Sorting at Scale

ESC101: Fundamentals of Computing

Purushottam Kar

This Week

- Usual lecture schedule Mon, Tue, Wed 12-1PM
- Usual lab schedule Mon, Tue, Wed, Thu 2-5PM
- Usual tutorial schedule Fri 12-1PM
- Joint tutorial for B1 and B5 in L19 (same time as above)
- Major quiz and end sem lab exam marks should get declared within this week
- Will also release all remaining quiz and lab marks

End-sem Theory Exam

- **Date**: November 25th, 2018 (Sunday)
- **Time**: 9AM – 12 noon (morning)
- Not my doing – I like to sleep in on Sundays too ☹
- **Rooms**: assigned seating (like mid sem exam)
- Will be mailed to you – **sit at your own room/own seat**
  - If you do not then you will waste time moving to your proper seat
- **Syllabus**: till whatever is covered till Nov 16th tutorial
- Make-up Exam as per DoAA, SUGC guidelines
- Open handwritten notes – no printouts, mobiles, iPads.

What is Sorting?

> “Sorting is the process of arranging items systematically, ordered by some criterion”

What is Sorting?

Useful in itself – internet search and recommendation systems

"Sorting is the process of arranging items systematically, ordered by some criterion"

What is Sorting?

Useful in itself – internet search and recommendation systems

“ Sorting is the process of arranging items systematically, ordered by some criterion ”

Makes searching very fast – can search within n sorted elements in just O(log n) operations

What is Sorting?

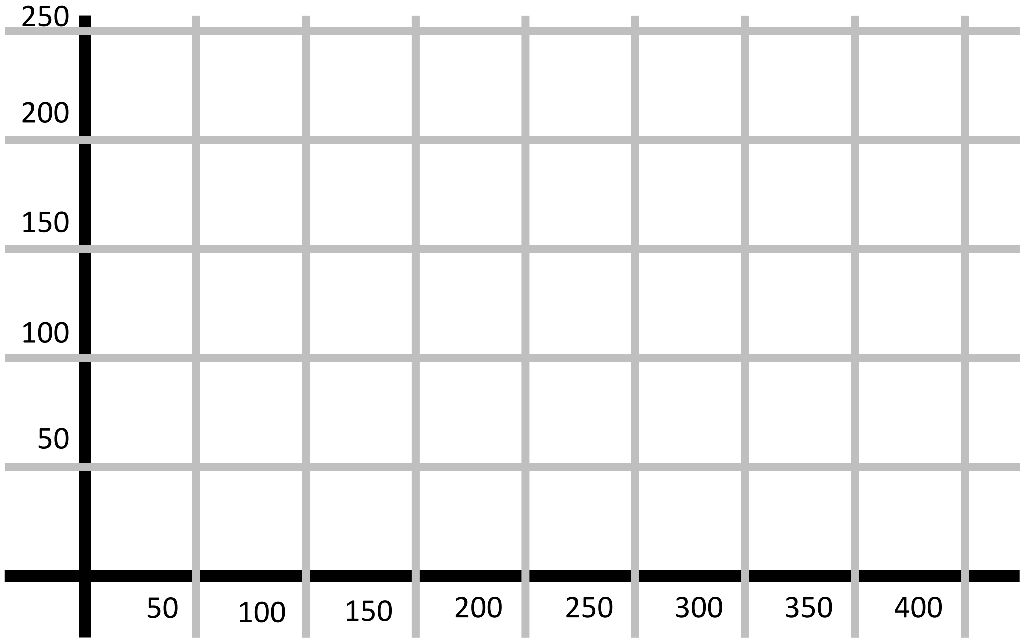
Useful in itself – internet search and recommendation systems

"Sorting is the process of arranging items systematically, ordered by some criterion"

Search within n unsorted elements can take as much as O(n) operations ☹

Makes searching very fast – can search within n sorted elements in just O(log n) operations

What is Sorting?



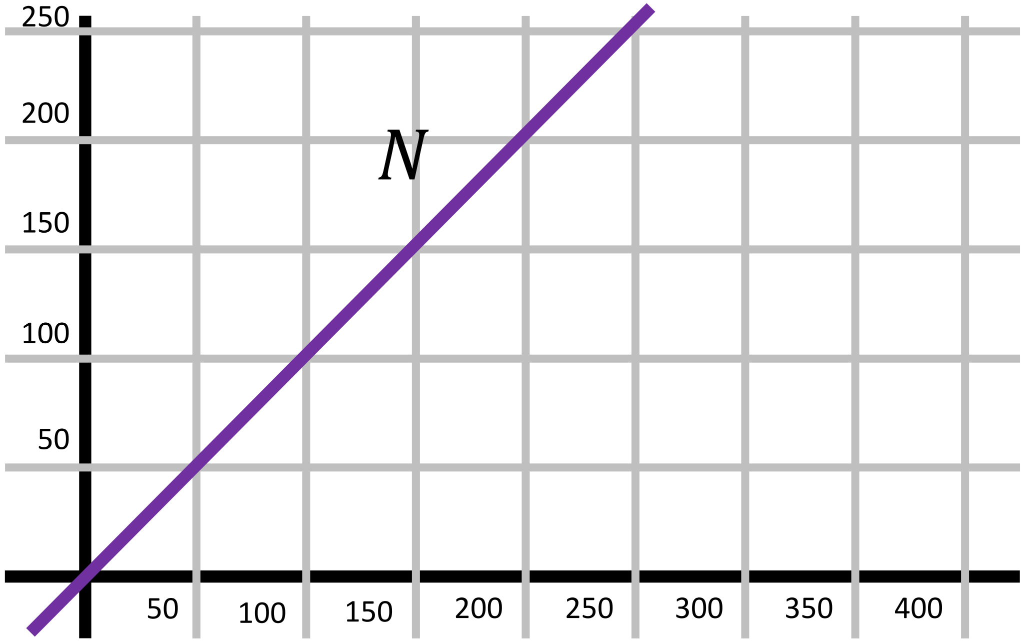
Useful in itself – internet search and recommendation systems

“Sorting is the process of arranging items systematically, ordered by some criterion”

Search within n unsorted elements can take as much as O(n) operations ☹

Makes searching very fast – can search within n sorted elements in just O(log n) operations

What is Sorting?



Useful in itself – internet search and recommendation systems

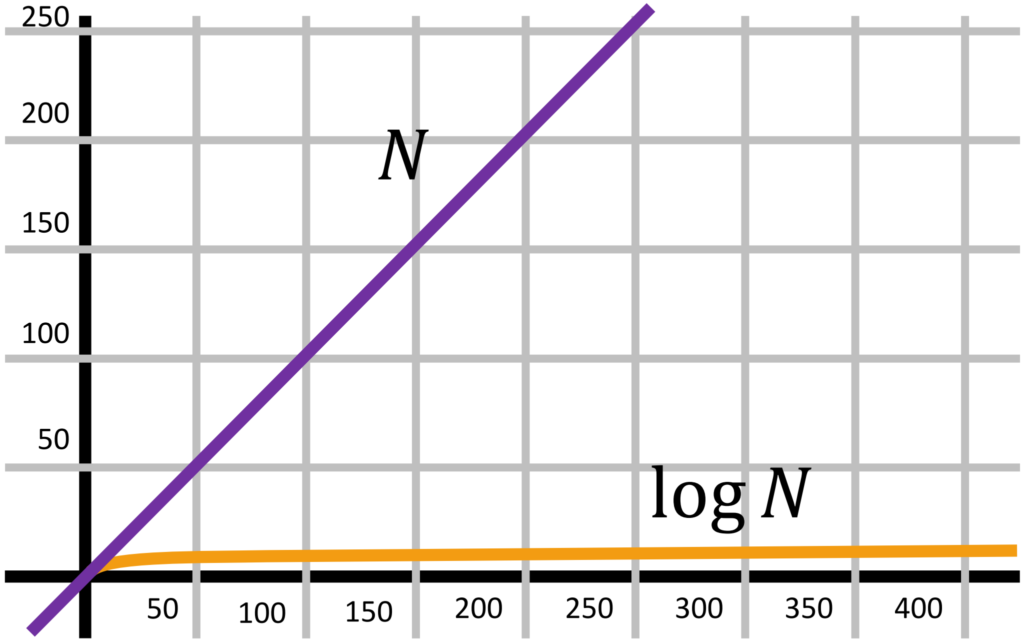
“Sorting is the process of arranging items systematically, ordered by some criterion”

Search within n unsorted elements can take as much as O(n) operations ☹

Makes searching very fast – can search within n sorted elements in just O(log n) operations

What is Sorting?

Useful in itself – internet search and recommendation systems



“Sorting is the process of arranging items systematically, ordered by some criterion”

Search within n unsorted elements can take as much as O(n) operations ☹

Makes searching very fast – can search within n sorted elements in just O(log n) operations

Sorting Algorithms

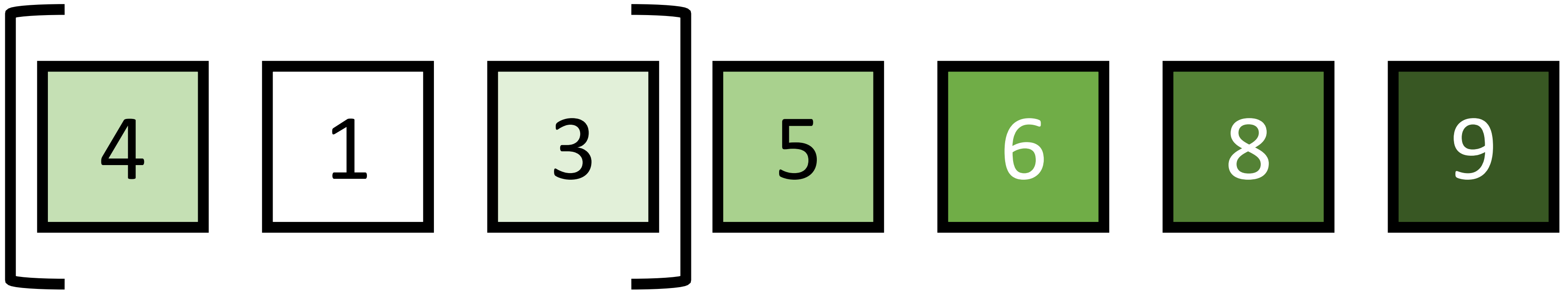
Selection Sort

Selection Sort

- One of the many (many) algorithms for sorting – very simple
- Like binary search, maintains *active range* $a[0{:}R]$ with $0 \leq R < N$
  - Initially the active range is entire array i.e. $R = N - 1$
- Invariants: we will ensure two things
  - At all points of time, the non-active portion will be sorted in ascending order i.e. for all $R \leq i < j$ we will ensure $a[i] \leq a[j]$
  - The non-active elements will never be smaller than the elements in the active range i.e. if $i \leq R < j$ then $a[i] \leq a[j]$
- The active region will shrink by one element at each step

Selection Sort

- One of the many (many) algorithms for sorting – very simple
- Like binary search, maintains *active range* $a[0{:}R]$ with $0 \le R < N$
  - Initially the active range is entire array i.e. $R = N - 1$
- Invariants: we will ensure two things
  - At all points of time, the non-active portion will be sorted in ascending order i.e. for all $R \le i < j$ we will ensure $a[i] \le a[j]$
  - The non-active elements will never be smaller than the elements in the active range i.e. if $i \le R < j$ then $a[i] \le a[j]$
- The active region will shrink by one element at each step



Selection Sort

Selection Sort

- Notice that we never have to touch the non-active region ☺

Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region

Selection Sort

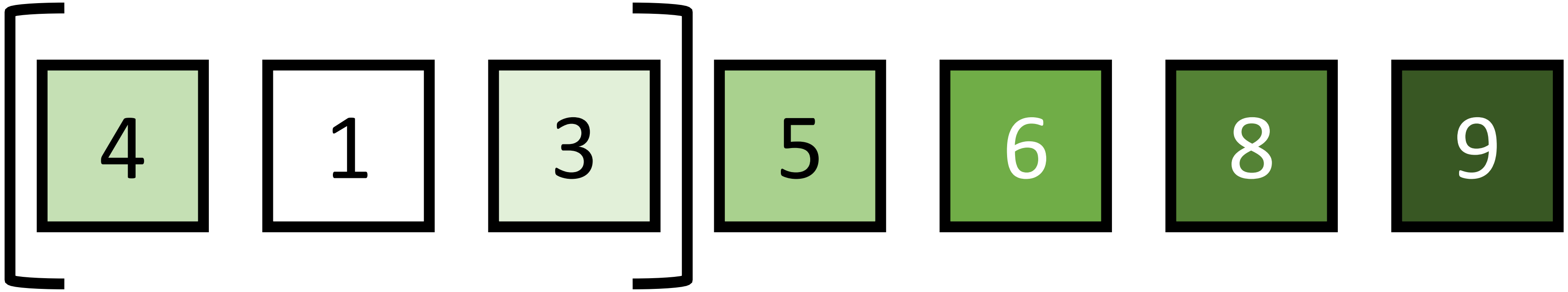
- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region

Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap

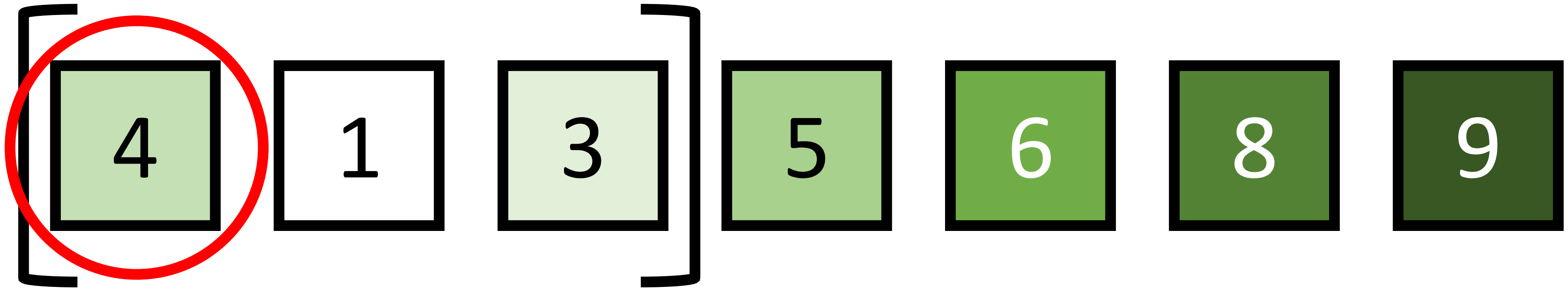
Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



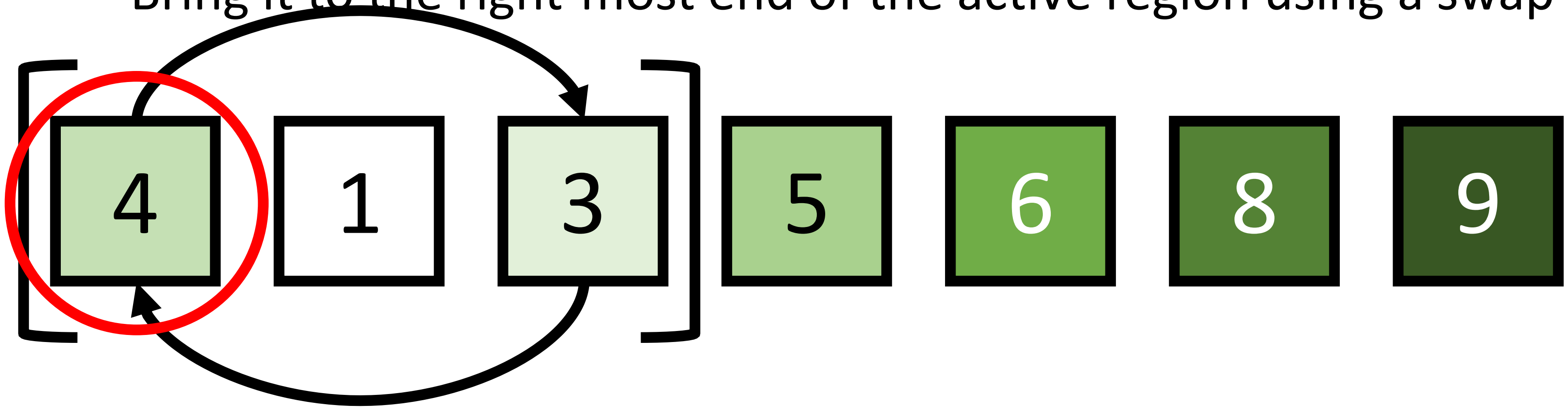
Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



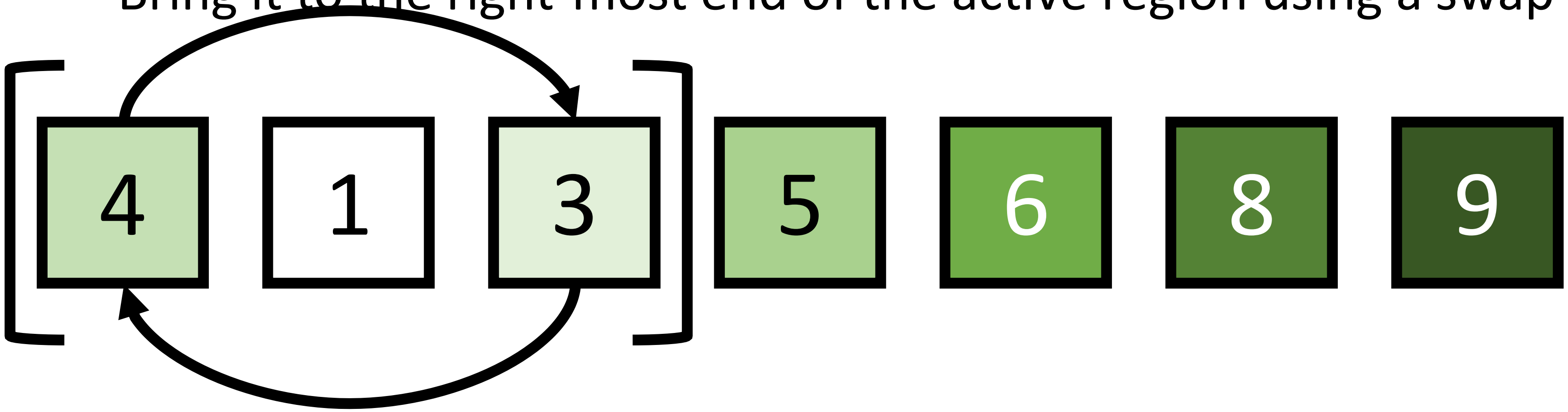
Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



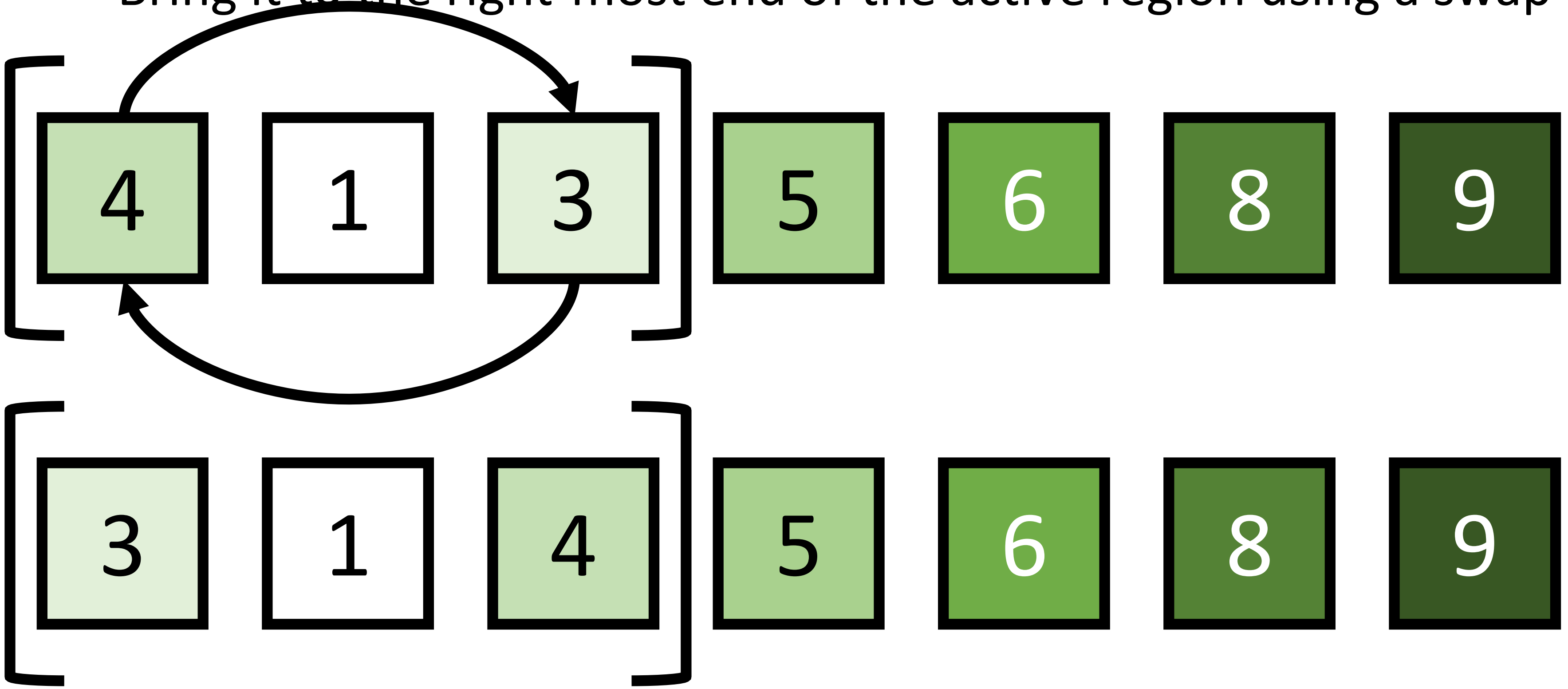
Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



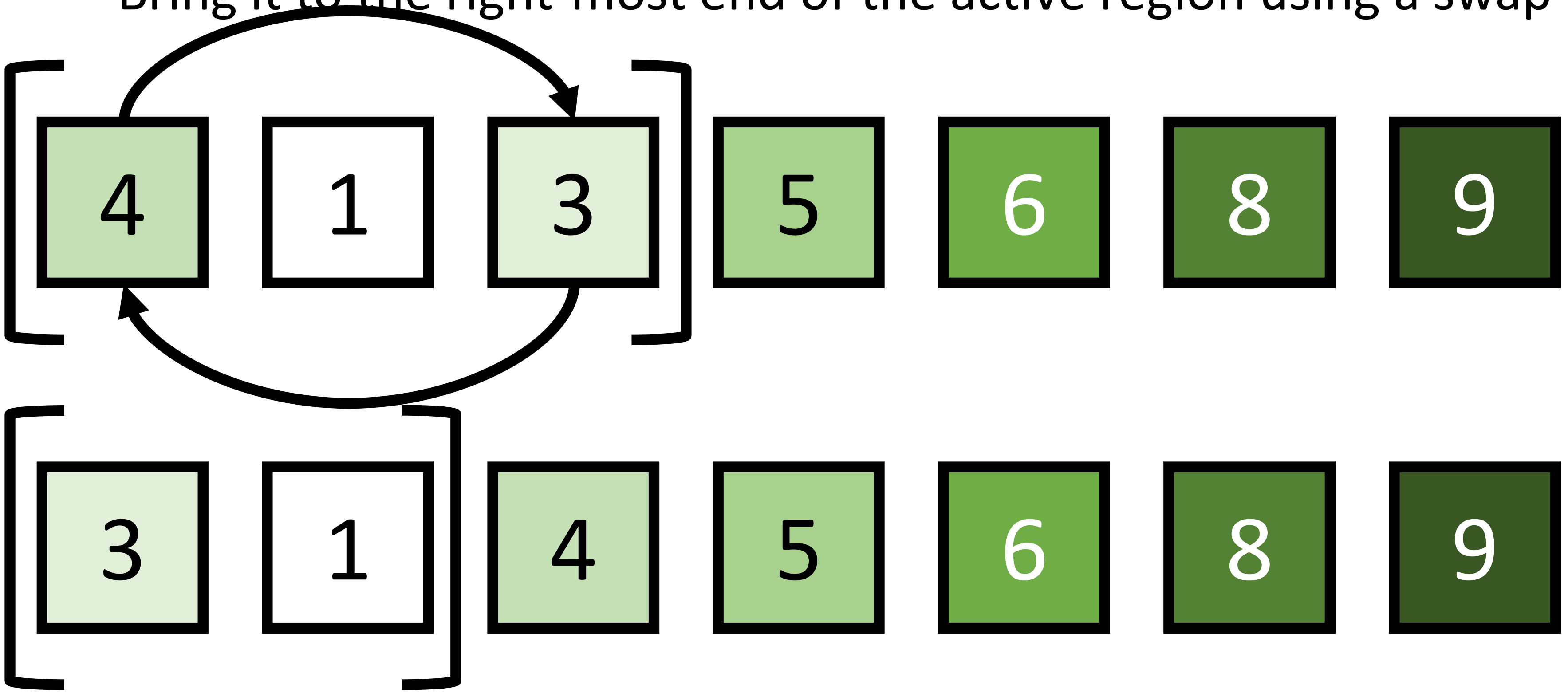
Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



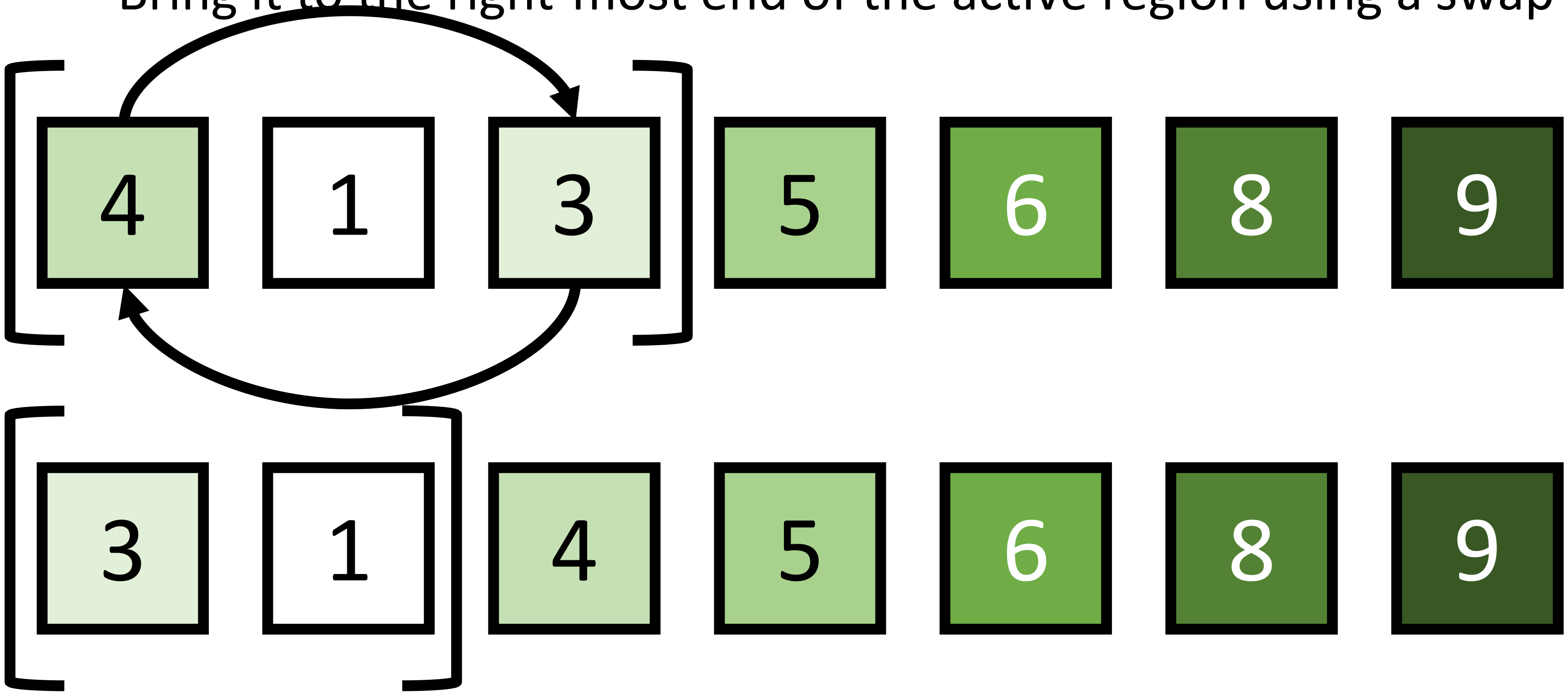
Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



Selection Sort

- Notice that we never have to touch the non-active region ☺
- To maintain the invariant and still shrink the active region
  - We search for the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



Verify that all promises of the invariant still hold

Selection Sort

SELECTION SORT

1. Given: Array a with N elements
2. For $R = N - 1; R > 0; R --$ *//Initial active range is full array*
   1. $i \leftarrow \text{FINDMAX}(a, 0, R)$ *//Location of largest element in* $a[0, R]$
   2. $\text{SWAP}(a, i, R)$ *//Bring largest element to the end*

SWAP

1. Given: Array a, location i, j
2. Let $tmp \leftarrow a[i]$
3. Let $a[i] \leftarrow a[j]$
4. Let $a[j] \leftarrow tmp$

FINDMAX

1. Given: Array a, locations i, j
2. Let $k \leftarrow i, \max = a[k]$
3. For $l = i; i \leq j; l++$
   1. If $a[l] > \max, \max = a[l], k = l$
4. Return k

Selection Sort

Exercise: convert this to proper C code

SELECTION SORT

1. Given: Array a with N elements
2. For $R = N - 1; R > 0; R - -$ *//Initial active range is full array*
   1. $i \leftarrow \text{FINDMAX}(a, 0, R)$ *//Location of largest element in* $a[0, R]$
   2. $\text{SWAP}(a, i, R)$ *//Bring largest element to the end*

SWAP

1. Given: Array a, location i, j
2. Let $tmp \leftarrow a[i]$
3. Let $a[i] \leftarrow a[j]$
4. Let $a[j] \leftarrow tmp$

FINDMAX

1. Given: Array a, locations i, j
2. Let $k \leftarrow i, \max = a[k]$
3. For $l = i; i \leq j; l + +$
   1. If $a[l] > \max, \max = a[l], k = l$
4. Return k

Selection Sort

Exercise: write a recursive version

Exercise: convert this to proper C code

SELECTION SORT

1. Given: Array a with N elements
2. For $R = N - 1; R > 0; R - -$ *//Initial active range is full array*
   1. $i \leftarrow \text{FINDMAX}(a, 0, R)$ *//Location of largest element in* $a[0, R]$
   2. $\text{SWAP}(a, i, R)$ *//Bring largest element to the end*

SWAP

1. Given: Array a, location i, j
2. Let $tmp \leftarrow a[i]$
3. Let $a[i] \leftarrow a[j]$
4. Let $a[j] \leftarrow tmp$

FINDMAX

1. Given: Array a, locations i, j
2. Let $k \leftarrow i, \max = a[k]$
3. For $l = i; i \leq j; l + +$
   1. If $a[l] > \max, \max = a[l], k = l$
4. Return k

Time Complexity

- Let $T(N)$ be the time taken for selection sort to sort N elements
- Let $M(N)$ be the time taken to find location of max of N elements
- At any time step when active region is $[0{:}\,R]$, we do two things
  - Find the largest element within the active region – takes time $M(R+1)$
  - Swap the largest element with the element at $a[R]$ - takes time c (const)
- Thus, we have $T(N) \leq M(N) + c + T(N-1)$
- It is easy to show that $M(N) \leq d \cdot N$ for all N for some constant d
- Exercise: expand the recurrence as before and show that
  $$T(N) \leq \mathcal{O}(N^2)$$
  Assume $T(1) \leq c$
- Notice that selection sort doesn't need any extra memory (except a few tmp variables to store one integer each) – *in-place sorting*

Summary

Summary

- Applications of sorting: ranking, recommendation, internet search

Summary

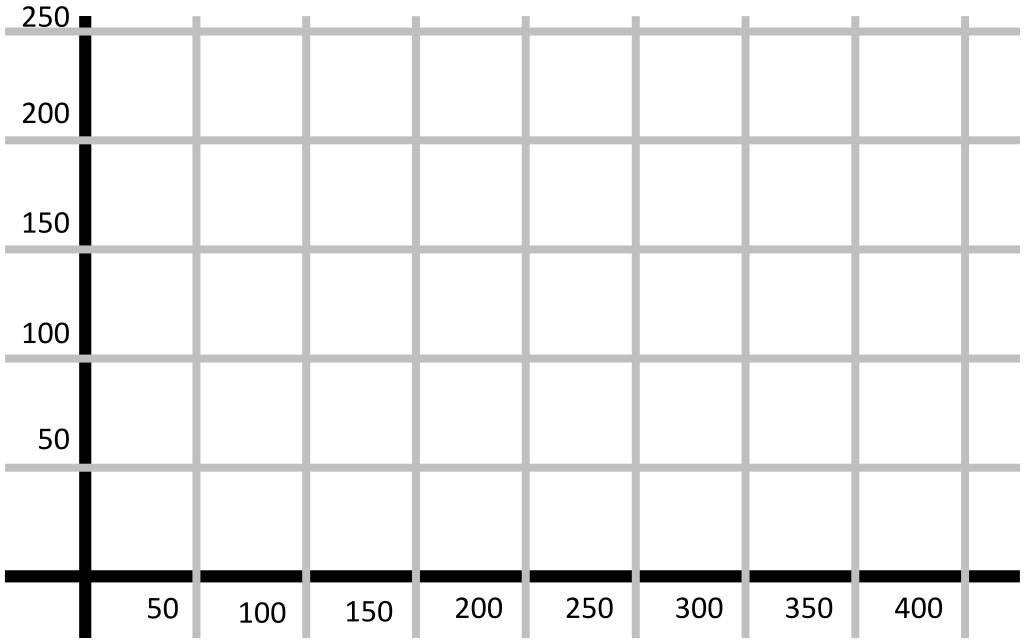
- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)

Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)

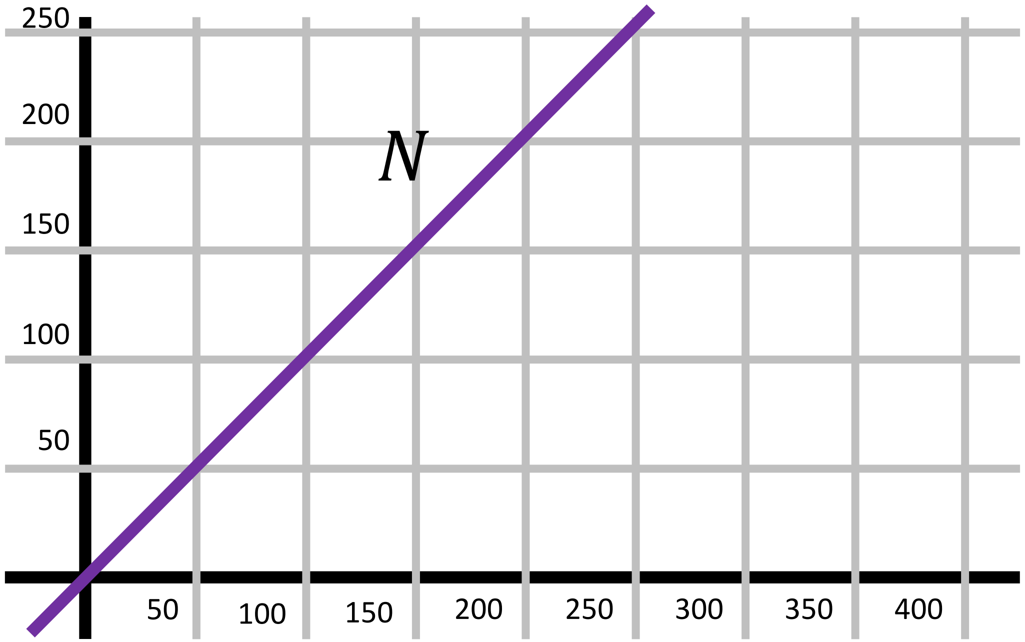
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)



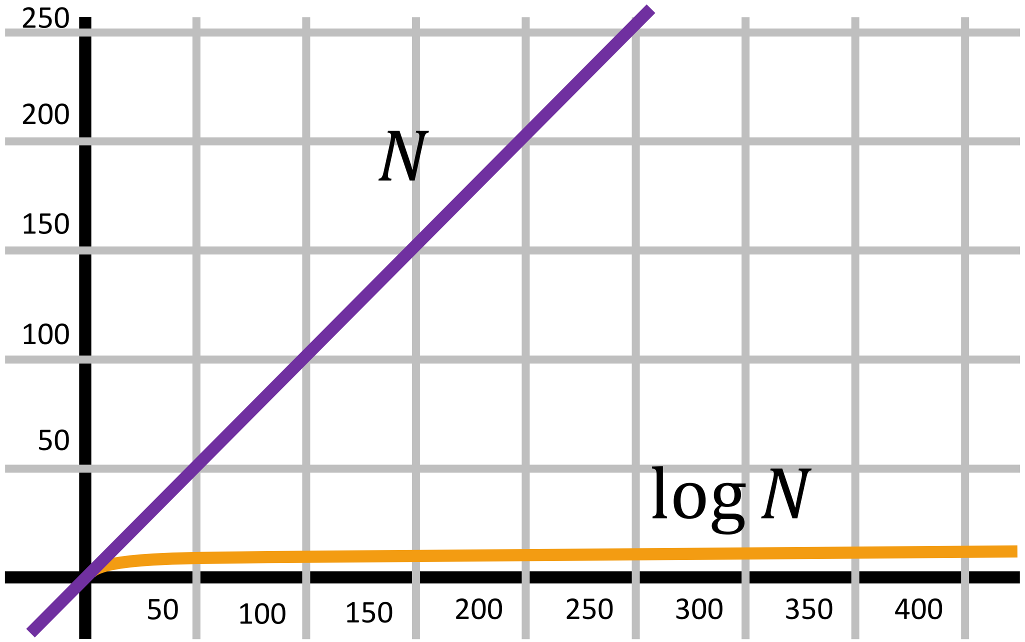
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)



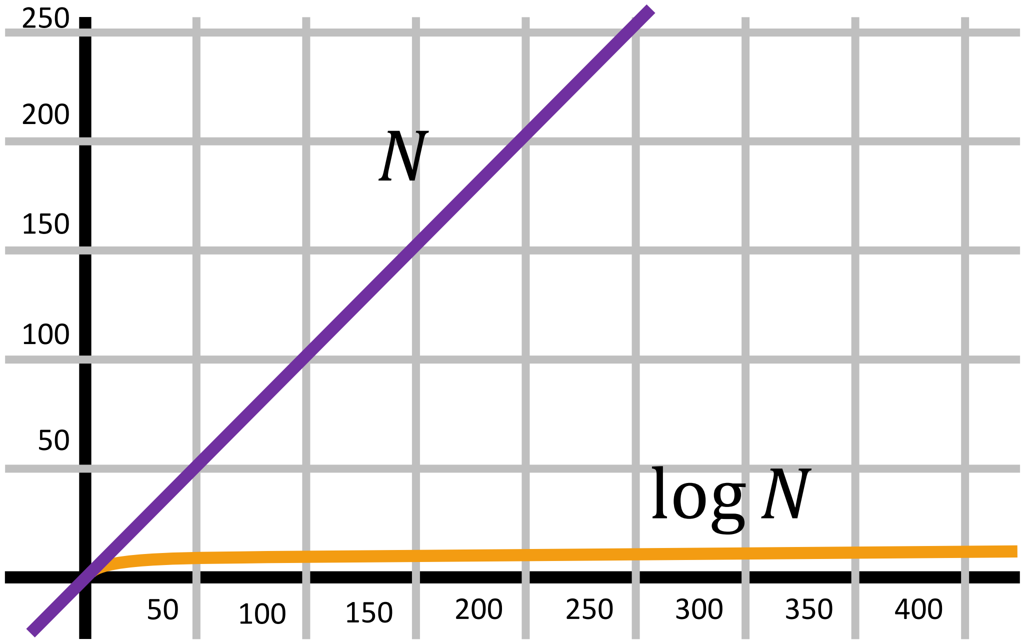
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)



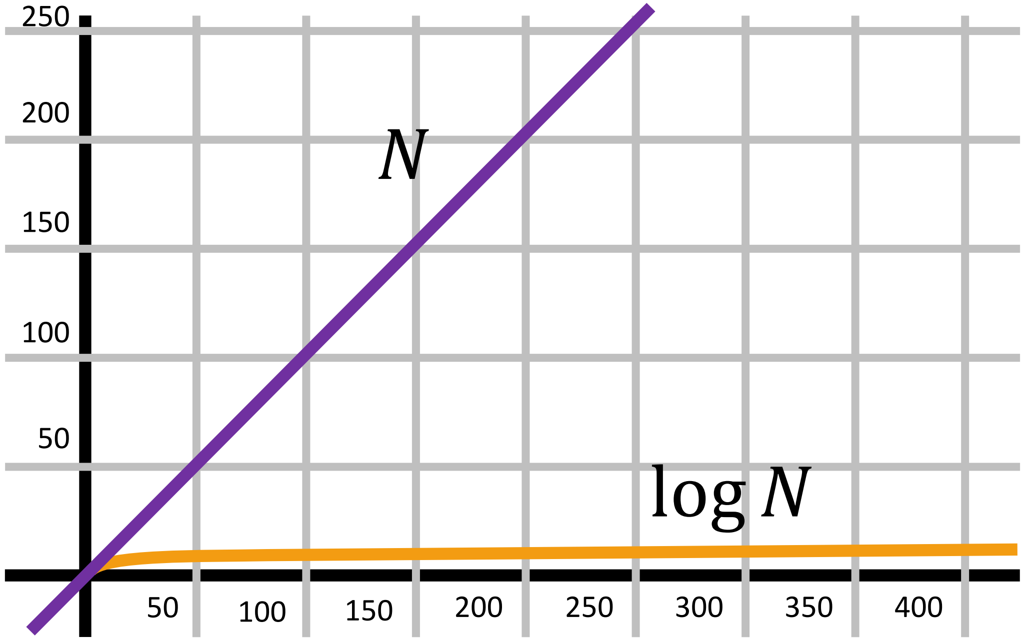
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)



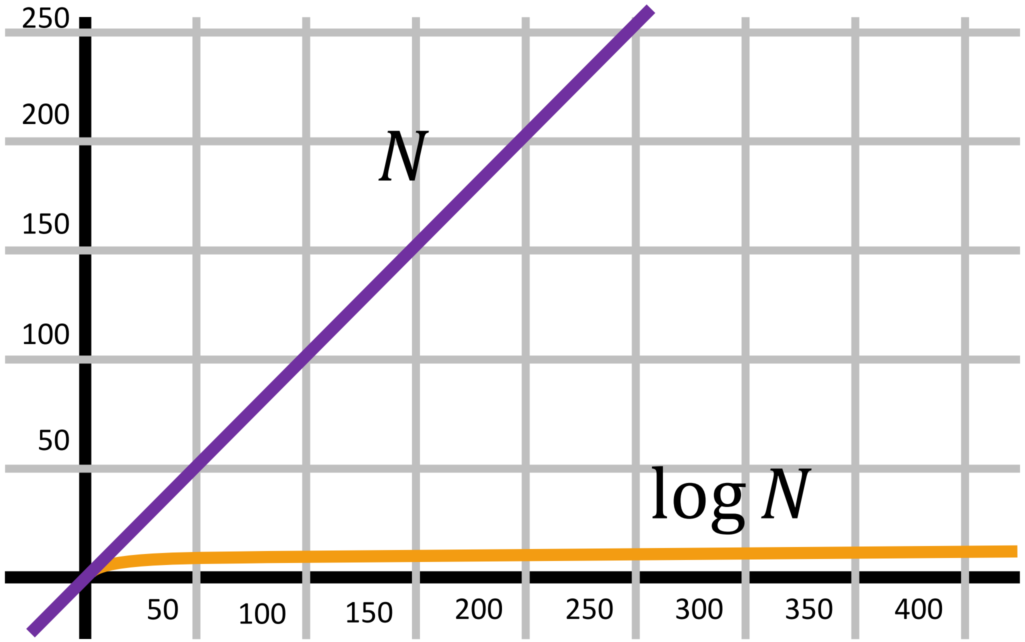
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)
- Next: fast sorting O(N log N)



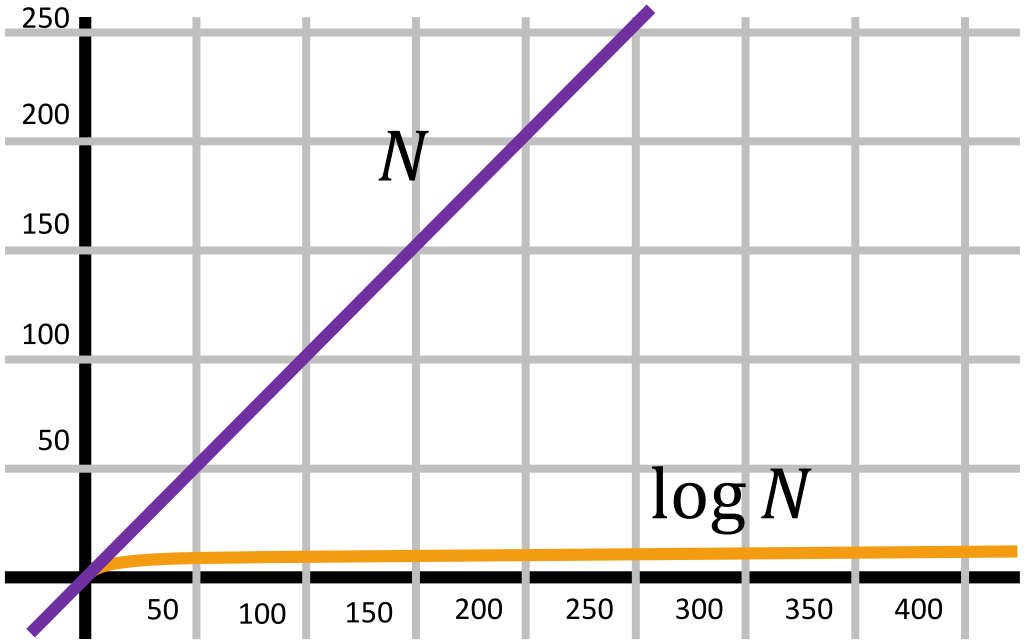
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)
- Next: fast sorting O(N log N)
  - Merge Sort



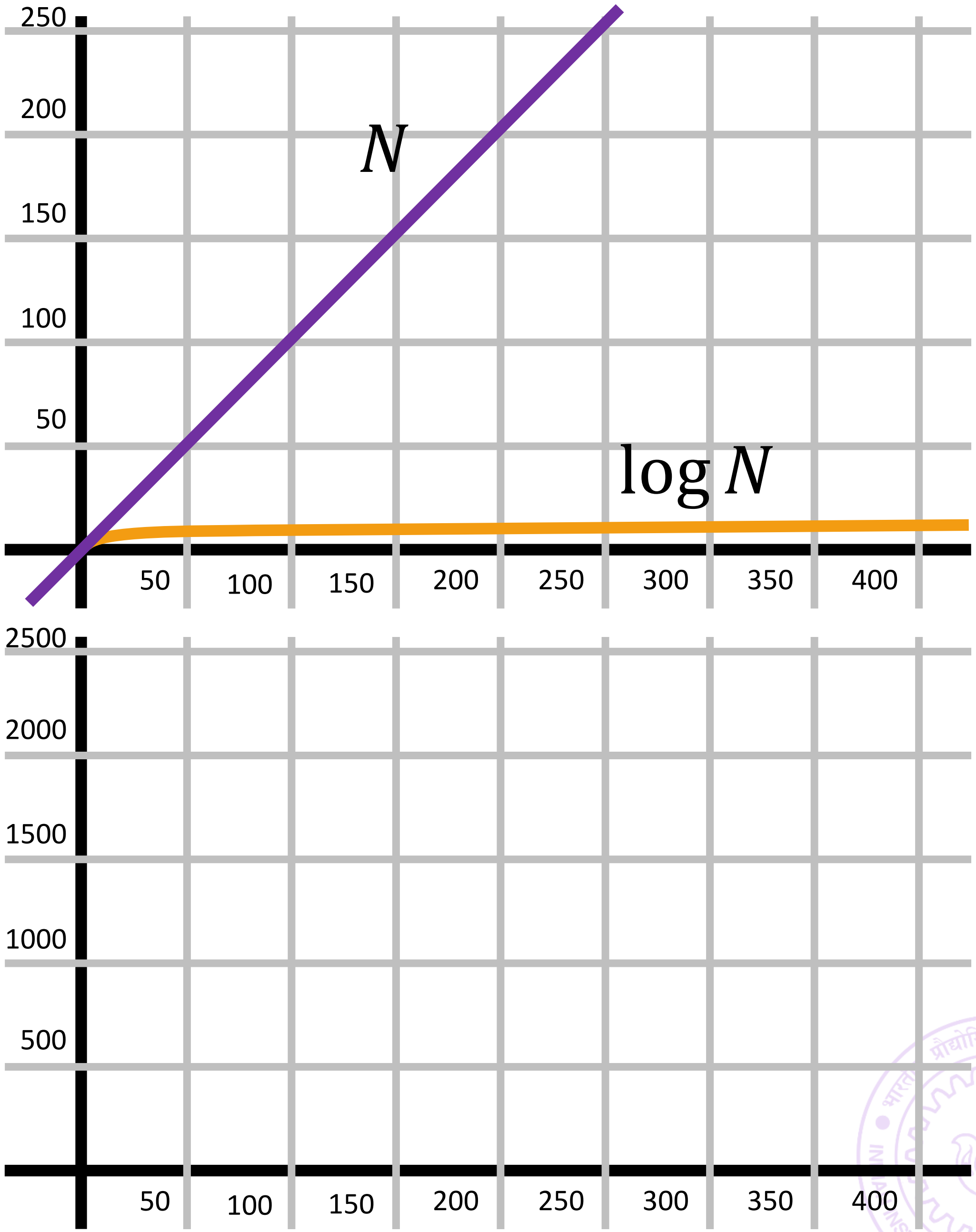
Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)
- Next: fast sorting O(N log N)
  - Merge Sort
  - Quick Sort



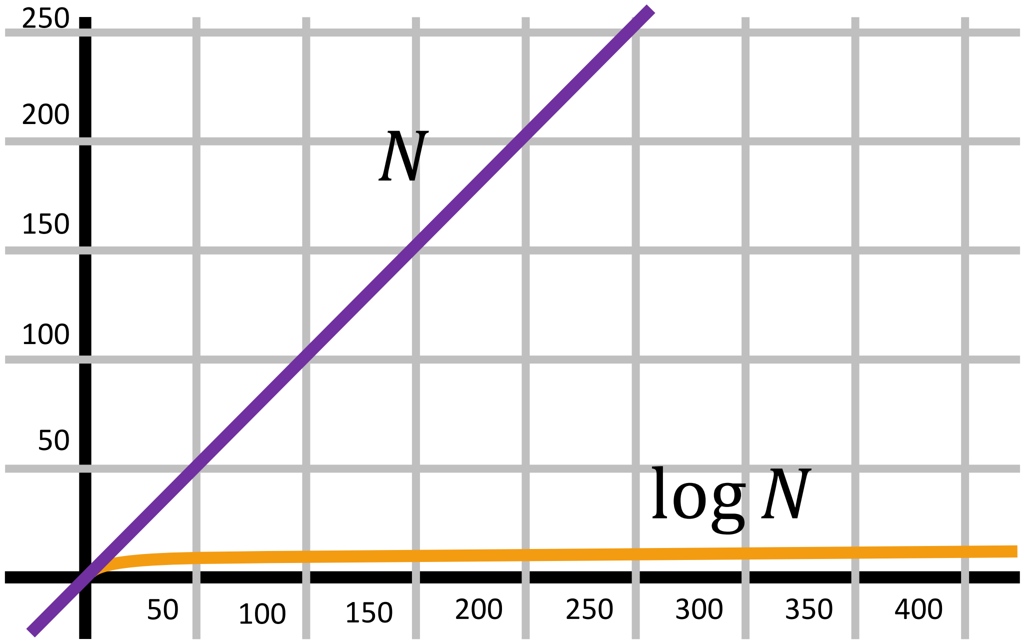
Summary

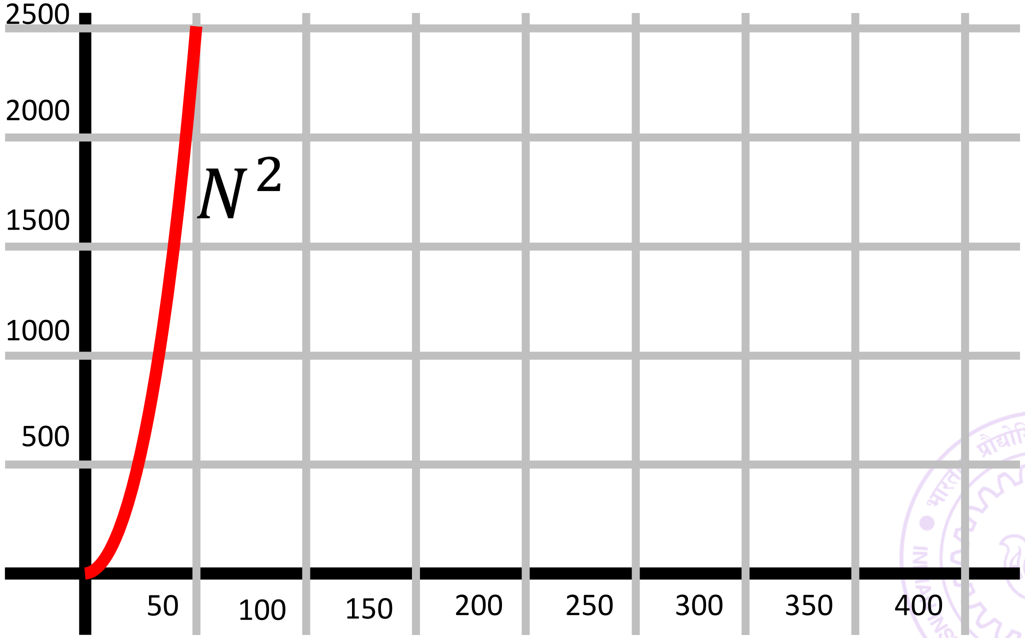
- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)
- Next: fast sorting O(N log N)
  - Merge Sort
  - Quick Sort



Summary

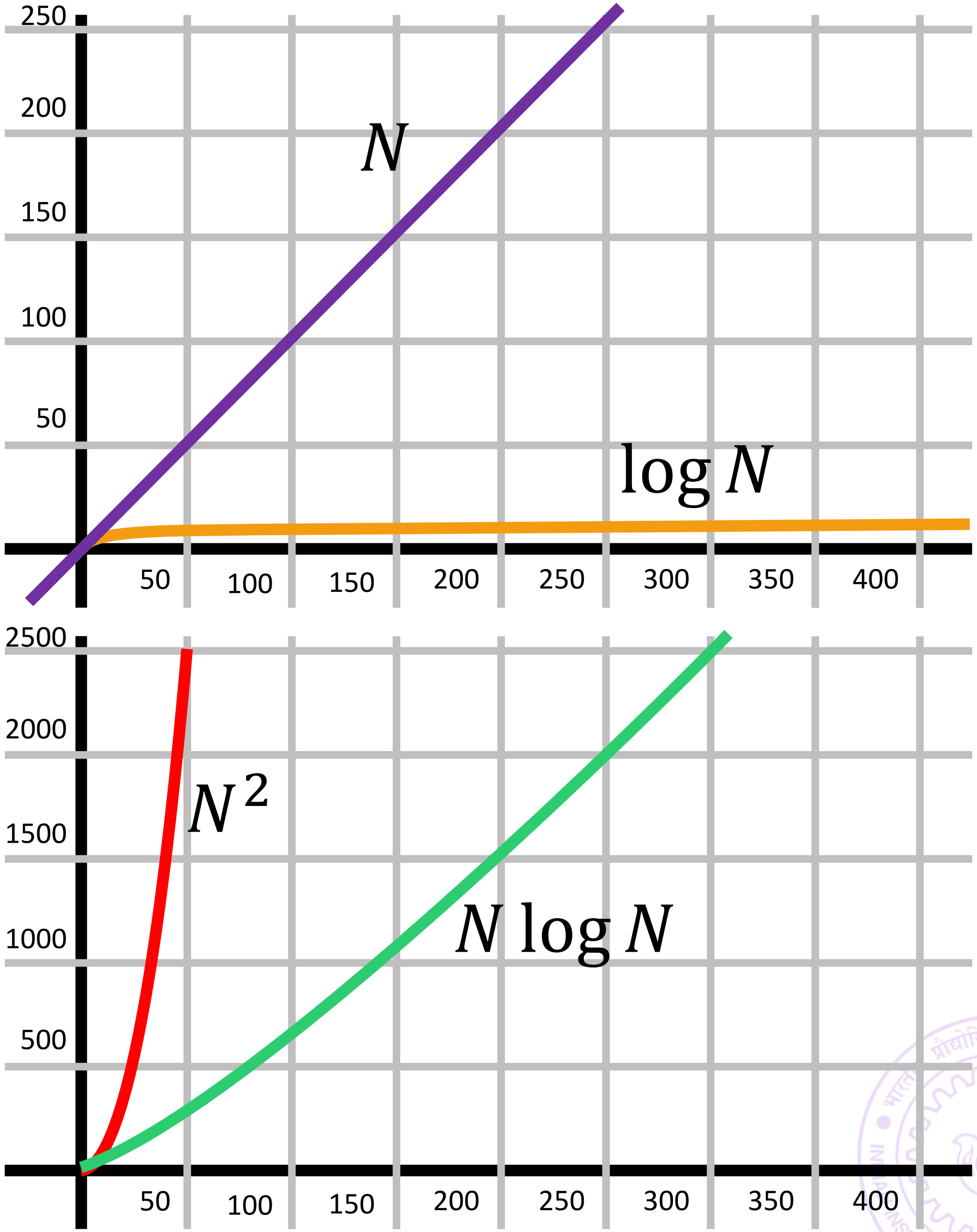
- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)
- Next: fast sorting O(N log N)
  - Merge Sort
  - Quick Sort





Summary

- Applications of sorting: ranking, recommendation, internet search
- Brute force search O(N)
- Fast searches on sorted arrays: binary search O(log N)
- Selection sort O(N^2)
- Next: fast sorting O(N log N)
  - Merge Sort
  - Quick Sort



Partition based Sorting Techniques

- Let $T(N)$ be the time taken for selection sort to sort N elements
- Let $M(N)$ be the time taken to find location of max of N elements
- For selection sort, we have $T(N) \leq M(N) + c + T(N-1)$
- Active region shrank too slowly which gave us $T(N) \leq \mathcal{O}(N^2)$
- Selection sort is quite an expensive algorithm (imagine $\mathcal{O}(N^2)$ time complexity for $N = 1{,}000{,}000$ items ☹) – much better can be done
- Will study two algorithms based on divide and conquer technique
- Both techniques split an array of N elements into two arrays, sorts each smaller array and then does some clean up operations
  - Merge Sort: popular for sorting large scale databases
  - Quick Sort: extremely popular in general (see qsort() in stdlib.h)

Sorting Algorithms

Merge Sort

Merge Sort

Merge Sort

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

Merge Sort

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

Merge Sort

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

Merge Sort

[9 1 9 4 2 1 2 7] [5 8 3 5 2 1 4 8]

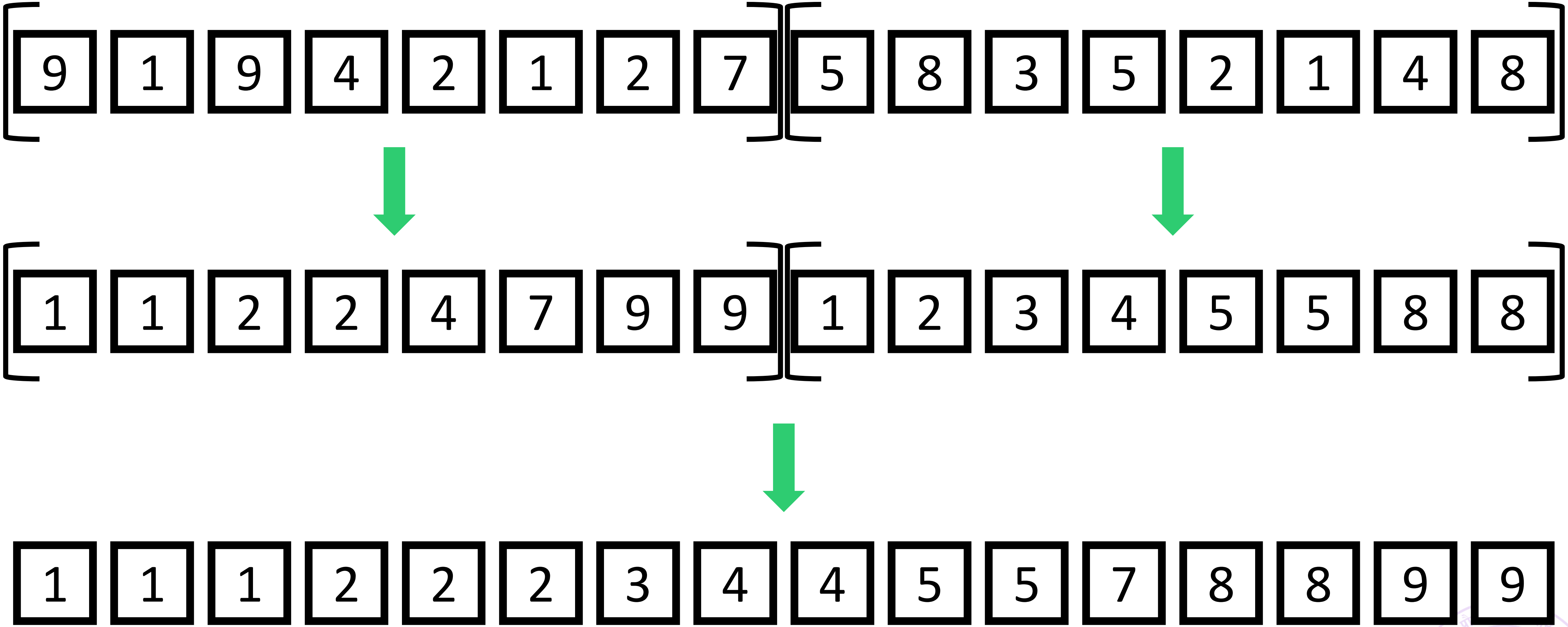
[1 1 2 2 4 7 9 9]

Merge Sort

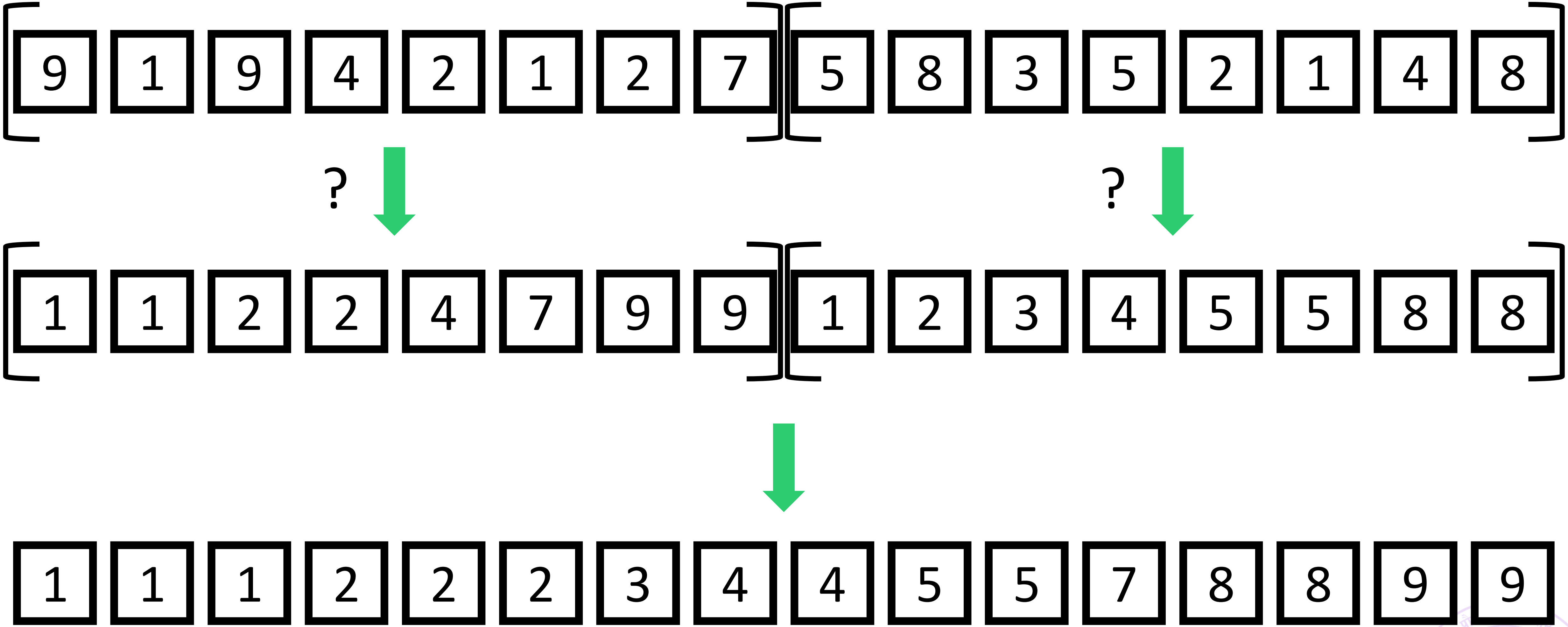
9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	1	2	2	4	7	9	9	1	2	3	4	5	5	8	8

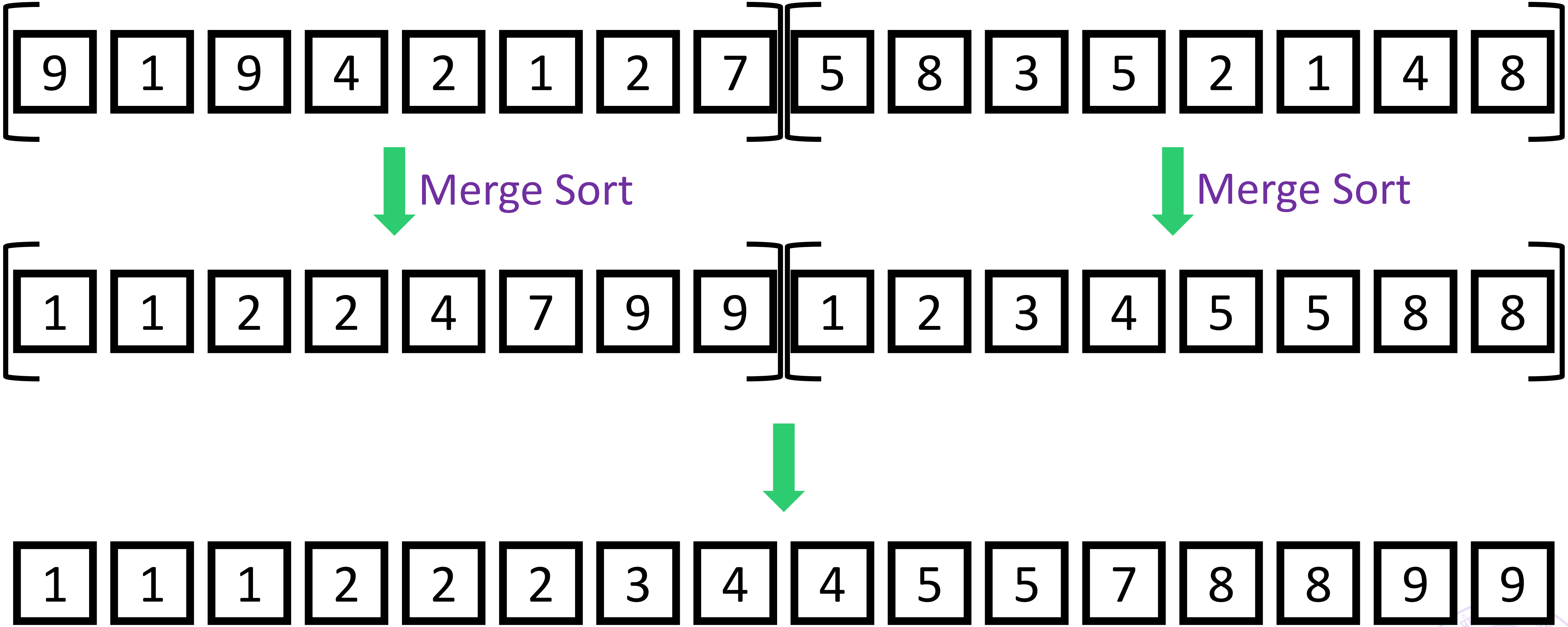
Merge Sort



Merge Sort



Merge Sort


9 1 9 4 2 1 2 7
5 8 3 5 2 1 4 8
Merge Sort
Merge Sort
1 1 2 2 4 7 9 9
1 2 3 4 5 5 8 8
1 1 1 2 2 2 3 4 4 5 5 7 8 8 9 9

Merge Sort

[9 | 1 | 9 | 4 | 2 | 1 | 2 | 7] [5 | 8 | 3 | 5 | 2 | 1 | 4 | 8]

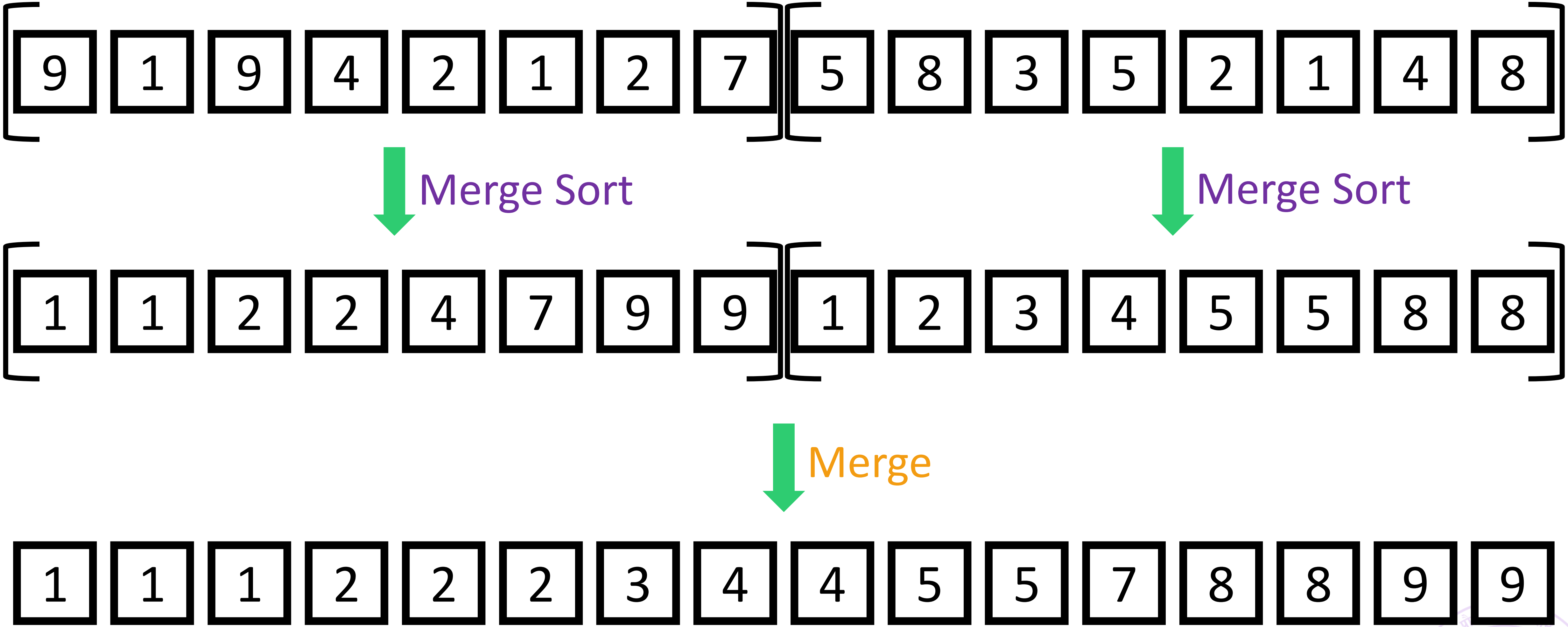
Merge Sort ↓ Merge Sort ↓

[1 | 1 | 2 | 2 | 4 | 7 | 9 | 9] [1 | 2 | 3 | 4 | 5 | 5 | 8 | 8]

? ↓

1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 5 | 5 | 7 | 8 | 8 | 9 | 9

Merge Sort



9 1 9 4 2 1 2 7 5 8 3 5 2 1 4 8
Merge Sort
Merge Sort
1 1 2 2 4 7 9 9 1 2 3 4 5 5 8 8
Merge
1 1 1 2 2 2 3 4 4 5 5 7 8 8 9 9

Merge Sort

[9 | 1 | 9 | 4 | 2 | 1 | 2 | 7] [5 | 8 | 3 | 5 | 2 | 1 | 4 | 8]

Merge Sort ↓ Merge Sort ↓

[1 | 1 | 2 | 2 | 4 | 7 | 9 | 9] [1 | 2 | 3 | 4 | 5 | 5 | 8 | 8]

↓ Merge

| 1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 5 | 5 | 7 | 8 | 8 | 9 | 9 |

Trick: Merging two sorted arrays is very easy!

Merge Sort

MERGE SORT

1. Given: Array a with N elements
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $C \leftarrow \text{ceil}(N/2)$ *//Find the "middle" of the array*
4. $p \leftarrow$ MERGESORT($a[0:C-1]$) *//Sort the left half*
5. $q \leftarrow$ MERGESORT($a[C:N-1]$) *//Sort the right half*
6. Return MERGE(p, q) *//Merge the two halves*

Merge Sort

MERGE SORT

1. Given: Array a with N elements
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $C \leftarrow \text{ceil}(N/2)$ *//Find the "middle" of the array*
4. $p \leftarrow \text{MERGESORT}(a[0: C-1])$ *//Sort the left half*
5. $q \leftarrow \text{MERGESORT}(a[C: N-1])$ *//Sort the right half*
6. Return MERGE(p, q) *//Merge the two halves*

Uses a lot of extra memory p, q. Even MERGE uses extra memory – not good! Need an in-place version

Merge Sort

MERGE SORT

1. Given: Array a with N elements
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $C \leftarrow \text{ceil}(N/2)$ *//Find the "middle" of the array*
4. $p \leftarrow \text{MERGESORT}(a[0:C-1])$
5. $q \leftarrow \text{MERGESORT}(a[C:N-1])$
6. Return $\text{MERGE}(p,q)$ //

A sort algorithm is called *in-place* if it does not use extra memory e.g. extra arrays, to sort the given array

Uses a lot of extra memory p, q. Even MERGE uses extra memory – not good! Need an in-place version

Merge Sort

MERGE SORT

1. Given: Array a with N elements
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $C \leftarrow \text{ceil}(N/2)$ *//Find the "middle" of the array*
4. $p \leftarrow$ MERGESORT($a[0: C-1]$)
5. $q \leftarrow$ MERGESORT($a[C: N-1]$)
6. Return MERGE(p, q) //

A sort algorithm is called *in-place* if it does not use extra memory e.g. extra arrays, to sort the given array

Uses a lot of extra memory p, q. Even MERGE uses extra memory – not good! Need an in-place version

Why did we split as $[0: C-1], [C: N-1]$ and not as $[0: C], [C+1: N-1]$? Hint: end-case.

Merge Sort

Why didn't we split as $[0: N-2], [N-1: N-1]$?
No need to find middle element. Also, would have made one of the mergesort calls so simple!

1. Given: Array a with N elements
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $C \leftarrow \text{ceil}(N/2)$ *//Find the "middle" of the array*
4. $p \leftarrow$ MERGESORT$(a[0: C-1])$
5. $q \leftarrow$ MERGESORT$(a[C: N-1])$
6. Return MERGE(p, q) //

A sort algorithm is called *in-place* if it does not use extra memory e.g. extra arrays, to sort the given array

Uses a lot of extra memory p, q. Even MERGE uses extra memory – not good! Need an in-place version

Why did we split as $[0: C-1], [C: N-1]$ and not as $[0: C], [C+1: N-1]$? Hint: end-case.

Time Complexity

- Let $T(N)$ be the time taken for merge sort to sort N elements
- Let $M(N)$ be time merging two sorted arrays with total N elements
- Thus, we have $T(N) \leq 2 \cdot T(N/2) + M(N) + d$ (d: find middle index)
- We will show next that we can do $M(N) \leq c \cdot N$ time
- This recurrence is a bit harder to solve but we can still try

$$T(N/2) \leq 2 \cdot T(N/4) + c \cdot N/2 + d$$
$$T(N) \leq 4 \cdot T(N/4) + 2c \cdot N + (1 + 2) \cdot d$$
$$T(N) \leq 2^k \cdot T\left(N/2^k\right) + kc \cdot N + 2^k \cdot d$$

- Set $k = \text{ceil}(\log N)$ and use $T(1) \leq c$ to get $T(N) \leq \mathcal{O}(N \log N)$
- The version of merging we will show uses extra $\mathcal{O}(N)$ memory. Can you develop a version that uses only 2-3 extra integer variables i.e. an *in-place* version of merge sort?

Time Complexity

If we had split as $[0: N-2], [N-1: N-1]$ then $T(N) \leq T(N-1) + T(1) + M(N)$ would have given us $T(N) = \mathcal{O}(N^2)$ (divide properly to rule powerfully ☺)

- Let $T(N)$ be the time take… sort N elements
- Let $M(N)$ be time merging two sorted arrays with total N elements
- Thus, we have $T(N) \leq 2 \cdot T(N/2) + M(N) + d$ (d: find middle index)
- We will show next that we can do $M(N) \leq c \cdot N$ time
- This recurrence is a bit harder to solve but we can still try

$$T(N/2) \leq 2 \cdot T(N/4) + c \cdot N/2 + d$$
$$T(N) \leq 4 \cdot T(N/4) + 2c \cdot N + (1+2) \cdot d$$
$$T(N) \leq 2^k \cdot T\left(N/2^k\right) + kc \cdot N + 2^k \cdot d$$

- Set $k = \text{ceil}(\log N)$ and use $T(1) \leq c$ to get $T(N) \leq \mathcal{O}(N \log N)$
- The version of merging we will show uses extra $\mathcal{O}(N)$ memory. Can you develop a version that uses only 2-3 extra integer variables i.e. an *in-place* version of merge sort?

The Merge Operation

- Given 2 arrays int $a[M], b[N]$; both sorted in ascending order
- Want a combined array int $c[M+N]$; sorted in ascending order
- Will maintain *active ranges* for both arrays $a[0: R_1]$ and $b[0: R_2]$ with $0 \leq R_1 < M$ and $0 \leq R_2 < N$
  - Initially the active ranges are the entire arrays i.e. $R_1 = M-1, R_2 = N-1$
- Invariant: at all points of time, we will ensure that elements in the non-active regions of the arrays would have been inserted into c at their proper locations
- At least one active region will shrink by one element at each step
- Trick: the largest element of c can be found in $\mathcal{O}(1)$ time since the arrays a, b are sorted. If unsorted it would have taken $\mathcal{O}(M+N)$

The Merge Operati

Actually, we have already done this – see Week 7 Monday lab question 1 ☺

- Given 2 arrays int $a[M], b[N]$; both sorted in ascending order
- Want a combined array int $c[M + N]$; sorted in ascending order
- Will maintain *active ranges* for both arrays $a[0: R_1]$ and $b[0: R_2]$ with $0 \leq R_1 < M$ and $0 \leq R_2 < N$
  - Initially the active ranges are the entire arrays i.e. $R_1 = M - 1, R_2 = N - 1$
- Invariant: at all points of time, we will ensure that elements in the non-active regions of the arrays would have been inserted into c at their proper locations
- At least one active region will shrink by one element at each step
- Trick: the largest element of c can be found in $\mathcal{O}(1)$ time since the arrays a, b are sorted. If unsorted it would have taken $\mathcal{O}(M + N)$

The Merge Operation

The Merge Operation

1	1	2	2	4	7	9	9

A

The Merge Operation

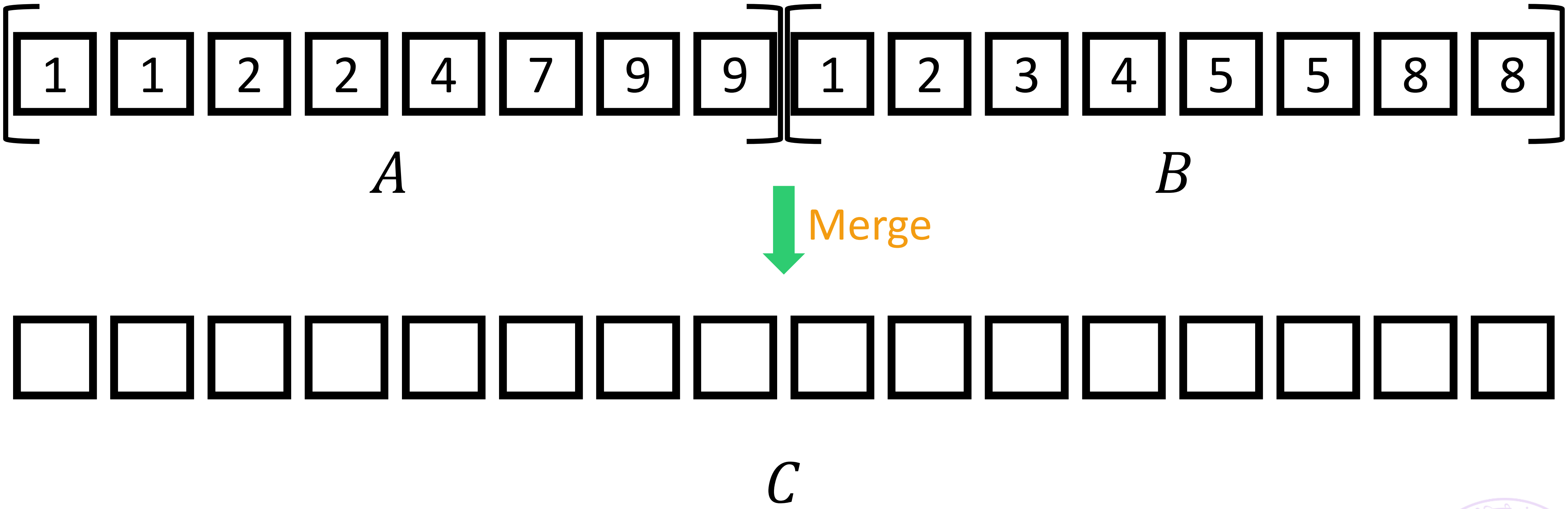
1	1	2	2	4	7	9	9

A

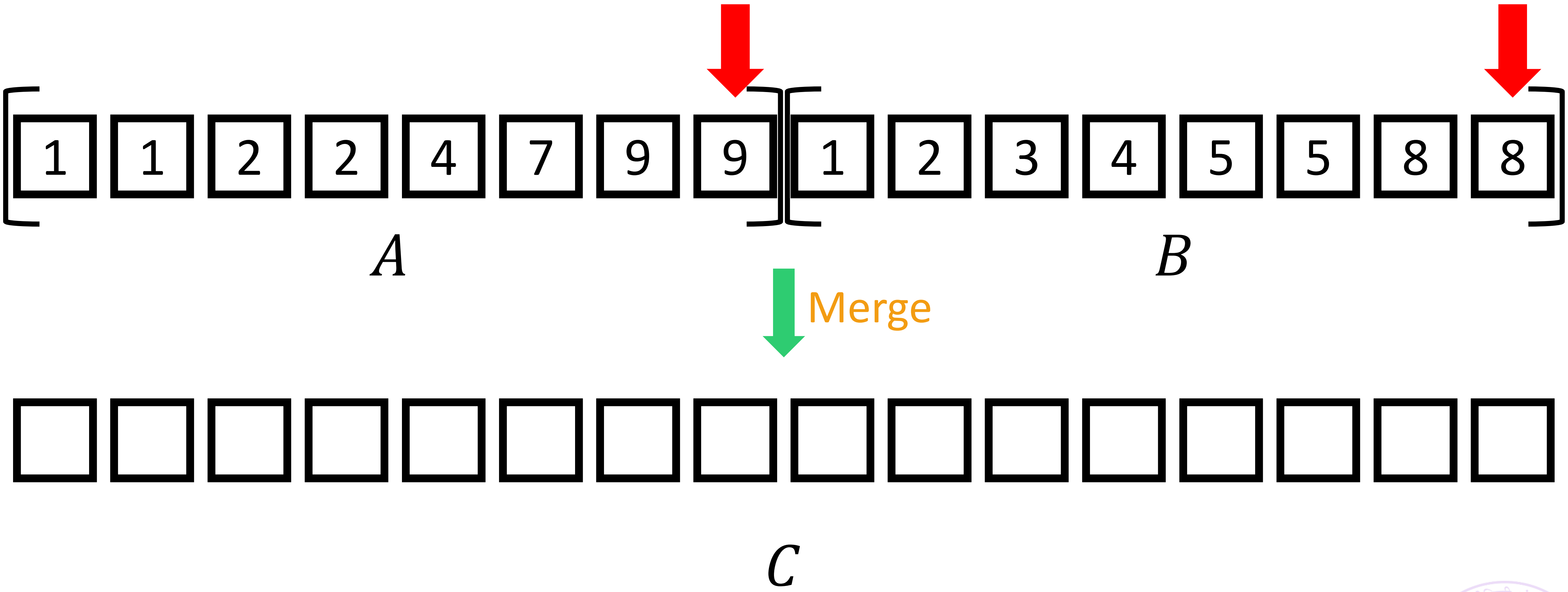
1	2	3	4	5	5	8	8

B

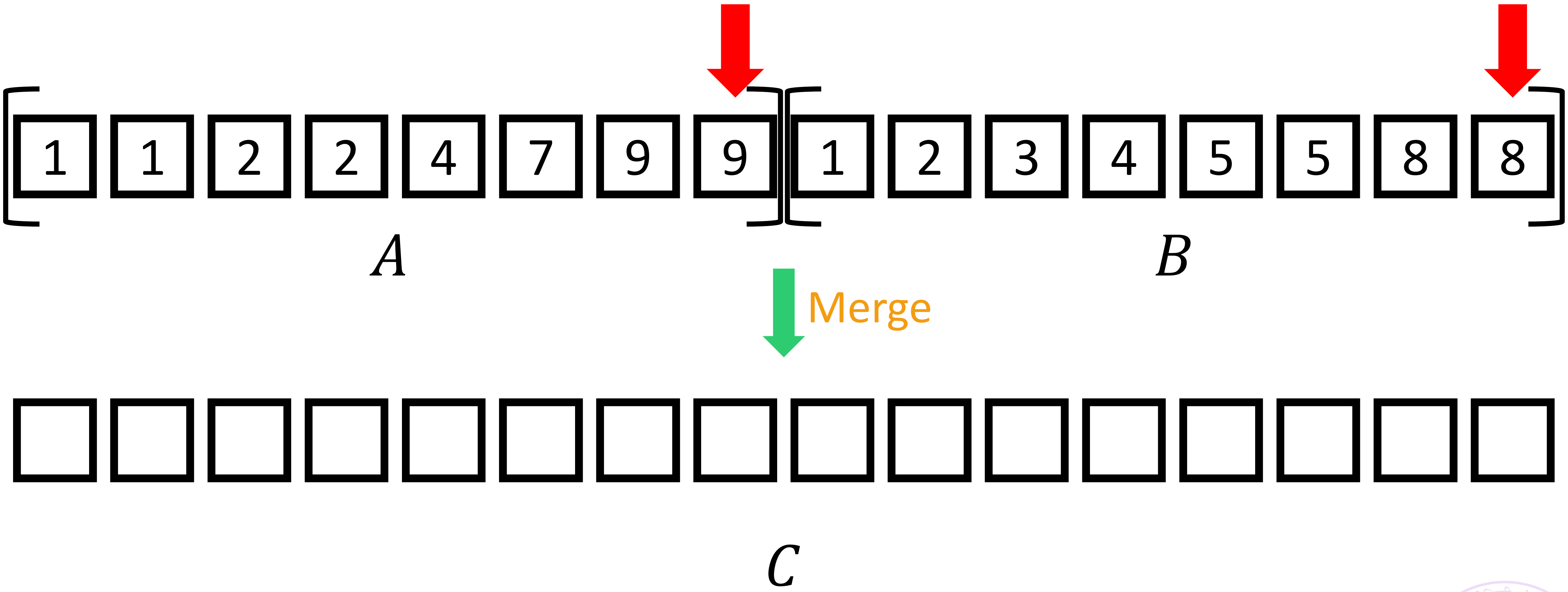
The Merge Operation



The Merge Operation

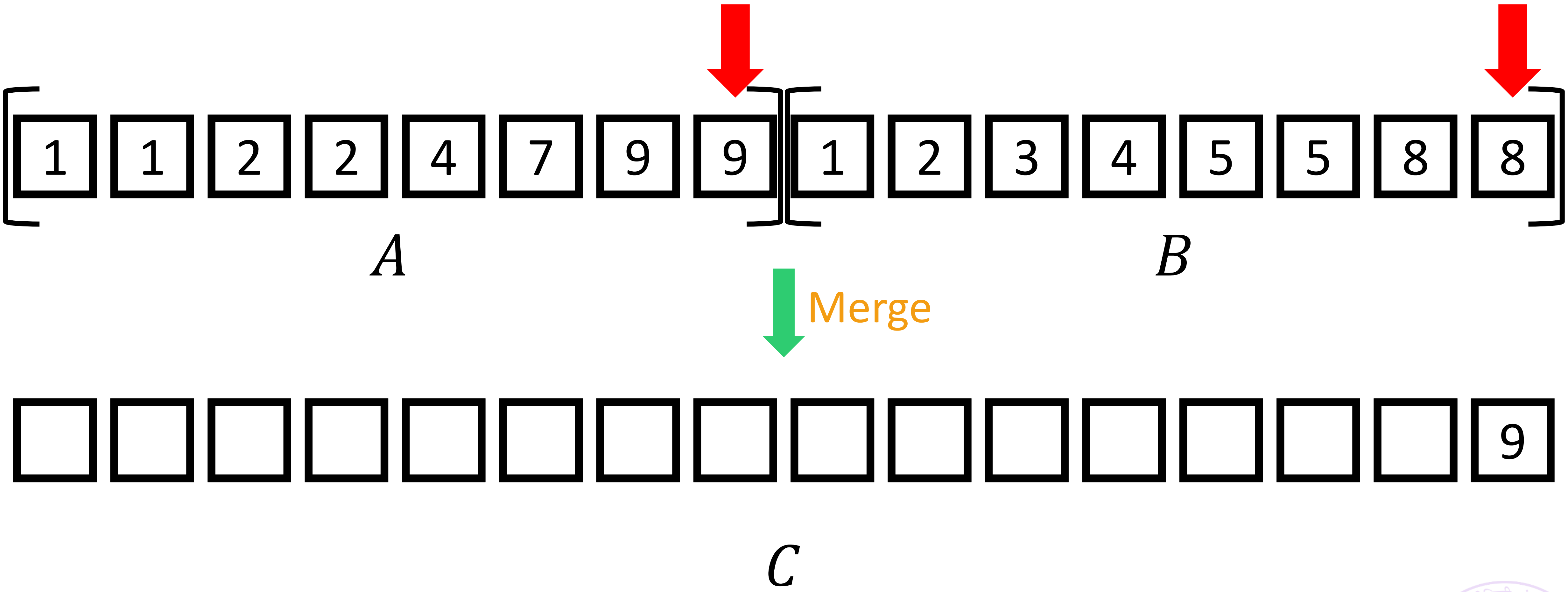


The Merge Operation



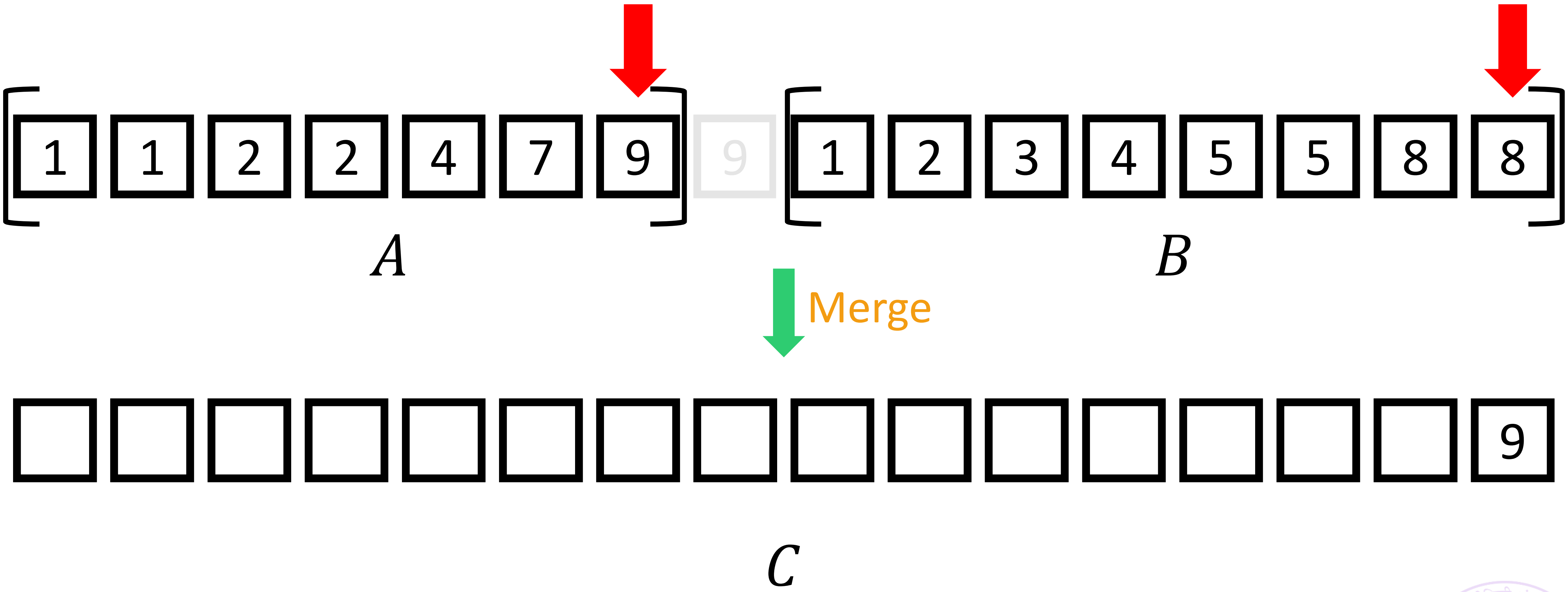
9 is larger: A wins

The Merge Operation

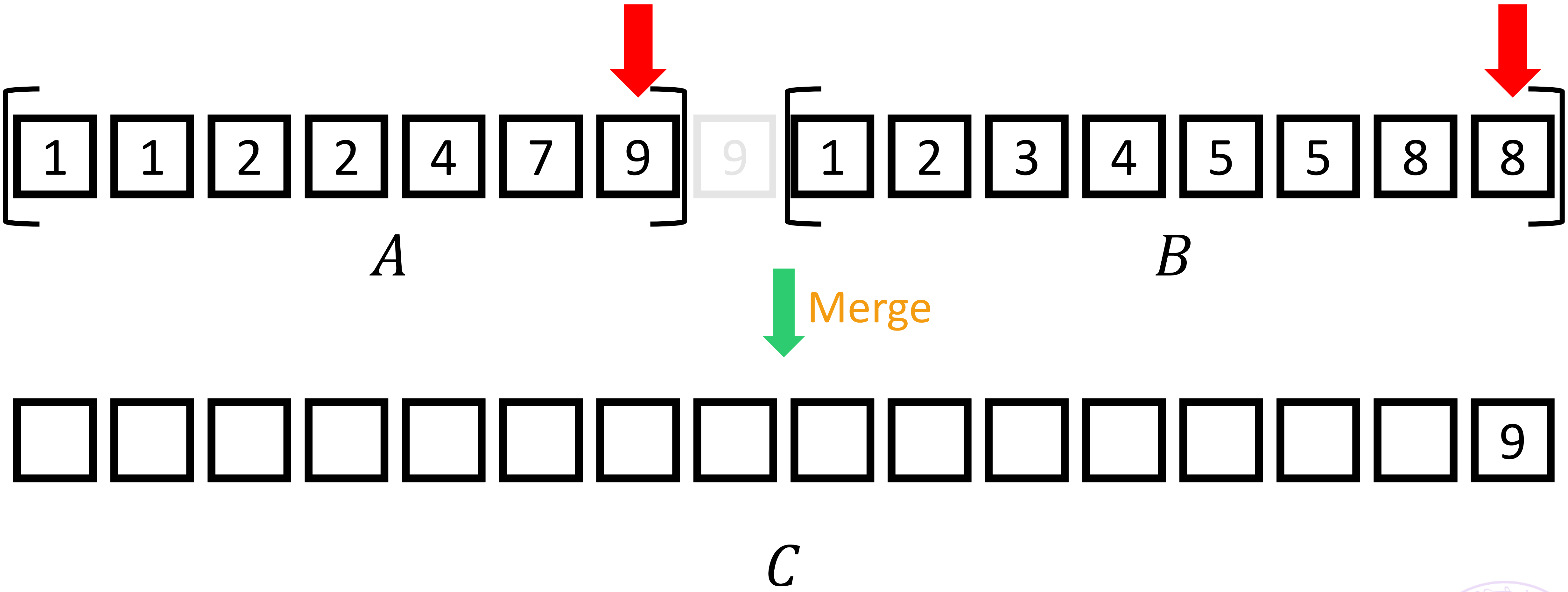


9 is larger: A wins

The Merge Operation

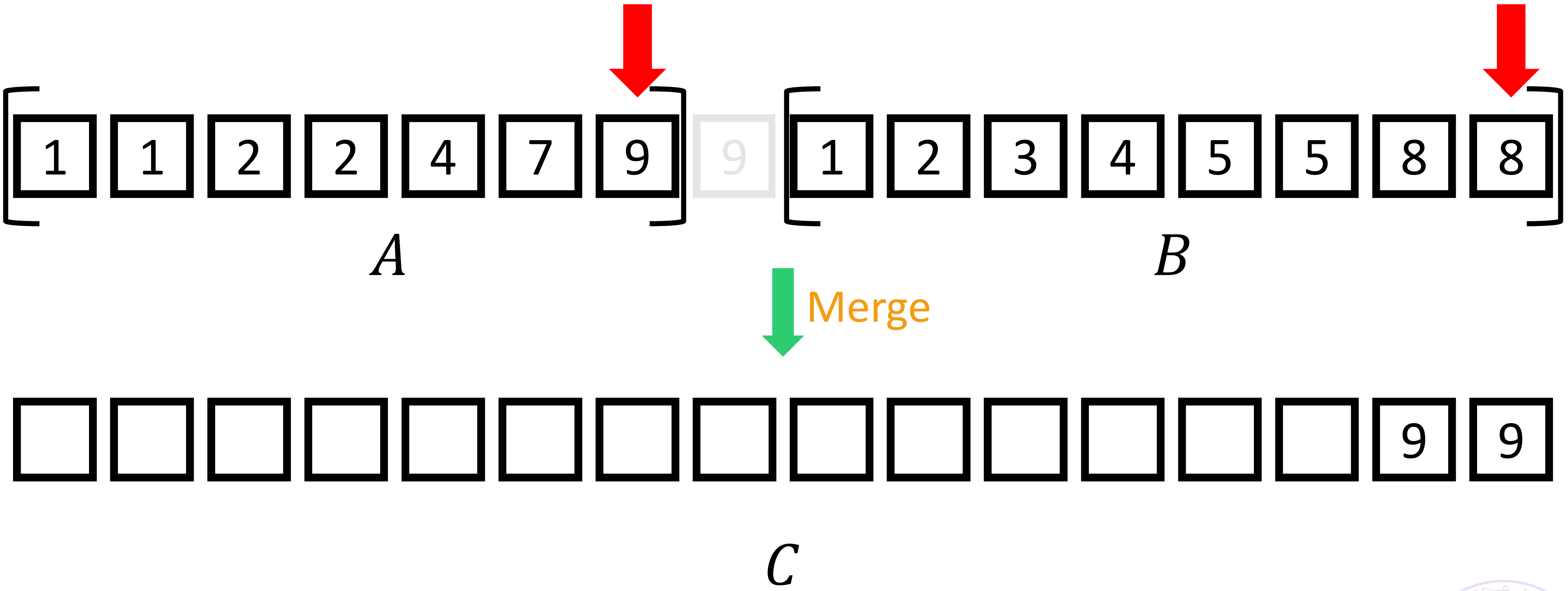


The Merge Operation



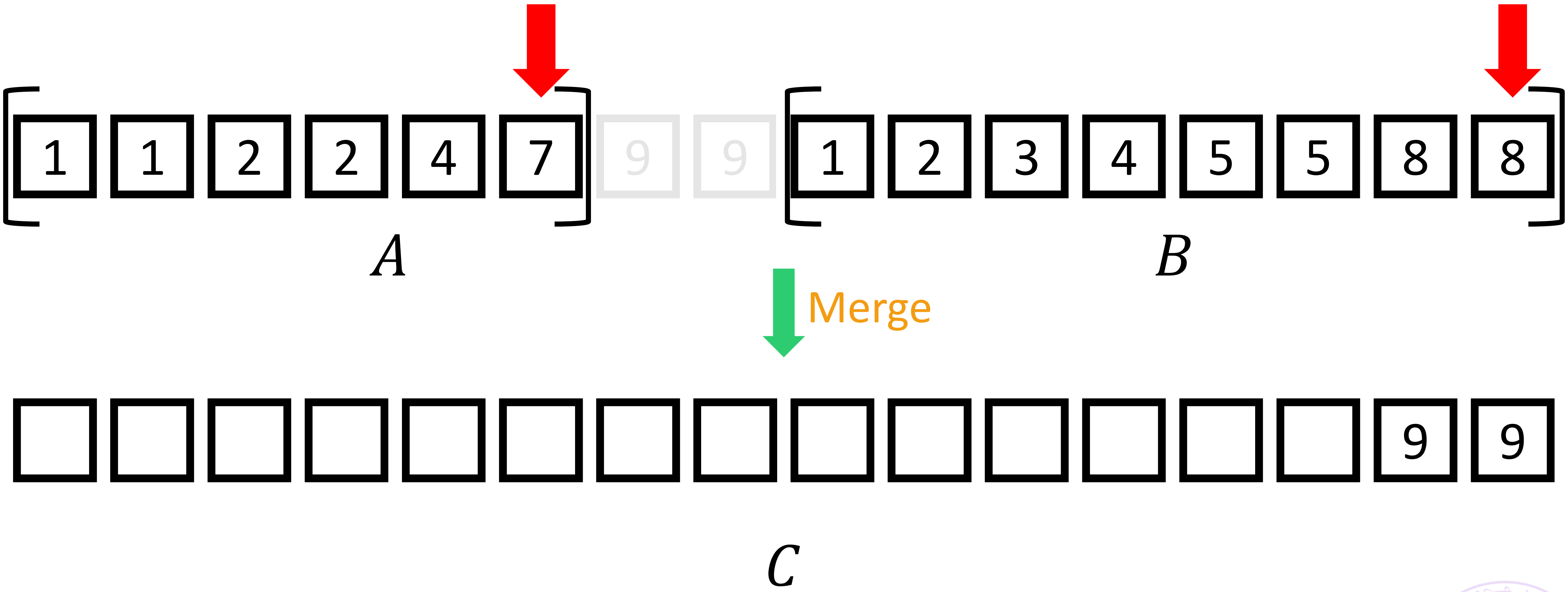
9 is larger: A wins again

The Merge Operation

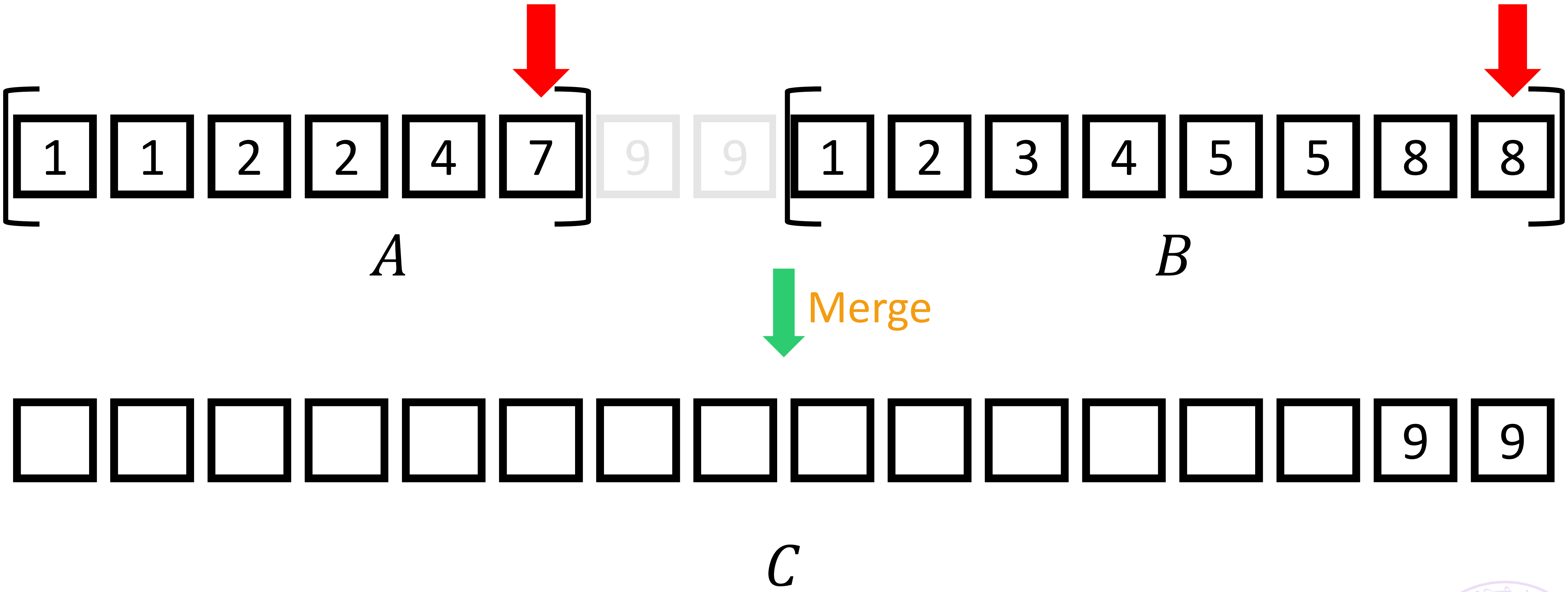


9 is larger: A wins again

The Merge Operation

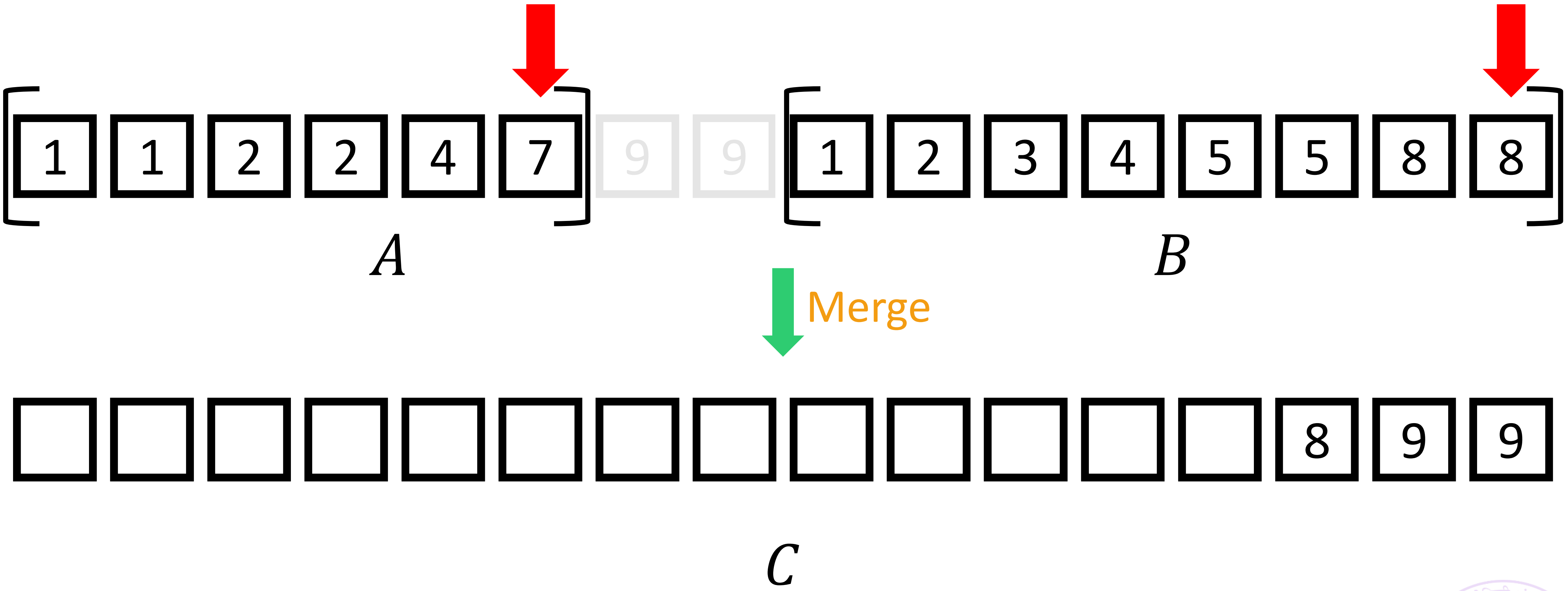


The Merge Operation



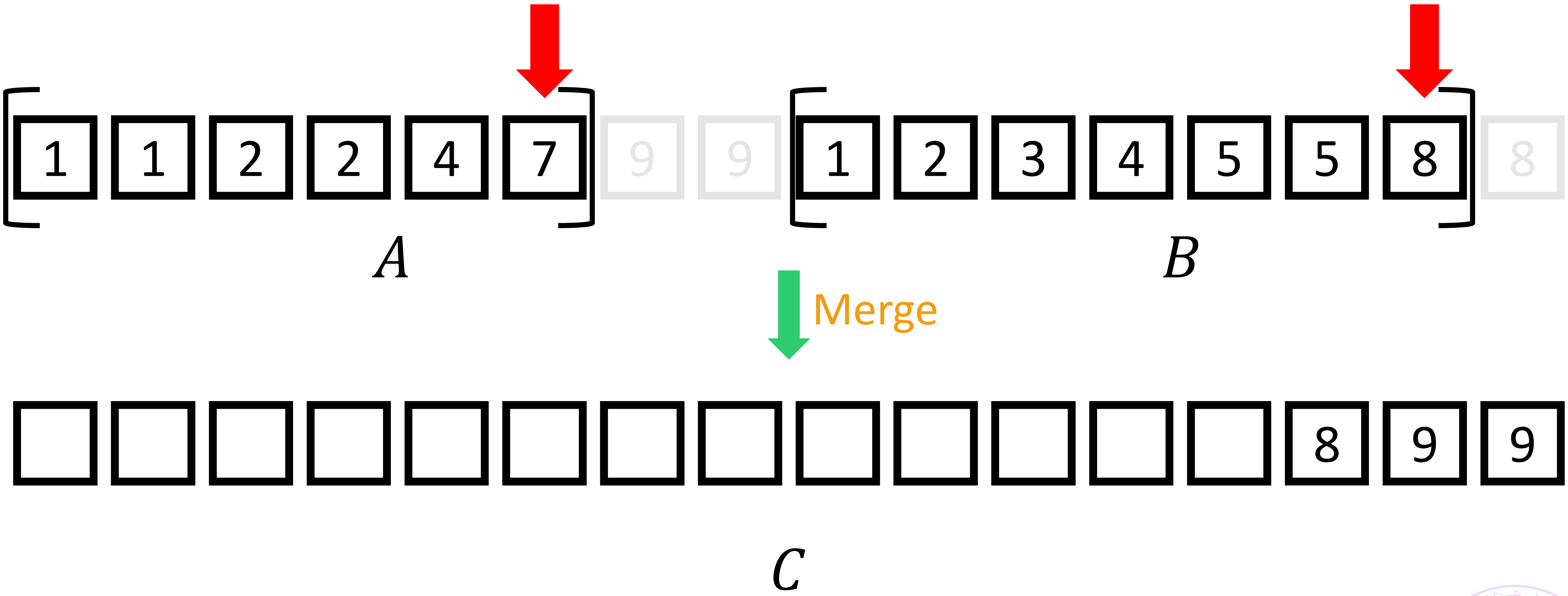
8 is larger: B wins

The Merge Operation

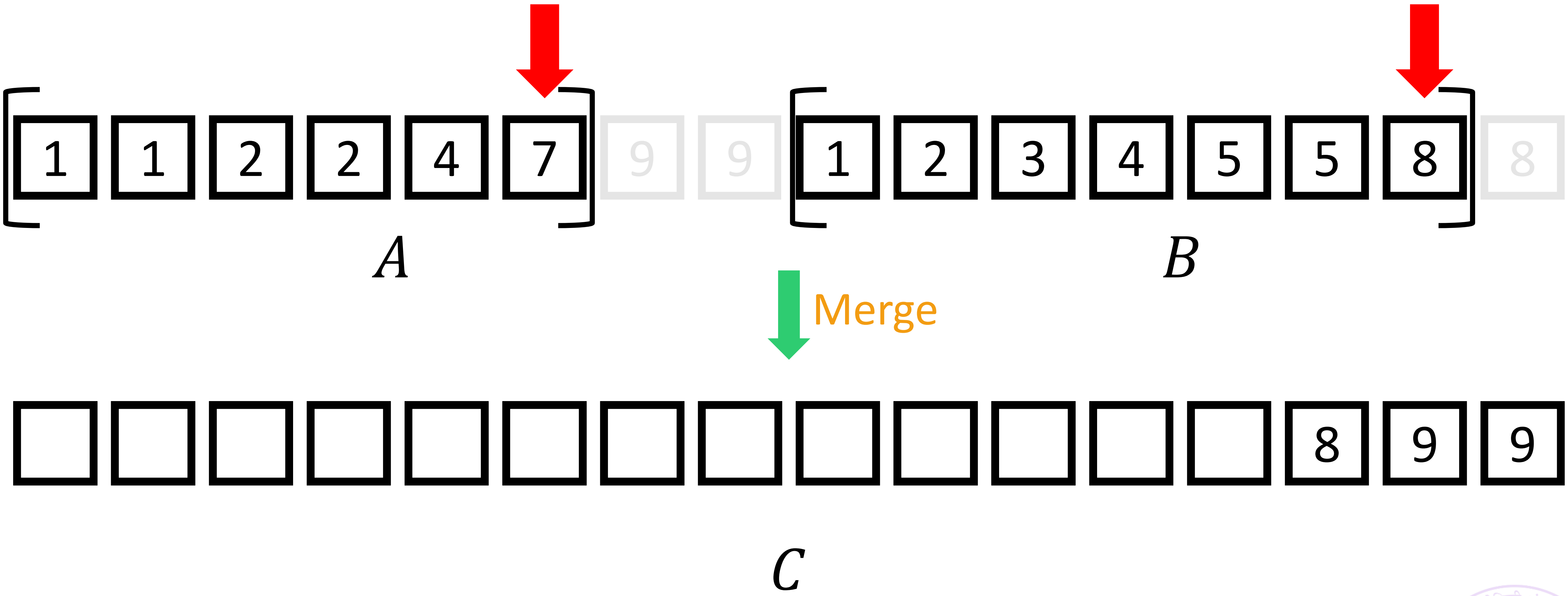


8 is larger: B wins

The Merge Operation

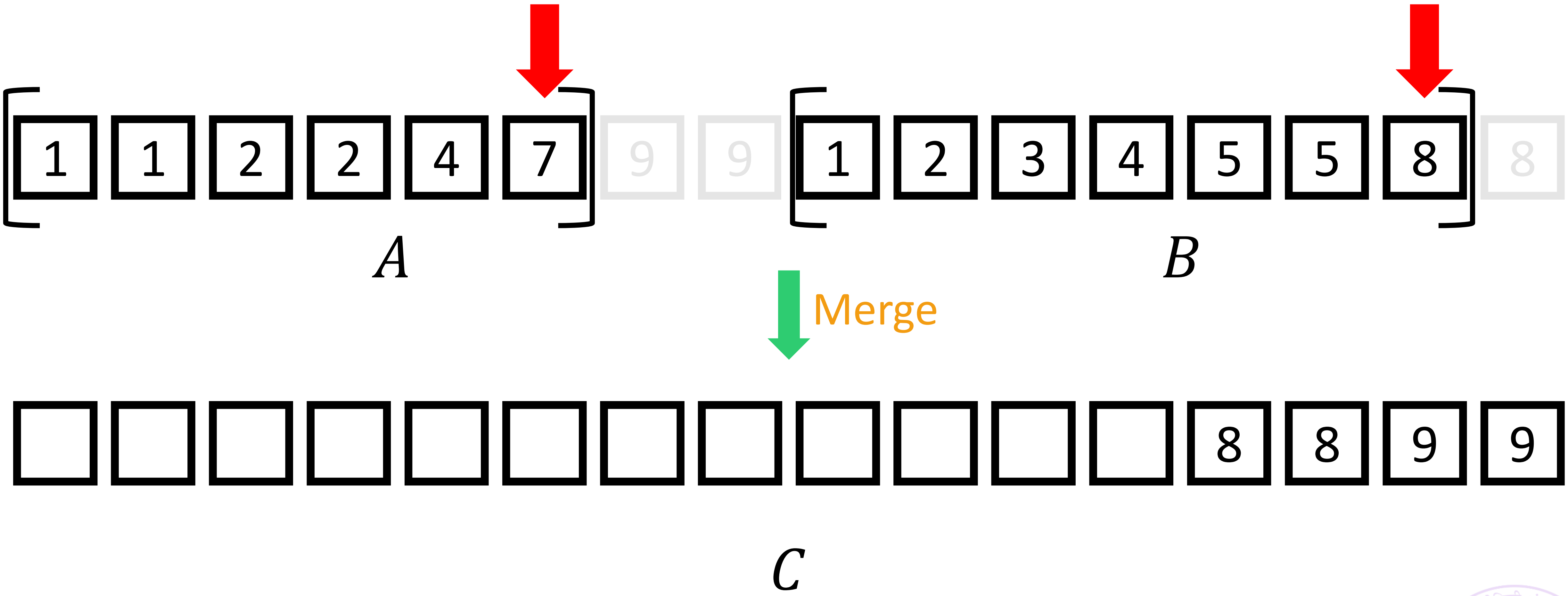


The Merge Operation



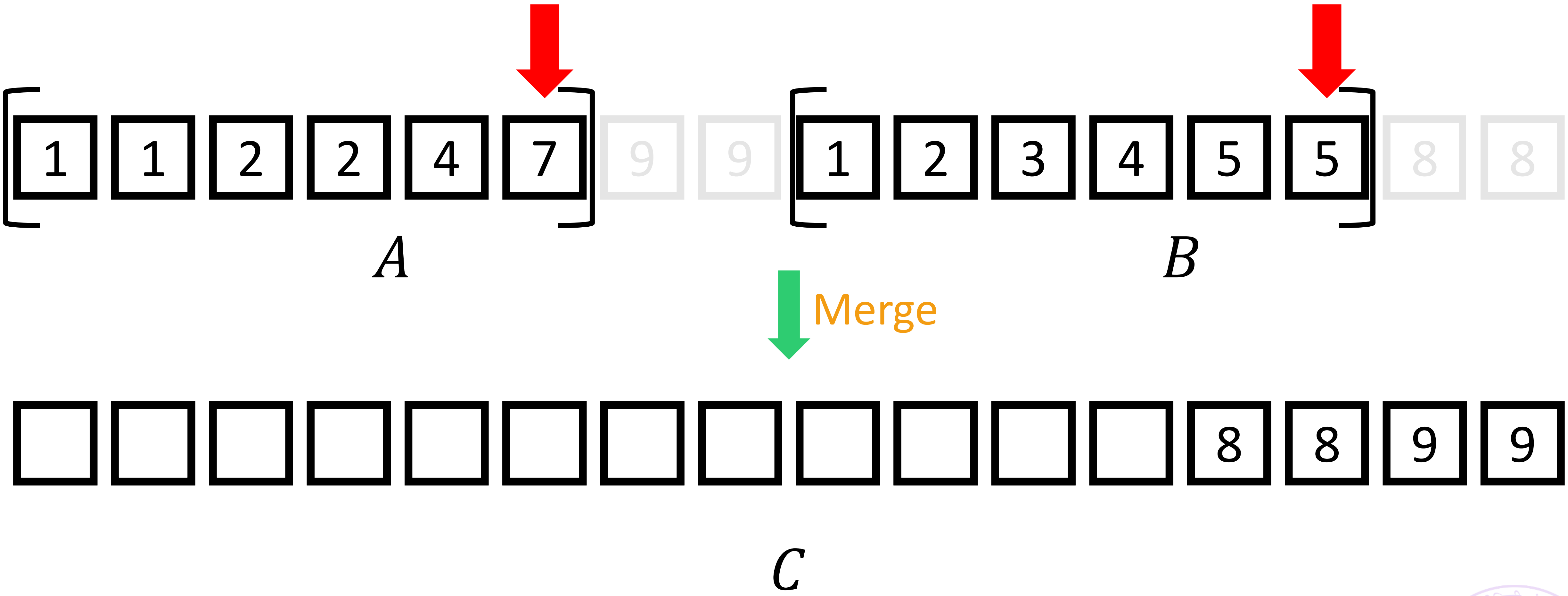
8 is larger: B wins again

The Merge Operation

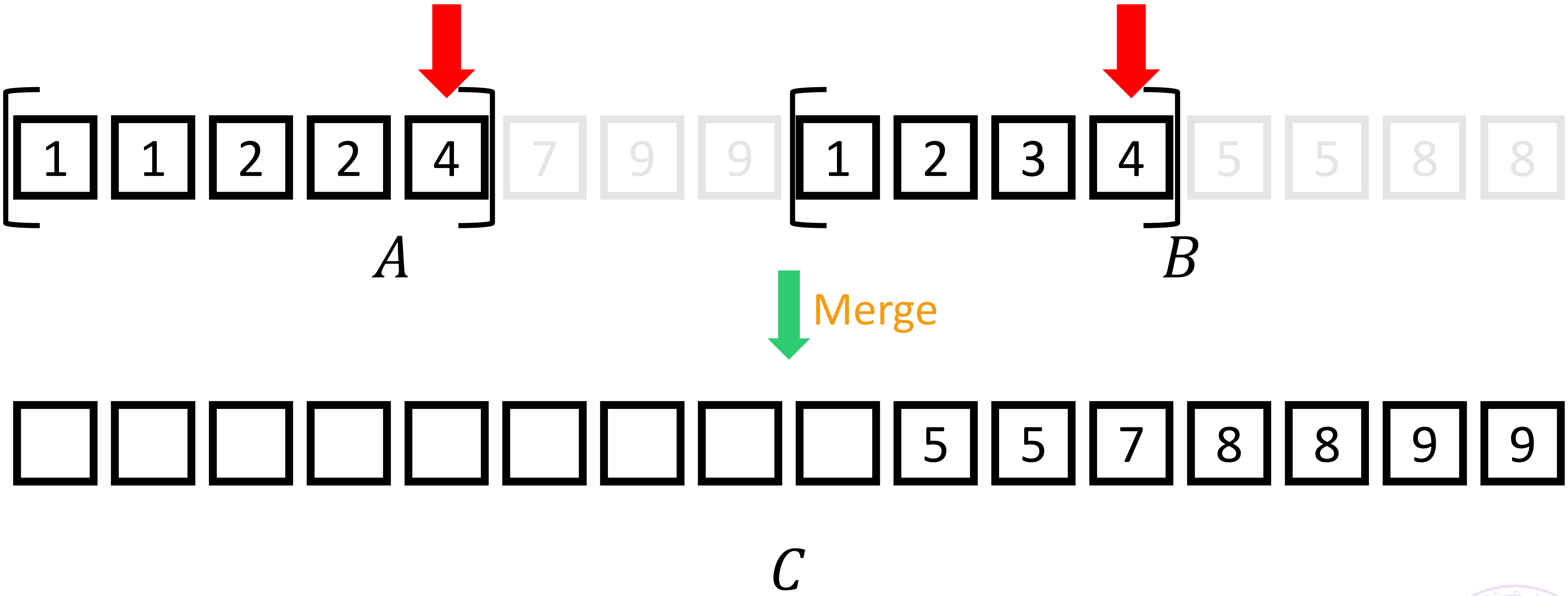


8 is larger: B wins again

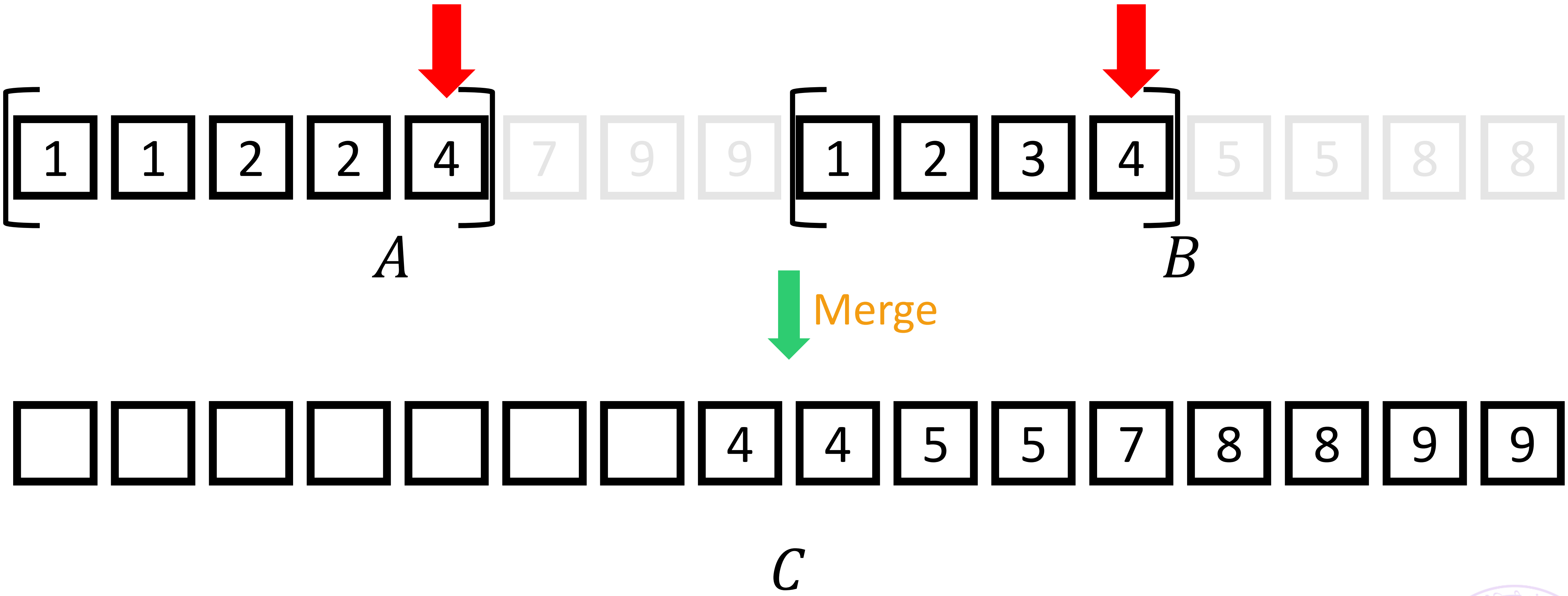
The Merge Operation



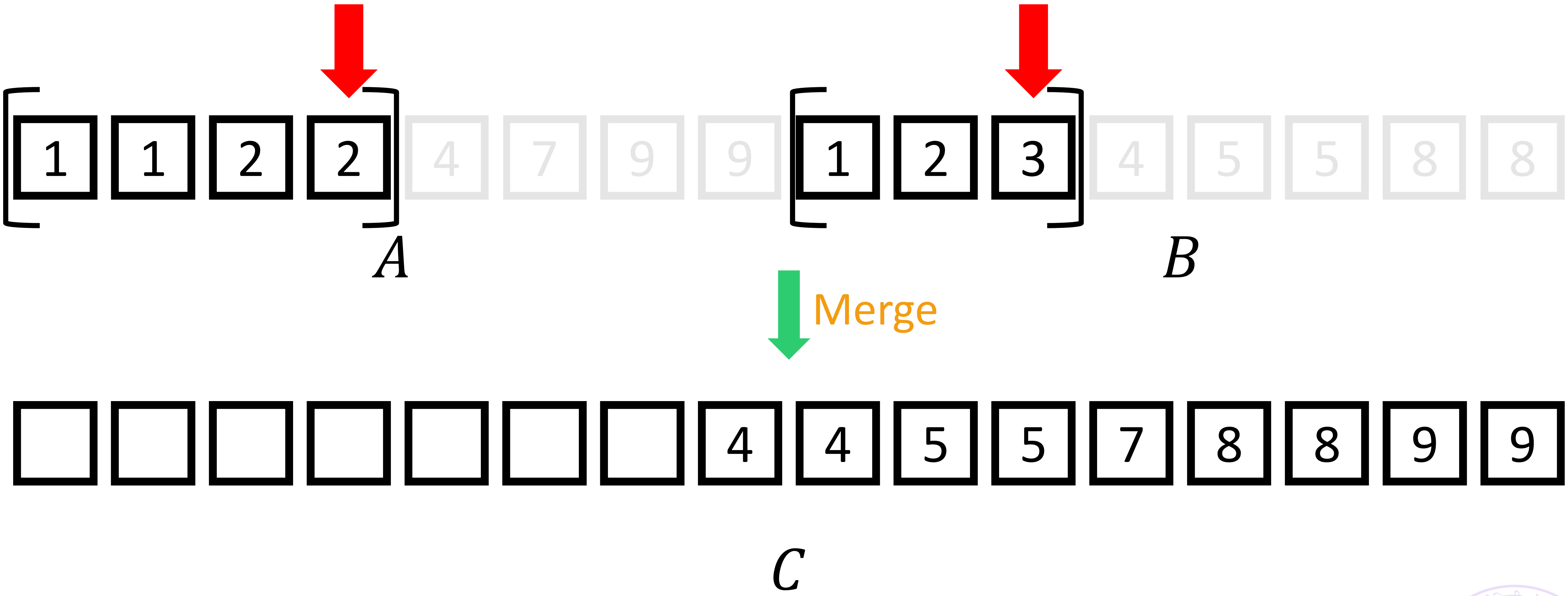
The Merge Operation



The Merge Operation



The Merge Operation



The Merge Operation

MERGE

1. Given: Sorted arrays a, b with M, N elements respectively.
2. Let int $c[M+N], R_1 \leftarrow M-1, R_2 \leftarrow N-1$
3. For $k = M+N-1; k \geq 0; k$--
   1. If $R_1 < 0$ then $c[k] = b[R_2]; R_2$-- *//We exhausted a*
   2. Elseif $R_2 < 0$ then $c[k] = a[R_1]; R_1$-- *//We exhausted b*
   3. Elseif $a[R_1] \geq b[R_2]$ then $c[k] = a[R_1]; R_1$-- *//Both active, a wins*
   4. Else $c[k] = b[R_2]; R_2$-- *//Both active, b wins*
4. Return c

The Merge Operation

MERGE

1. Given: Sorted arrays a, b with M, N elements respectively.
2. Let int $c[M+N], R_1 \leftarrow M-1, R_2 \leftarrow N-1$
3. For $k = M+N-1; k \geq 0; k$--
   1. If $R_1 < 0$ then $c[k] = b[R_2]; R_2$-- *//We exhausted a*
   2. Elseif $R_2 < 0$ then $c[k] = a[R_1]; R_1$-- *//We exhausted b*
   3. Elseif $a[R_1] \geq b[R_2]$ then $c[k] = a[R_1]; R_1$-- *//Both active, a wins*
   4. Else $c[k] = b[R_2]; R_2$-- *//Both active, b wins*
4. Return c

Exercise: show that merging two arrays of size M, N takes time $\mathcal{O}(M+N)$

The Merge Operation

MERGE

1. Given: Sorted arrays a, b with M, N elements respectively.
2. Let int $c[M+N], R_1 \leftarrow M-1, R_2 \leftarrow N-1$
3. For $k = M+N-1; k \geq 0; k$--
   1. If $R_1 < 0$ then $c[k] = b[R_2]; R_2$-- *//We exhausted a*
   2. Elseif $R_2 < 0$ then $c[k] = a[R_1]; R_1$-- *//We exhausted b*
   3. Elseif $a[R_1] \geq b[R_2]$ then $c[k] = a[R_1]; R_1$-- *//Both active, a wins*
   4. Else $c[k] = b[R_2]; R_2$-- *//Both active, b wins*
4. Return c

Exercise: write an *in-place* version – cant allocate $c[M+N]$

Exercise: show that merging two arrays of size M, N takes time $\mathcal{O}(M+N)$

Sorting Algorithms

Quick Sort

Quick Sort

- Very popular sorting algorithm – try this before anything else
- $\mathcal{O}(N \log N)$ time complexity but in practice faster than merge sort
- Merge sort lazily divides the array into two equal halves, sorts the halves recursively and then spends time merging them
- Quick sort is more careful in splitting the array so that no need for merging once the subarrays are sorted!
- Based on a cool trick known as *partitioning*
- Analysis of quick sort is much more advanced – in worst case quicksort takes $\mathcal{O}(N^2)$ time but this happens very very rarely.
- On average quicksort enjoys $\mathcal{O}(N \log N)$ time complexity

The Partition Technique

- Given array int $a[N]$ and any element of the array p (called pivot)
- Create a new array int $b[N]$ which is arranged as follows

$$[\text{elements of } a \leq p, \qquad p, \qquad \text{elements of } a \geq p]$$

The Partition Technique

- Given array int $a[N]$ and any element of the array p (called pivot)
- Create a new array int $b[N]$ which is arranged as follows

$$[\text{elements of } a \leq p, \qquad p, \qquad \text{elements of } a \geq p]$$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

The Partition Technique

- Given array int $a[N]$ and any element of the array p (called pivot)
- Create a new array int $b[N]$ which is arranged as follows

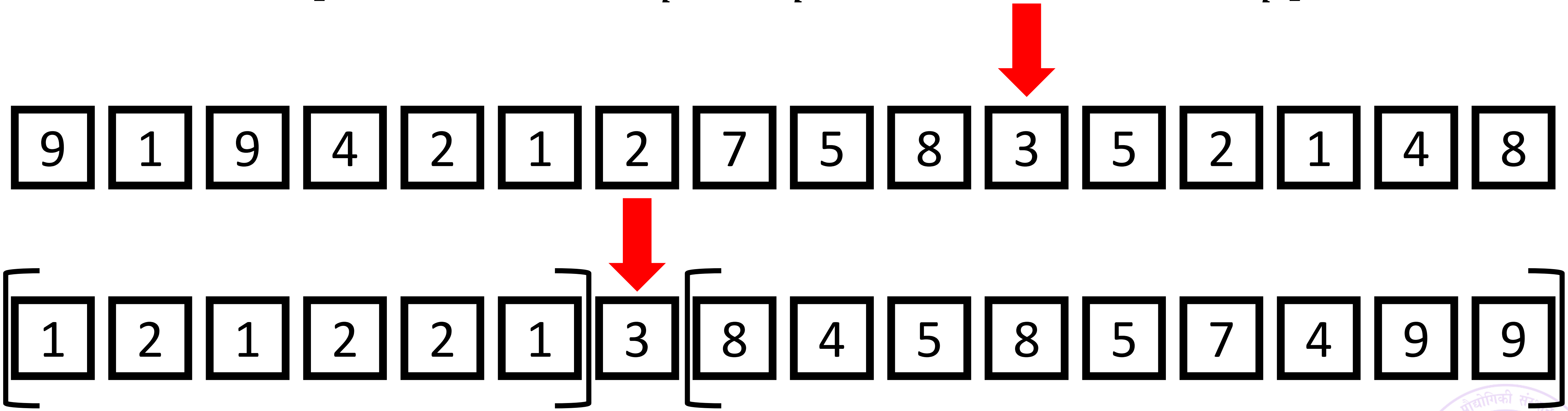
$$[\text{elements of } a \leq p, \qquad p, \qquad \text{elements of } a \geq p]$$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

The Partition Technique

- Given array int $a[N]$ and any element of the array p (called pivot)
- Create a new array int $b[N]$ which is arranged as follows

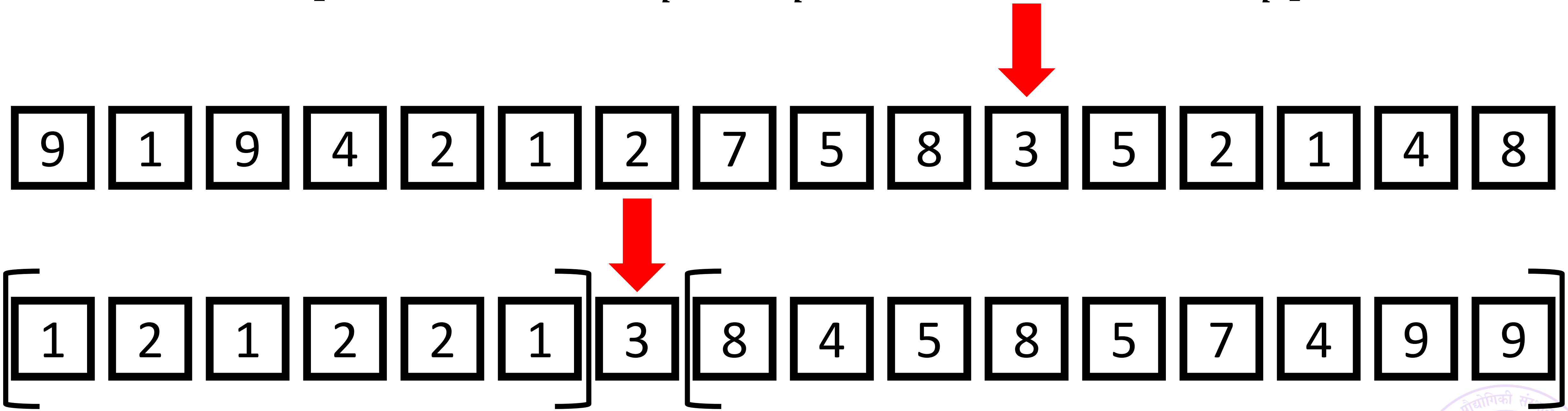
$$[\text{elements of } a \leq p, \qquad p, \qquad \text{elements of } a \geq p]$$



The Partition Technique

- Given array int $a[N]$ and any element of the array p (called pivot)
- Create a new array int $b[N]$ which is arranged as follows

$$[\text{elements of } a \leq p, \qquad p, \qquad \text{elements of } a \geq p]$$



- Notice that left and right halves are not sorted yet! 😐
- Also, the two halves are not balanced (of same size) either 😐

Quick Sort

$$\underbrace{[\,\boxed{1}\ \boxed{2}\ \boxed{1}\ \boxed{2}\ \boxed{2}\ \boxed{1}\,]}_{L}\ \boxed{3}\ \underbrace{[\,\boxed{8}\ \boxed{4}\ \boxed{5}\ \boxed{8}\ \boxed{5}\ \boxed{7}\ \boxed{4}\ \boxed{9}\ \boxed{9}\,]}_{R}$$

- Notice that even though the subarrays L, R not sorted, every element of L is smaller than or equal to every element of R
- This means that if we sort L, R recursively, no need to merge ☺
- Key to quicksort's success – partition and recursively sort!
- Will discuss a partition algorithm that ensures a stricter condition
  $$[\text{elements of } a < p, \qquad \text{all instances of } p, \qquad \text{elements of } a > p]$$
- However, our algorithm will use extra memory
- Time complexity analysis of quicksort beyond scope of ESC101

Quick Sort

QUICKSORT

1. Given: Array a with N elements
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $p \leftarrow$ CHOOSEPIVOT(a) *//Choose a pivot value*
4. Let $(b, i) \leftarrow$ PARTITION(a, p) *//Partition along chosen pivot*
5. QUICKSORT($b[0{:}\,i-1]$) *//Sort the left half*
6. QUICKSORT($b[i+1, N-1]$) *//Sort the right half*
7. Return b

Quick Sort

QUICKSORT

i is the new location of the pivot element

1. Given: Array a with
2. If $N < 2$ return a … singleton array is sorted
3. Let $p \leftarrow$ CHOOSEPIVOT(a) *//Choose a pivot value*
4. Let $(b, i) \leftarrow$ PARTITION(a, p) *//Partition along chosen pivot*
5. QUICKSORT$(b[0{:}\, i-1])$ *//Sort the left half*
6. QUICKSORT$(b[i+1, N-1])$ *//Sort the right half*
7. Return b

Quick Sort

QUICKSORT

i is the new location of the pivot element

1. Given: Array a with [illegible]
2. If $N < 2$ return a [illegible] singleton array is sorted
3. Let $p \leftarrow$ CHOOSEPIVOT(a) *//Choose a pivot value*
4. Let $(b, i) \leftarrow$ PARTITION(a, p) *//Partition along chosen pivot*
5. QUICKSORT$(b[0{:}\, i - 1])$ *//Sort the left half*
6. QUICKSORT$(b[i + 1, N - 1])$ *//Sort the right half*
7. Return b

Common choices for pivot value

- $a[0]$ or $a[N - 1]$ i.e. end elements
- $a[i]$ for $i \sim \text{random}(N)$ i.e. a random element
- MEDIAN(a) i.e. median element of the array

Quick Sort

QUICKSORT

i is the new location of the pivot element

1. Given: Array a with
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $p \leftarrow$ CHOOSEPIVOT(a) *//Choose a pivot value*
4. Let $(b, i) \leftarrow$ PARTITION(a, p) *//Partition along chosen pivot*
5. QUICKSORT$(b[0{:}\, i-1])$ *//Sort the left half*
6. QUICKSORT$(b[i+1, N-1])$ *//Sort the right half*
7. Return b

Most popular, inexpensive

Common choices for pivot value
- $a[0]$ or $a[N-1]$ i.e. end elements
- $a[i]$ for $i \sim \text{random}(N)$ i.e. a random element
- MEDIAN(a) i.e. median element of the array

Quick Sort

QUICKSORT

i is the new location of the pivot element

1. Given: Array a with ...
2. If $N < 2$ return a *//An empty or singleton array is sorted*
3. Let $p \leftarrow$ CHOOSEPIVOT(a) *//Choose a pivot value*
4. Let $(b, i) \leftarrow$ PARTITION(a, p) *//Partition along chosen pivot*
5. QUICKSORT$(b[0: i-1])$ *//Sort the left half*
6. QUICKSORT$(b[i+1, N-1])$ *//Sort the right half*
7. Return b

Most popular, inexpensive

Also common, inexpensive

Common choices for pivot value

- $a[0]$ or $a[N-1]$ i.e. end elements
- $a[i]$ for $i \sim \text{random}(N)$ i.e. a random element
- MEDIAN(a) i.e. median element of the array

Quick Sort

QUICKSORT

1. Given: Array a with
2. If $N < 2$ return a ... singleton array is sorted
3. Let $p \leftarrow$ CHOOSEPIVOT(a) *//Choose a pivot value*
4. Let $(b, i) \leftarrow$ PARTITION(a, p) *//Partition along chosen pivot*
5. QUICKSORT$(b[0\colon i-1])$ *//Sort the left half*
6. QUICKSORT$(b[i+1, N-1])$ *//Sort the right half*
7. Return b

i is the new location of the pivot element

Common choices for pivot value

- $a[0]$ or $a[N-1]$ i.e. end elements
- $a[i]$ for $i \sim \mathrm{random}(N)$ i.e. a random element
- MEDIAN(a) i.e. median element of the array

Most popular, inexpensive

Also common, inexpensive

Ensures balanced partition but expensive

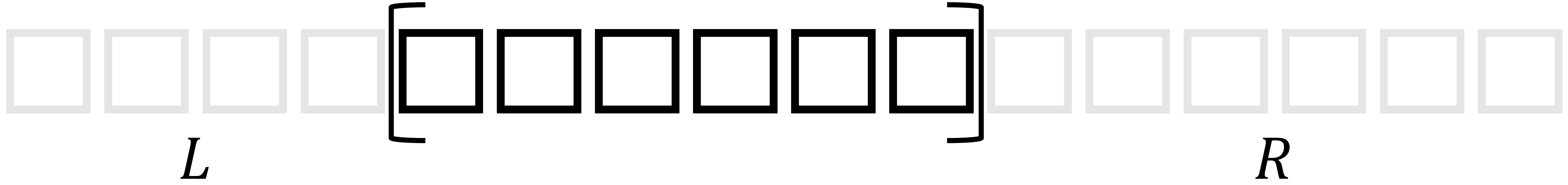
The Partition Procedure

The Partition Procedure

- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺

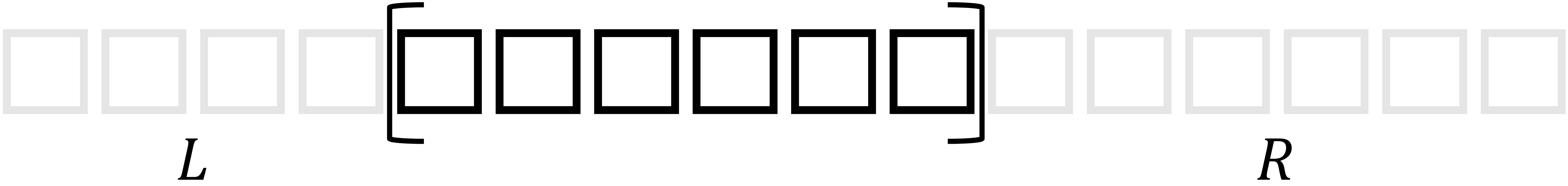
The Partition Procedure

- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺



The Partition Procedure

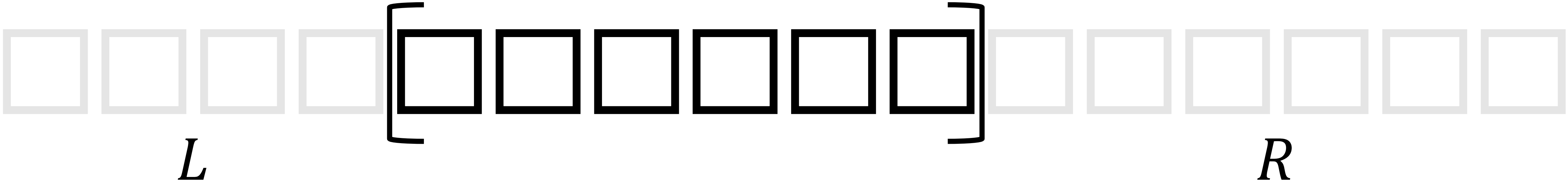
- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺



- Invariant: elements in the left inactive region are strictly less than the pivot, those in right invariant region strictly larger than pivot

The Partition Procedure

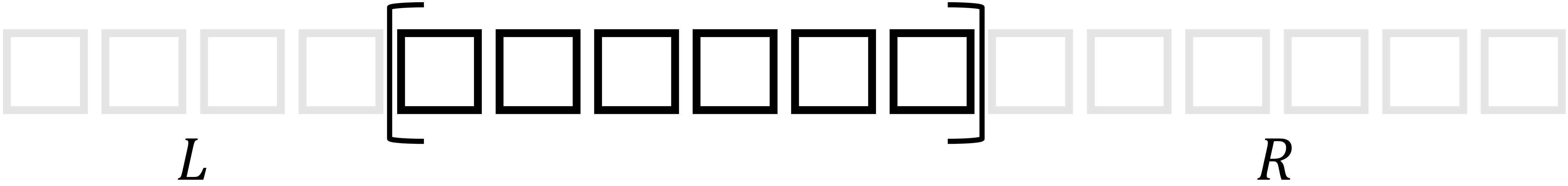
- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺



- Invariant: elements in the left inactive region are strictly less than the pivot, those in right invariant region strictly larger than pivot
- What about element(s) equal to the pivot – need to be careful

The Partition Procedure

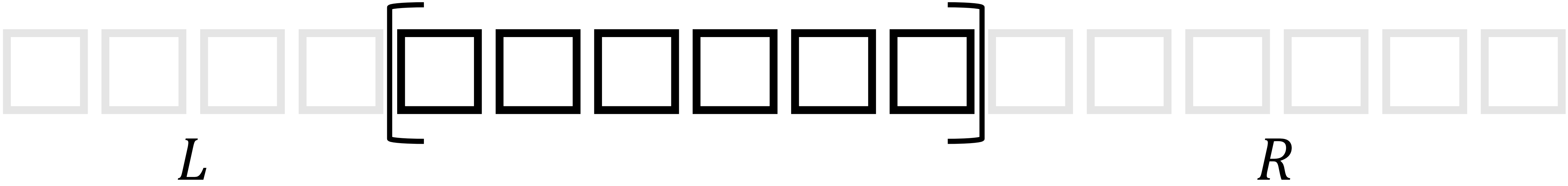
- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺



- Invariant: elements in the left inactive region are strictly less than the pivot, those in right invariant region strictly larger than pivot
- What about element(s) equal to the pivot – need to be careful
- Lets see a visualization of the partition procedure in action

The Partition Procedure

- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺



- Invariant: elements in the left inactive region are strictly less than the pivot, those in right invariant region strictly larger than pivot
- What about element(s) equal to the pivot – need to be careful
- Lets see a visualization of the partition procedure in action
- Note: these regions will be maintained on a separate array and not the original array – we will only take a simple left-to-right pass on the original array

The Partition Procedure

The Partition Procedure

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

The Partition Procedure

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

L R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

L R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

L R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

L R

$9 > 4$ i.e. belongs to R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

															9

L R

$9 > 4$ i.e. belongs to R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

															9

L R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

															9

L R

$1 < 4$ i.e. belongs to L

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1															9

L R

$1 < 4$ i.e. belongs to L

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1															9

L R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1															9

L R

$9 > 4$ i.e. belongs to R

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1														9	9

L R

$9 > 4$ i.e. belongs to R

The Partition Procedure

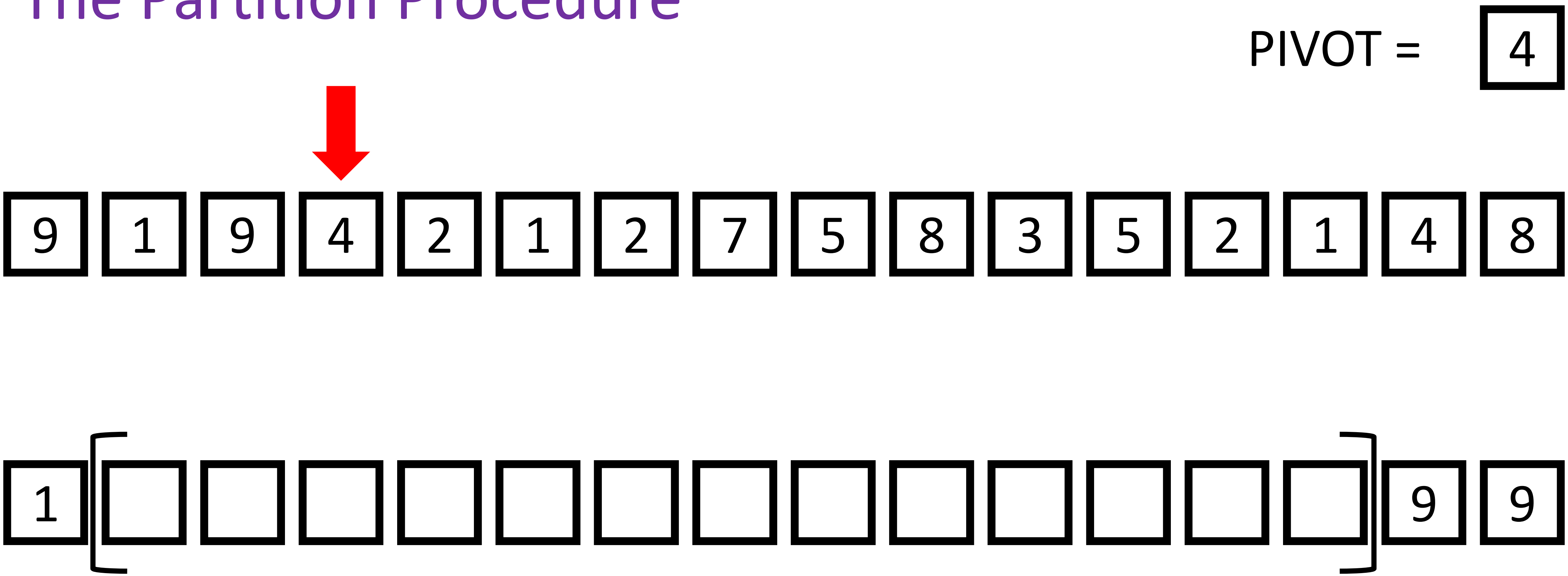
PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1														9	9

L R

The Partition Procedure



L R

Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Proc

We will insert all occurrences of the pivot element 4 after we are done with non-pivot elements

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

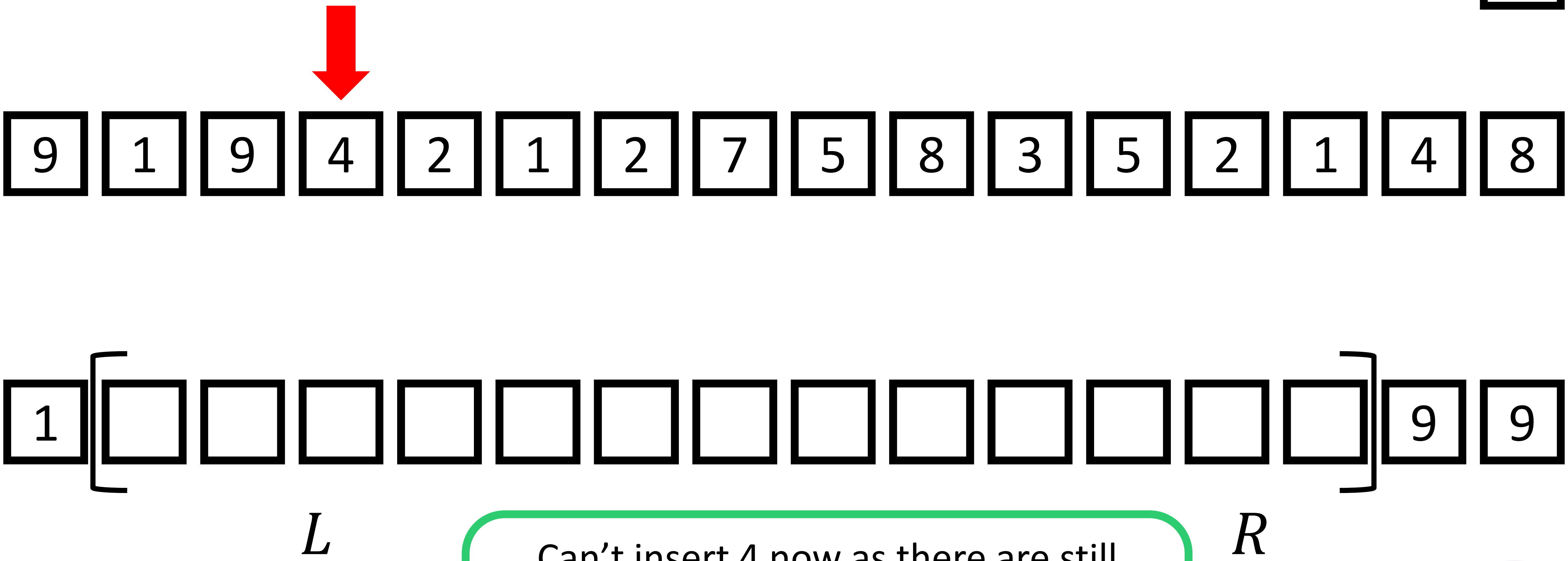
1														9	9

L R

Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

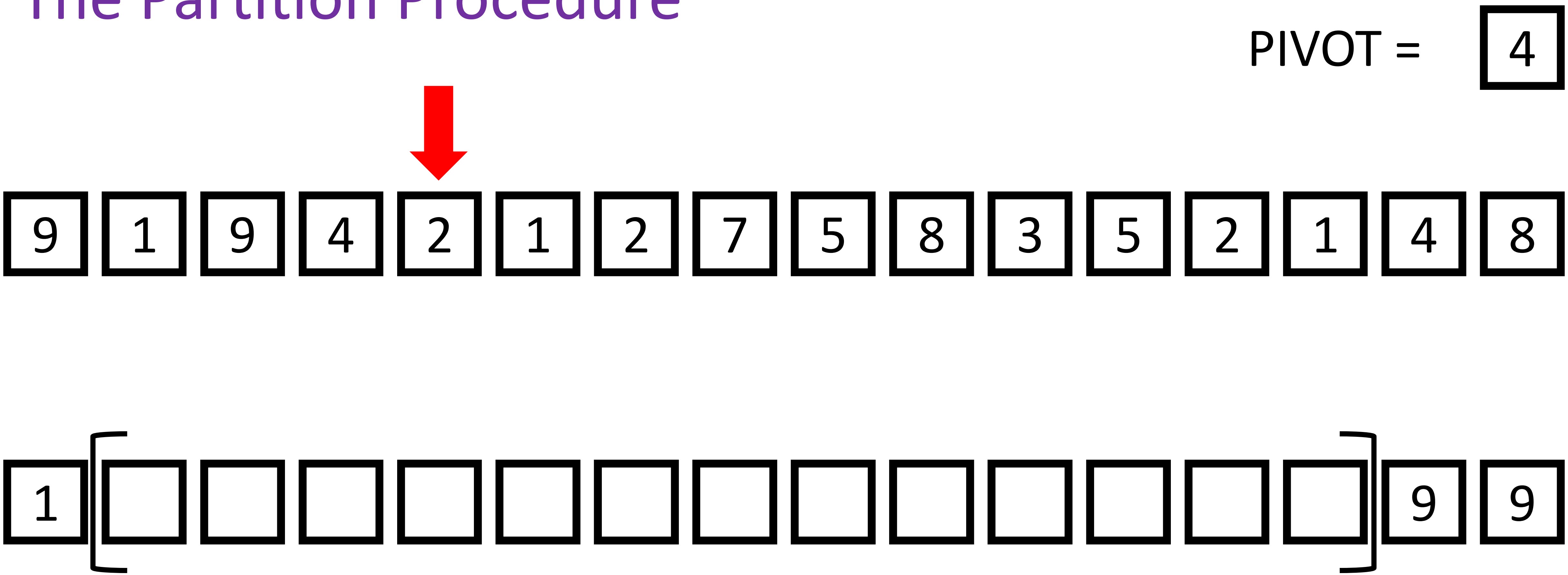
The Partition Procedure

PIVOT = $\boxed{4}$



Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

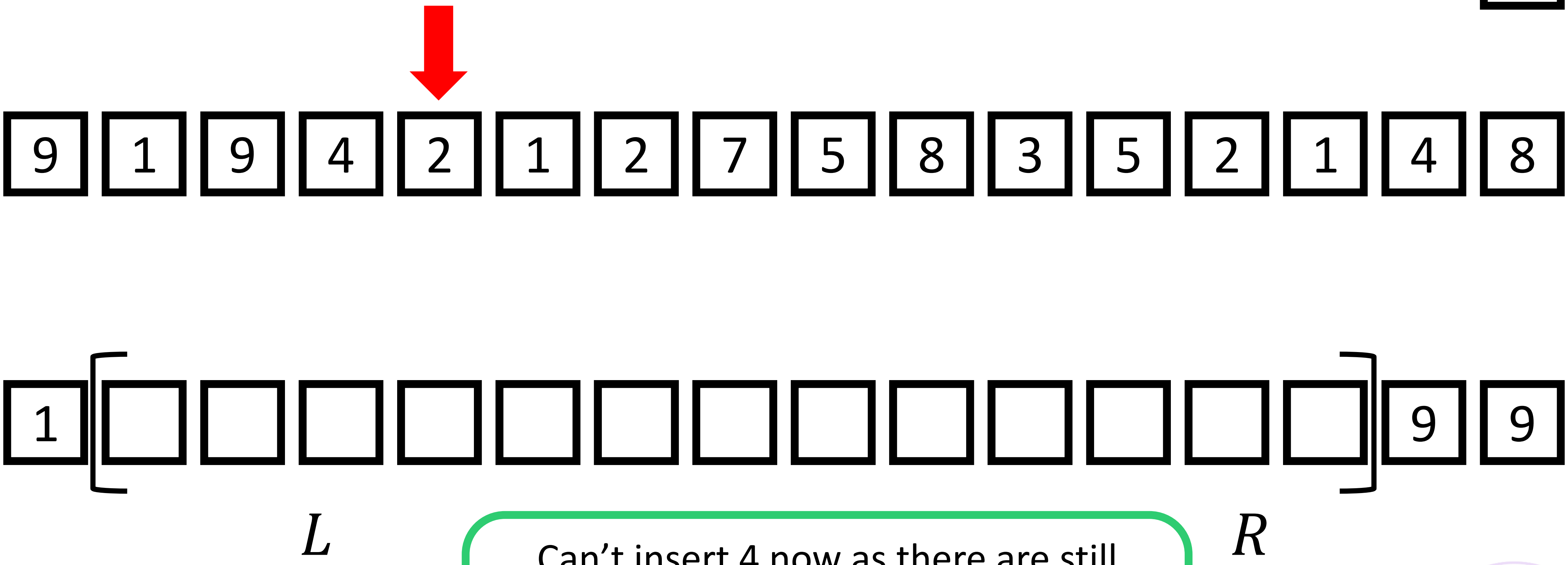


L R

Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = 4

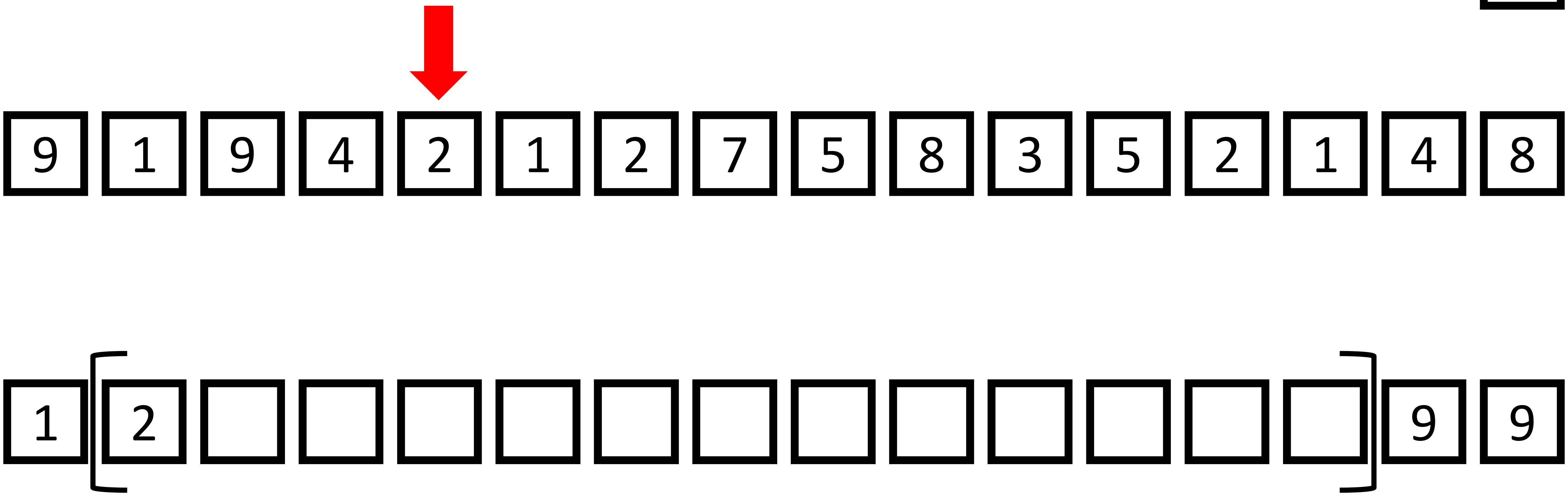


$2 < 4$ i.e. belongs to L

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$



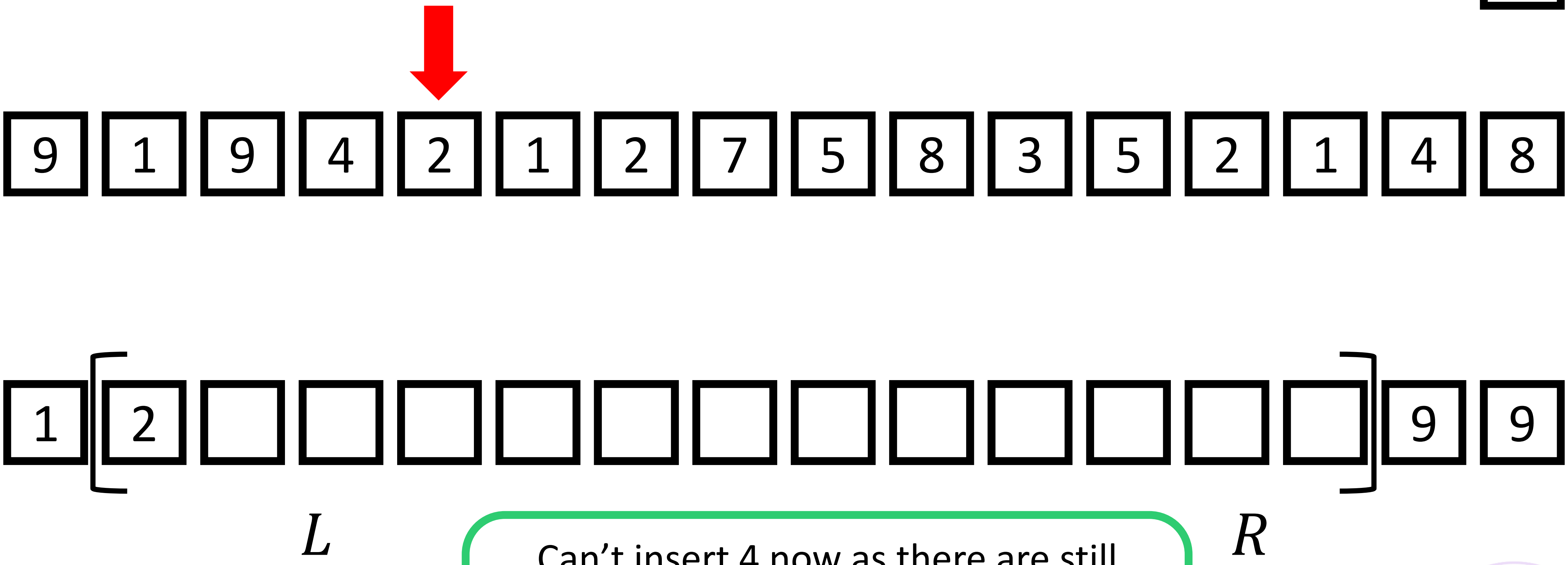
L R

$2 < 4$ i.e. belongs to L

Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$



Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

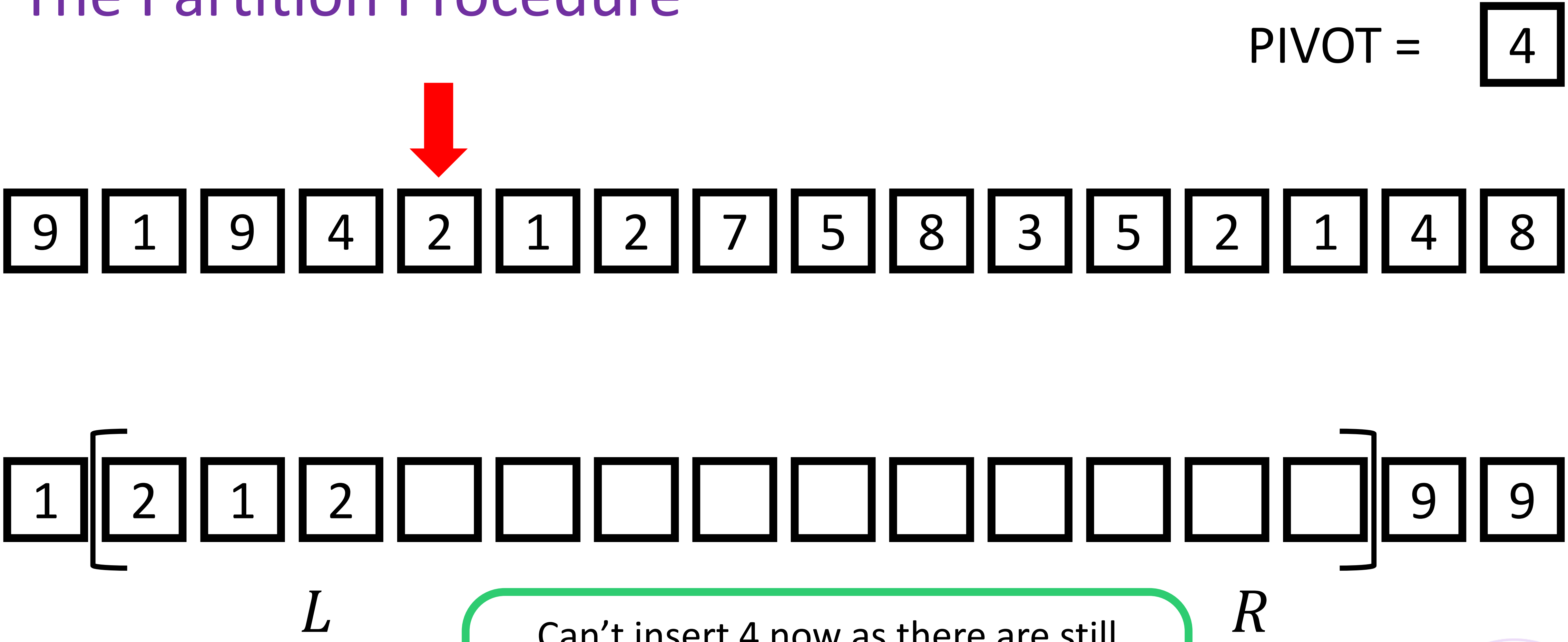
9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1												9	9

L R

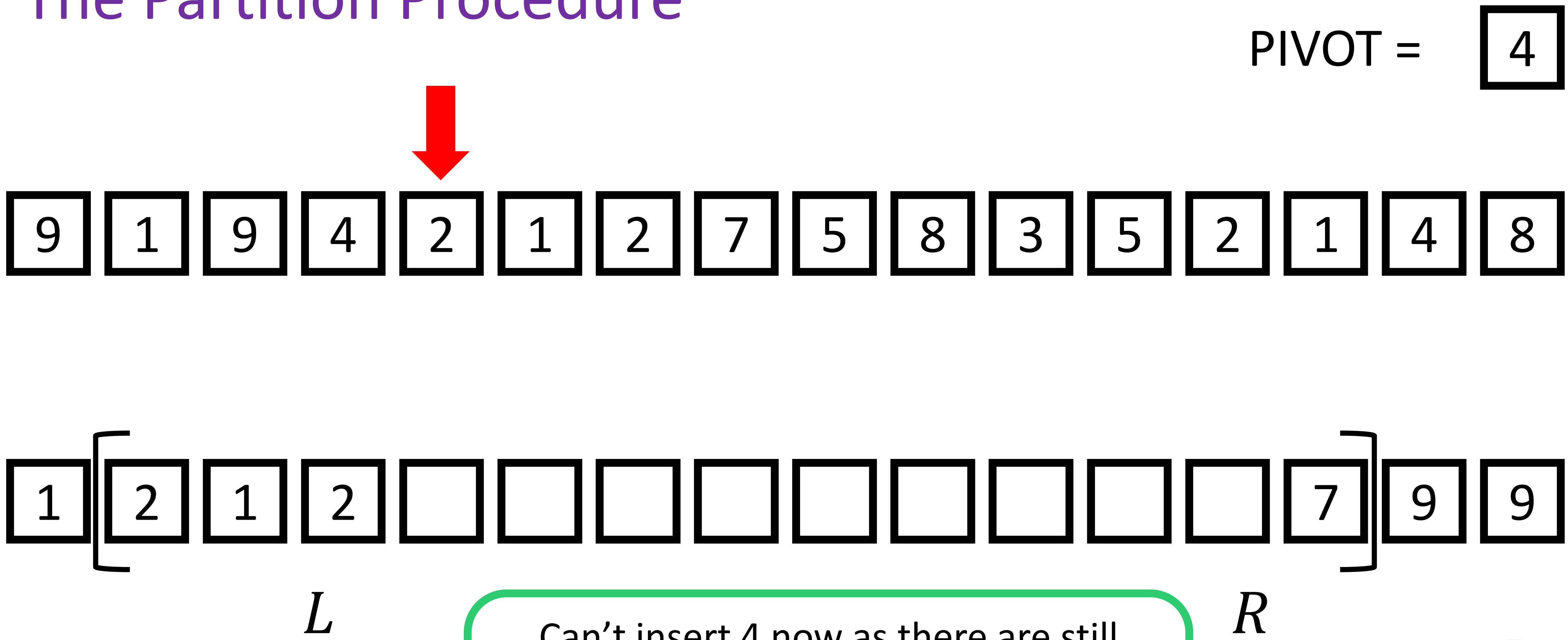
Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure



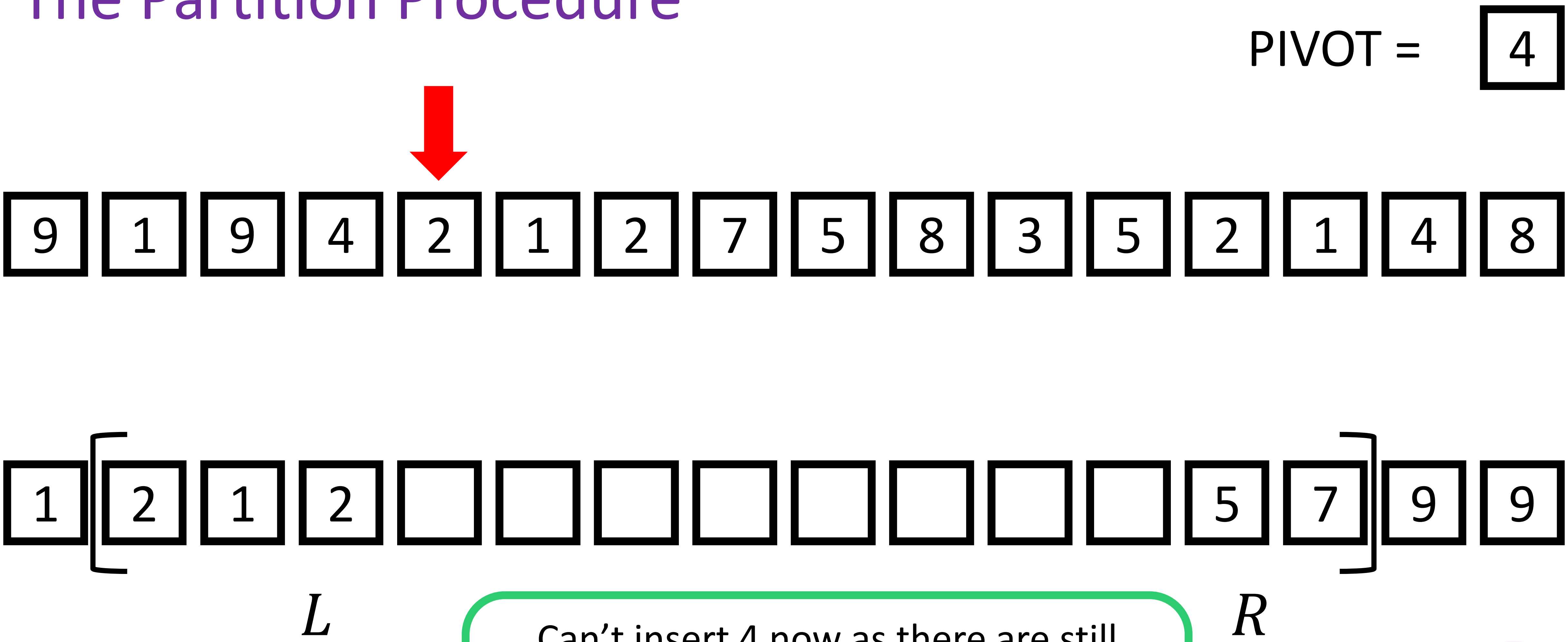
Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure



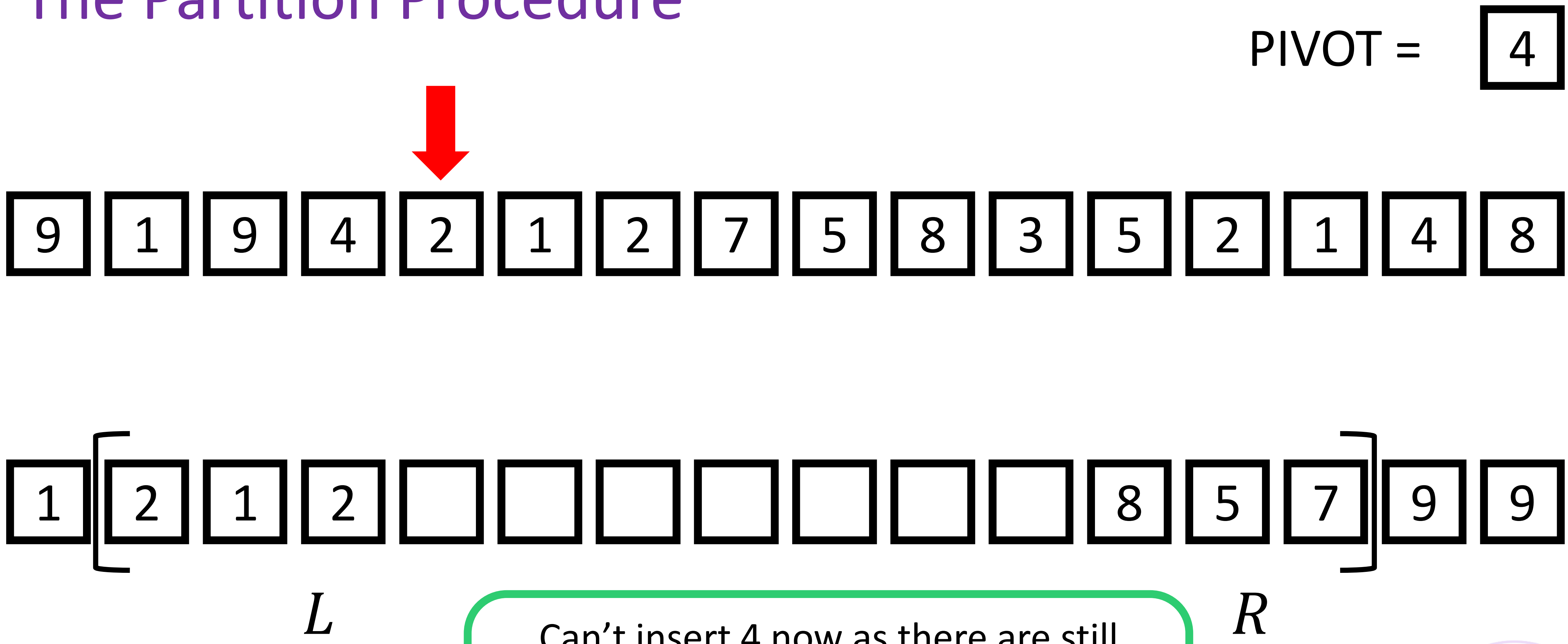
Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure



Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure



Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

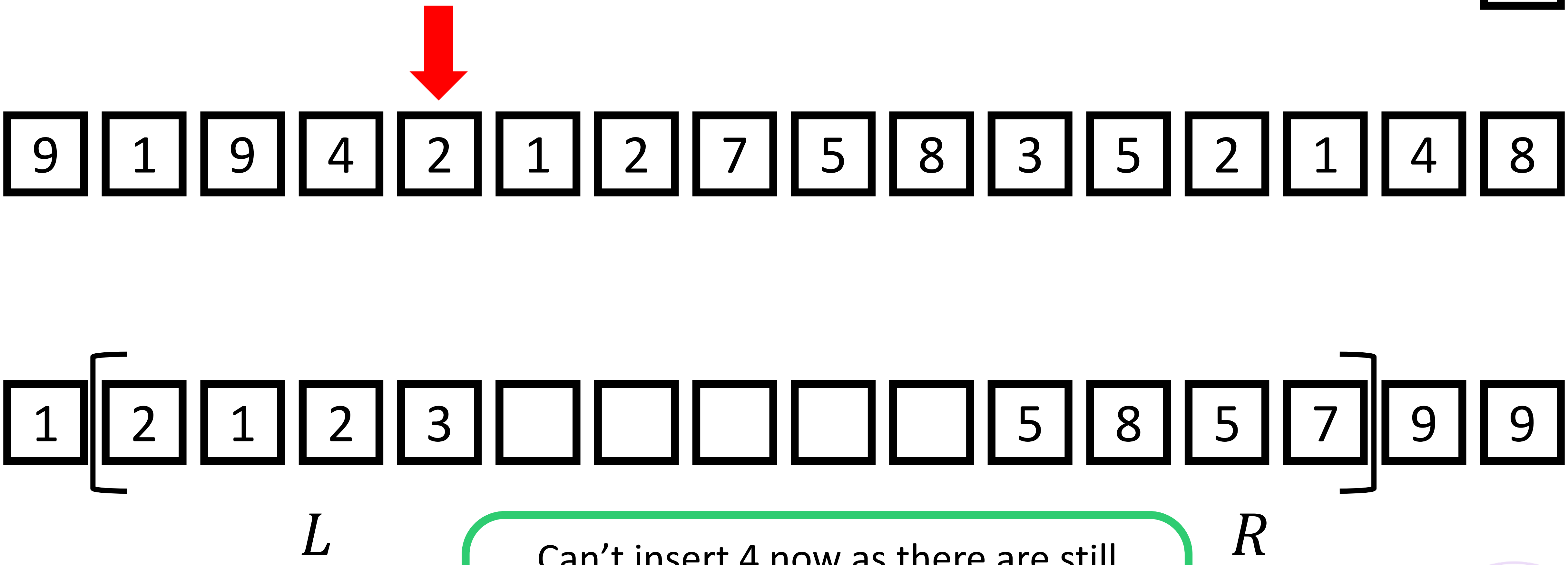
1	2	1	2	3							8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

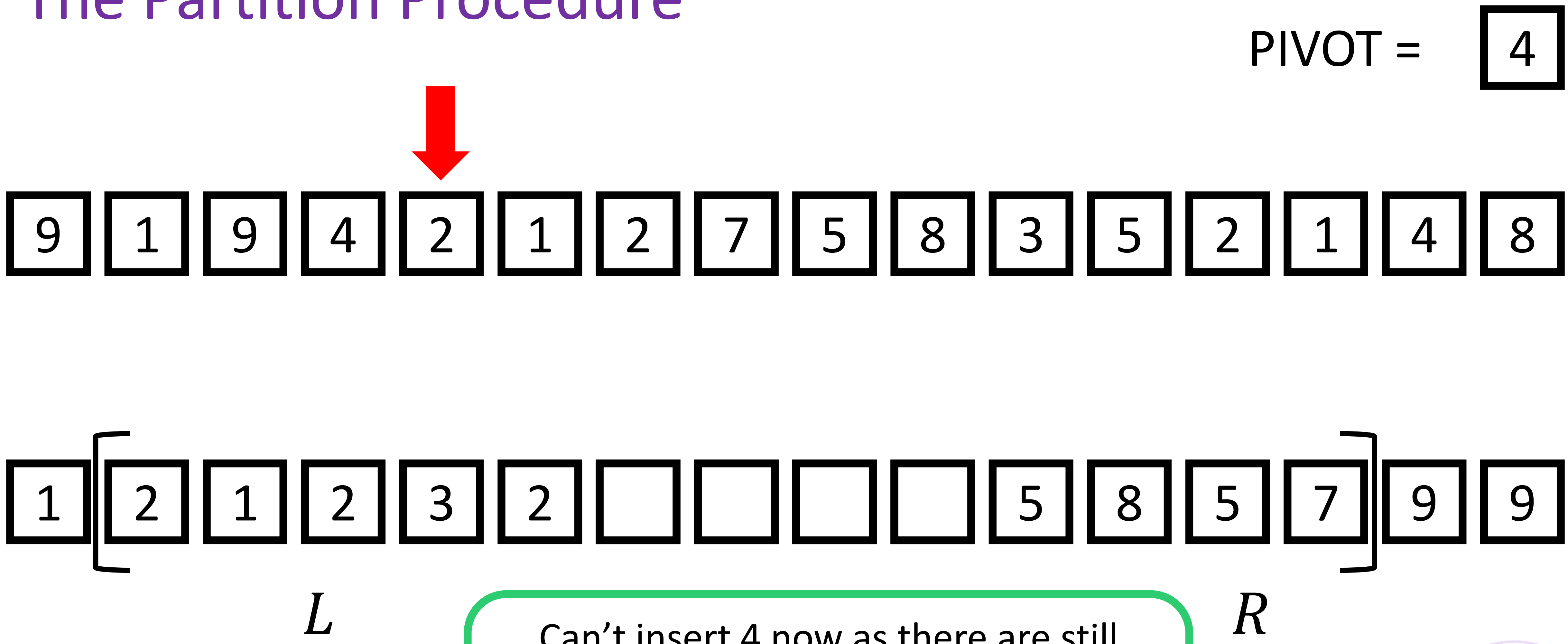
The Partition Procedure

PIVOT = $\boxed{4}$



Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure



Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1				5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1				5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1				5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1	[		]	5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

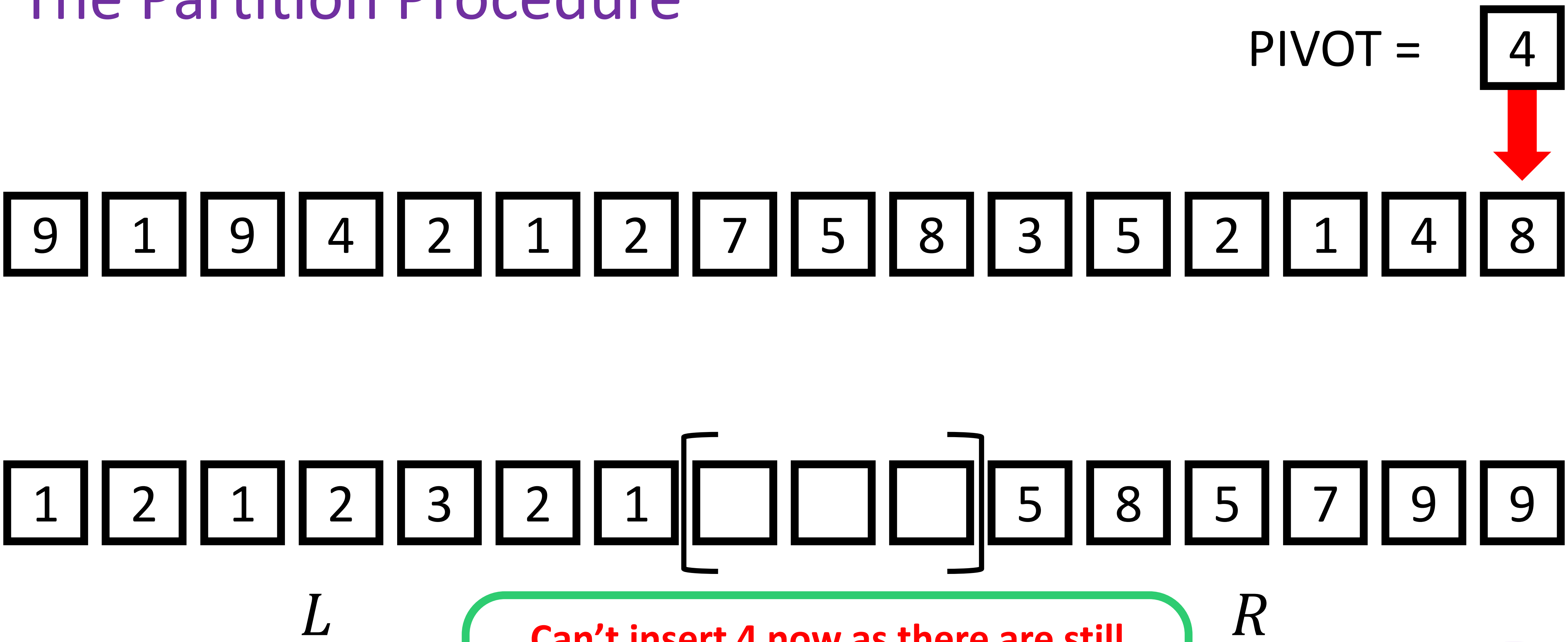
1	2	1	2	3	2	1				5	8	5	7	9	9

L R

$8 > 4$ i.e. belongs to R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure



Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1			8	5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1			8	5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

We are sure now that any blank spaces left must be occurrences of pivot 4 that we omitted earlier

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1			8	5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

We are sure now that any blank spaces left must be occurrences of pivot 4 that we omitted earlier

PIVOT = $\boxed{4}$

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	3	2	1	[4	4]	8	5	8	5	7	9	9

L R

Can't insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our invariant later

The Partition Procedure

PARTITION

1. Given: Array a with N elements, pivot element p
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$ *//Initialize b to be an empty array*
3. For $i = 0; i < N; i$++
    1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and L++ *//We found a left element*
    2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and R-- *//We found a right element*
4. For $i = L; i \leq R; i$++
    1. Let $b[j] \leftarrow p$ *//Fill up the remaining places with the pivot*
5. Return (b, R)

The Partition Procedure

PARTITION

1. Given: Array a with N elements, pivot eler
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$ //Initia
3. For $i = 0; i < N; i$++
    1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and L++ *//We found a left element*
    2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and R-- *//We found a right element*
4. For $i = L; i \leq R; i$++
    1. Let $b[j] \leftarrow p$ *//Fill up the remaining places with the pivot*
5. Return (b, R)

Verify that after the first loop has ended, we must have $L < R$ i.e. some space left for pivot

The Partition Procedure

PARTITION

1. Given: Array a with N elements, pivot eler
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$ //Initia
3. For $i = 0; i < N; i$++
   1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and L++ *//We found a left element*
   2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and R-- *//We found a right element*
4. For $i = L; i \leq R; i$++
   1. Let $b[j] \leftarrow p$ *//Fill up the remaining places with the pivot*
5. Return (b, R)

Verify that after the first loop has ended, we must have $L < R$ i.e. some space left for pivot

R has to be (one of) the new location(s) of the pivot element

The Partition Procedure

PARTITION

1. Given: Array a with N elements, pivot eler
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$ //Initia
3. For $i = 0; i < N; i$++
   1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and L++ *//We found a left element*
   2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and R-- *//We found a right element*
4. For $i = L; i \leq R; i$++
   1. Let $b[j] \leftarrow p$ *//Fill up the remaini*
5. Return (b, R)

Verify that after the first loop has ended, we must have $L < R$ i.e. some space left for pivot

In fact, the entire range $b[L:R]$ is filled with the pivot element

R has to be (one of) the new location(s) of the pivot element

The Partition Procedure

PARTITION

1. Given: Array a with N elements, pivot eler
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$ *//Initia*
3. For $i = 0; i < N; i$++
   1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and L++ *//We found a left element*
   2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and R-- *//We found a right element*
4. For $i = L; i \leq R; i$++
   1. Let $b[j] \leftarrow p$ *//Fill up the remaini*
5. Return (b, R)

Verify that after the first loop has ended, we must have $L < R$ i.e. some space left for pivot

In fact, the entire range $b[L{:}\,R]$ is filled with the pivot element

R has to be (one of) the new location(s) of the pivot element

Explore/invent yourself an *in-place* partitioning algorithm

The Partition Procedure

PARTITION

1. Given: Array a with N elements, pivot eler
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$ //Initia
3. For $i = 0; i < N; i$++
   1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and L++ *//We found a left element*
   2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and R-- *//We found a right element*
4. For $i = L; i \leq R; i$++
   1. Let $b[j] \leftarrow p$ *//Fill up the remaini*
5. Return (b, R)

Verify that after the first loop has ended, we must have $L < R$ i.e. some space left for pivot

In fact, the entire range $b[L{:}R]$ is filled with the pivot element

R has to be (one of) the new location(s) of the pivot element

Hint: the in-place algorithm uses an identical notion of inactive regions but swaps elements at the boundaries of the regions which are wrongly placed

Explore/invent yourself an *in-place* partitioning algorithm

Choice of Pivot

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm
- Suppose we are so unlucky that we always end up choosing the smallest or the largest element of the array as the pivot

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm
- Suppose we are so unlucky that we always end up choosing the smallest or the largest element of the array as the pivot

9	1	8	4	2	1	2	7	5	8	3	5	2	1	4	8

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm
- Suppose we are so unlucky that we always end up choosing the smallest or the largest element of the array as the pivot

9	1	8	4	2	1	2	7	5	8	3	5	2	1	4	8

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm
- Suppose we are so unlucky that we always end up choosing the smallest or the largest element of the array as the pivot

9	1	8	4	2	1	2	7	5	8	3	5	2	1	4	8

1	2	1	2	2	1	3	8	4	5	8	5	7	4	8	9

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm
- Suppose we are so unlucky that we always end up choosing the smallest or the largest element of the array as the pivot

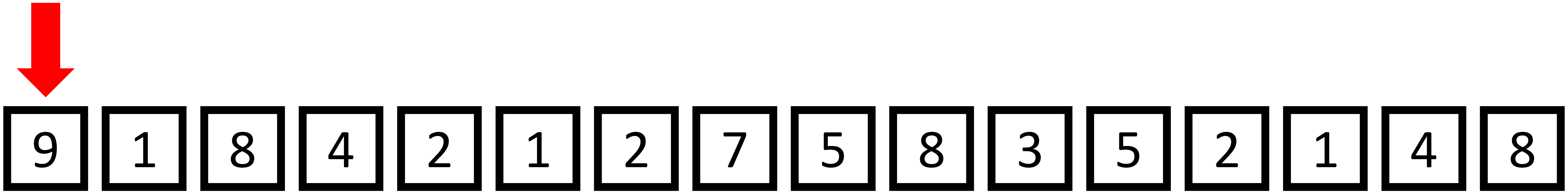
9	1	8	4	2	1	2	7	5	8	3	5	2	1	4	8

- Choosing an element close to the median is most beneficial

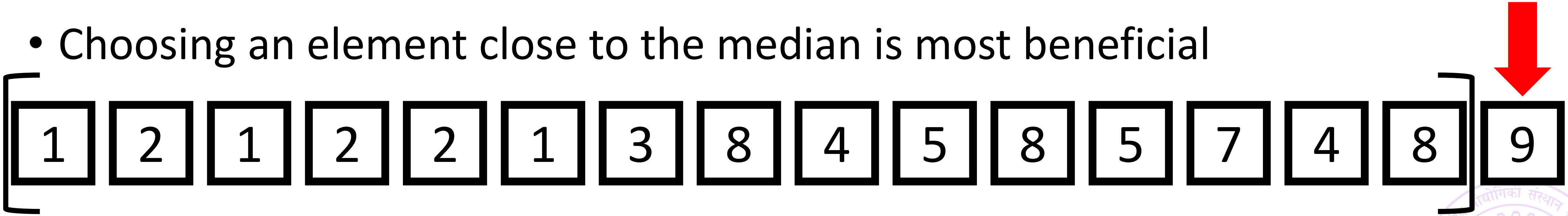
1	2	1	2	2	1	3	8	4	5	8	5	7	4	8	9

Choice of Pivot

- Most crucial step in quicksort – may make or break the algorithm
- Suppose we are so unlucky that we always end up choosing the smallest or the largest element of the array as the pivot



- Choosing an element close to the median is most beneficial



- Quicksort becomes selection sort i.e. O(N^2) time ☹

Choice of Pivot

- Most crucial step in quicksort – may r
- Suppose we are so unlucky that we al the largest element of the array as the

Ironically, if the array is already sorted and we use end elements as pivots, then quicksort takes $\mathcal{O}(N^2)$ time ☹

9	1	8	4	2	1	2	7	5	8	3	5	2	1	4	8

- Choosing an element close to the median is most beneficial

1	2	1	2	2	1	3	8	4	5	8	5	7	4	8	9

- Quicksort becomes selection sort i.e. O(N^2) time ☹

Some folklore wisdom

- "Slow" algorithms with $\mathcal{O}(N^2)$ time (selection/insertion/bubble sort) actually faster than merge/quicksort for small arrays!
- Constants hidden by $\mathcal{O}(\cdot)$ are the devil here – overheads in merge/ quicksort: $cN^2 < dN \log N$ if N is really small e.g. $N/\log N < d/c$.
- When executing recursive algorithms like Merge/Quicksort, once subarrays become small ~10-50, call insertion/selection sort
- Several *integer-sorting* algorithms like Radix sort, Counting sort. Work only on integer arrays but can sort in $\mathcal{O}(N)$ time!!
- Speed is just one aspect of sorting algorithms. Many other aspects
  - Additional memory usage (is it an in-place sorting method or not?)
  - Stability (does the algorithm preserve the order of repeated elements?)
  - Is the method extra quick at sorting partially sorted arrays? Qsort isn't ☺
- We have very good knowledge of sorting – ESO207/CS345