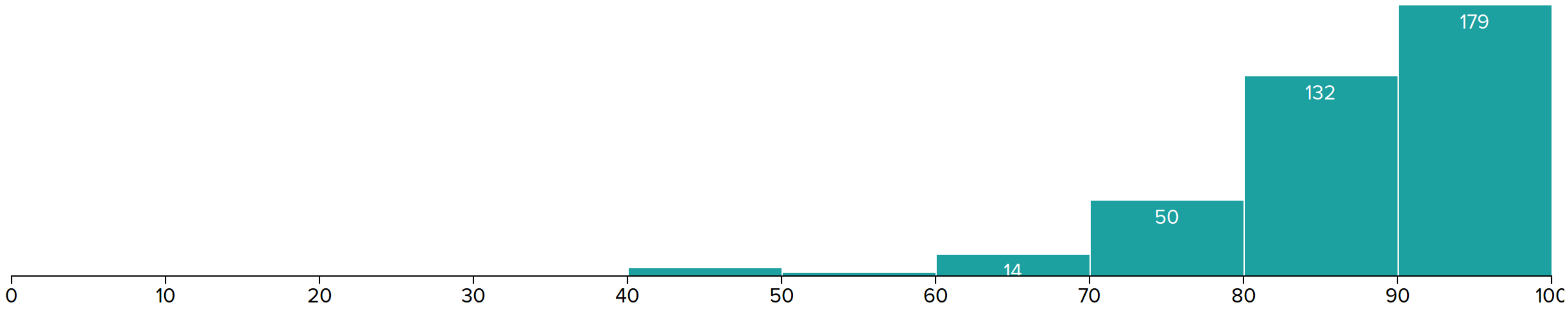
Lecture 6:
How Fast Can We Sort?

https://www.bloomberg.com/graphics/2017-fast-and-furious/

These slides include material originally prepared by Dr. Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

- Exam 1 graded



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
45.0	89.0	100.0	86.62	10.06

Announcements

- Exam 1 graded
    - Regrade requests due by 3/3
- Lab 6
    - Out Wednesday, due 3/8
    - Practice with recurrences, sorting, searching
    - Will not have a coding portion or pre-lab (wait for Lab 7)

What Do We Know About Sorting?

- We know a couple of worst-case Θ(n log n) algorithms

- HeapSort
  - Insert all inputs into a heap
  - Extract in sorted order
  - Lab 3 unit test did this

- MergeSort (Thursday's studio!)
  - Based on linear merge of two sorted arrays
  - Divide-and-conquer algorithm

What Other Sorting Algorithms Exist?

- BubbleSort – $\Theta(n^2)$
- InsertionSort – $\Theta(n^2)$
- ShellSort – $\Theta(n^2)$, or $\Theta(n^{4/3})$, or $\Theta(n \log^2 n)$, or … [many different variants]
- QuickSort – $\Theta(n \log n)$ [*if we work at it; see 347*]
- …
- See "The Sounds of Sorting" website for audio/visual intuition

How Fast *Can* We Sort?

- Multiple worst-case Θ(n log n) time algorithms
- All the others we listed are slower!
- Is there a faster sorting algorithm?

To answer, we need to be more precise about what "sorting algorithm" means…

What is a Sorting Algorithm Allowed to Do?

- Computers are not infinitely powerful…
- They can do only limited work in constant time.
- In particular, they can make limited decisions about their inputs in constant time.

What is a Sorting Algorithm Allowed to Do?

- Comput
- They time.
- In par ons about

“Model of Computation” – which operations can your computer do in constant time?

Limited Decisions for Sorting

- All the sorting algorithms we listed work on any `Comparable` data type.
- The only way they inspect the input is by comparing pairs of elements to each other!
- **Can answer “Is x > y?” in constant time.**

Limited Decisions for Sorting

- All the
  Com
- The g two
  elem
- **Can**

Any sorting algorithm that inspects its input only via pairwise comparisons is called a “*comparison sort*.”

An Aside on Comparisons

- If we can test “$x > y$”…
- We can also test “$x \leq y$” (NOT $x > y$)
- Hence, we can test “$x = y$” ($x \leq y$ AND $y \leq x$), “$x \geq y$” ($x = y$ OR $x > y$), and “$x < y$” (NOT $x \geq y$)

We can implement all ordered comparisons in $O(1)$ >’s.

Reformulating the Question

- **How many comparisons do we need to sort an input array of size n?**

- If each comparison takes constant time, and comparison is the dominant cost of sorting…

- …then # of comparisons gives time complexity of sorting.

What We Know

- We know of algorithms that use $\Theta(n \log n)$ comparisons to sort an array of size n.

- Hence, # of required comparisons for *fastest possible algorithm* is **???(n log n)**

What We Know

- We know of algorithms that use $\Theta(n \log n)$ comparisons to sort an array of size n.

- Hence, # of required comparisons for *fastest possible algorithm* is **$O(n \log n)$**

What We Know

- We know of algorithms that use $\Theta(n \log n)$ comparisons to sort an array of size n.

- Hence, # of required comparisons for *fastest possible algorithm* is **$O(n \log n)$**

- Any *fixed* sorting algorithm gives **upper bound** on cost of fastest possible algorithm.

What We Want

- Is there an f(n) for which every comparison sort requires **Ω(f(n))** comparisons to sort an array of size n?

- That is, can we find an asymptotic **lower bound** on cost of *any* comparison sort?

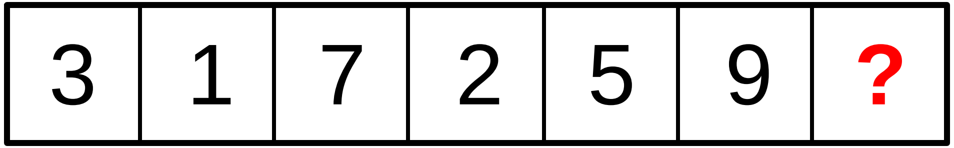
A Trivial Lower Bound

- Claim: every comparison sort takes time **Ω(n)**.

A Trivial Lower Bound

- Claim: every comparison sort takes time **Ω(n)**.

- Pf: a correct sorting algorithm must inspect every element of its input array *at least once*.



3	1	7	2	5	9	**?**

A Trivial Lower Bound

- Claim: every comparison sort takes time **Ω(n)**.
- Pf: a correct sorting algorithm must inspect every element of its input array *at least once*.

1	2	3	5	7	9	**?**

A Trivial Lower B

- Claim: every com
- Pf: a correct sorti
  of its input array a

We have no idea what this value is, so cannot determine correct place for it in order.

1	2	3	5	7	9	?

A Trivial Lower Bound

- Claim: every comparison sort takes time **Ω(n)**.

- Pf: a correct sorting algorithm must inspect every element of its input array *at least once*.

- Each comparison inspects only 2 elements, so we need at least **???** comparisons.

A Trivial Lower Bound

- Claim: every comparison sort takes time **Ω(n)**.

- Pf: a correct sorting algorithm must inspect every element of its input array *at least once*.

- Each comparison inspects only 2 elements, so we need at least **n/2** comparisons. **QED**

Can We Improve This Lower Bound?

- Yes, but it will take a bit more work.

- Need a way to represent **any** possible comparison sort

Can We Improve This Lower Bound?

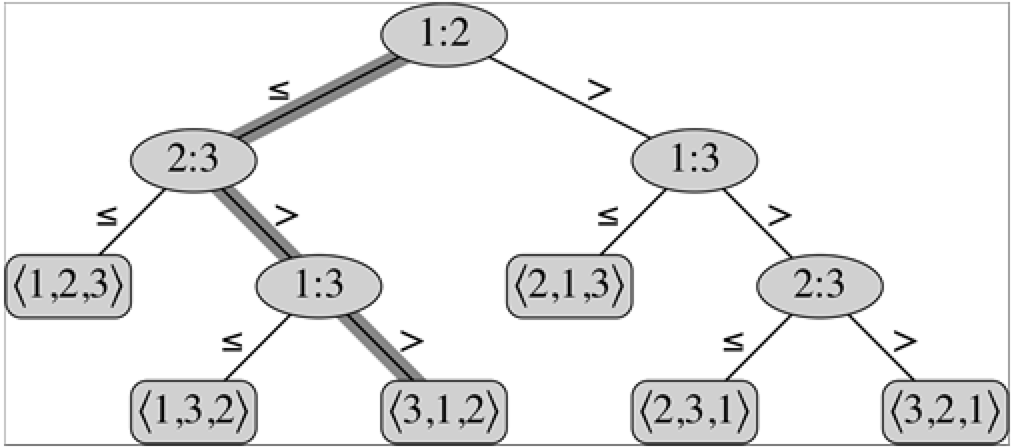
- Yes, but it will take a bit more work.
- Need a way to represent **any** possible comparison sort
- *(Even algorithms we have never imagined!)*
- **Will use properties of representation to prove bound.**

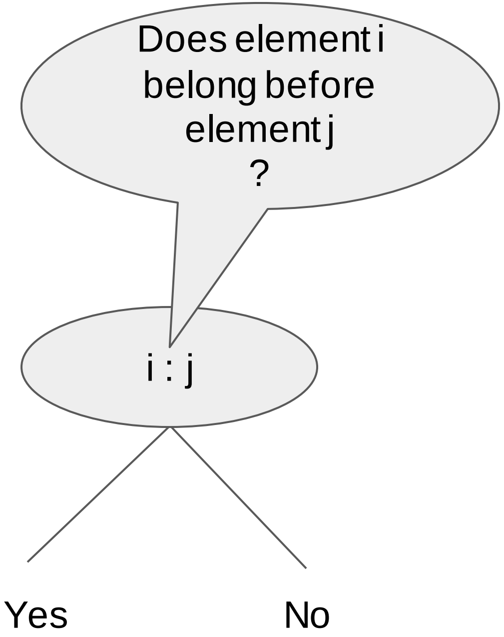
A New Way to Represent Algorithms

- Given an input array of size n….
- Any fixed sorting algorithm compares elements according to some logic.
- Choice of later comparisons might depend on results of earlier ones.
- Will use a tree to encode logic of comparison sequence.

Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)





Does element i belong before element j ?

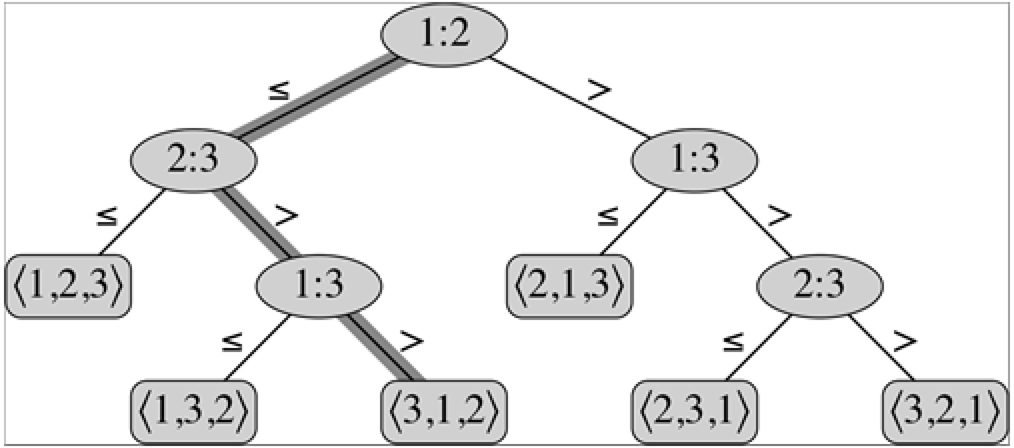
i : j

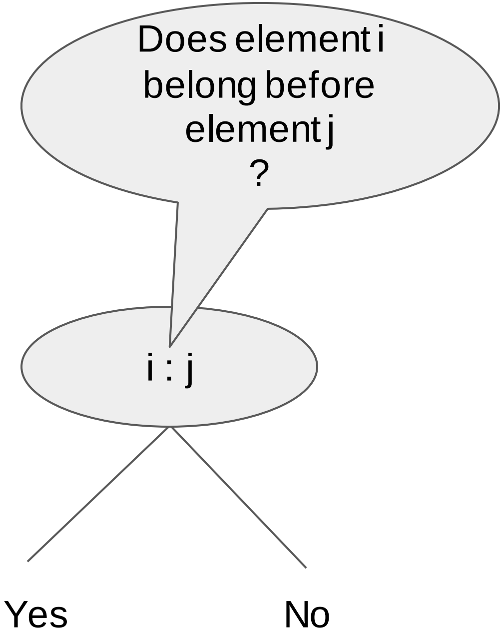
Yes

No

One Possible decision tree for sorting using comparisons

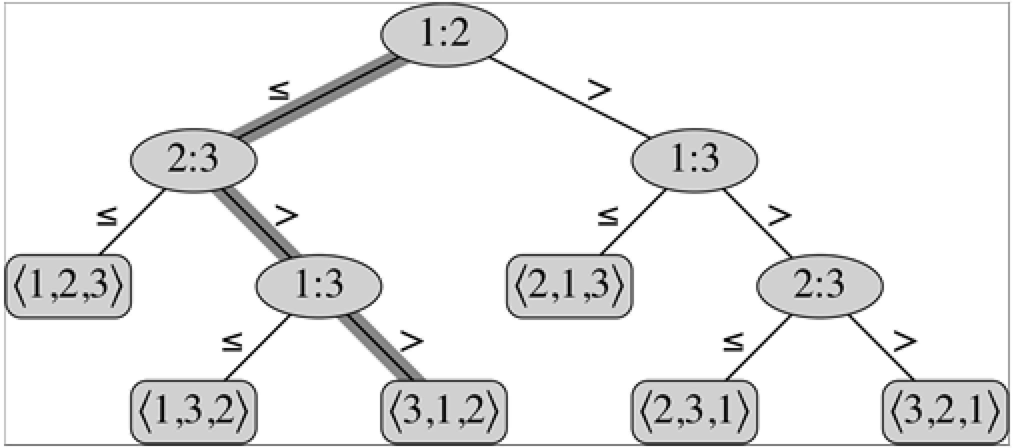
Sorting 3 elements (Figure 8.1 from text)



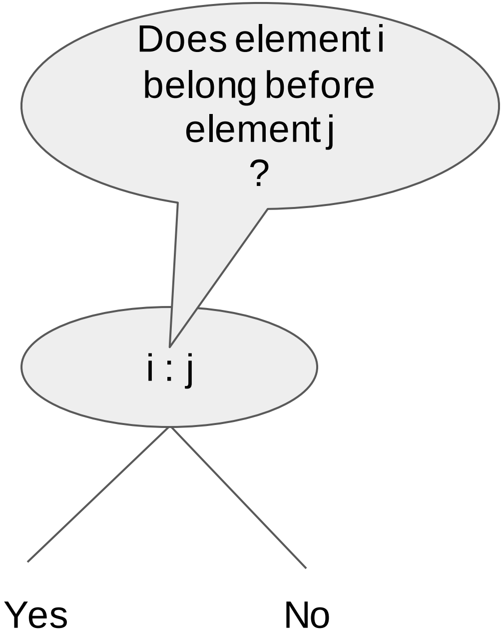


Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)

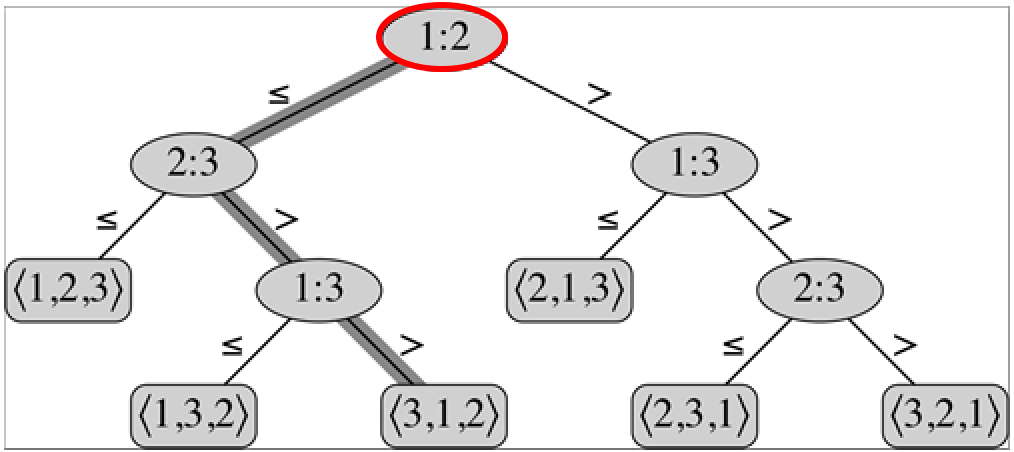


75 99 50

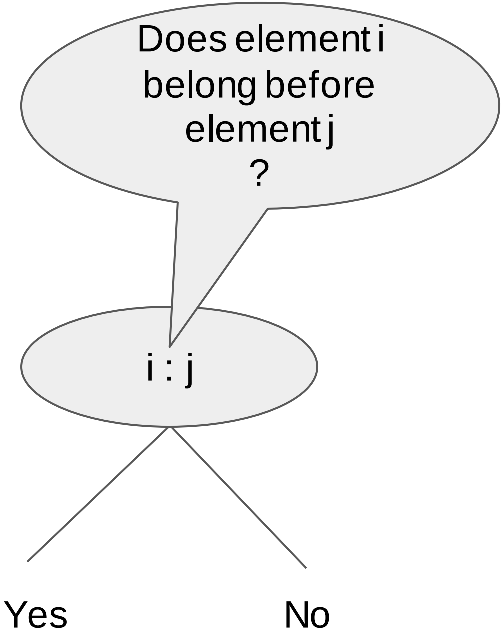

Does element i belong before element j ?

Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



75 99 50



Does element i belong before element j ?

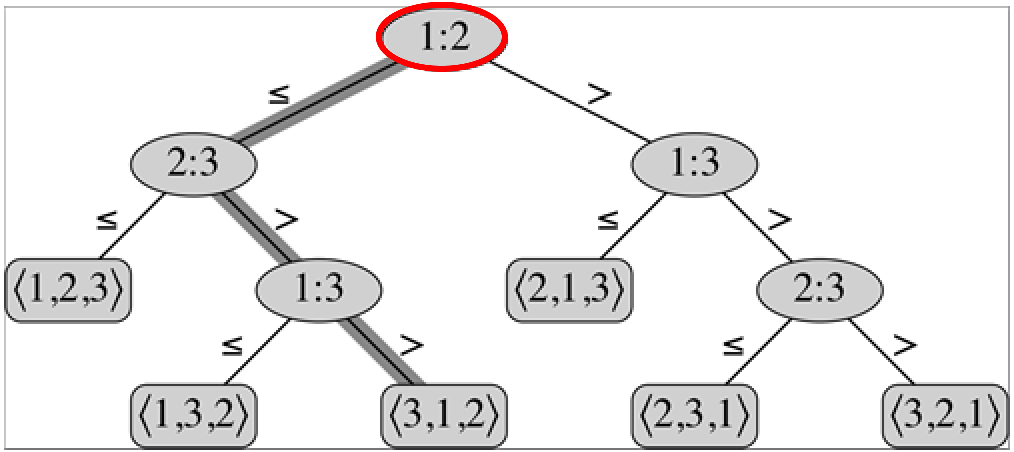
i : j

Yes

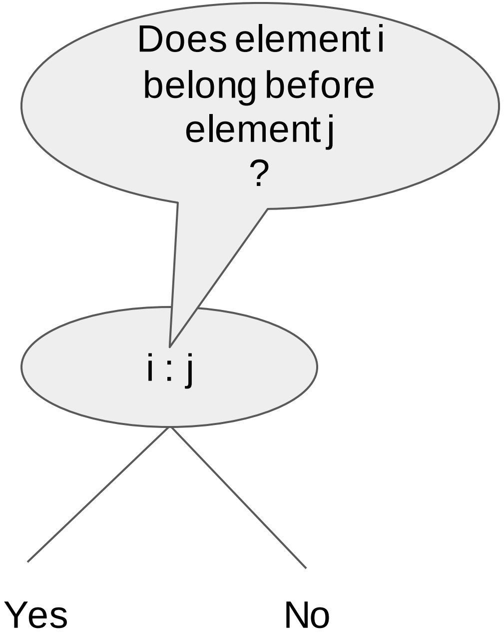
No

Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



75 99 50


Does element i belong before element j ?

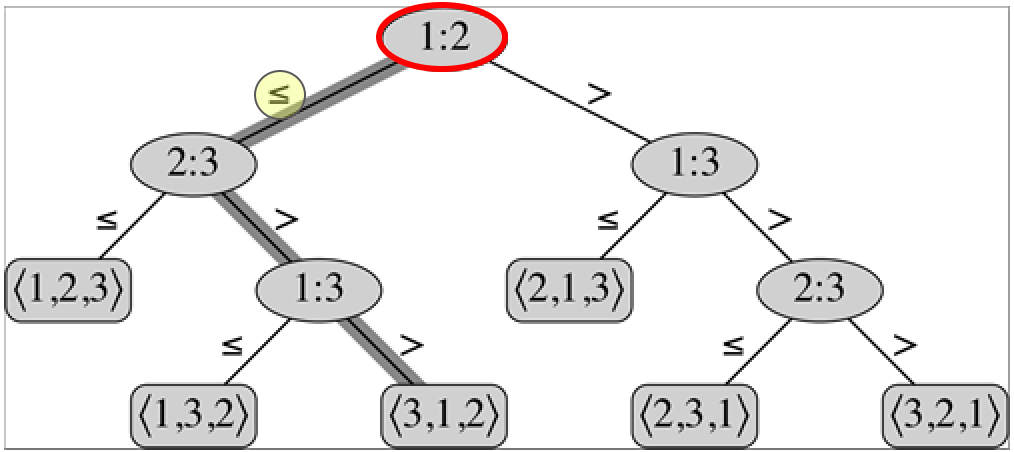
i : j

Yes

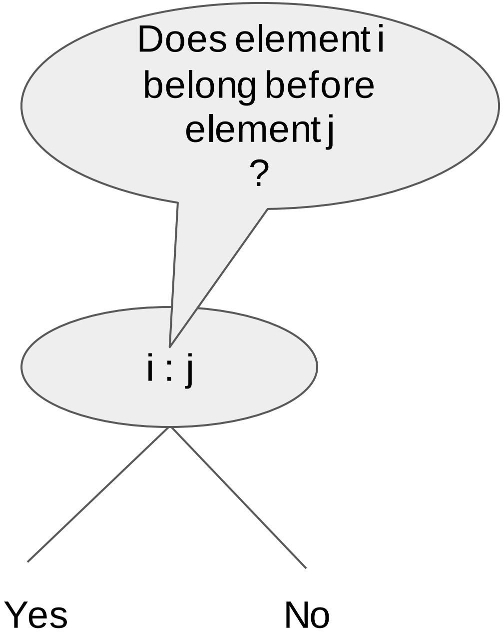
No

Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



75 99 50


Does element i belong before element j ?

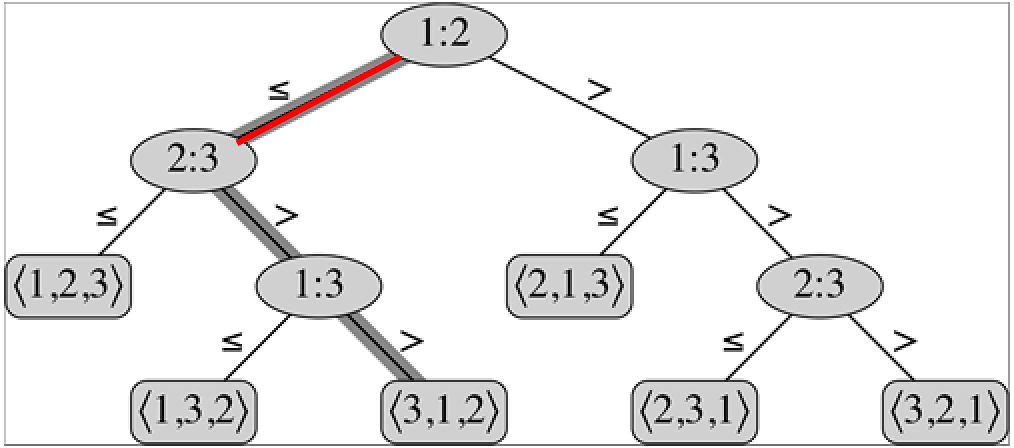
i : j

Yes

No

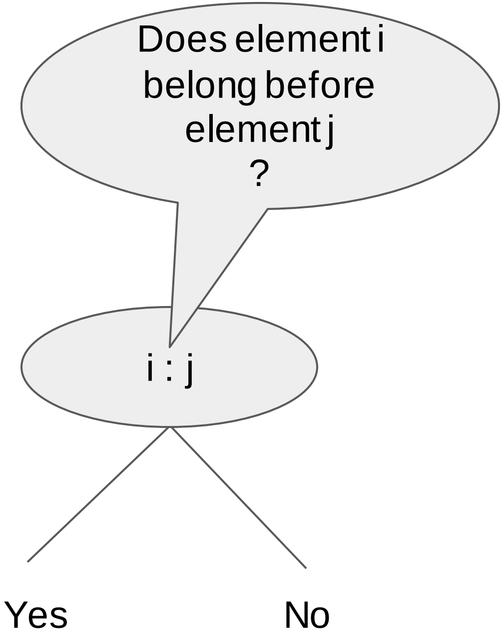
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



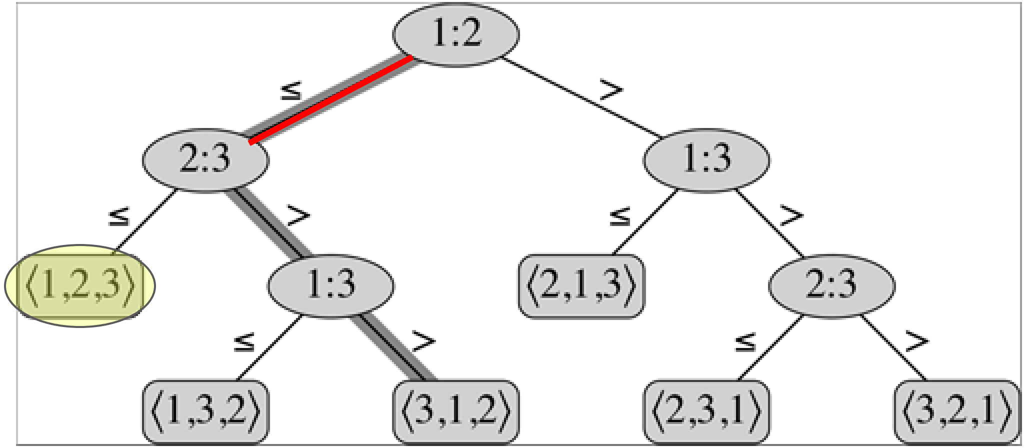
75 99 50

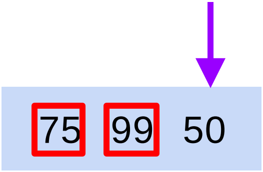
This leaves 3 places 50 could go



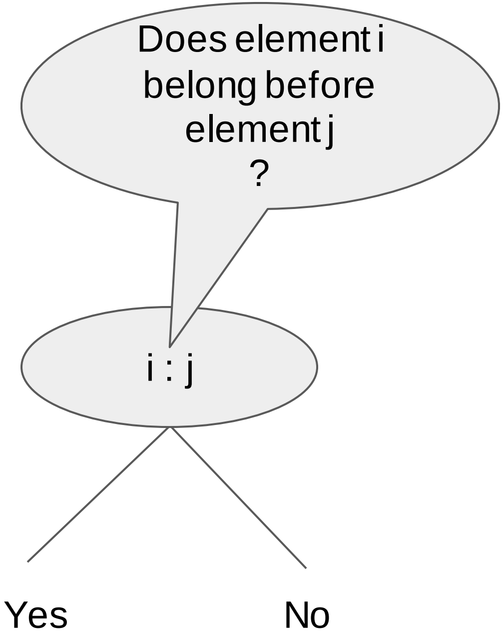
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



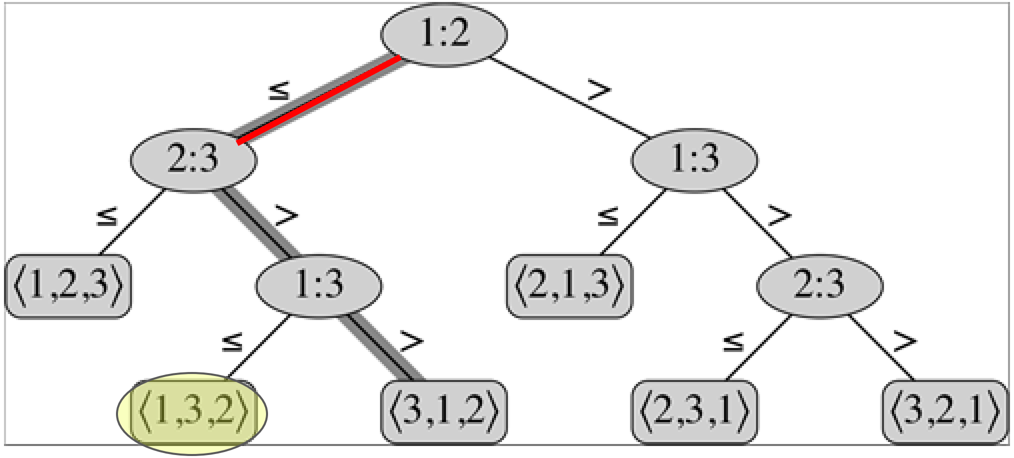


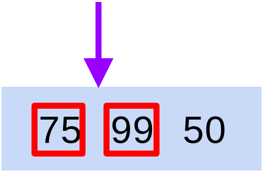
This leaves 3 places 50 could go



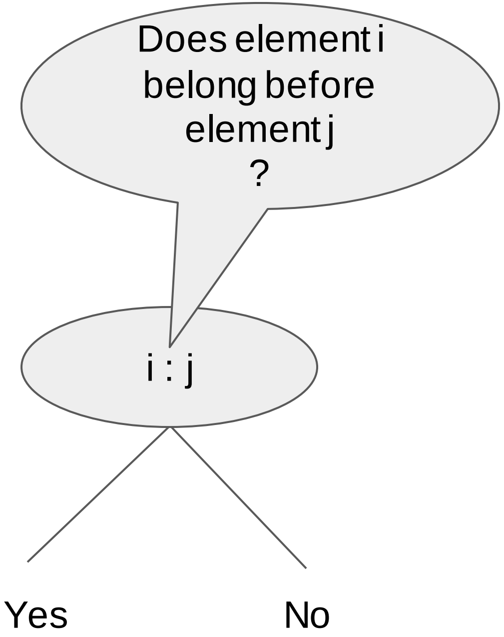
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



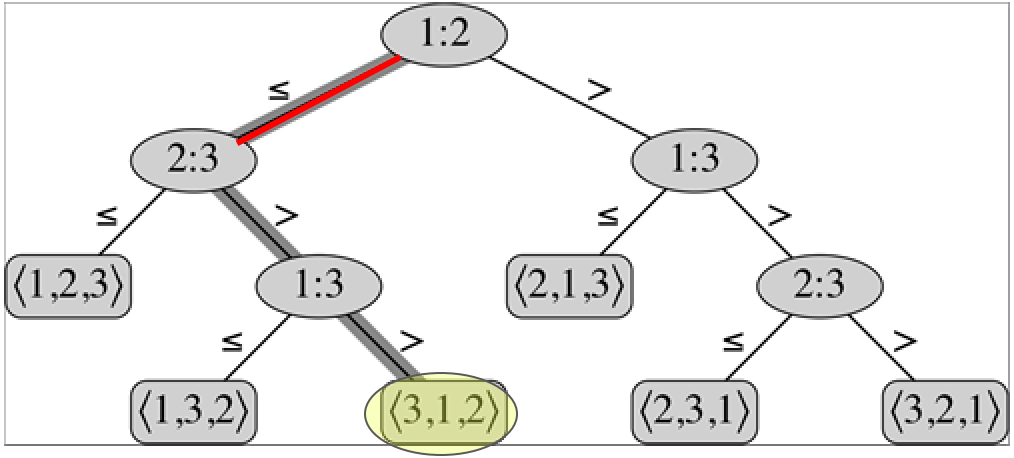


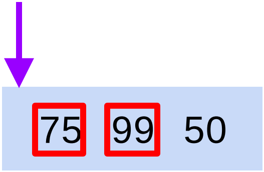
This leaves 3 places 50 could go



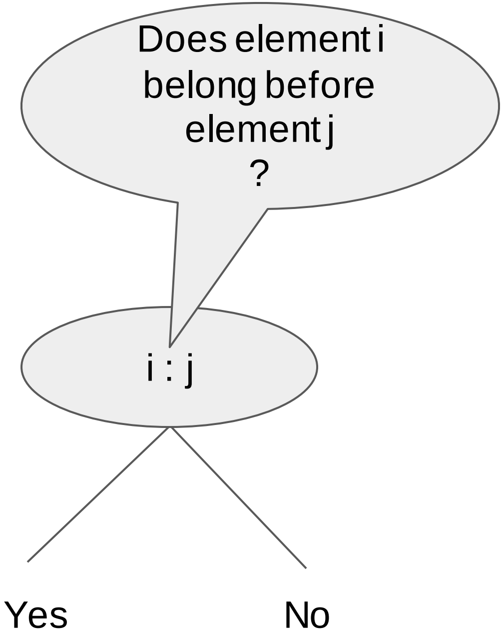
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



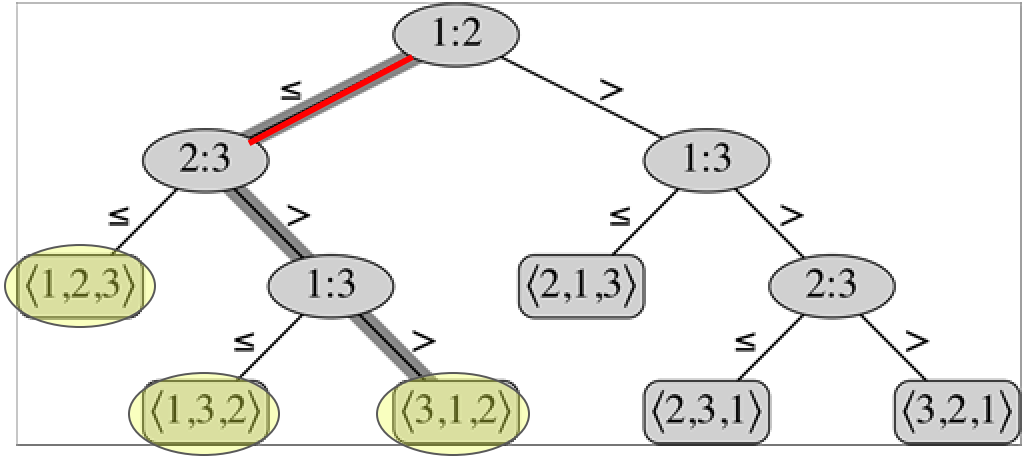


This leaves 3 places 50 could go

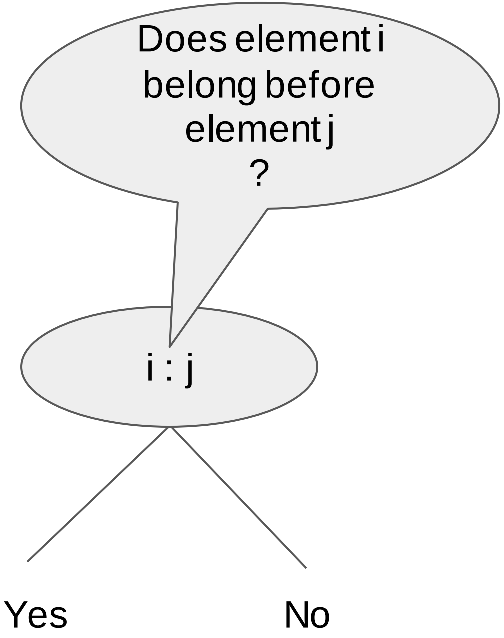


Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



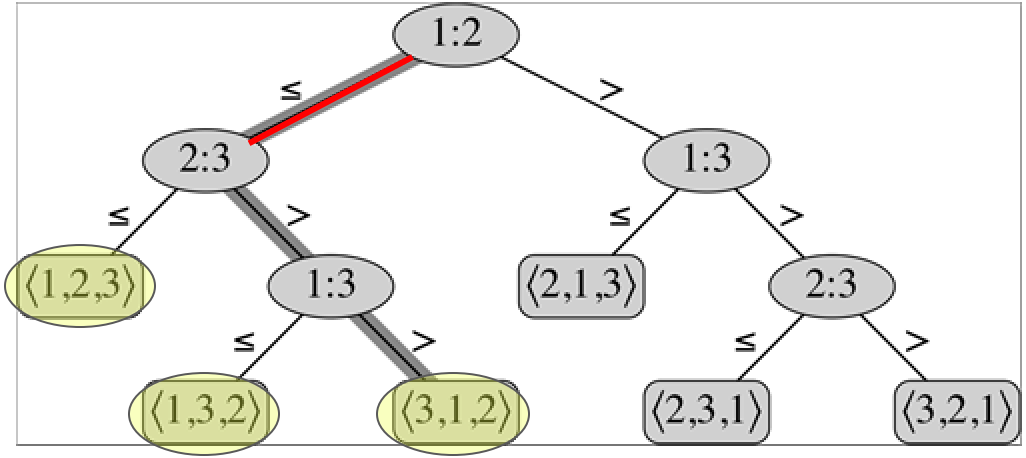
75 99 50



We cannot figure this out with just one comparison

Decision tree for sorting using comparisons

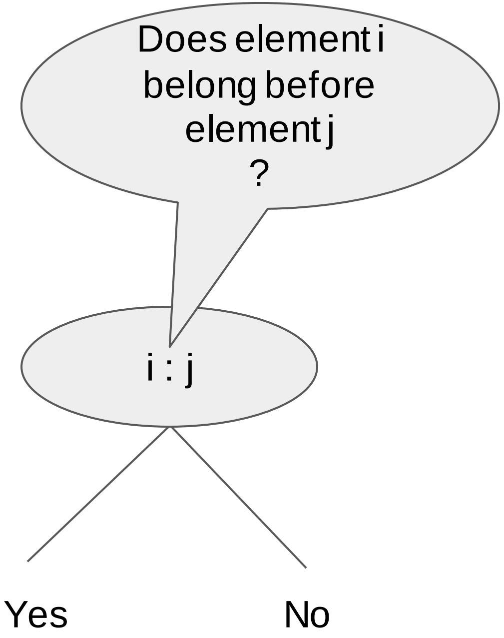
Sorting 3 elements (Figure 8.1 from text)



We cannot figure this out with just one comparison

75 99 50

Try to do it with just one


Does element i belong before element j ?

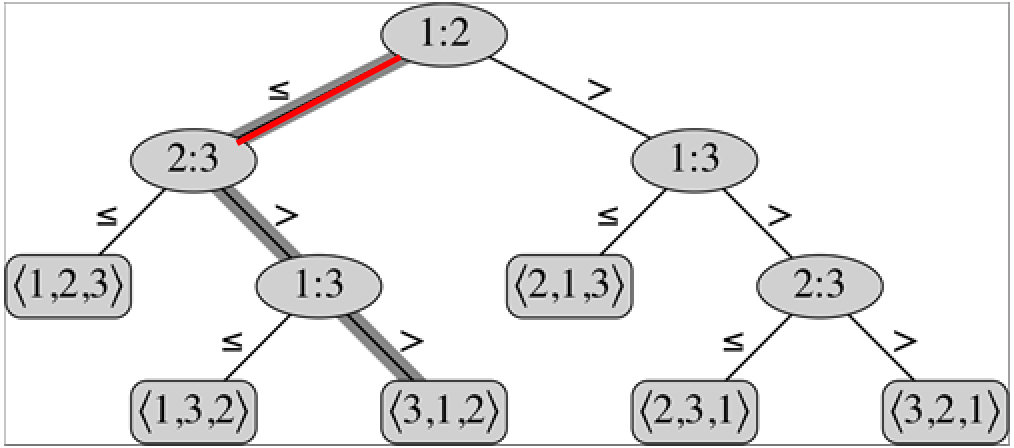
i : j

Yes

No

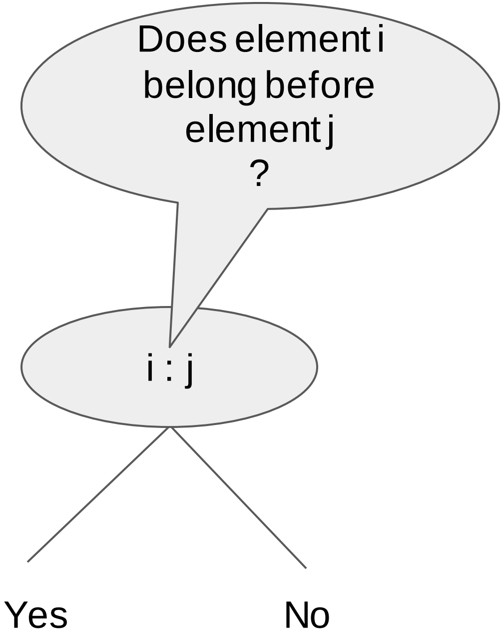
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



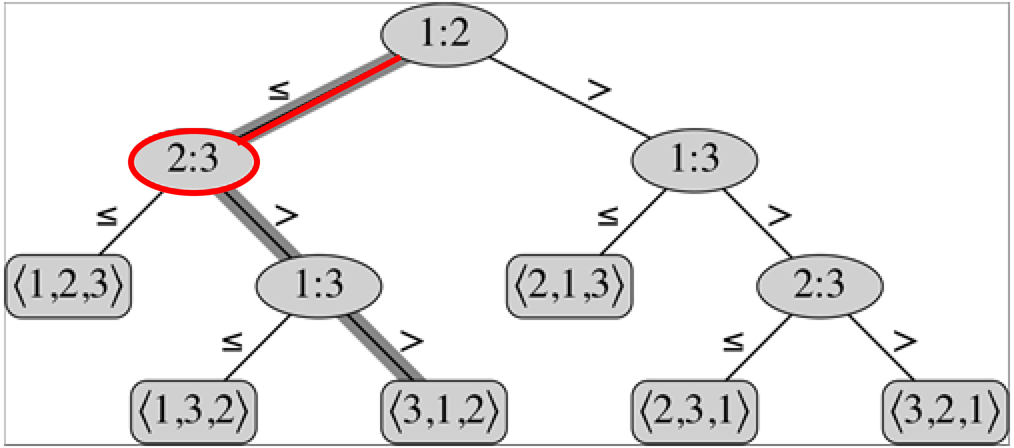
75 99 50

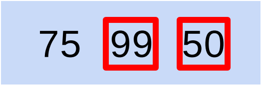
This leaves 3 places 50 could go


Does element i belong before element j ?

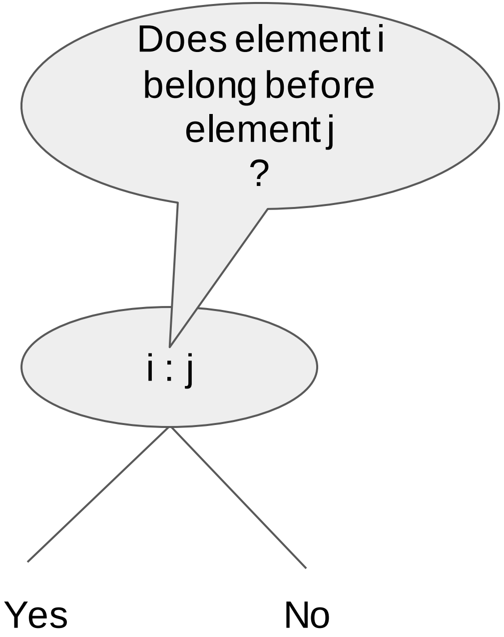
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



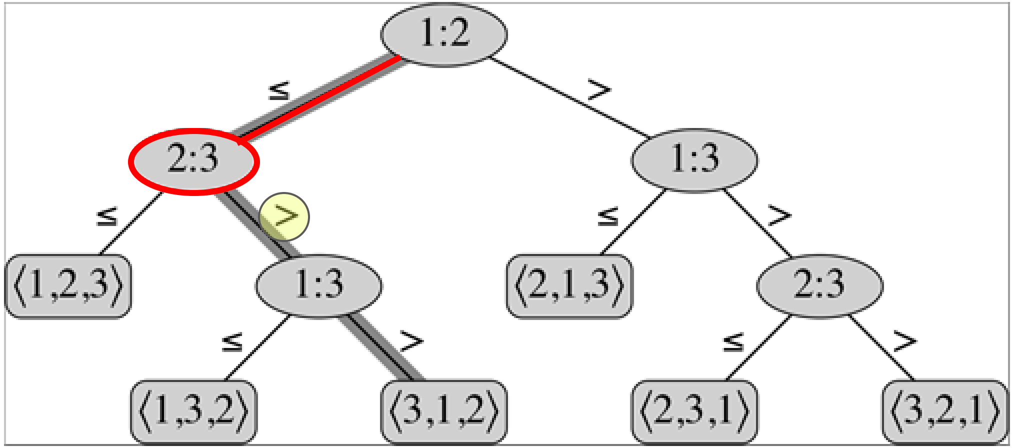


This leaves 3 places 50 could go



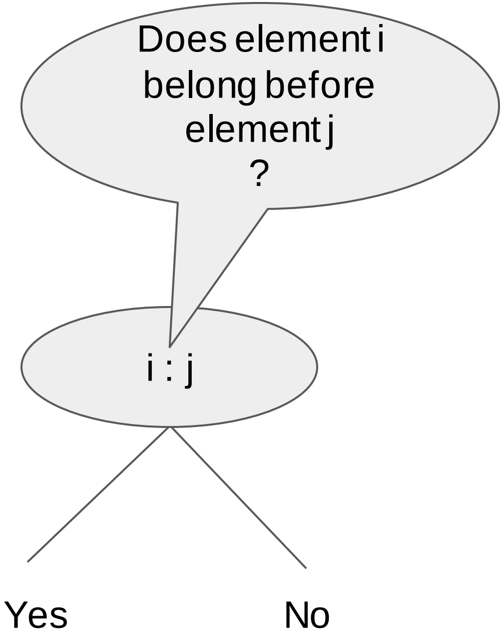
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



75 99 50

This leaves 3 places 50 could go



Does element i belong before element j ?

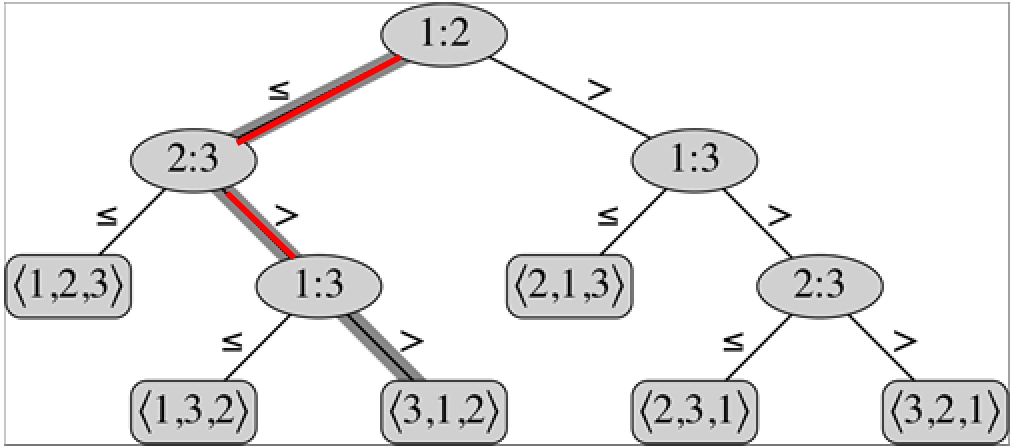
i : j

Yes

No

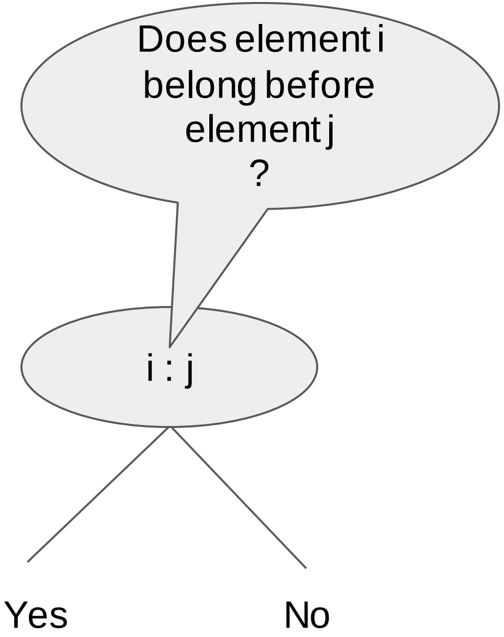
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



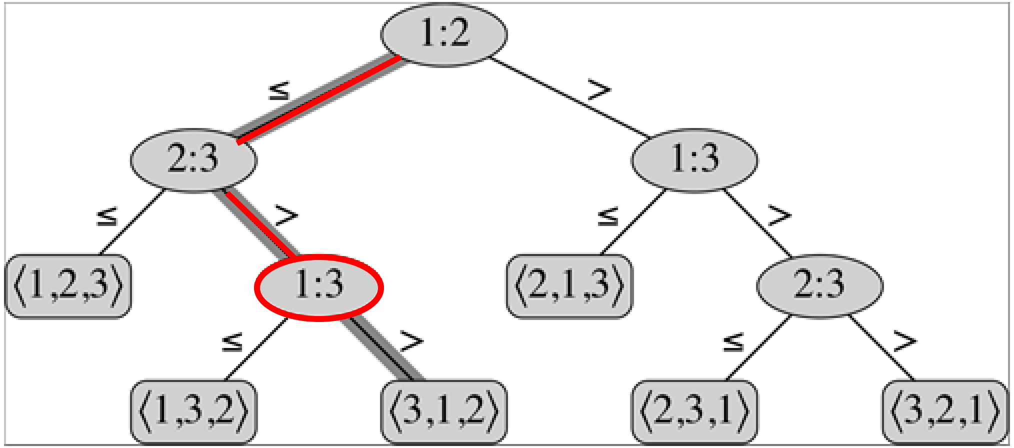
75 99 50

This leaves 3 places 50 could go


Does element i belong before element j ?

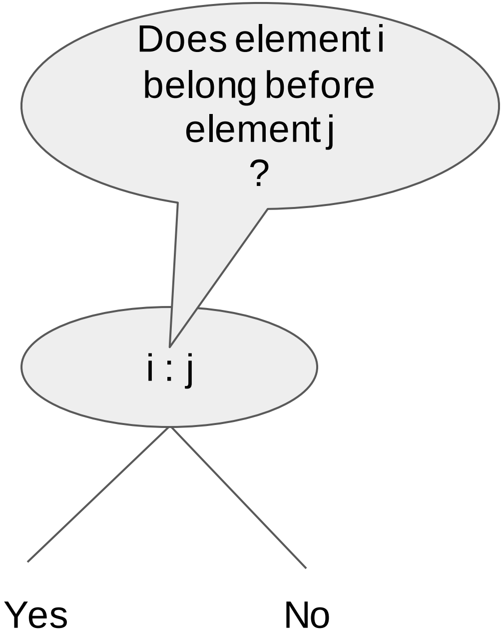
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



75 99 50

This
leaves 3
places 50
could go


Does element i
belong before
element j
?

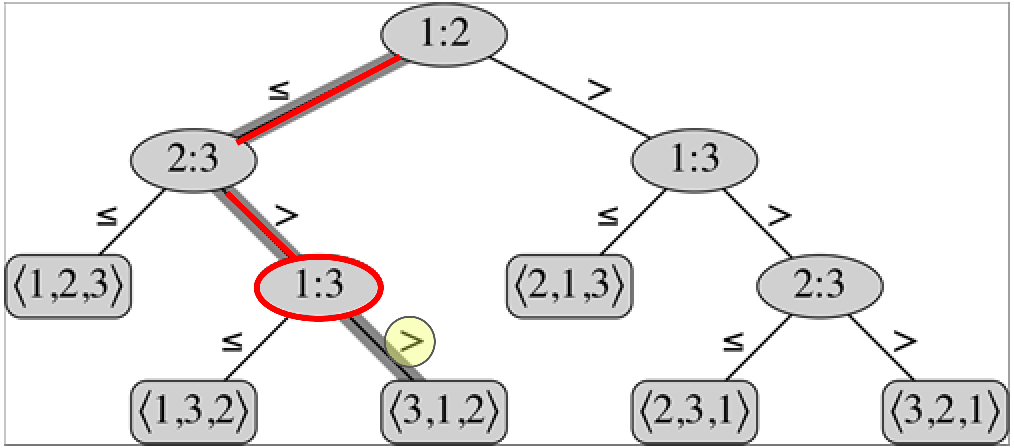
i : j

Yes

No

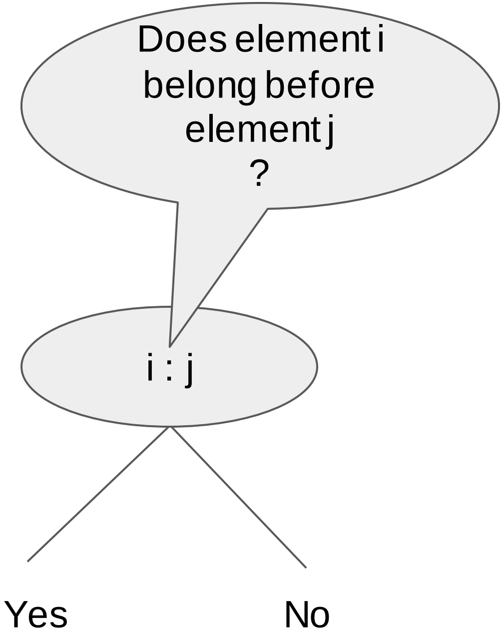
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



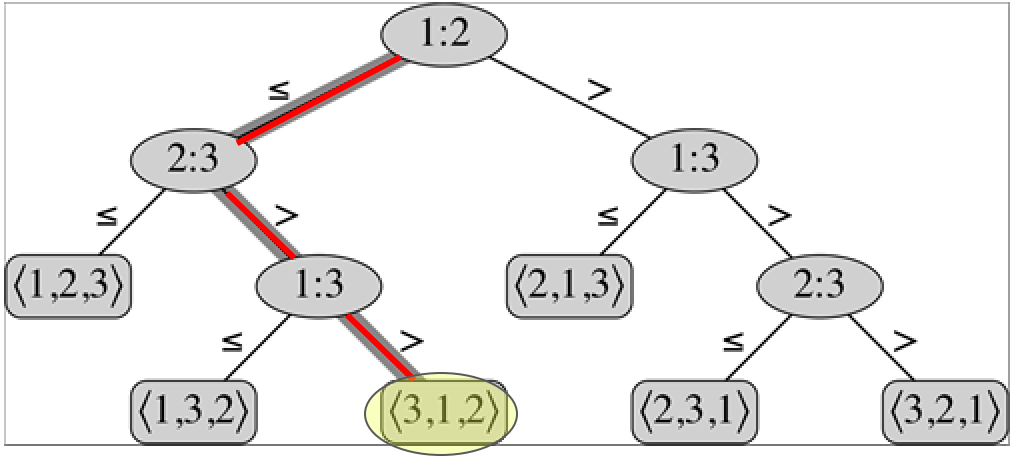
75 99 50

This leaves 3 places 50 could go


Does element i belong before element j ?

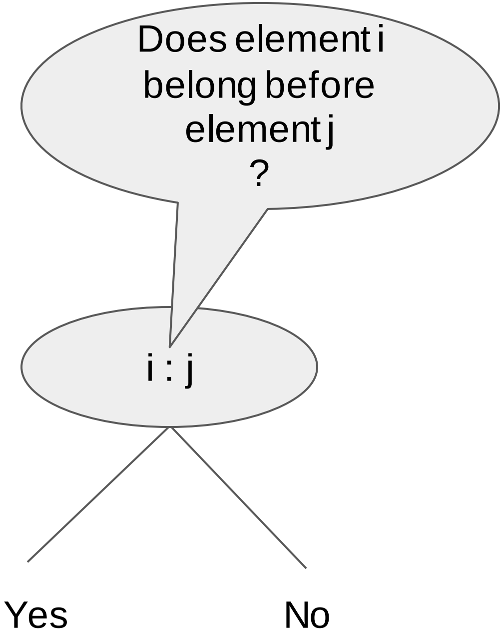
Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



75 99 50

We find our answer here


Does element i belong before element j ?

i : j

Yes

No

How to Read a Decision Tree

- For fixed input size n…
- Start at root
- Do specified operation at each internal node and follow edge based on outcome
- Leaf reached represents answer

Decision trees ←→ Sorting Algorithms

- For any comparison sort, we can construct its decision tree on inputs of size n

- (Just trace what the code does for every possible input of size n)

- For any decision tree representing a correct comparison sort, we can derive equivalent code.

- (Follow the tree as shown above.)

Decision trees ←→ Sorting Algorithms

- For any comparison sort, we can construct its decision tree on inputs of size
- (Just t ze n)
- For an t, we can derive
- (Follow the tree as shown above.)

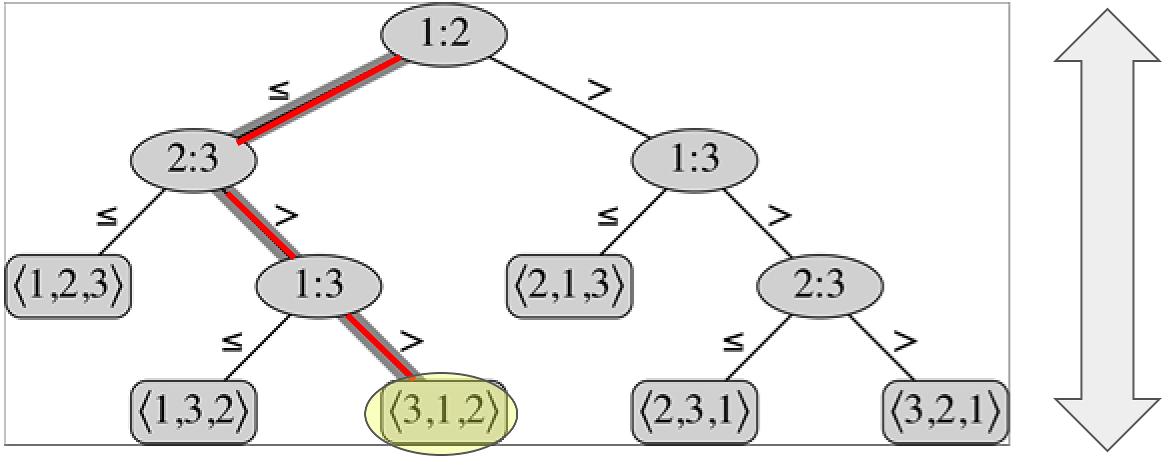
Hence, a lower bound on running time for **all** decision trees holds for **all** comparison sorts, and vice versa.

What’s the Running Time of a Decision Tree?

- As many comparisons as it takes to get from root to leaf…
- … in the worst case → maximum depth
- Hence, running time of a decision tree is its height.

Decision tree for sorting using comparisons

Sorting 3 elements (Figure 8.1 from text)



height = 3

Generic Lower Bound Argument

- Suppose **every** decision tree for a problem of size n has at least **t(n)** leaves.
- Moreover, the operation labeling each internal node has at most **w** possible outcomes.
- **Claim**: the problem requires at least $\log_w t(n)$ operations to solve.

Generic Lower Bound Argument

- **Claim**: the problem requires at least **$\log_w t(n)$** operations to solve.

- **Pf**: Tree starts with one root
- Every level of tree increases # nodes by a factor ≤ w

Generic Lower Bound Argument

- **Claim**: the problem requires at least **$\log_w t(n)$** operations to solve.

- **Pf**: Tree starts with one root
- Every level of tree increases # nodes by a factor ≤ w
- Need enough levels h s.t. $w^h \geq t(n)$.

Generic Lower Bound Argument

- **Claim**: the problem requires at least **$\log_w t(n)$** operations to solve.

- **Pf**: Tree starts with one root
- Every level of tree increases # nodes by a factor ≤ w
- Need enough levels h s.t. $w^h \geq t(n)$.
- Hence, $h \geq \log_w t(n)$. QED

Application to Comparison Sorting

- Every node of the tree is a comparison using >.
- Hence, w = ???.

Application to Comparison Sorting

- Every node of the tree is a comparison using >.
- Hence, w = 2. [# of outcomes for “Is x > y?”]
- Every leaf of the tree is a possible sorted order of n elements.
- Hence, t(n) = ???

Application to Comparison Sorting

- Every node of the tree is a comparison using >.
- Hence, w = 2. [# of outcomes for “Is x > y?”]
- Every leaf of the tree is a possible sorted order of n elements.
- Hence, t(n) = **n!** **[Really?]**

t(n) = n!

- When n = 1, only one sorted order.

$t(n) = n!$

- When $n = 1$, only one sorted order.
- For $n > 1$, assume all elements of input distinct.

t(n) = n!

- When n = 1, only one sorted order.
- For n > 1, assume all elements of input distinct.
- n possibilities for which element goes first in output.

t(n) = n!

- When n = 1, only one sorted order.
- For n > 1, assume all elements of input distinct.
- n possibilities for which element goes first in output.
- Given 1st choice, # of orders of remaining n-1 elts is (n-1)!

$t(n) = n!$

- When $n = 1$, only one sorted order.
- For $n > 1$, assume all elen
- n possibilities for which el
- Given 1st choice, # of orders of remaining $n-1$ elts is $(n-1)!$

We applied the inductive hypothesis.

t(n) = n!

- When n = 1, only one sorted order.
- For n > 1, assume all elements of input distinct.
- n possibilities for which element goes first in output.
- Given 1st choice, # of orders of remaining n-1 elts is (n-1)!
- Hence, n x (n-1)! = n! possible orders of n elements. QED

Summary

- For comparison sorting, $w = 2$, $t(n) = n!$
- Hence, by our general theorem, sorting an array of size n requires at least $\log_2 n!$ comparisons in the worst case.

Summary

- For comparison sorting, w = 2, t(n) = n!
- Hence, by our general theorem, sorting an array of size n requires at least $\log_2 n!$ comparisons in the worst case.
- ***Wait, how big is log n! ???***

Bounding log(n!)

- log(n!) = log(n x (n-1) x (n-2) x ... x 1)

Bounding log(n!)

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log n + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

Bounding log(n!)

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\leq \log(n) + \log(n) + \log(n) + \ldots + \log(n) + \log(n)$

Bounding log(n!)

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\leq \log(n) + \log(n) + \log(n) + \ldots + \log(n) + \log(n)$

  $= n \log(n)$

Bounding log(n!)

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\leq \log(n) + \log(n) + \log(n) + \ldots + \log(n) + \log(n)$

  $= n \log(n)$

  So $\log(n!) = O(n \log n)$ <-- Upper Bound

Bounding log(n!)

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\leq \log(n) + \log(n) + \log(n) + \ldots + \log(n) + \log(n)$

  $= n \log(n)$

  So $\log(n!) = O(n \log n)$ <-- Upper Bound

  --look familiar? bound on *n* heap operations!

Bounding log(n!): lower bound*

- log(n!) = log(n x (n-1) x (n-2) x ... x 1)

  = log (n) + log (n-1) + log (n-2) + … + log(2) + log(1)

* proof written by WUSTL student Aidan Kelley

Bounding log(n!): lower bound

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\geq \log(n) + \log(n-1) + \ldots + \log(n/2 + 1)$ <-- first n/2 terms

Bounding log(n!): lower bound

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\geq \log(n) \quad + \log(n-1) + \ldots + \log(n/2 + 1)$ <-- first n/2 terms

  $\geq \log(n/2) + \log(n/2) + \ldots + \log(n/2)$ <-- l.b. on each term

Bounding log(n!): lower bound

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \dots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \dots + \log(2) + \log(1)$

  $\geq \log(n) + \log(n-1) + \dots + \log(n/2 + 1)$ <-- first n/2 terms

  $\geq \log(n/2) + \log(n/2) + \dots + \log(n/2)$ <-- l.b. on each term

  $= (n/2) \log(n/2)$

Bounding log(n!): lower bound

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\geq \log(n) + \log(n-1) + \ldots + \log(n/2 + 1)$ <-- first n/2 terms

  $\geq \log(n/2) + \log(n/2) + \ldots + \log(n/2)$ <-- l.b. on each term

  $= (n/2) \log(n/2)$

  $= (n/2)(\log n - \log 2)$

Bounding log(n!): lower bound

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

$$= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$$

$$\geq \log(n) \quad + \log(n-1) + \ldots + \log(n/2 + 1) \qquad \text{<-- first n/2 terms}$$

$$\geq \log(n/2) + \log(n/2) + \ldots + \log(n/2) \qquad \text{<-- l.b. on each term}$$

$$= (n/2) \log(n/2)$$

$$= (n/2)(\log n - \log 2)$$

$$= (n \log n / 2) - (n/2)$$

Bounding log(n!): lower bound

- log(n!) = log(n x (n-1) x (n-2) x ... x 1)

  = log (n) + log (n-1) + log (n-2) + … + log(2) + log(1)

  ≥ log (n) + log (n-1) + … + log(n/2 + 1) <-- first n/2 terms

  ≥ log (n/2) + log (n/2) + … + log(n/2) <-- l.b. on each term

  = (n/2) log (n/2)

  = (n/2)(log n – log 2)

  = (n log n / 2) - (n/2)

  So log(n!) = Ω(n log n) <-- Lower bound

* proof written by WUSTL student Aidan Kelley

Bounding log(n!)

- log(n!) = log(n x (n-1) x (n-2) x ... x 1)

  = log (n) + log (n-1) + log (n-2) + … + log(2) + log(1)

  log(n!) = O(n log n)   <-- Upper bound

  log(n!) = Ω(n log n)   <-- Lower bound

Bounding log(n!)

- $\log(n!) = \log(n \times (n-1) \times (n-2) \times \ldots \times 1)$

  $= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(2) + \log(1)$

  $\log(n!) = O(n \log n)$ <-- Upper bound

  $\log(n!) = \Omega(n \log n)$ <-- Lower bound

$\log(n!) = \Theta(n \log n)$

Summary

- For comparison sorting, w = 2, t(n) = n!

- Hence, by our general theorem, sorting an array of size n requires **at least $\log_2 n!$** comparisons in the worst case.

Summary

- For comparison sorting, w = 2, t(n) = n!
- Hence, by our general theorem, sorting an array of size n requires **Ω(n log n)** comparisons in the worst case.
- **→ MergeSort and HeapSort are asymptotically optimal comparison sorts!**

OK, so it’s **impossible** to sort in time less than n log n…

OK, so it’s **impossible** to sort in time less than n log n… using comparisons

Next, let's see **how** we **can** sort in time less than n log n.

Breaking the n log n barrier

- Our lower bound is for **comparison sorts**, which work on items from any totally ordered set.

- To sort faster, we need to be able to inspect input using ops other than comparisons.

- Will limit attention to sorting **integers**.

Counting Sort

- Assume our inputs are n integers in range [0, k).
- Count how often each value occurs in input.
- Write that many values to output.

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	
1	
2	
3	
4	

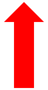
Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	
1	1
2	
3	
4	

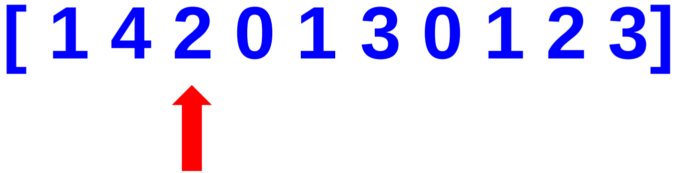
Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]



Value	Count
0	
1	1
2	
3	
4	1

Counting Sort (k = 5)



[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	
1	1
2	1
3	
4	1

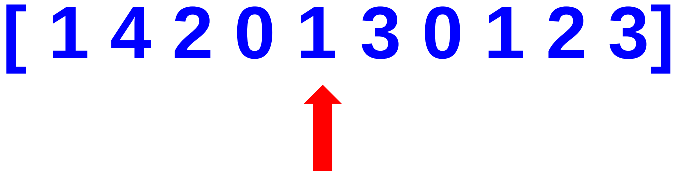
Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

↑

Value	Count
0	1
1	1
2	1
3	
4	1

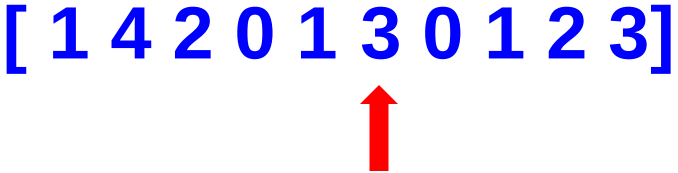
Counting Sort (k = 5)



[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	1
1	2
2	1
3	
4	1

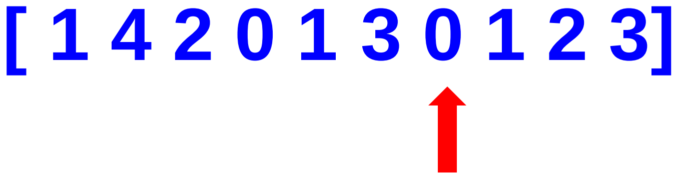
Counting Sort (k = 5)



[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	1
1	2
2	1
3	1
4	1

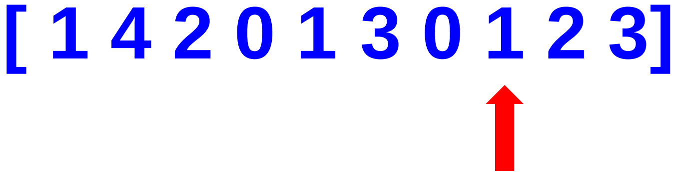
Counting Sort (k = 5)



[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	2
1	2
2	1
3	1
4	1

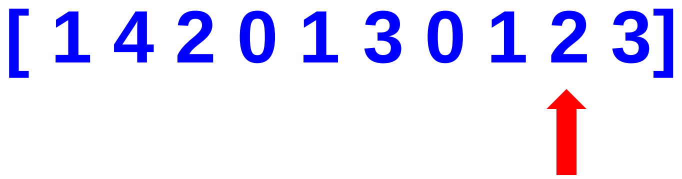
Counting Sort (k = 5)



[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	2
1	3
2	1
3	1
4	1

Counting Sort (k = 5)



[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	2
1	3
2	2
3	1
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[ ]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[0 0]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[0 0 1 1 1]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[0 0 1 1 1 2 2]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[0 0 1 1 1 2 2 3 3]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[0 0 1 1 1 2 2 3 3 4]

Value	Count
0	2
1	3
2	2
3	2
4	1

Counting Sort (k = 5)

[1 4 2 0 1 3 0 1 2 3]

[0 0 1 1 1 2 2 3 3 4]

Value	Count
0	2
1	3
2	2
3	2
4	1

Cost = $\Theta(n + k)$

Fun Facts About Counting Sort

- Counting sort is a “linear-time” sort (in n)
    - Still depends on k
- Can extend to sort arbitrary items with integer keys
- Highly efficient when max value k is small vs n

Fun Facts About Counting Sort

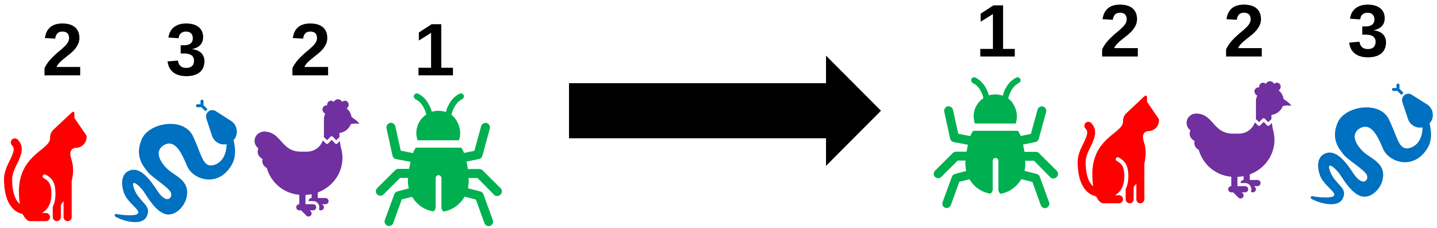
- Counting sort is a “linear-time” sort (in n)
    - Still depends on k
- Can extend to sort arbitrary items with integer keys
- Highly efficient when max value k is small vs n
- But what if k is large?

Radix Sort

- Divide each input integer into d digits
- Digits may be in any base k; we'll use base 10 in example
- Sort using d successive passes of counting sort
- jth pass uses jth digit of each input as sorting key

Radix Sort – Key Requirements

- Sort using d successive passes of counting sort.
- We sort by **least significant** digit first.
- Sort in each pass must be **stable** – never inverts order of two inputs with the same key.



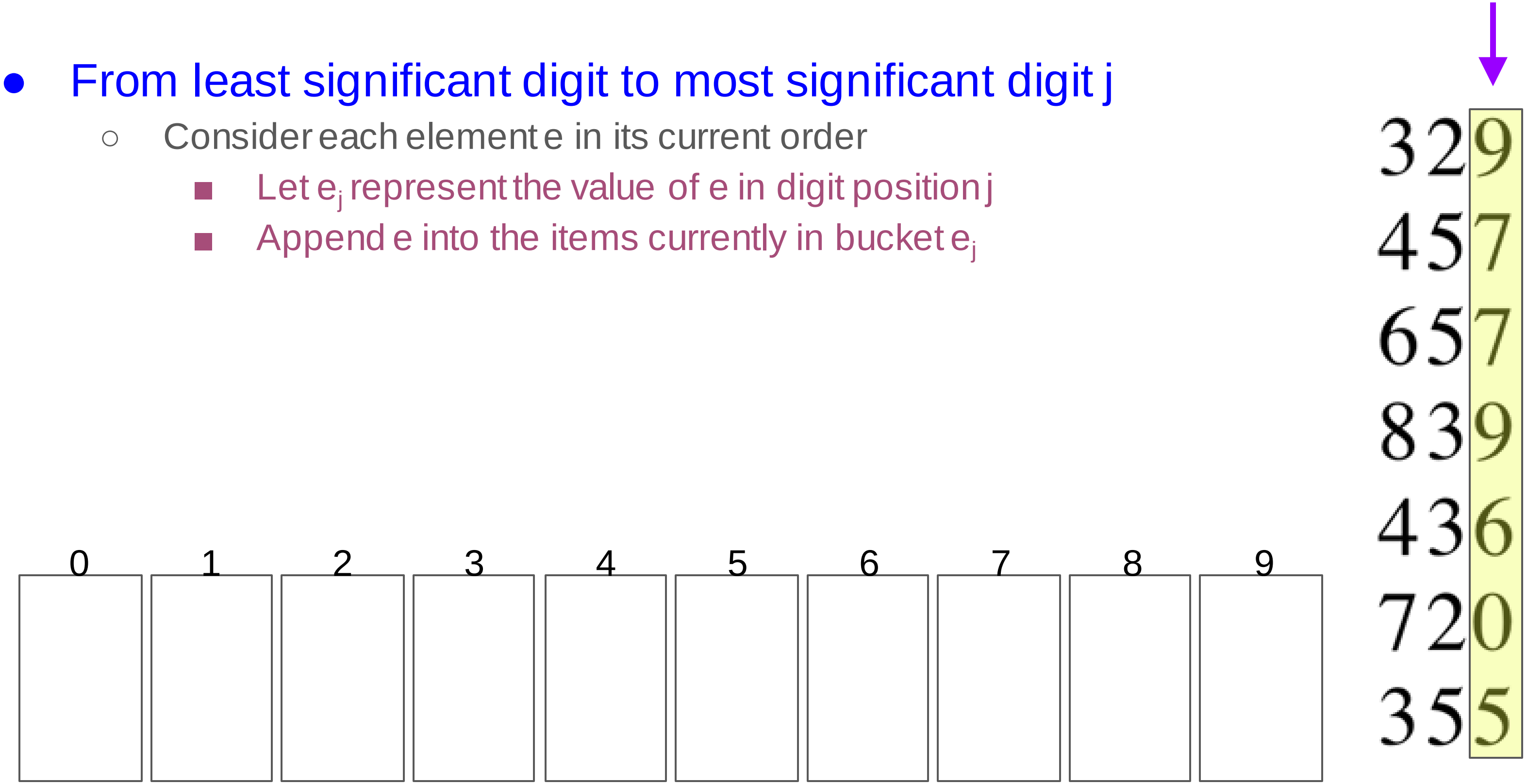
Radix Sort – Key Requirements

- Sort using d successive passes of counting sort.
- We sort by **least significant** digit first.
- Sort in each pass must be **stable** – neve f two inputs with the same key.

NOT STABLE!

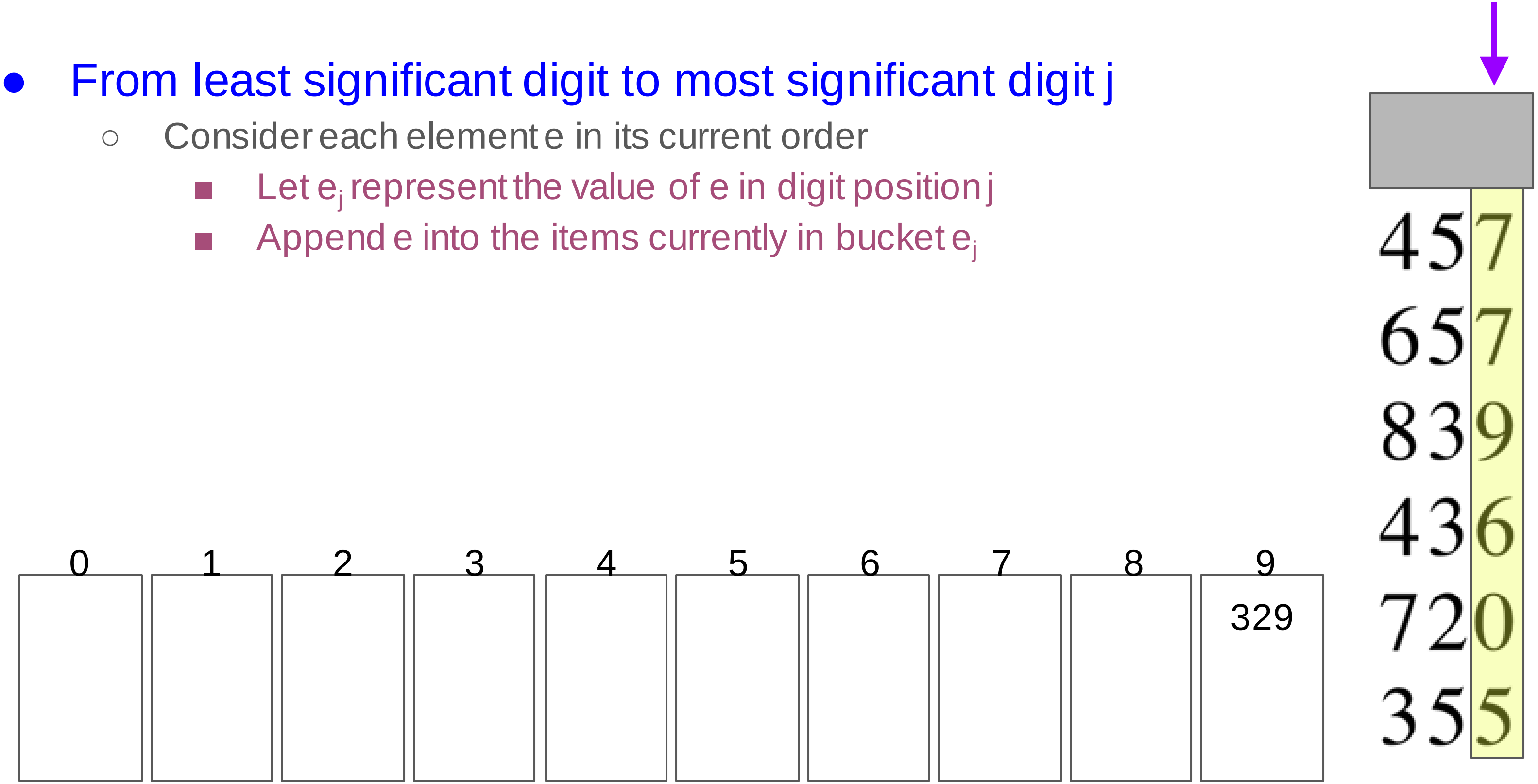
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



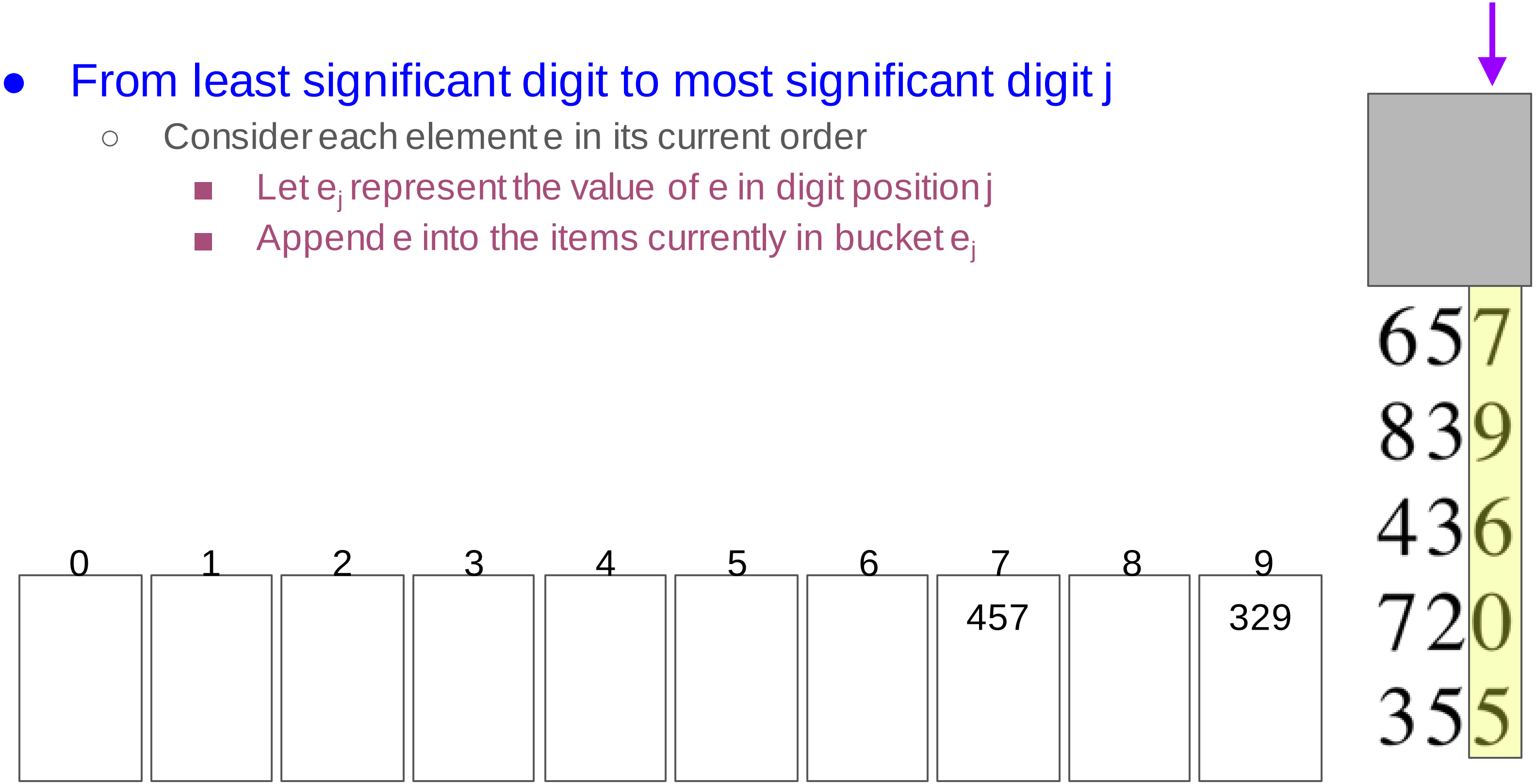
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



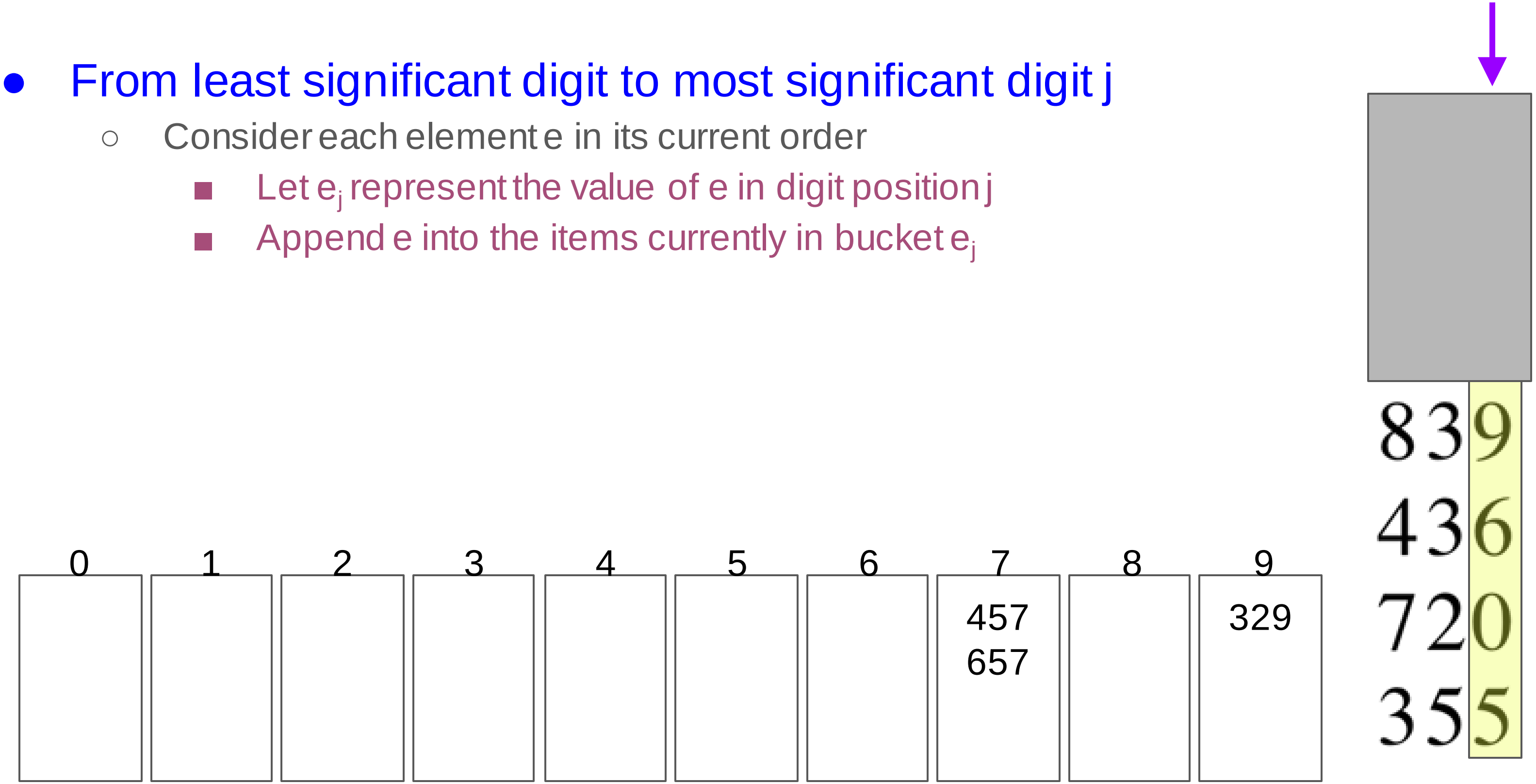
Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



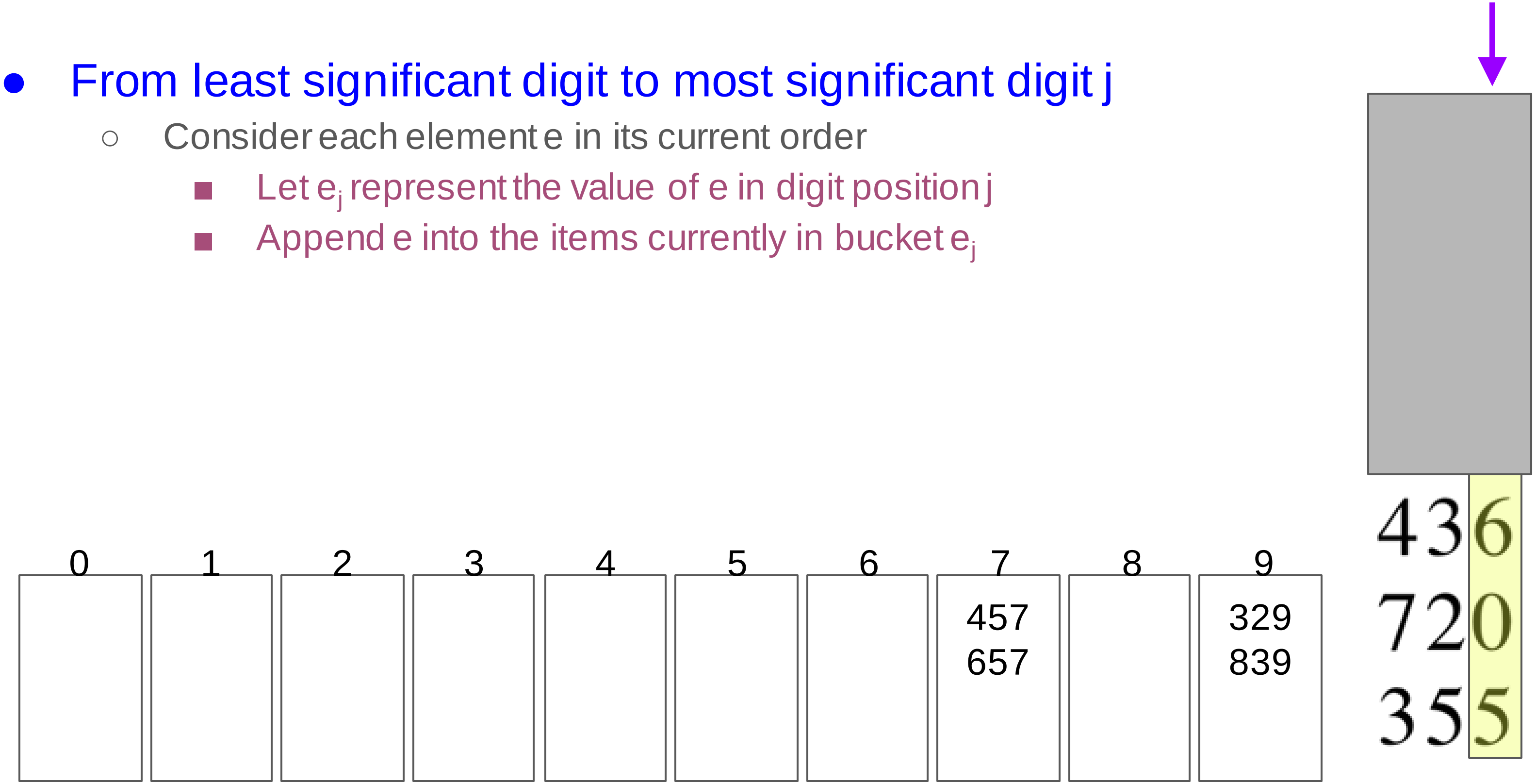
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



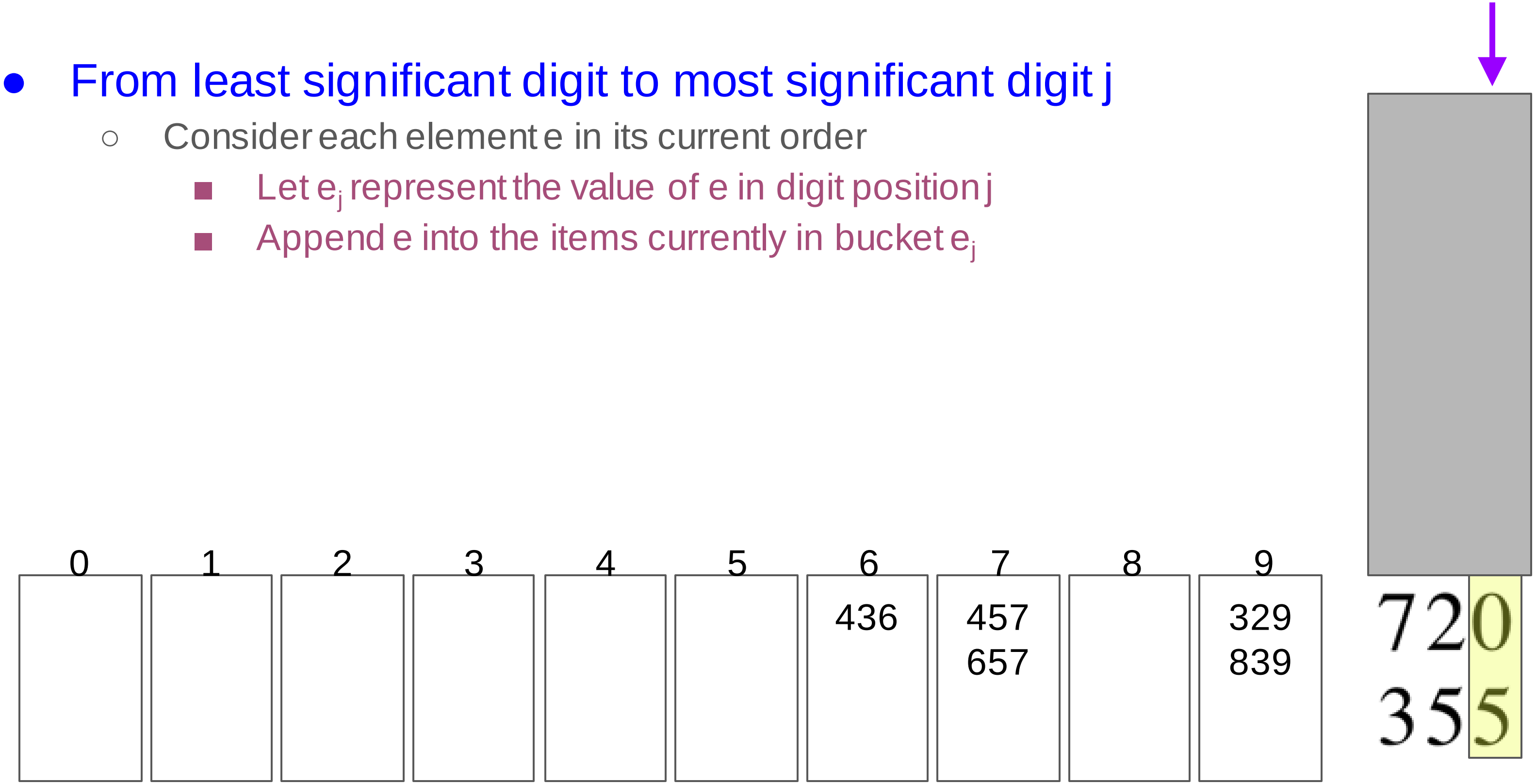
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

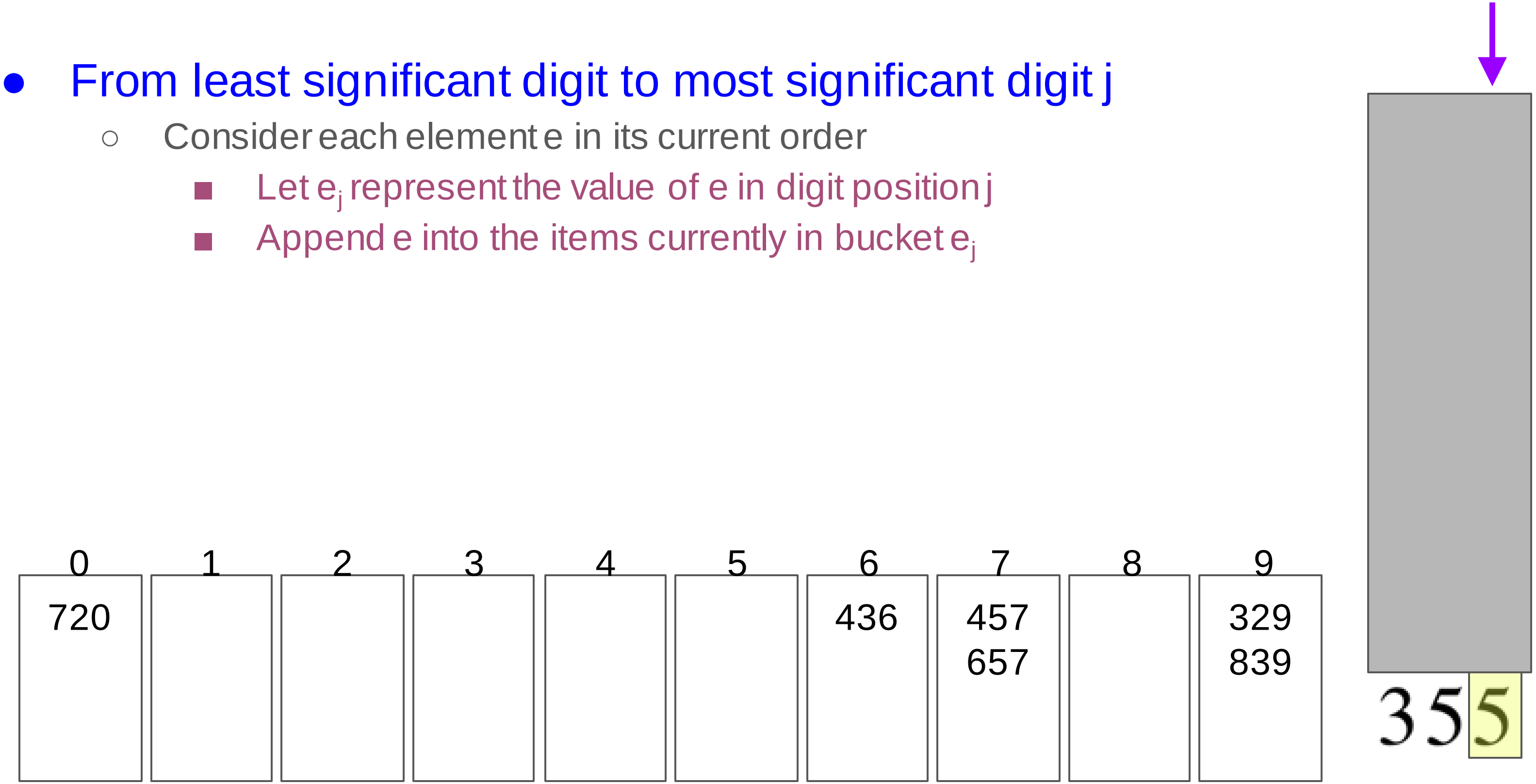


0	1	2	3	4	5	6	7	8	9
						436	457 657		329 839

720
355

Radix Sort

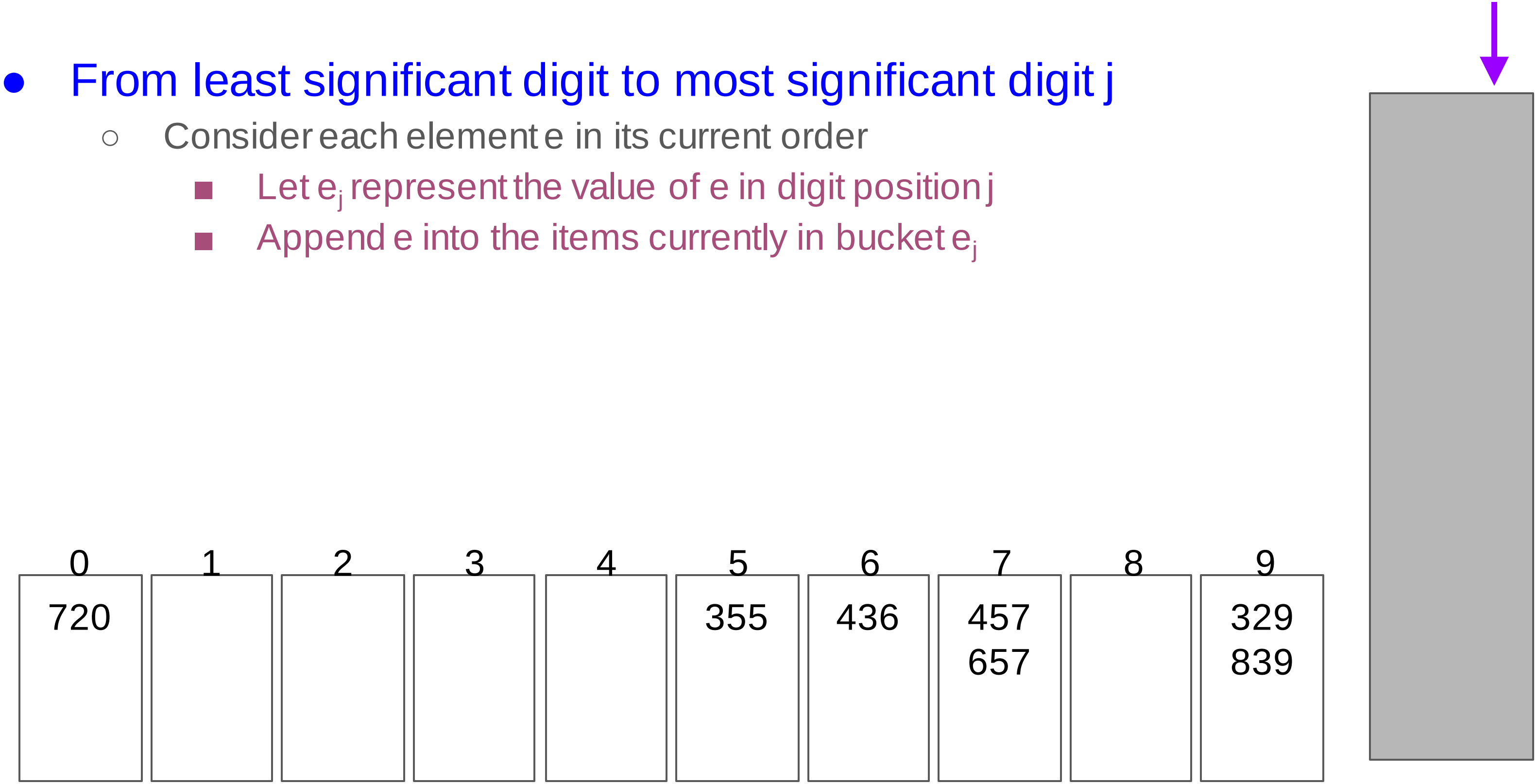
- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



0	1	2	3	4	5	6	7	8	9
720						436	457 657		329 839

Radix Sort

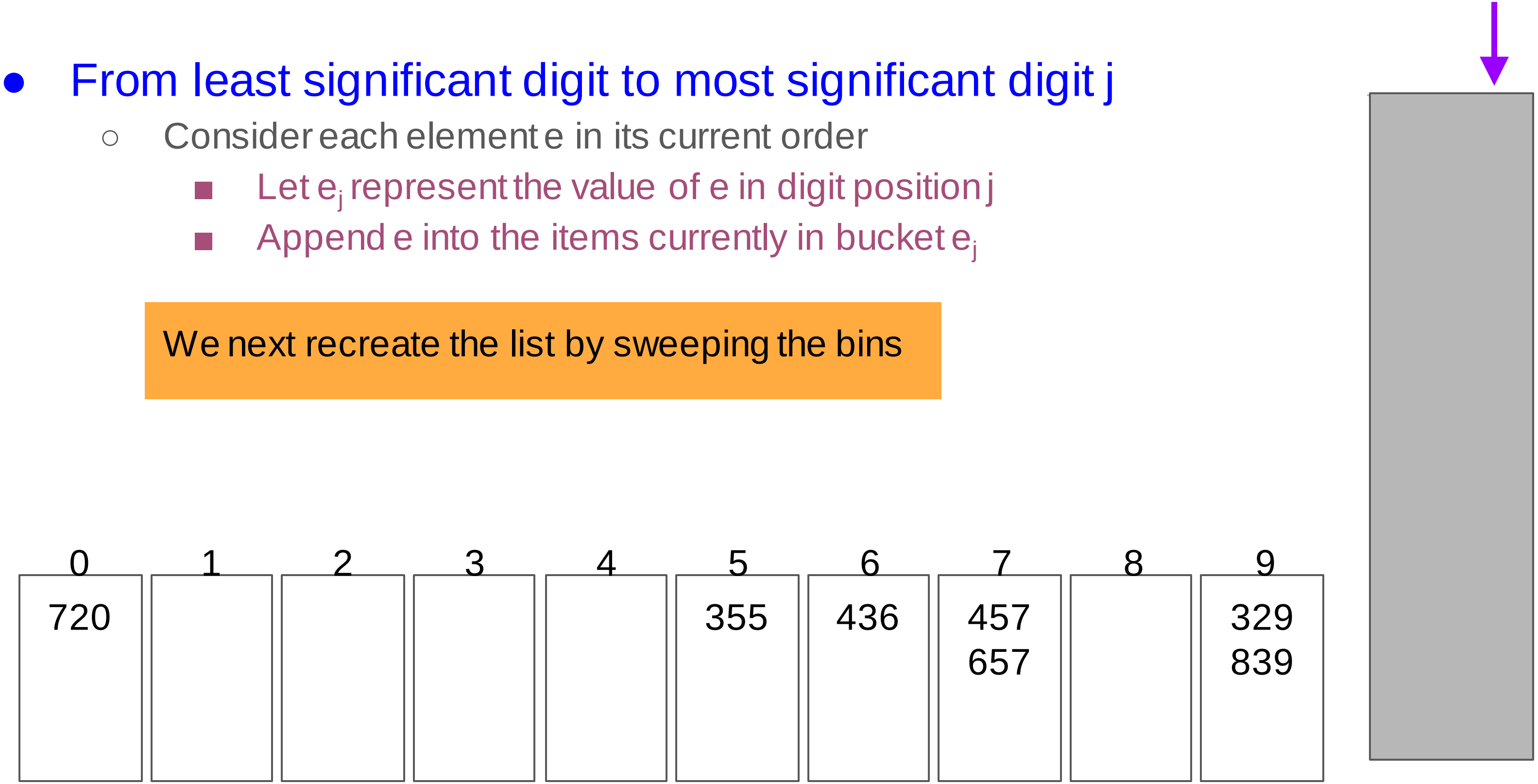
- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j

We next recreate the list by sweeping the bins

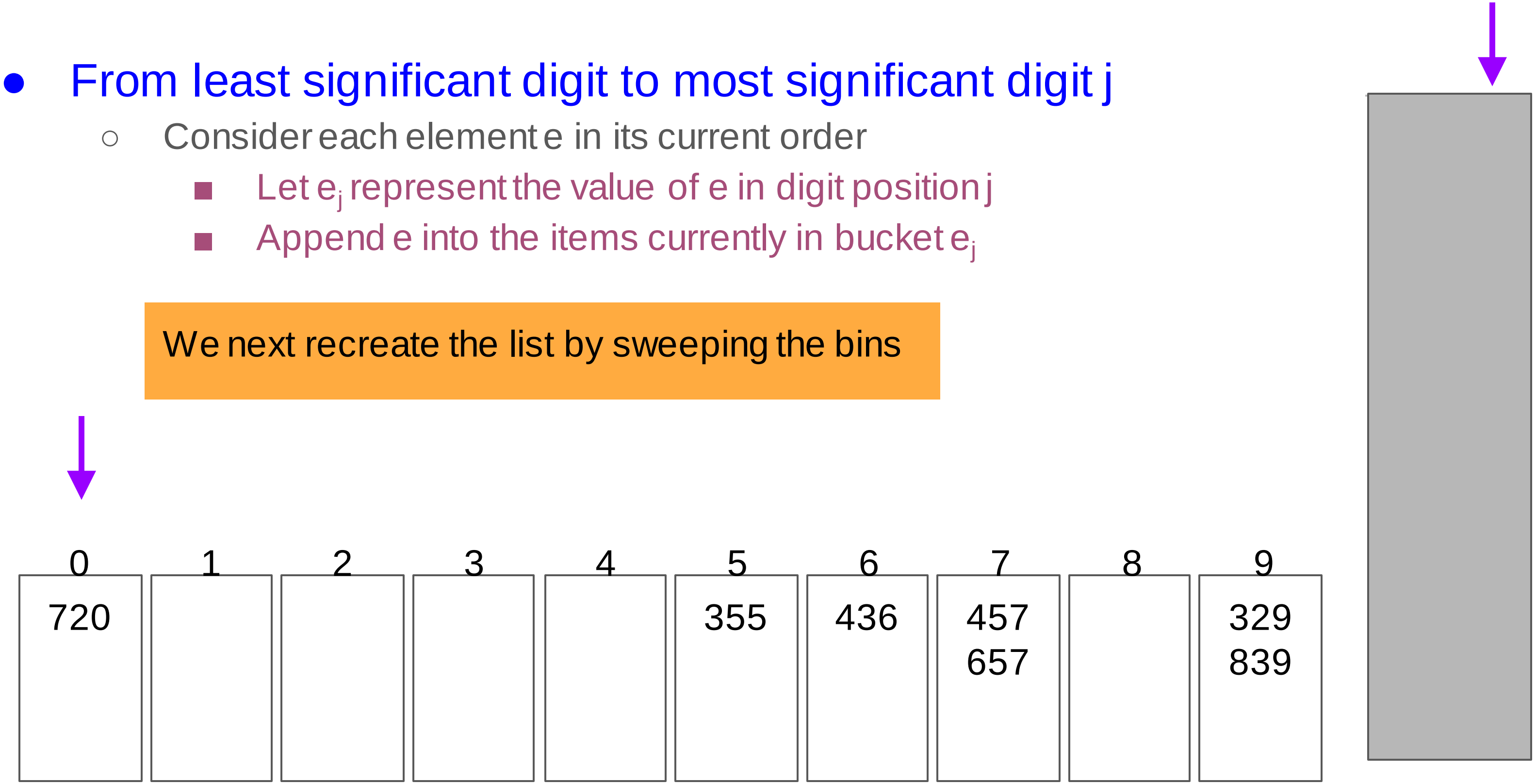


0	1	2	3	4	5	6	7	8	9
720					355	436	457 657		329 839

Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

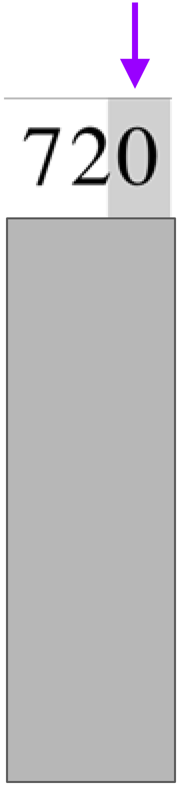
We next recreate the list by sweeping the bins

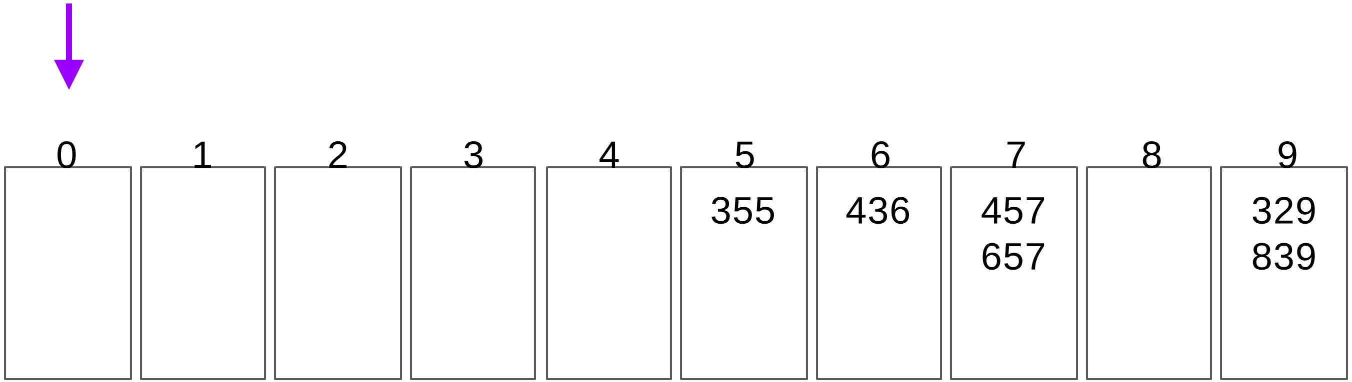


0	1	2	3	4	5	6	7	8	9
720					355	436	457 657		329 839

Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j

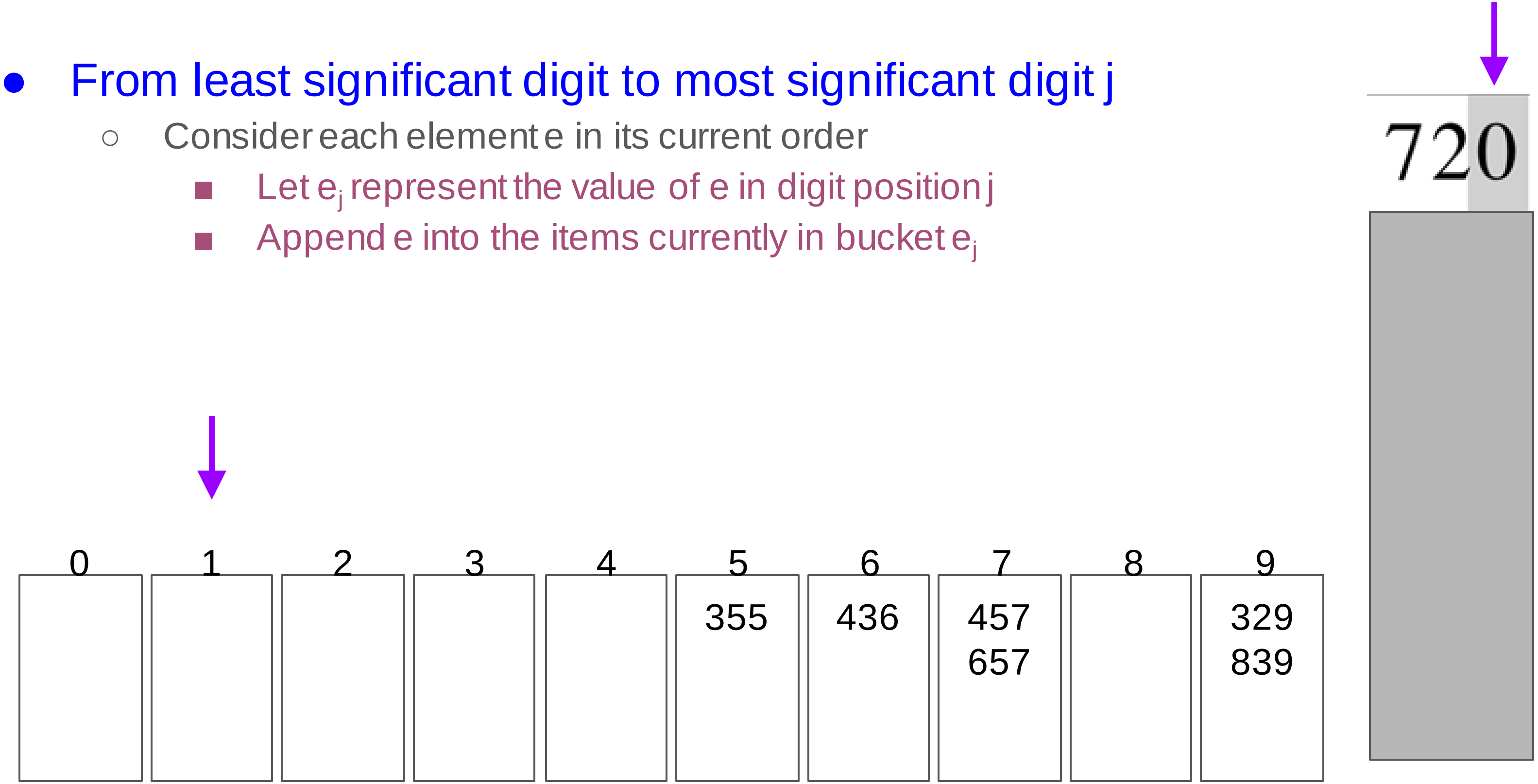




0	1	2	3	4	5	6	7	8	9
					355	436	457 657		329 839

Radix Sort

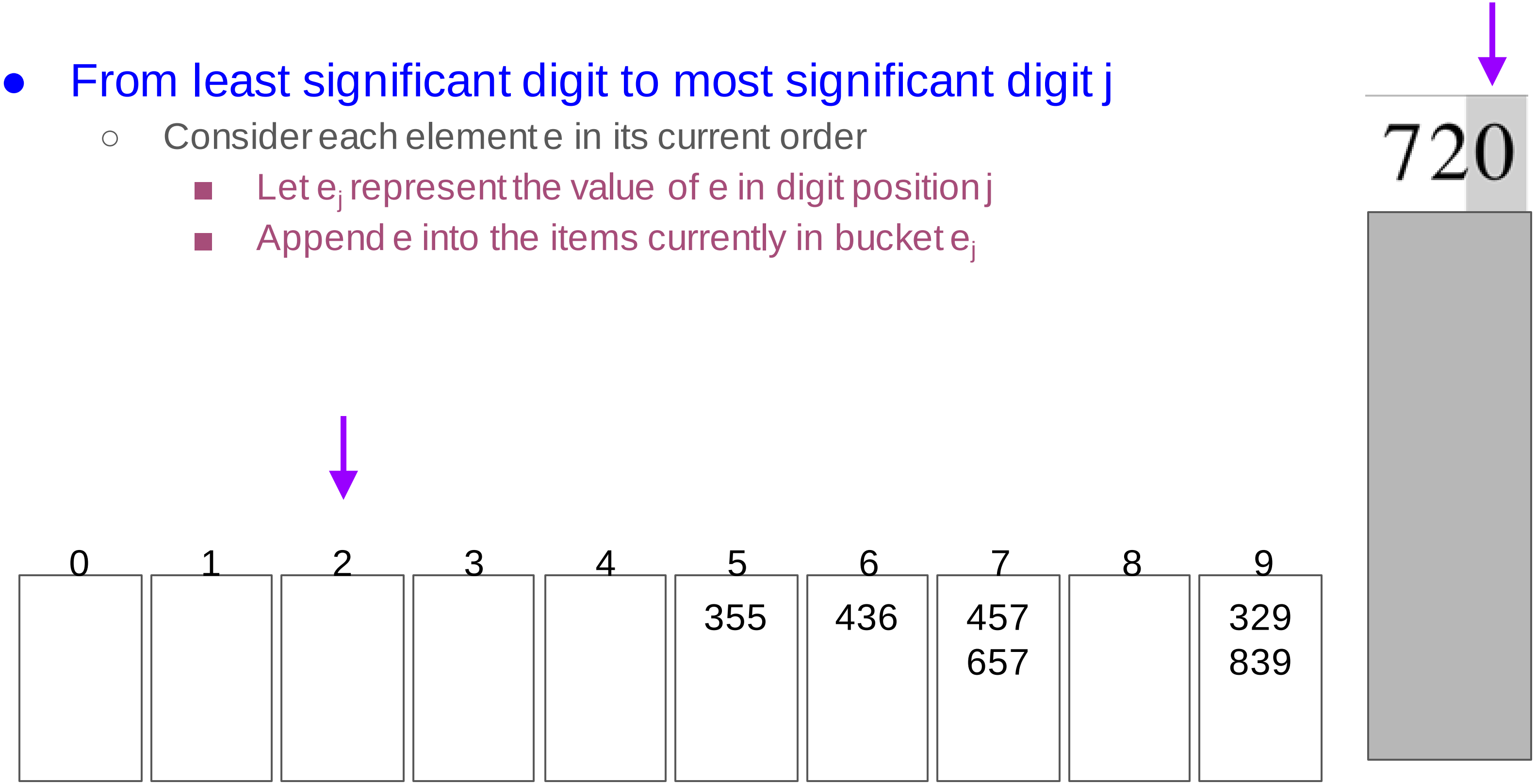
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



0	1	2	3	4	5	6	7	8	9
					355	436	457 657		329 839

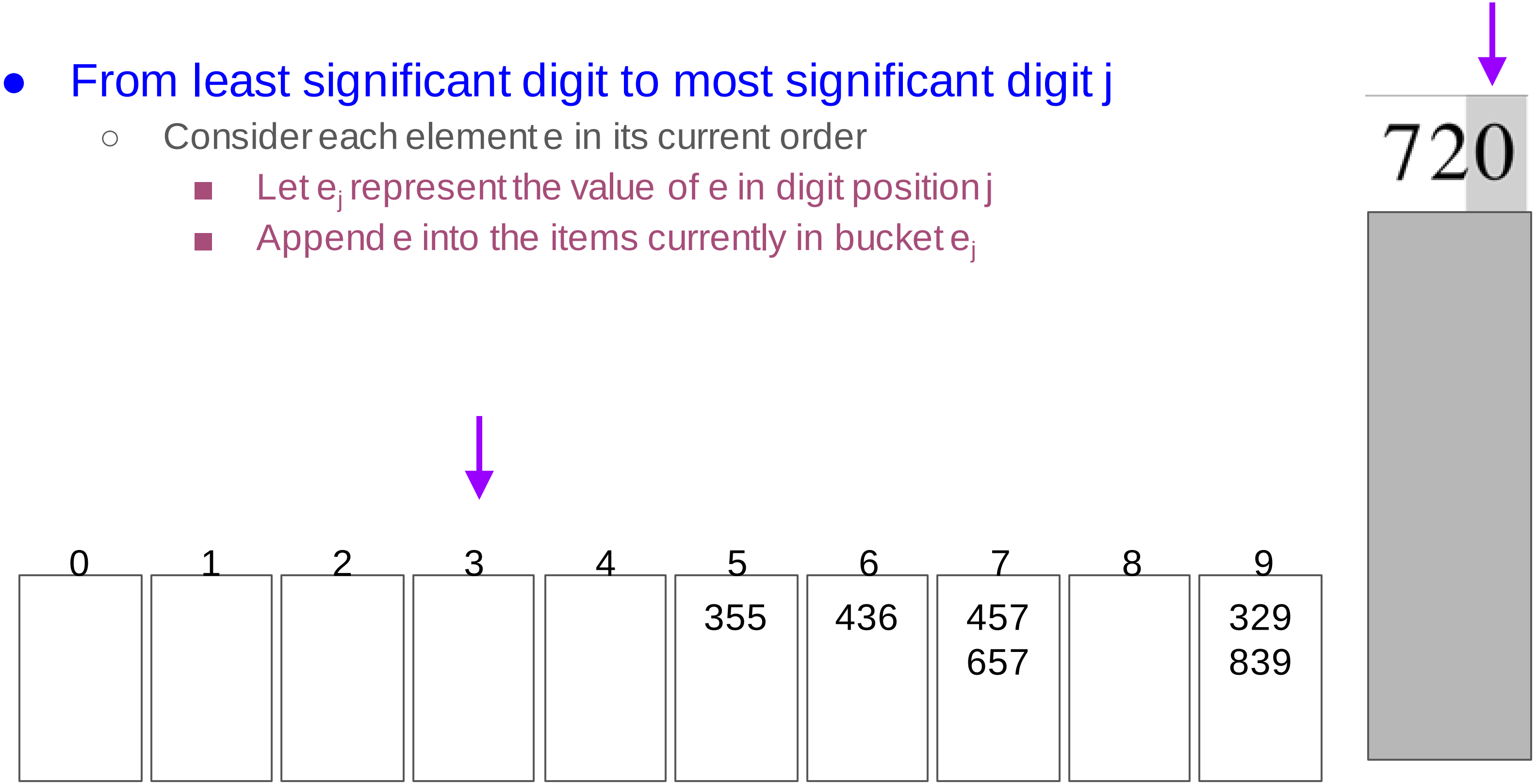
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

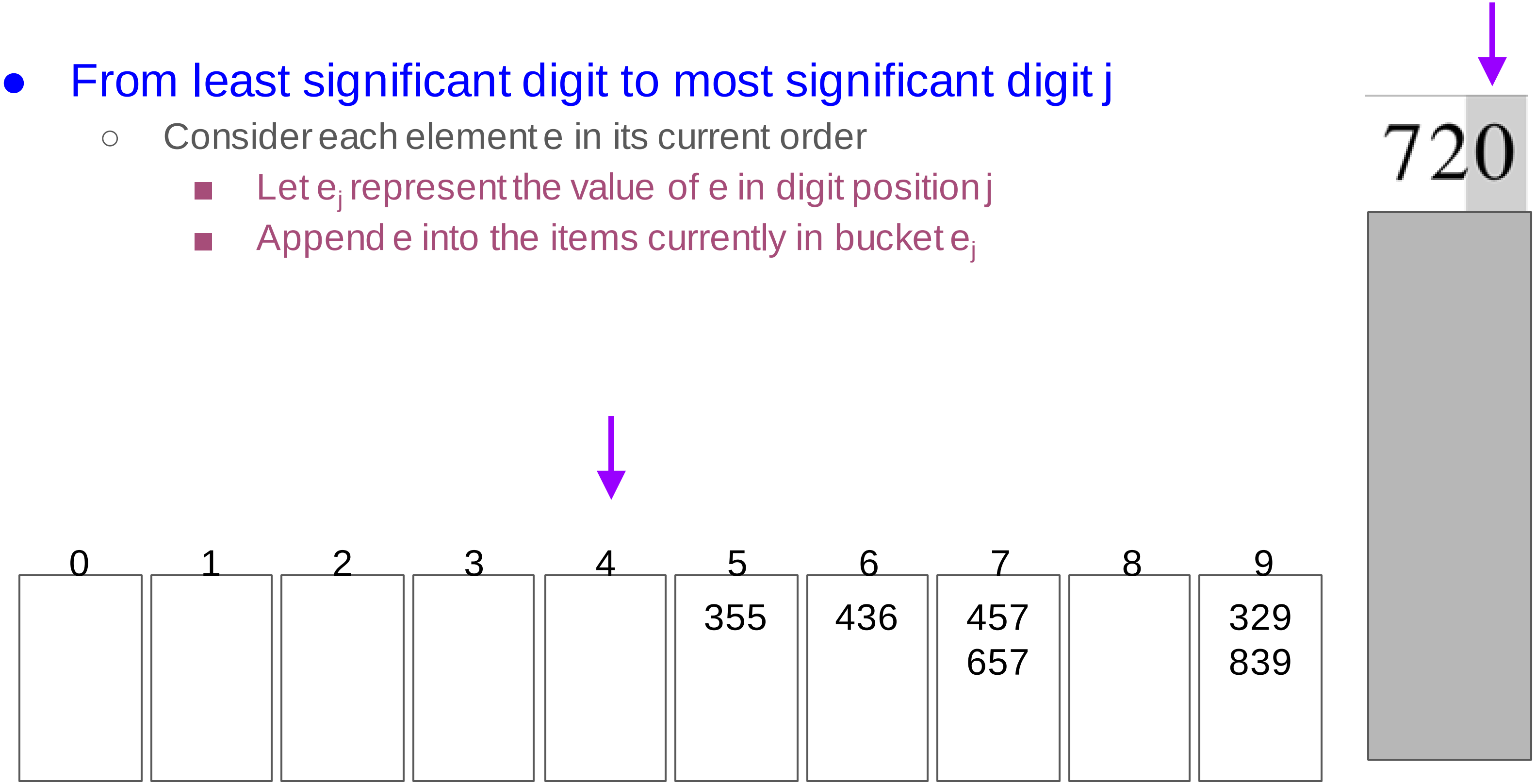


0	1	2	3	4	5	6	7	8	9
					355	436	457 657		329 839

720

Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

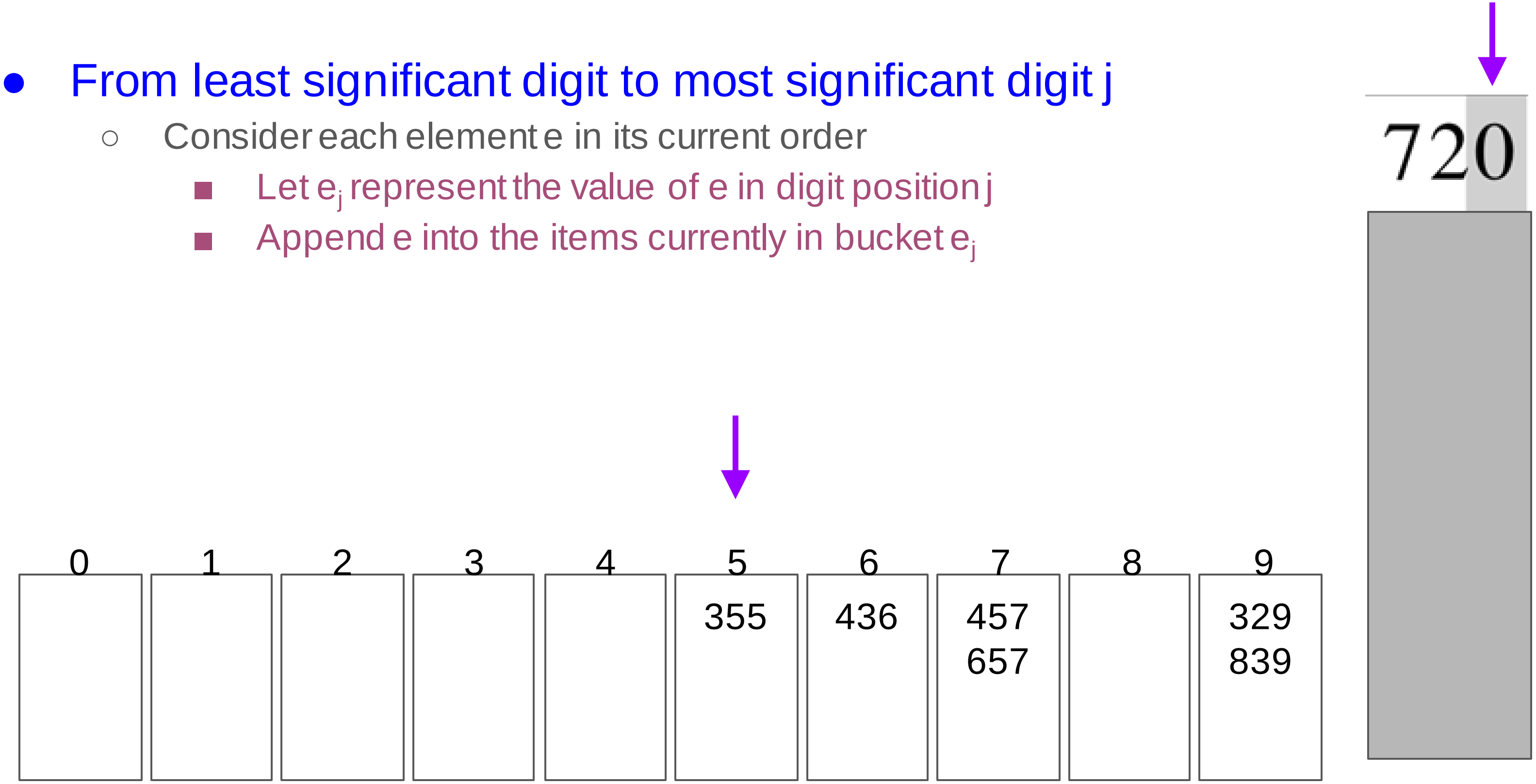


720

0	1	2	3	4	5	6	7	8	9
					355	436	457 657		329 839

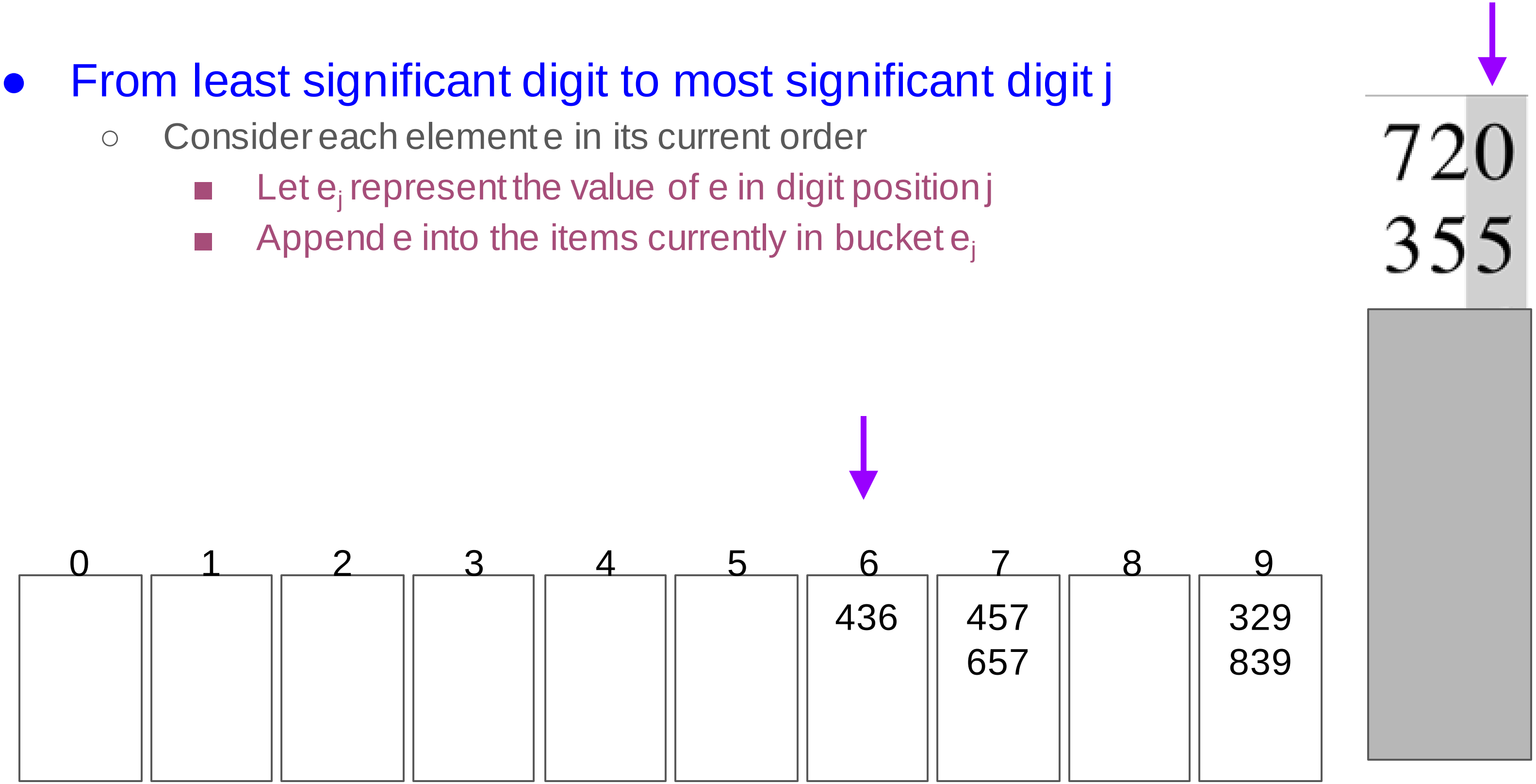
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



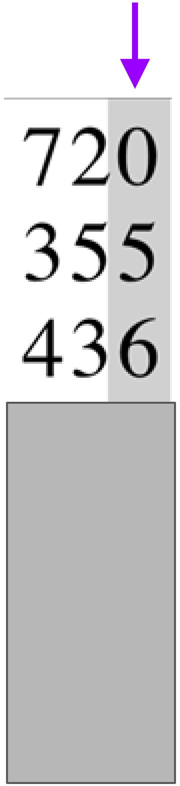
Radix Sort

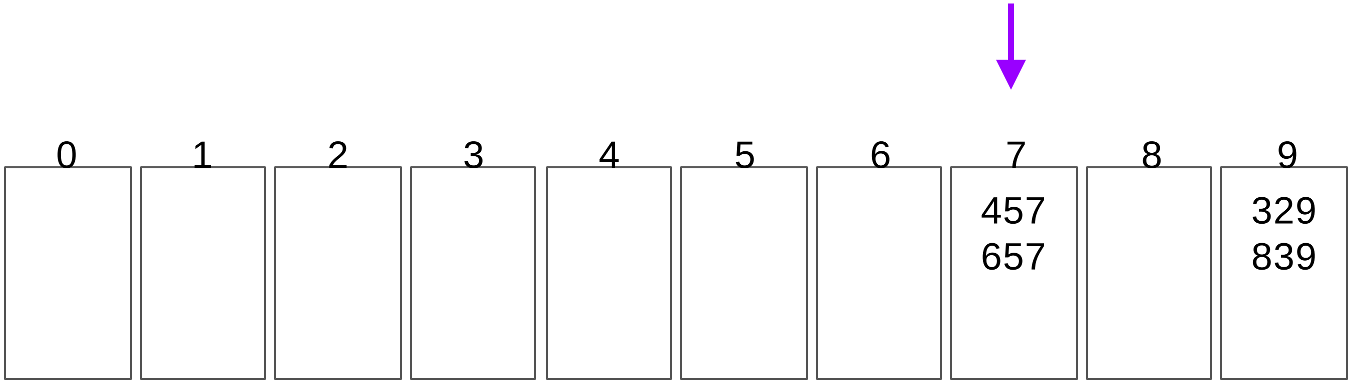
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

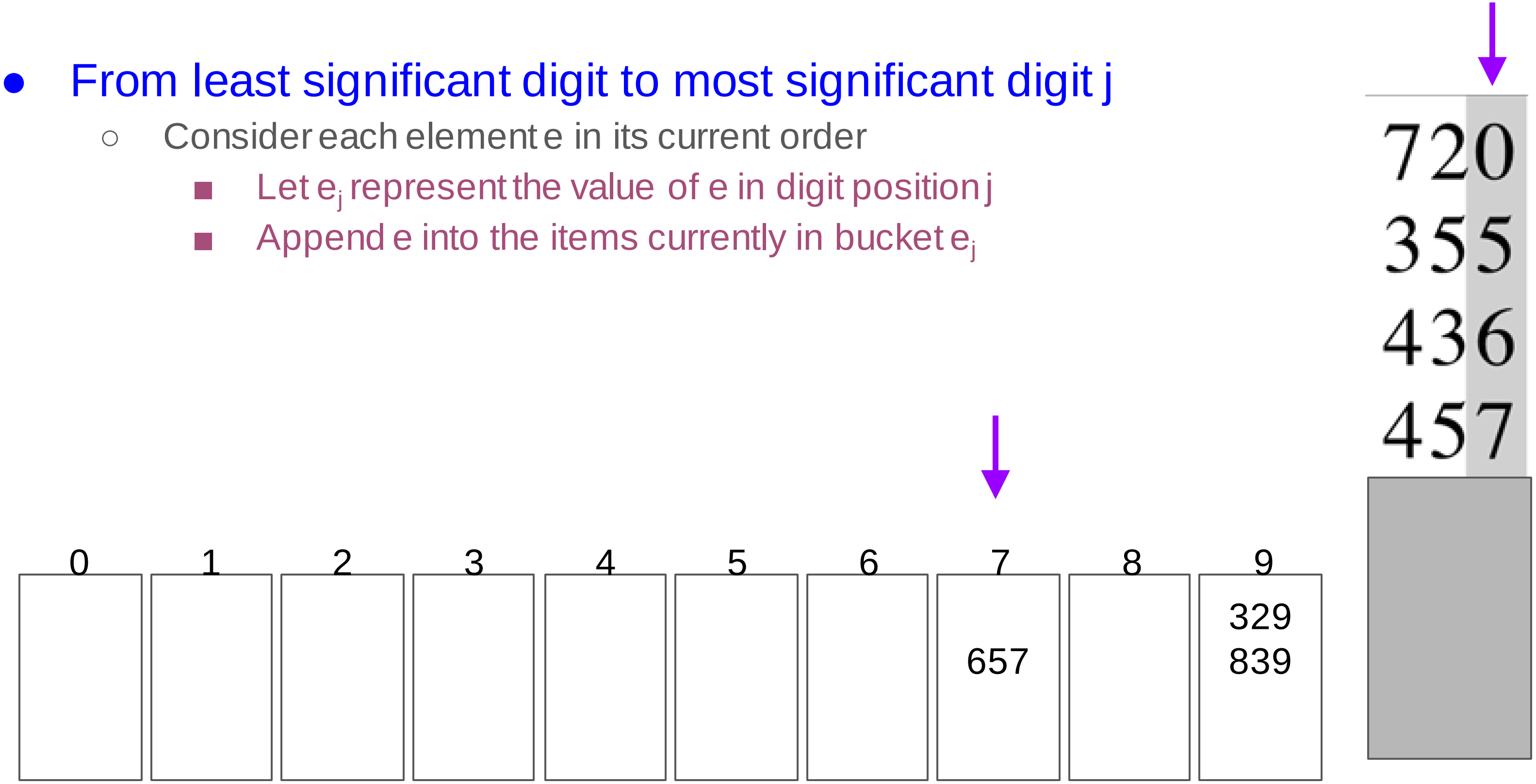
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j





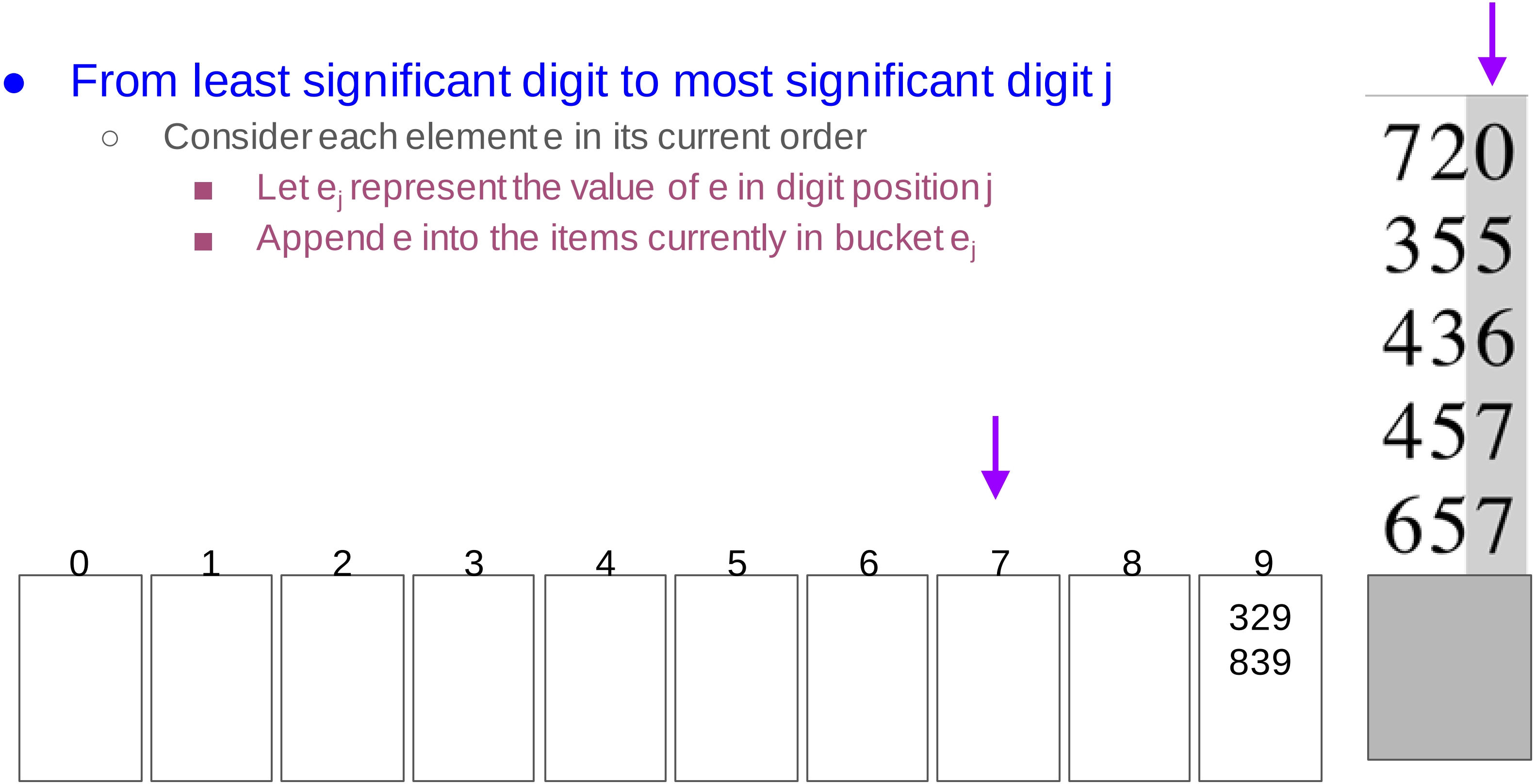
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



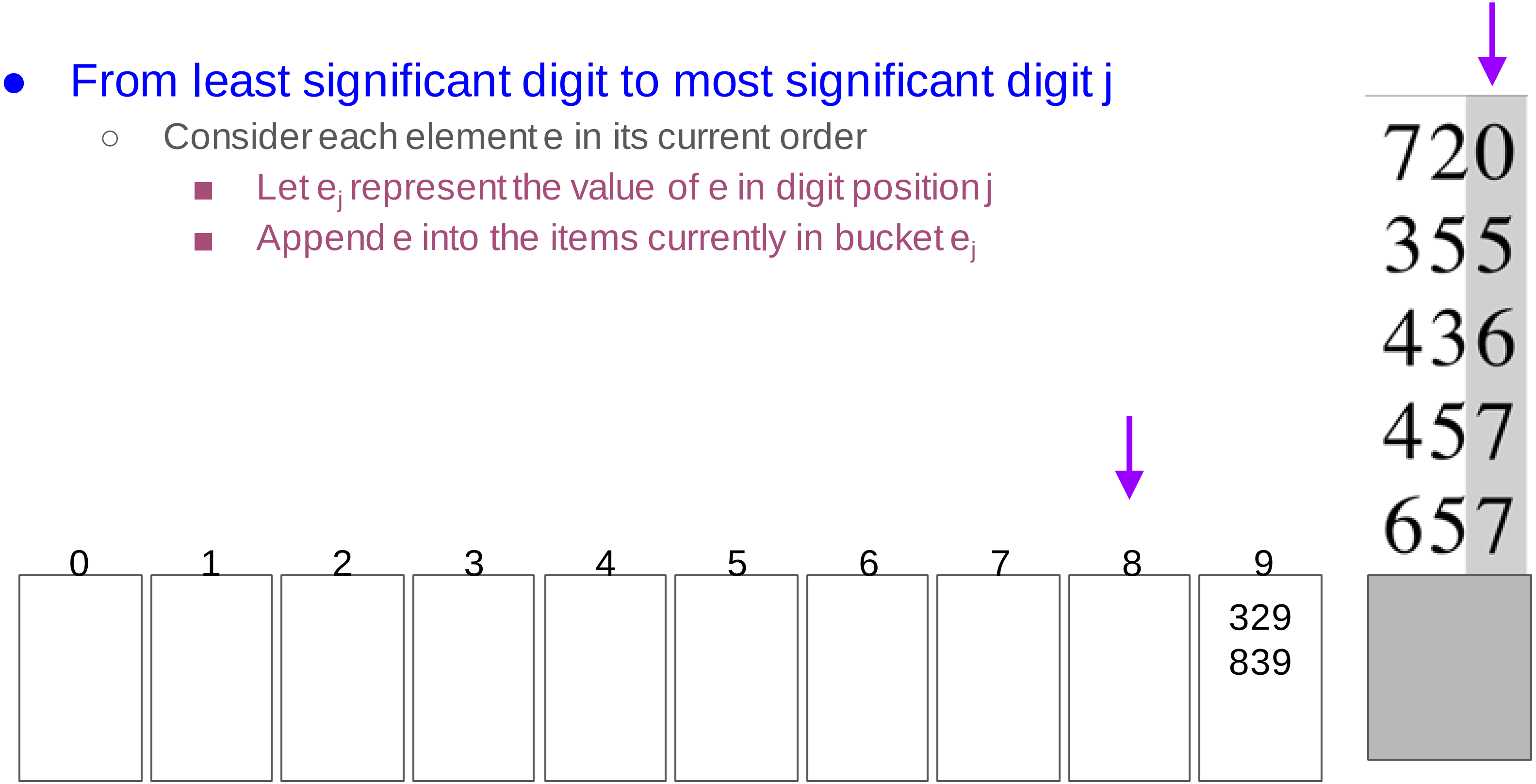
Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



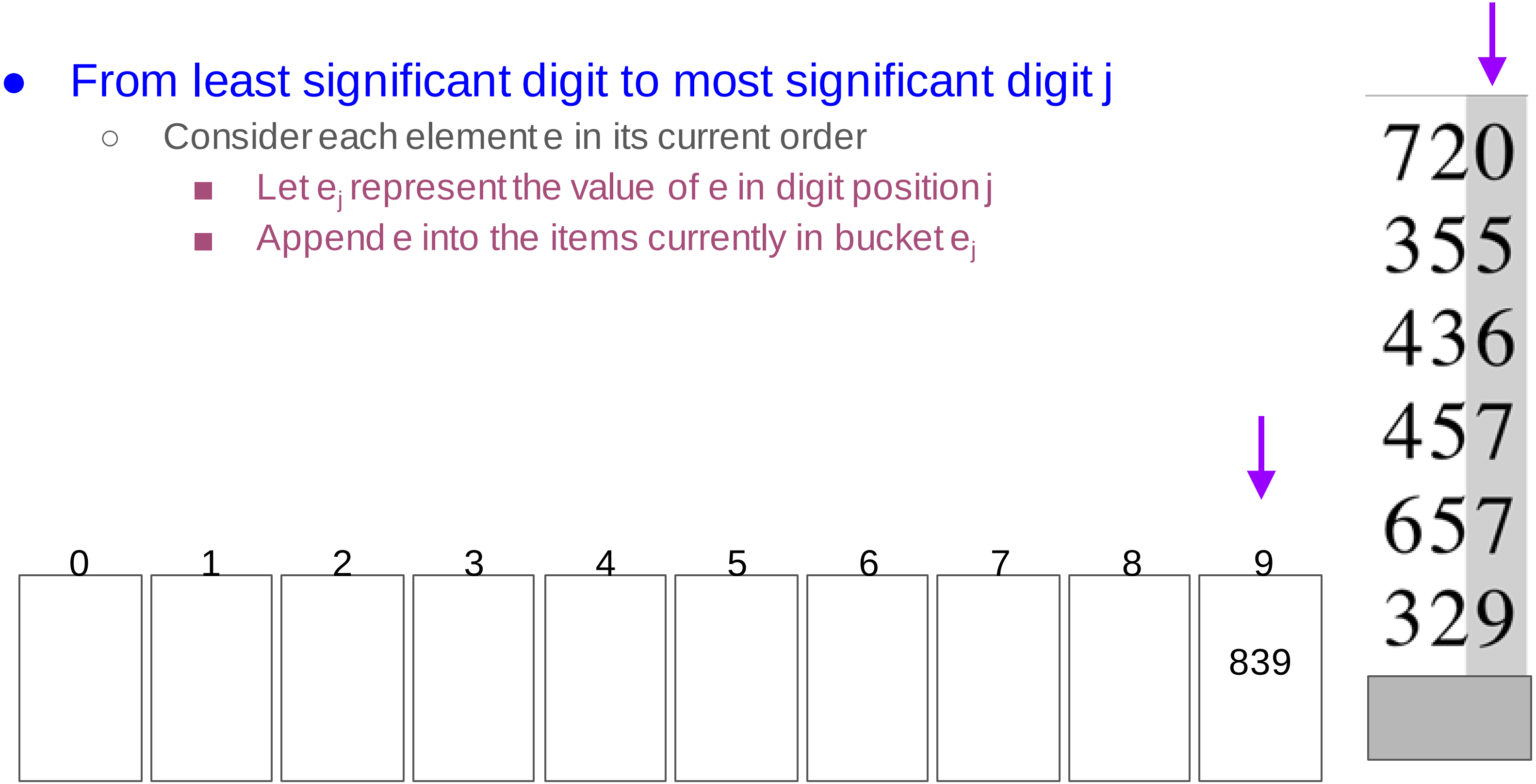
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



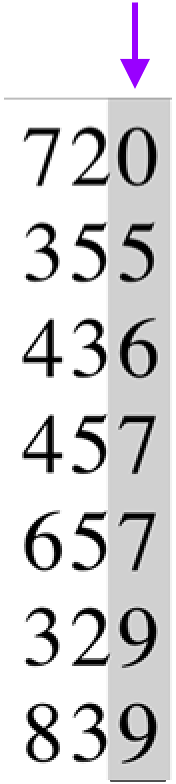
Radix Sort

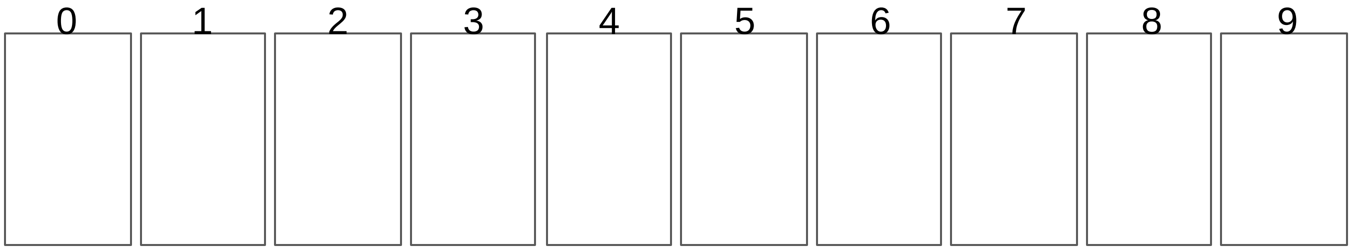
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

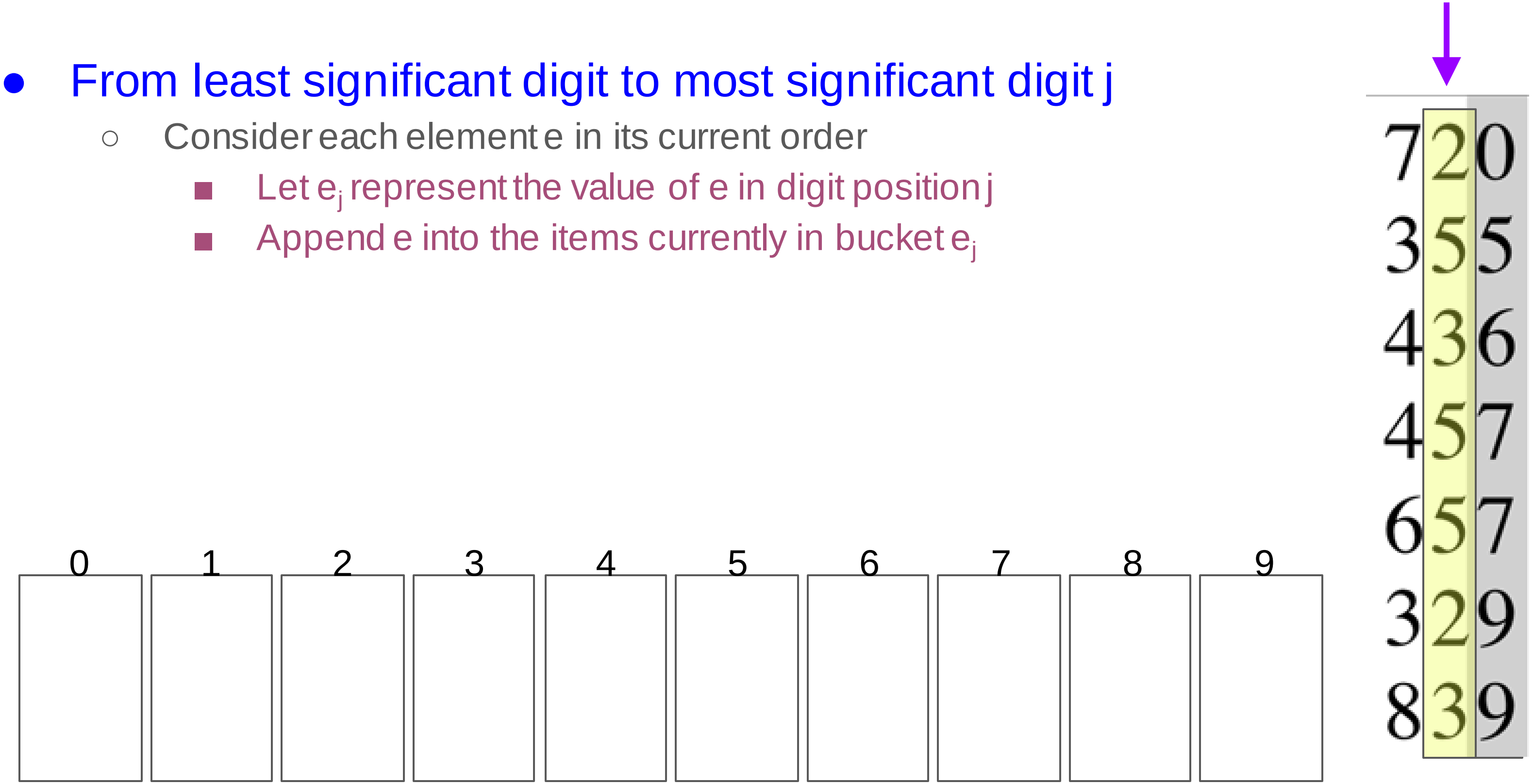
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j





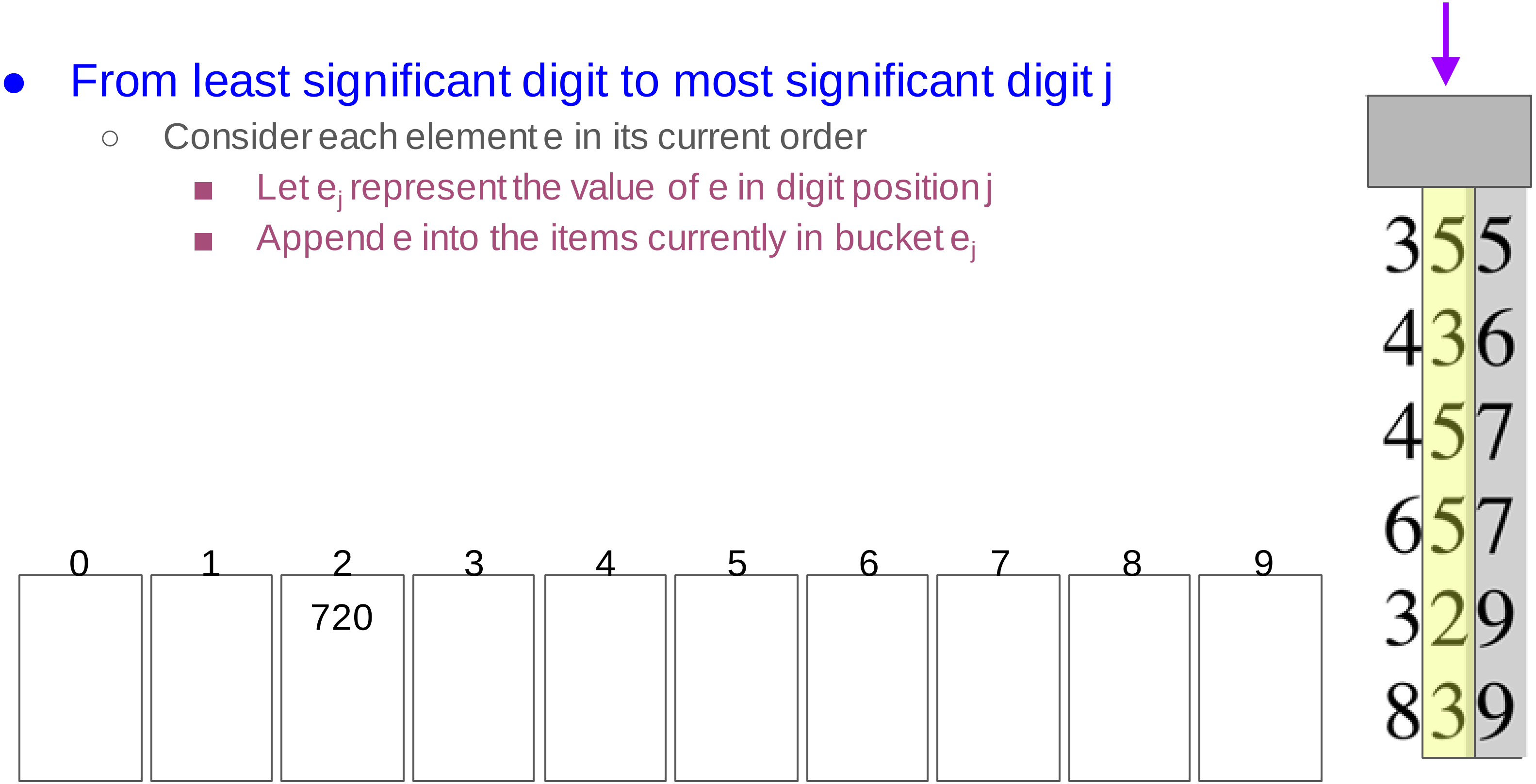
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



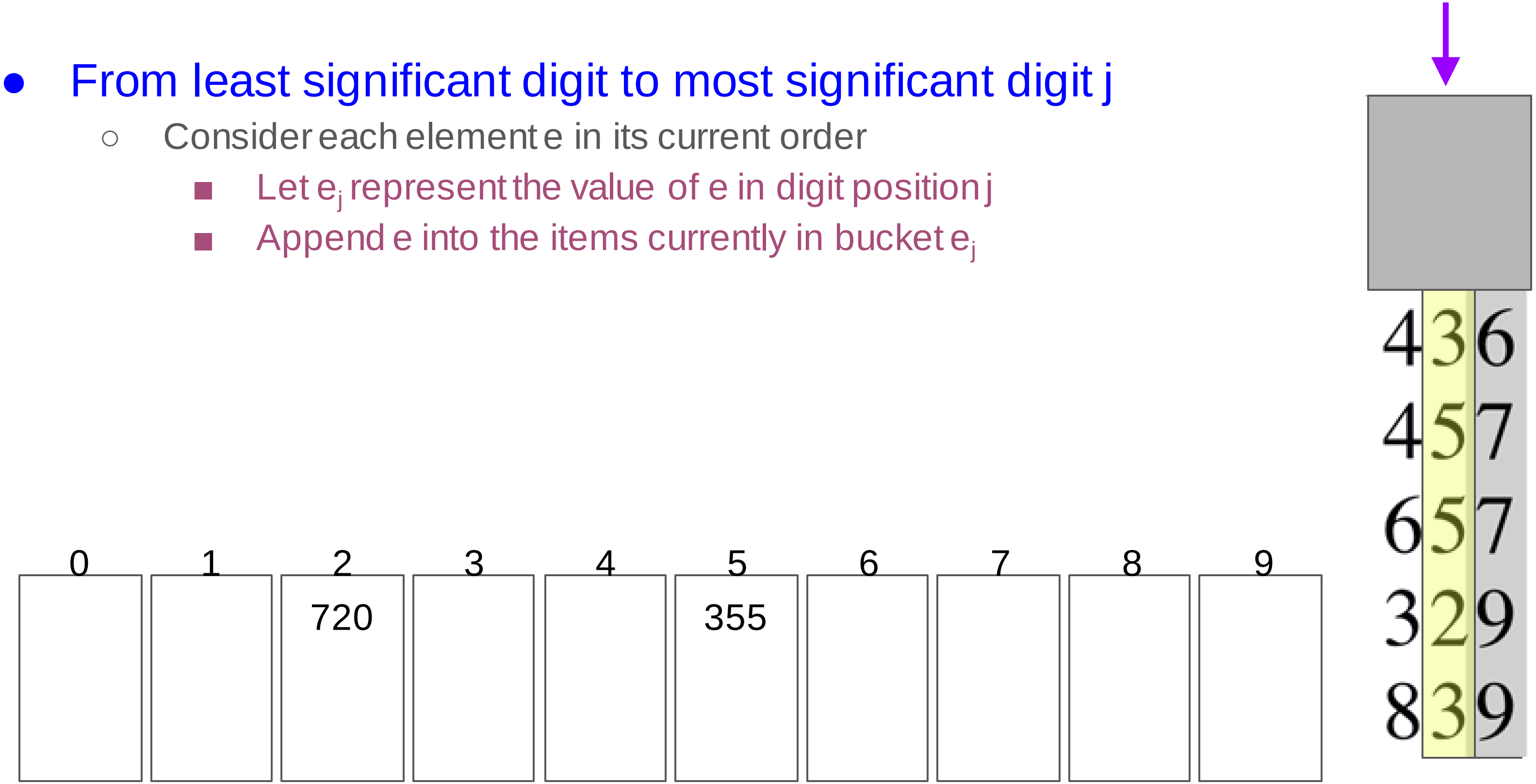
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

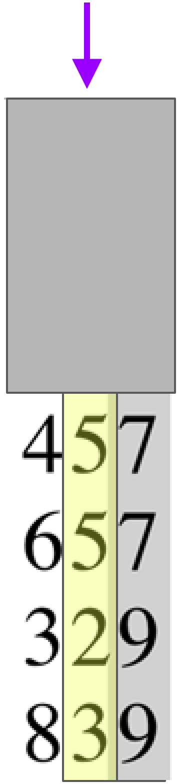
- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



Radix Sort

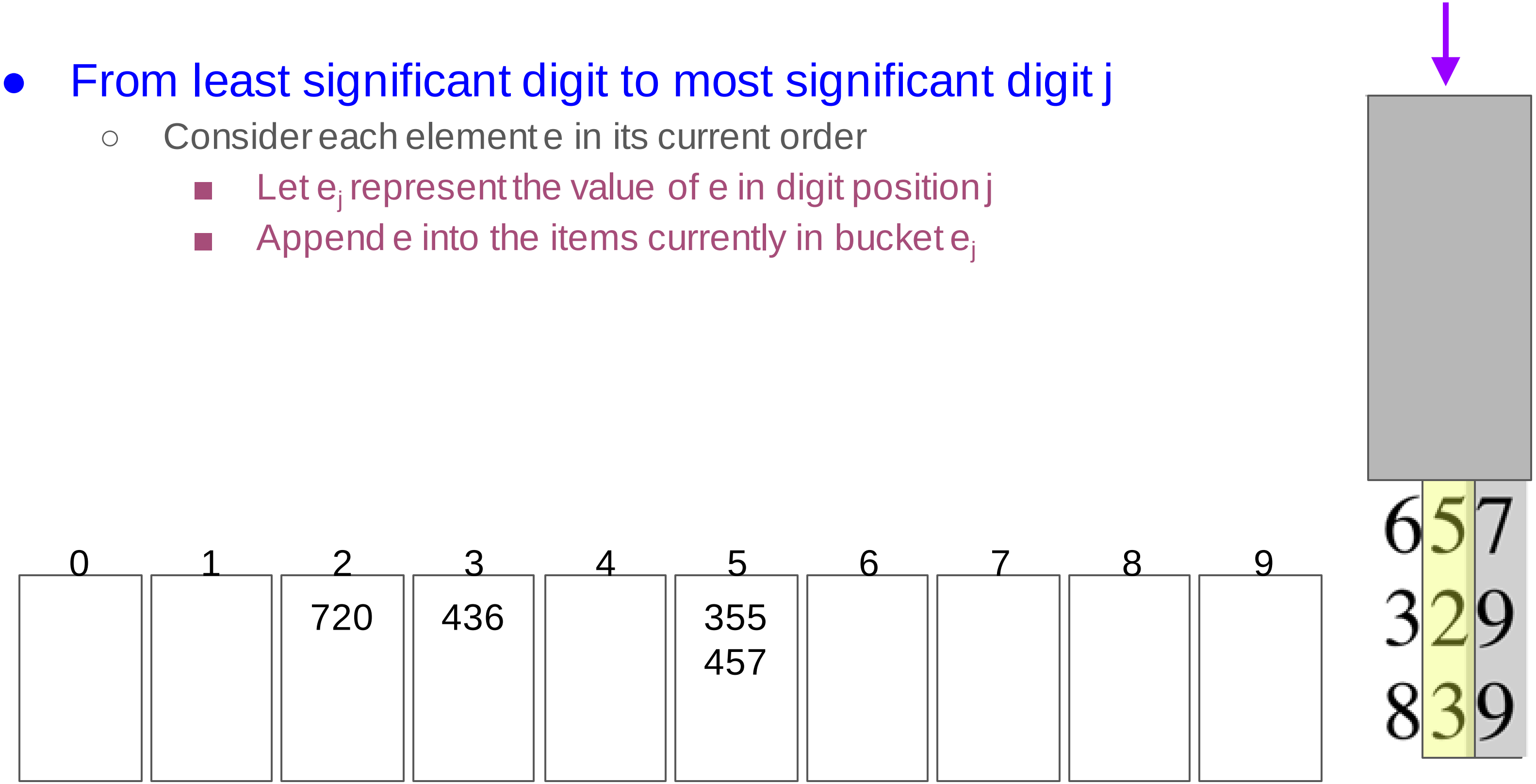
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

0	1	2	3	4	5	6	7	8	9
		720	436		355				



Radix Sort

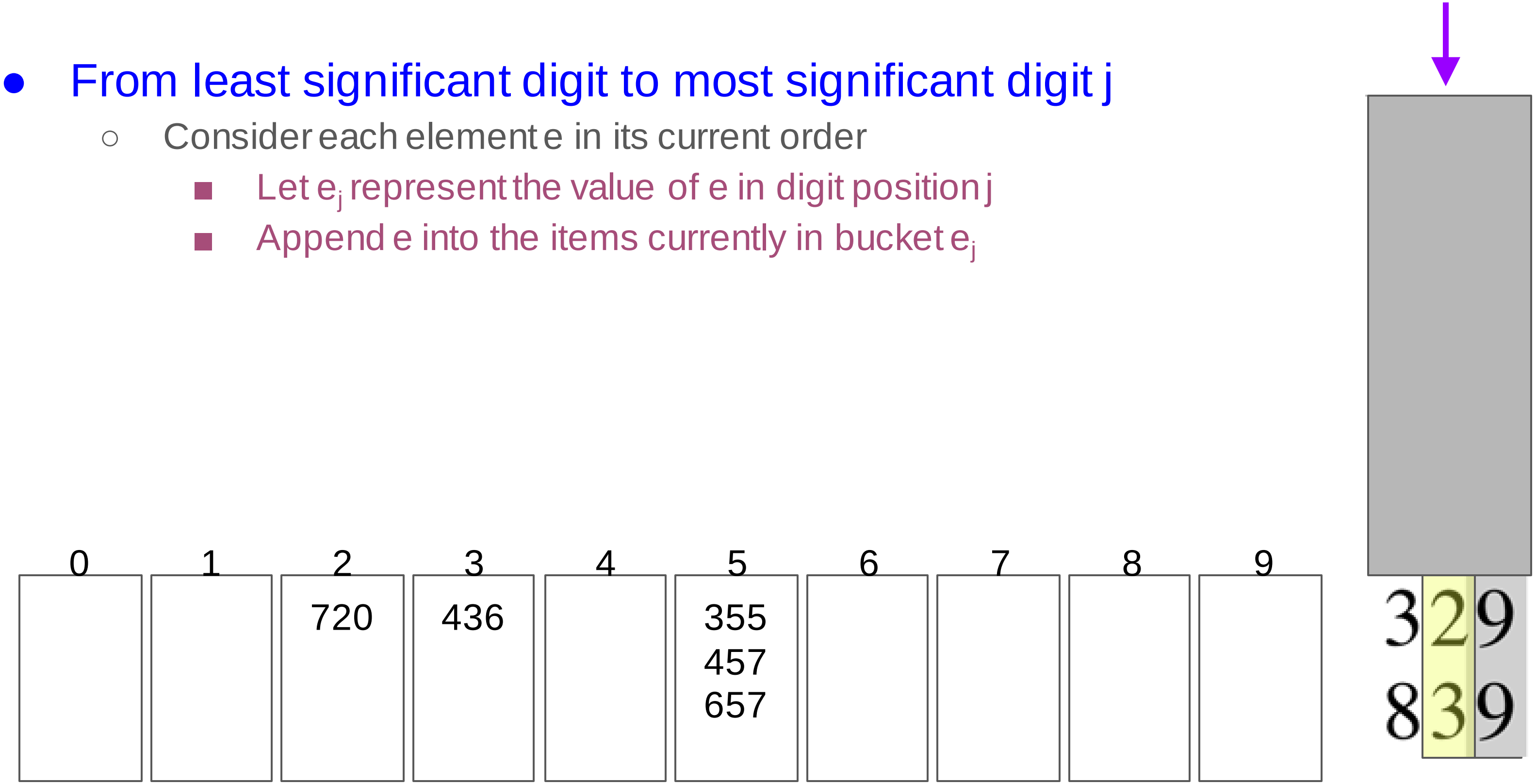
- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



0	1	2	3	4	5	6	7	8	9
		720	436		355 457				

Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

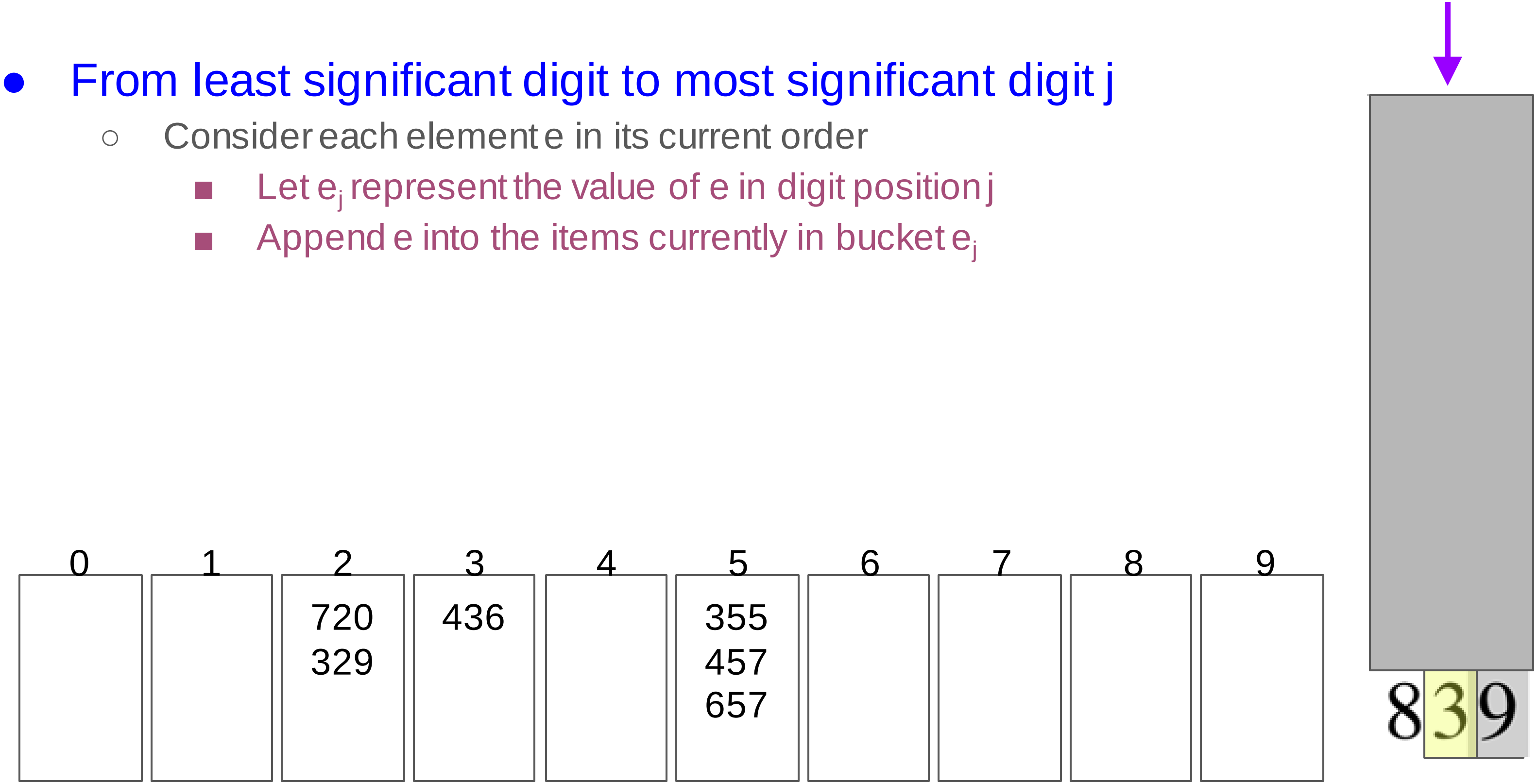


0	1	2	3	4	5	6	7	8	9
		720	436		355 457 657				

329
839

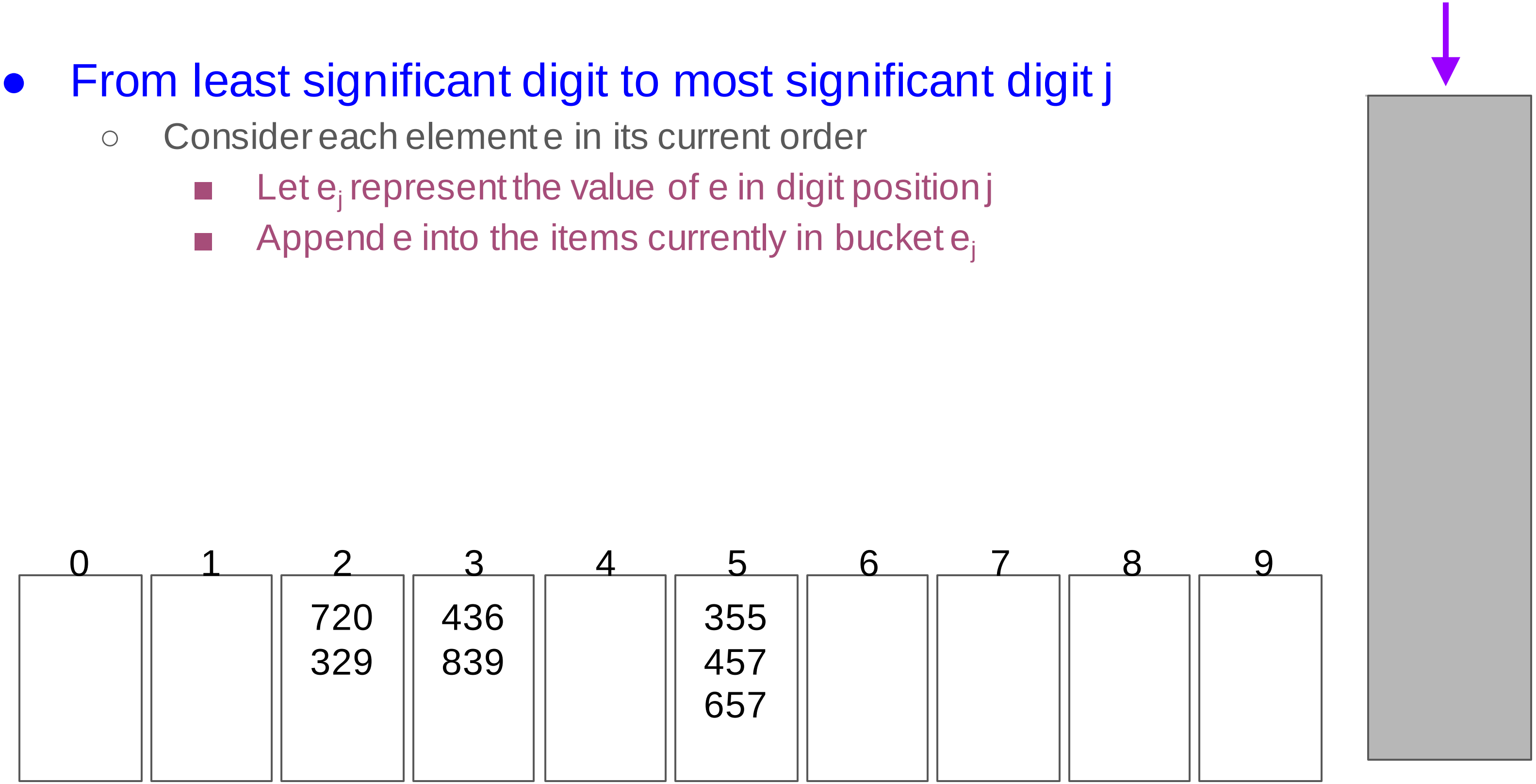
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



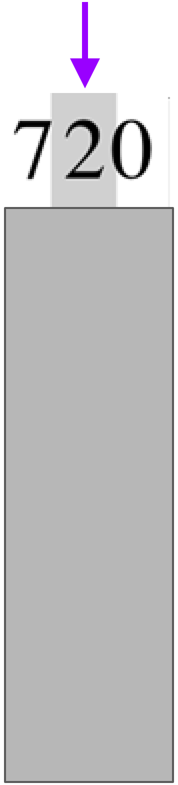
Radix Sort

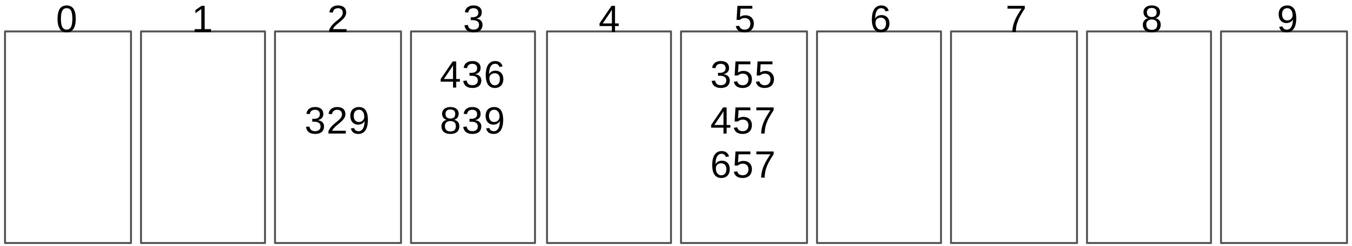
- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

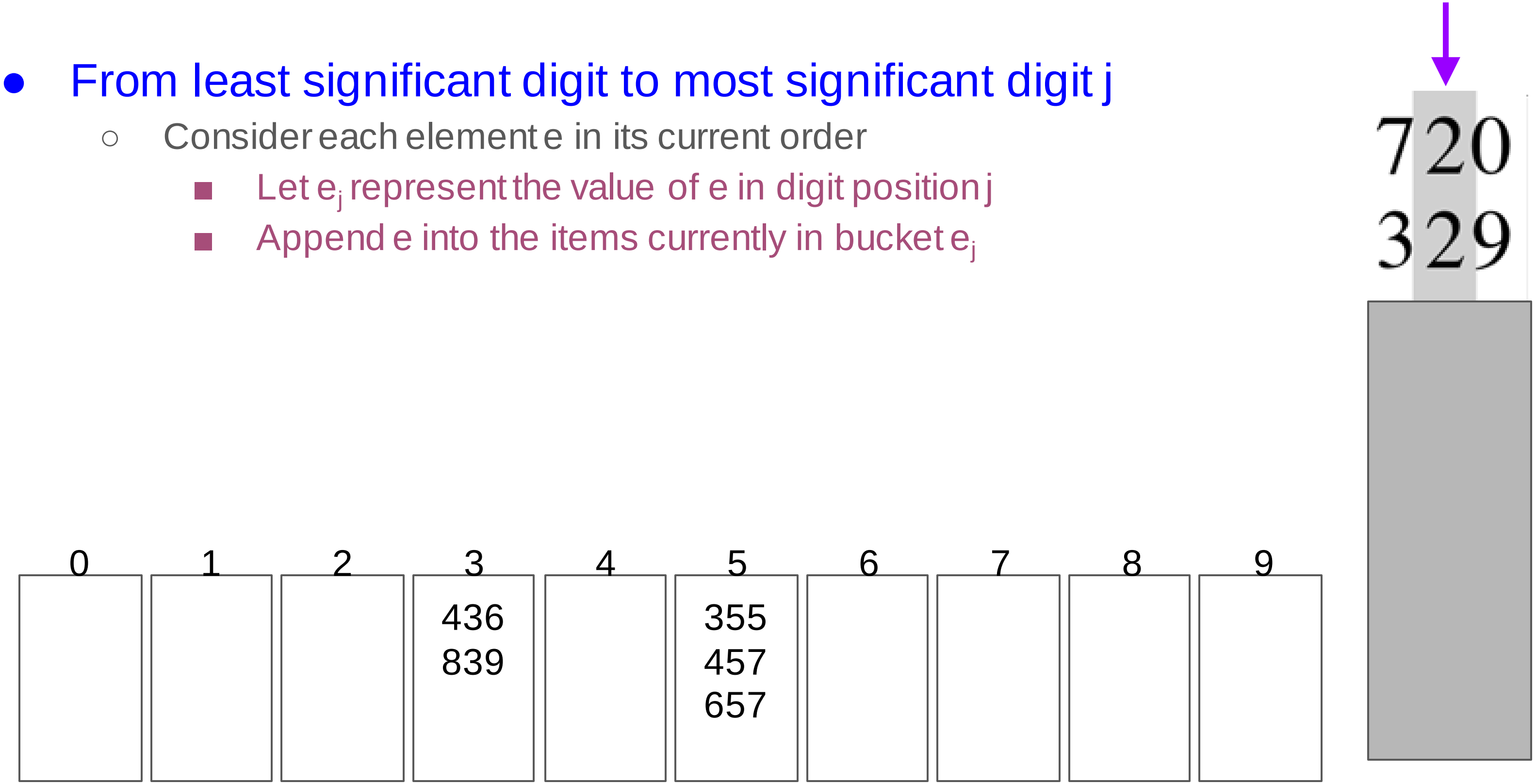




0	1	2	3	4	5	6	7	8	9
		329	436 839		355 457 657				

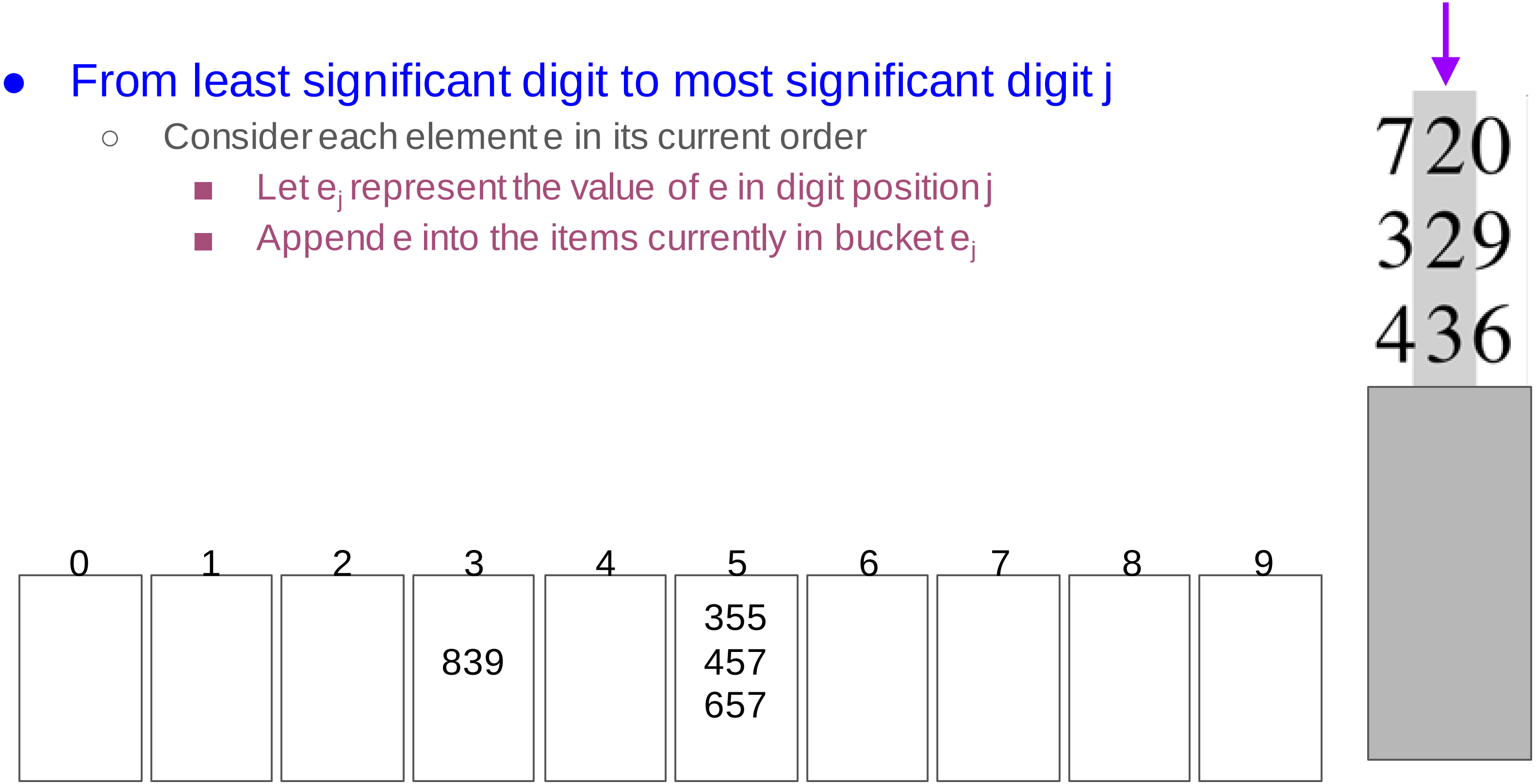
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

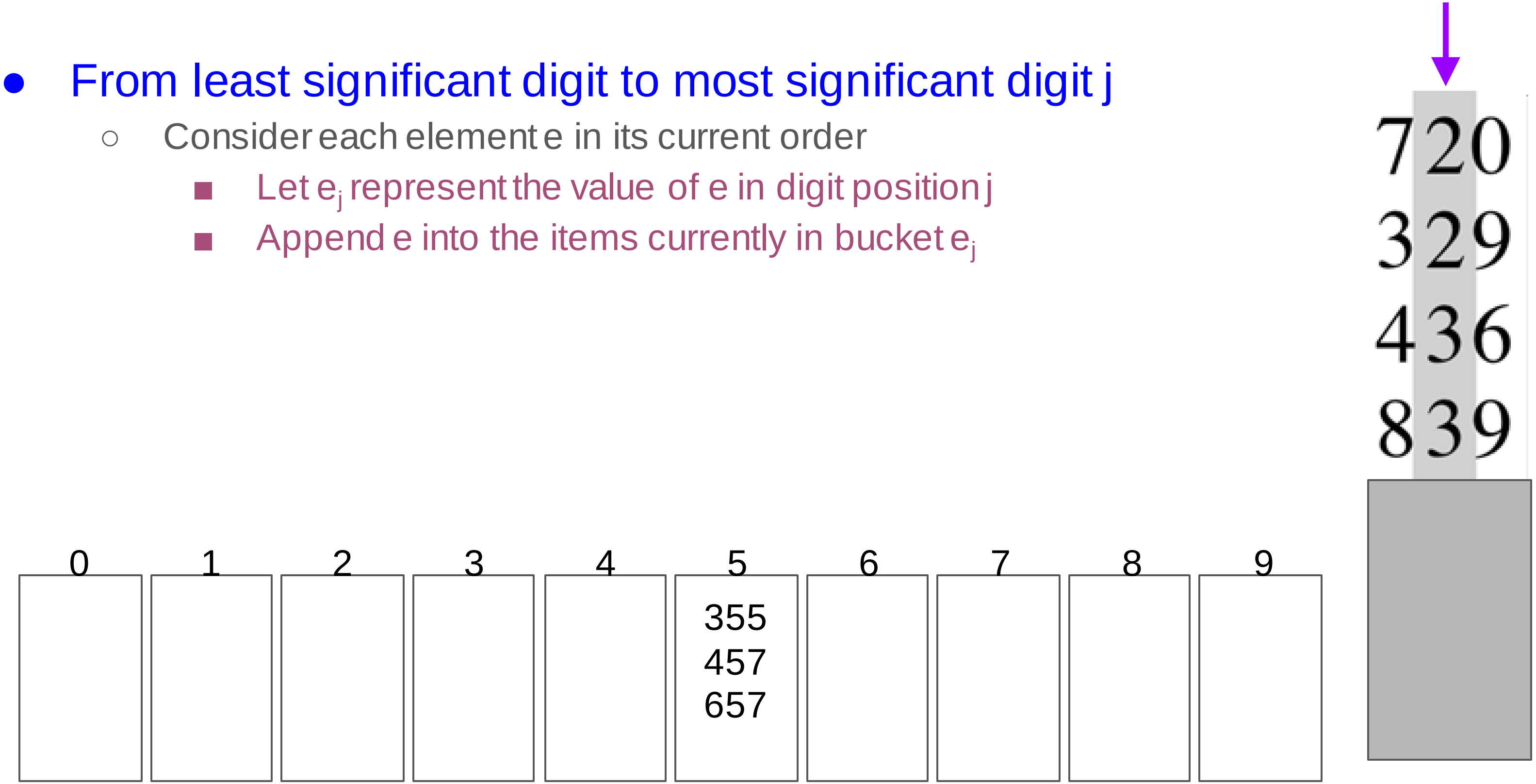


720
329
436

0	1	2	3	4	5	6	7	8	9
			839		355 457 657				

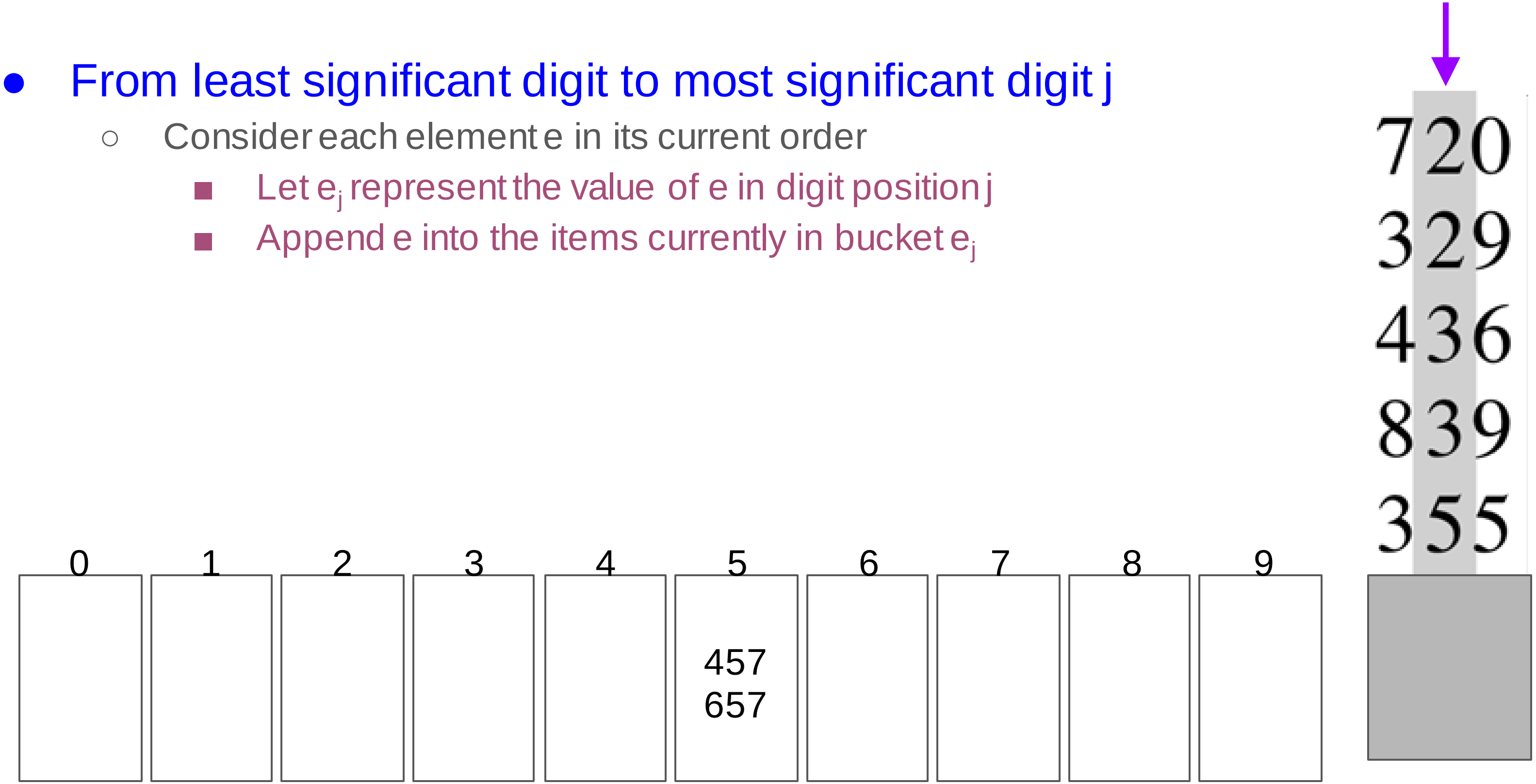
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



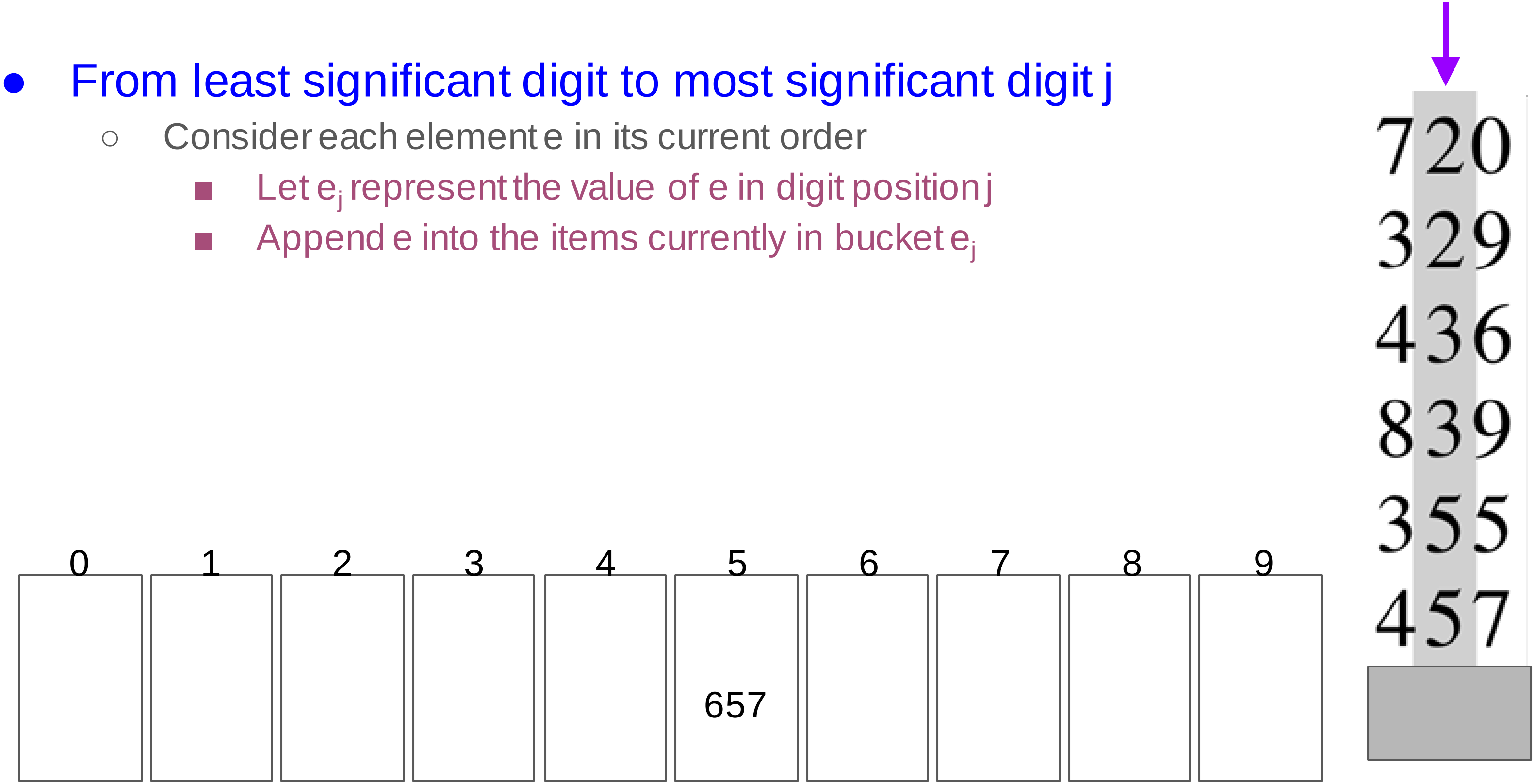
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



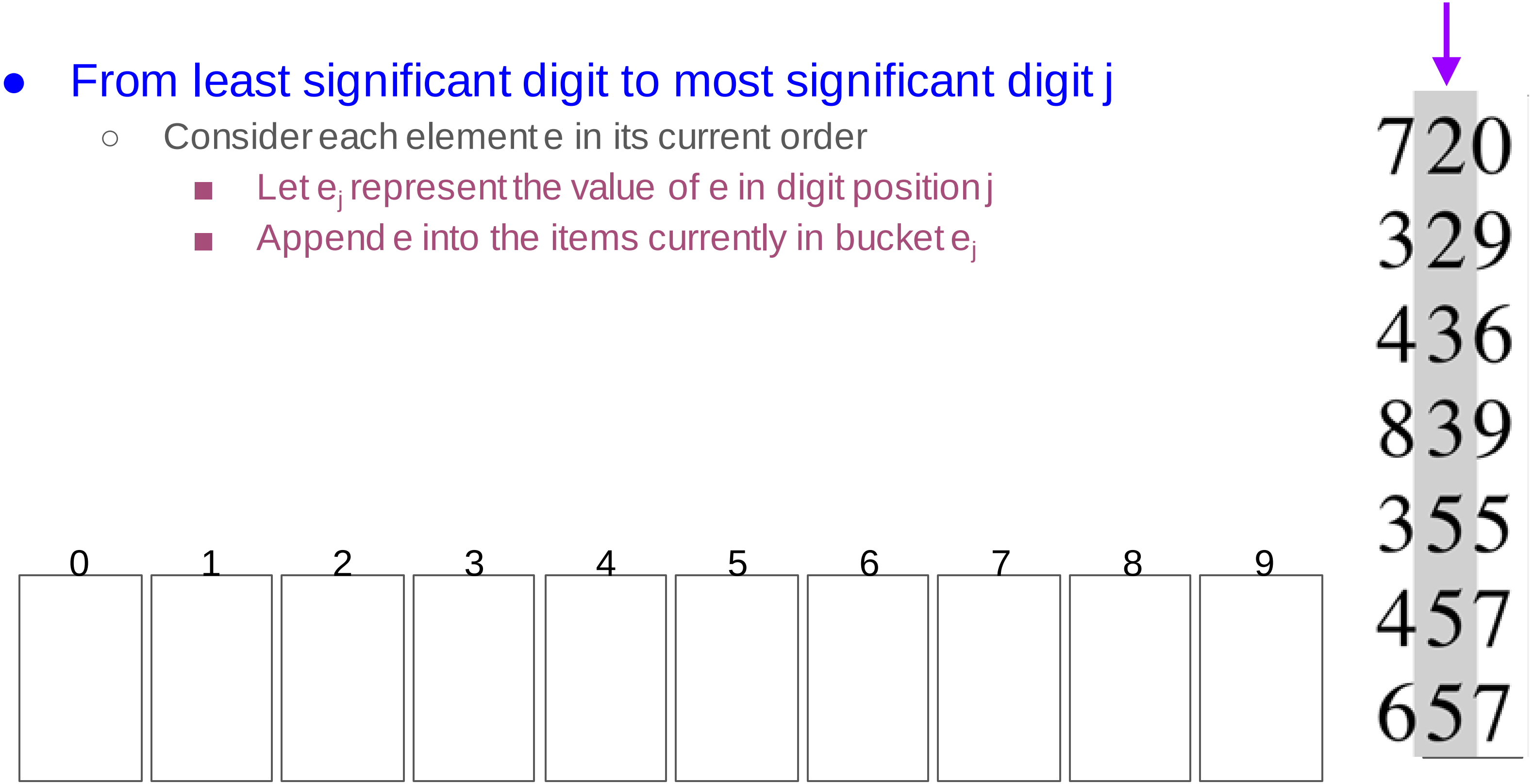
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



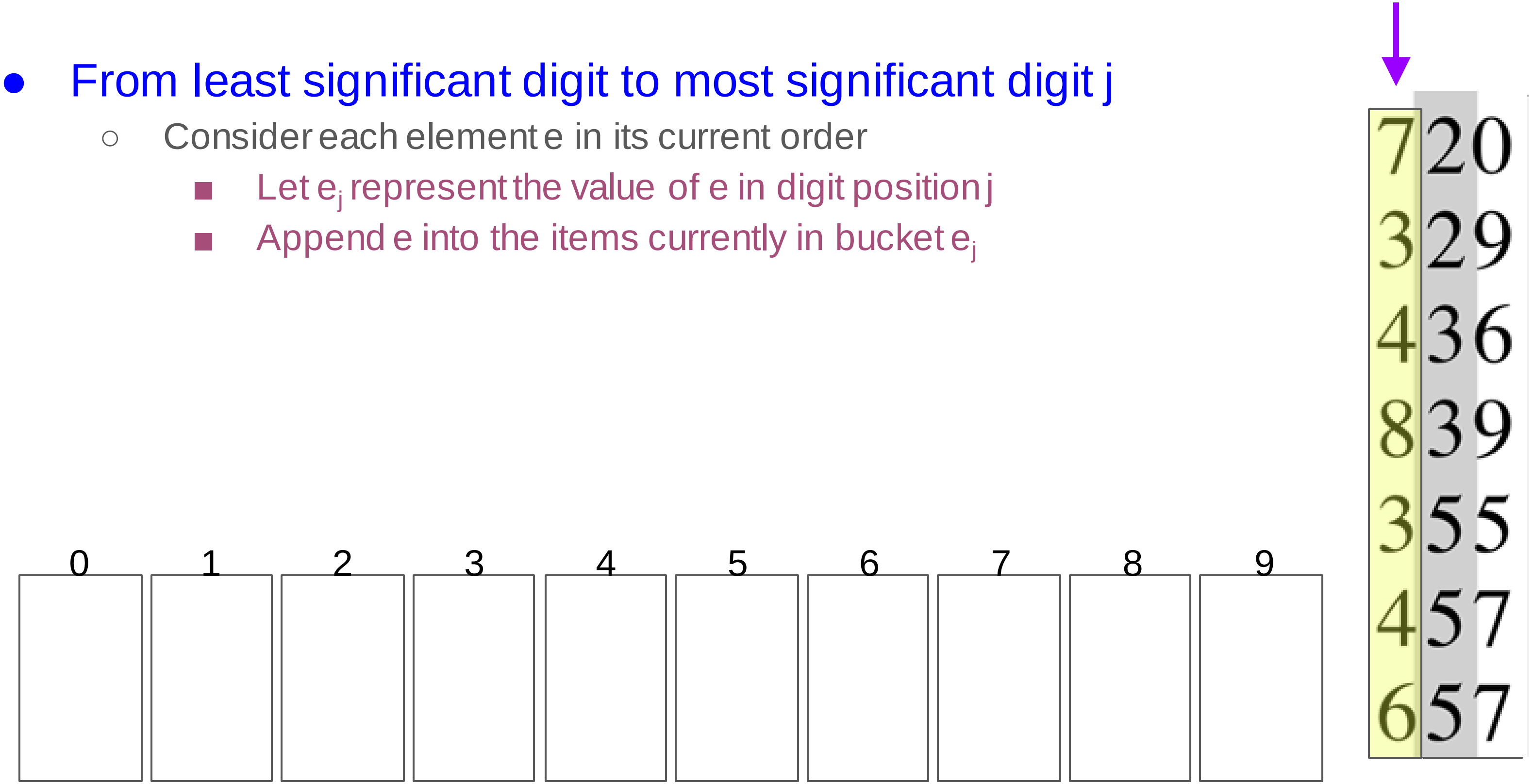
Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



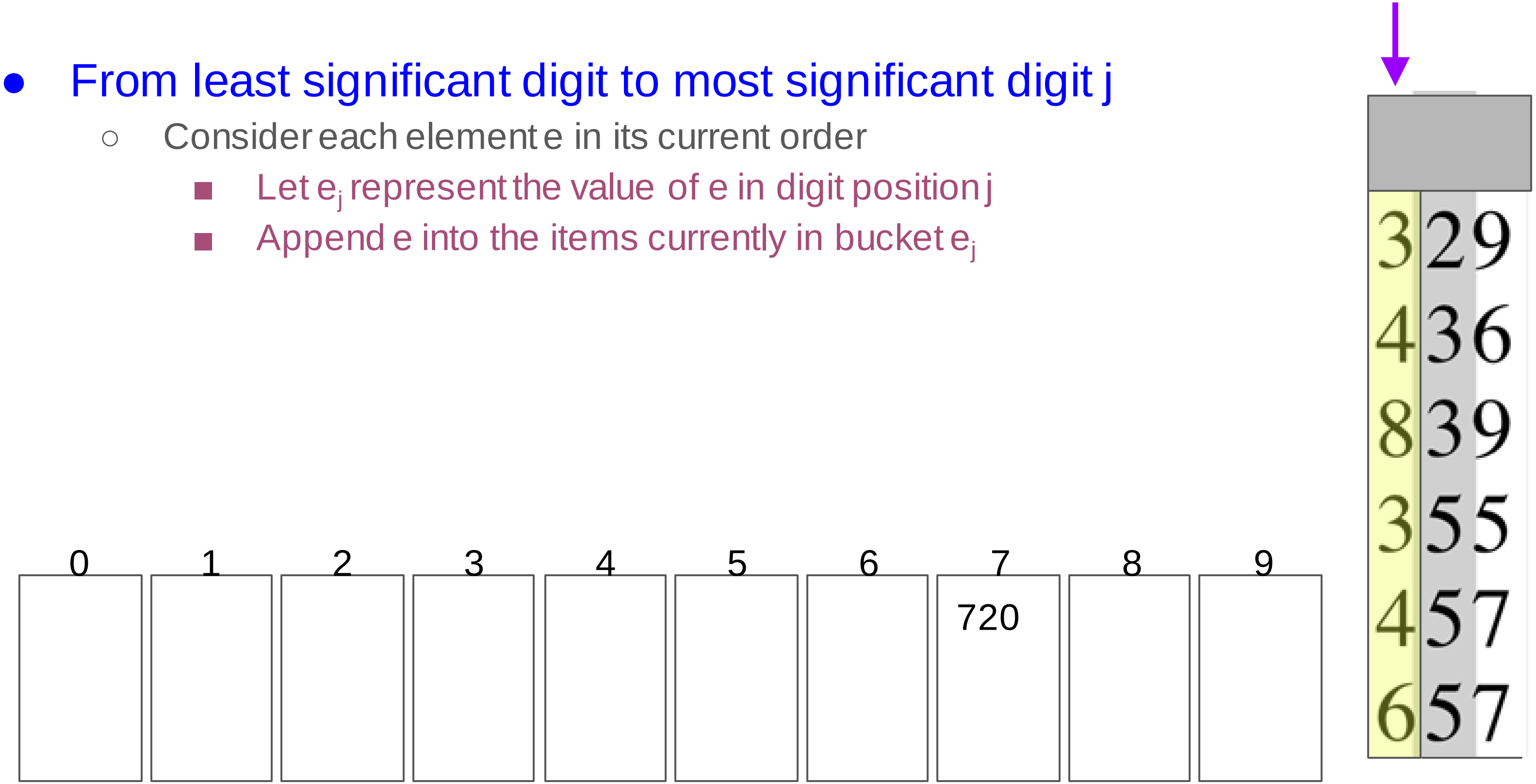
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



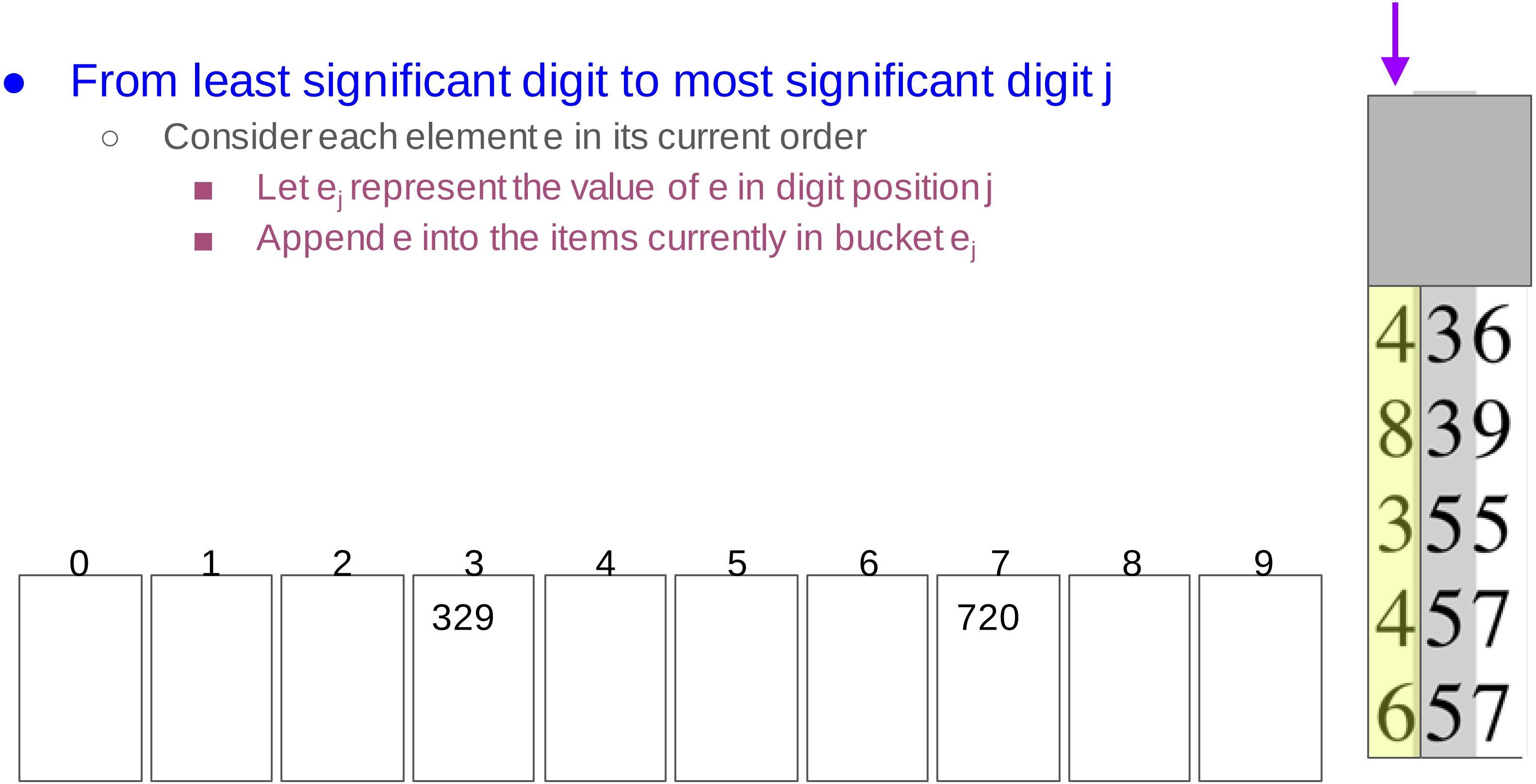
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



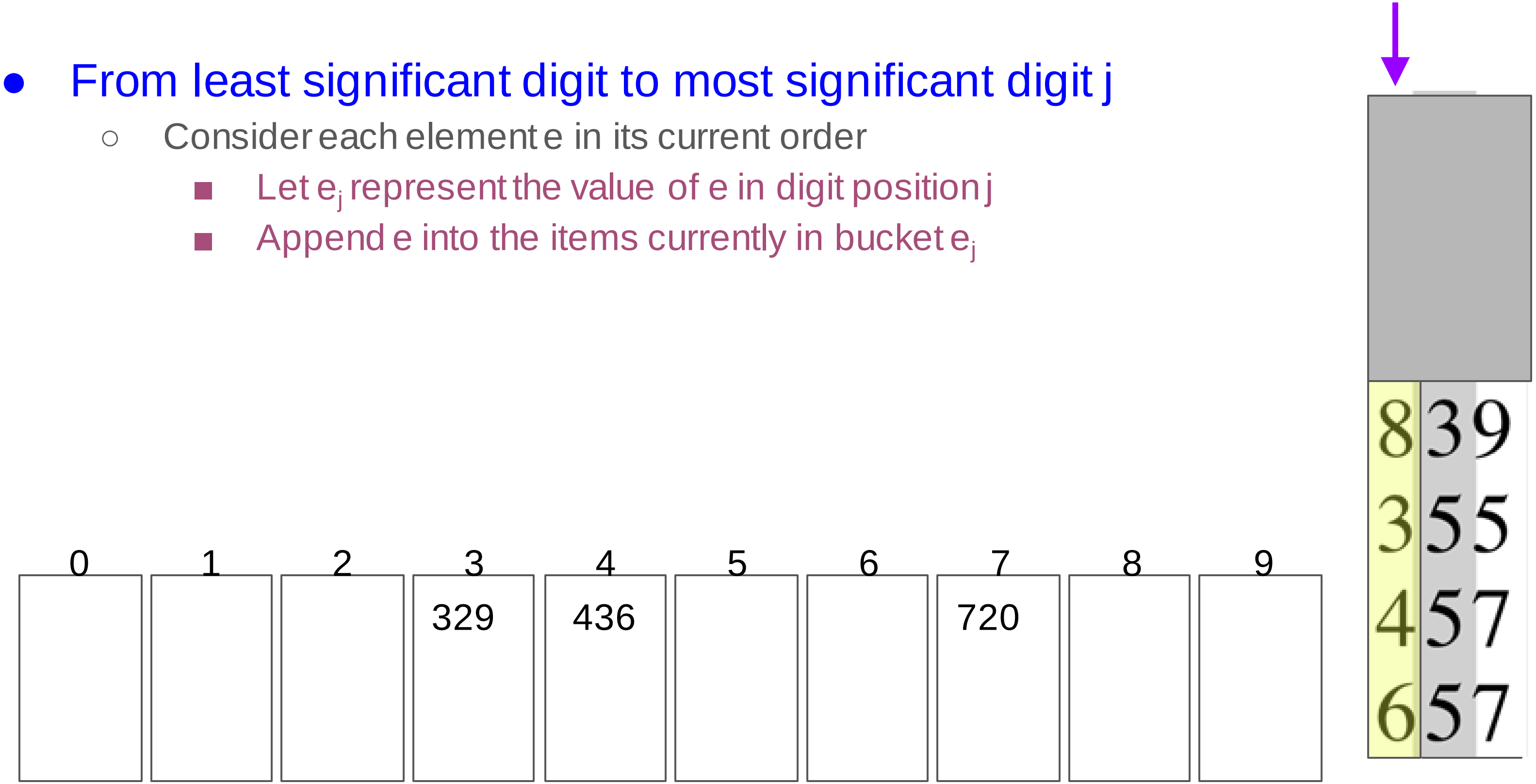
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



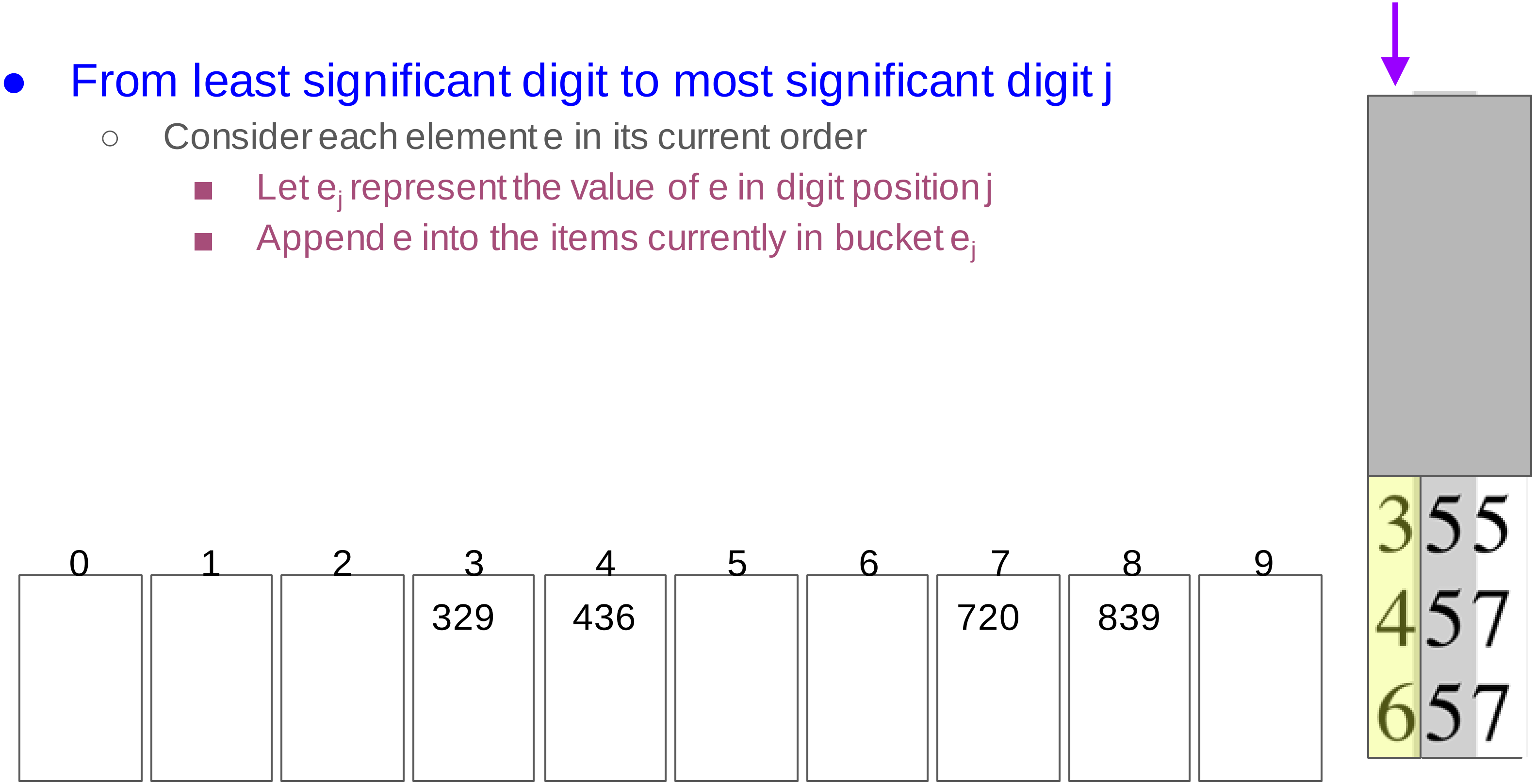
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



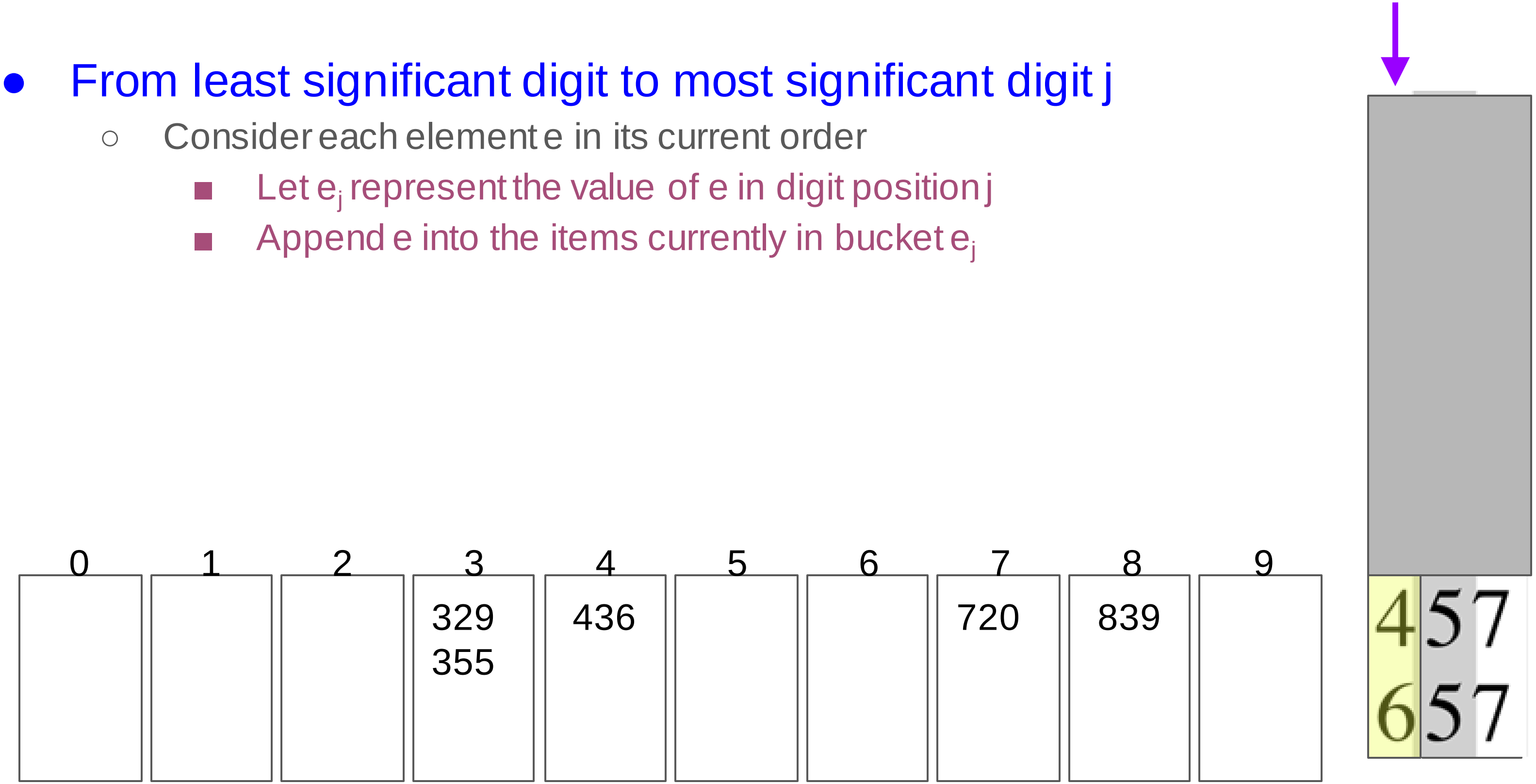
Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

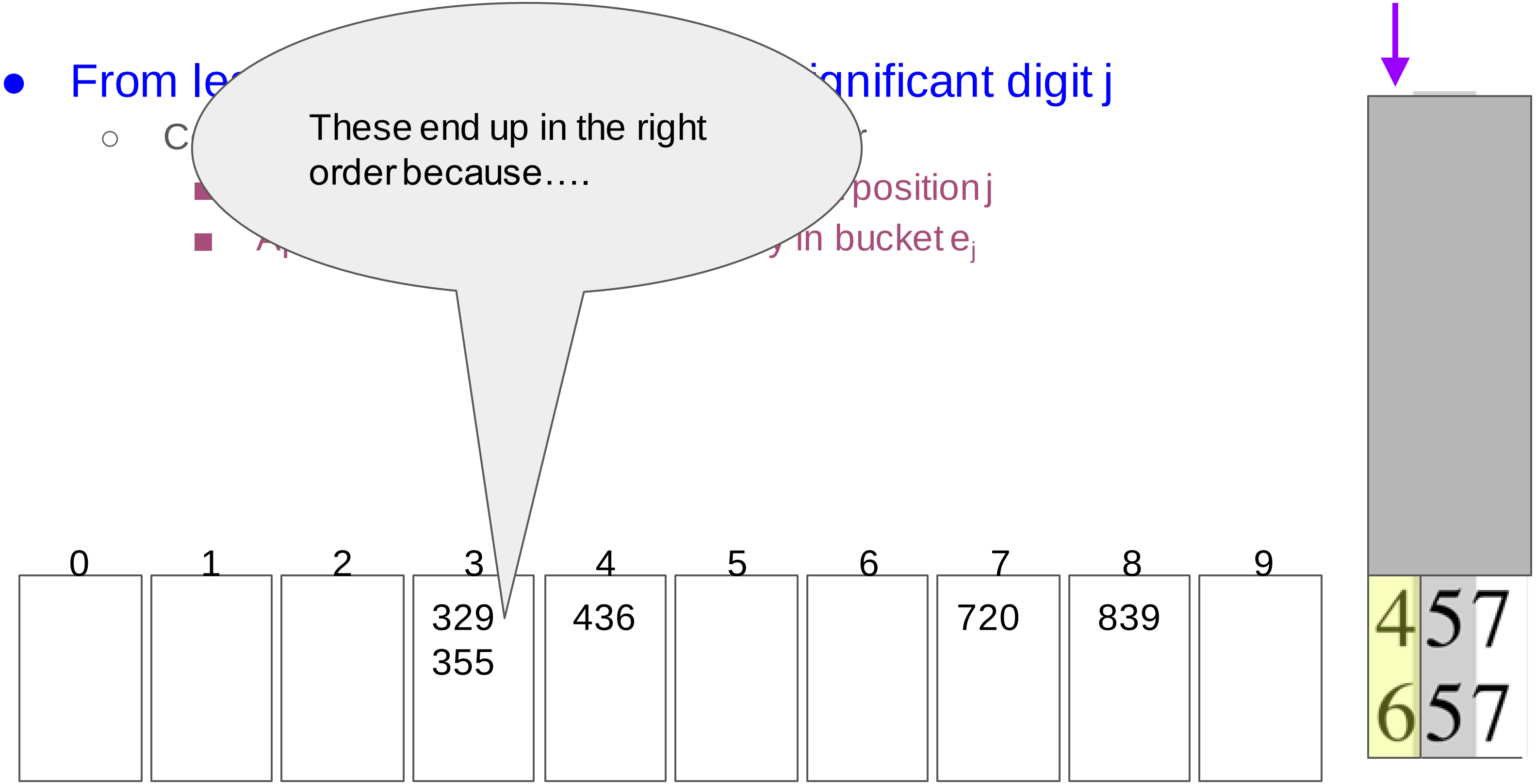


0	1	2	3	4	5	6	7	8	9
			329 355	436			720	839	

Radix Sort

- From le ignificant digit j
  - C r
    - position j
    - A y in bucket e_j

These end up in the right order because….

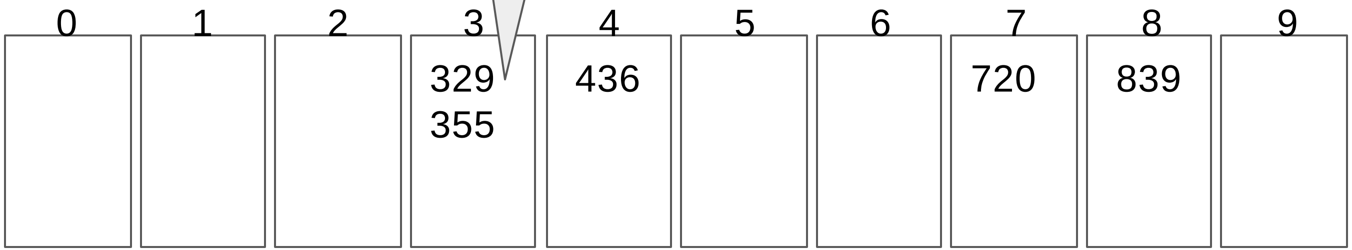


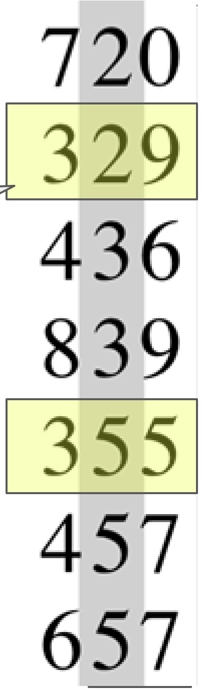
Radix Sort

- From le ignificant digit j
  - C r
    - position j
    - A y in b

These end up in the right order because….

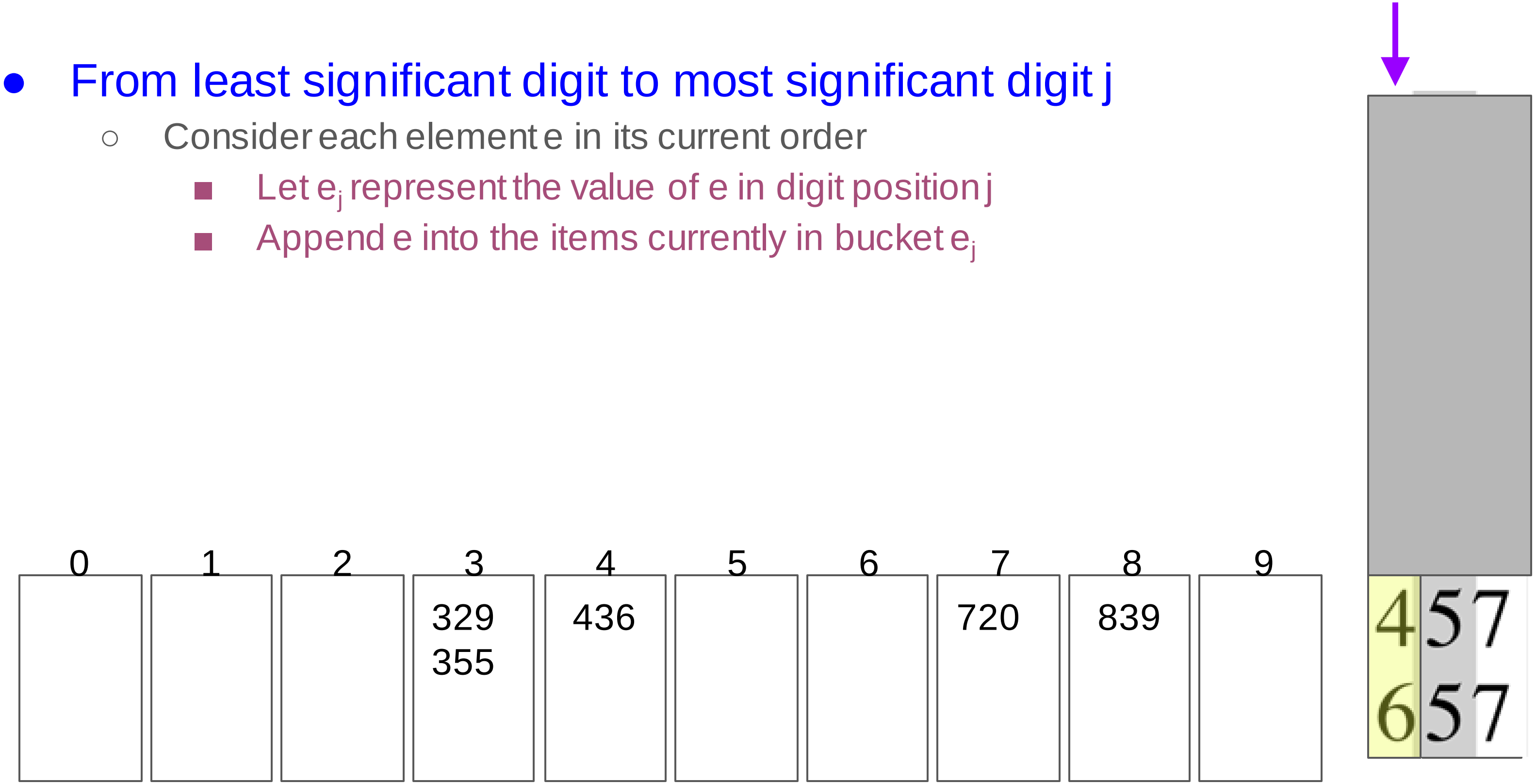
The sort is *stable*: the original order of elements is preserved within a bin





Radix Sort

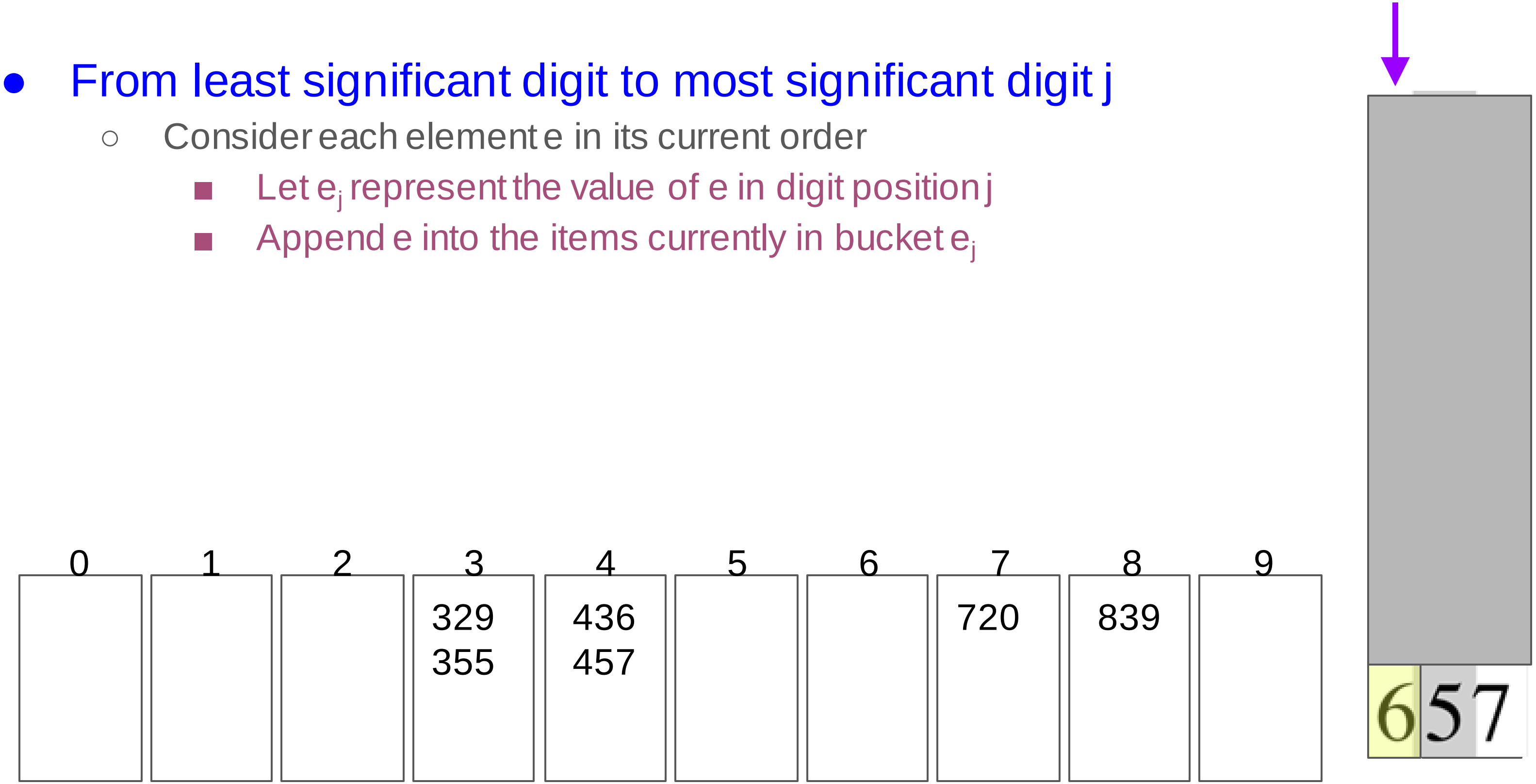
- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j



0	1	2	3	4	5	6	7	8	9
			329 355	436			720	839	

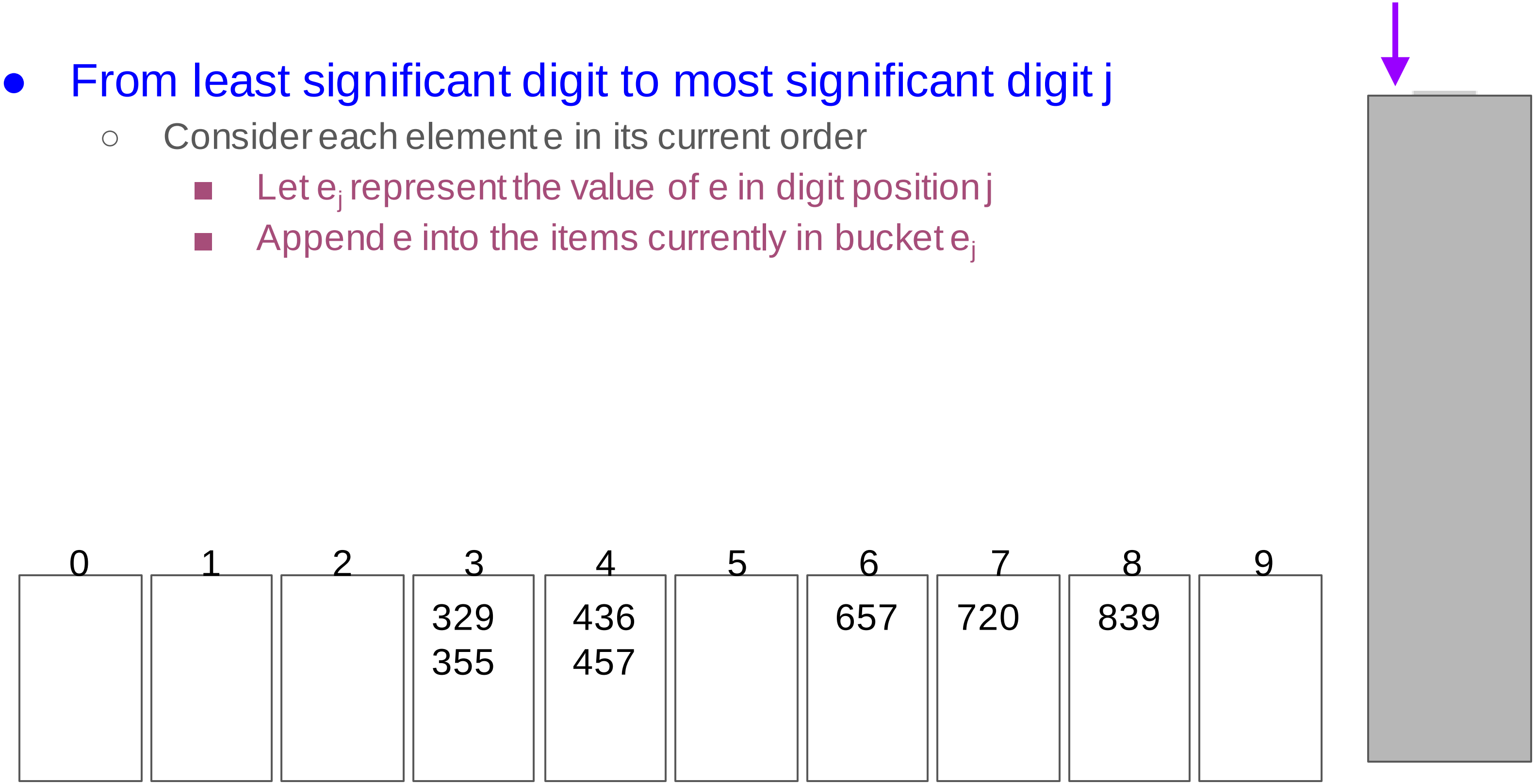
Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j



Radix Sort

- From least significant digit to most significant digit j
    - Consider each element e in its current order
        - Let e_j represent the value of e in digit position j
        - Append e into the items currently in bucket e_j

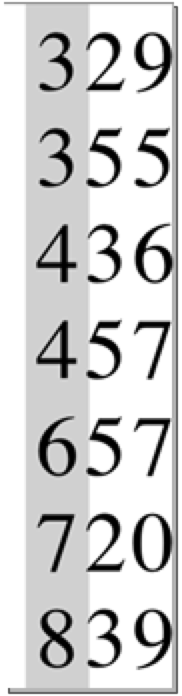


0	1	2	3	4	5	6	7	8	9
			329 355	436 457		657	720	839	

Radix Sort

- From least significant digit to most significant digit j
  - Consider each element e in its current order
    - Let e_j represent the value of e in digit position j
    - Append e into the items currently in bucket e_j

0	1	2	3	4	5	6	7	8	9
			329 355	436 457		657	720	839	


329
355
436
457
657
720
839

Radix Sort

- Alternative visualization: Radix sort on the playground

Why Does Radix Sort Work?

- **Invariant –** after j sorting passes, input is sorted by its jth least significant digits. *[Prove inductively on j]*

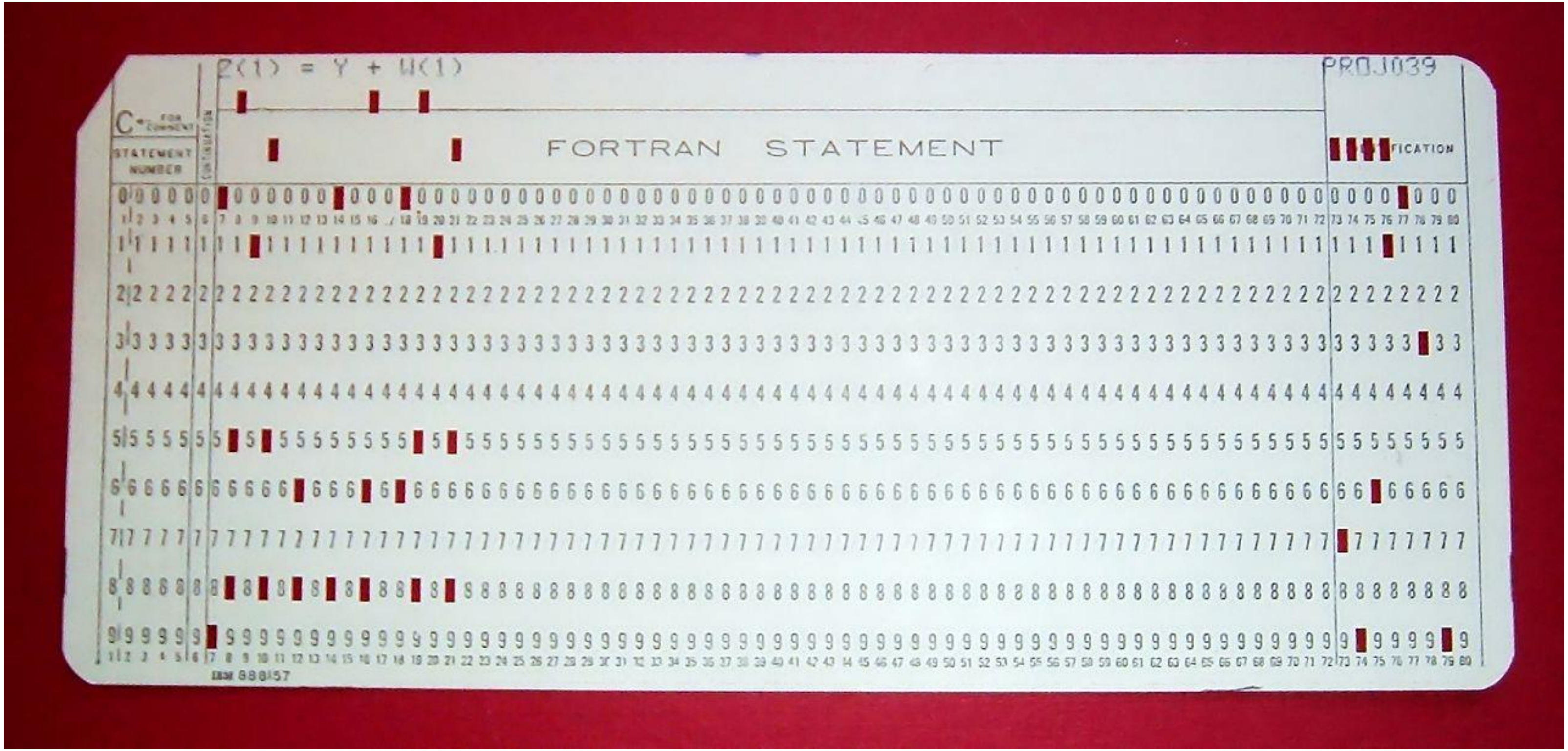
- Stability is needed to show that invariant holds for inputs with equal-valued jth digits.

- *(Proof left as exercise – see Lab 6.)*

What Does Radix Sort Cost?

- **d** passes of counting sort
- Each pass takes time **Θ(n + k)**
  - **Why?**
- Hence, total time is **Θ(d(n+k))**

Application: Sorting Punch Cards



https://en.wikipedia.org/wiki/Computer_programming_in_the_punched_card_era

Whoops! We Dropped our Deck!

FUNDAMENTAL
PHOTOGRAPHS

Application: Sorting Punch Cards

https://en.wikipedia.org/wiki/IBM_card_sorter

Sorting

End of notes