

ساختمان داده و الگوریتم ها (CE203)

جلسه هفتم: کران پایین برای
مرتب سازی و مرتب سازی خطی

سجاد شیرعلی شهرضا

پاییز 1401

دوشنبه، 25 مهر 1401

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 8.1
- یادآوری مهلت ارسال تمرین اول: 8 صبح شنبه 30 مهر
- مهلت نظرسنجی دوم: 8 صبح شنبه 30 مهر

کران پایین برای مرتب سازی

آیا می توان الگوریتم مرتب سازی بهتر از $O(n \lg n)$ هم طراحی کرد؟

$O(n \log n)$ ALGORITHMS WE'VE SEEN

- MergeSort
 - Worst-case $\Theta(n \log n)$ time.
- QuickSort
 - Expected: $\Theta(n \log n)$

$O(n \log n)$ ALGORITHMS WE'VE SEEN

- MergeSort
 - Worst-case $\Theta(n \log n)$ time
- QuickSort
 - Expected: $\Theta(n \log n)$

THE QUESTION IS...
***CAN WE DO
BETTER?***

INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

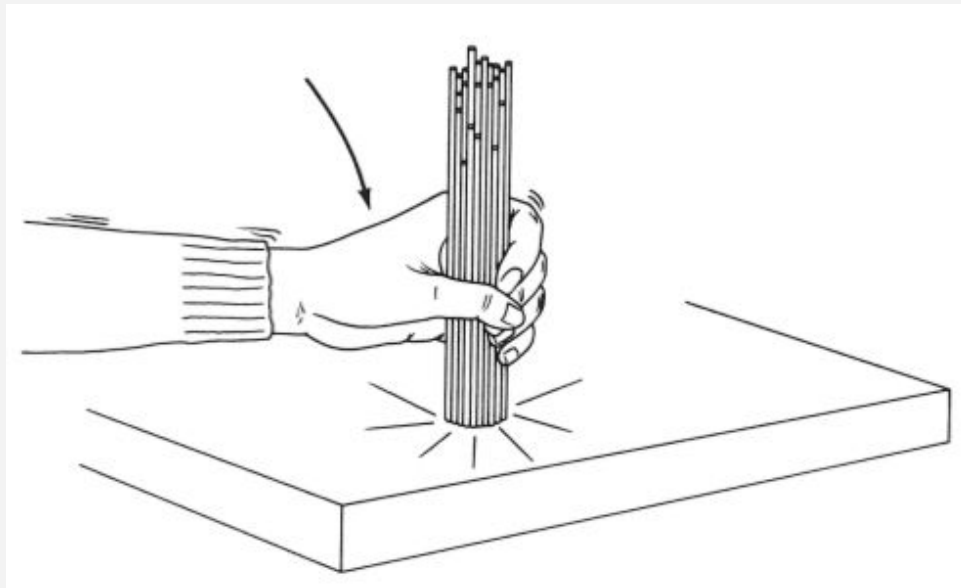
- For each number, break off a piece of spaghetti whose length is that number $O(n)$

INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number $O(n)$
- Take all the spaghetti in your fist, and push their lower sides against the table $O(1)$

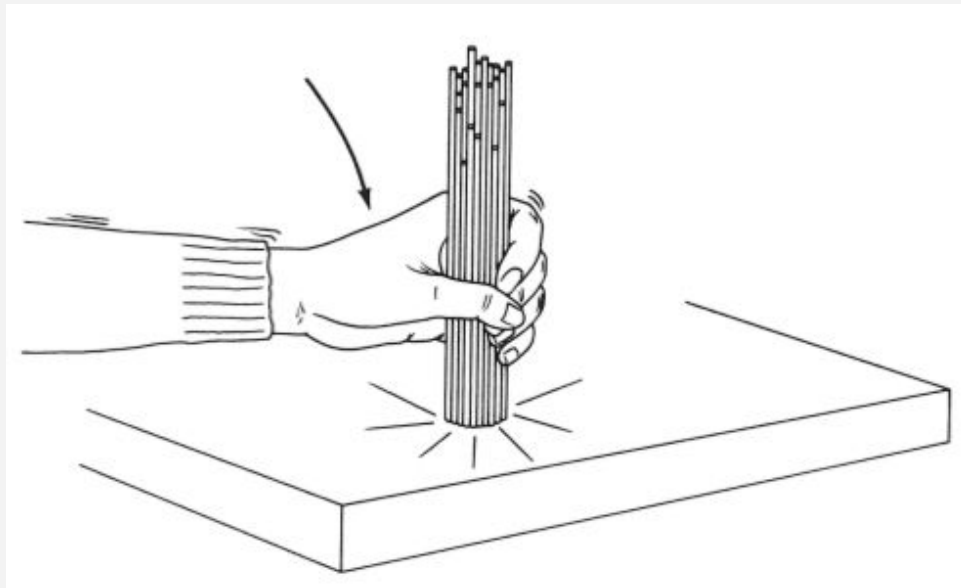


INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number $O(n)$
- Take all the spaghetti in your fist, and push their lower sides against the table $O(1)$
- Lower your other hand on the bundle of spaghetti - the first spaghetti you touch is the longest one. Remove it, transcribe its length, and repeat until all spaghetti have been removed. $O(n)$

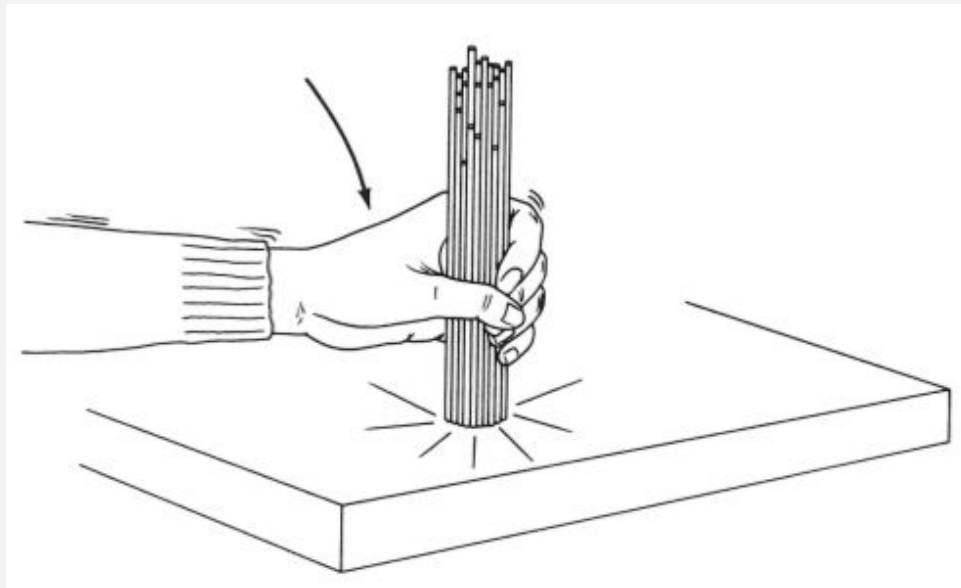


INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number $O(n)$
- Take all the spaghetti in your fist, and push their lower sides against the table $O(1)$
- Lower your other hand on the bundle of spaghetti - the first spaghetti you touch is the longest one. Remove it, transcribe its length, and repeat until all spaghetti have been removed. $O(n)$



While you shouldn't take this algorithm too seriously... it does raise some important questions!



سوال؟

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

Input: some real numbers

Output: sorted real numbers

Operations allowed: breaking spaghetti,
dropping on tables, lowering hand

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

Input: some real numbers

Output: sorted real numbers

Operations allowed: breaking spaghetti,
dropping on tables, lowering hand

In a CS class where we're more concerned with what computers can do, the first model seems more reasonable.



سوال؟

COMPARISON-BASED SORTING

- **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.**
- Examples: Insertion Sort, MergeSort, QuickSort

COMPARISON-BASED SORTING

- **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.**
- Examples: Insertion Sort, MergeSort, QuickSort

“Comparison-based sorting algorithms” are general-purpose.

The algorithm makes no assumption about the input elements other than that they belong to some totally ordered set.

COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

$A[0]$

$A[1]$

$A[2]$

$A[3]$

COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

$A[0]$

$A[1]$

$A[2]$

$A[3]$

Is $A[1]$ bigger than $A[3]$?

Yes!

A Comparison-based
Sorting Algorithm

*All-knowing
Genie*

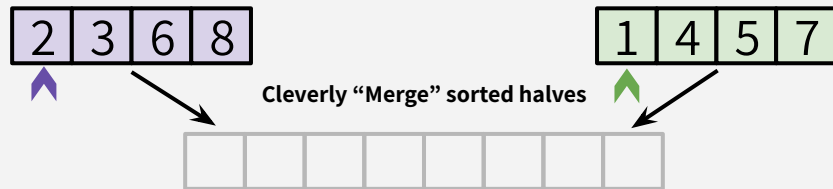
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



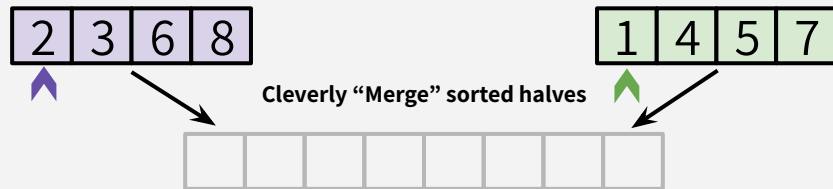
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **1**?

MergeSort
algorithm

*All-knowing
Genie*

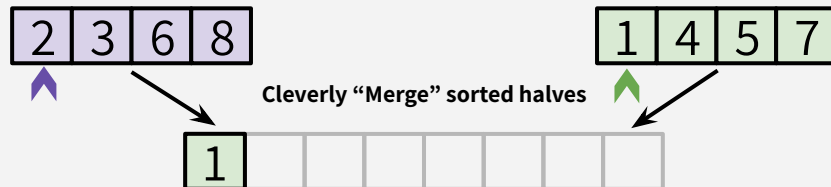
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **1**?

Yes!

MergeSort
algorithm

*All-knowing
Genie*

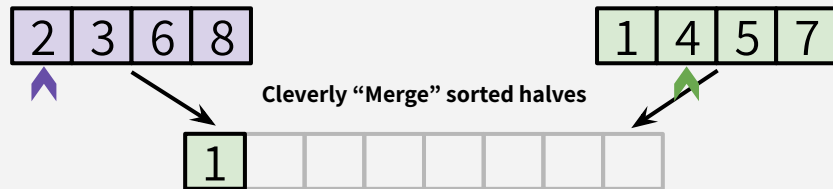
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this genie this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **4**?

MergeSort
algorithm

*All-knowing
Genie*

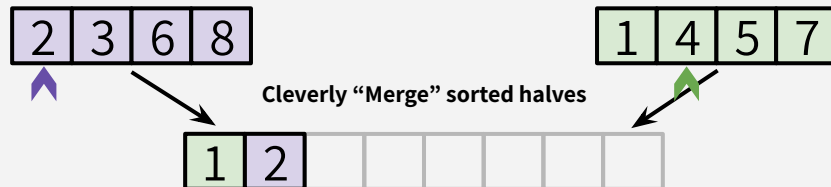
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **2** bigger than **4**?

MergeSort
algorithm

No!

*All-knowing
Genie*

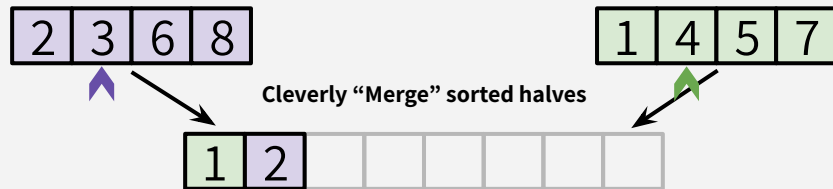
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is 3 bigger than 4 ?

MergeSort
algorithm

*All-knowing
Genie*

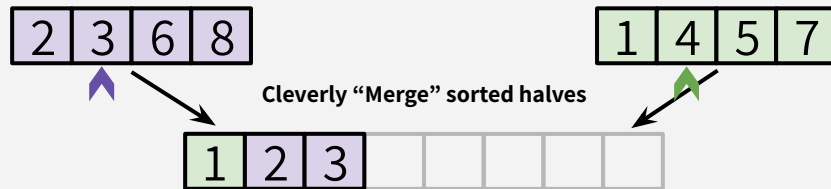
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **3** bigger than **4**?

MergeSort
algorithm

No!

*All-knowing
Genie*

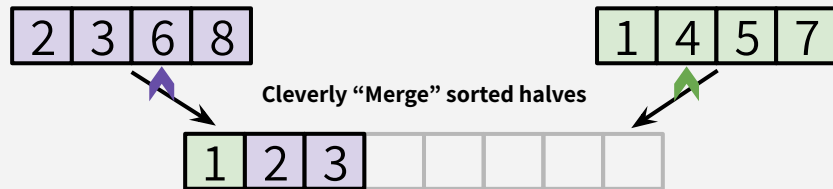
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this *genie* this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **6** bigger than **4**?

MergeSort
algorithm

*All-knowing
Genie*

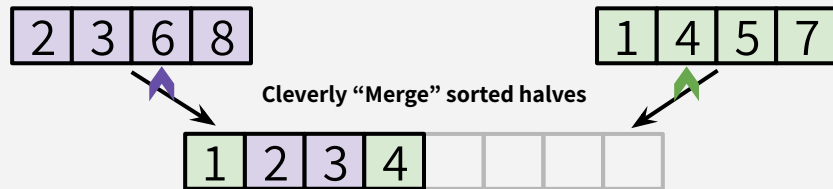
COMPARISON-BASED SORTING

In other words, the only way you can interact with the array:

For two indices i and j , is $A[i]$ bigger than $A[j]$?

(I find it helpful to imagine that there is a *genie* who knows what the right order is, and you can only ask this YES/NO question to figure out how to sort the items)

For example, MergeSort works like this:



Is **6** bigger than **4**?

MergeSort
algorithm

Yes!

*All-knowing
Genie*

COMPARISON-BASED SORTING

Theorem:

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

PROOF IDEA

Theorem:

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

- Any deterministic comparison-based algorithm can be represented as a decision tree with $n!$ leaves
- The worst-case runtime is at least the length of the longest path in the decision tree
- All decision trees with $n!$ leaves have a longest path with length at least $\log(n!) = \Omega(n \log n)$
- So, any comparison-based sorting algorithm must have worst-case runtime at least $\Omega(n \log n)$

THE GOOD NEWS

Theorem:

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

This bound also applies to the expected runtime of *randomized* comparison-based sorting algorithms!
The proof is out of scope of this class, but it relies on this theorem.

This means that MergeSort is optimal!

(This is one of the cool things about proving lower bounds - we know when we can declare victory!)

THE GOOD NEWS

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

This bound also applies to
The proof is by induction.

THE QUESTION IS...
**CAN WE DO
BETTER?**

-based sorting algorithms!
theorem.

This means that the algorithm is optimal!

**using a model of computation that's
less silly than spaghetti?*

(This is one of the reasons why we don't prove lower
bounds - we know when we can declare victory!)



سوال؟

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:

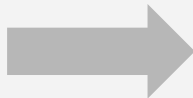
arbitrary elements whose values
we could never directly access,
process, or take advantage of
(i.e. we could only interact with
them via comparisons)

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

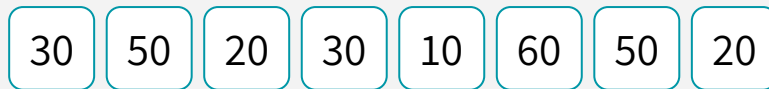
20

COUNTING SORT

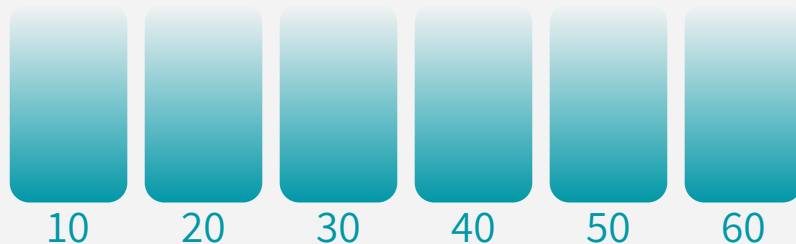
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:



Buckets:

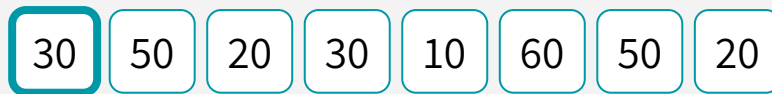


COUNTING SORT

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:



Buckets:

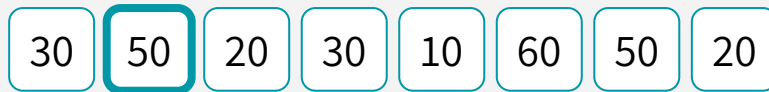


COUNTING SORT

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:



Buckets:

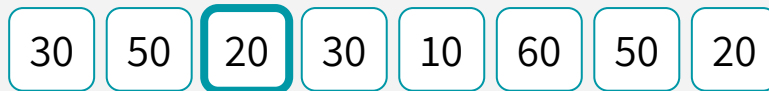


COUNTING SORT

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:



Buckets:

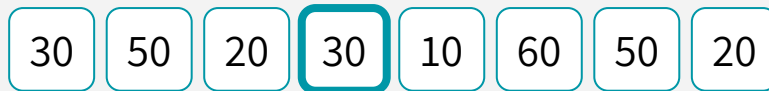


COUNTING SORT

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:



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:



Buckets:

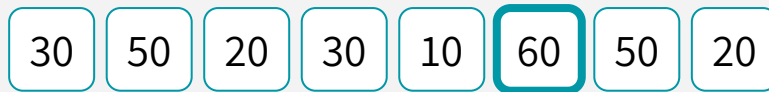


COUNTING SORT

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:



Buckets:

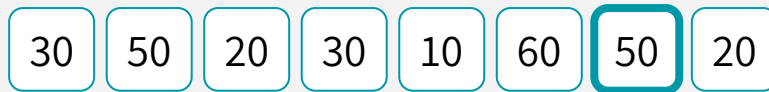


COUNTING SORT

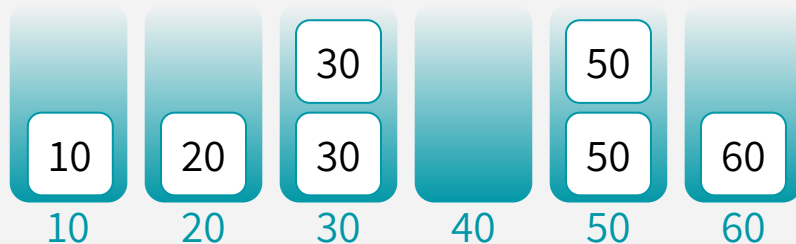
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:



Buckets:

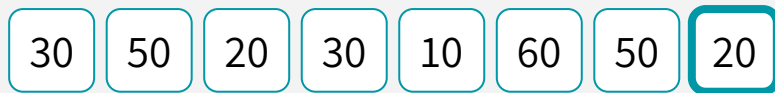


COUNTING SORT

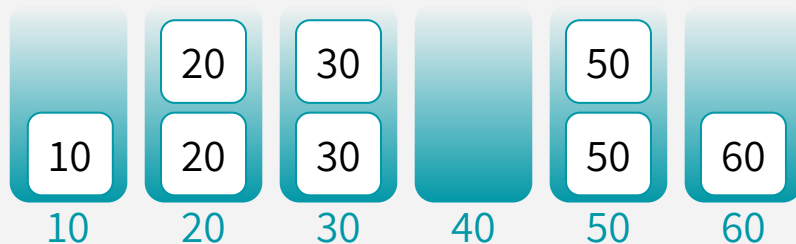
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:



Buckets:

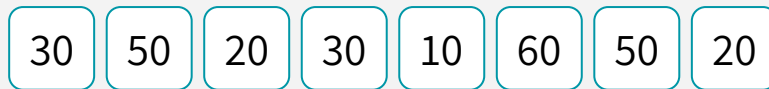


COUNTING SORT

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:



Buckets:



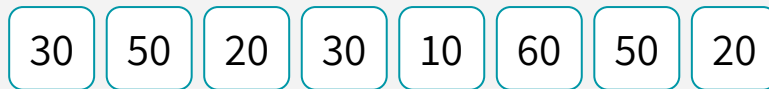
Output:

COUNTING SORT

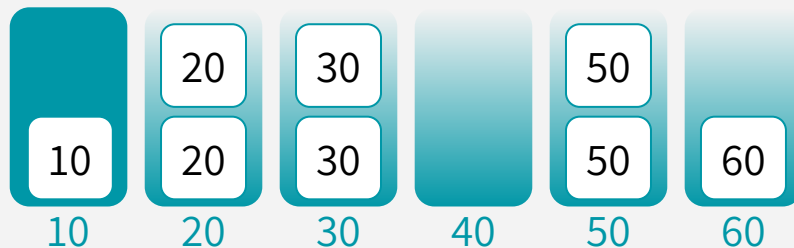
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:



Buckets:



Output:

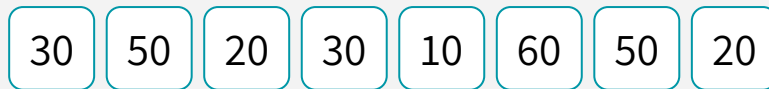


COUNTING SORT

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 $\{10, 20, 30, 40, 50, 60\}$

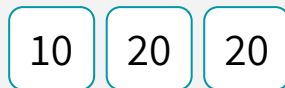
Input:



Buckets:



Output:

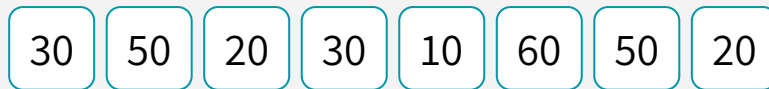


COUNTING SORT

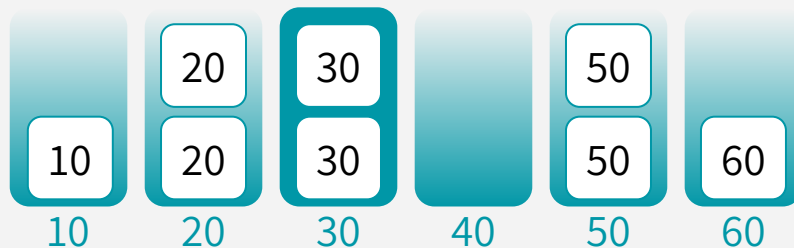
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

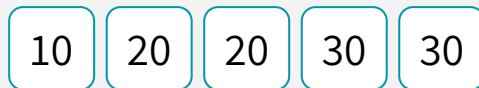
Input:



Buckets:



Output:

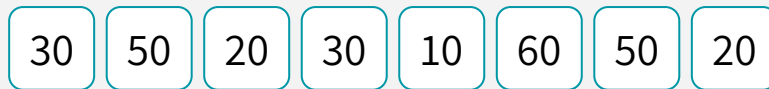


COUNTING SORT

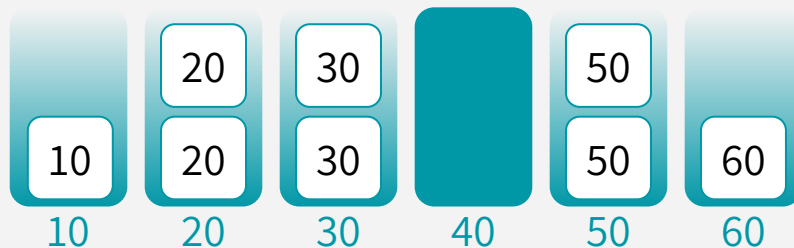
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

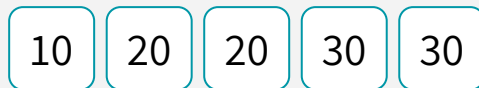
Input:



Buckets:



Output:

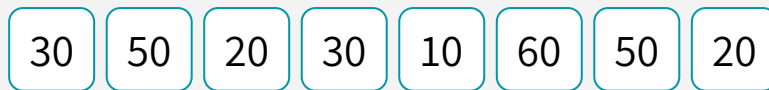


COUNTING SORT

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:



Buckets:



Output:

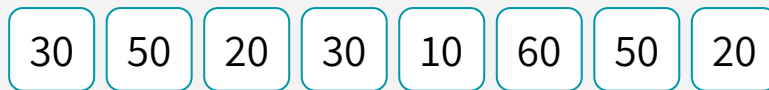


COUNTING SORT

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 $\{10, 20, 30, 40, 50, 60\}$

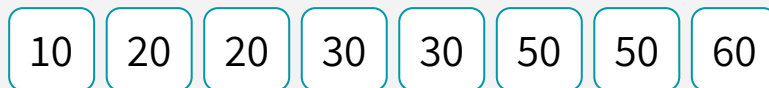
Input:



Buckets:



Output:

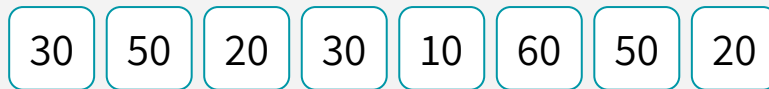


COUNTING SORT

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:



Buckets:



Output:



Sorted in time:
 $O(n)$

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

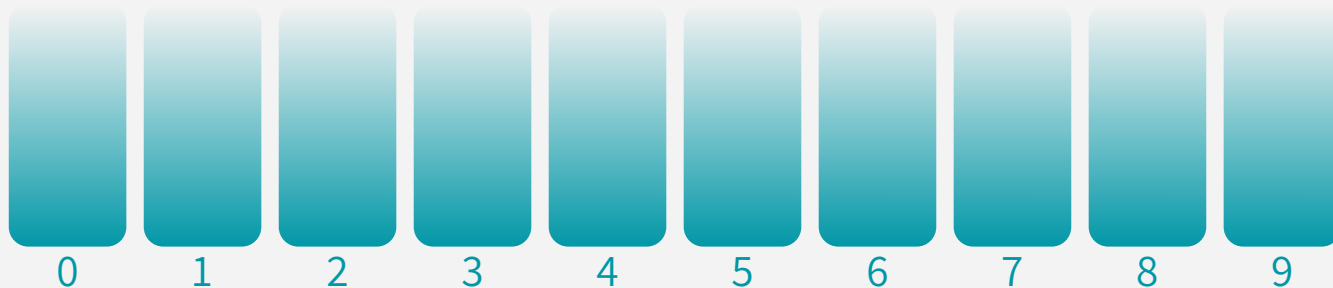
RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:

21 345 13 101 50 234 1

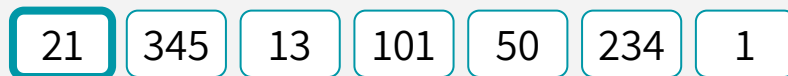
Buckets:



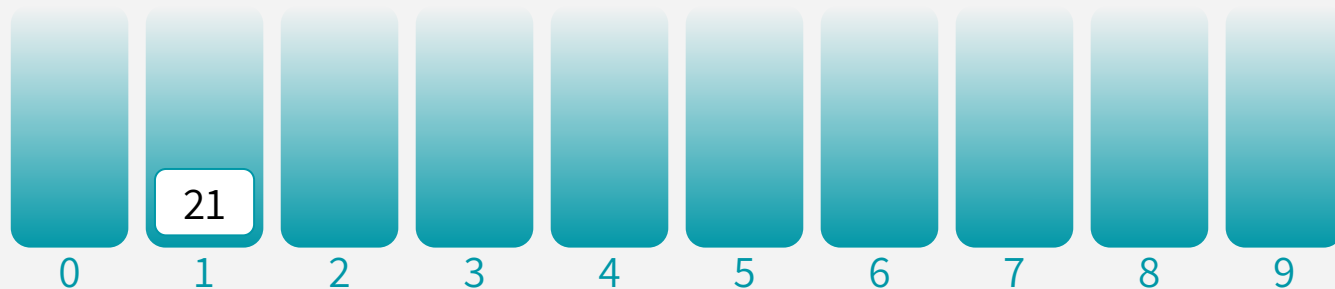
RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:



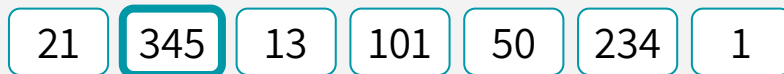
Buckets:



RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:



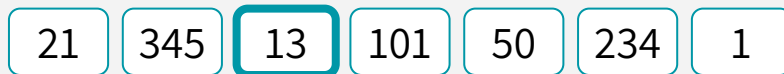
Buckets:



RADIX SORT

STEP 1: CountingSort on the least significant digit

Input:



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:



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:

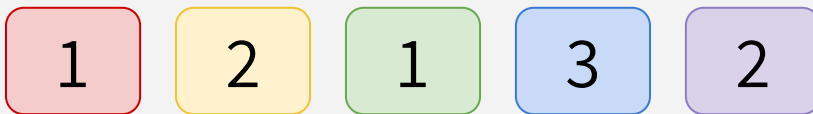
50 21 101 1 13 234 345

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)

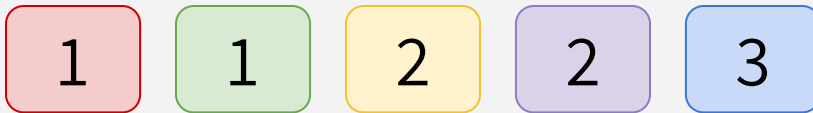
STABLE SORTING

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:



Sorted Output:
(if algorithm is stable)



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

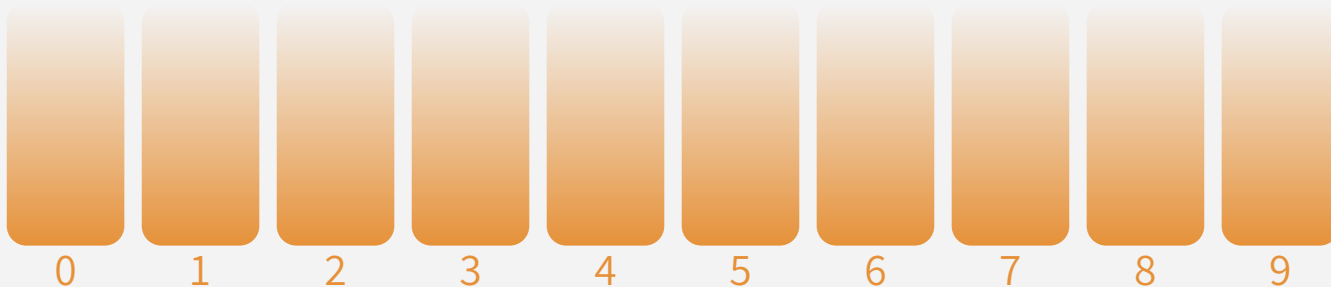
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 1 13 234 345

Buckets:



RADIX SORT

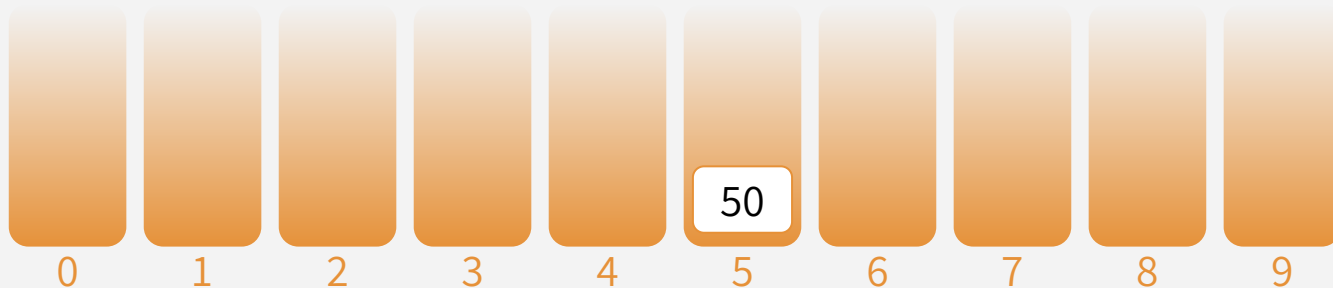
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 1 13 234 345

Buckets:



RADIX SORT

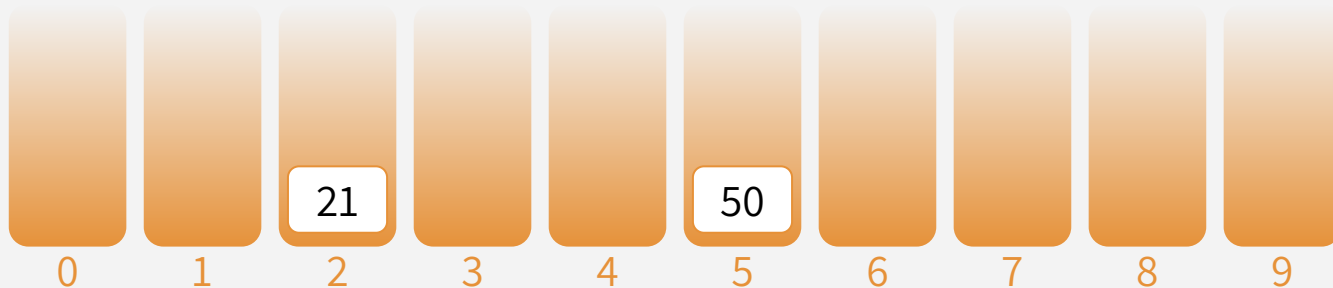
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 1 13 234 345

Buckets:



RADIX SORT

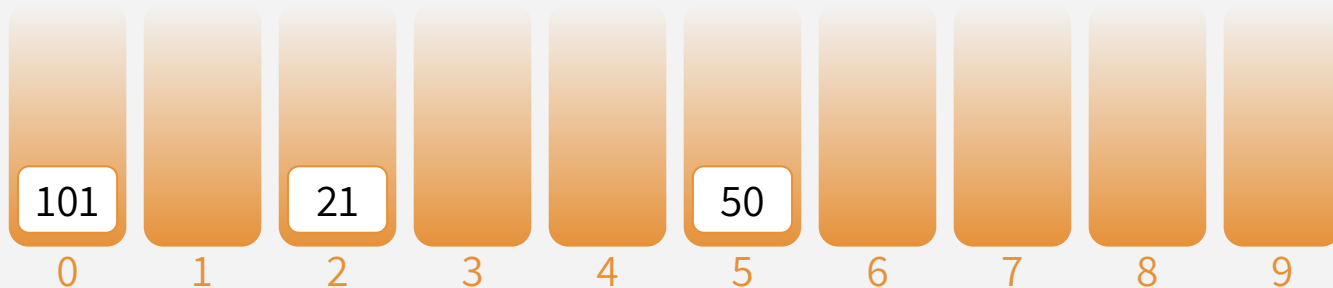
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 1 13 234 345

Buckets:



RADIX SORT

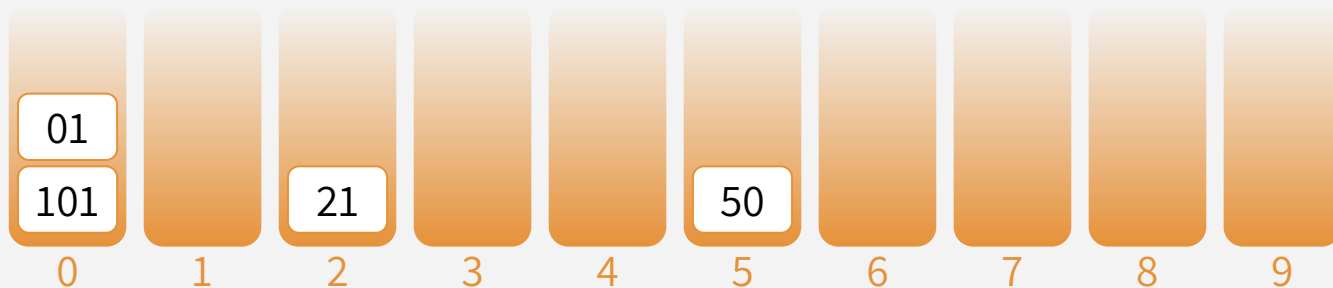
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 01 13 234 345

Buckets:



RADIX SORT

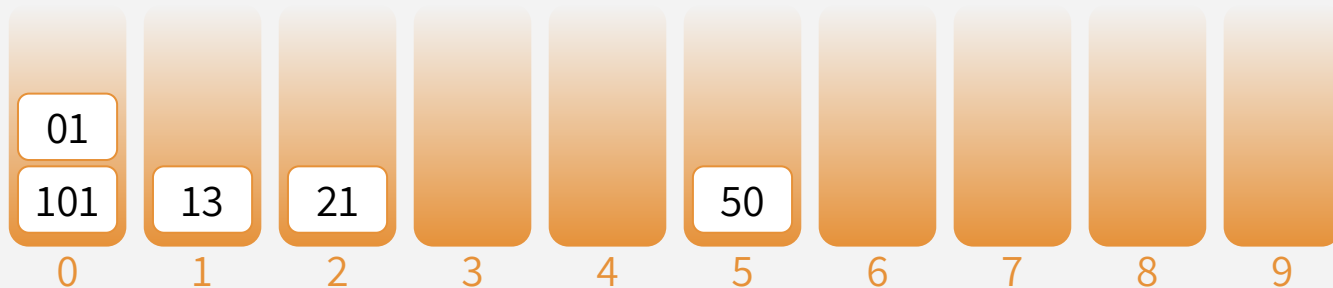
STEP 2: CountingSort on the 2nd least significant digit

Input:

(output from STEP 1)

50 21 101 01 13 234 345

Buckets:



RADIX SORT

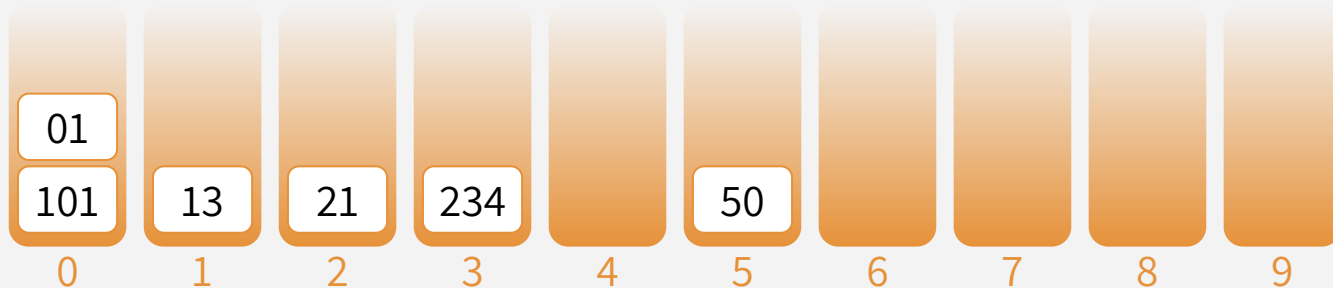
STEP 2: CountingSort on the 2nd least significant digit

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:



RADIX SORT

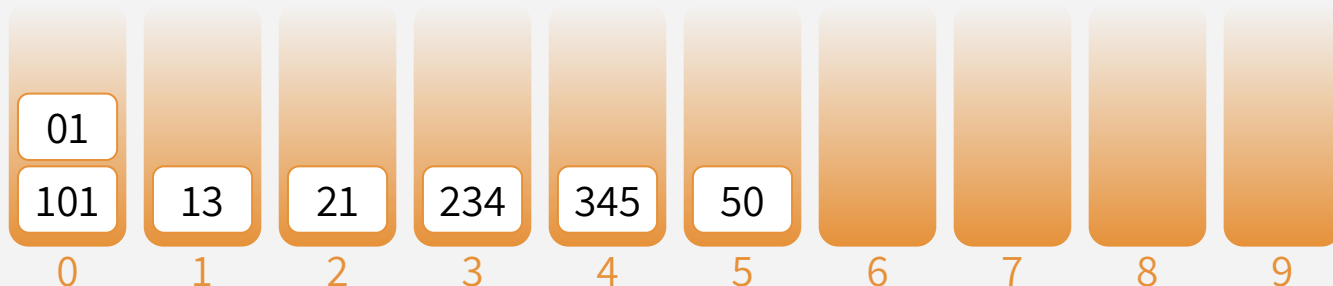
STEP 2: CountingSort on the 2nd least significant digit

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)

101 001 13 21 234 345 50

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:



RADIX SORT

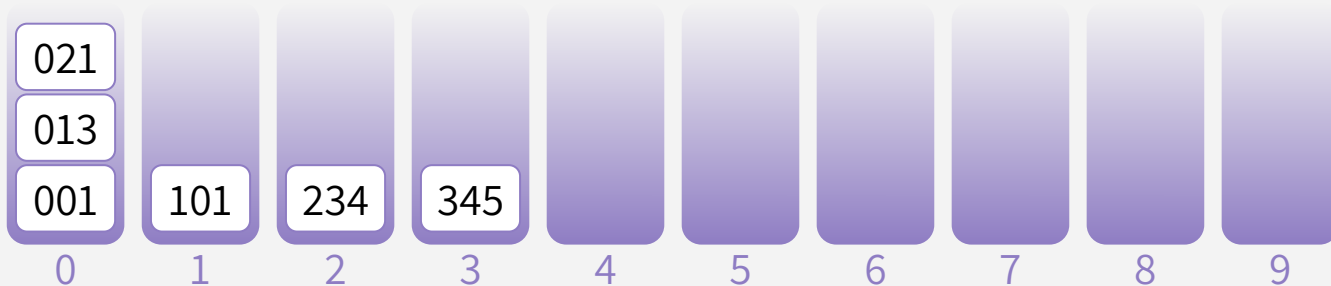
STEP 3: CountingSort on the 3rd least significant digit

Input:

(output from STEP 2)

101 001 013 021 234 345 50

Buckets:

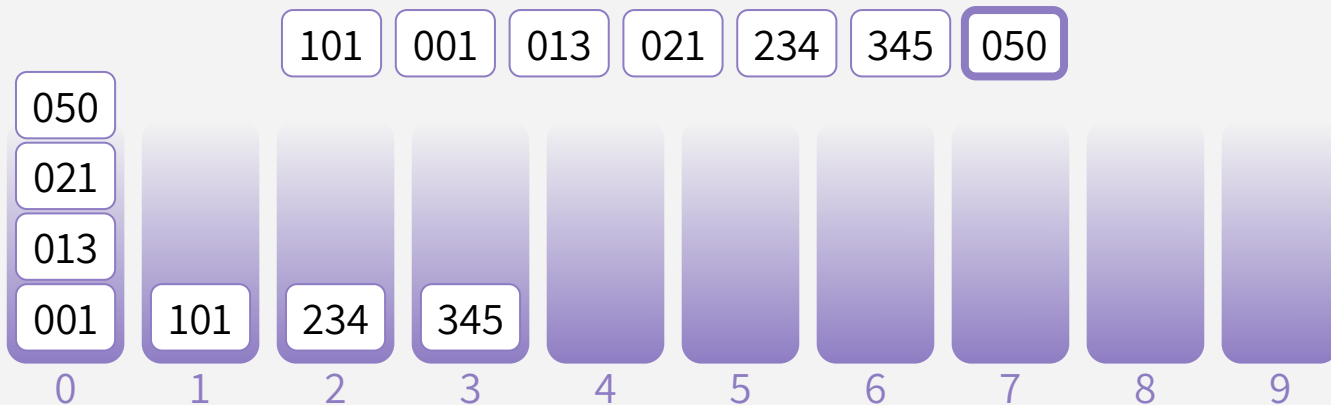


RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

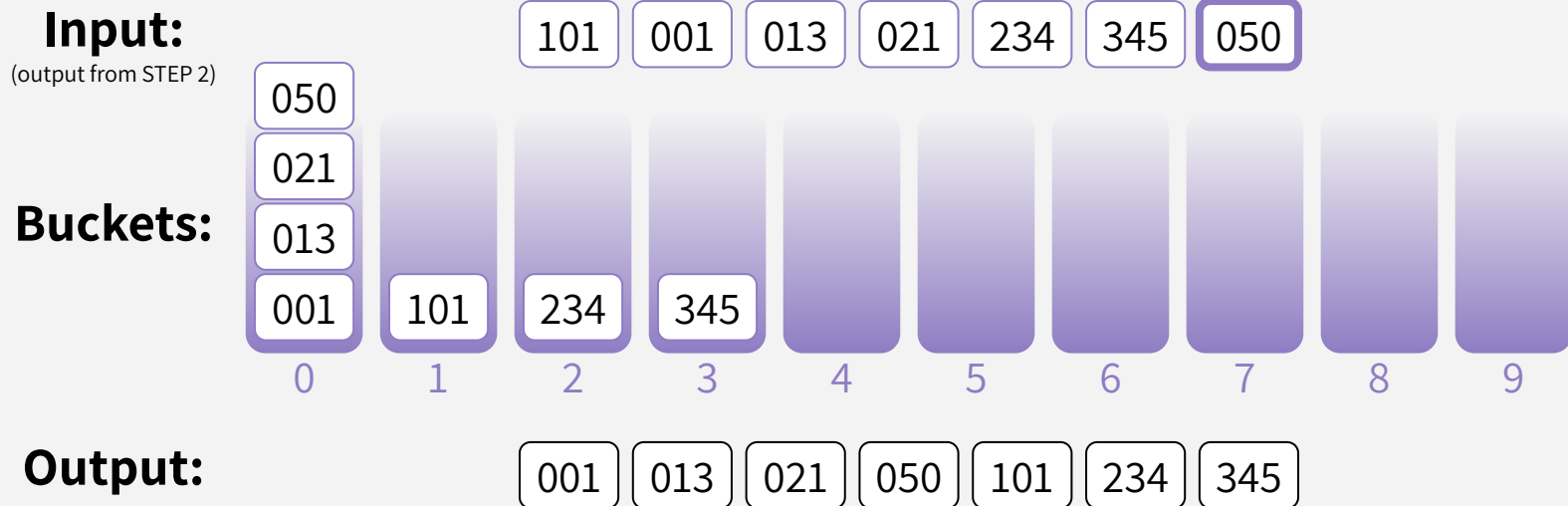
Input:
(output from STEP 2)

Buckets:



RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit



It worked!



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 \Rightarrow **$O(n)$**

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 \Rightarrow **$O(n)$**

What is the total running time?

$O(nd)$

HOW GOOD IS $O(nd)$?

Sorting n integers in **base 10**, each of which is in $\{1, 2, \dots, M\}$:

For example, if $M = 1234$:

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

How many iterations are there?

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

How long does each iteration take?

Initialize 10 buckets + put n numbers in 10 buckets $\Rightarrow O(n)$

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)

USING A DIFFERENT BASE

Let's say base r

How many iterations are there?

$$d = \lfloor \log_r M \rfloor + 1 \text{ iterations}$$

How long does each iteration take?

Initialize r buckets + put n numbers in r buckets $\Rightarrow O(n + r)$

What is the total running time?

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

USING A DIFFERENT BASE

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

How many iterations are there?

$$d = \lfloor \log_n M \rfloor + 1 \text{ iterations}$$

How long does each iteration take?

Initialize n buckets + put n numbers in n buckets $\Rightarrow O(n+n) = O(n)$

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?

$$d = \lfloor \log_n M \rfloor + 1 \text{ iterations}$$


How long does each iteration take?

Initialize n buckets + put n numbers in n buckets $\Rightarrow O(n+n) = O(n)$

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

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؟