary does not have control over the randomness in the algorithm

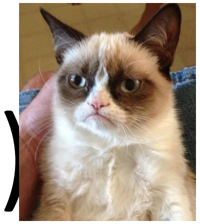
Worst-case runtime

- Adversary controls random choices in your algorithm
  - Ex: the adversary chooses “random” pivots in a manner designed to hinder QuickSort, implement SlowSort instead
- QuickSort → O(n^2)

[If time permits]

Lower Bounds for Sorting Algorithms

Lower bound of Ω(n log(n))

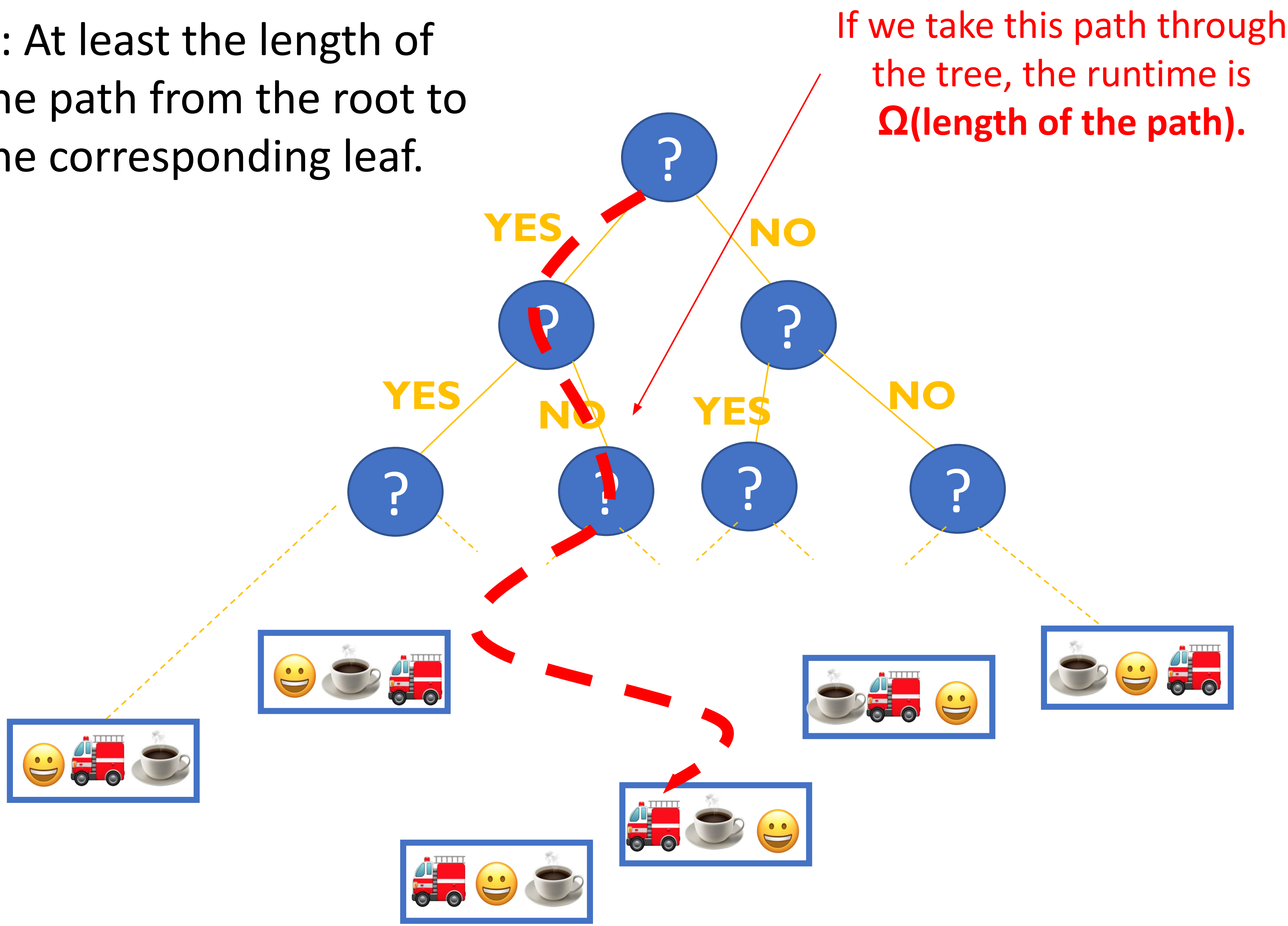


Theorem: Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.

Proof recap:

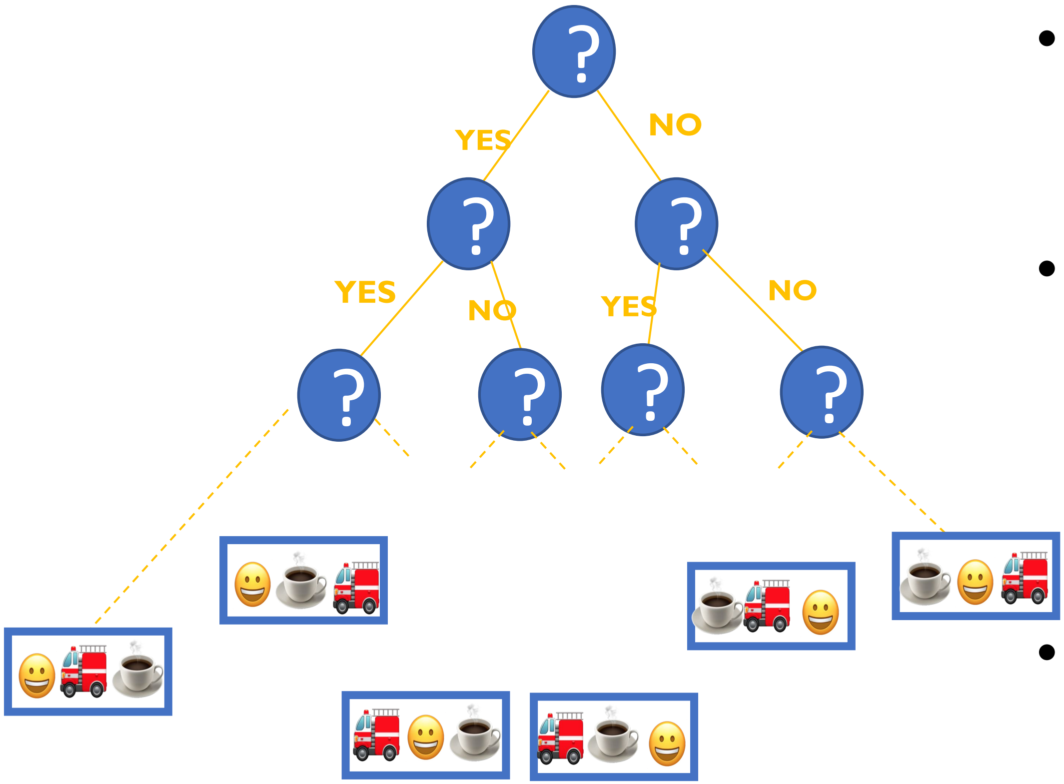
- Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.
- The worst-case running time is at least the depth of the decision tree.
- All decision trees with n! leaves have depth Ω(n log(n)).
- So any comparison-based sorting algorithm must have worst-case running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

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



How long is the longest path?

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

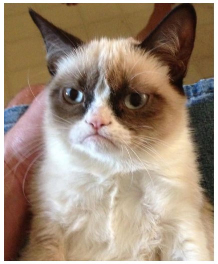


- This is a binary tree with at least __n!__ leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth __log(n!)__.
- So in all such trees, the longest path is at least log(n!).

- 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)).

Some “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.

Bad Side: we can’t improve on n*log(n)
Bright Side: we know we’re done and can focus on other problems

There’s a key word on these theorems though…

Non-comparison based model of computation

- The items you are sorting have meaningful values, meaning we can somehow evaluate them without the need of a direct comparison

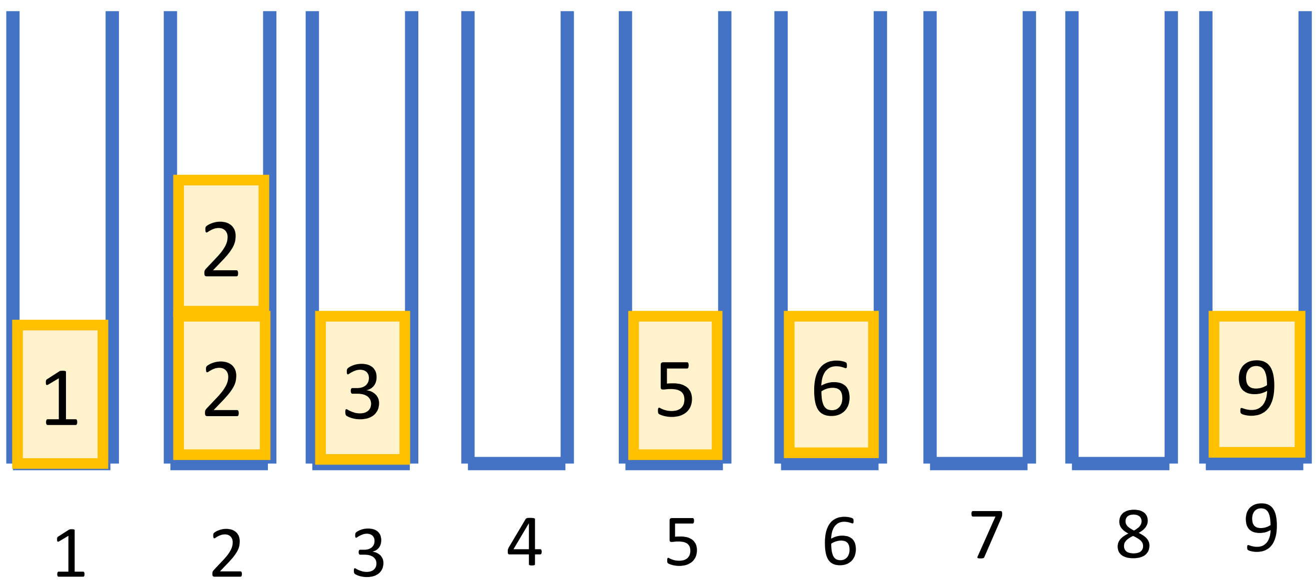
9	6	3	5	2	1	2

instead of

Counting Sort

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

Counting Sort: | 9 | 6 | 3 | 5 | 2 | 1 | 2 |



Concaten
ate the
buckets!

SORTE
D!

In time

CountingSort vs Ω(n log n) lower bound

Why did CountingSort beat our $\Omega(n \log n)$ lower bound on sorting?

Short, technically correct answer:

It's not comparison based.

Longer answer and *one* possible intuition:

In our example, $r = 10$

1. At every step, we place element in one of r buckets.
   So our decision tree is r-ary instead of binary.

2. We plan to place n numbers in r buckets.
   Instead of n! orders, there are $\underbrace{r \times r \times \cdots \times r}_{n \text{ times}} = r^n$ ways to do this.

An r-ary tree of depth $O(n)$ can have r^n leaves, so that's why we have time $O(n)$

Radix Sort

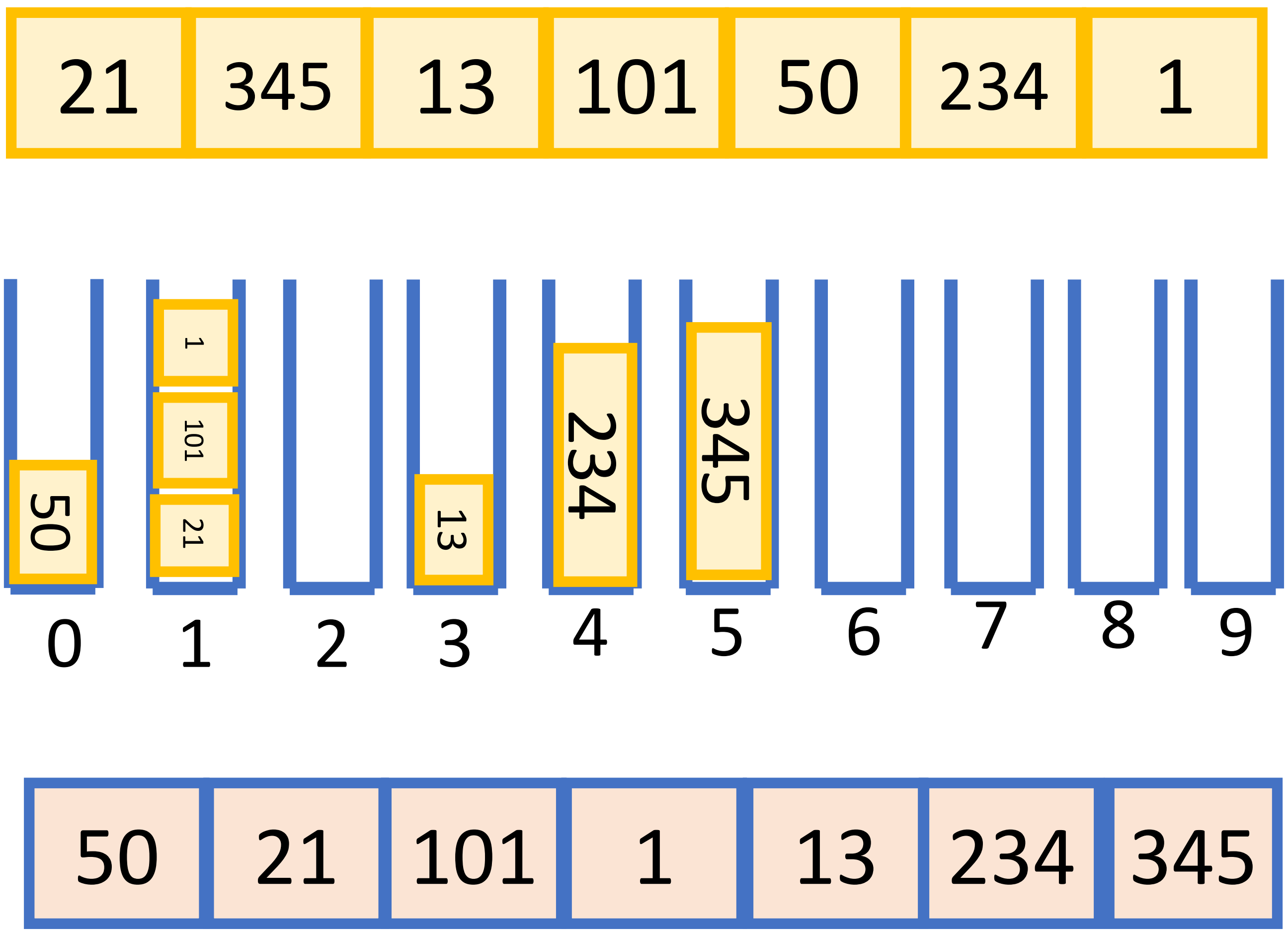
Radix Sort

- Clearly however we were only sorting 1-digit numbers, so it’s easy to know in which bucket they go
- How can we generalize to sorting arbitrary U-digit numbers?
- Could we sort strings?

- For sorting integers up to size U or lexicographically sorting strings

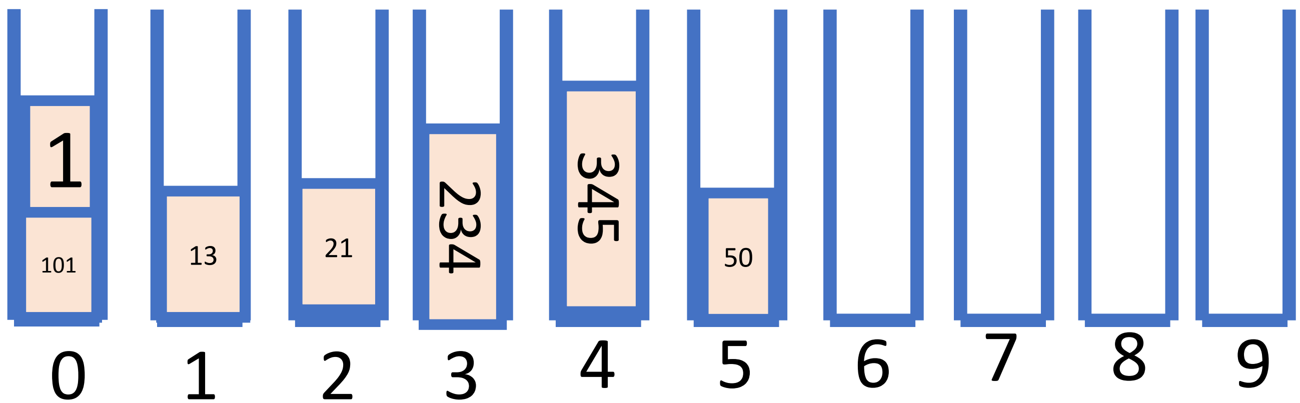
- Idea: Bucket Sort on the least-significant digit/letter first, then the next least-significant, and so on until the maximum length

Step 1: CountingSort on least significant digit



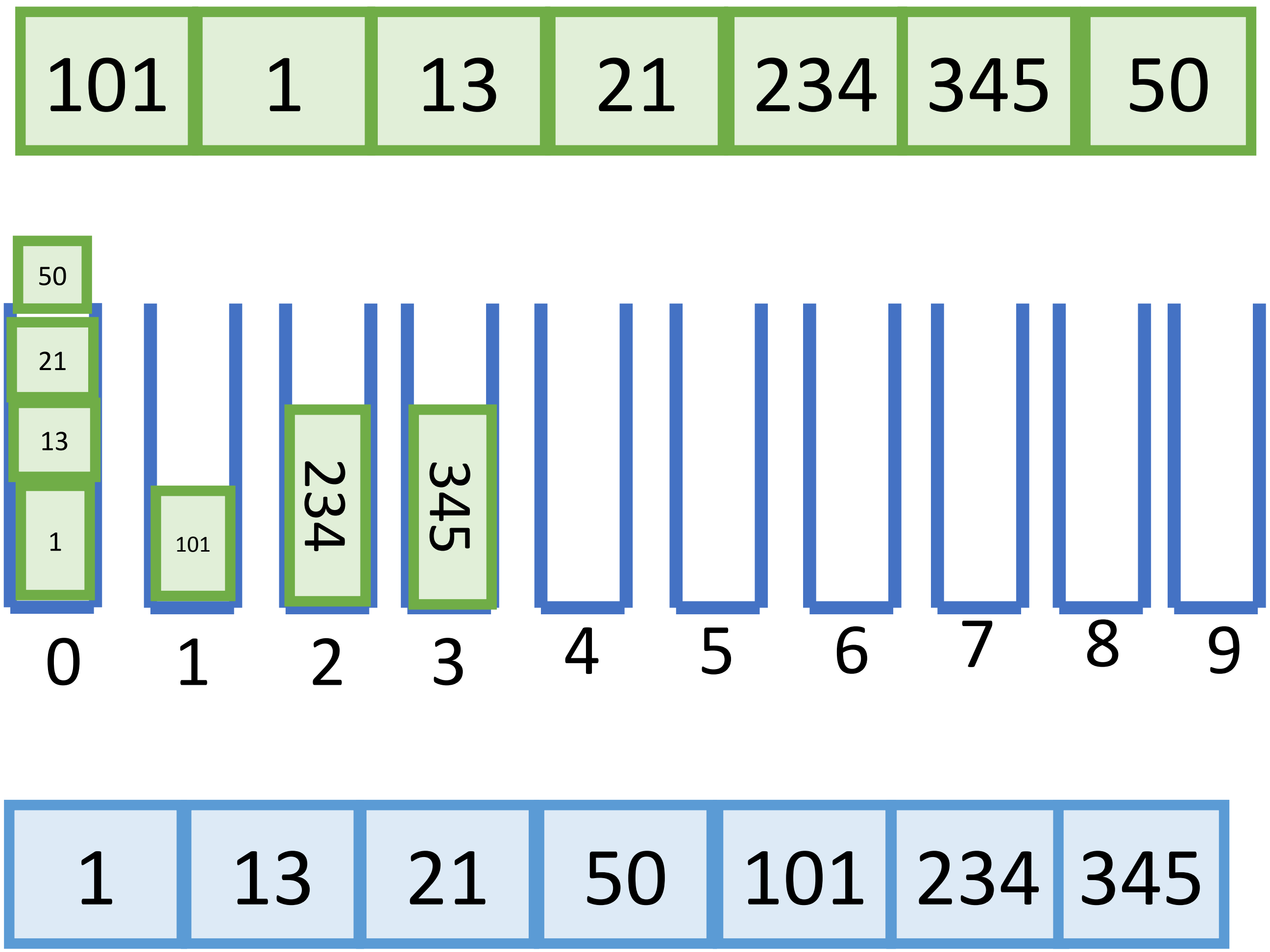
Step 2: CountingSort on the 2nd least sig. digit

50	21	101	1	13	234	345



101	1	13	21	234	345	50

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



It worked!!

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**

Next array is sorted by the first two digits.

1**01**	**01**	**13**	**21**	2**34**	3**45**	**50**

Next array is sorted by all three digits.

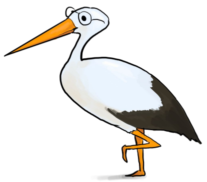
001	**013**	**021**	**050**	**101**	**234**	**345**

Sorted array

General running time of Radix Sort

- Say we want to sort:
  - n integers,
  - maximum size U-1,
  - in base r.
- Number of iterations of Radix Sort:
  - $d = \lceil \log_r(U) \rceil$
- Time per iteration:
  - Initialize r buckets, put n items into them
  - $O(n + r)$ total time.
- Total time:
  - $O\big(d \cdot (n + r)\big) = O\big(\lceil \log_r(U) \rceil \cdot (n + r)\big)$

Convince yourself that this is the right formula for d.



Thank You!