ساختمان داده و الگوریتم ها (CE203)

جلسه یازدهم: مرتب سازی خطی

سجاد شیرعلی شهرضا
پاییز 1400
شنبه، 8 آبان 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 8.1
- امتحانک دوم:
  - دوشنبه این هفته (پس فردا)، 10 آبان 1400
  - در طی ساعت کلاس به صورت برخط (مشابه امتحانک اول)

مرتب سازی خطی

الگوریتم های مرتب سازی که بر مبنای مقایسه نیستند!

A NEW MODEL OF COMPUTATION

The elements we're working with have meaningful values.

A NEW MODEL OF COMPUTATION

The elements we’re working with have meaningful values.

Before:

arbitrary elements whose values we could never directly access, process, or take advantage of (i.e. we could only interact with them via comparisons)

A NEW MODEL OF COMPUTATION

The elements we're working with have meaningful values.

Before:

arbitrary elements whose values we could never directly access, process, or take advantage of (i.e. we could only interact with them via comparisons)

Now (examples):

9	18	27	4	9	18	27

not-too-large integers

Dec	Feb	Oct	May

months in a year

مرتب سازی شمارشی

COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

COUNTING SORT

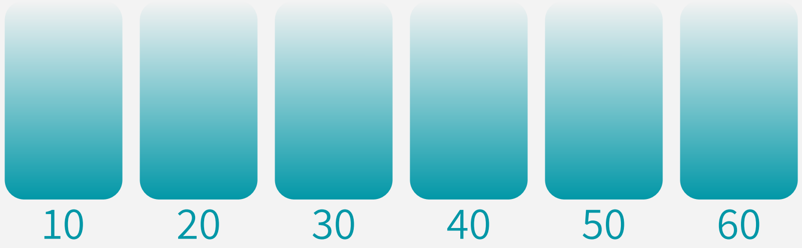
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

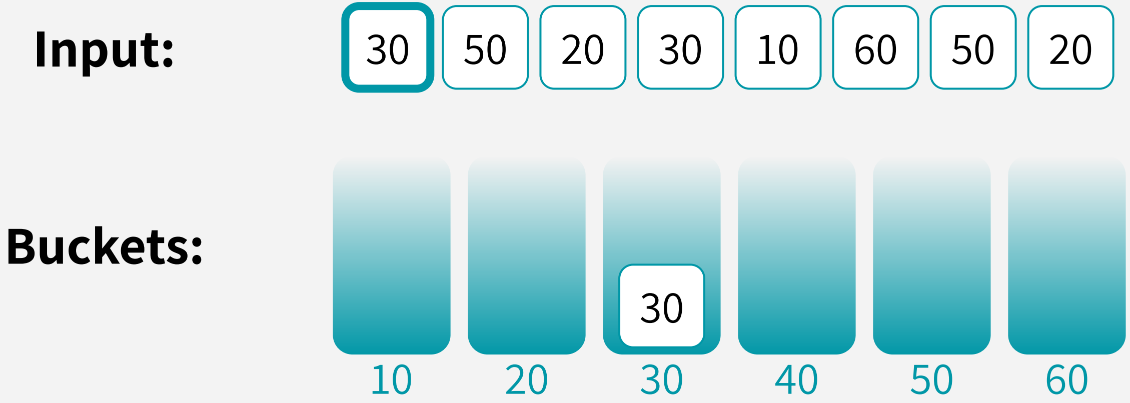
Buckets:



COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

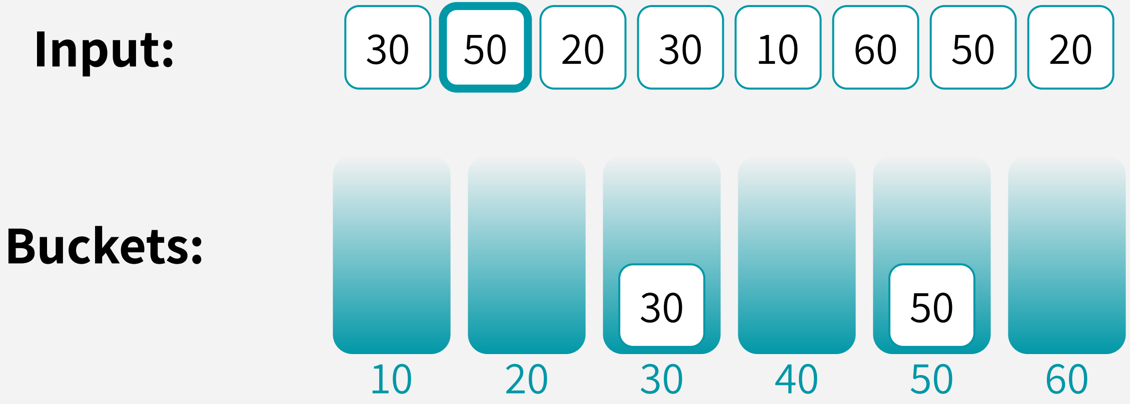
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

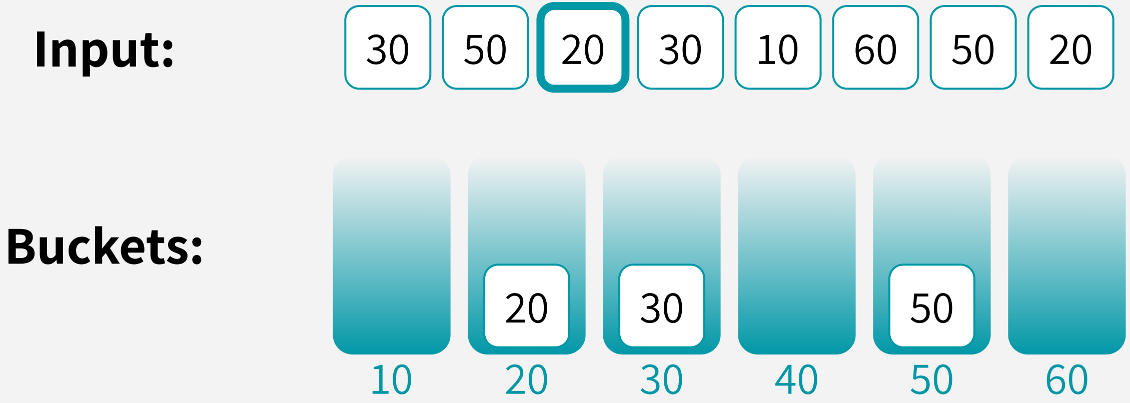
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

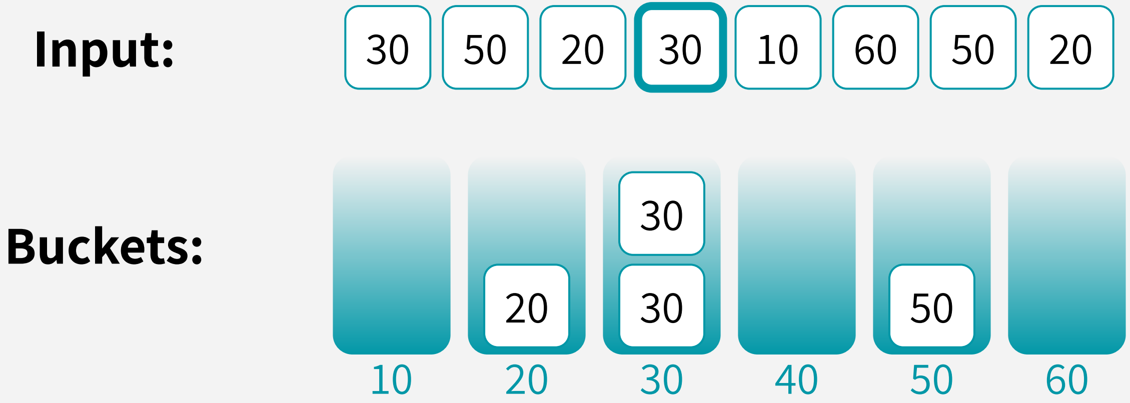
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}



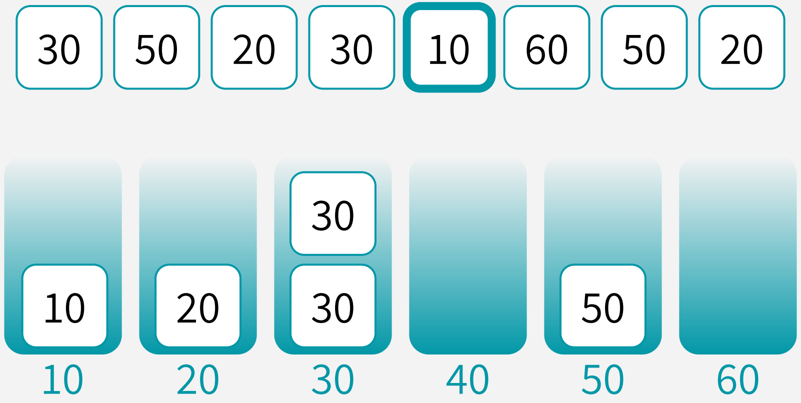
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

Buckets:



COUNTING SORT

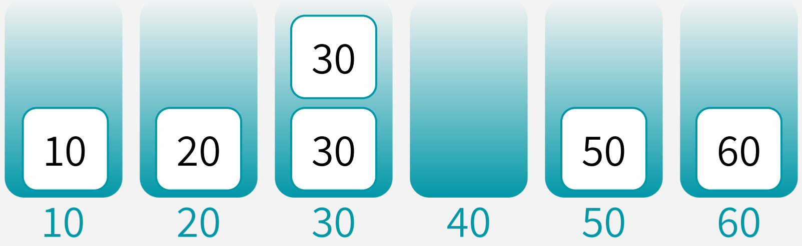
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

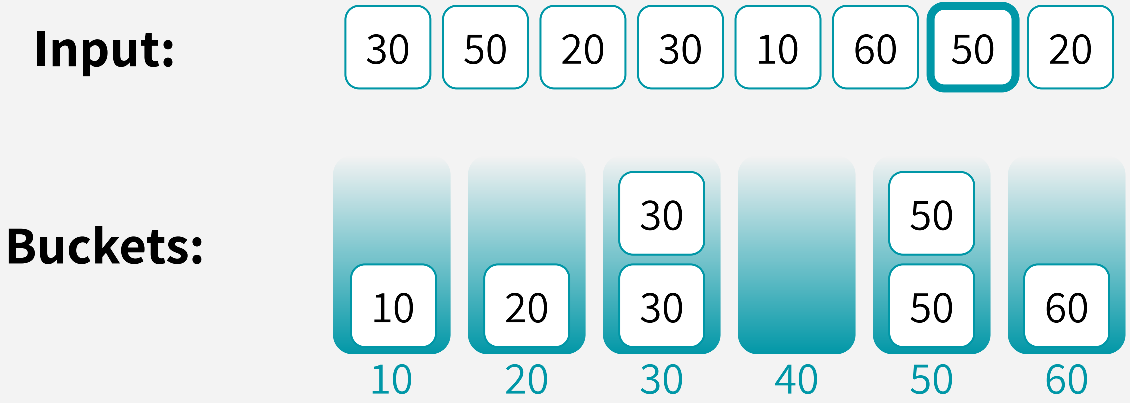
Buckets:



COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}



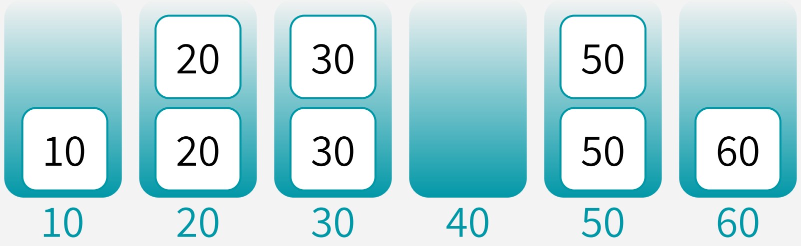
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 50 20 30 10 60 50 20

Buckets:



COUNTING SORT

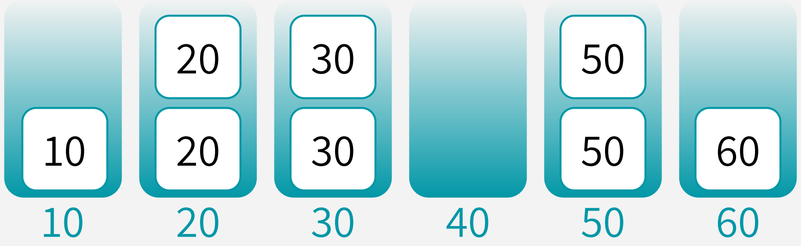
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

Buckets:



Output:

COUNTING SORT

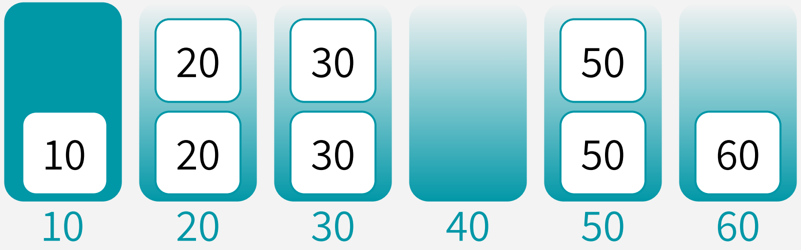
We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

Buckets:



Output:

10

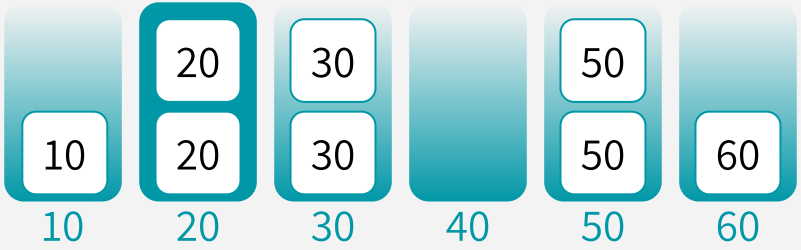
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 50 20 30 10 60 50 20

Buckets:



Output: 10 20 20

COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 50 20 30 10 60 50 20

Buckets:

10	20	30	40	50	60
	20	30		50	
10	20	30		50	60

Output: 10 20 20 30 30

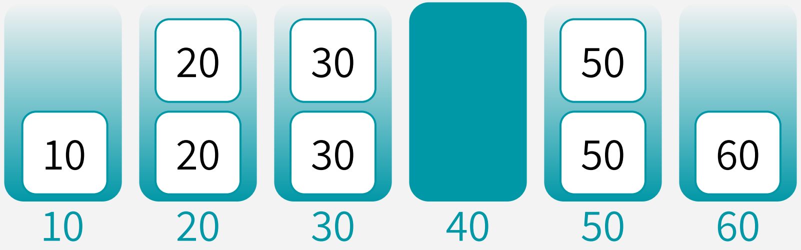
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 | 50 | 20 | 30 | 10 | 60 | 50 | 20

Buckets:



Output: 10 | 20 | 20 | 30 | 30

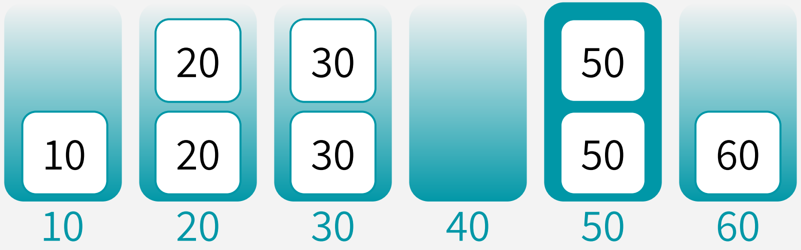
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 50 20 30 10 60 50 20

Buckets:



Output: 10 20 20 30 30 50 50

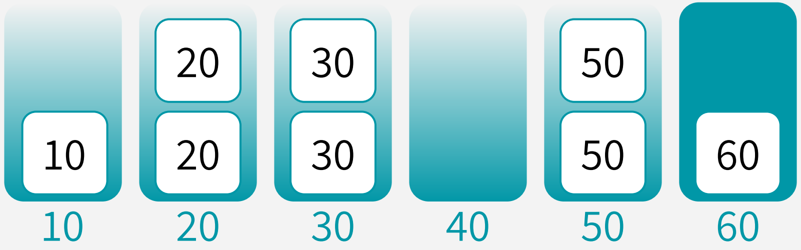
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 50 20 30 10 60 50 20

Buckets:



Output: 10 20 20 30 30 50 50 60

COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input:

30	50	20	30	10	60	50	20

Buckets:

10	20	30	40	50	60
	20	30		50	
10	20	30		50	60

Sorted in time:
O(n)

Output:

10	20	20	30	30	50	50	60

COUNTING SORT

Assumptions:

We are able to know what bucket to put something in.

We know what values might show up ahead of time.

There aren’t too many such values.

If there are too many possible values that could show up,
then we need a bucket per value…
This can easily amount to a lot of space.

سوال؟

مرتب سازی مبنایی

الگوریتم مرتب سازی برای اعداد صحیح کوچکتر از M
(و یا در حالت کلی تر، برای مرتب سازی رشته ها)

RADIX SORT

For sorting integers where the maximum value of any integer is M.
(This can be generalized to lexicographically sorting strings as well)

IDEA:

Perform CountingSort on the least-significant digit first,
then perform CountingSort on the next least-significant, and so on...

Instead of a bucket per possible value, **we just need to maintain a bucket per possible value that a single digit (or character) can take on!**

e.g. 10 buckets labeled 0, 1, …, 9

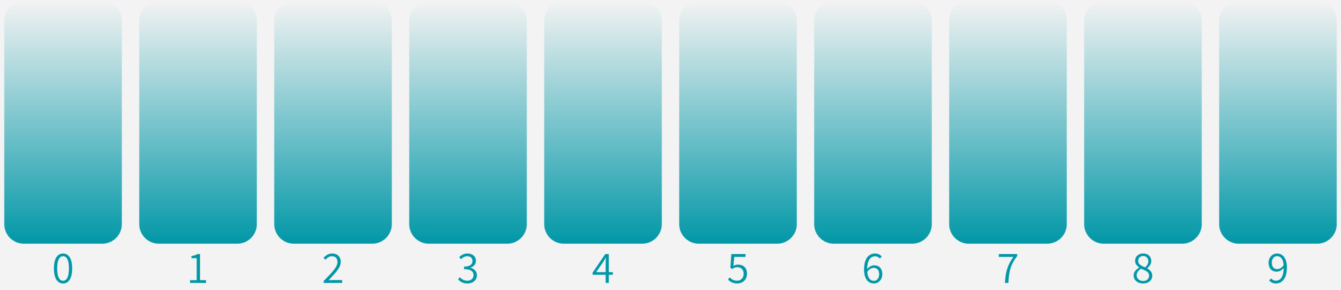
RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

21	345	13	101	50	234	1

Buckets:

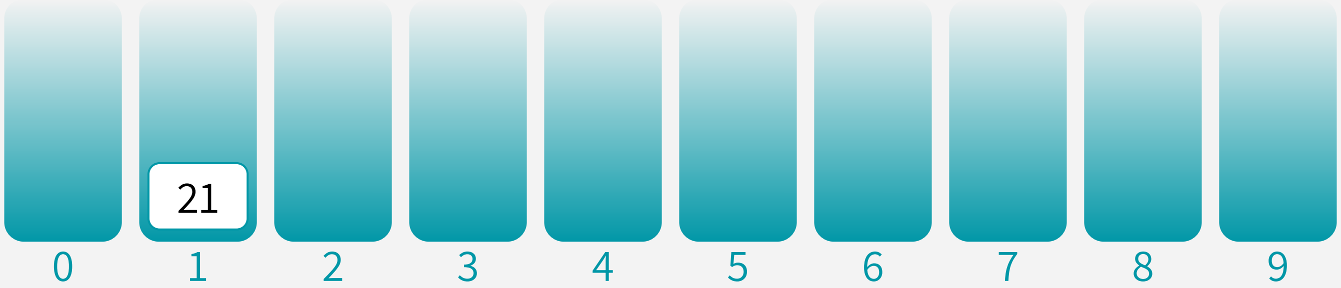


RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:



RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:

RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:

RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

21	345	13	101	50	234	1

Buckets:

RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

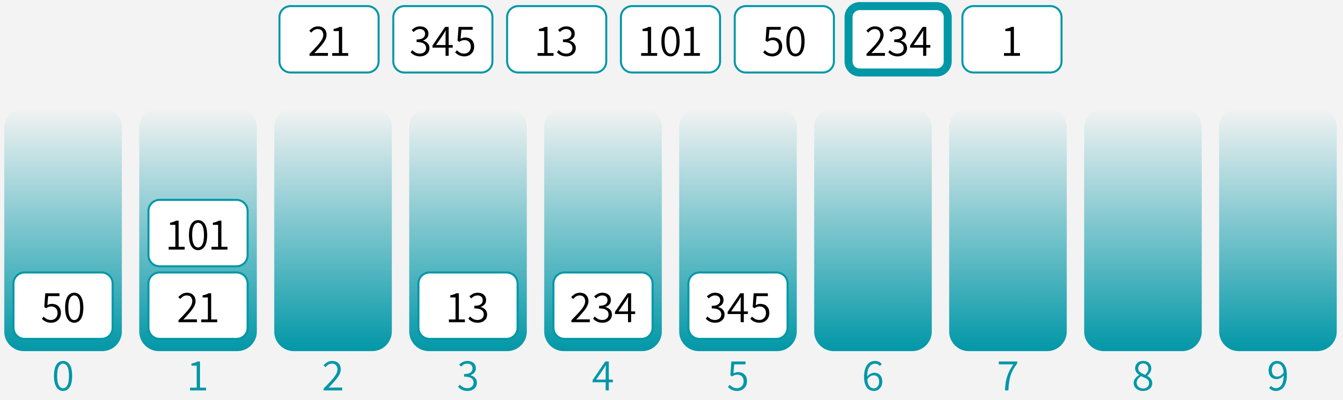
Buckets:

RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

Buckets:



RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:

RADIX SORT

STEP 1: CountingSort on the least significant digit

Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:

Output: 50 | 21 | 101 | 1 | 13 | 234 | 345

When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

QUICK ASIDE: STABLE SORTING

We say a sorting algorithm is STABLE if two objects with equal values appear in the same order in the sorted output as they appear in the input.

Input:	1	2	1	3	2
Sorted Output: (if algorithm is stable)	1	1	2	2	3

The red 1 appeared before the green 1 in the input, so they have to also appear in this order in the output!

The yellow 2 appeared before the purple 2 in the input, so they have to also appear in this order in the output!

RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	1	13	234	345

Buckets:

0	1	2	3	4	5	6	7	8	9

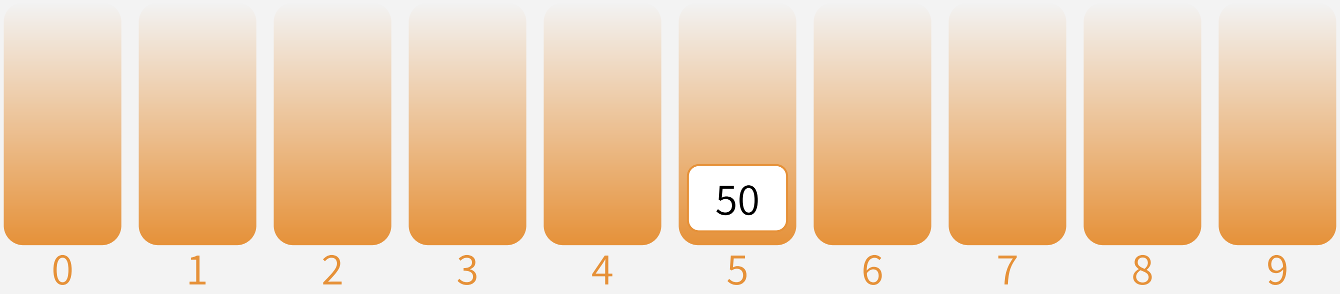
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	1	13	234	345

Buckets:



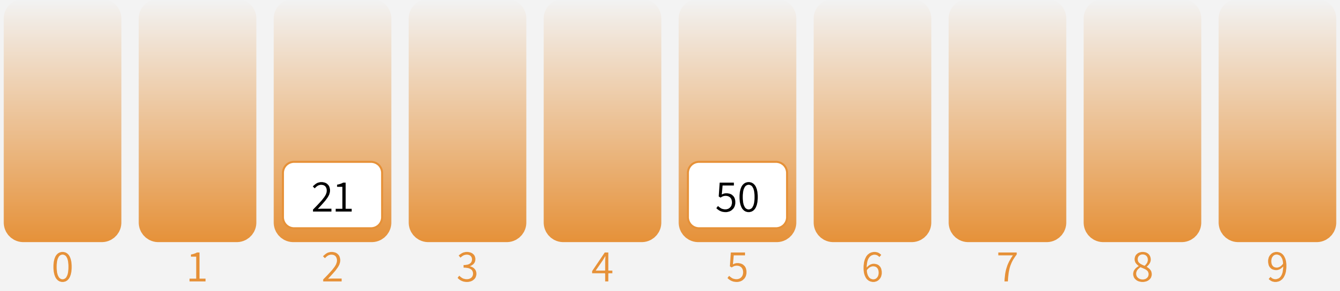
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	**21**	101	1	13	234	345

Buckets:



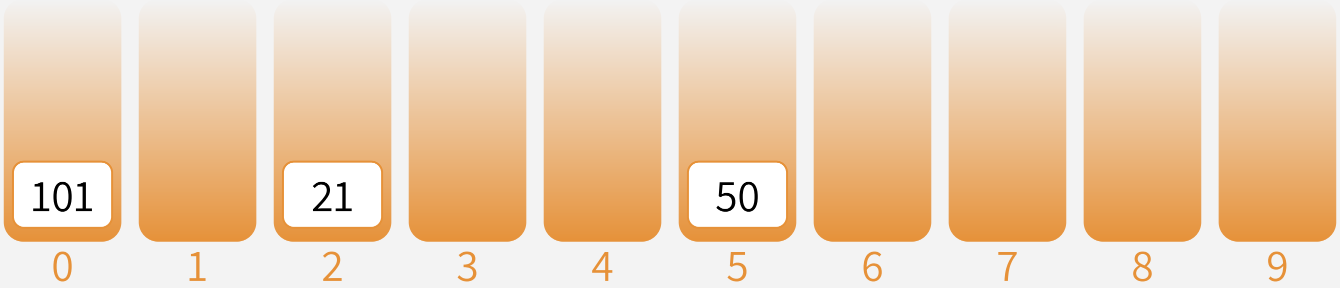
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

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

Buckets:

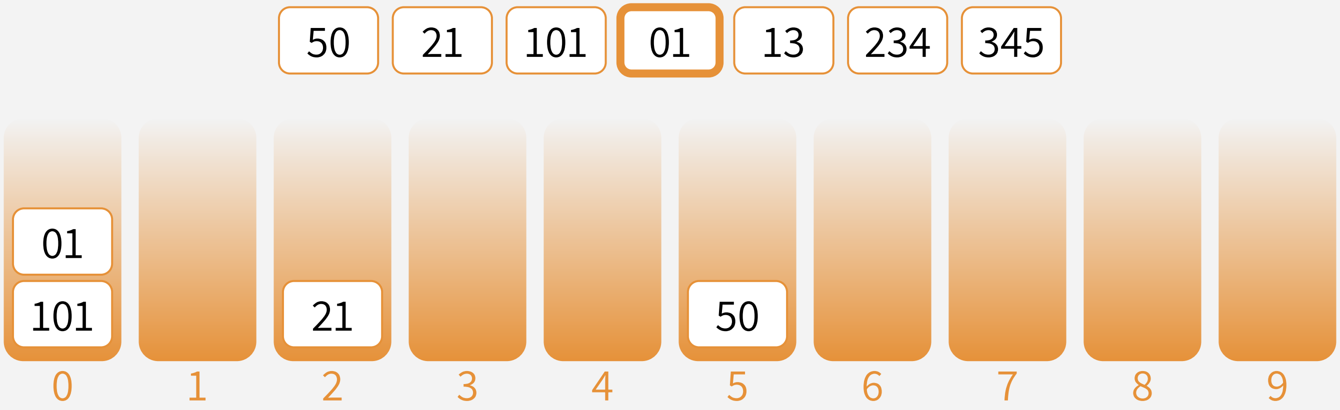


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

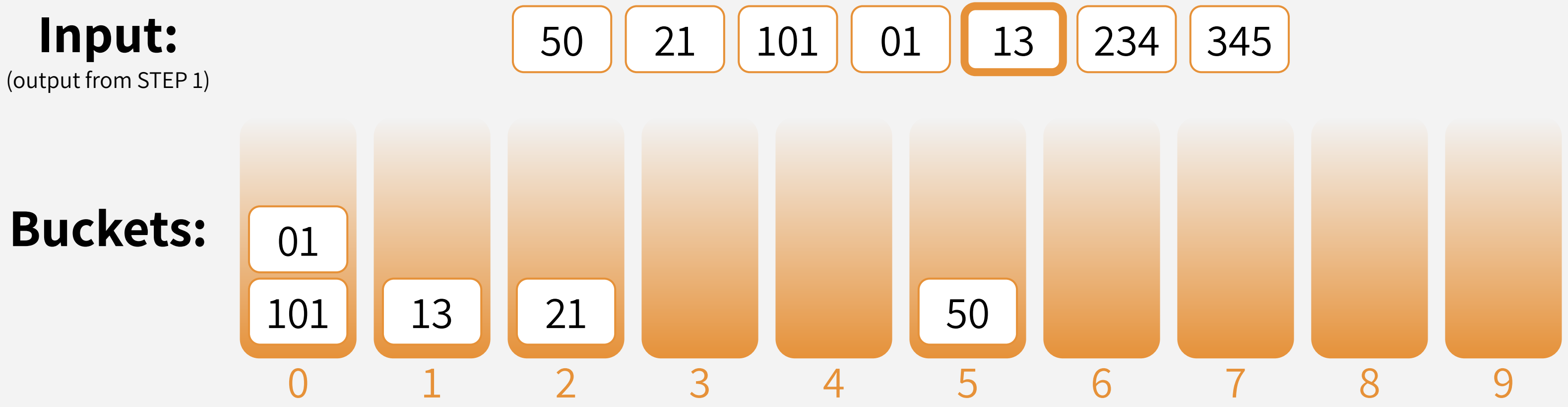
Input:
(output from STEP 1)

Buckets:



RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit



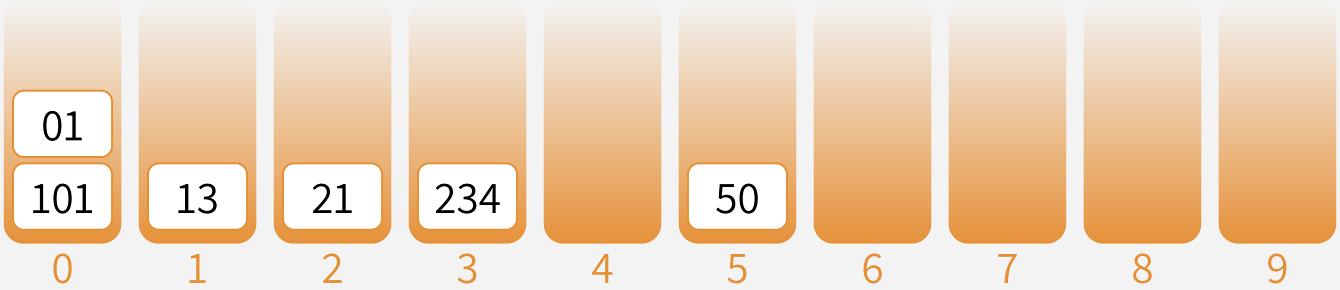
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50 | 21 | 101 | 01 | 13 | 234 | 345

Buckets:



RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	01	13	234	345

Buckets:

0	1	2	3	4	5	6	7	8	9
01									
101	13	21	234	345	50				

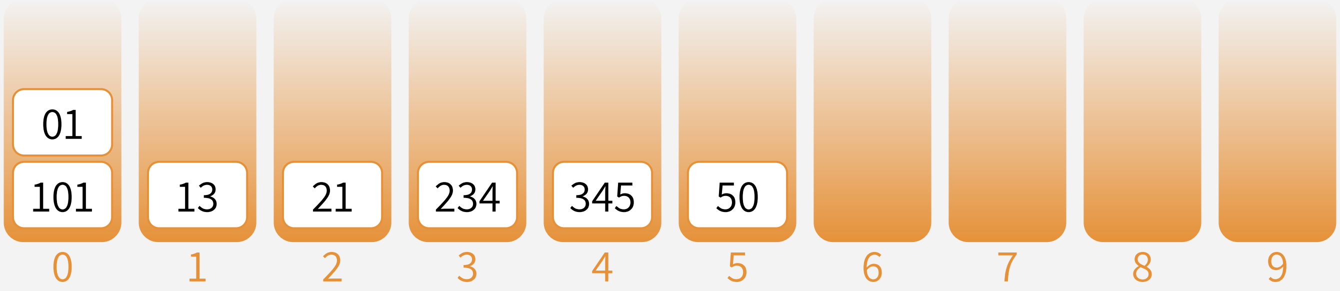
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50 21 101 01 13 234 345

Buckets:



Output:

101 01 13 21 234 345 50

When creating the output list, make sure bucket items exit in FIFO order
(i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	01	13	21	234	345	50

Buckets:

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	01	13	21	234	345	50

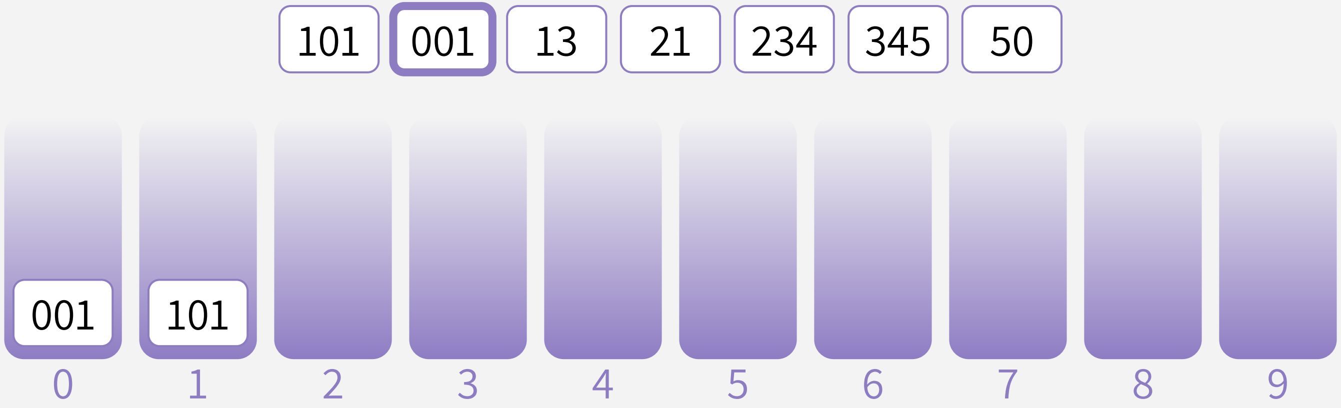
Buckets:

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

Buckets:



RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101 | 001 | 013 | 21 | 234 | 345 | 50

Buckets:

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	234	345	50

Buckets:

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	**234**	345	50

Buckets:

0	1	2	3	4	5	6	7	8	9
021									
013									
001	101	234							

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	234	345	50

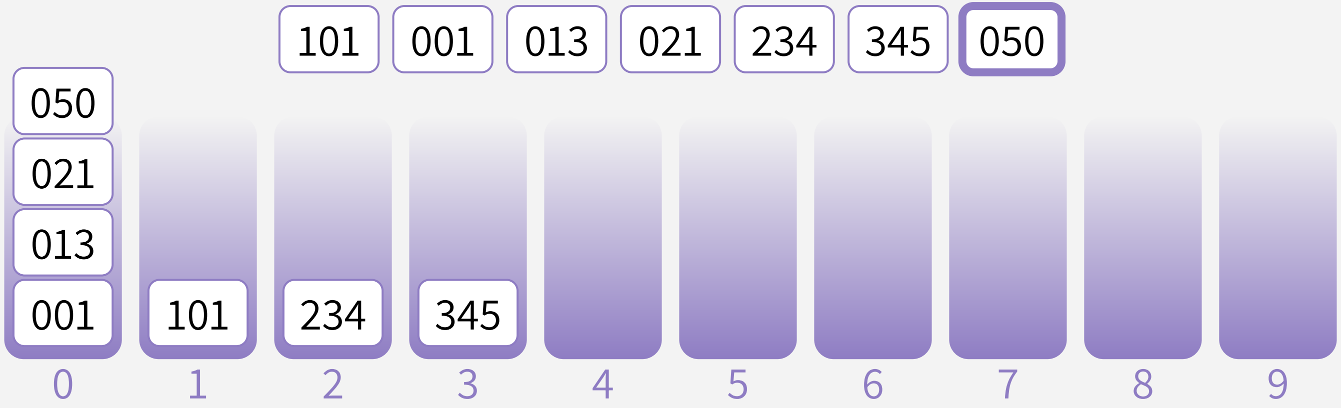
Buckets:

RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

Buckets:



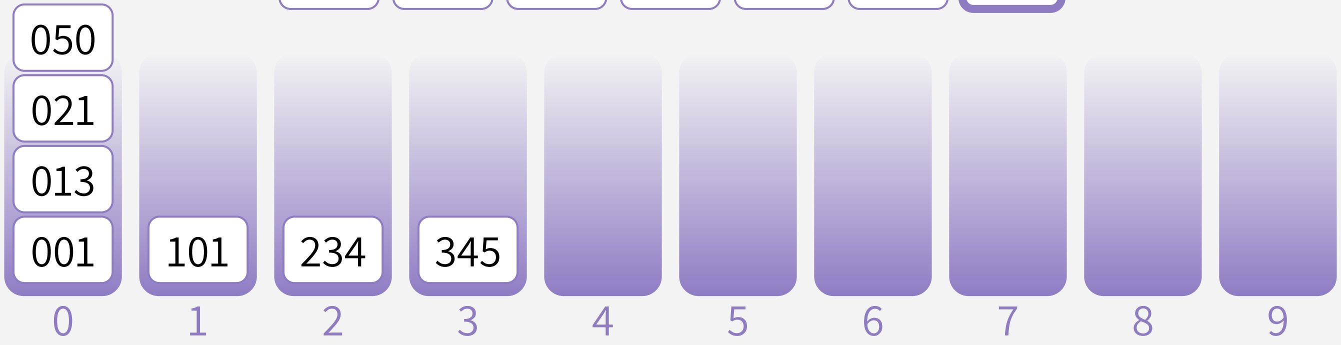
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101 001 013 021 234 345 050

Buckets:



Output:

001 013 021 050 101 234 345

It worked! But why does it work???

سوال؟

درستی مرتب سازی مبنایی

چرا اعداد مرتب شدند؟

WHY DOES RADIX SORT WORK?

Input:	21	345	13	101	50	234	1

WHY DOES RADIX SORT WORK?

Input: 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**

WHY DOES RADIX SORT WORK?

Input:

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

WHY DOES RADIX SORT WORK?

Input:

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 the first THREE digits (aka fully sorted)

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

WHY DOES RADIX SORT WORK?

Proof by Induction!

We'll perform induction on the number of iterations, and we'll use weak induction here:

FROM WEEK ONE!

ITERATIVE ALGORITHMS

1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration **i**
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for k ⇒ prove k+1
4. **Conclusion**: IH holds for i = # total iterations ⇒ yay!

WHY DOES RADIX SORT WORK?

INDUCTIVE HYPOTHESIS (IH)

After the **i**-th iteration, the array A is sorted by the first **i** least-significant digits

WHY DOES RADIX SORT WORK?

INDUCTIVE HYPOTHESIS (IH)

After the **i**-th iteration, the array A is sorted by the first **i** least-significant digits

BASE CASE

The IH holds for i = 0 because A is trivially sorted by 0 least-significant digits.

WHY DOES RADIX SORT WORK?

INDUCTIVE HYPOTHESIS (IH)

After the **i**-th iteration, the array A is sorted by the first **i** least-significant digits

BASE CASE

The IH holds for $i = 0$ because A is trivially sorted by 0 least-significant digits.

INDUCTIVE STEP *(weak induction)*

Let k be an integer, where $0 < k \leq d$ (d is the number of digits). Assume that the IH holds for $i = k-1$, so the array is already sorted by the first k-1 least-significant digits. We need to show that after the k-th iteration, the array is sorted by the first k least-sig. digits.

At a high level, since the “buckets as FIFO-queue” implementation of CountingSort is *stable*, elements that get placed in the same bucket during this k-th round of CountingSort still maintain their previous relative ordering, so they are *still* in order of their k-1 least-sig. digits. Since this k-th round CountingSort sorts A by the k-th digit of the elements, this ultimately means that the elements are going to be sorted by their k least-significant digits.

This can be made more rigorous!

WHY DOES RADIX SORT WORK?

INDUCTIVE HYPOTHESIS (IH)

After the **i**-th iteration, the array A is sorted by the first **i** least-significant digits

BASE CASE

The IH holds for $i = 0$ because A is trivially sorted by 0 least-significant digits.

INDUCTIVE STEP *(weak induction)*

Let k be an integer, where $0 < k \leq d$ (d is the number of digits). Assume that the IH holds for $i = k-1$, so the array is already sorted by the first k-1 least-significant digits. We need to show that after the k-th iteration, the array is sorted by the first k least-sig. digits.

At a high level, since the "buckets as FIFO-queue" implementation of CountingSort is *stable*, elements that get placed in the same bucket during this k-th round of CountingSort still maintain their previous relative ordering, so they are *still* in order of their k-1 least-sig. digits. Since this k-th round CountingSort sorts A by the k-th digit of the elements, this ultimately means that the elements are going to be sorted by their k least-significant digits.

This can be made more rigorous!

CONCLUSION

By induction, we conclude that the IH holds for all $0 \leq i \leq d$. In particular, it holds for $i = d$, so after the last iteration, the array is sorted by all the digits. Hence, it is sorted!

سوال؟

زمان اجرای مرتب سازی مبنایی

چقدر طول می کشد؟

RADIX SORT RUNTIME

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

21	345	13	101	50	234	1

How many iterations are there?

How long does each iteration take?

What is the total running time?

RADIX SORT RUNTIME

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

21	345	13	101	50	234	1

How many iterations are there?

d iterations

How long does each iteration take?

What is the total running time?

RADIX SORT RUNTIME

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

21	345	13	101	50	234	1

How many iterations are there?

d iterations

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets ⇒ **O(n)**

What is the total running time?

RADIX SORT RUNTIME

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

21	345	13	101	50	234	1

How many iterations are there?

d iterations

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets ⇒ **O(n)**

What is the total running time?

O(nd)

HOW GOOD IS O(nd)?

O(nd) isn't so great if we are sorting n integers in base 10, each of which is in {1,2, …,M}:

How many iterations are there?

How long does each iteration take?

What is the total running time?

HOW GOOD IS O(nd)?

O(nd) isn't so great if we are sorting n integers in base 10, each of which is in {1,2, ...,M}:

For example, if M = 1234:

$\lfloor \log_{10} 1234 \rfloor + 1$

$= 3 + 1 = 4$

How many iterations are there?

d = $\boldsymbol{\lfloor \log_{10} M \rfloor + 1}$ iterations

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets ⇒ **O(n)**

What is the total running time?

O(nd) = **O(n log M)**

We just simplified the expression a bit (took out floor and the +1)

HOW GOOD IS O(nd)?

O(nd) isn't so great if we are sorting n integers in base 10, each of which is in {1,2, ...,M}:

For example, if M = 1234:

$\lfloor \log_{10} 1234 \rfloor + 1$
$= 3 + 1 = 4$

How many iterations are there?

d = $\boldsymbol{\lfloor \log_{10} M \rfloor + 1}$ iterations

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets ⇒ **O(n)**

What is the total running time?

O(nd) = **O(n log M)**

We just simplified the expression a bit (took out floor and the +1)

If M is ~n or greater, this is not really any improvement from MergeSort!

سوال؟

HOW GOOD IS O(nd)?

O(nd) isn't so great if we are sorting n ... base 10, each of which is in {1,2, ...,M}:

For example, if M = 1234:

$\lfloor \log_{10} 1234 \rfloor + 1$
$= 3 + 1 = 4$

Ho... re?

Ho... e?
Initialize 10 bu... uckets ⇒ **O(n)**

What ... time?
O(nd) ... **(n log M)**

We just simplified the expression a bit (took out floor and the +1)

THE QUESTION IS...
CAN WE DO BETTER?

If M is ~n or greater, this is not really any improvement from MergeSort!

USING A DIFFERENT BASE

RadixSort with base 10 doesn’t seem so good… How does the base affect the runtime?

USING A DIFFERENT BASE

RadixSort with base 10 doesn't seem so good… How does the base affect the runtime?

Let's say base **r**

How many iterations are there?

d = $\lfloor \log_r M \rfloor + 1$ iterations

How long does each iteration take?

Initialize **r** buckets + put n numbers in **r** buckets ⇒ **O(n + r)**

What is the total running time?

$O(d \cdot (n+r)) = \mathbf{O(\ (\lfloor \log_r M \rfloor + 1) \cdot (n + r))}$

USING A DIFFERENT BASE

RadixSort with base 10 doesn't seem so good… How does the base affect the runtime?

Let's say base **r**

How many iterations are there?

$d = \lfloor \log_r M \rfloor + 1$ iterations

How long does each iteration take?

Initialize **r** buckets + put n numbers in **r** buckets ⇒ **O(n + r)**

What is the total running time?

$O(d \cdot (n+r)) = O\left((\lfloor \log_r M \rfloor + 1) \cdot (n + r) \right)$

Bigger base r ⇒ fewer iterations, but more buckets to initialize!

USING A DIFFERENT BASE

A reasonable sweet spot: **let r = n**

USING A DIFFERENT BASE

A reasonable sweet spot: **let r = n**

How many iterations are there?

$d = \lfloor \log_n M \rfloor + 1$ iterations

How long does each iteration take?

Initialize **n** buckets + put n numbers in **n** buckets ⇒ **O(n+n) = O(n)**

What is the total running time?

$O(d \cdot n) = O((\lfloor \log_n M \rfloor + 1) \cdot n)$

USING A DIFFERENT BASE

A reasonable sweet spot: **let r = n**

How many iterations are there?

d = $\lfloor \log_n M \rfloor + 1$ iterations

How long does each iteration take?

Initialize **n** buckets + put n numbers in **n** buckets ⇒ **O(n+n) = O(n)**

What is the total running time?

$O(d \cdot n) = O((\lfloor \log_n M \rfloor + 1) \cdot n)$

This term is a constant!

If $M \leq n^c$ for some constant c, then $O((\lfloor \log_n M \rfloor + 1) \cdot n) = O(n)$

USING A DIFFERENT BASE

A reasonable sweet spot: **let r = n**

This means that the running time of RadixSort using a base of **r = n** (instead of base 10 from earlier examples) depends on how big M is in terms of n. The formula is:

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

This is O(n) when $M \leq n^c$.

The number of buckets need is r = n.

If $M \leq n^c$ for some constant c, then $O((\lfloor \log_n M \rfloor + 1) \cdot n) = O(n)$

RADIX SORT RECAP

Radix Sort can sort **n integers of size at most n^{100}** (or n^{c} for any constant c) in time **O(n)**.

RADIX SORT RECAP

Radix Sort can sort **n integers of size at most n^{100}** (or n^C for any constant c) in time **O(n)**.

If your sorting task involves integers that have size much bigger than n (or n^C), like 2^n, maybe you shouldn't use Radix Sort because you wouldn't get linear time.

RADIX SORT RECAP

Radix Sort can sort **n integers of size at most n^{100}** (or n^C for any constant c) in time **O(n)**.

If your sorting task involves integers that have size much bigger than n (or n^C), like 2^n, maybe you shouldn't use Radix Sort because you wouldn't get linear time.

It matters how you pick the base! In general, if you have **n** elements, **M** = max size of any element, and **r** is the base:

Runtime of Radix Sort = $\mathbf{O(\ (\lfloor \log_r M \rfloor + 1) \cdot n)}$

WHY BOTHER WITH COMPARISON-BASED SORTING?

Comparison-based sorting algorithms can handle arbitrary comparable elements!
And with numbers, it can handle sorting with high precision & arbitrarily large values:

π	$\frac{1234}{9876}$	e	43!	4.10598425	n^n	31

Radix Sort requires us to look at all digits, which is problematic — π and e both have infinitely many! And n^n is big enough to make Radix Sort slow...

Radix Sort is also not in place (you need those buckets!), so it could require more space.

سوال؟