Advanced Data Structure and Algorithm

Sorting Lower Bounds & Linear Sorting

LAST TIME

- Randomized Algorithms
- BogoSort & **QuickSort**!

WHAT WE'LL COVER TODAY

- Sorting lower bounds
  - What model of computation have we been working with?
- Linear-Time sorting!

SORTING LOWER BOUNDS

We’ve seen O(n log n) sorting algorithms… can we do better?

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 Θ(n log n) time.
- QuickSort
  - Expected: Θ(n log n)

THE QUESTION IS…

CAN WE DO BETTER?

WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

COMPARISON-BASED SORTING

- **You want to sort an array of items**
- **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 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 genie 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:

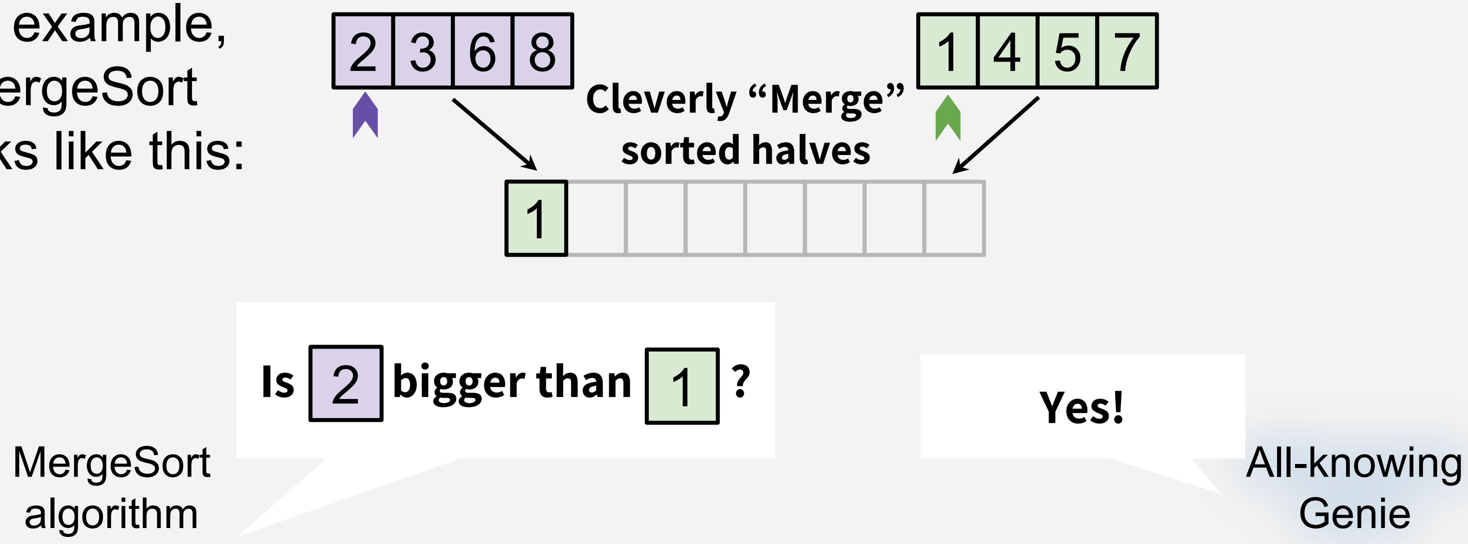
COMPARISON-BASED SORTING

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:



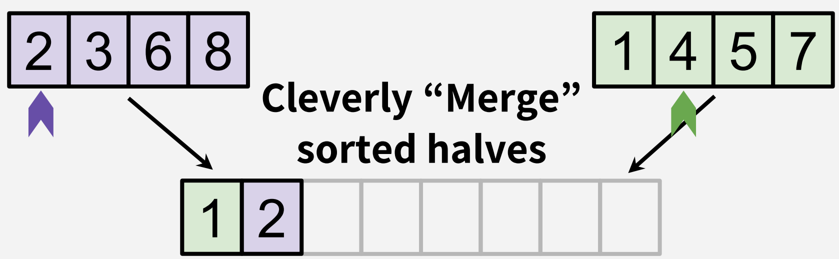
COMPARISON-BASED SORTING

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 **?**

No!

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

No!

MergeSort algorithm

All-knowing Genie

COMPARISON-BASED SORTING

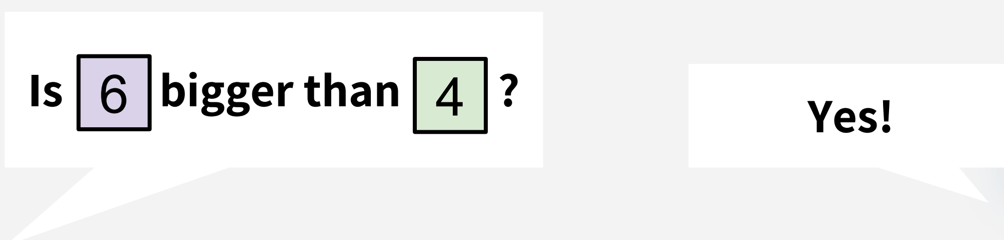
In other words, the only way you can interact with the array:

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:

MergeSort algorithm



All-knowing Genie

COMPARISON-BASED SORTING

Theorem:

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

COMPARISON-BASED SORTING

Theorem:

Any deterministic comparison-based sorting algorithm must take Ω(n log n) time.

Think about it like this: this is the input format that your algorithm is ready to accept.

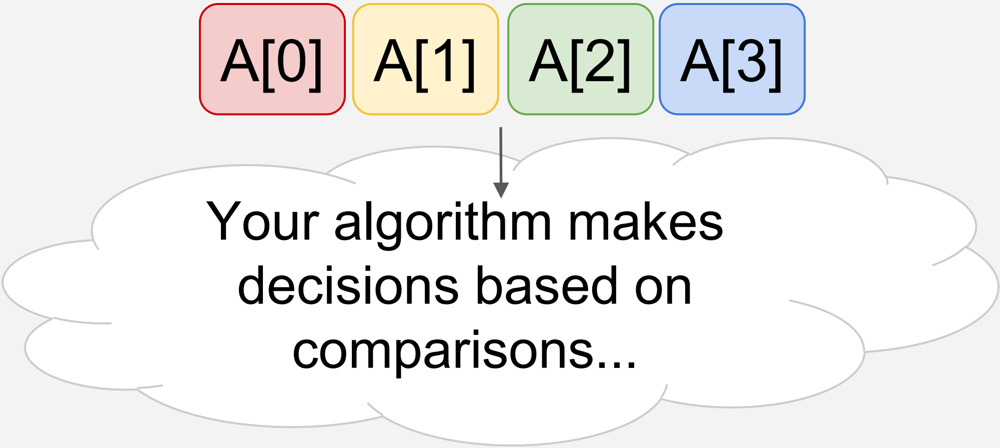
A[0]	A[1]	A[2]	A[3]

COMPARISON-BASED SORTING

Theorem:

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

Think about it like this: this is the input format that your algorithm is ready to accept.

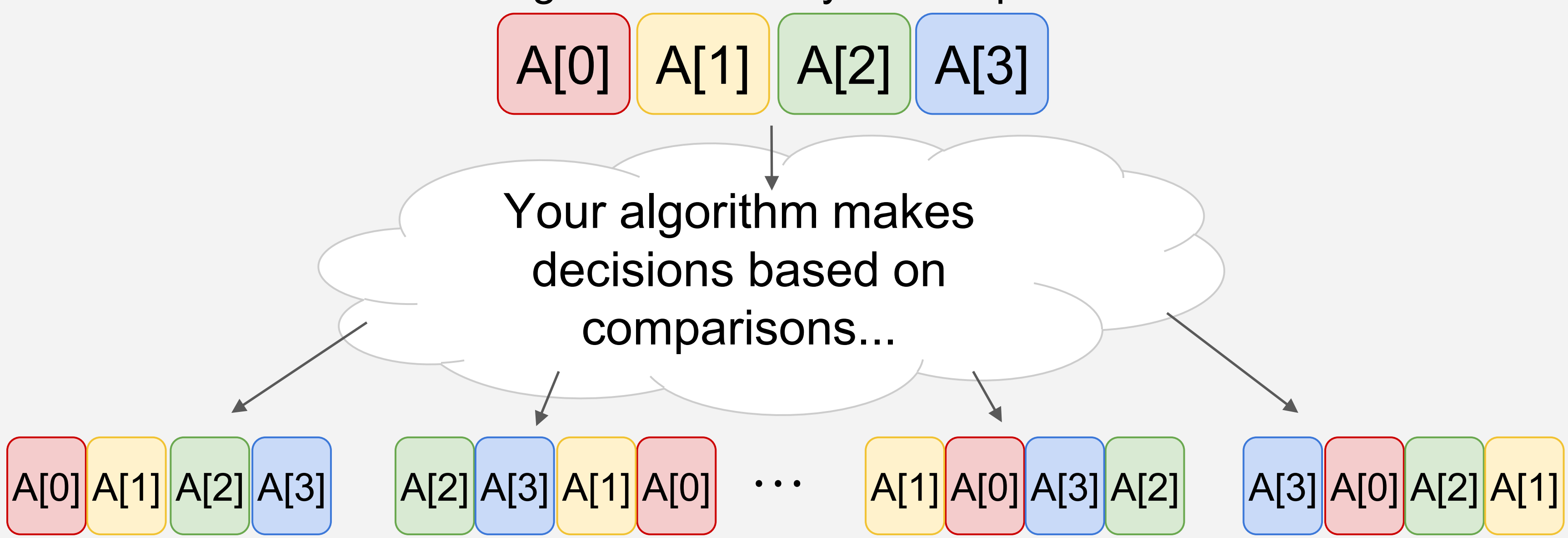


COMPARISON-BASED SORTING

Theorem:

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

Think about it like this: this is the input format that your algorithm is ready to accept.



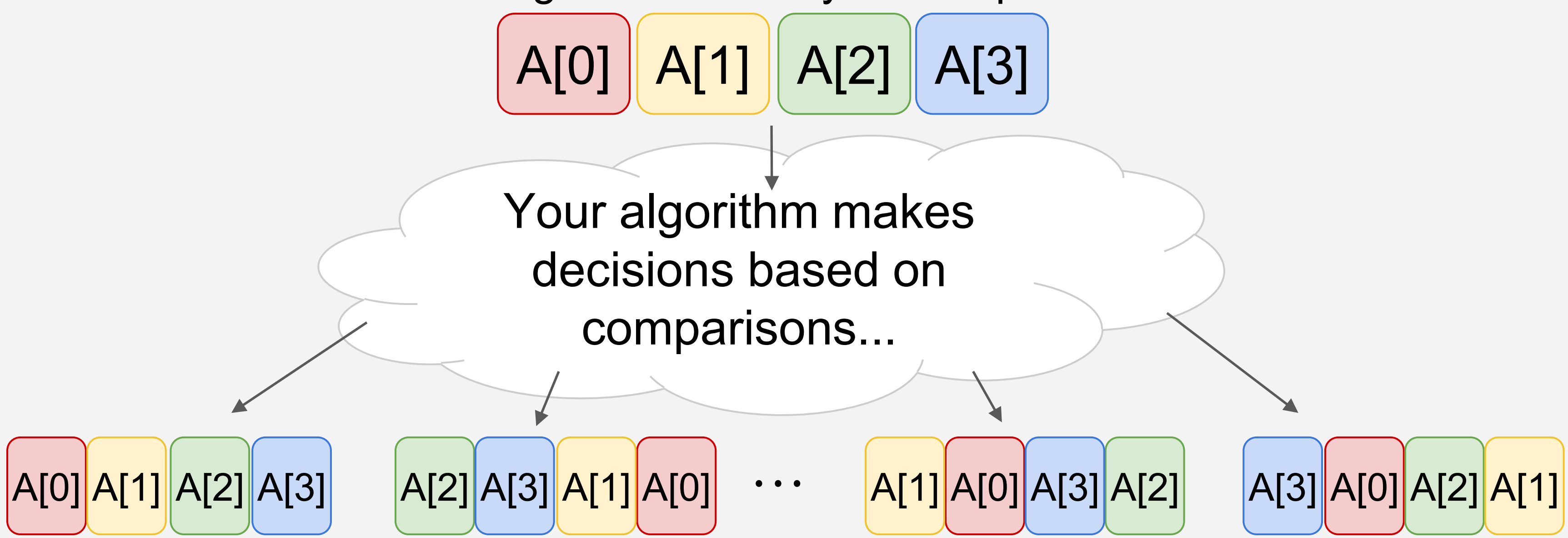
Your algorithm needs to be able to output any one of ______ possible orderings

COMPARISON-BASED SORTING

Theorem:

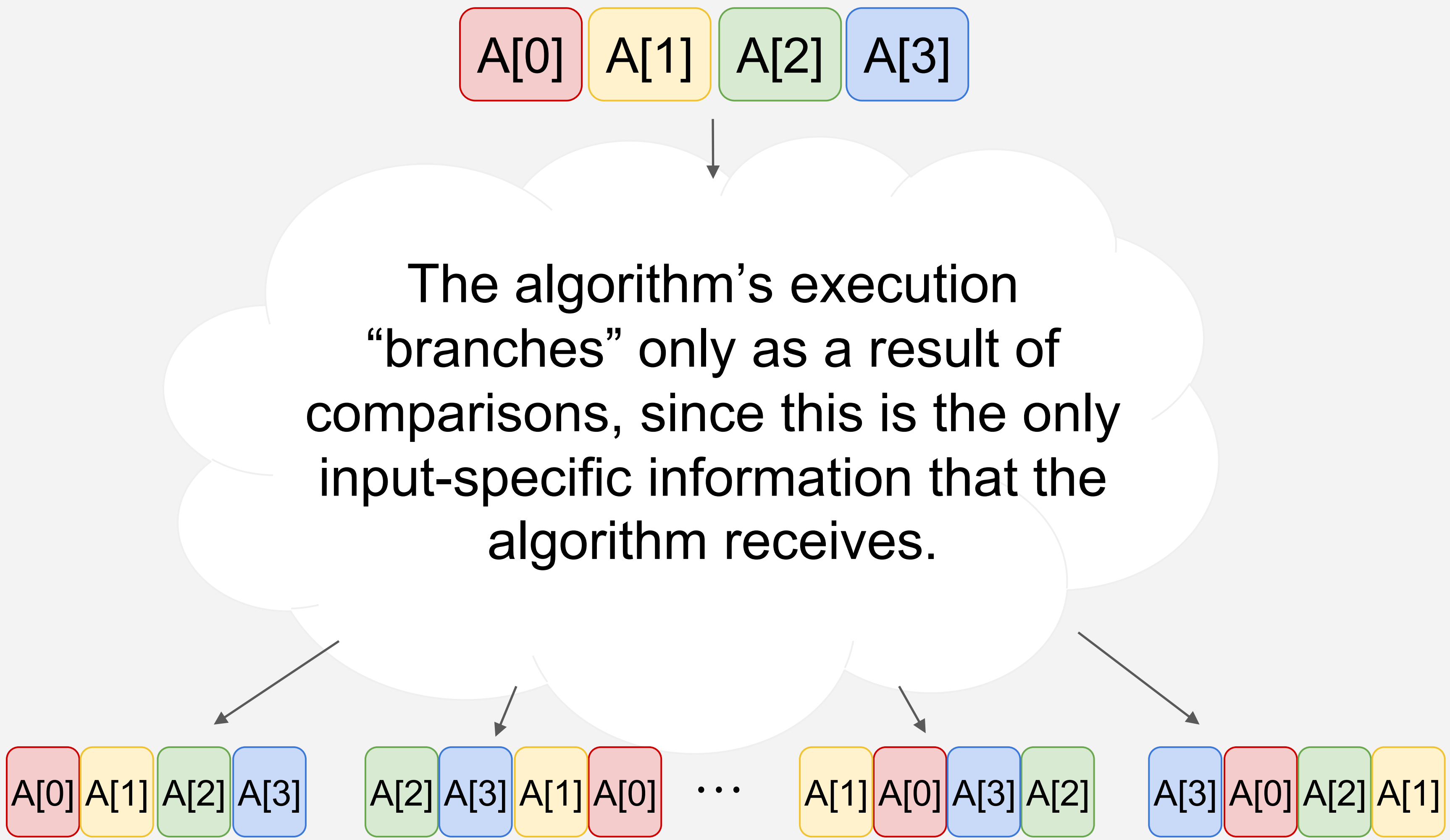
Any deterministic comparison-based sorting algorithm must take Ω(n log n) time.

Think about it like this: this is the input format that your algorithm is ready to accept.



Your algorithm needs to be able to output any one of **n!** possible orderings

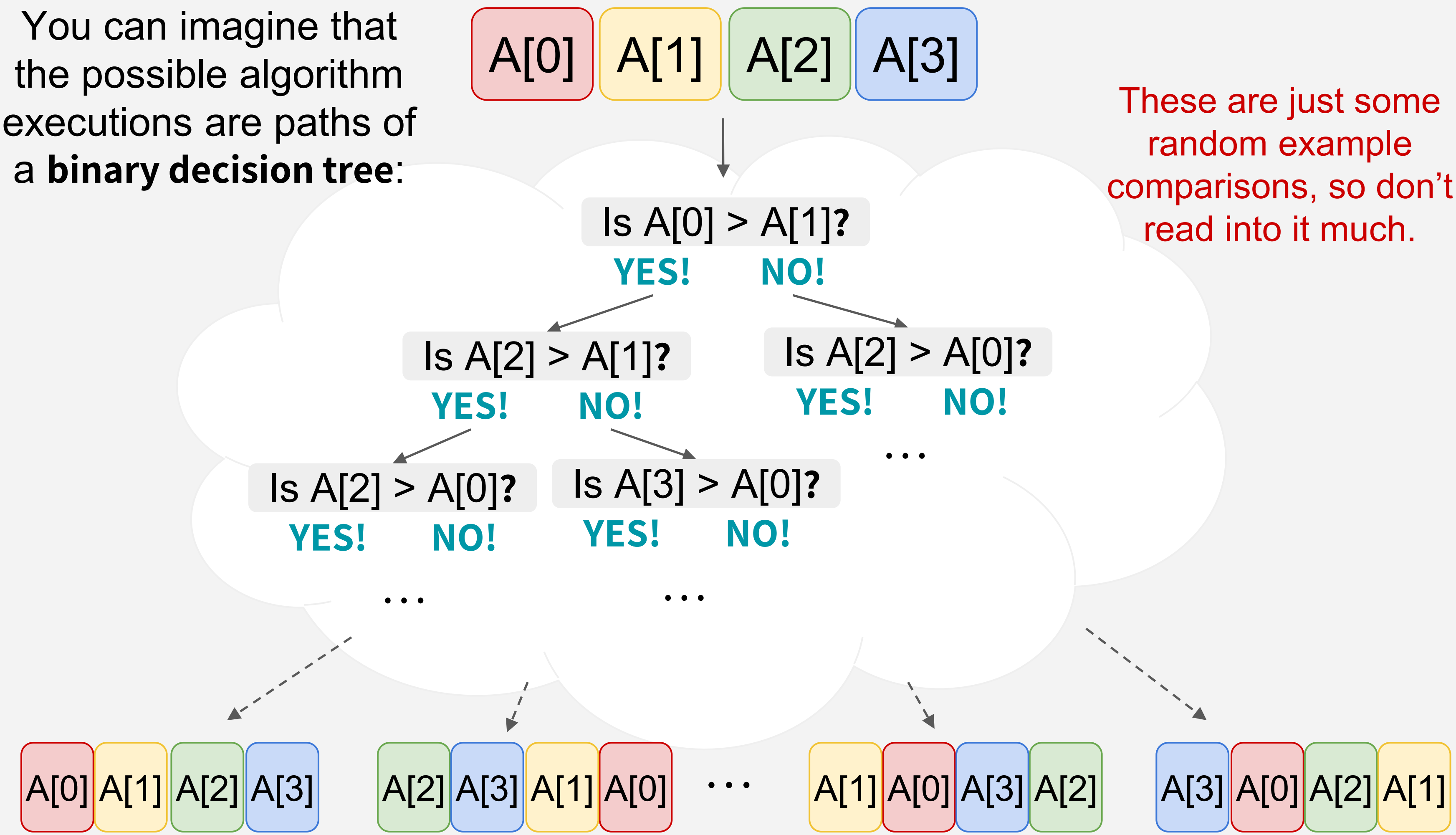
COMPARISON-BASED SORTING



Your algorithm needs to be able to output any one of **n!** possible orderings

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of a **binary decision tree**:

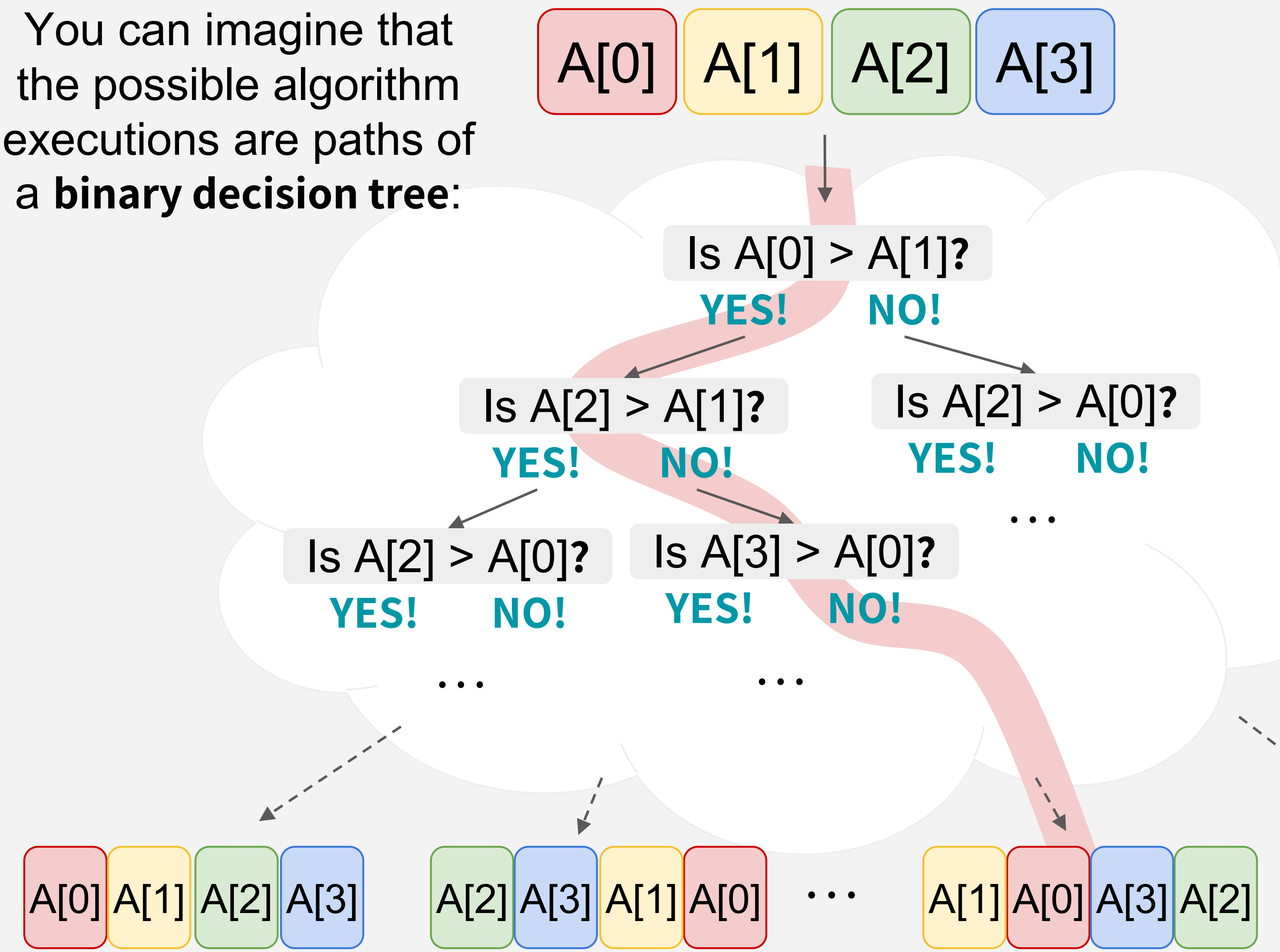


These are just some random example comparisons, so don't read into it much.

Your algorithm needs to be able to output any one of __**n!**__ possible orderings

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of a **binary decision tree**:



Every possible execution of the algorithm is represented as one path from the **root** to a **leaf**

Your algorithm needs to be able to output any one of __**n!**__ possible orderings

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of a **binary decisi**

A[0] A[1] A[2] A[3]

This is a binary tree with at least **n!** leaves.

What is the length of the longest possible path?

ery possible
xecution of
e algorithm
epresented
s one path
m the **root**
to a **leaf**

A[0] A[1] A[2] A[3]   A[2] A[3] A[1] A[0]   · · ·   A[1] A[0] A[3] A[2]   A[3] A[0] A[2] A[1]

Your algorithm needs to be able to output any one of __**n!**__ possible orderings

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of

A[0] A[1] A[2] A[3]

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!)

Thus, in all binary trees with at least n! leaves, **the longest path has length at least log(n!)**

A[0] A[1] A[2] A[3]   A[2] A[3] A[1] A[0]   . . .   A[1] A[0] A[3] A[2]   A[3] A[0] A[2] A[1]

Your algorithm needs to be able to output any one of __**n!**__ possible orderings

COMPARISON-BASED SORTING

The longest path has length at least log(n!)

Consequently, any execution of a comparison-based sorting algorithm has to perform at least log(n!) steps.

The worst-case runtime is at least log(n!) = Ω(n log n).

COMPARISON-BASED SORTING

$$\begin{aligned}
\log n! &= \log(1 \cdot 2 \cdot 3 \cdots n) \\
&= \log 1 + \log 2 + \log 3 + \cdots + \log n \\
&= \log 1 + \cdots + \log \tfrac{n}{2} + \cdots + \log n \\
&\geq \log \tfrac{n}{2} + \log\left(\tfrac{n}{2} + 1\right) + \cdots + \log n && \text{(i.e., the larger half of the sum)} \\
&\geq \log\left(\tfrac{n}{2}\right) + \log\left(\tfrac{n}{2}\right) + \cdots + \log\left(\tfrac{n}{2}\right) && \text{(adding } \tfrac{n}{2} \text{ times)} \\
&= \log\left(\tfrac{n}{2} \cdot \tfrac{n}{2} \cdots \tfrac{n}{2}\right) && \text{(} \tfrac{n}{2} \text{ times)} \\
&= \log\left(\tfrac{n}{2}^{\frac{n}{2}}\right) \\
&= \tfrac{n}{2} log\left(\tfrac{n}{2}\right) && \text{(by log exponent rule)}
\end{aligned}$$

Thus, $\log(n!) \geq \frac{n}{2}\log\left(\frac{n}{2}\right)$, so we conclude that $\log(n!) = \Omega(n \log n)$.

PROOF RECAP

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

Theorem:

Any determinis ... sed sorting algorithm ... time.

This bound also ap ... sed sorting algorithms! ... em.

This mea ... ptimal!

(This is one ... ut proving lower bou ... we can d...

THE QUESTION IS...

CAN WE DO BETTER?

LINEAR-TIME SORTING

Beyond comparison-based sorting algorithms!

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

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

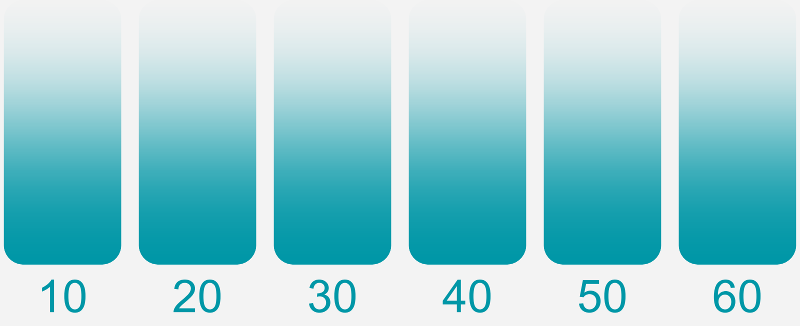
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

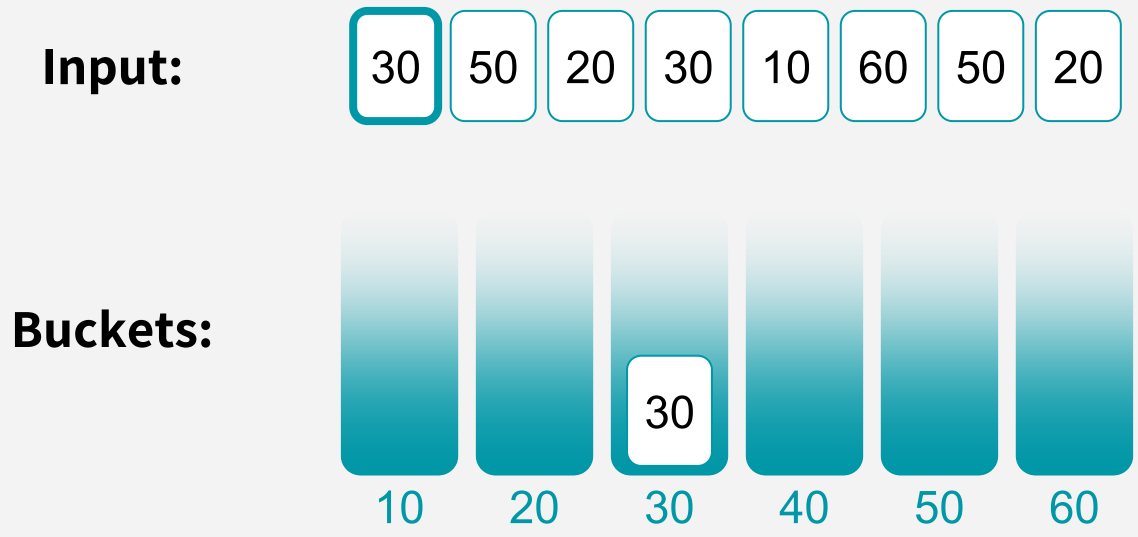
Buckets:



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}

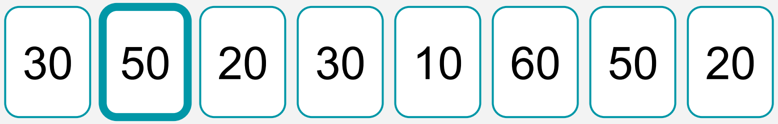


COUNTING SORT

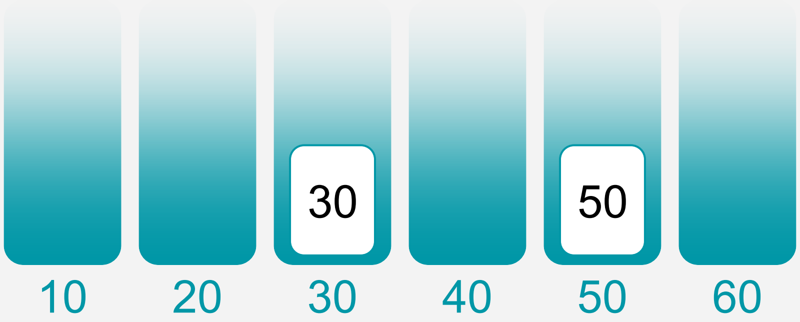
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:



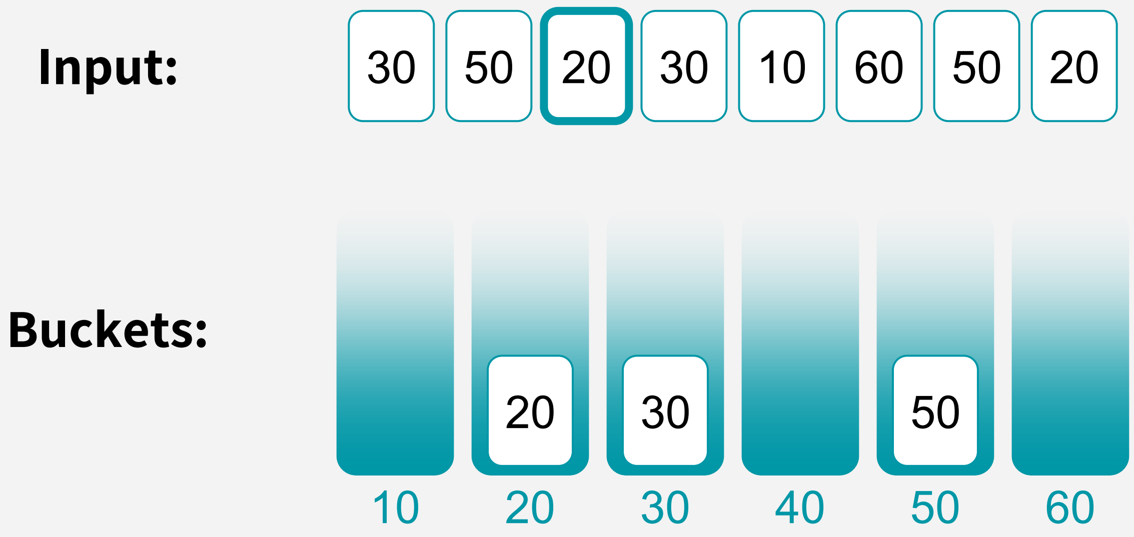
Buckets:



COUNTING SORT

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

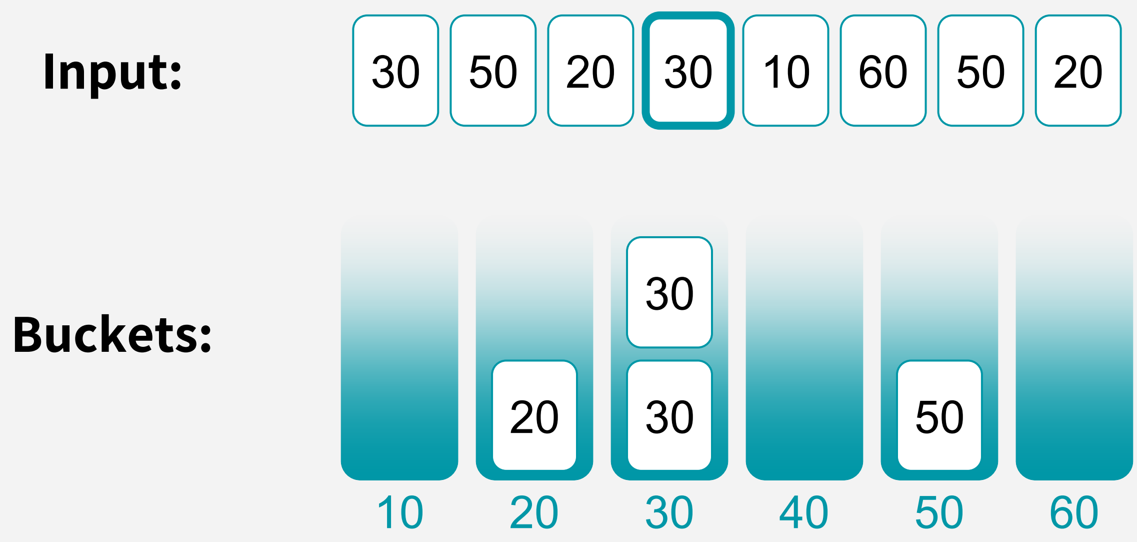
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

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

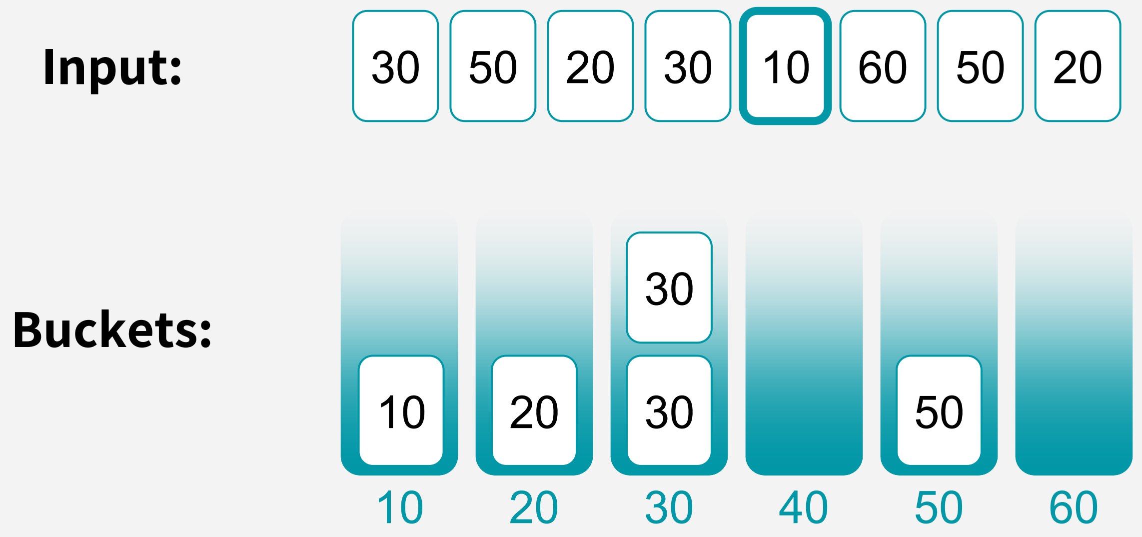
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

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

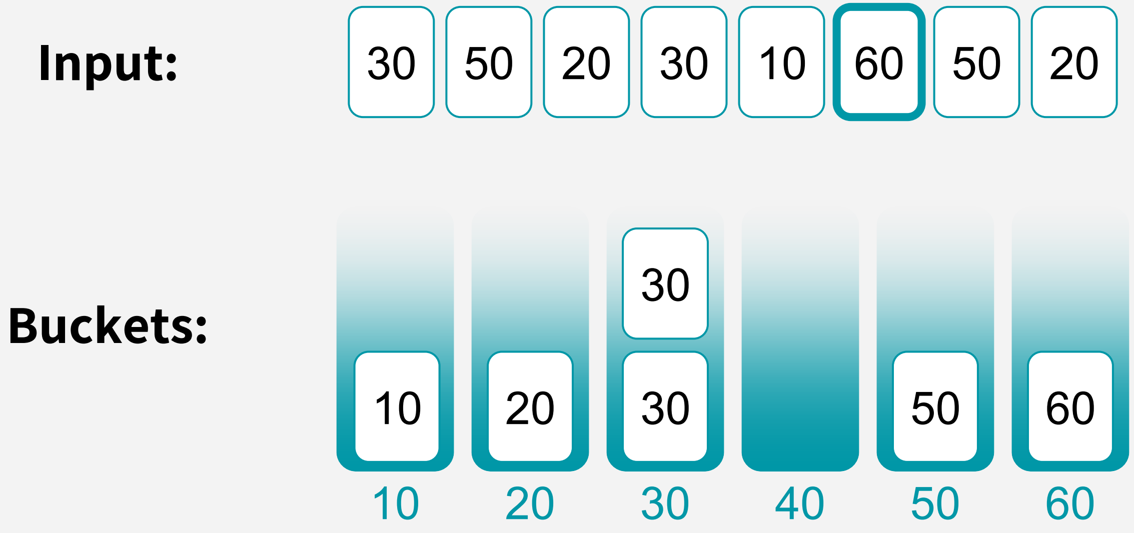
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

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

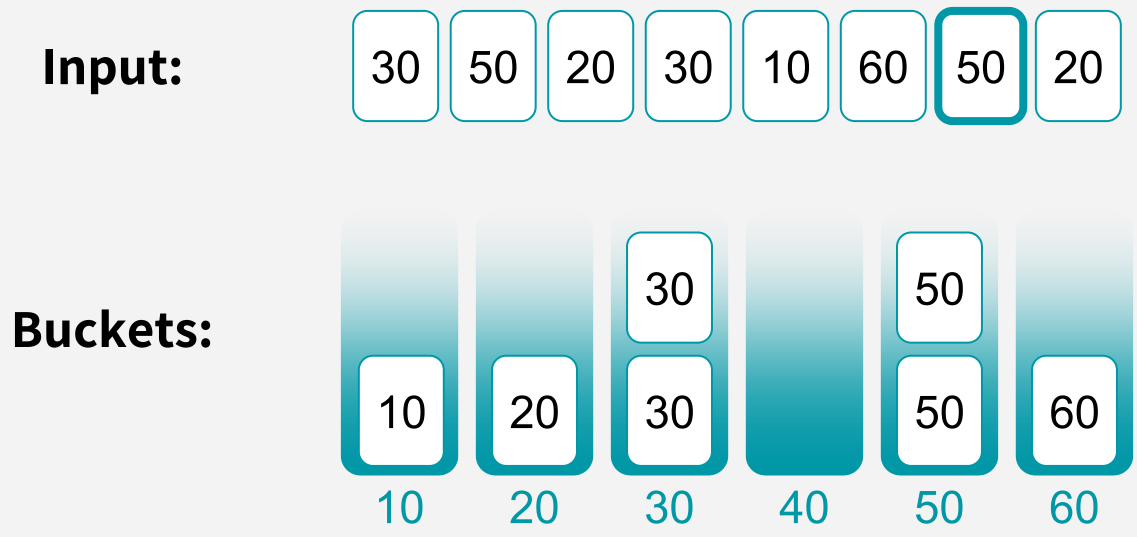
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

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

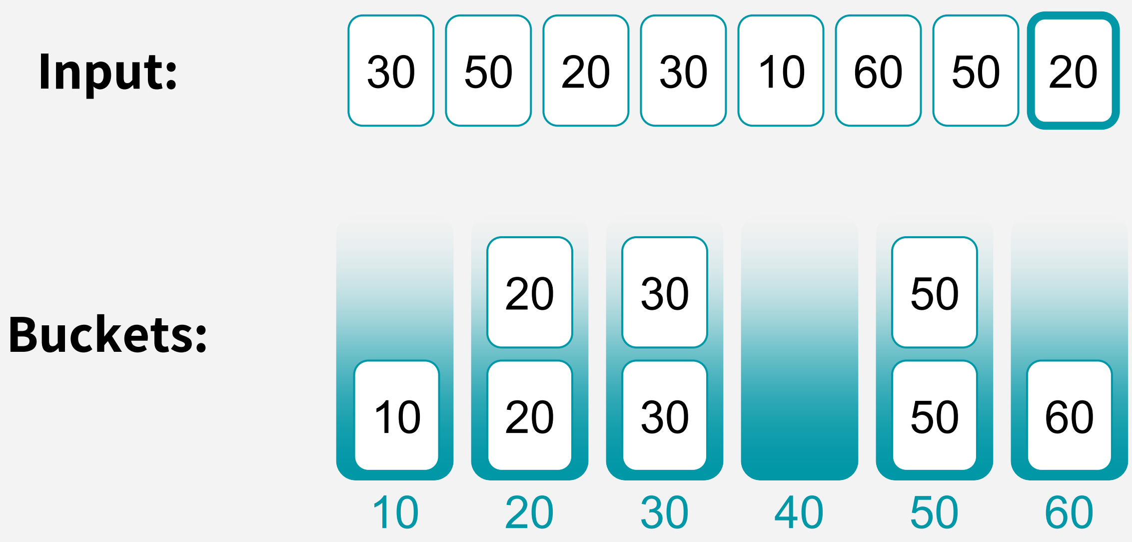
For example: elements are integers in {10, 20, 30, 40, 50, 60}



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}

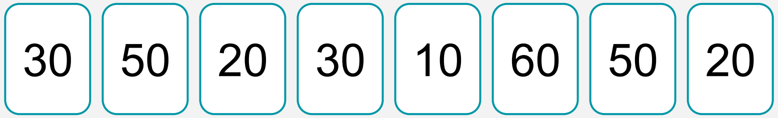


COUNTING SORT

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

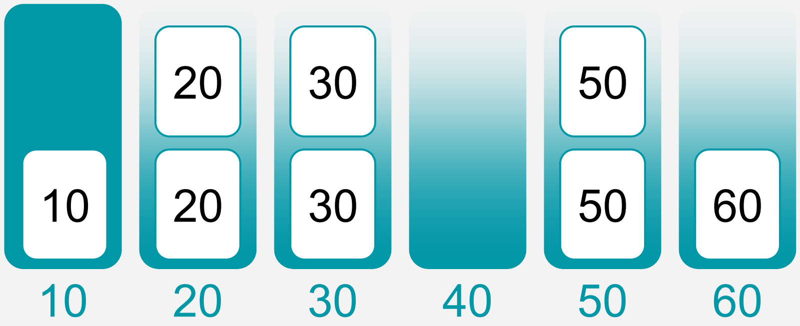
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:



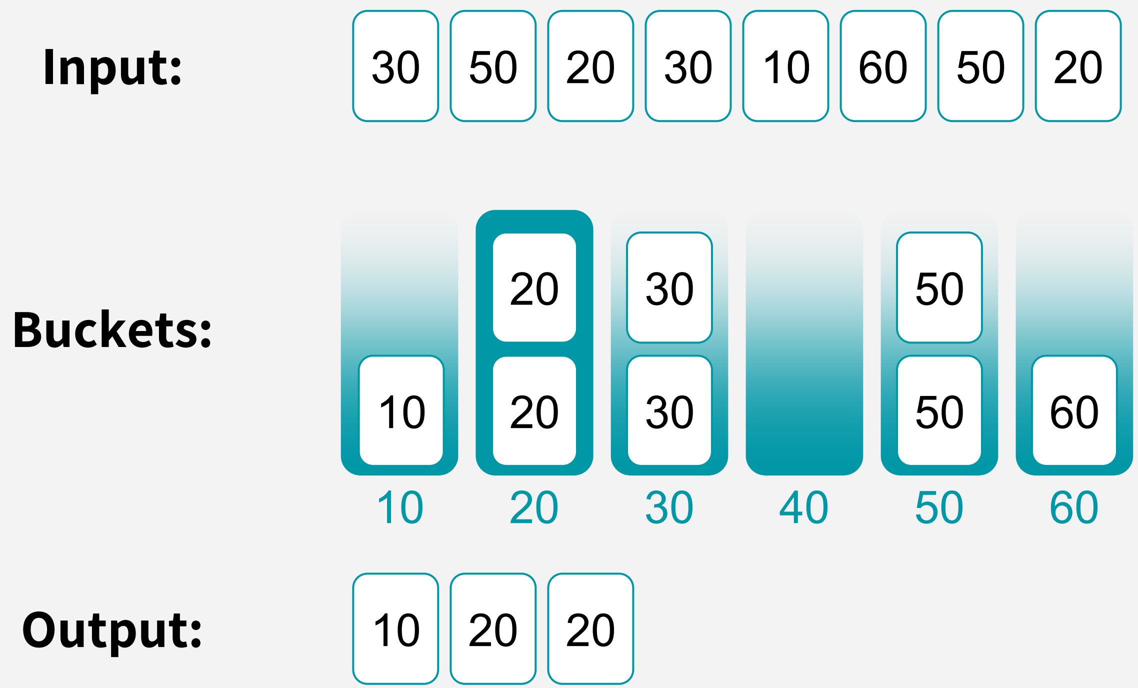
Output:

10

COUNTING SORT

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

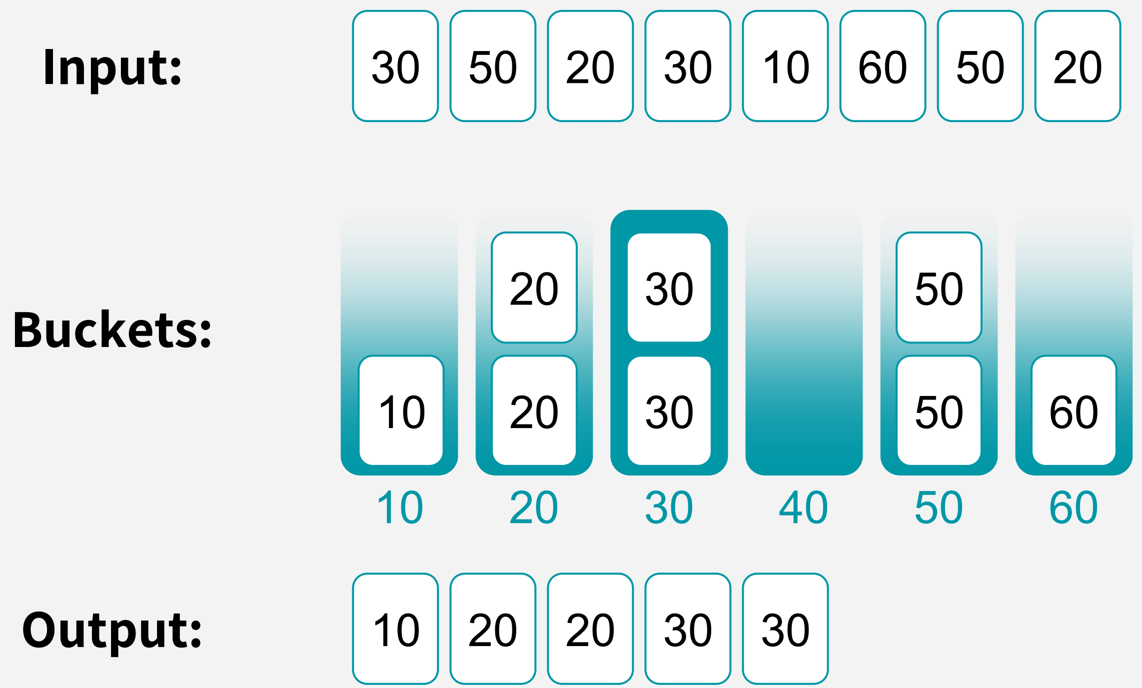
For example: elements are integers in {10, 20, 30, 40, 50, 60}



COUNTING SORT

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

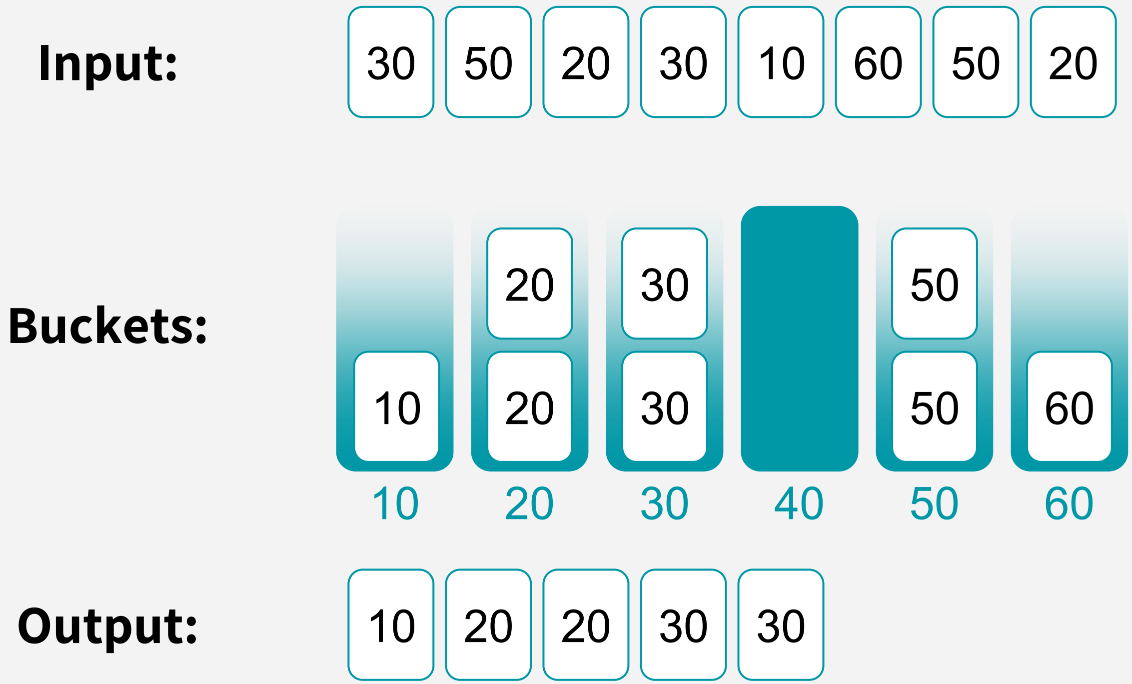
For example: elements are integers in {10, 20, 30, 40, 50, 60}



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

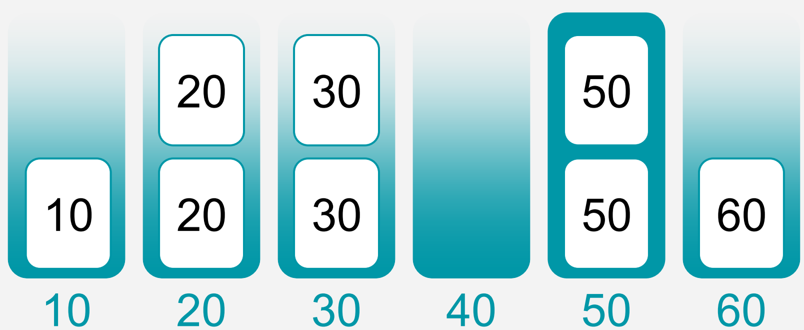
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:



Output:

10	20	20	30	30	50	50

COUNTING SORT

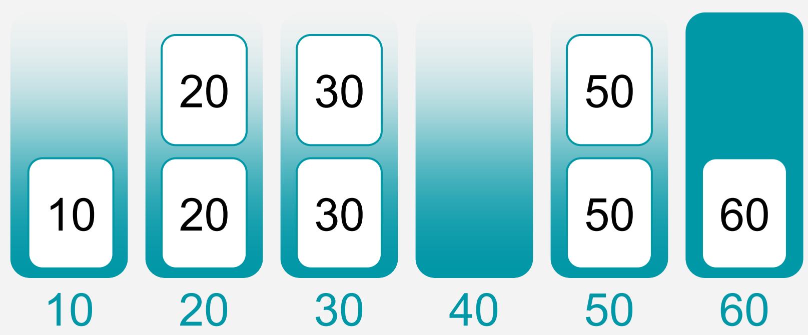
We assume that there are only k different possible values in the array (and we know these k values in advance)

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

Input:

30	50	20	30	10	60	50	20

Buckets:



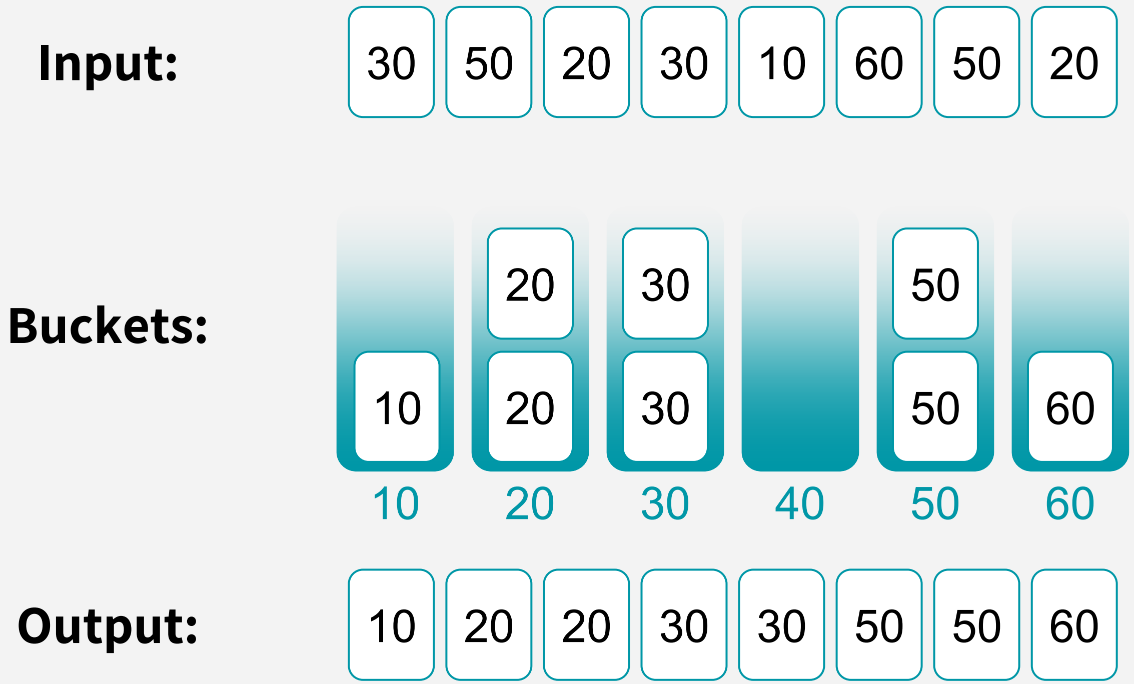
Output:

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}



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.

RADIX SORT

A sorting algorithm for integers up to size M

(or more generally, for sorting strings)

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!**

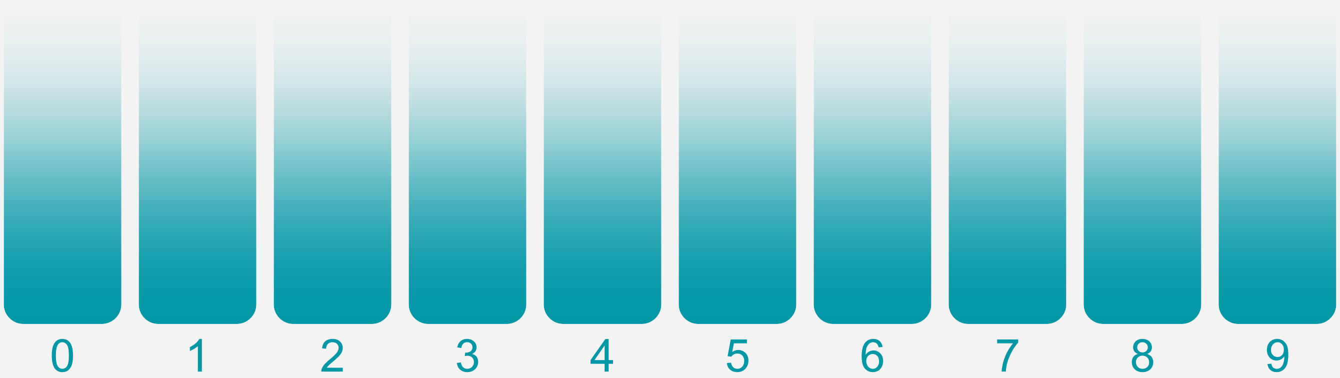
e.g. 10 buckets labeled 0, 1, …, 9

RADIX SORT

STEP 1: CountingSort on the least significant digit

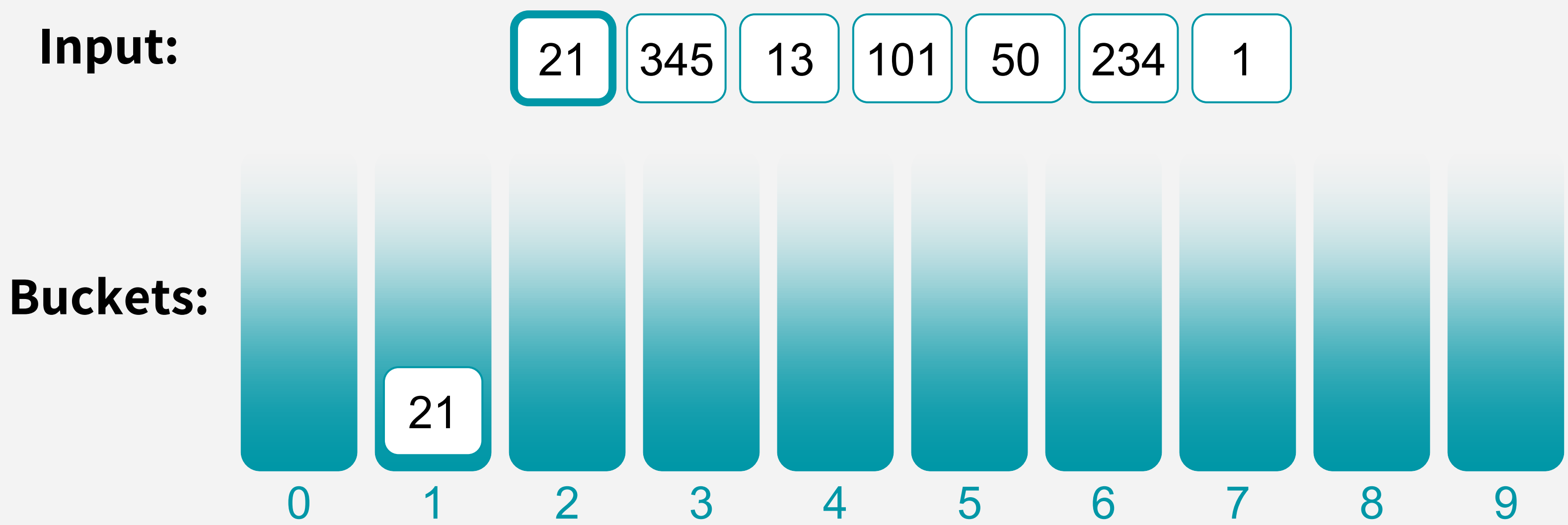
Input: 21 | 345 | 13 | 101 | 50 | 234 | 1

Buckets:



RADIX SORT

STEP 1: CountingSort on the least significant digit

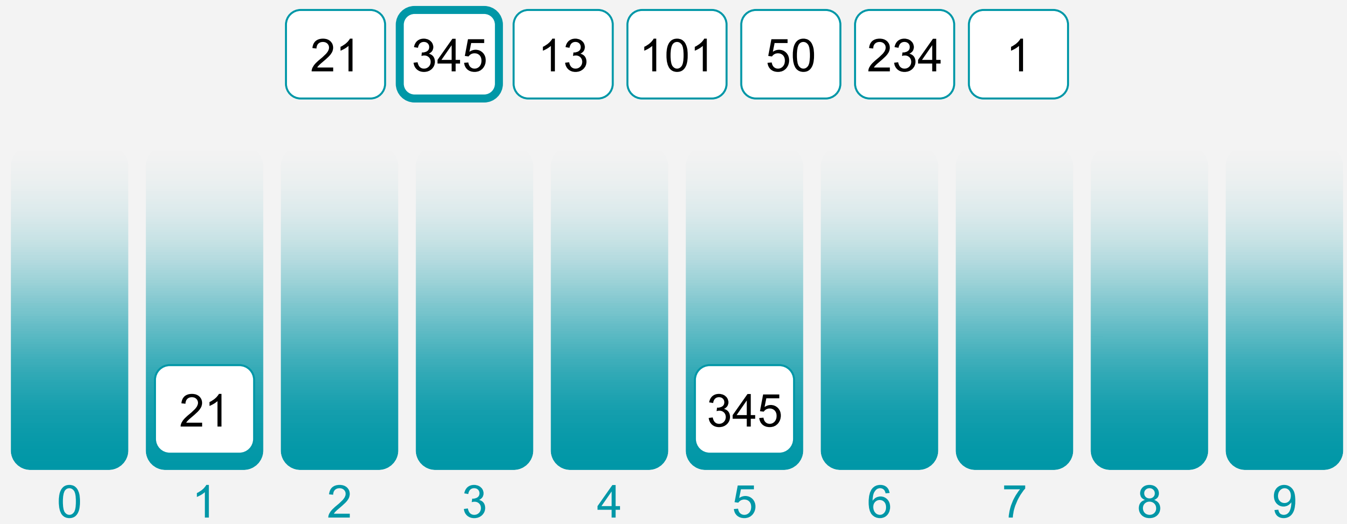


RADIX SORT

STEP 1: CountingSort on the least significant digit

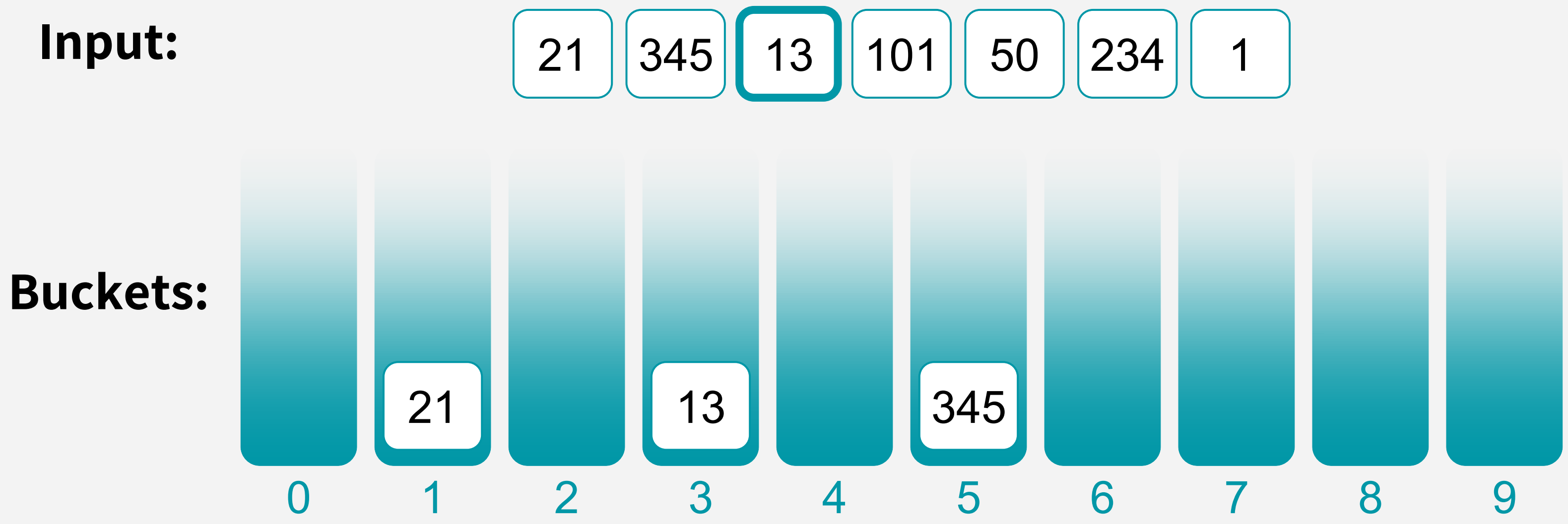
Input:

Buckets:



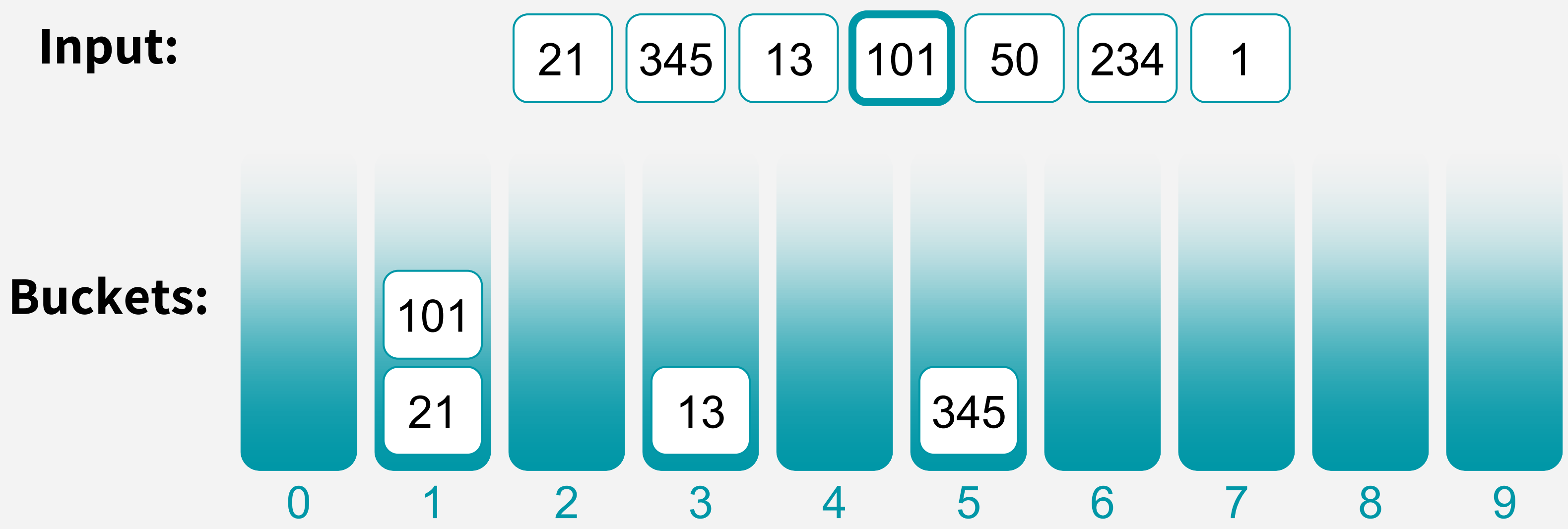
RADIX SORT

STEP 1: CountingSort on the least significant digit



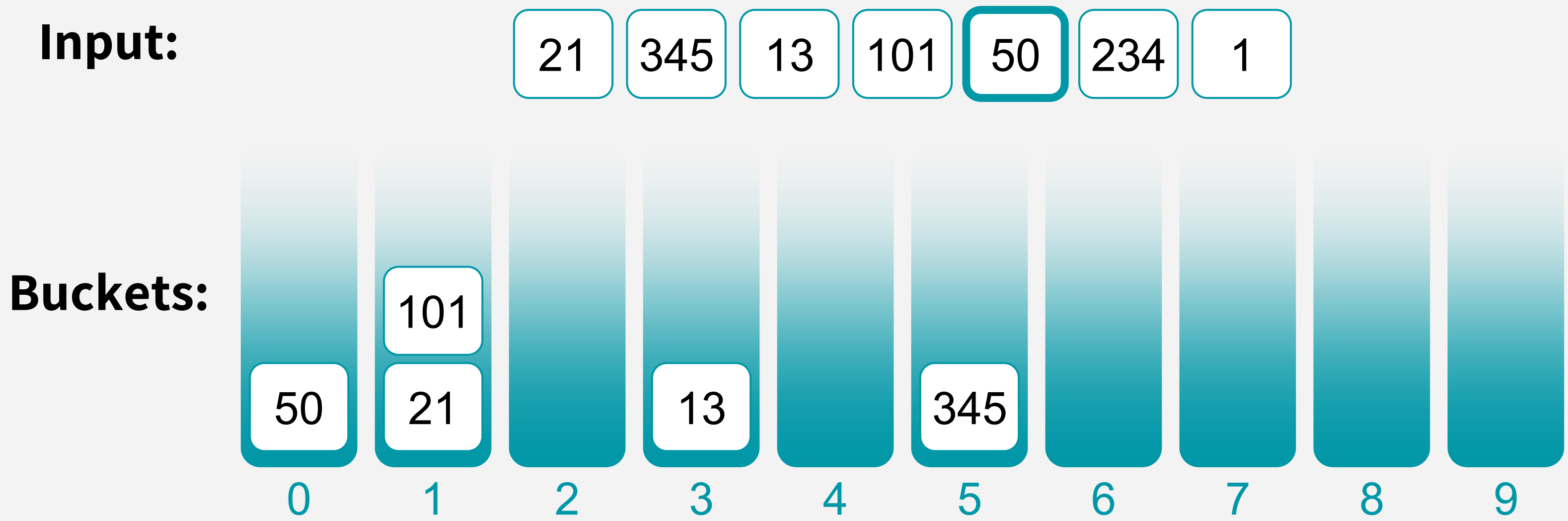
RADIX SORT

STEP 1: CountingSort on the least significant digit



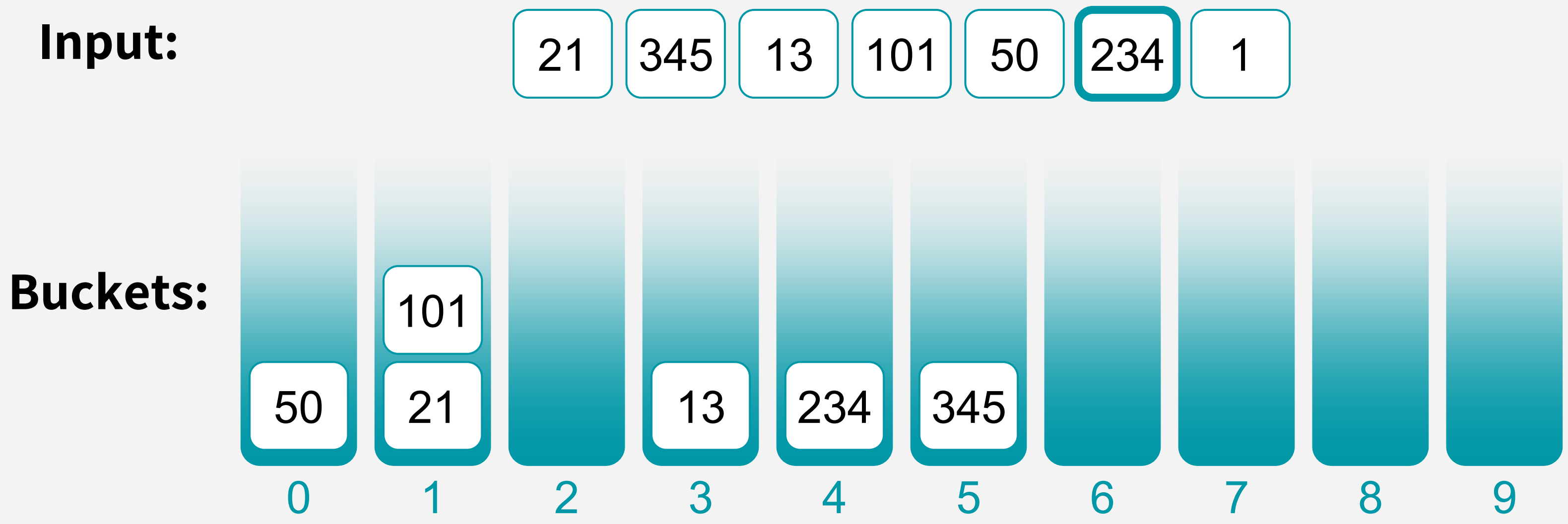
RADIX SORT

STEP 1: CountingSort on the least significant digit



RADIX SORT

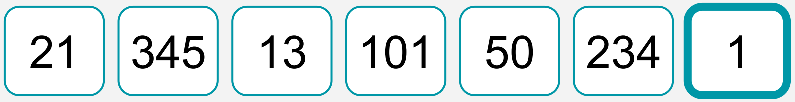
STEP 1: CountingSort on the least significant digit



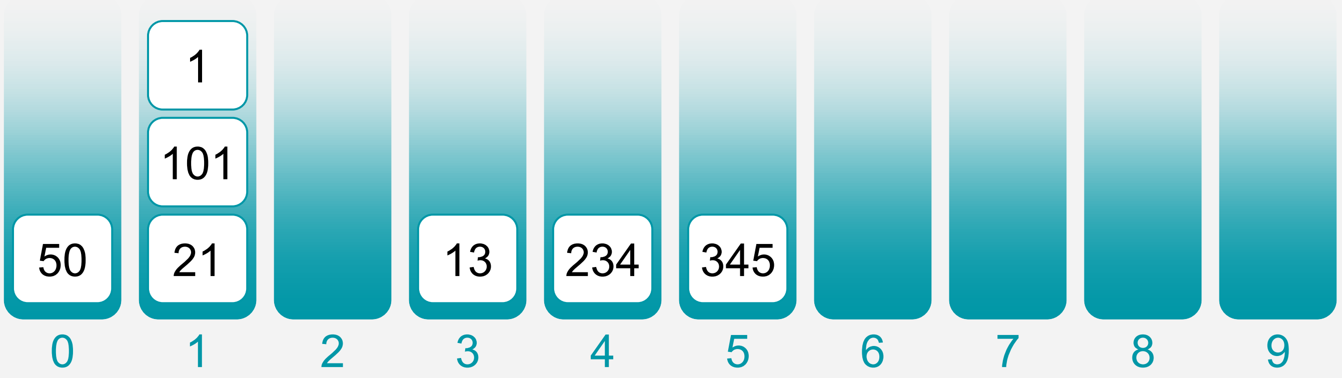
RADIX SORT

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:

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

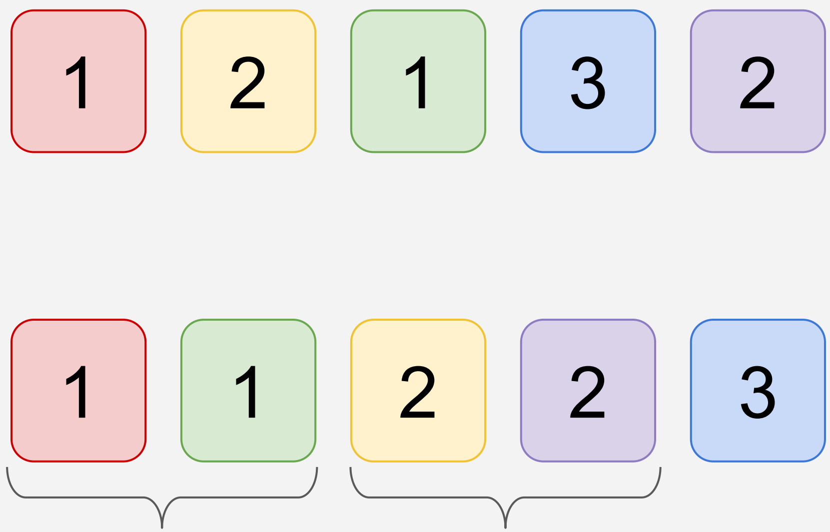
When creating the output list, make sure bucket items exit in FIFO order (i.e. use a *stable* implementation of CountingSort, where buckets are FIFO queues)

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:

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!

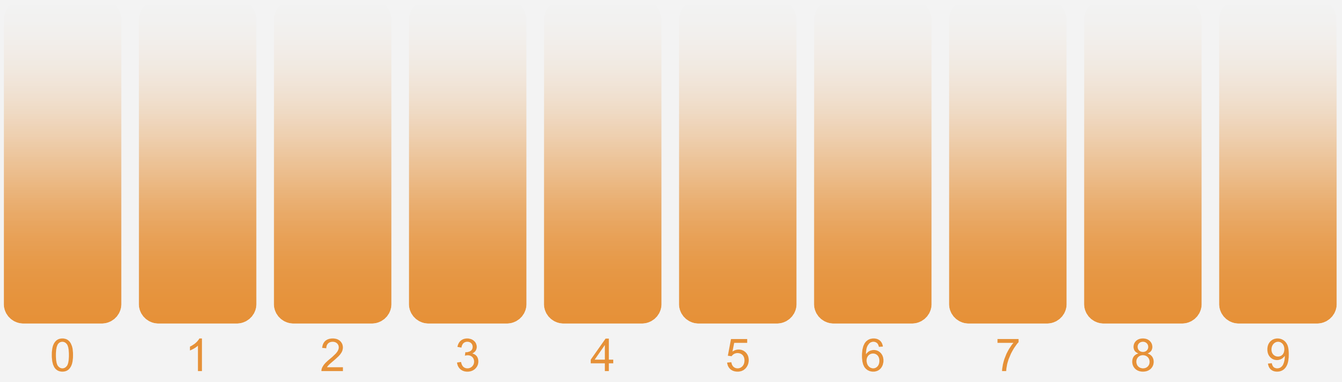
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	1	13	234	345

Buckets:

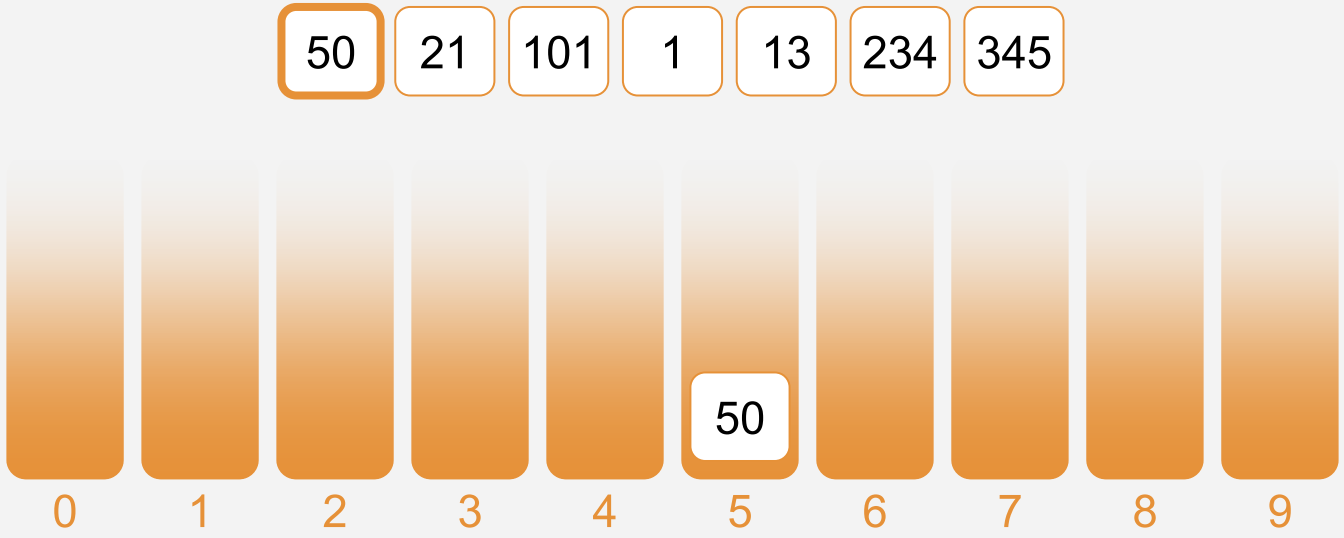


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

Buckets:



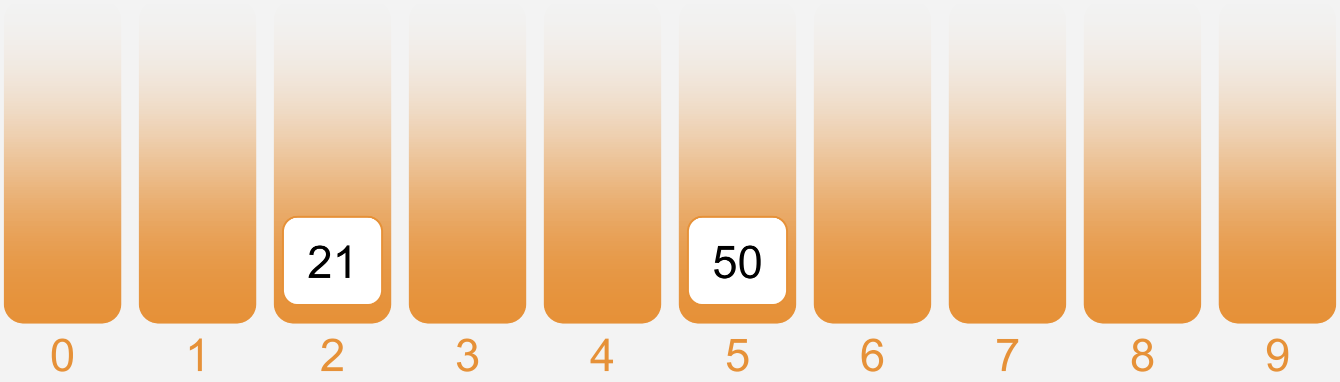
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	1	13	234	345

Buckets:

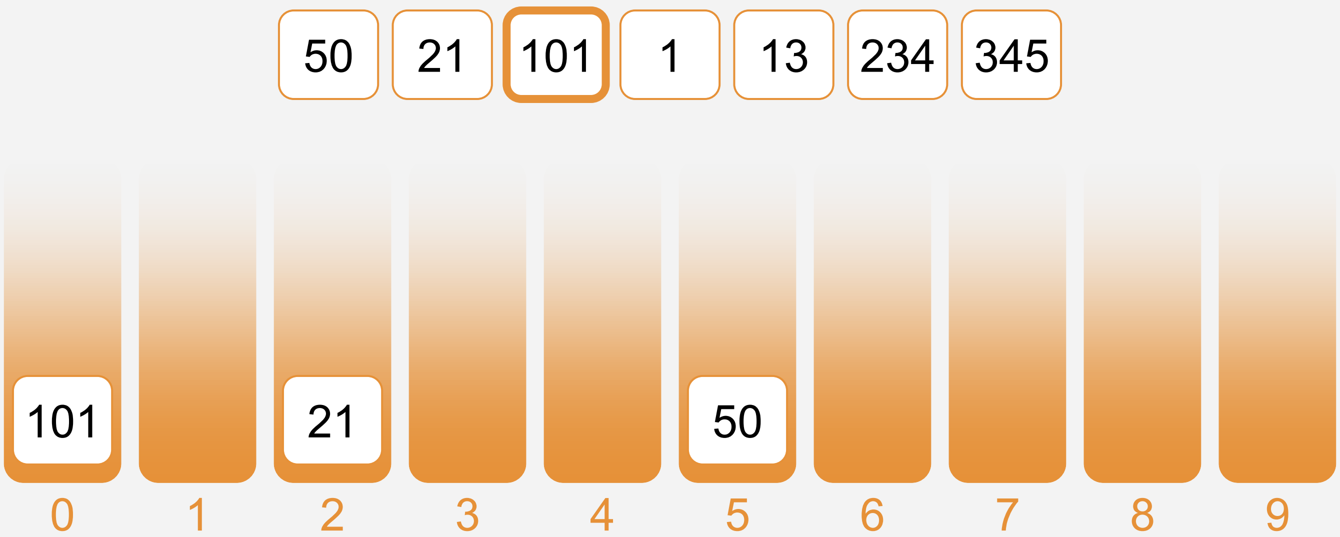


RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

Buckets:



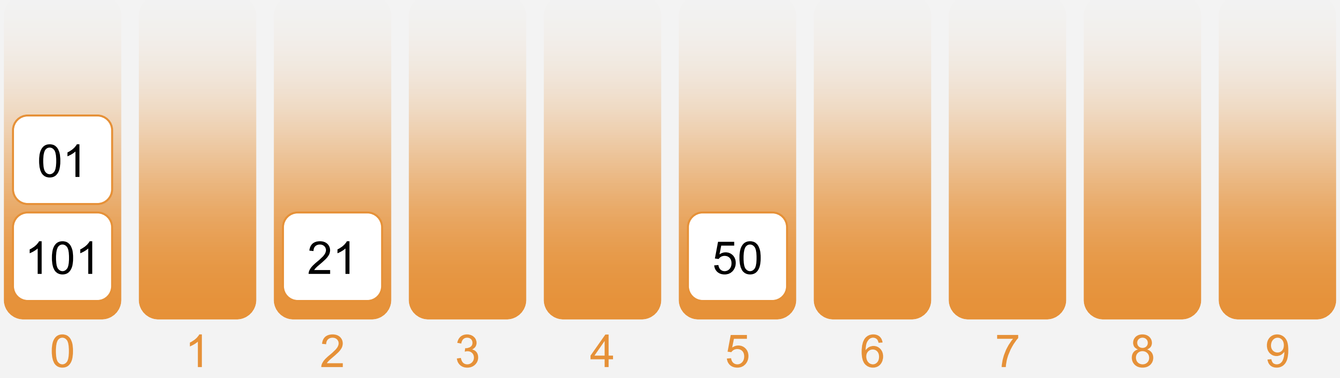
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50	21	101	01	13	234	345

Buckets:



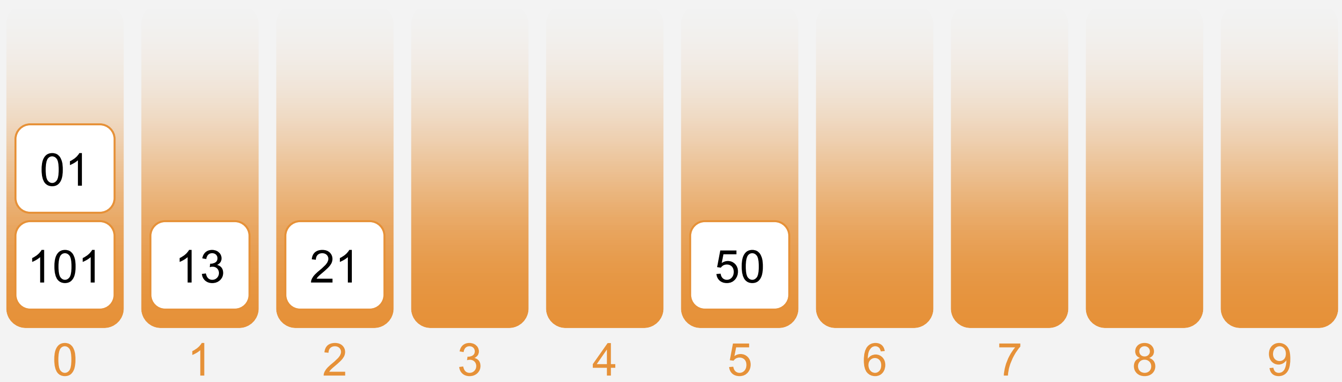
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

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

Buckets:



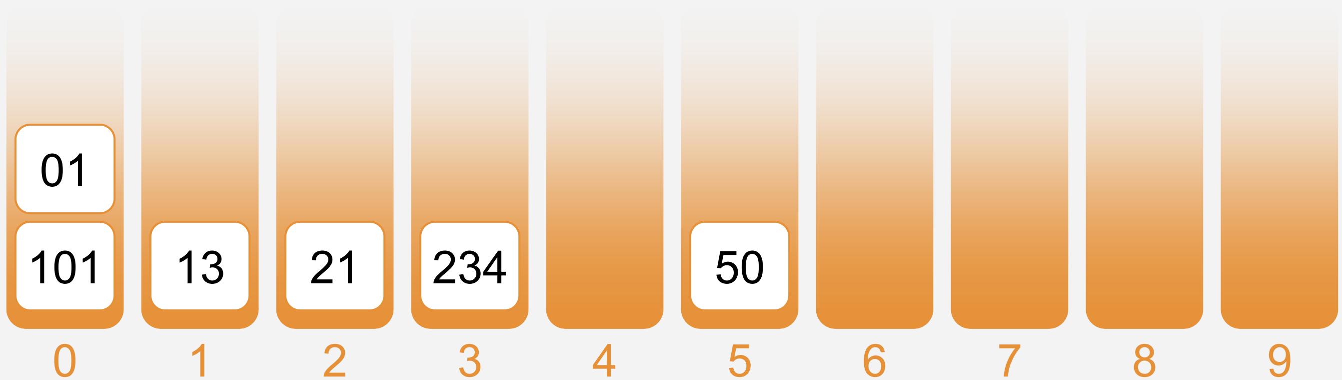
RADIX SORT

STEP 2: CountingSort on the 2nd least significant digit

Input:
(output from STEP 1)

50 21 101 01 13 **234** 345

Buckets:



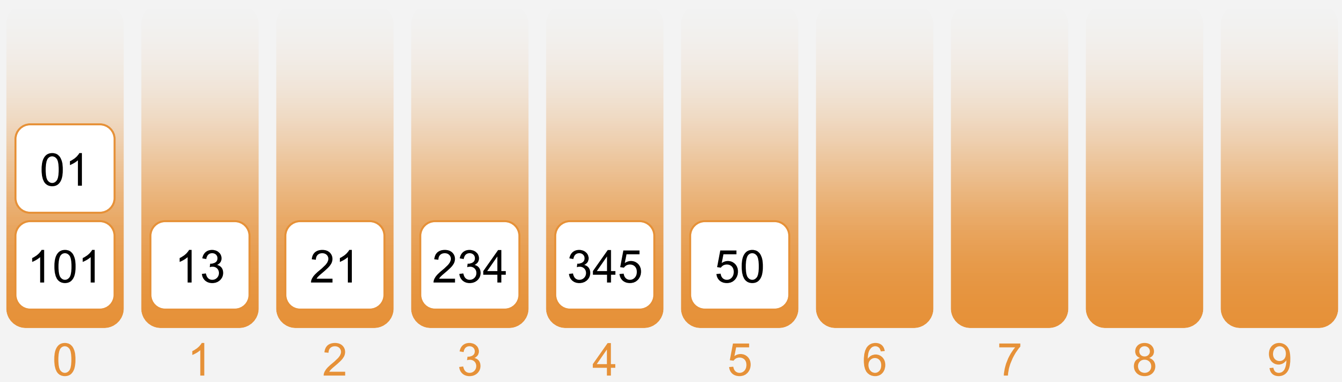
RADIX SORT

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)

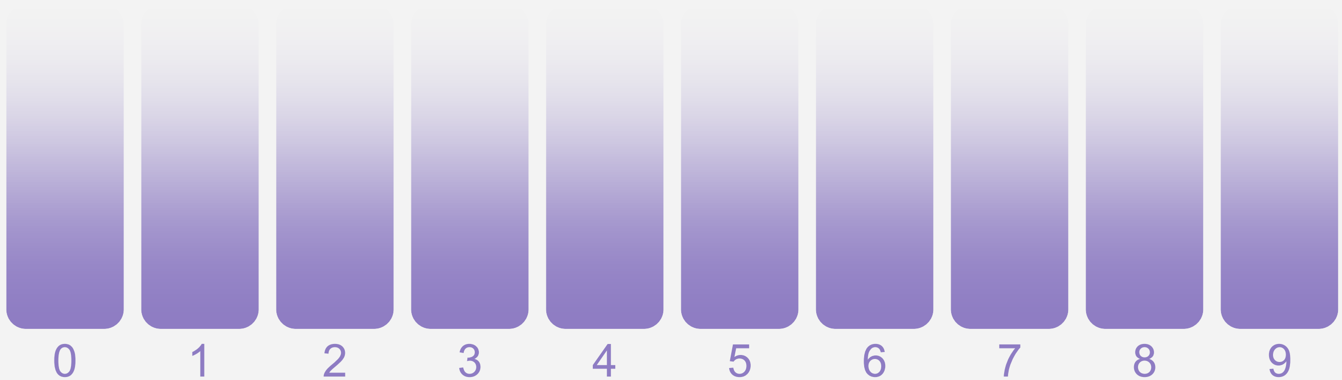
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	01	13	21	234	345	50

Buckets:



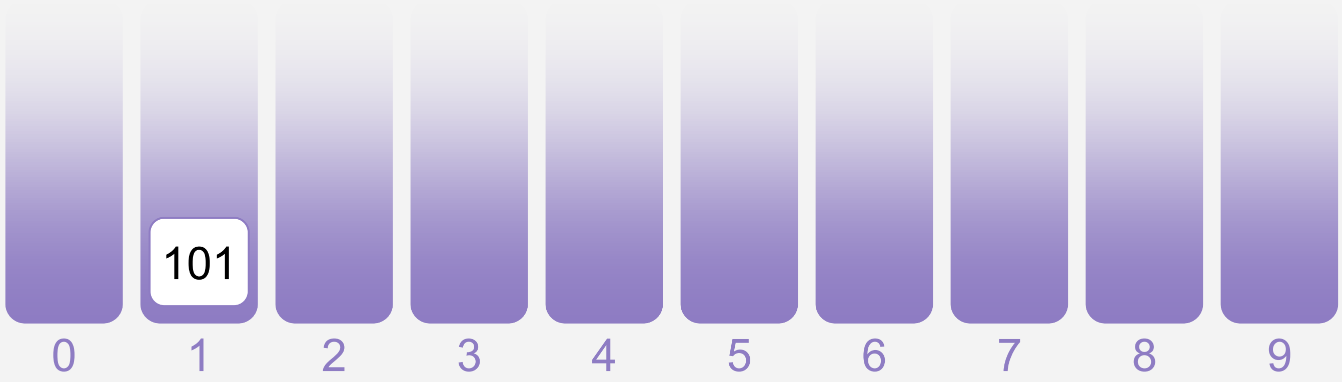
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	01	13	21	234	345	50

Buckets:



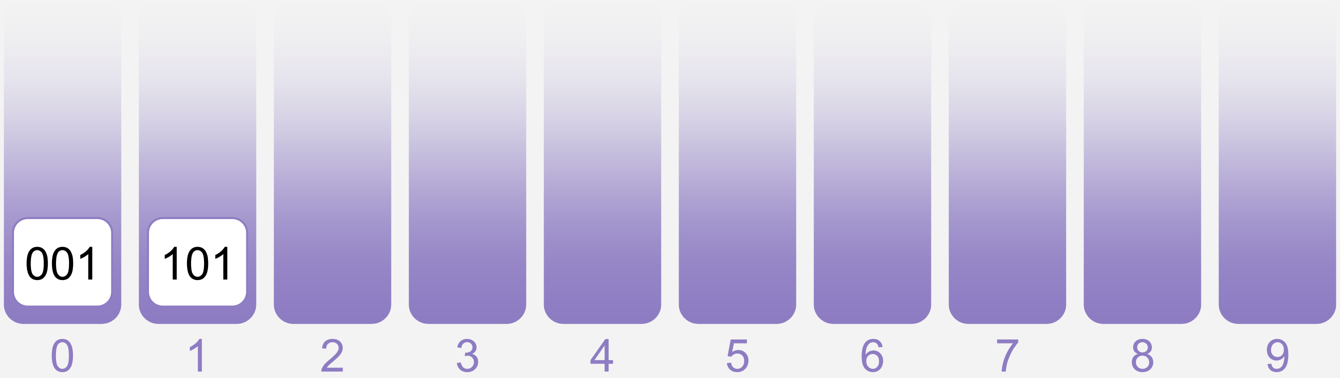
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

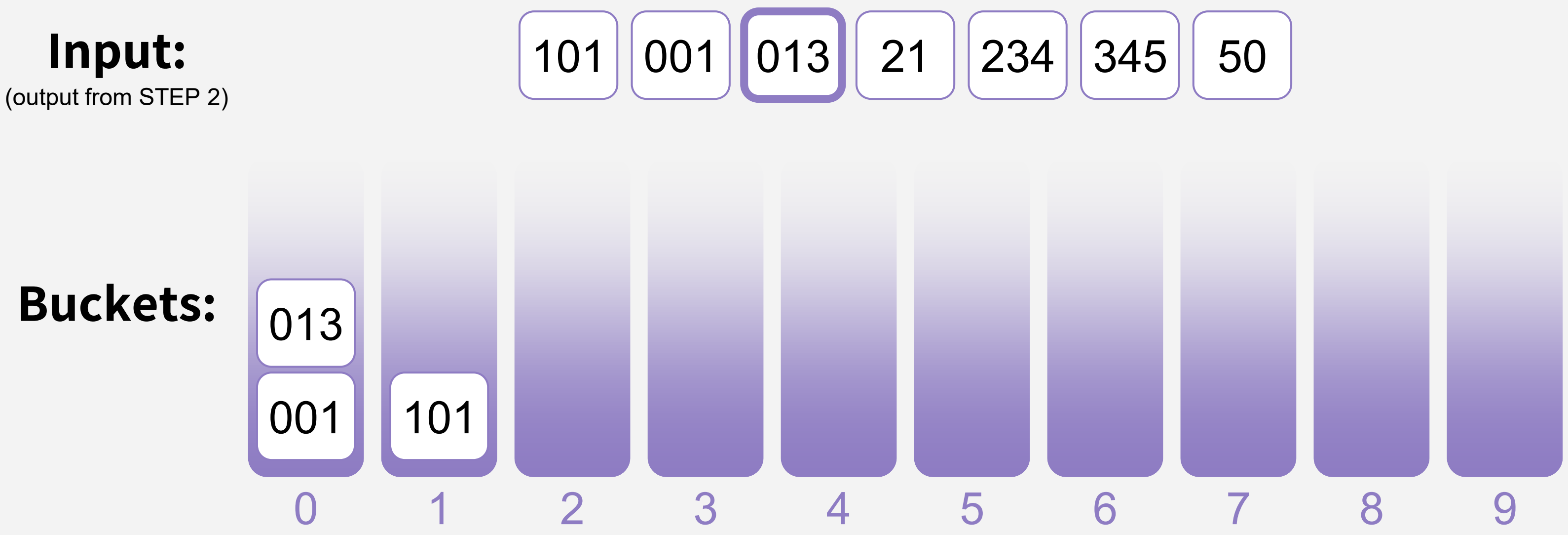
101	001	13	21	234	345	50

Buckets:



RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit



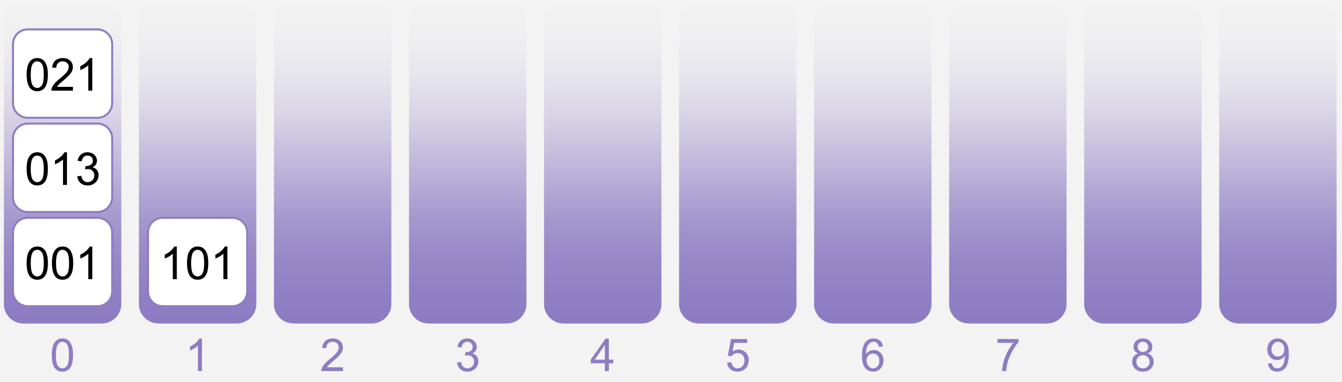
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	234	345	50

Buckets:



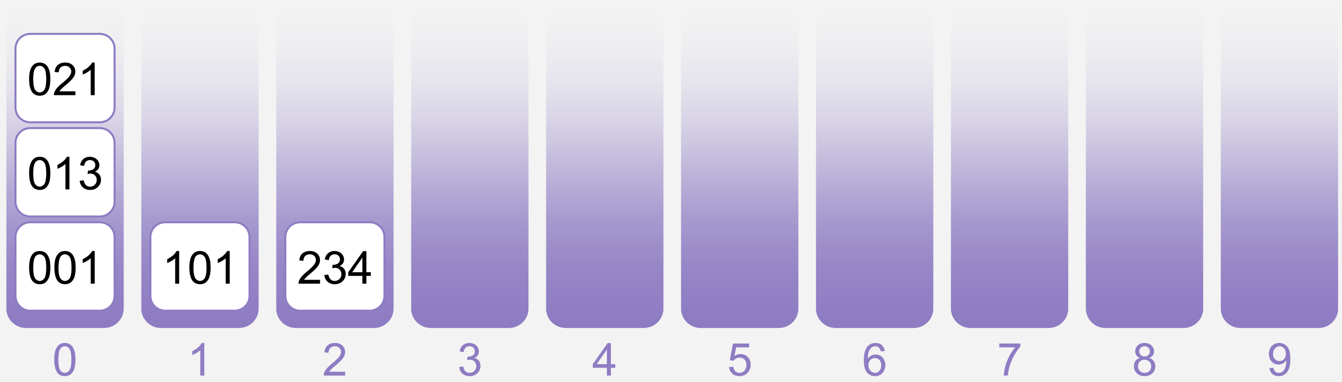
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101 | 001 | 013 | 021 | **234** | 345 | 50

Buckets:



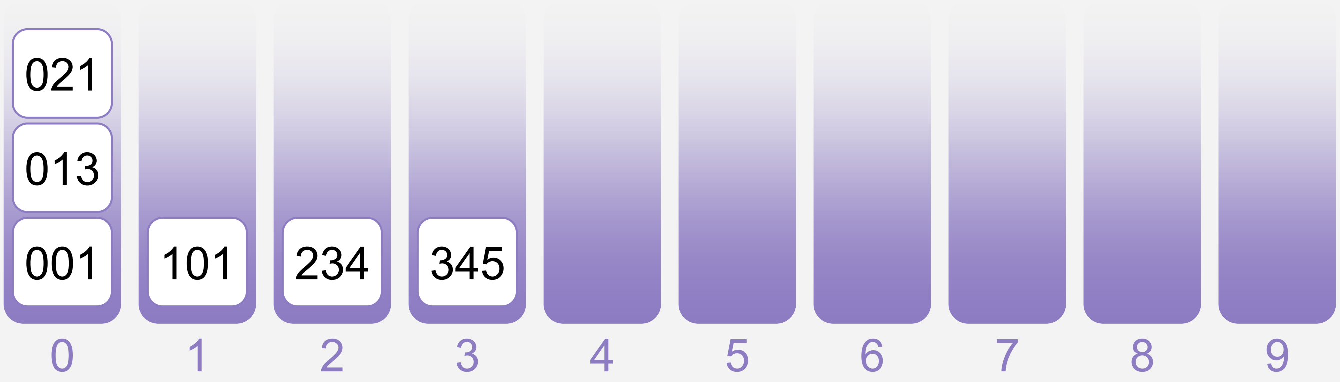
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input:
(output from STEP 2)

101	001	013	021	234	345	50

Buckets:



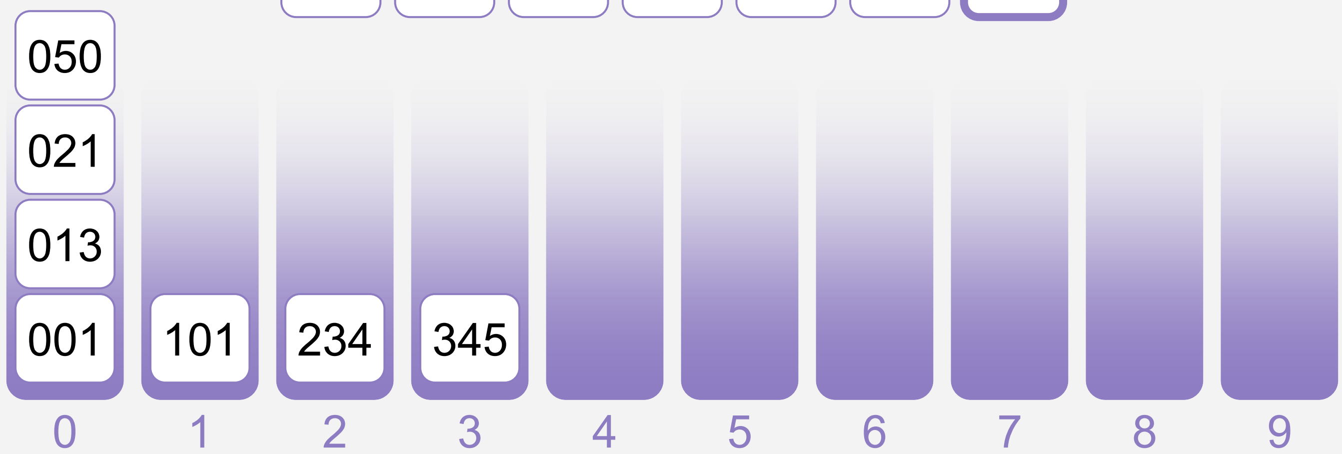
RADIX SORT

STEP 3: CountingSort on the 3rd least significant digit

Input: 101 001 013 021 234 345 050

(output from STEP 2)

Buckets:



Output: 001 013 021 050 101 234 345

It worked! But why does it work???

RADIX SORT CORRECTNESS

Why is Radix Sort correct?

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

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 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.

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.

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!

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!

RADIX SORT RUNTIME

What is the runtime of Radix Sort?

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

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?

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

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?

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)**

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

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?

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

How long does each iteration take?

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?

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

How long does each iteration take?

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

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

How long does each iteration take?

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

What is the total running time?

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 = $\lfloor \log_{10} M \rfloor + 1$ 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)

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 = $\lfloor \log_{10} M \rfloor + 1$ 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!

HOW GOOD IS O(nd)?

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

For example,
if M = 1234:
$\lfloor \log_{10} 1234 \rfloor + 1$
$= 3 + 1 = 4$

How m here?
d = ns

How lo take?
Initialize 10 ers in 10

What is time?
O(nd M)

THE QUESTION IS...

CAN WE DO BETTER?

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!

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?

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

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?

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 ⇒ $O(n + r)$

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 $\Rightarrow$ $O(n + r)$

What is the total running time?

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

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?

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

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?

How long does each iteration take?

What is the total running time?

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?

What is the total running time?

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?

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 $\Rightarrow$ $O(n+n) = O(n)$

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?

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

How long does each iteration take?

Initialize **n** buckets + put n numbers in **n** buckets $\Rightarrow$ $\mathbf{O(n+n) = O(n)}$

What is the total running time?

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

If $M \leq n^c$ for some constant c, then

This term is a constant!

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

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 ≤ n^c.

The number of buckets neeed is r = n.

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

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.

RECAP

- Any deterministic comparison-based sorting algorithm must take **Ω(n log n)** time.
- Linear Time sorting is possible with a different model of computation!
    - **Counting Sort**: super simple but doesn't work well with if your numbers take on too many values (too many buckets)
    - **Radix Sort**: performs Counting Sort digit-by-digit, and its runtime is linear if the maximum value of any element isn't too too large!

NEXT TIME

- Trees (Binary Search Trees & Balanced Trees)!

Acknowledgement

- Stanford University