# CPS 305

## Data Structures

Prof. Alex Ufkes

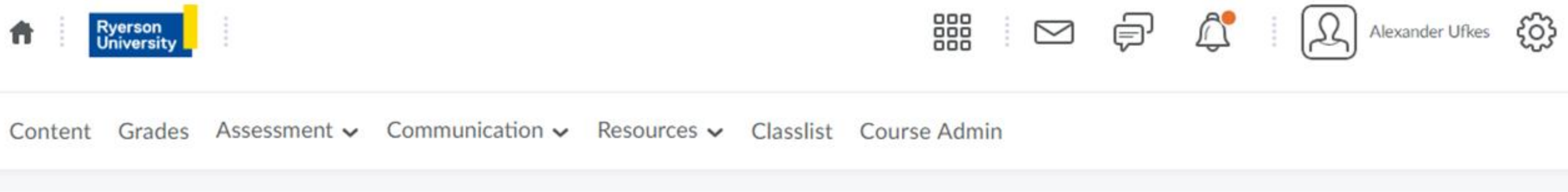
**Topic 4:** Advanced Sorting

Toronto Metropolitan University

# Notice!

**Obligatory copyright notice in the age of digital delivery and online classrooms:**

# Course Administration




- Lab 3 posted
- Attend the lab to get help from your TA.

# **Previously:** Comparison Sorting

**Time complexity VS implementation complexity**

- Simple $O(n^2)$ algorithms VS complex O(nlogn) algorithms

**In-place VS requiring extra memory**

- Sorting *in-place* means we do not require any helper arrays

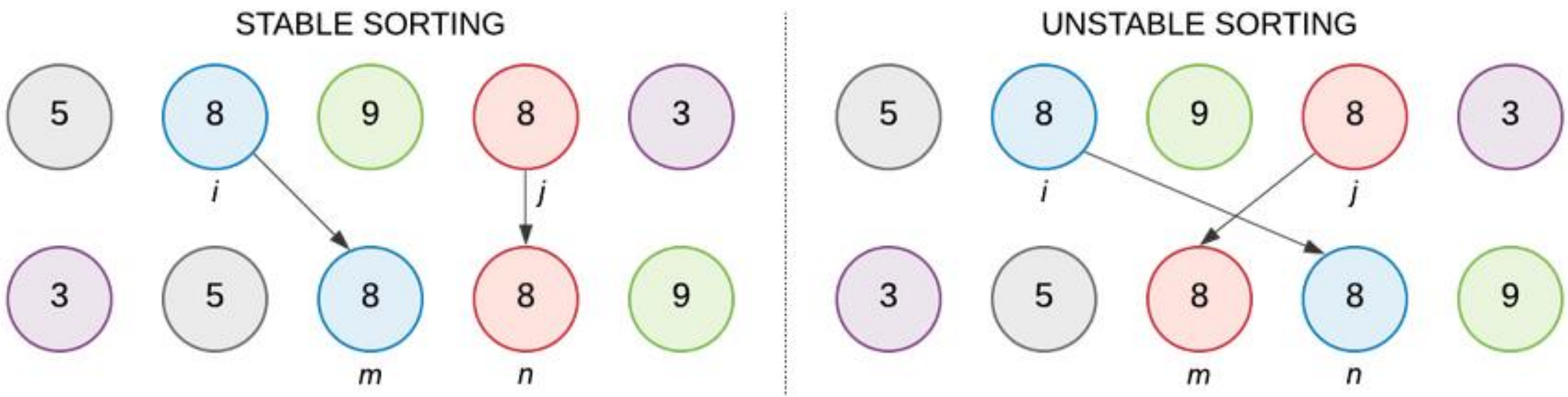
**Stable VS unstable:**

- Stable sort: Relative position of equal elements doesn’t change

# **Previously:** Stable Sorting

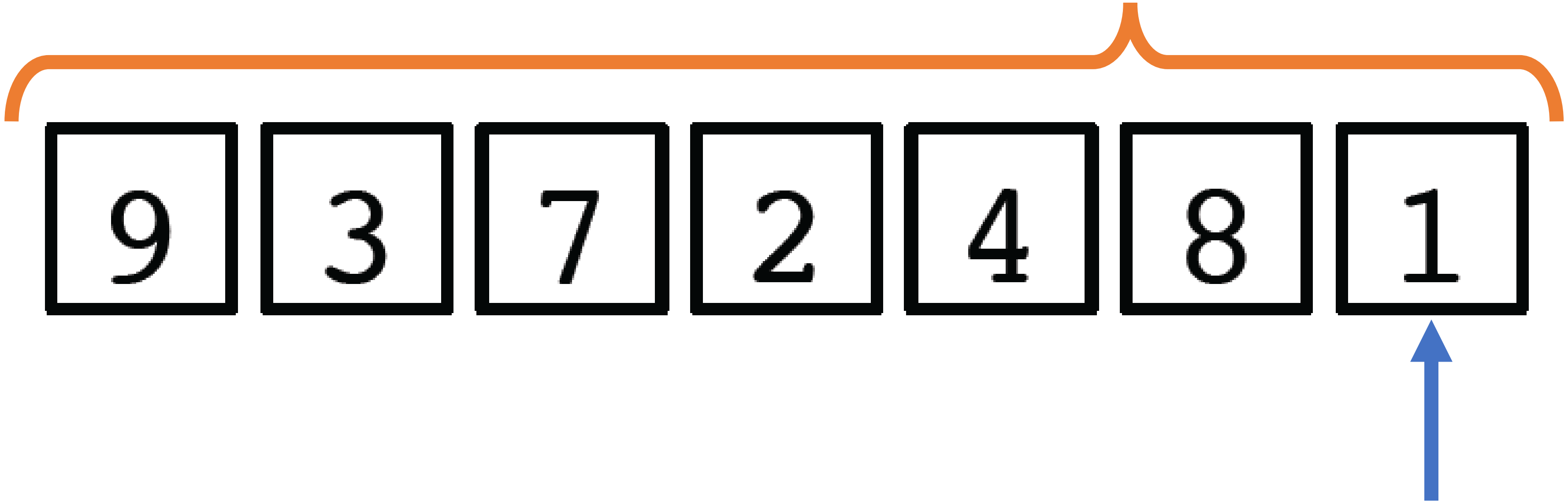
**Stable VS unstable:**

- Stable sort: Relative position of equal elements doesn't change

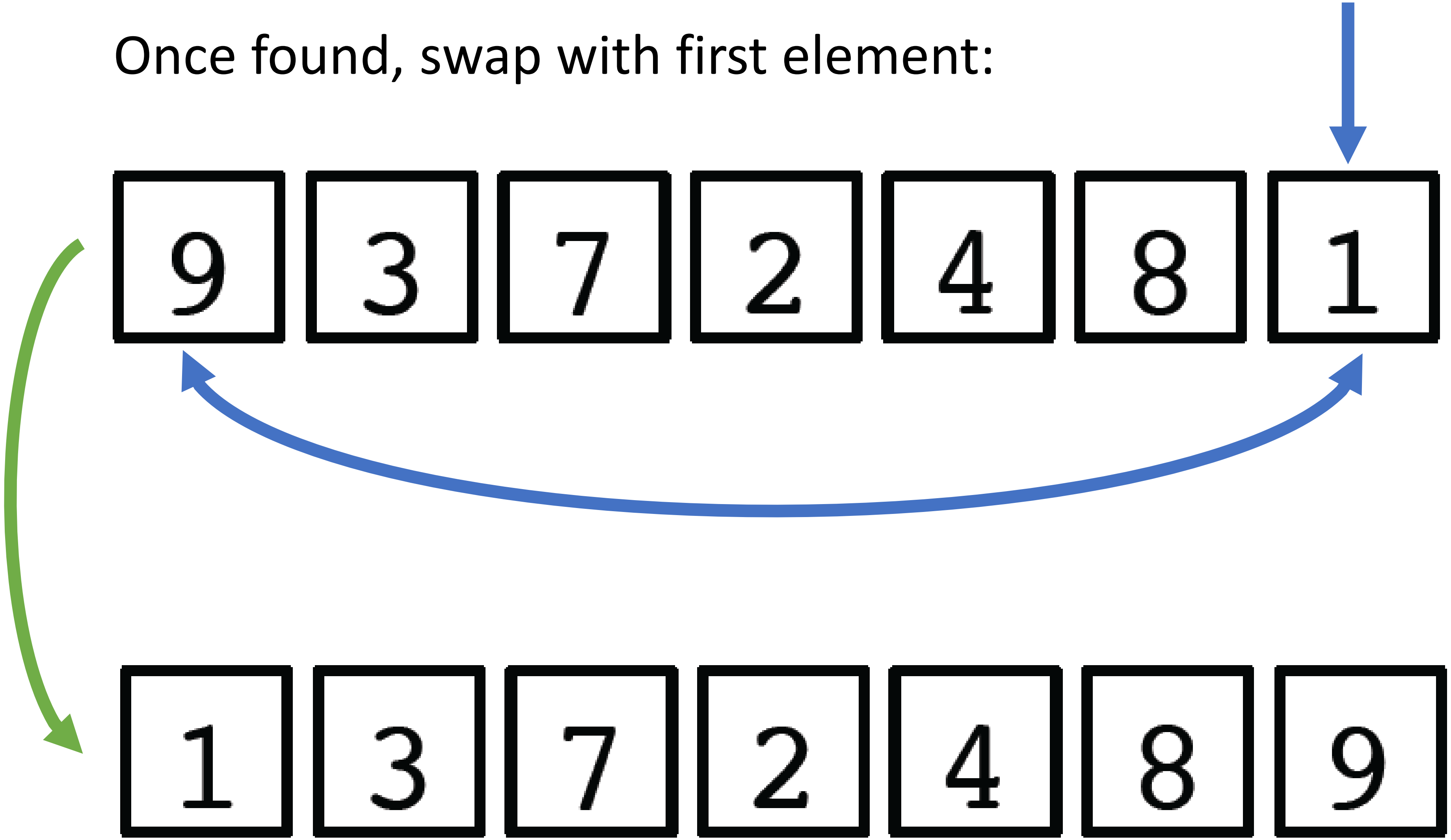


# Selection Sort

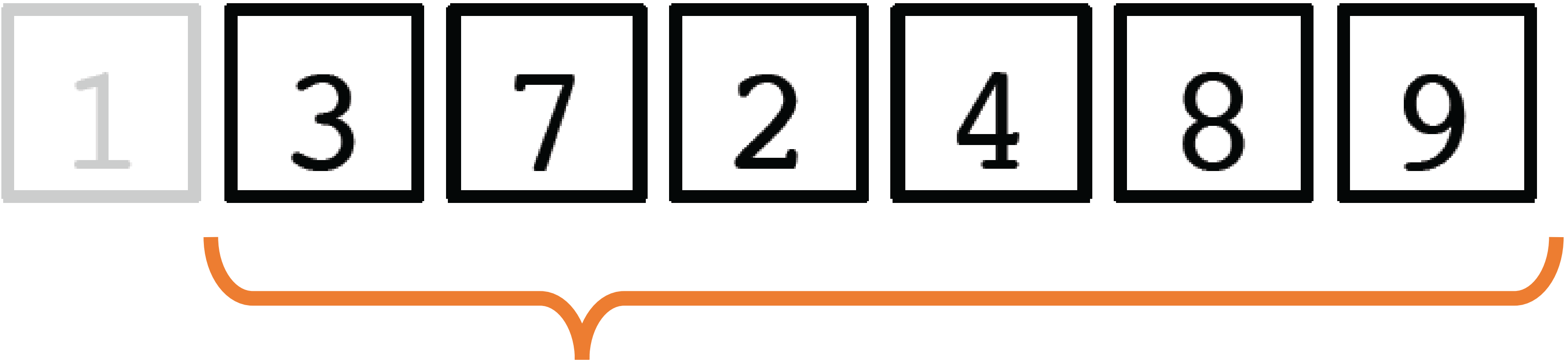
**Selection Sort:** Repeatedly find smallest element and move it to the front of the *unsorted* region



Once found, swap with first element:



We can now be **certain** that the first element is in the correct location:




**New unsorted region!**

# Selection Sort

## It’s ***BAD***

- It’s good as “*my first sorting algorithm*”
- Bad for sorting in an efficient manner
- Performance is identical in best-case and worst-case scenarios.
- Even if the list is ***already sorted***, selection sort takes just as long to perform.
- **Why?** Finding minimum value is ***always*** O(n)

# Moving On…

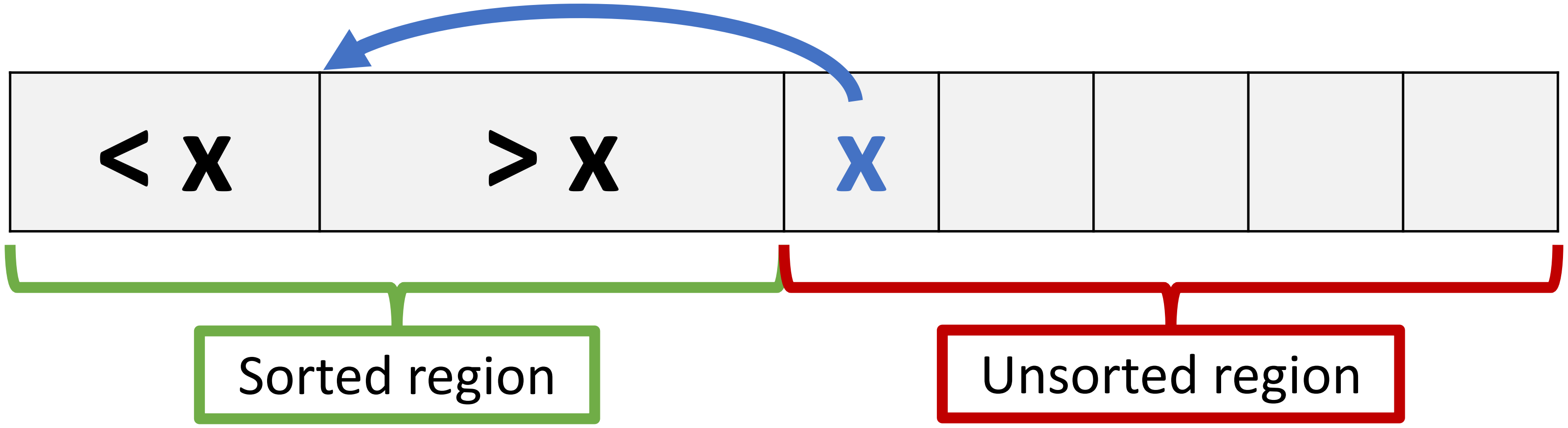
# Selection Sort

It’s ***BAD***

- Even among O($n^2$) sorting algorithms, it’s bad.
- We’ll see a much more attractive O($n^2$) algorithm today, and then move on to O(nlogn) sorting algorithms.

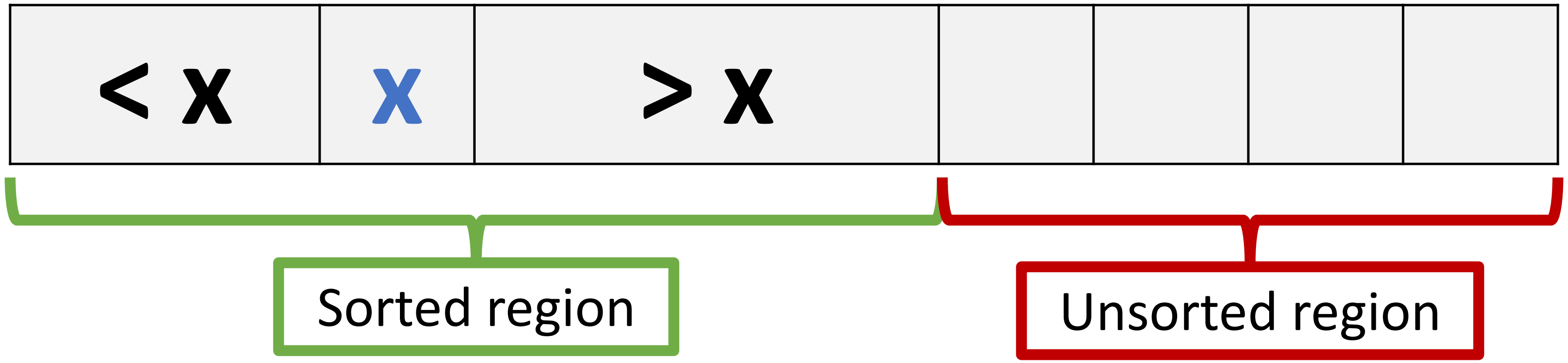
# **Insertion Sort:** Algorithm

- Iterate through the array, consuming one element at a time.
- Move each element from the front of the unsorted region to its correct place in the sorted region.



# **Insertion Sort:** Algorithm

- Iterate through the array, consuming one element at a time.
- Move each element from the front of the unsorted region to its correct place in the sorted region.



# **Insertion Sort:** Algorithm

**For any unsorted list:**

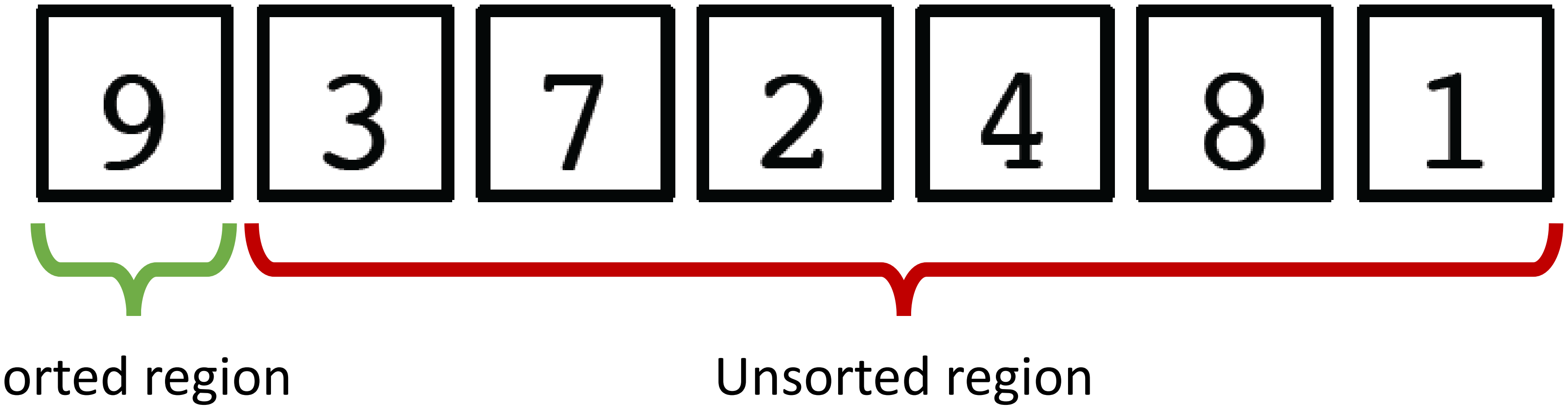
- Treat the first element as a sorted sub-list of size 1

**Then, given a sorted list of size *k*–1**

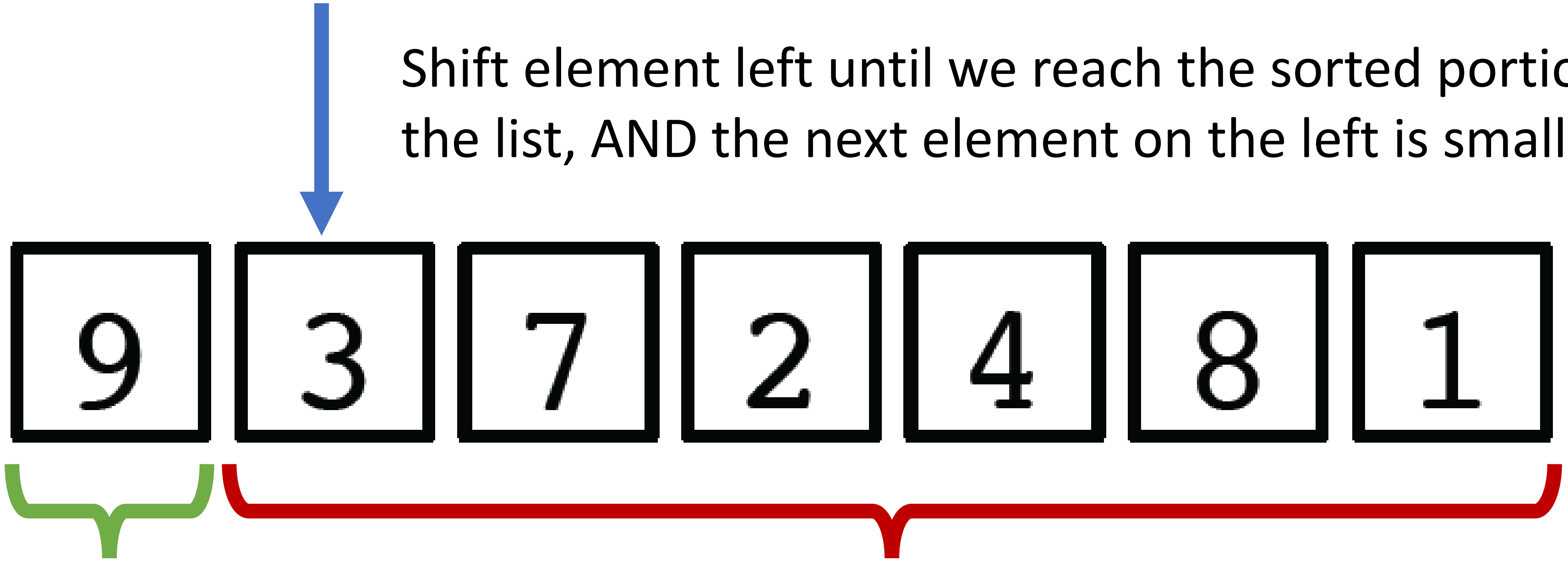
- Insert the $k^{th}$ item in the unsorted list into the sorted list
- The sorted sub-list is now of size $k + 1$

# Insertion Sort

**Insertion Sort:** Every iteration removes next element from the unsorted region and inserts it into the correct position within the sorted region.

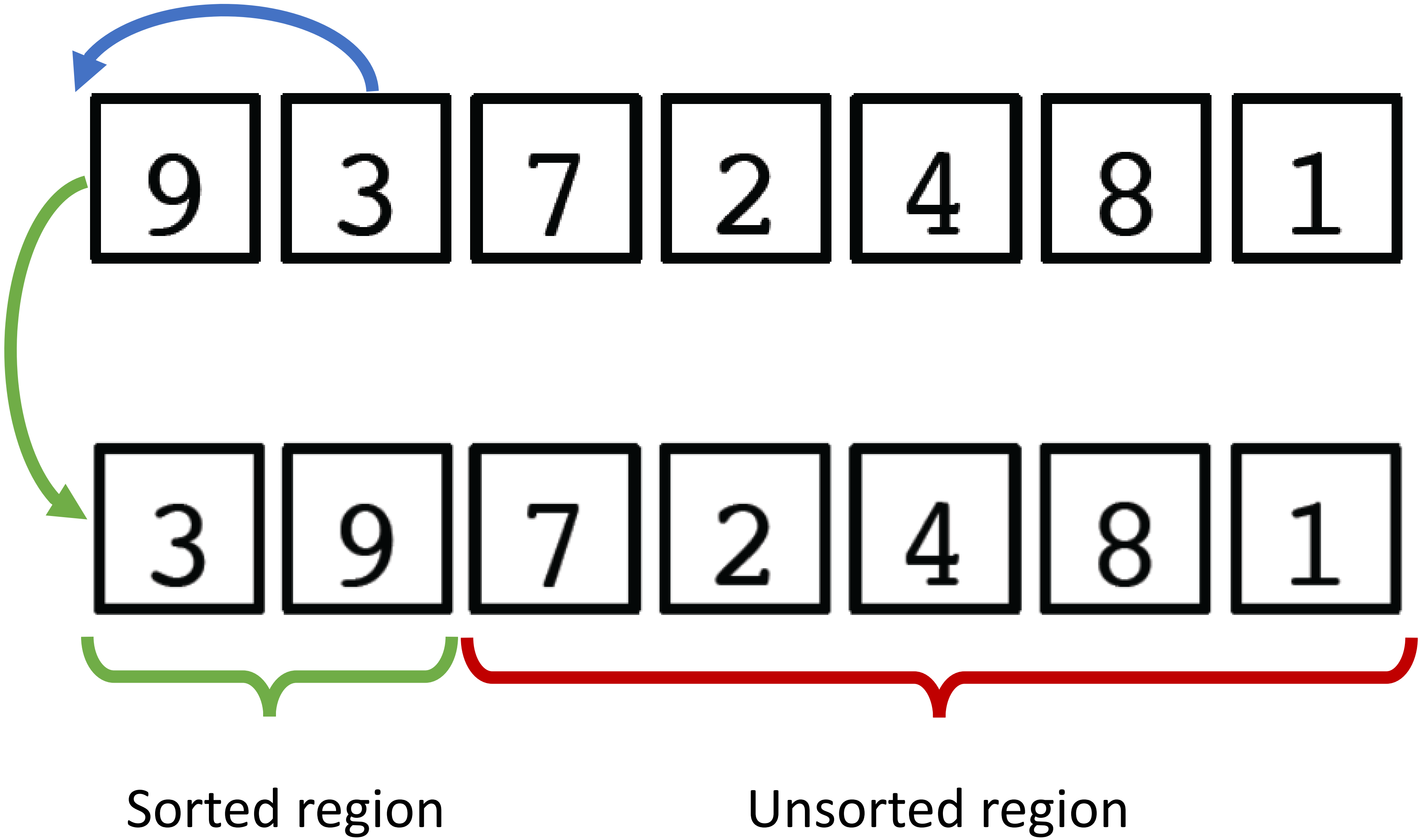


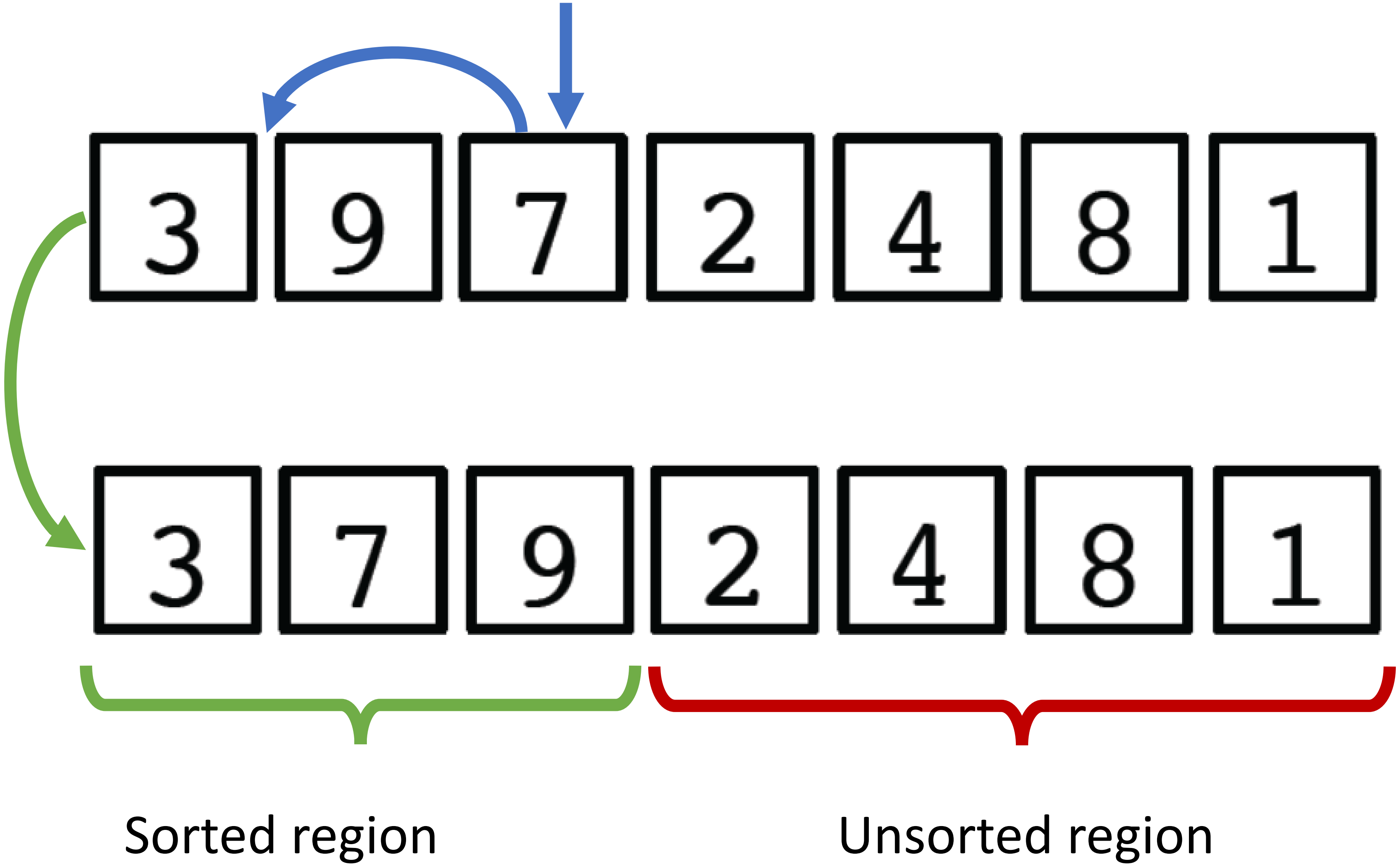
Shift element left until we reach the sorted portion of the list, AND the next element on the left is smaller.

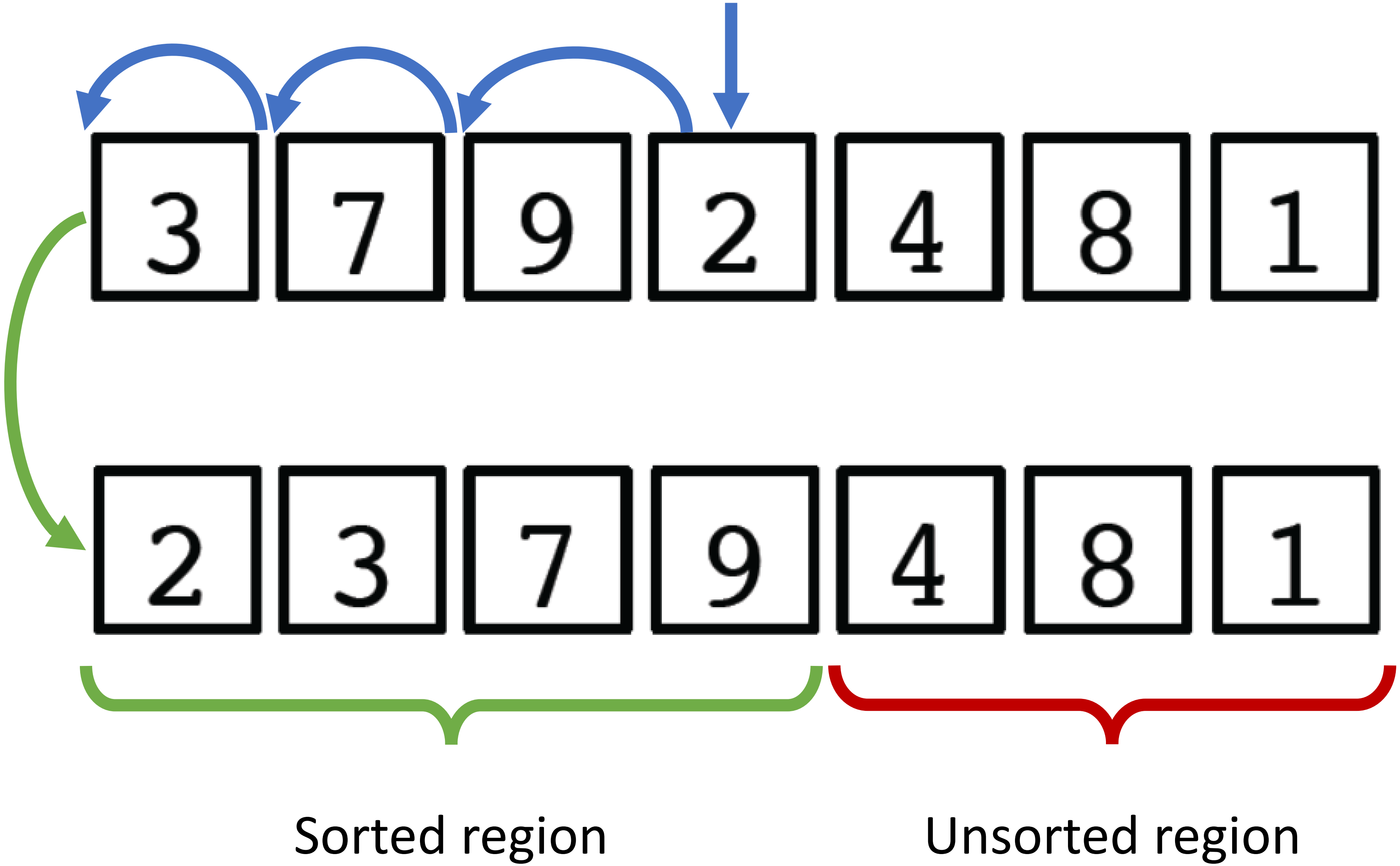


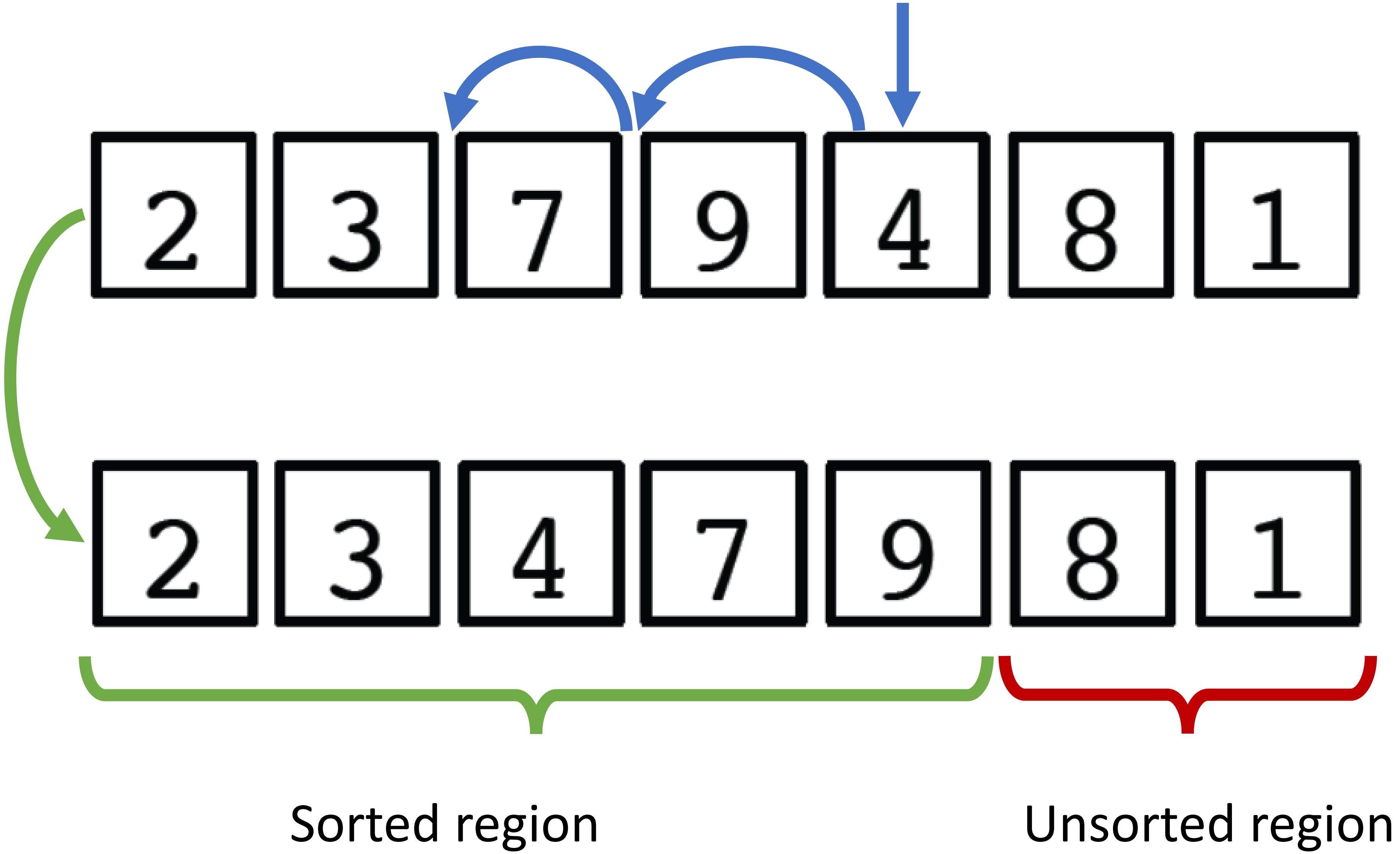

Sorted region

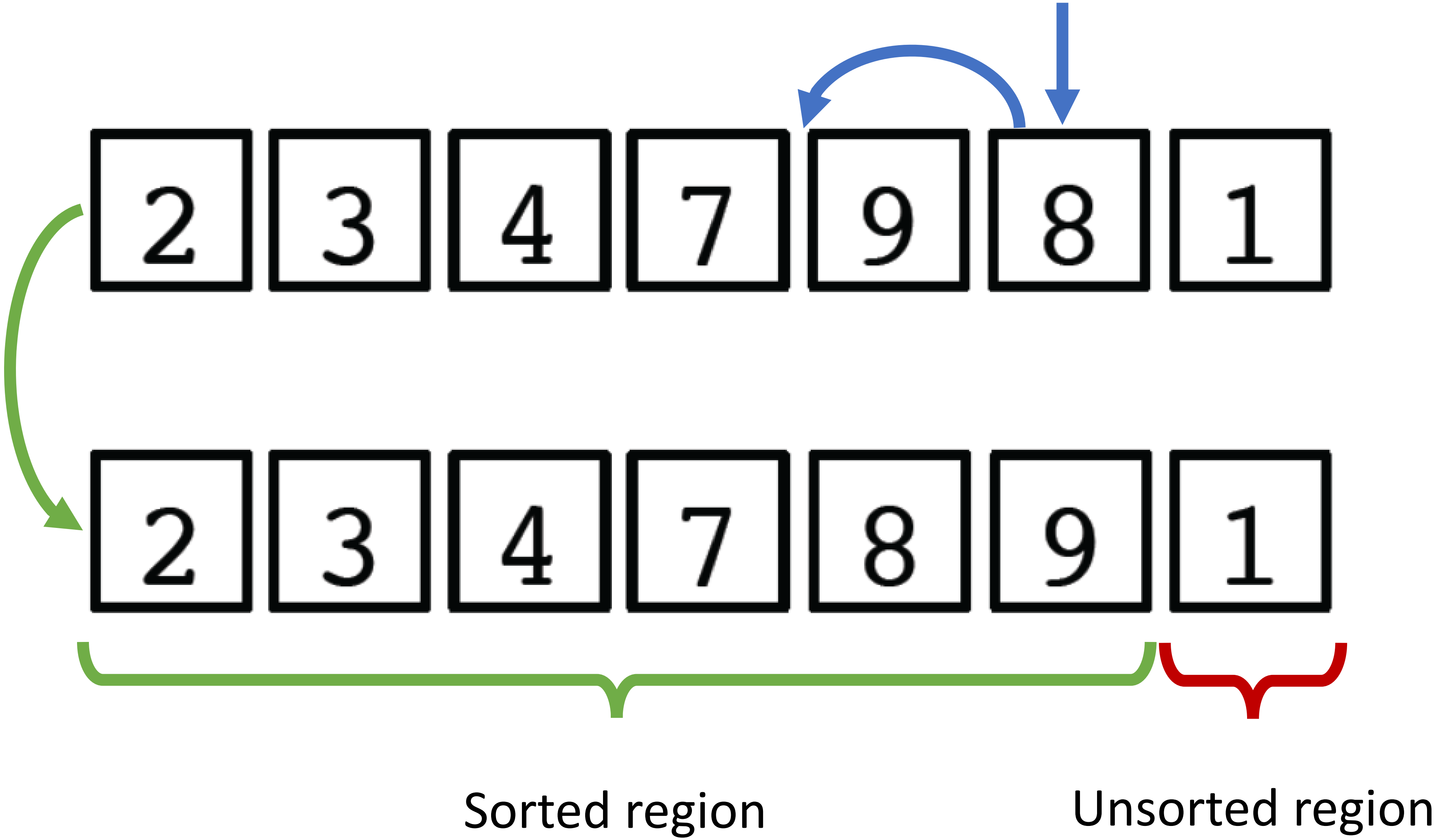
Unsorted region

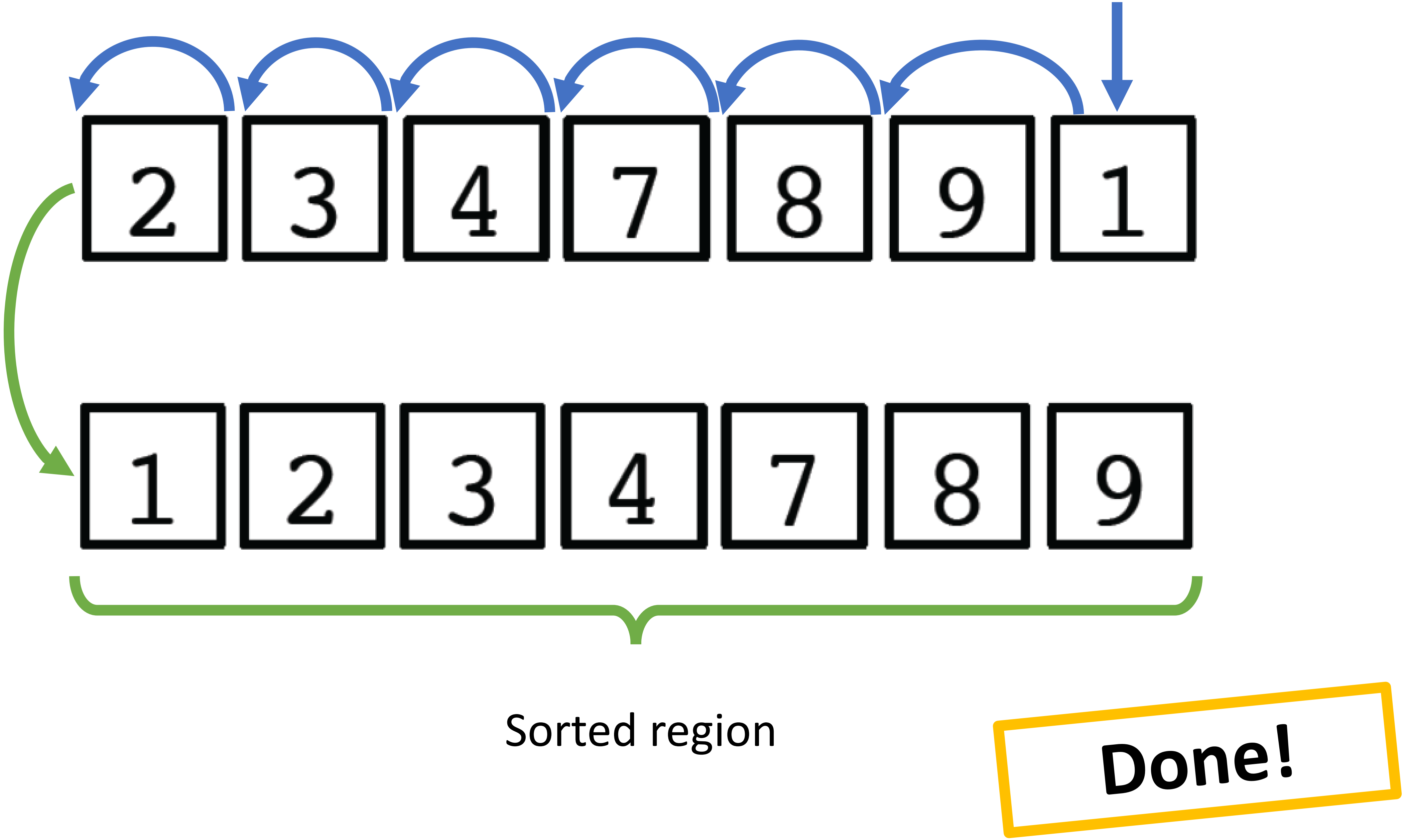

9 3 7 2 4 8 1
3 9 7 2 4 8 1
Sorted region
Unsorted region


3
9
7
2
4
8
1
3
7
9
2
4
8
1
Sorted region
Unsorted region


3 7 9 2 4 8 1
2 3 7 9 4 8 1
Sorted region
Unsorted region


2 3 7 9 4 8 1
2 3 4 7 9 8 1
Sorted region
Unsorted region


2
3
4
7
9
8
1
2
3
4
7
8
9
1
Sorted region
Unsorted region


2 3 4 7 8 9 1
1 2 3 4 7 8 9
Sorted region
Done!

```lisp
(defun insertion-sort (vec comp)
  (dotimes (i (1- (length vec)))
    (do ((j i (1- j))) ; starts at i, decrements by 1
      ((minusp j)) ; checks if j is negative
      (if (funcall comp (aref vec (1+ j)) (aref vec j))
        (rotatef (aref vec (1+ j)) (aref vec j))
        (return))
  ) ) )
  vec
)
```

Shift element left until we hit the front OR a smaller element

Exit inner loop if left element is smaller than right element.

Return vec (now sorted) at the end

```lisp
* (defvar a (make-array 6 :initial-contents '(3 1 0 7 8 2)))
A
* (insertion-sort a '<)
#(0 1 2 3 7 8)
```

# Selection VS Insertion

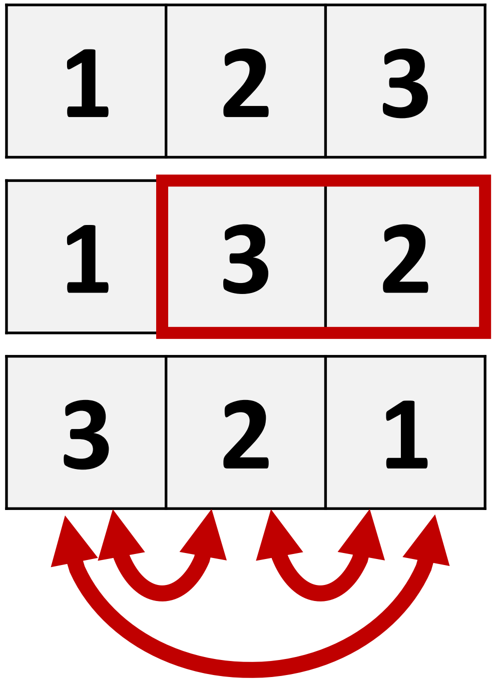
**Insertion sort is far more powerful:**

- It shifts elements only as far as they need to move.
- If the list is already sorted, no shifting is required!
- The efficiency of insertion sort depends on the initial *sorted-ness* of the list.
- In the worst case? It's just as bad as selection sort.
- In the best case? It's much, ***much*** better!
- Selection sort is ***always*** bad.

**Observation:** The inner loop body will only run as many times as there are ***inversions***

# Removing Inversions

An *inversion* refers to any two elements that are out of order.



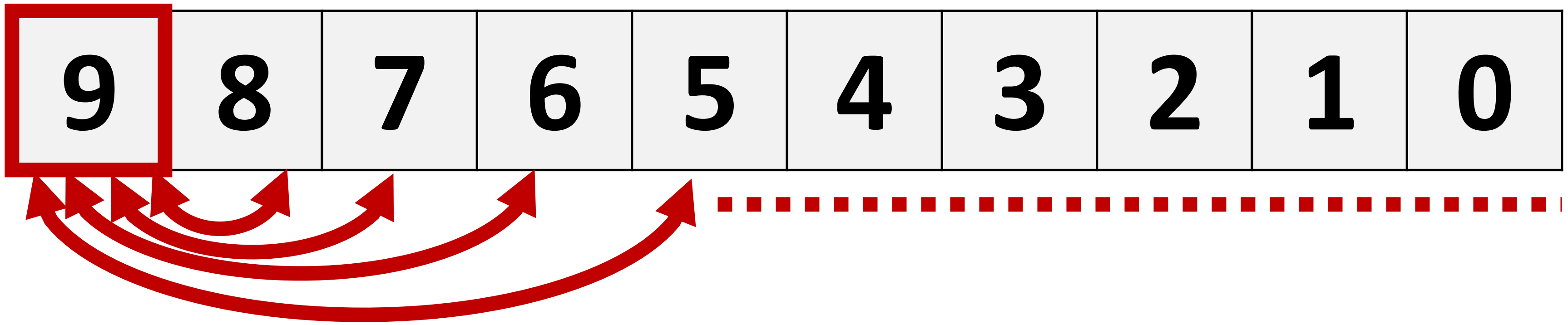

0 inversions

1 inversion

3 inversions

# Removing Inversions

A better example:




- An array in perfect reverse order has $O(n^2)$ inversions.
- An array in near-sorted order has $O(n)$ inversions.

$$= n-1 + n-2 + \ldots + 2 + 1$$

$$= \frac{(n-1)(n)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \quad \boxed{= O(n^2)}$$

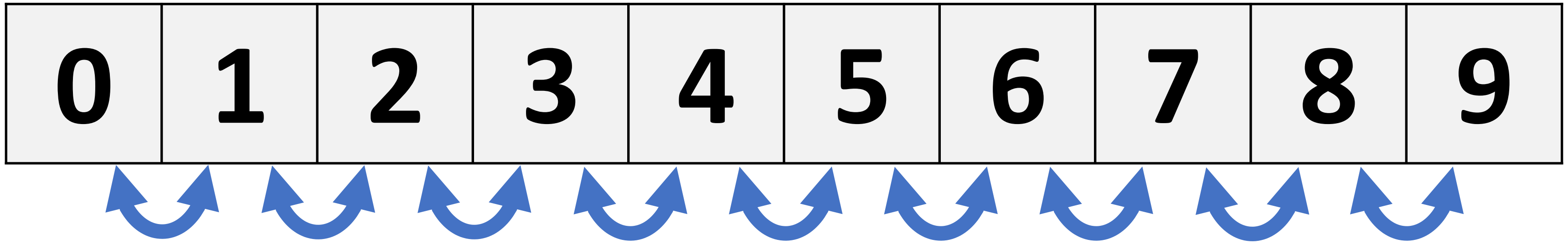
```lisp
(defun insertion-sort (vec comp)
  (dotimes (i (1- (length vec)))
    (do ((j i (1- j))) ; starts at i, decrements by 1
      ((minusp j)) ; checks if j is negative
      (if (funcall comp (aref vec (1+ j)) (aref vec j))
        (rotatef (aref vec (1+ j)) (aref vec j))
        (return)
  ) ) )
  vec
)
```

Removes one inversion per execution

# **Insertion Sort:** Complexity Analysis

Best case, the list is already sorted.

We will make n-1 comparisons:




**Insertion sort is O(n) in the best case!**

# **Insertion Sort:** Complexity Analysis

Worst case, the list is in perfect reverse order:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

- Each element in the unsorted portion will be compared to every element in the sorted portion.

```
a[1]     1 comparison
a[2]     2 comparisons
…
…
a[n-1]   n-1 comparisons
```

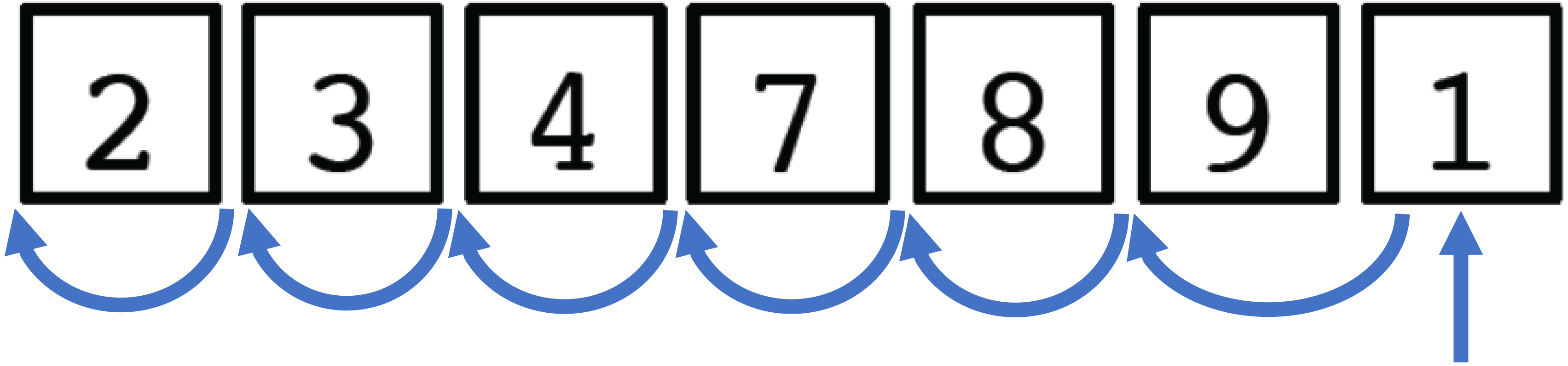
= 1 + 2 + 3 + … + n-2 + n-1

$$= \frac{(n-1)(n)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \quad \boxed{= O(n^2)}$$

# **Insertion Sort:** Optimization #1

In addition to comparisons, we're swapping elements (exchanges)

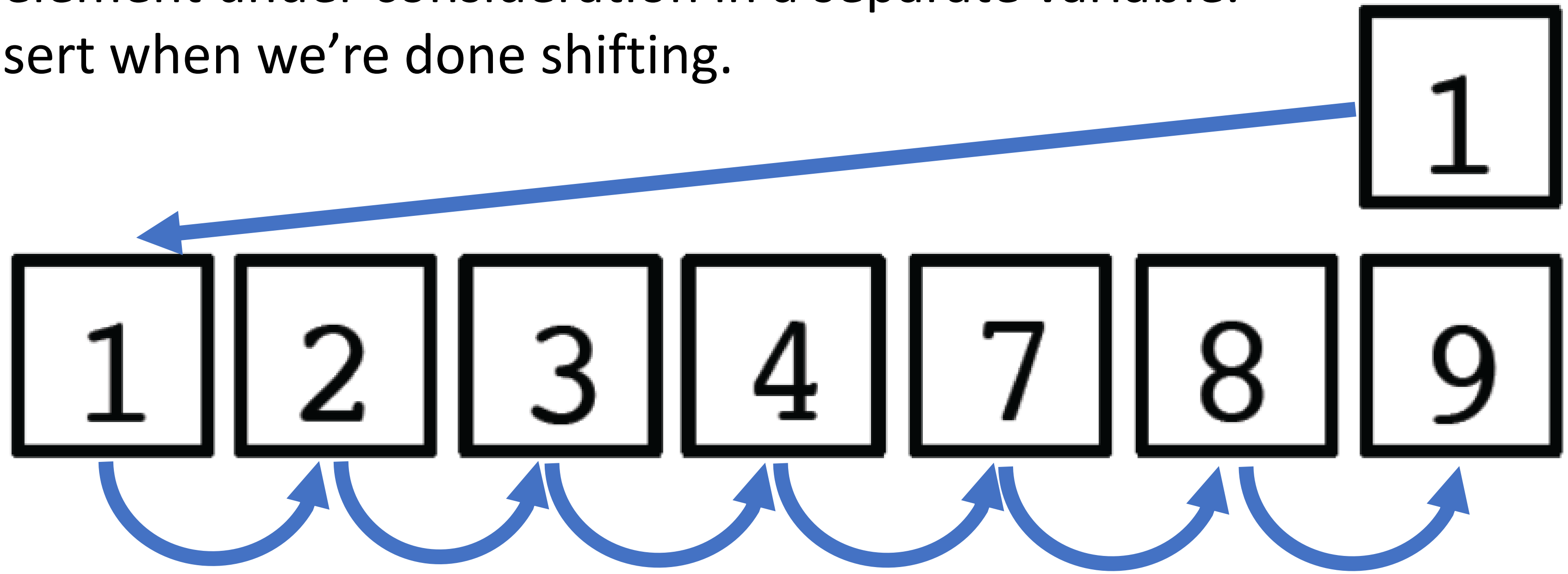
- Best case? List is sorted, zero exchanges.
- Worst case? We're performing as many exchanges as comparisons.



# **Insertion Sort:** Optimization #1

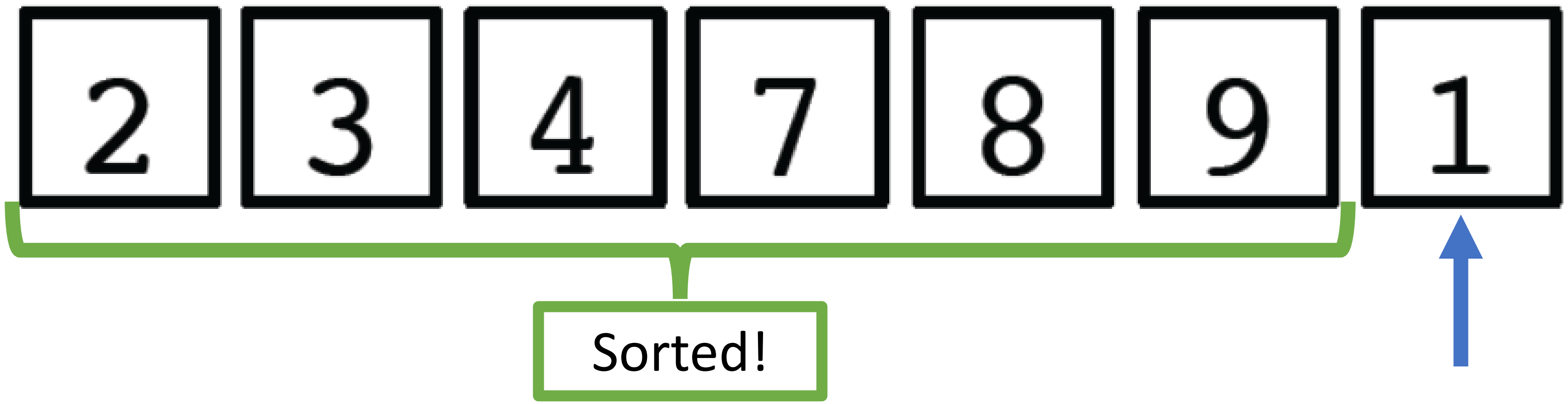
***Worst case?*** *We're performing as many exchanges as comparisons.*

- That's dumb. Perform a *half-exchange* instead.
- Instead of *exchanging* elements, just shift them down.
- Save element under consideration in a separate variable.
- Re-insert when we're done shifting.



# **Insertion Sort:** Optimization #2

- We need to find out where the next element fits
- We do this by… searching… the sorted portion. Hmm…




**Binary search the sorted portion!**

# **Insertion Sort:** Optimization #2

**Binary search the sorted portion!**

- We find the insertion point for each element in **`O(logn)`**
- This reduces the number of comparisons to **`O(nlogn)`**
- The algorithm is still **$O(n^2)$**
- Once we find the insertion point, we still have to shift the elements over.

# **Insertion Sort:** In Practice?

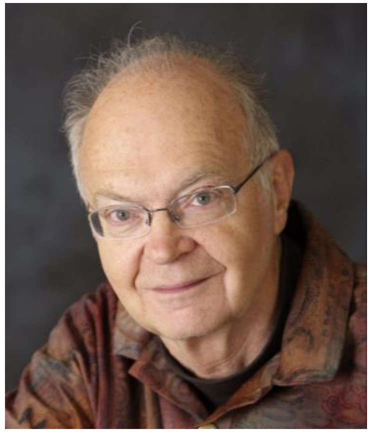
- Bad for large arrays – $O(n^2)$
- It turns out that, in practice, insertion sort is one of the fastest options for sorting very small arrays ($n<10$)
- Even faster than quicksort! (Coming up)
- Quicksort implementations often employ ***insertion sort*** on small sub-arrays.
- The exact size where insertion sort pulls ahead is determined experimentally. Can vary by machine.

6 5 3 1 8 7 2 4

# Bubble Sort

# Bubble Sort

- Starting at the front of an unsorted array, swap adjacent elements if they are out of order.
- Thus, *bubbling* the largest element to the end



*"The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems"*

**- Donald Knuth**

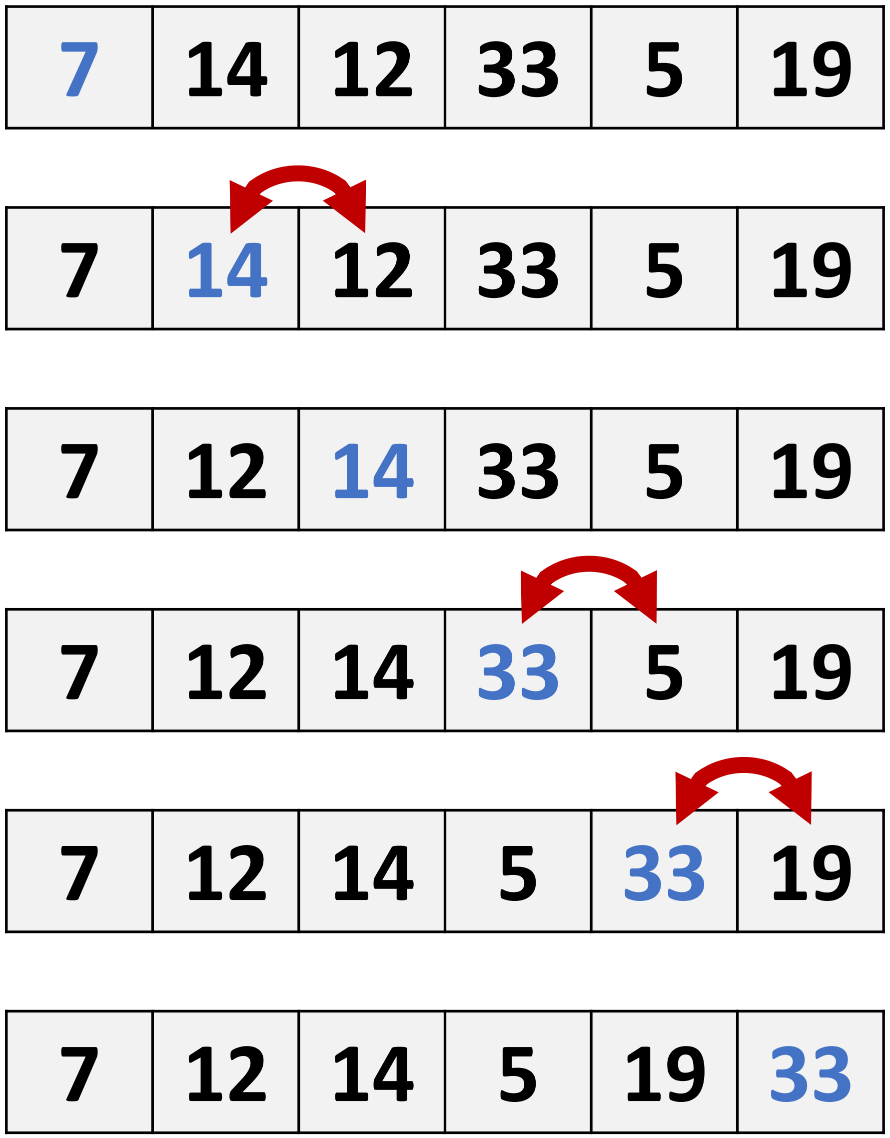
# Bubble Sort:
## Example

Consider the unsorted array to the right:

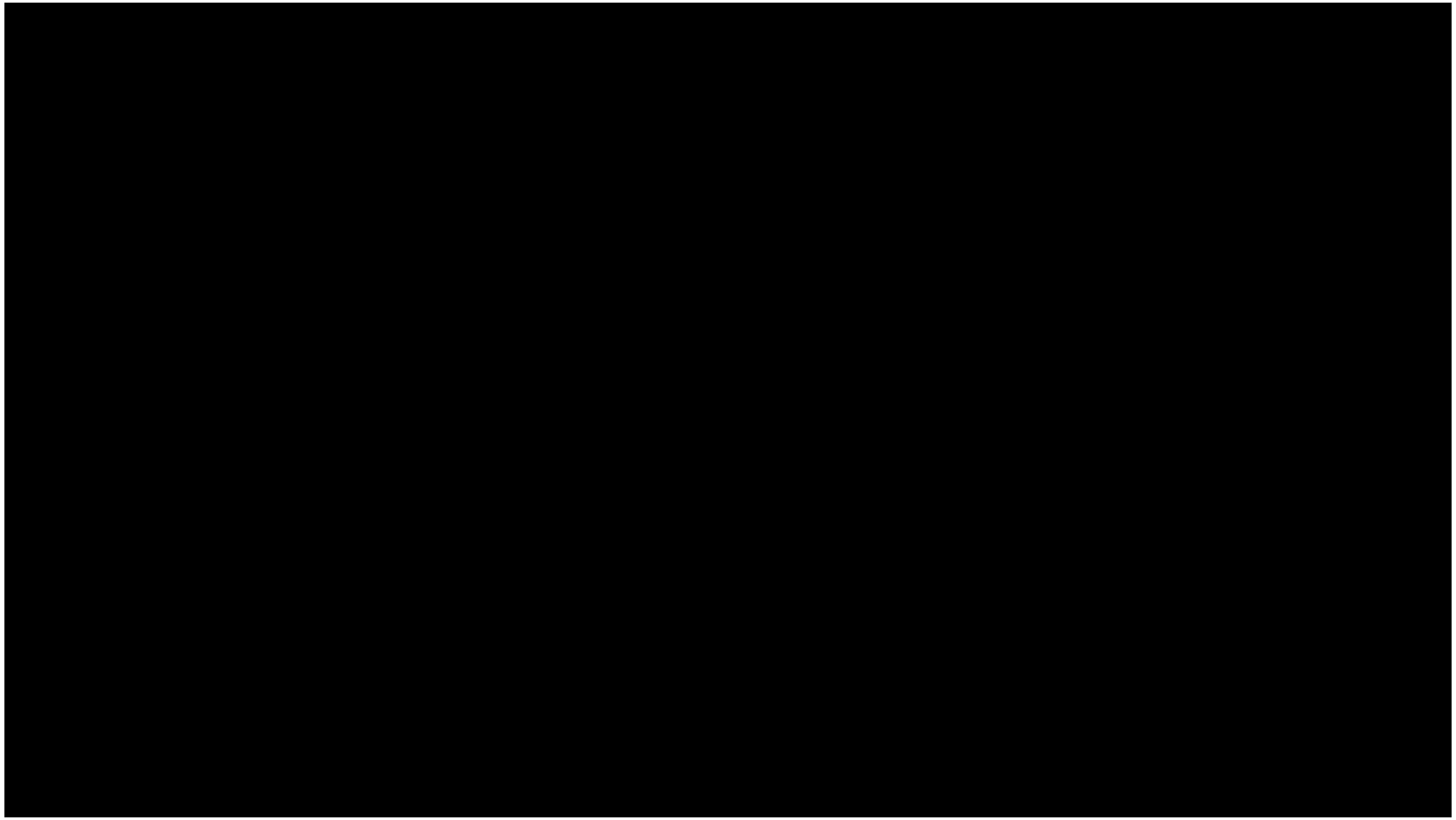
**Start with first element, and move forward:**

- If the current and next items are in order, continue with the next item, otherwise
- Swap the two entries

- After one iteration, largest element is in the last location.
- Rinse and repeat, stopping one position shorter each iteration.



# Thanks, Obama



# Moving on to $O(n \log n)$

**Two classics:** Mergesort and Quicksort

5 Notes
6 References

## Examples [ edit ]

Some of the most well-known comparison sorts include:

- Quicksort
- Heapsort
- Shellsort
- Merge sort
- Introsort
- ~~Insertion sort~~
- ~~Selection sort~~
- ~~Bubble sort~~
- Odd–even sort
- Cocktail shaker sort
- Cycle sort
- Merge insertion (Ford–Johnson) sort
- Smoothsort
- Timsort

## Performance limits and advantages of different sorting techniques [ edit ]

There are fundamental limits on the performance of comparison sorts. A comparison sort must have an average-case lower bound Ω(…
known as linearithmic time. This is a consequence of the limited information available through comparisons alone — or, to put it different…

# **Merge Sort:** Algorithm

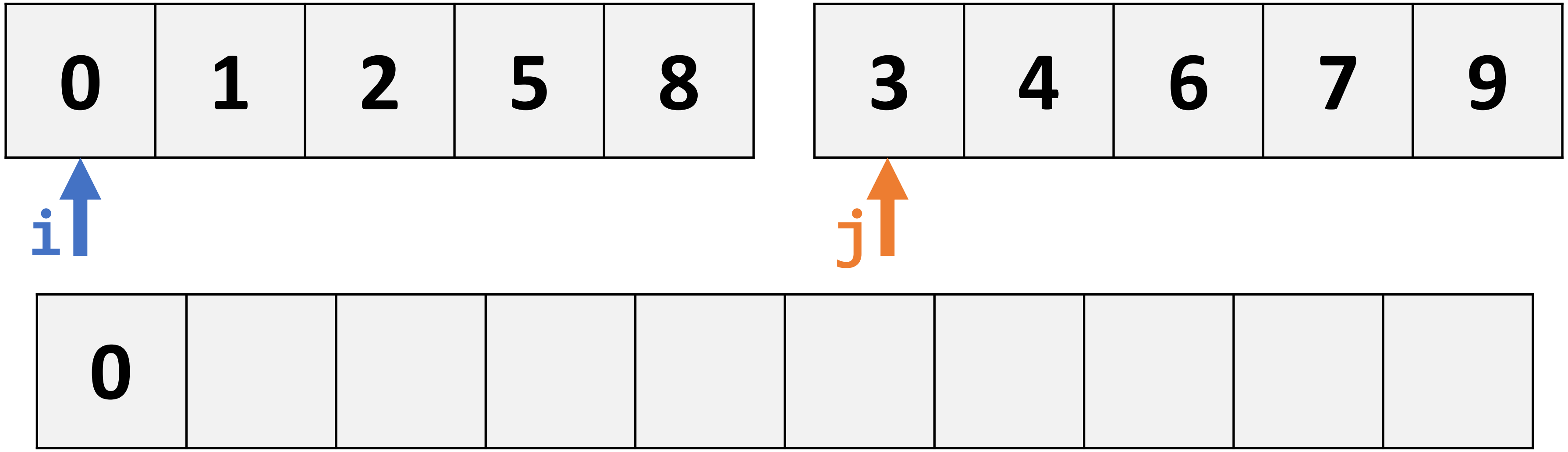
Recursive *Divide-and-Conquer*:

**Simple in concept:**

- Divide array into two halves
- Recursively sort each half
- Base case: list of size 1 is sorted
- Merge halves.

# **Merge Sort:** Merging

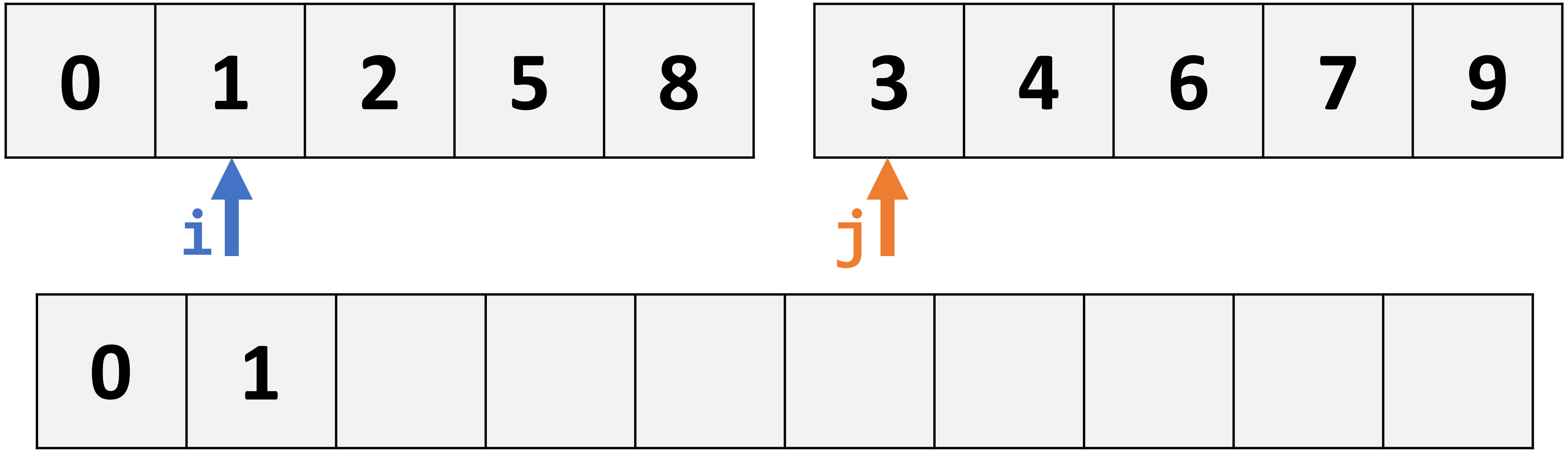
- First, let’s assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




`arr1[i] < arr2[j]?`

# **Merge Sort:** Merging

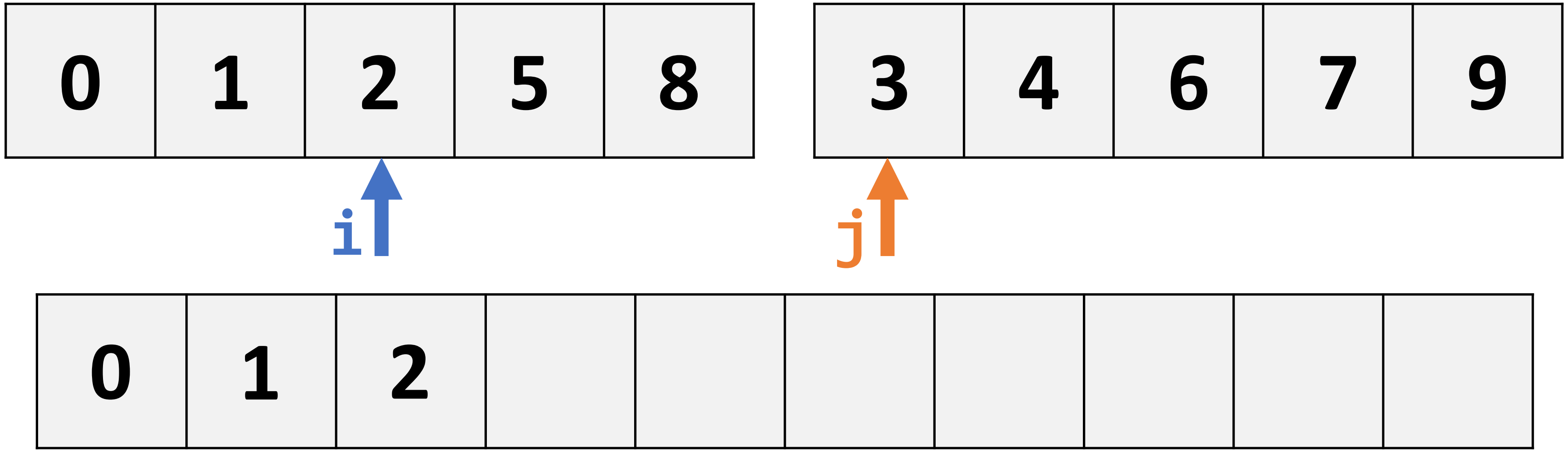
- First, let's assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




**`arr1[i] < arr2[j]?`**

# **Merge Sort:** Merging

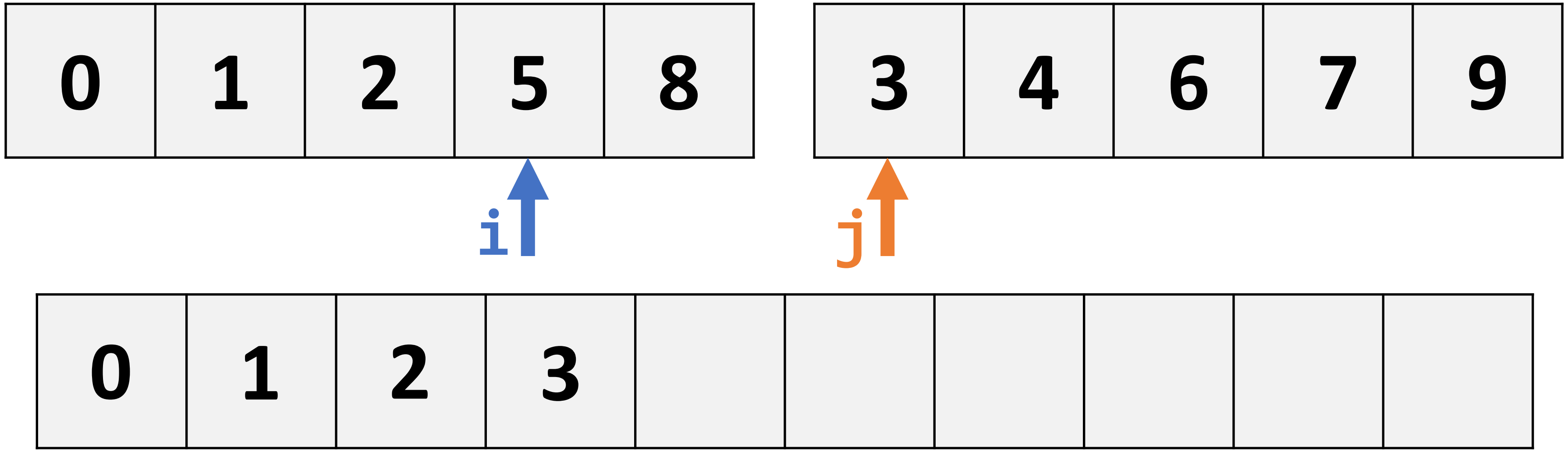
- First, let's assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




`arr1[i] < arr2[j]?`

# **Merge Sort:** Merging

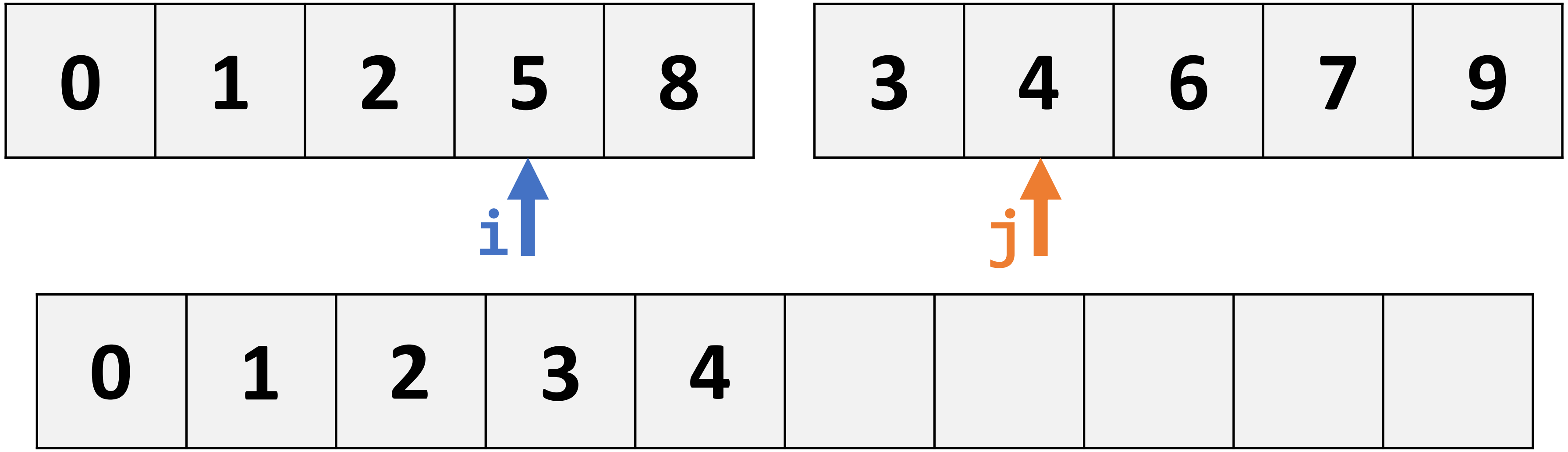
- First, let’s assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




**`arr1[i] < arr2[j]?`**

# **Merge Sort:** Merging

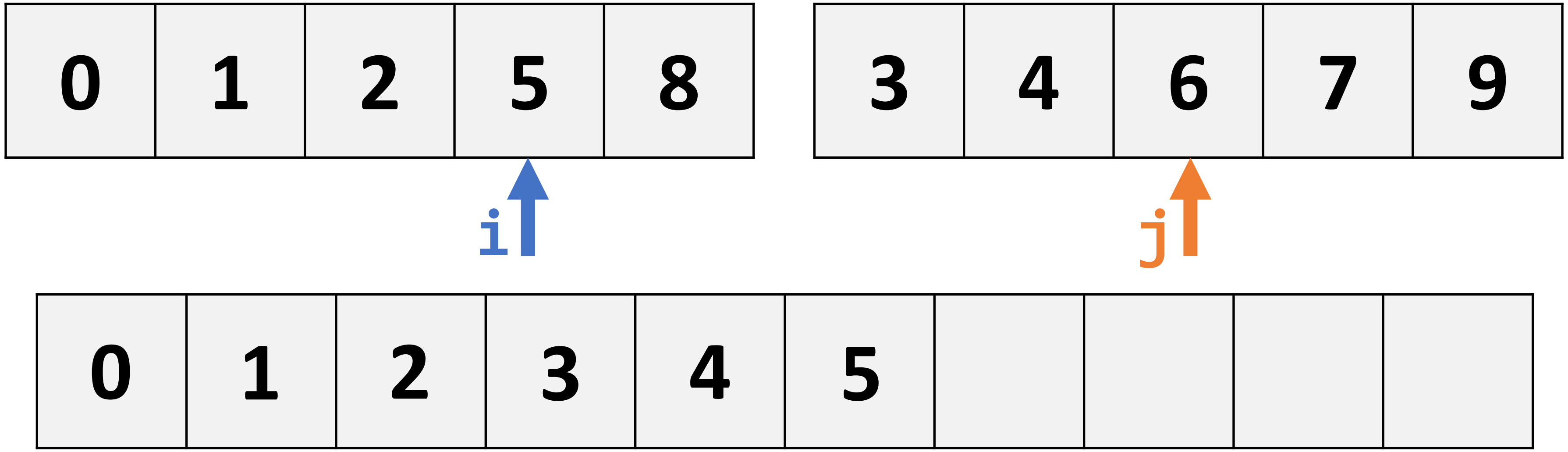
- First, let's assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




**`arr1[i] < arr2[j]?`**

# **Merge Sort:** Merging

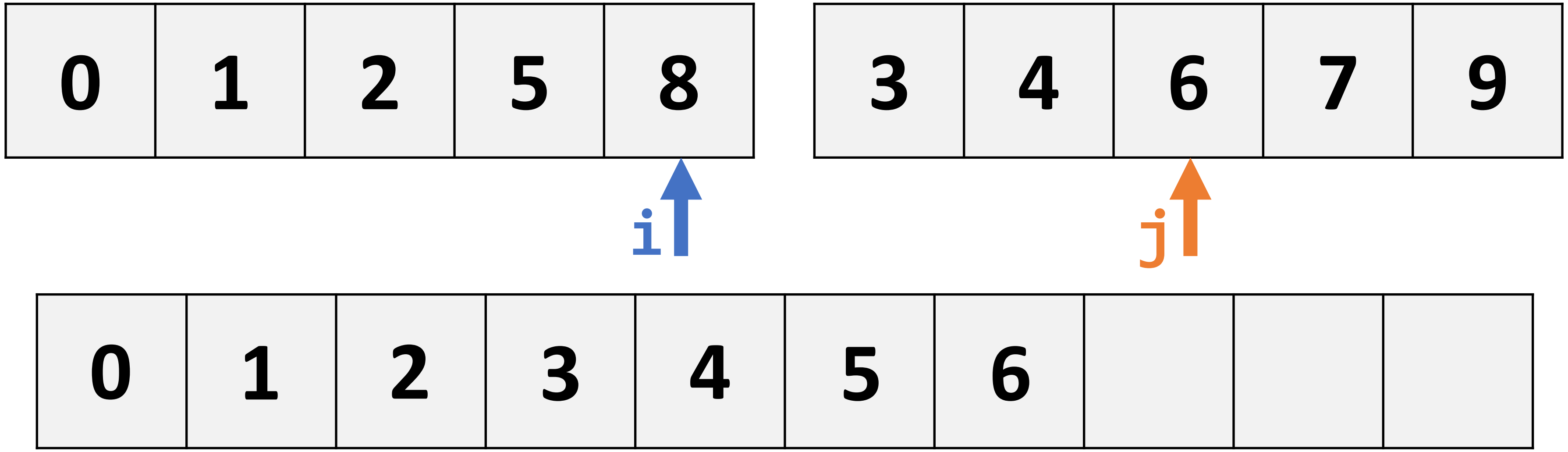
- First, let's assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




**`arr1[i] < arr2[j]?`**

# **Merge Sort:** Merging

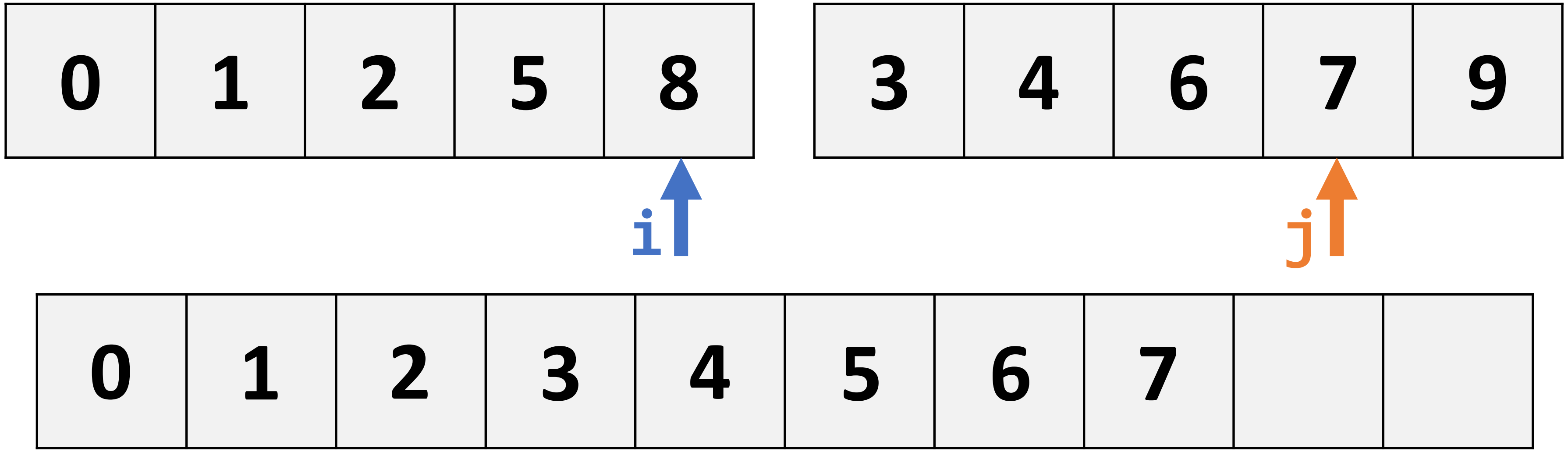
- First, let’s assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




```
arr1[i] < arr2[j]?
```

# **Merge Sort:** Merging

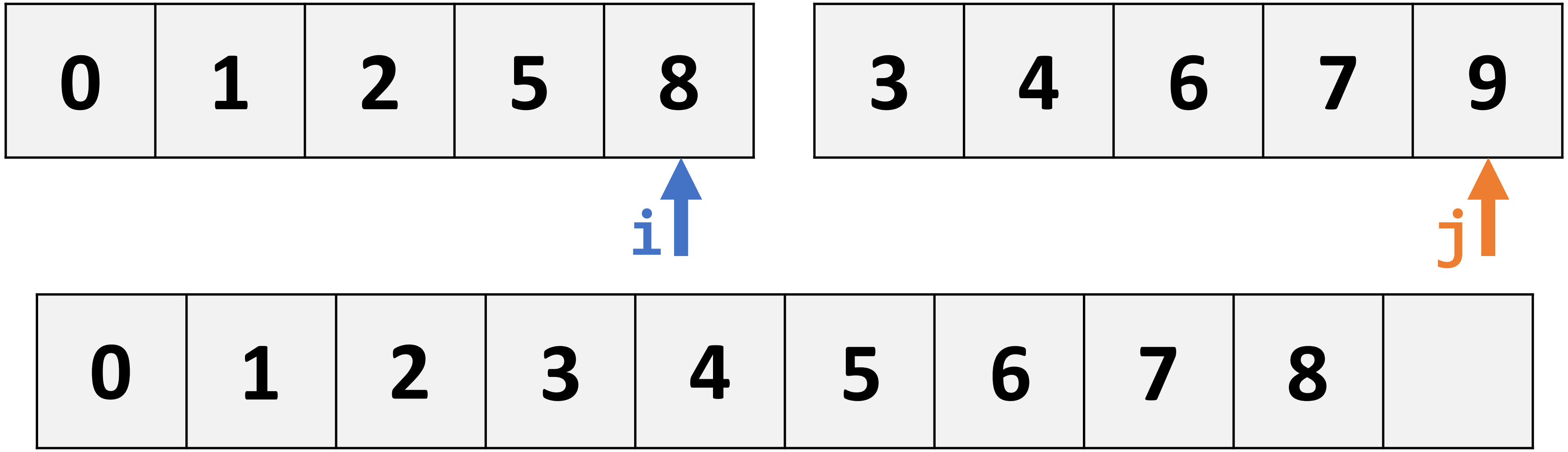
- First, let’s assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




**`arr1[i] < arr2[j]?`**

# **Merge Sort:** Merging

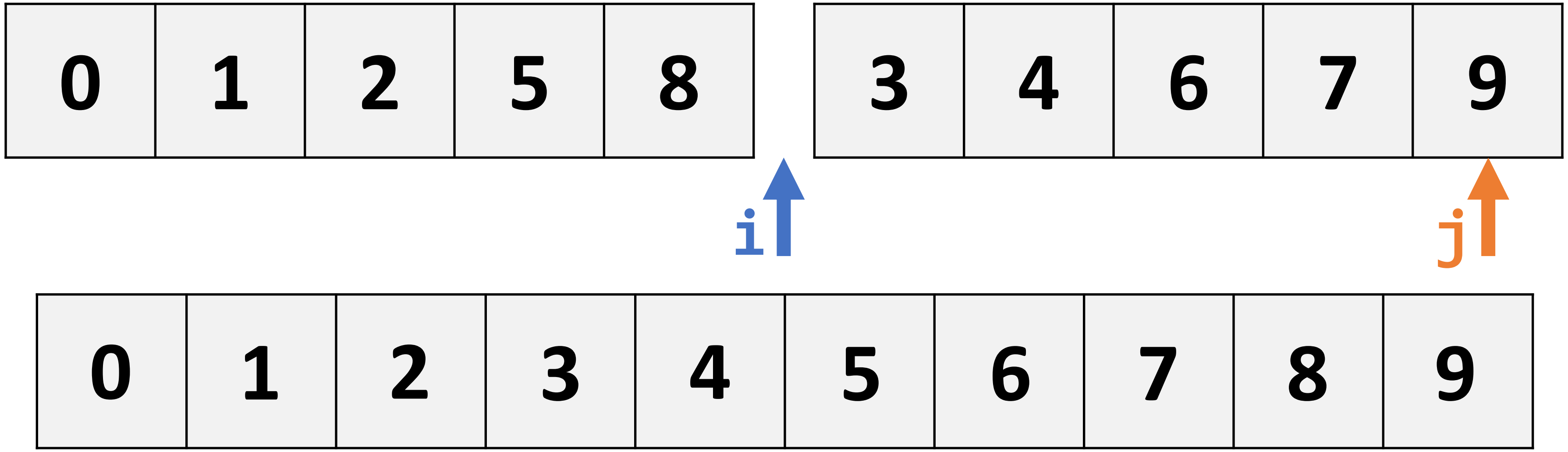
- First, let's assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




```
arr1[i] < arr2[j]?
```

# **Merge Sort:** Merging

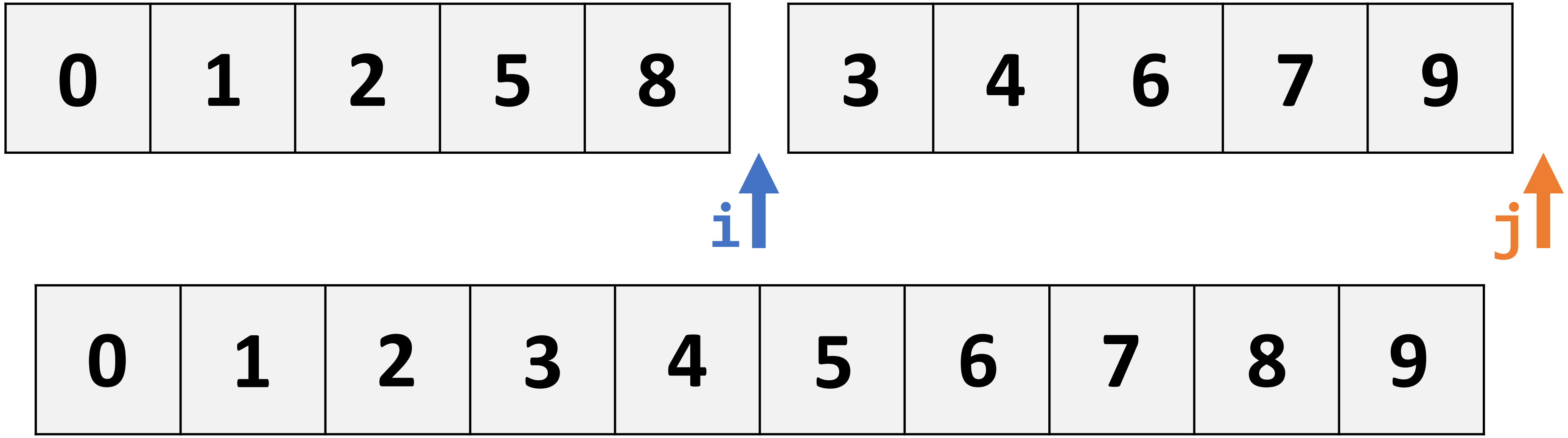
- First, let’s assume we have two sorted halves.
- How do we merge them? What is the cost of merging?
- Use an auxiliary array and two indexes:




**`arr1[i] < arr2[j]?`**

# **Merge Sort:** Merging

- Eventually, both indexes will reach the end of their respective halves.
- Each element, in each half, is visited exactly once.




`arr1[i] < arr2[j]?`

# **Merging:** Analysis

**Given two arrays of size $n1$ and $n2$:**

- The body of the merge loop will run a total of $n1 + n2$ times
- Hence, merging may be performed in $O(n1 + n2)$ time

If the arrays are approximately the same size, $N = n1$, $n1 \approx n2$, we can say that the run time is $O(2N) = \Theta(N)$

**Caveat:** We cannot merge two arrays in-place (*in O(n)*)

- This algorithm requires the allocation of a new array
- Therefore, the memory requirements are also $O(n)$

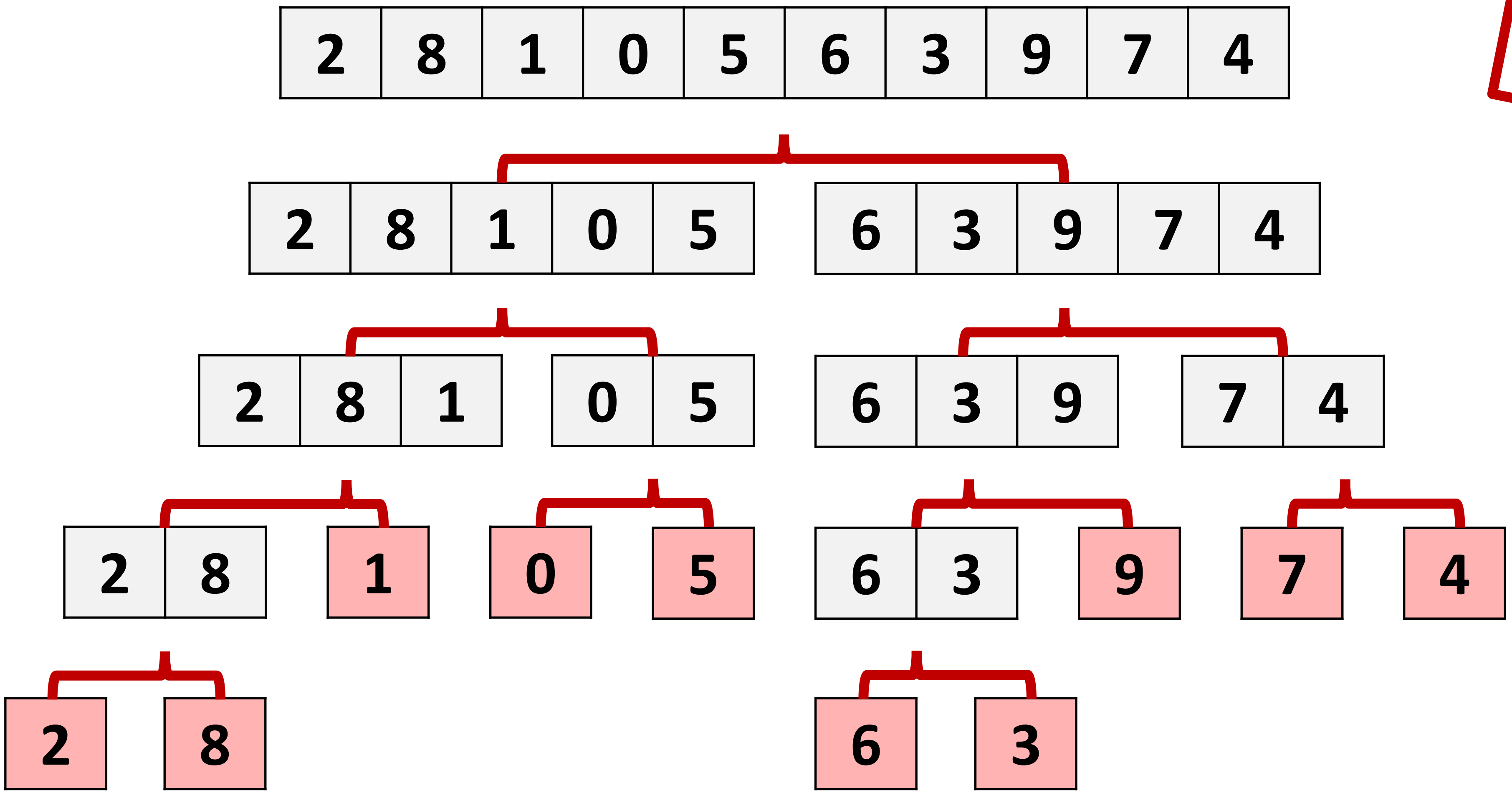
# **Splitting:** Analysis

We've seen the merging step, what about splitting?

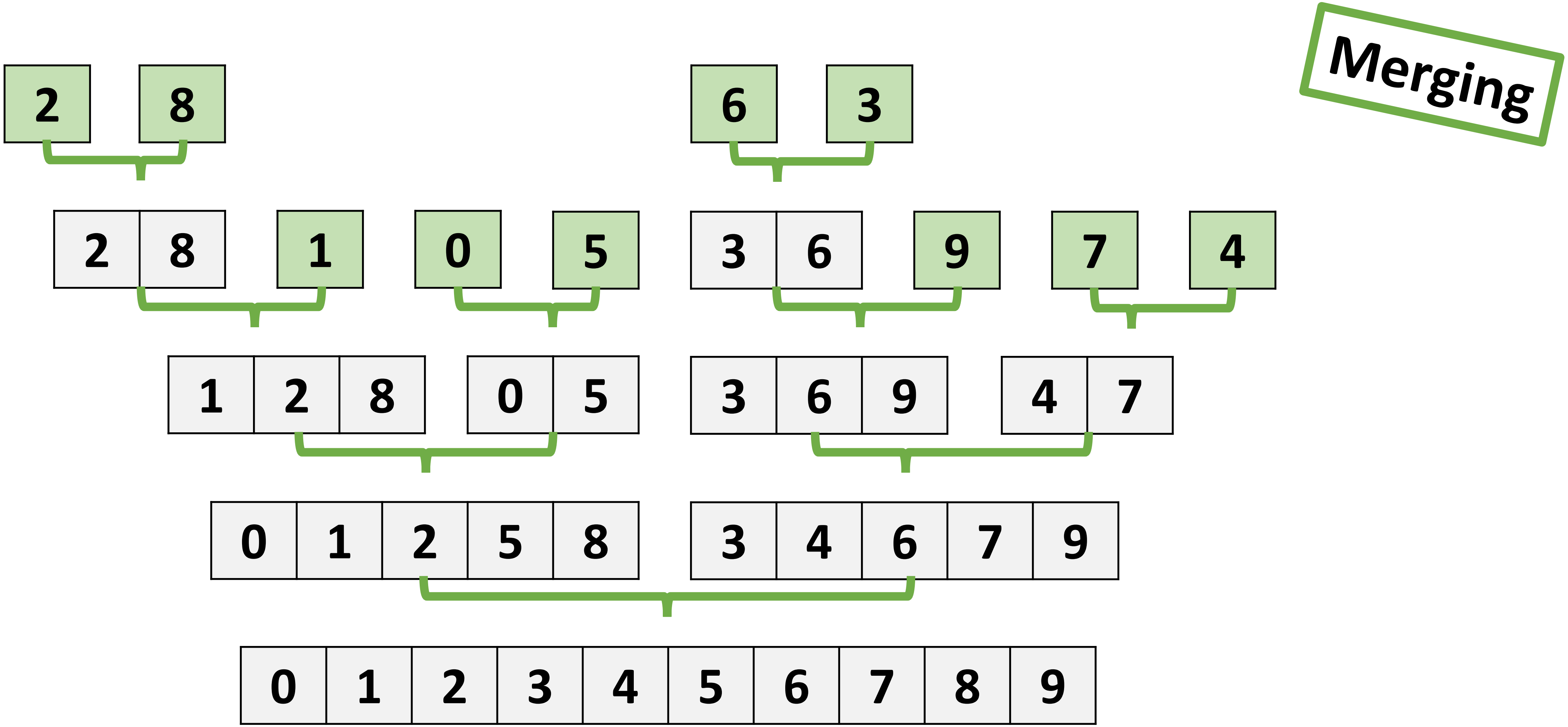
- Recursively split the array into smaller and smaller subarrays
- Stop splitting when subarray is a single element.
- At this scale, merging subarrays is a single comparison.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **2** | **8** | **1** | **0** | **5** | **6** | **3** | **9** | **7** | **4** |

**Recursive Splitting**



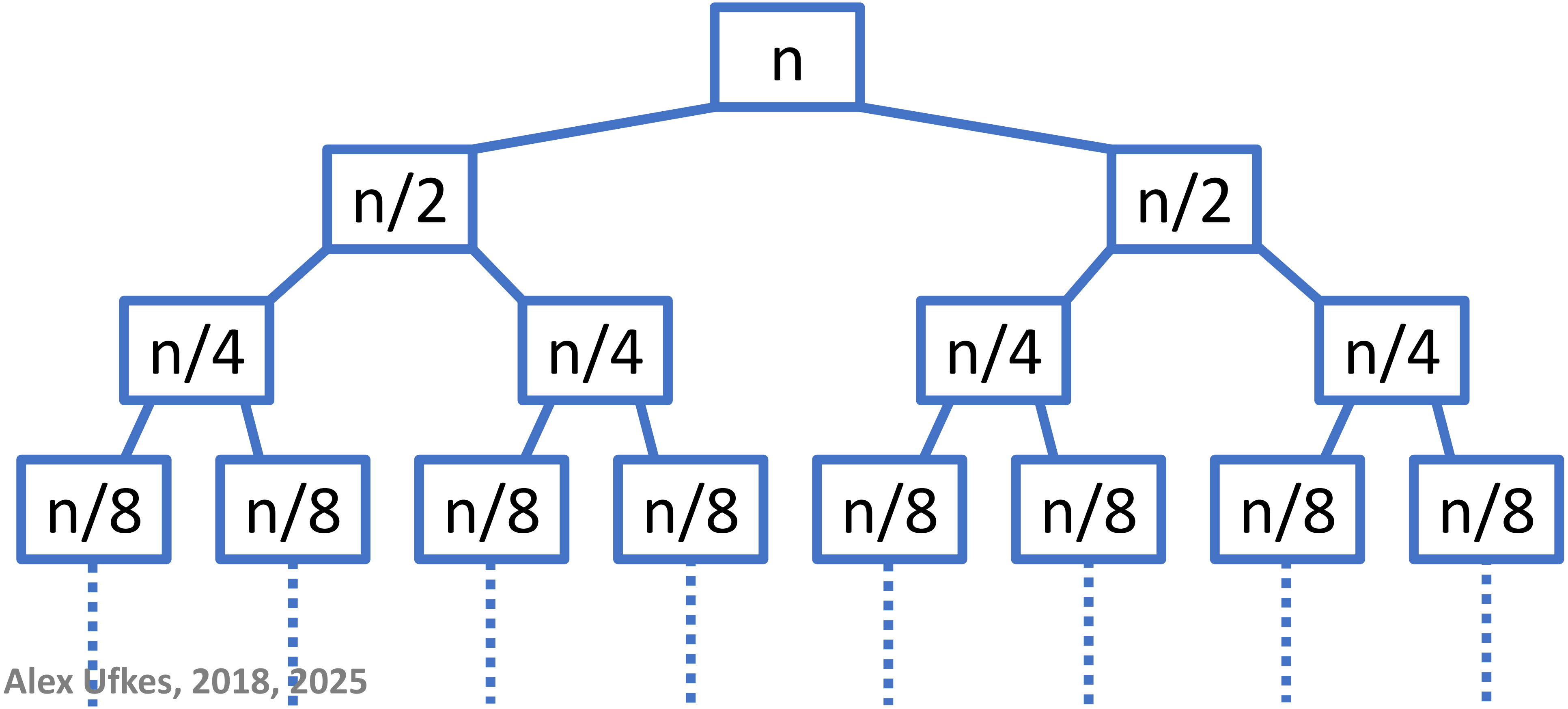

Stop splitting when each subarray contains only a single element




When recursive split hits base case, we can begin merging.

# **Merge Sort:** Analysis

- Total number of comparisons = sum of all merges
- How many comparisons in each merge?

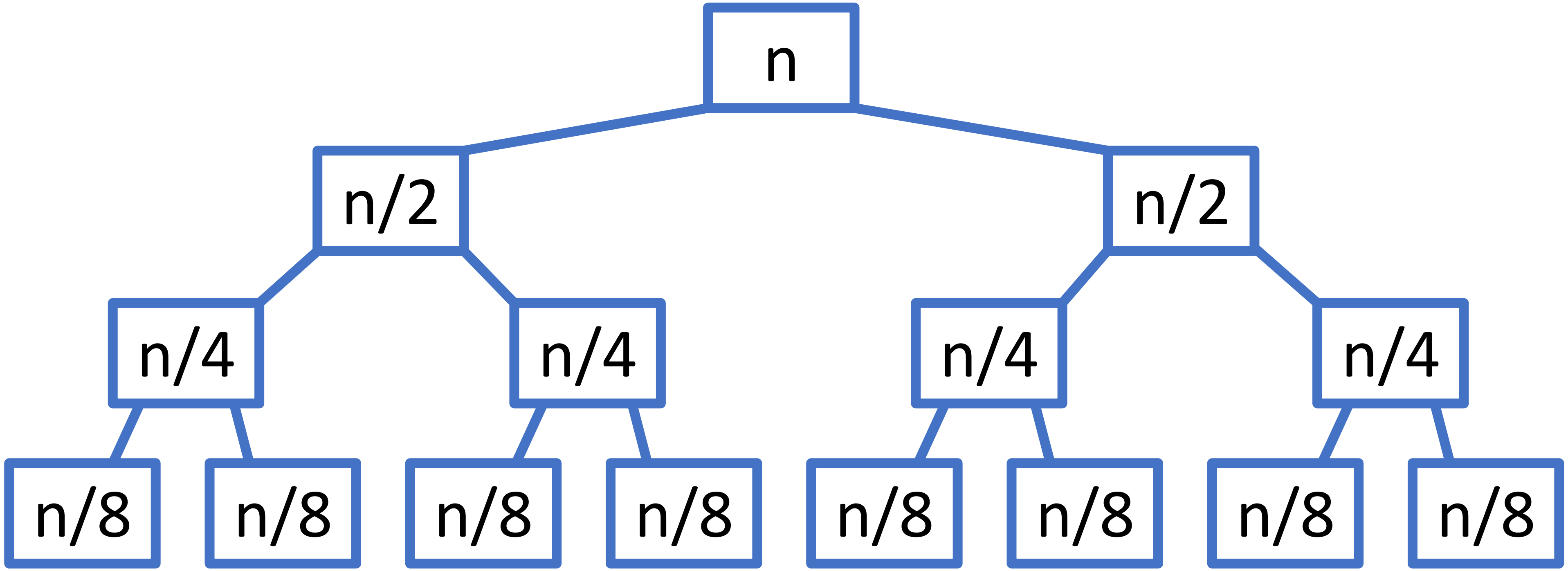



$$n$$

$$2\frac{n}{2} = n$$

$$4\frac{n}{4} = n$$

$$8\frac{n}{8} = n$$




$$n$$

$$2\frac{n}{2} = n$$

$$4\frac{n}{4} = n$$

$$8\frac{n}{8} = n$$

Height of tree is $\mathbf{log_2 n}$. Thus, $\mathbf{n log_2 n}$ comparisons.

# Sorting Recursively?


- We split the list into two sub-lists and sort them
- *How* should we sort those lists?

**Answer (theoretical):**

- If the size of these sub-lists is >1, use Merge sort again.
- If the sub-lists are of length 1, do nothing. A list of length 1 is sorted.

# Sorting Recursively

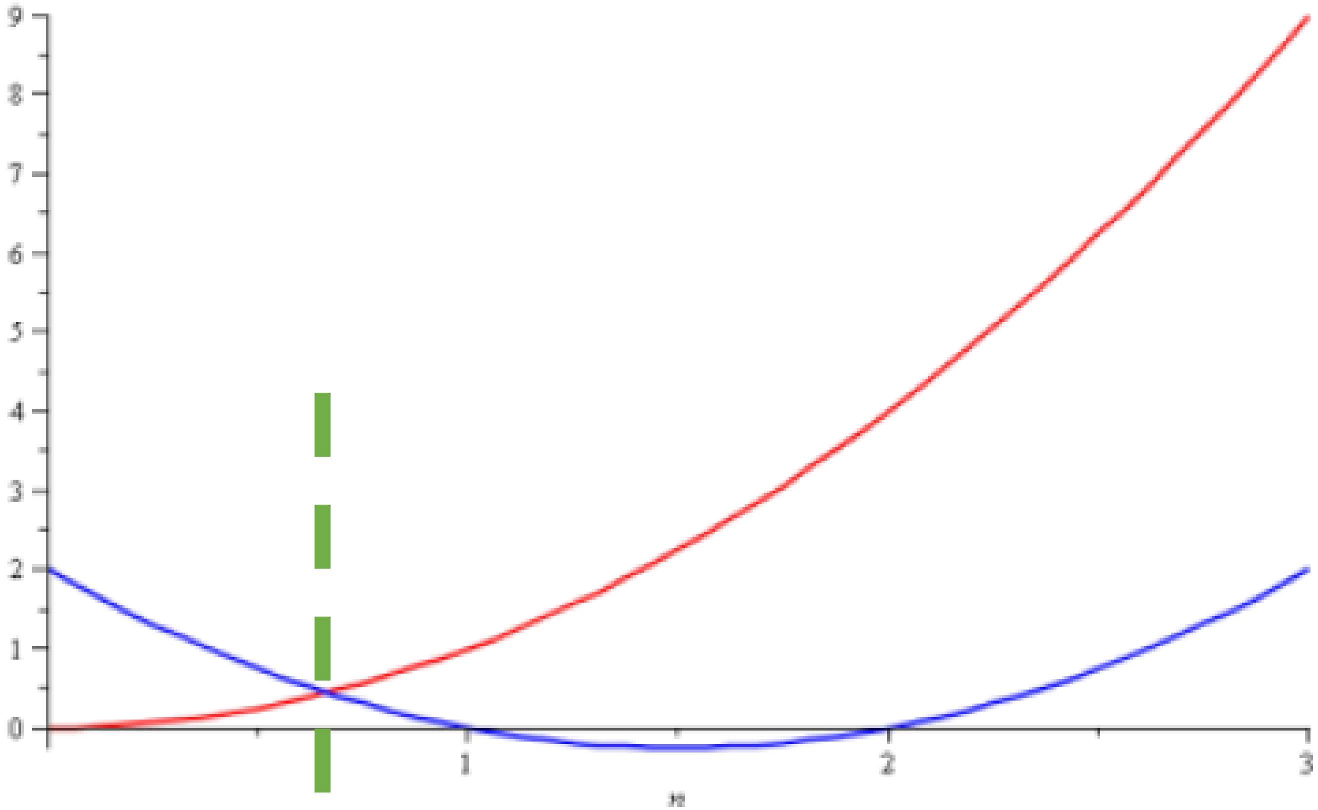
**Answer (theoretical):**

- If the size of these sub-lists is >1, use Merge sort again.
- If the sub-lists are of length 1, return. A list of length 1 is sorted.

**In practice?**

- Just because an algorithm has good asymptotic properties, doesn't mean it's good at all problem sizes.
- Asymptotic complexity describes behavior as size approaches infinity.

- In practice, once the sub-lists are shorter than some threshold, we use an algorithm with lower overhead.
- E.g. Insertion sort. Otherwise use Merge sort again.



# Example

Consider the following array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 77 | 49 | 35 | 61 | 48 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

If the sub-array to sort is of size 6 or less, we will call insertion sort.

# Example

**Current call:** `merge_sort(array, 0, 25)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 77 | 49 | 35 | 61 | 48 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Size of list? 25-0 = **25** elements, and 25 > 6.
- Thus we find the midpoint and call merge_sort recursively.

`merge_sort(array, 0, 12)`

`merge_sort(array, 0, 25)`

# Example

**Current call:** `merge_sort(array, 0, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 77 | 49 | 35 | 61 | 48 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Size of list? 12-0 = **12** elements, and 12 > 6.
- Thus we find the midpoint and call merge_sort recursively.

```
merge_sort(array, 0, 6)
```

```
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 0, 6)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 77 | 49 | 35 | 61 | 48 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Size of list? 6-0 = **6** elements, and 6 <= 6!
- Thus we call insertion sort on this sub-list.

```
insertion_sort(array, 0, 6)
```

```
merge_sort(array, 0, 6)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `insertion_sort(array, 0, 6)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 77 | 49 | 35 | 61 | 48 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Insertion sort will simply sort elements 0 – 5

```
insertion_sort(array, 0, 6)
merge_sort(array, 0, 6)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `insertion_sort(array, 0, 6)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Insertion sort will simply sort elements 0 – 5
- Sorting complete, function call returns.

```
insertion_sort(array, 0, 6)
merge_sort(array, 0, 6)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 0, 6)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge_sort(array, 0, 6)` is also complete. It returns.

```
merge_sort(array, 0, 6)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 0, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge_sort(array, 0, 12)` is not done yet!

```
merge_sort(array, 0, 6)
merge_sort(array, 6, 12)
```

```
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 6, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Size of list? 12-6 = **6** elements, and 6 <= 6!
- Thus we call insertion sort on this sub-list.

```
merge_sort(array, 6, 12)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `insertion_sort(array, 6, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 73 | 23 | 95 | 3 | 89 | 37 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Insertion sort will simply sort elements 6 – 11

```
insertion_sort(array, 6, 12)
merge_sort(array, 6, 12)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `insertion_sort(array, 6, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 3 | 23 | 37 | 73 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- Insertion sort will simply sort elements 6 – 11
- Sorting complete, function call returns.

```
insertion_sort(array, 6, 12)
merge_sort(array, 6, 12)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 6, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 3 | 23 | 37 | 73 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge_sort(array, 6, 12)` returns.

```
merge_sort(array, 6, 12)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 0, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 3 | 23 | 37 | 73 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge_sort(array, 0, 12)` is not done yet!

```
merge_sort(array, 0, 6)
merge_sort(array, 6, 12)
merge(array, 0, 6, 12)
```

```
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge(array, 0, 6, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 35 | 48 | 49 | 61 | 77 | 3 | 23 | 37 | 73 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge(array, 0, 6, 12)` does its thing.

```
merge(array, 0, 6, 12)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge(array, 0, 6, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 13 | 23 | 35 | 37 | 48 | 49 | 61 | 73 | 77 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge(array, 0, 6, 12)` does its thing.
- Call to merge() returns

```
merge(array, 0, 6, 12)
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 0, 12)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 13 | 23 | 35 | 37 | 48 | 49 | 61 | 73 | 77 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge_sort(array, 0, 12)` is finally done.

```
merge_sort(array, 0, 6)
merge_sort(array, 6, 12)
merge(array, 0, 6, 12)
```

```
merge_sort(array, 0, 12)
merge_sort(array, 0, 25)
```

# Example

**Current call:** `merge_sort(array, 0, 25)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 13 | 23 | 35 | 37 | 48 | 49 | 61 | 73 | 77 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

- `merge_sort(array, 0, 25)` is not yet done.

```
merge_sort(array, 0, 12)
merge_sort(array, 12, 25)
merge(array, 0, 12, 25)
```

`merge_sort(array, 0, 25)`

# Example

**Current call:** `merge_sort(array, 12, 25)`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 13 | 23 | 35 | 37 | 48 | 49 | 61 | 73 | 77 | 89 | 95 | 57 | 99 | 17 | 32 | 94 | 28 | 15 | 55 | 7 | 51 | 88 | 97 | 62 |

**And so on, and so forth.**

**What about run time?**

```
merge_sort(array, 12, 25)
merge_sort(array, 0, 25)
```

# Updated Complexity?

**NO!** This doesn't change the analysis. Mergesort is still **O(nlogn)**.

- We're calling Insertion sort for a known, small array size (i.e. n=6)
- Thus, the cost of using Insertion sort can be considered ***constant***.
- It doesn't scale with the size of the array, because we're only calling it below a fixed threshold.

**If we count the number of machine instructions:**

- Sort 6 items using Insertion sort
- Sort 6 items using Mergesort (recursive down to n=1)
- We will find that **Insertion** is faster than **Merge!**

# **Merge Sort:** Analysis

*Of course…*

- The value n=6 was chosen arbitrarily for this example.
- The best size at which to switch to insertion sort can vary based on implementation, machine architecture, compiler optimization, etc.

**When calling Arrays.sort() in Java:**

- Array of objects? Fancy combination of Merge and Insertion sorting, known as ***Timsort***.
- Array of primitives? Java uses a dual pivot quicksort.

# Stable?

**Meaning:** Items that are equal do not change their relative ordering.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 1 | 0 | 5 | 6 | 5 | 9 | 7 | 4 |

- In the sorted list, these values will not have changed order.
- Useless for primitives, but VERY useful for objects. We can have multiple sub-orderings.
- Sort students based on course first, section second, last name third.
- If we begin by sorting by last name, and then re-sort based on section, the last names will still be in order *within each section*

**sorted by time**

| | |
|---|---|
| Chicago | 09:00:00 |
| Phoenix | 09:00:03 |
| Houston | 09:00:13 |
| Chicago | 09:00:59 |
| Houston | 09:01:10 |
| Chicago | 09:03:13 |
| Seattle | 09:10:11 |
| Seattle | 09:10:25 |
| Phoenix | 09:14:25 |
| Chicago | 09:19:32 |
| Chicago | 09:19:46 |
| Chicago | 09:21:05 |
| Seattle | 09:22:43 |
| Seattle | 09:22:54 |
| Chicago | 09:25:52 |
| Chicago | 09:35:21 |
| Seattle | 09:36:14 |
| Phoenix | 09:37:44 |

**sorted by location (not stable)**

| | |
|---|---|
| Chicago | 09:25:52 |
| Chicago | 09:03:13 |
| Chicago | 09:21:05 |
| Chicago | 09:19:46 |
| Chicago | 09:19:32 |
| Chicago | 09:00:00 |
| Chicago | 09:35:21 |
| Chicago | 09:00:59 |
| Houston | 09:01:10 |
| Houston | 09:00:13 |
| Phoenix | 09:37:44 |
| Phoenix | 09:00:03 |
| Phoenix | 09:14:25 |
| Seattle | 09:10:25 |
| Seattle | 09:36:14 |
| Seattle | 09:22:43 |
| Seattle | 09:10:11 |
| Seattle | 09:22:54 |

*no longer sorted by time*

**sorted by location (stable)**

| | |
|---|---|
| Chicago | 09:00:00 |
| Chicago | 09:00:59 |
| Chicago | 09:03:13 |
| Chicago | 09:19:32 |
| Chicago | 09:19:46 |
| Chicago | 09:21:05 |
| Chicago | 09:25:52 |
| Chicago | 09:35:21 |
| Houston | 09:00:13 |
| Houston | 09:01:10 |
| Phoenix | 09:00:03 |
| Phoenix | 09:14:25 |
| Phoenix | 09:37:44 |
| Seattle | 09:10:11 |
| Seattle | 09:10:25 |
| Seattle | 09:22:43 |
| Seattle | 09:22:54 |
| Seattle | 09:36:14 |

*still sorted by time*

# Next Up

**Quicksort:**

- Faster on average than Mergesort

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."

**- C.A.R. Hoare (Quicksort creator)**

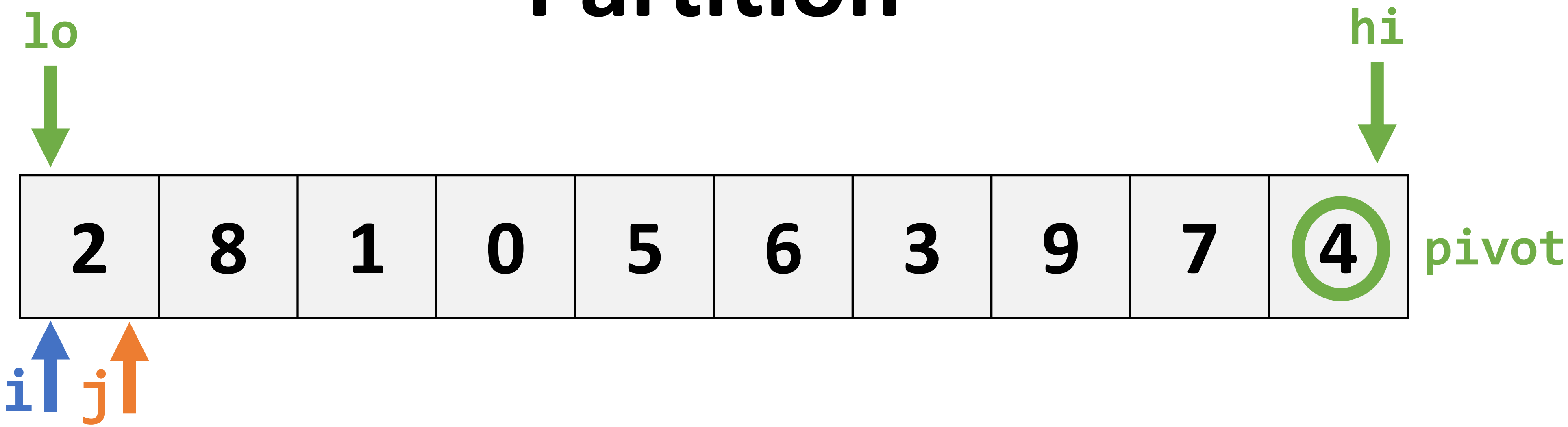
# Quick Sort

1. Pick an element, called the pivot, from the array.
2. <u>Partitioning</u> - Rearrange the array so that:
   - every element *smaller* than the pivot is to the left.
   - every element *larger* than the pivot is to the right.
3. Apply above steps recursively to each resulting subarray
   - Base case? Subarrays of size 0 or 1.

Pivot choice and partitioning can be done in several different ways, the choice of which greatly affects performance.

**Lomuto Scheme**

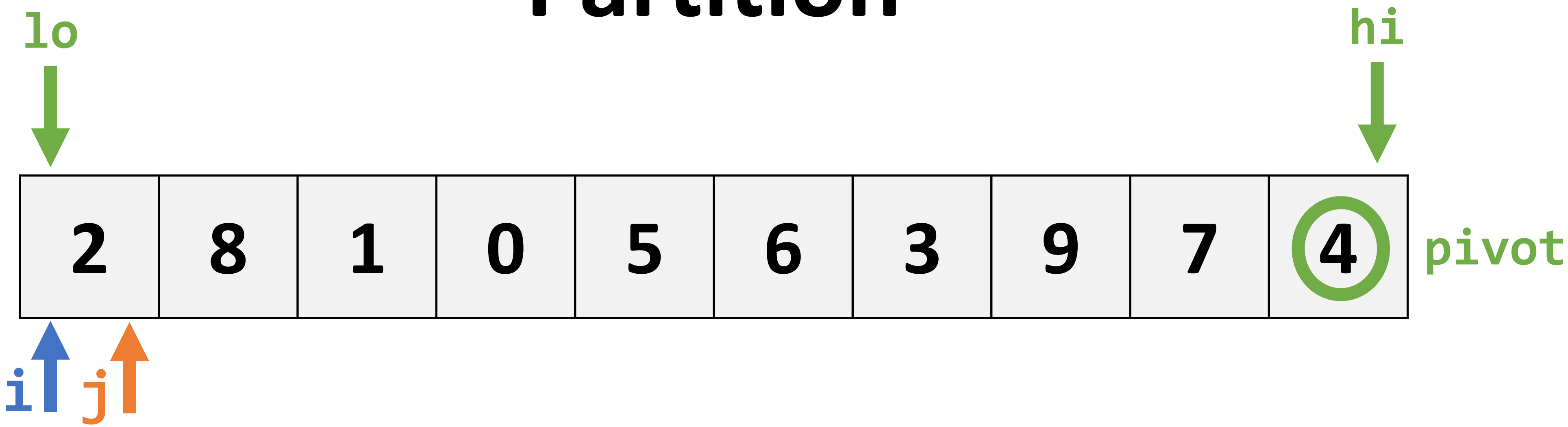
# Partition




- Select right-most element as pivot
- Maintain two indexes, **`i`** and **`j`**.
- First index = **`lo`**
- Last index = **`hi`**

Lomuto Scheme

# Partition



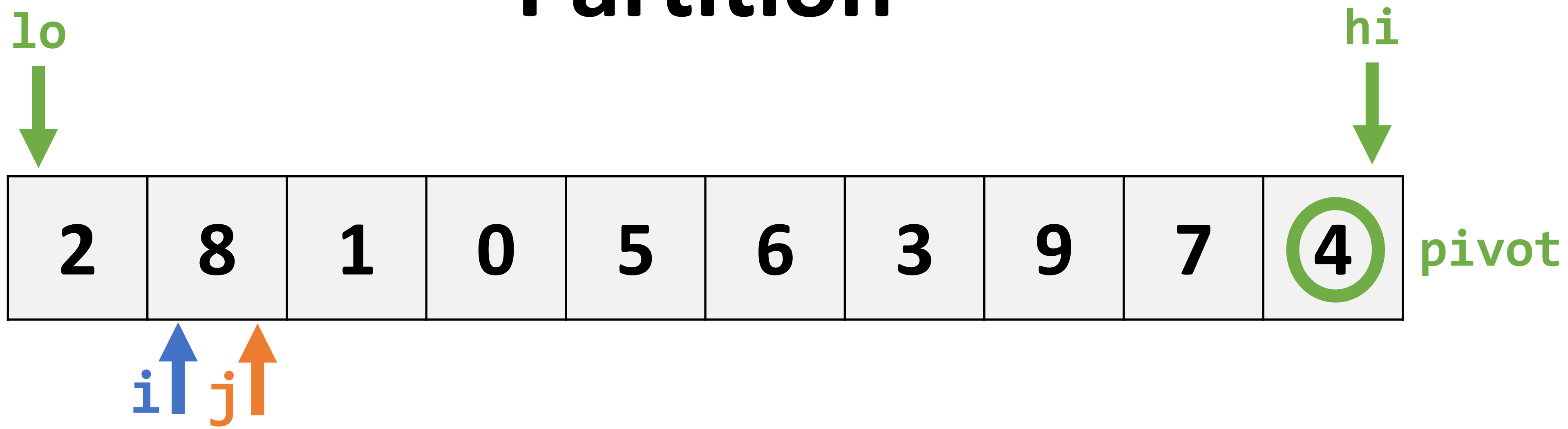

If `A[j] < pivot`:
- Swap `A[i]` and `A[j]`
- Increment `i`

Always increment `j`

**Stop when `j == hi`**

**Lomuto Scheme**

# Partition



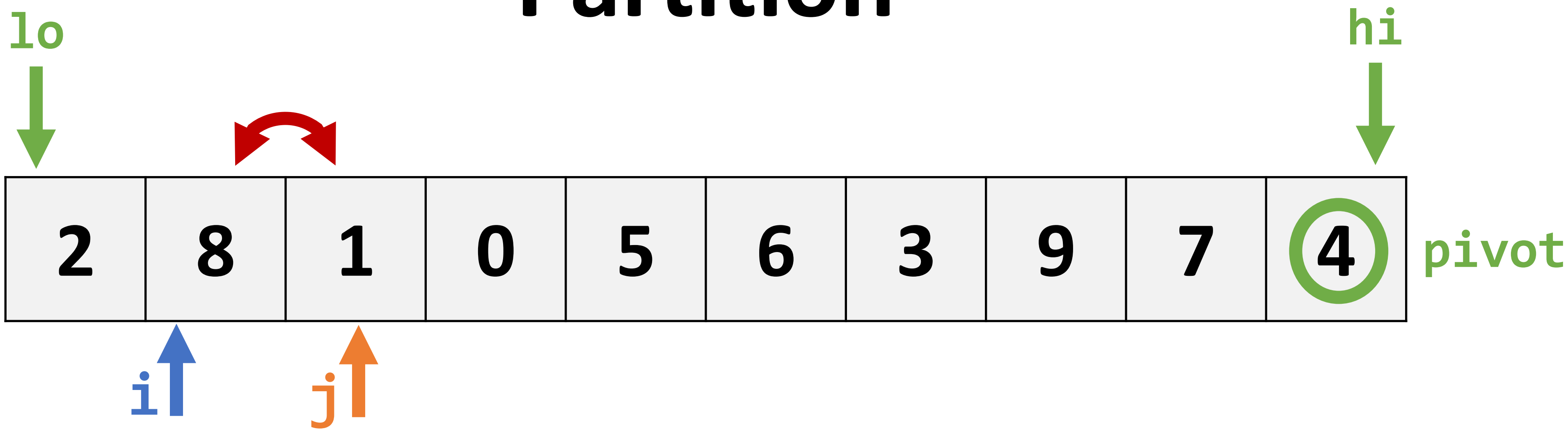

If `A[j] < pivot`:
- Swap `A[i]` and `A[j]`
- Increment `i`

Always increment `j`

**Stop when `j == hi`**

Lomuto Scheme

# Partition



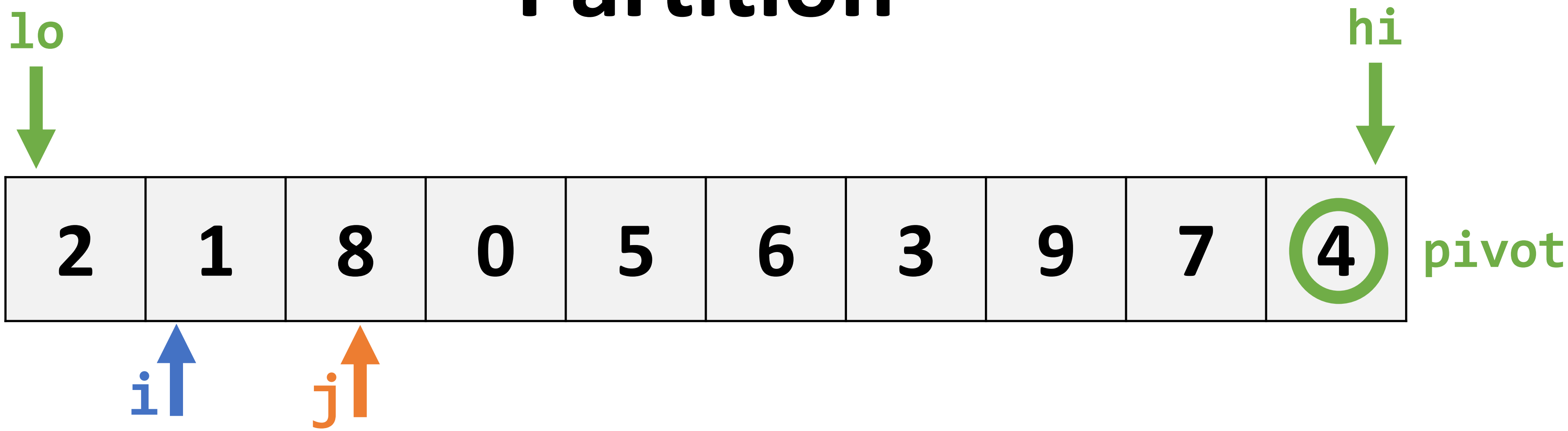

If `A[j] < pivot`:
- Swap `A[i]` and `A[j]`
- Increment `i`

Always increment `j`

**Stop when `j == hi`**

**Lomuto Scheme**

# Partition



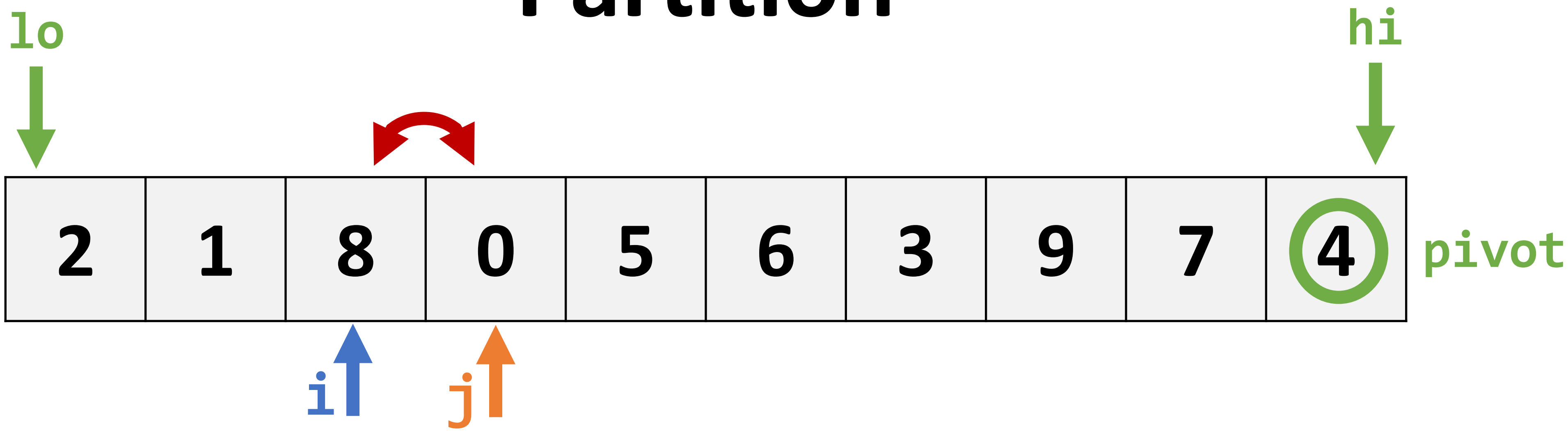

If `A[j] < pivot`:

- Swap `A[i]` and `A[j]`
- Increment `i`

Always increment `j`

**Stop when `j == hi`**


# Partition



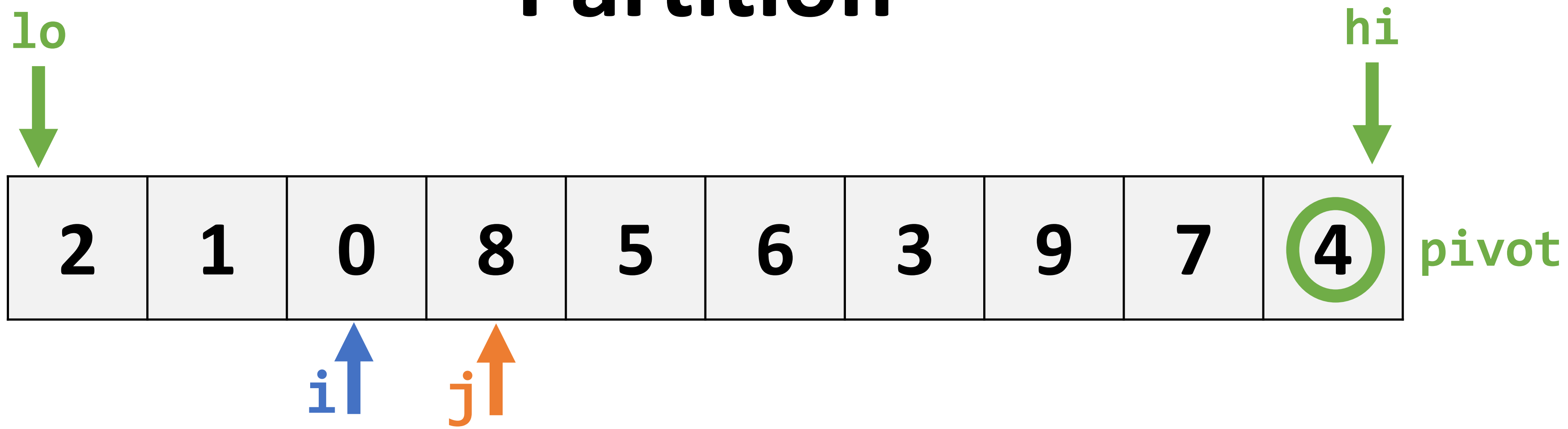

If A[j] < pivot:
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

Lomuto Scheme

# Partition




If A[j] < pivot:

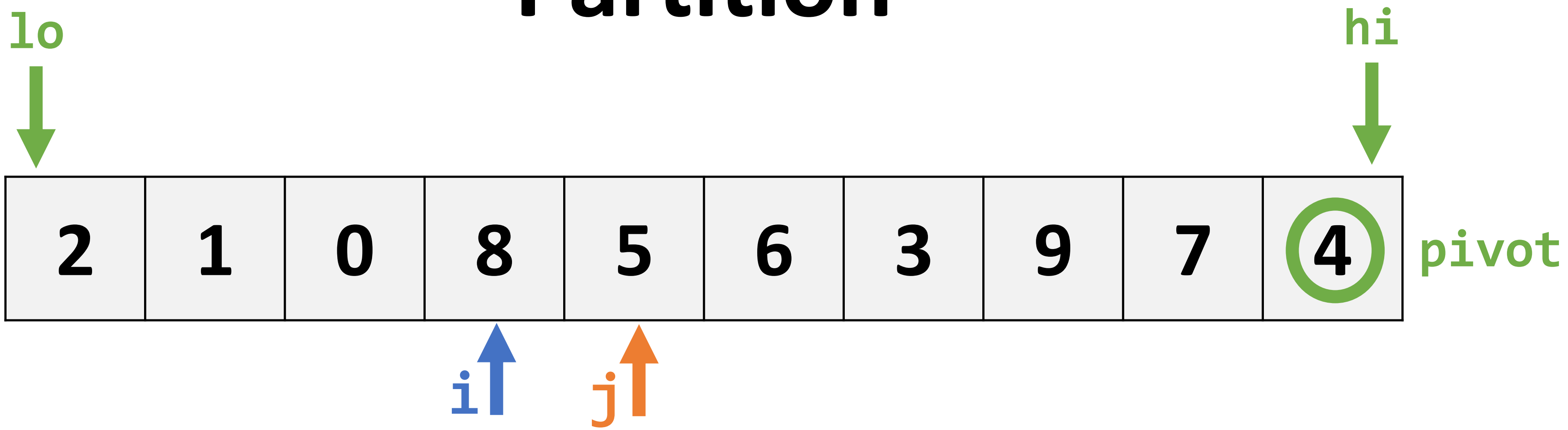
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

**Lomuto Scheme**

# Partition



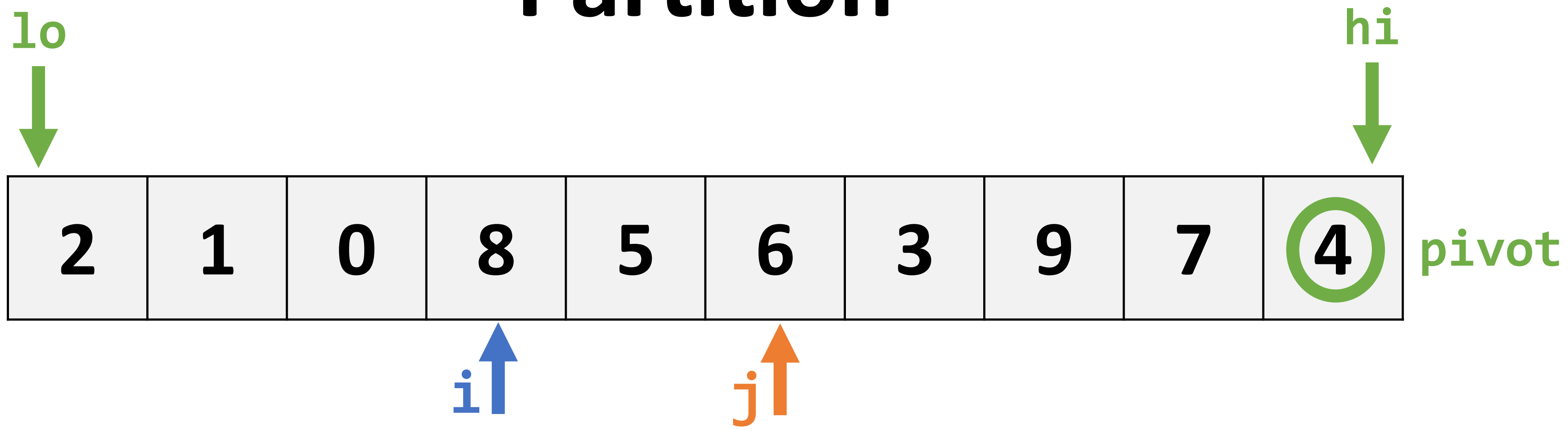

If A[j] < pivot:
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

Lomuto Scheme

# Partition



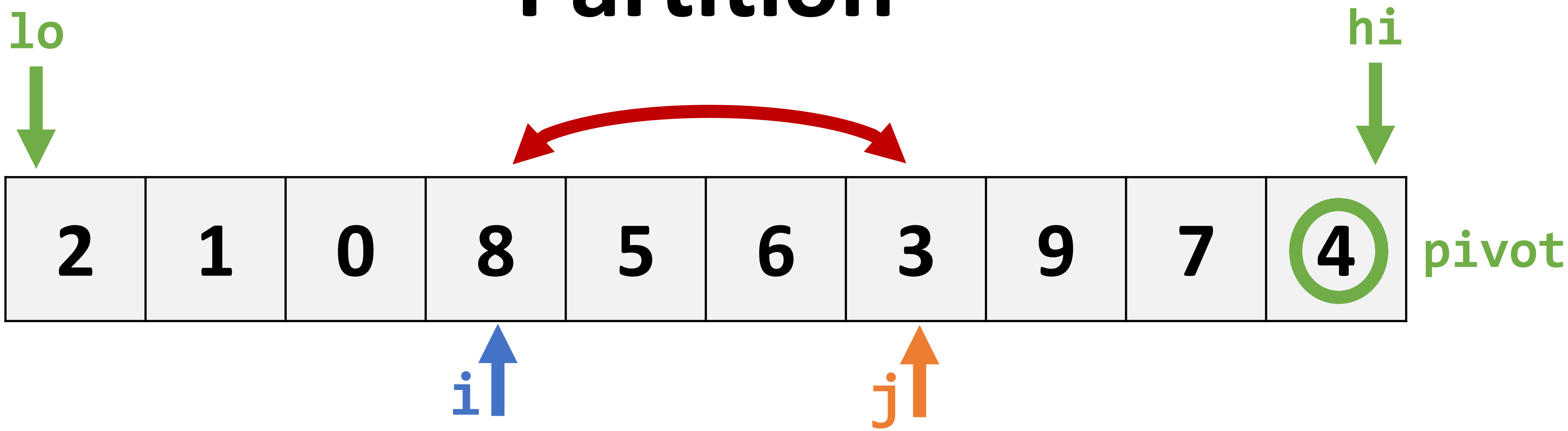

If `A[j] < pivot`:
- Swap `A[i]` and `A[j]`
- Increment `i`

Always increment `j`

**Stop when `j == hi`**

Lomuto Scheme

# Partition




If A[j] < pivot:

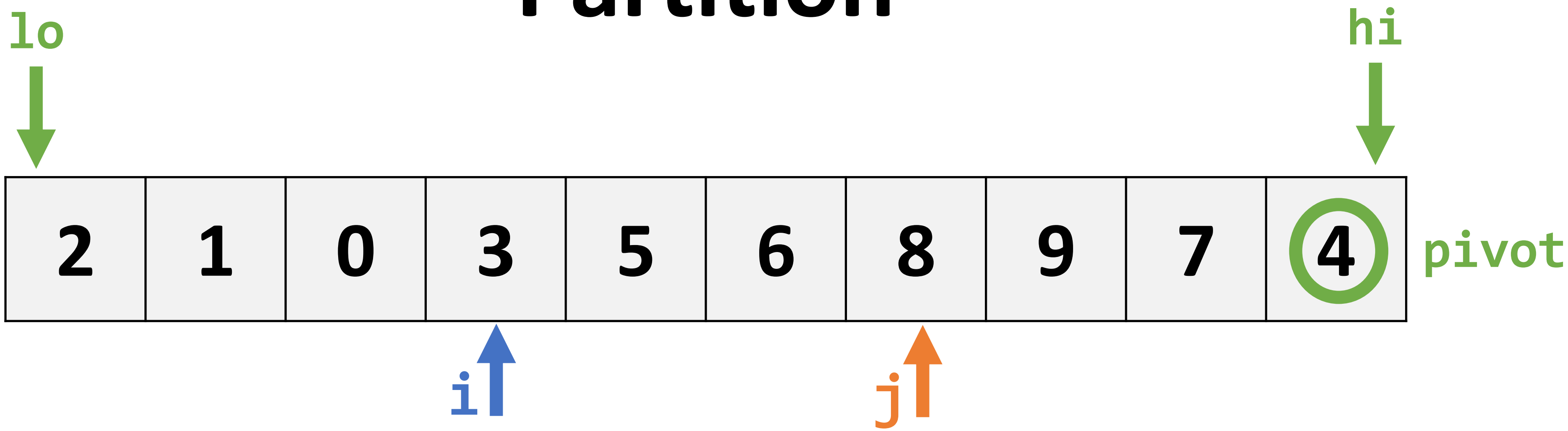
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

Lomuto Scheme

# Partition



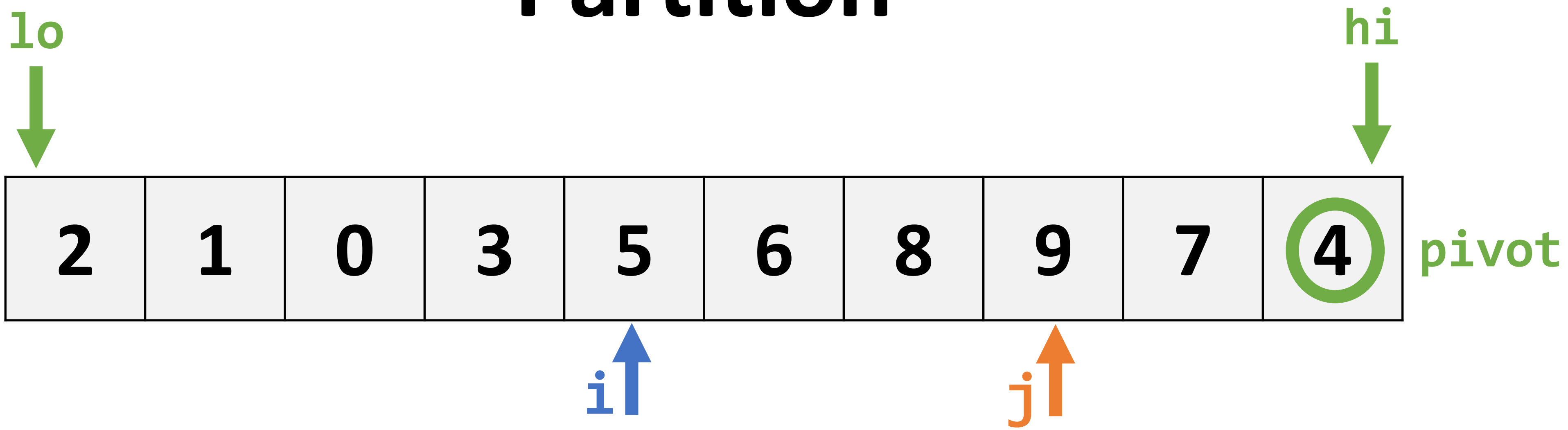

If A[j] < pivot:
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

Lomuto Scheme

# Partition




If A[j] < pivot:

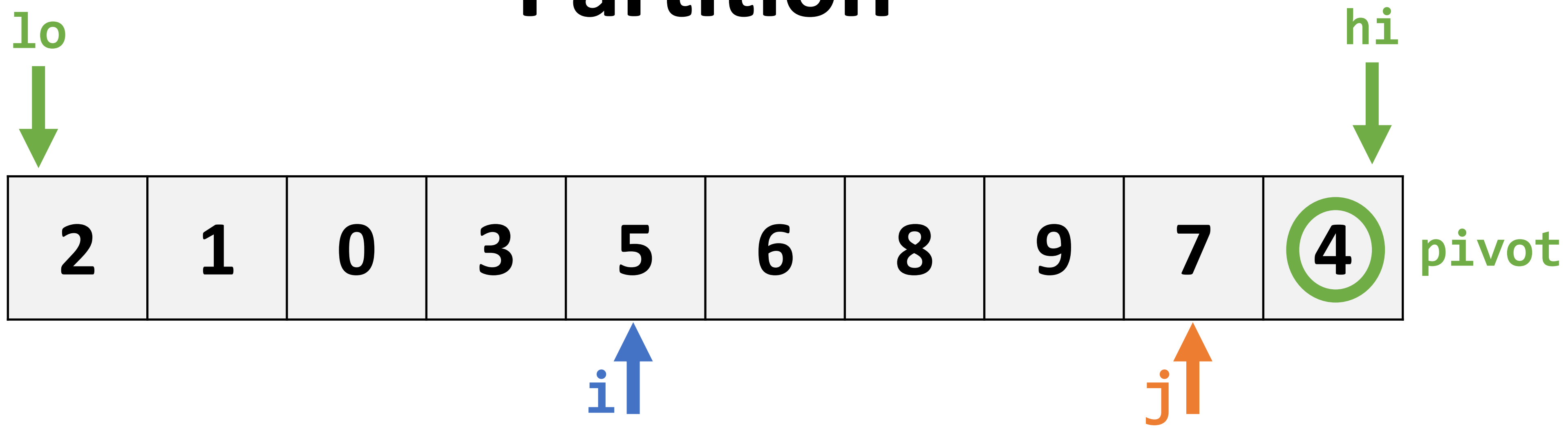
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

Lomuto Scheme

# Partition



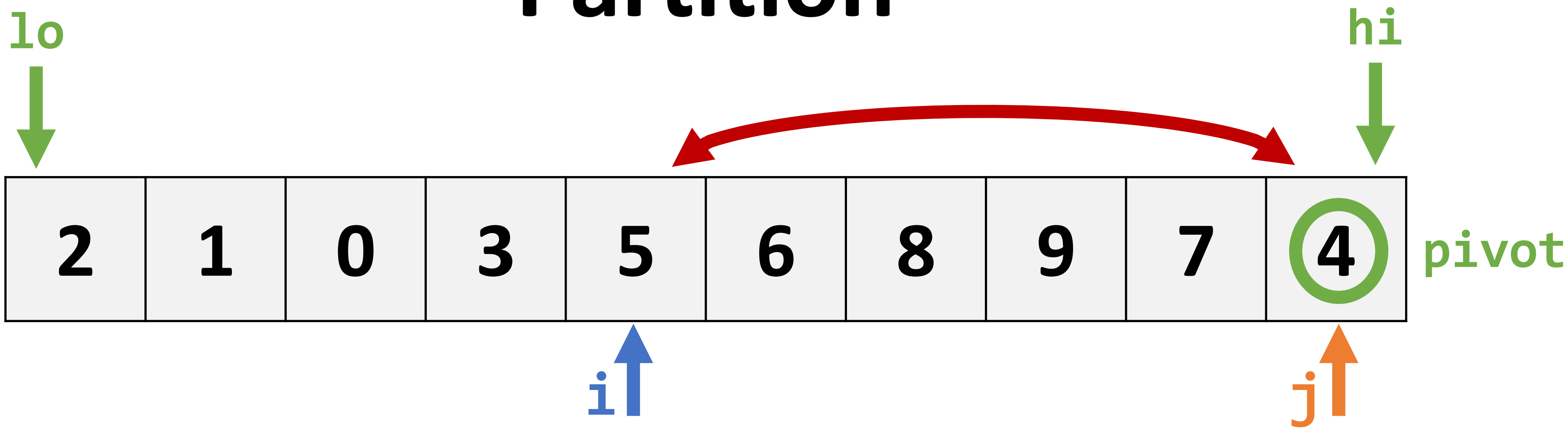

If A[j] < pivot:
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi**

Lomuto Scheme

# Partition




If A[j] < pivot:

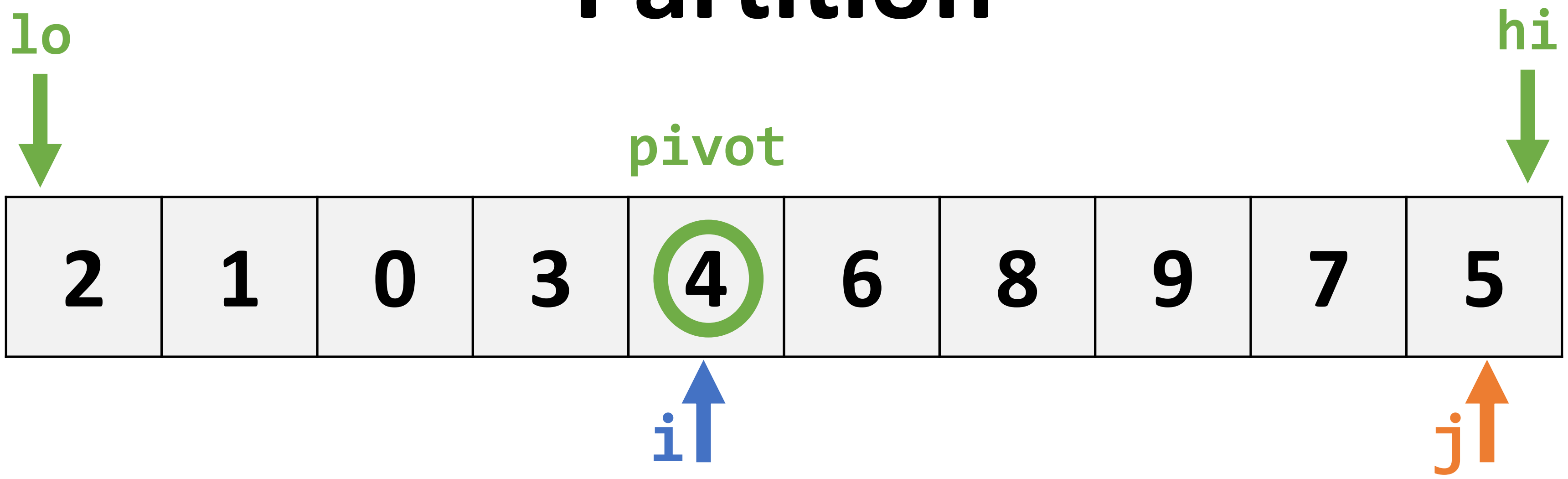
- Swap A[i] and A[j]
- Increment i

Always increment j

**Stop when j == hi** → Swap **A[i]** and **A[hi]**

Lomuto Scheme

# Partition



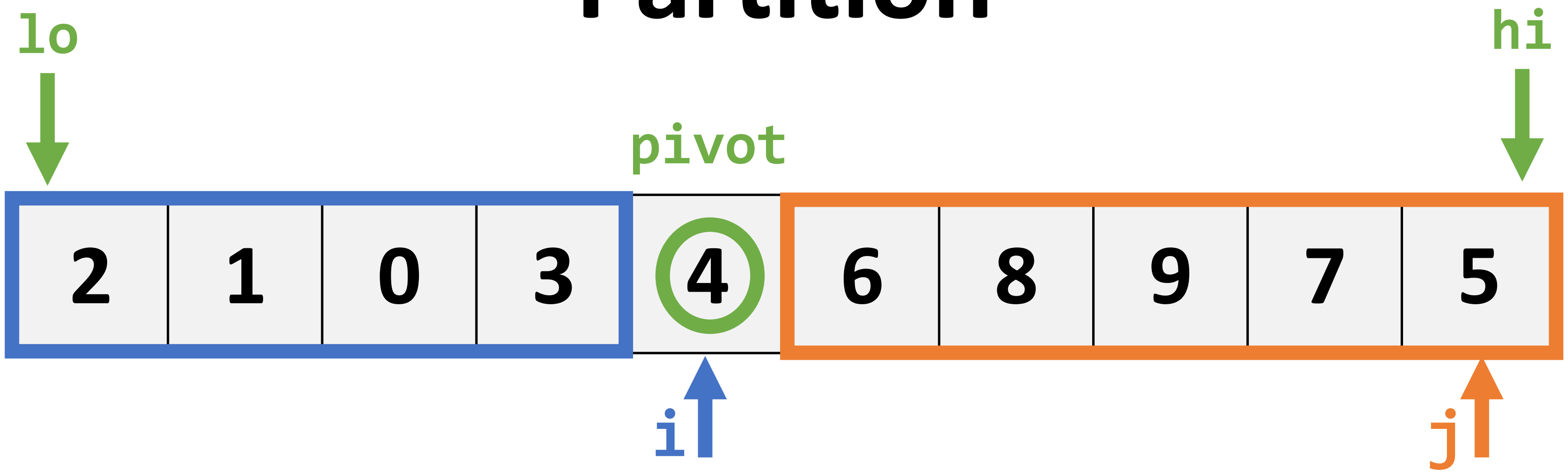

If `A[j] < pivot`:
- Swap `A[i]` and `A[j]`
- Increment `i`

Always increment `j`

**Stop when `j == hi`** → Swap **`A[i]`** and **`A[hi]`**

Lomuto Scheme

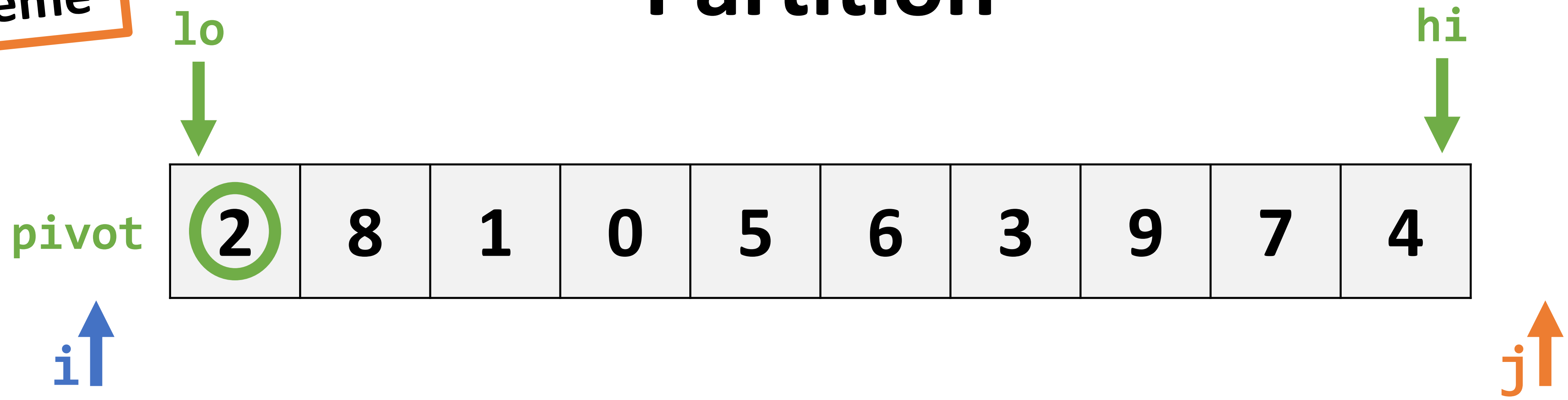
# Partition




- At this point, pivot is guaranteed to be in the correct spot.
- Partition operation is O(n).
- Repeat for sub-arrays **`lo`** to **`i-1`**, and **`i+1`** to **`hi`**.

Hoare Scheme

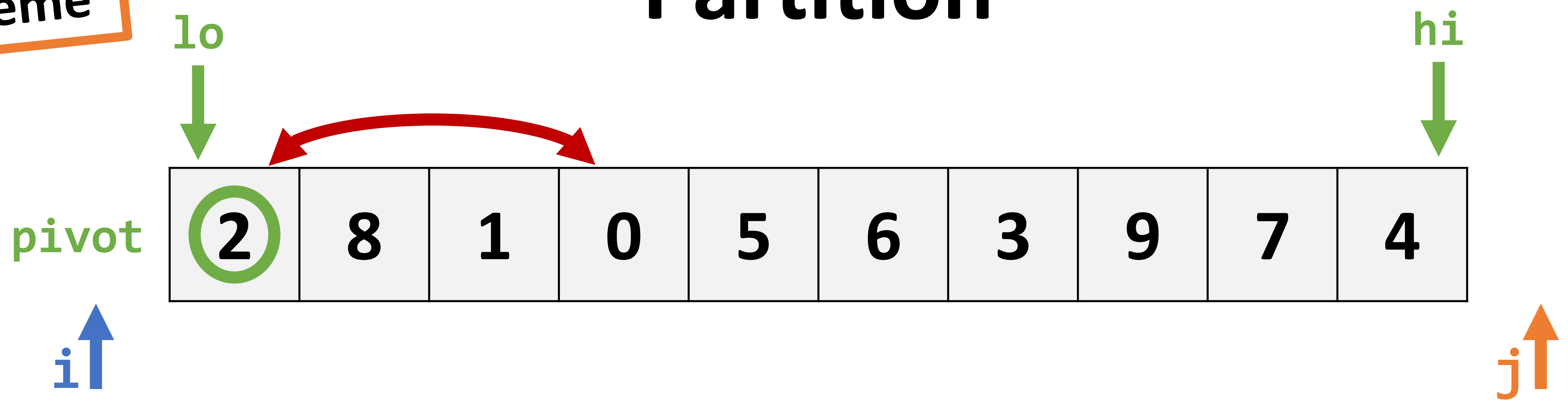
# Partition




- Select `lo` element as pivot
- This is Hoare's original proposal.
- Maintain two indexes, **`i`** and **`j`**.
- **`i`** = **`lo-1`**, **`j`** = **`hi+1`**

Hoare Scheme

# Partition

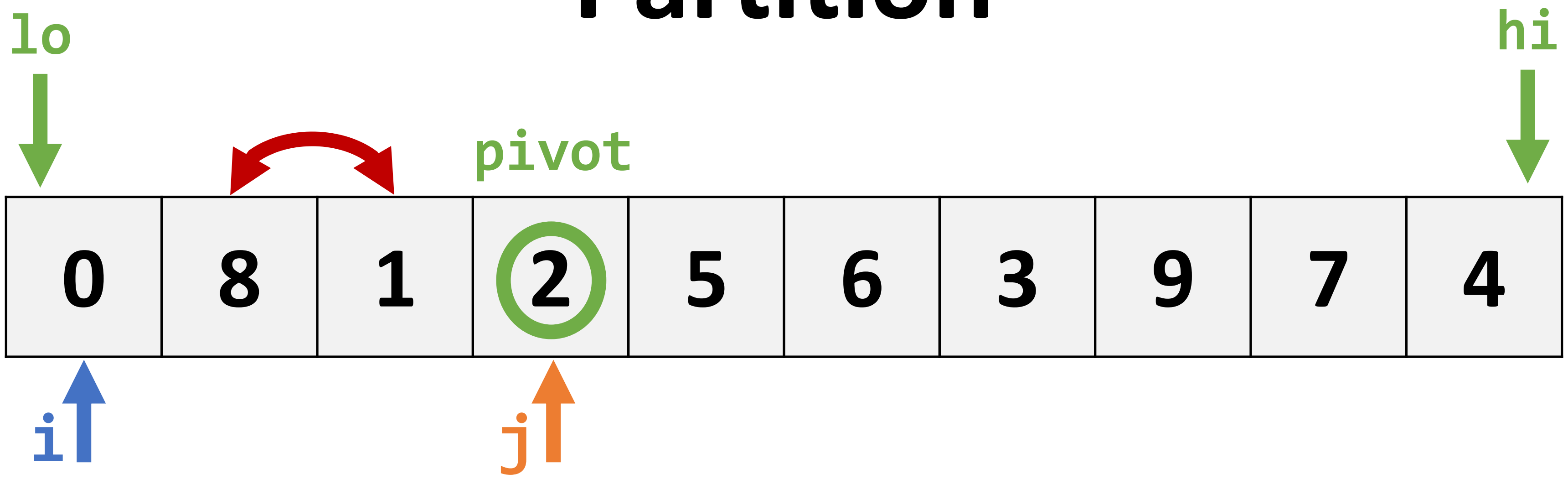



- **`i`** moves inwards until it finds an element >= pivot.
- **`j`** moves inwards until it finds an element <= pivot.
- Return **`j`** if **`i >= j`**
- Swap elements at **`i`** and **`j`**.

**Note!**
- **`i`** increments *before* comparison.
- **`j`** decrements *before* comparison.

Hoare Scheme

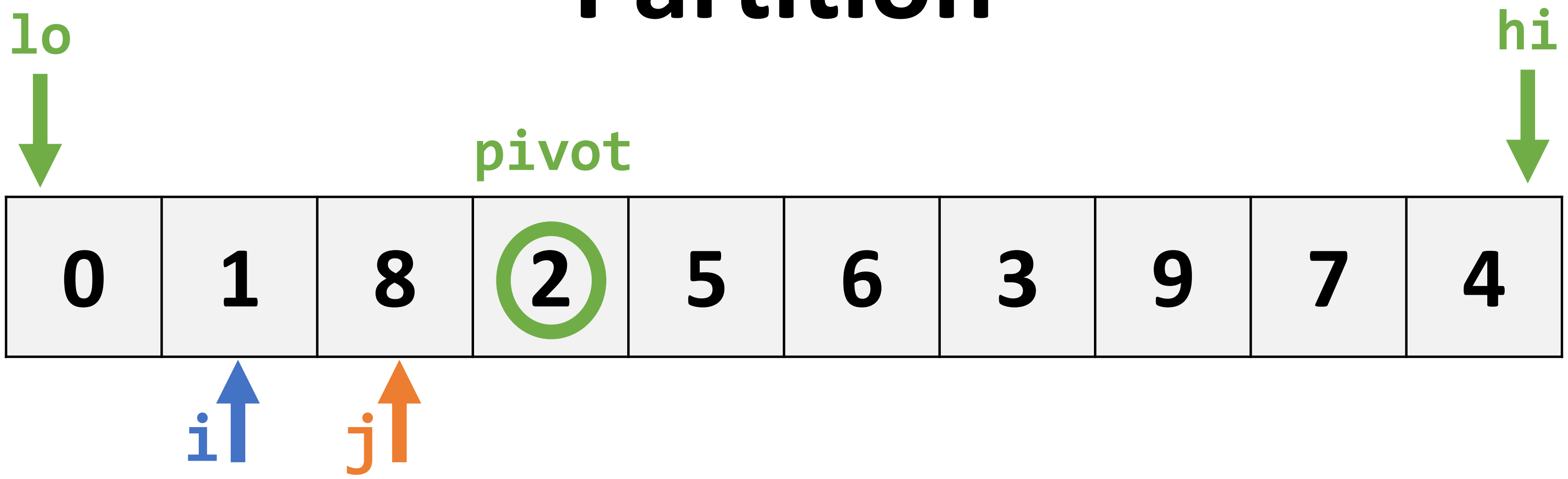
# Partition




- `i` moves inwards until it finds an element >= pivot.
- `j` moves inwards until it finds an element <= pivot.
- Return `j` if `i >= j`
- Swap elements at `i` and `j`.

Hoare Scheme

# Partition




- **`i`** moves inwards until it finds an element >= pivot.
- **`j`** moves inwards until it finds an element <= pivot.
- Return **`j`** if **`i >= j`**
- Swap elements at **`i`** and **`j`**.

Hoare Scheme

# Partition


- **i** moves inwards until it finds an element >= pivot.
- **j** moves inwards until it finds an element <= pivot.
- Return **j** if **i >= j**
- Swap elements at **i** and **j**.

**Notice!**

- Pivot is *not necessarily* in the right place.
- We repeat for subarrays **lo** to **j**, and **j+1** to **hi**

**Hoare Scheme**

# Partition

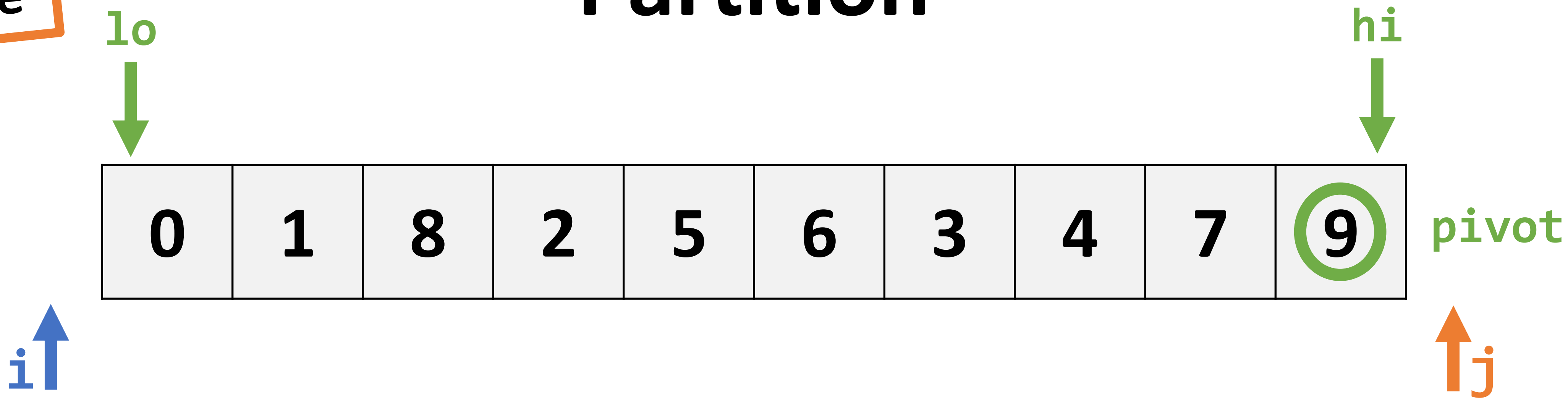

**Notice!**

- Pivot is *not necessarily* in the right place.
- We repeat for subarrays **`lo`** to **`j`**, and **`j+1`** to **`hi`**

- This is why we choose pivot to be the first element.
- This ensures that each side ends up with at least one element.

Hoare Scheme

# Partition

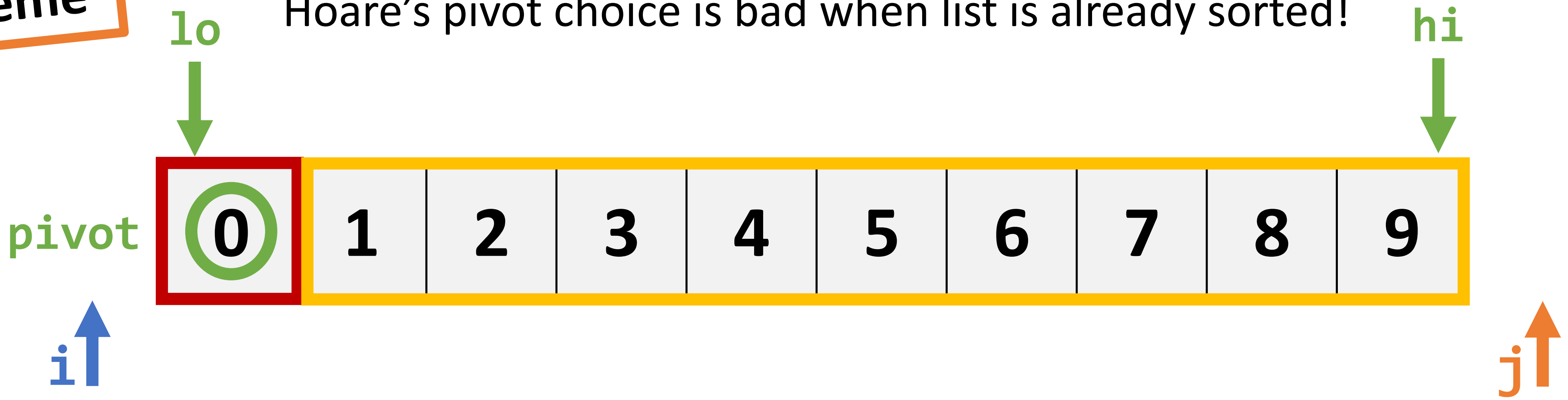



- **i** moves inwards until it finds an element >= pivot.
- **j** moves inwards until it finds an element <= pivot.
- Return **j** if **i >= j**
- Repeat for **lo** to **j**, and **j+1** to **hi** → **Empty!**

If we choose **hi** as pivot, and **hi** is largest element, right half will have zero elements

Hoare Scheme

Hoare's pivot choice is bad when list is already sorted!




- `i` moves inwards until it finds an element >= pivot.
- `j` moves inwards until it finds an element <= pivot.
- Return `j` if `i >= j`

- Partitioning is O(n)
- Optimally, partition cuts array in half each time.
- If the array is sorted, we get two subarrays of size 1 and n-1
- Thus, We partition O(n) times.
- **$O(n^2)$**

# **Partition:** Hoare VS Lomuto

- Hoare's scheme performs 3x fewer comparisons on average
- Lomuto's promises pivot is in the correct location
  - *Very valuable in other algorithms based on partitioning*
- Both are bad when the list is already sorted – $O(n^2)$
  - This has to do with choice of pivot.
  - Can we do better on that front?
  - **Recall:** Lomuto takes **`hi`**, Hoare takes **`lo`**

# Pivot Choice

- *Optimal* choice is the true median. This splits the list in half each partition.
- However, finding true median cannot be done in **O(n)**

**Sedgewick suggests the “Median of 3” rule (Hoare partitioning):**

- Pivot = median of first, middle, and last elements.
- This counters cases where the list is sorted, or reverse sorted.
- It is a reasonable estimate of the true median.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 8 | 2 | 5 | 6 | 3 | 9 | 7 | 4 |

# Pivot Choice

**Another common choice is a random pivot:**

- This combats $O(n^2)$ on sorted lists, but of course we might still randomly select the first or last element as a pivot.
- In practice, random and median-of-3 both outperform `hi` or `lo`.
- Median-of-3 does edge out random:

**Experimental:**

- Random pivot makes 1.386nlogn comparisons
- M-of-3 pivot makes 1.188nlogn comparisons
- Not **quite** this simple – nothing ever is.

# Pivot Choice

**<u>Not *quite* this simple – nothing ever is.</u>**

- Because M-of-3 is deterministic, it’s possible to contrive input arrays that “break” this approach to yield worst-case performance.
- Random pivots might be marginally less efficient, but there is no *predictable* worst-case data set.

**Choosing a good pivot has implications in other algorithms as well!**

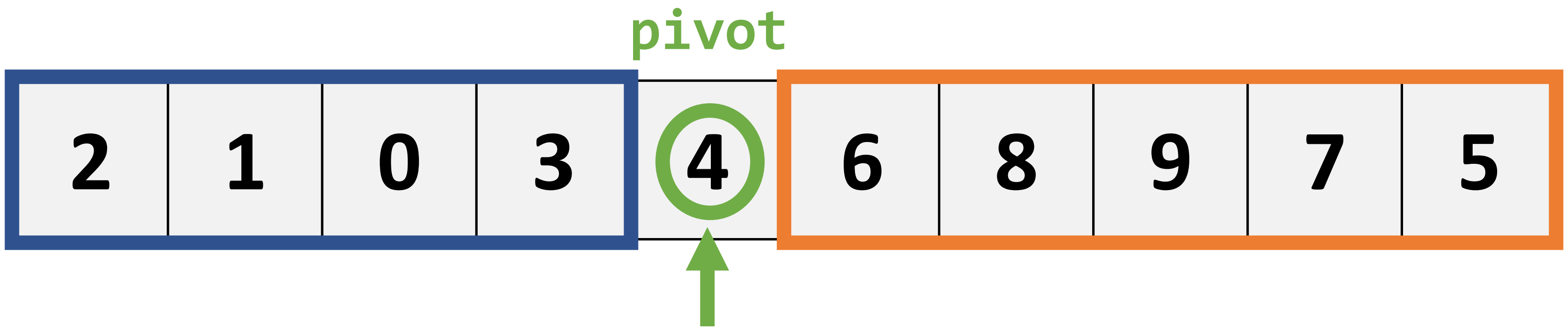
***For example…***

# Quickselect

- Algorithm for finding **k^th^** smallest element in an *unsorted* list.
- **Recall:** Finding smallest is easy, O(n)
  - Can we do better than O(nk) for the **k^th^**?
- Use the same overall approach as Quicksort
- Specifically, Lomuto's partition scheme.

Lomuto Scheme

# Find k$^{th}$ smallest element:




- After each partition, we know that the pivot is in the correct position!
- If the pivot is in position **x**, and:
  - **k < x** look in left partition, throw away the rest
  - **k > x** look in right partition, throw away the rest
  - **k == x** return x

**Lomuto Scheme**

# Find $\mathbf{k}^{th}$ smallest element:

pivot

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **2** | **1** | **0** | **3** | **4** | **6** | **8** | **9** | **7** | **5** |

- This is Quicksort, but with ***one*** recursive call instead of ***two***.
- Like Quicksort, pivot choice is critical. Recall what we said previously about choosing a pivot (M-of-3 vs random)
- Quickselect is O(n)-*expected* (assumes decent pivot).

# **Quickselect:** O(n)

Assume we split the list in half each partition:

$$T = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \dots$$

1st partition 2nd partition 3rd partition 4th partition

**And so on…**

# **Quickselect:** O(n)

Assume we split the list in half each partition:

$$T = \boxed{n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \dots}$$

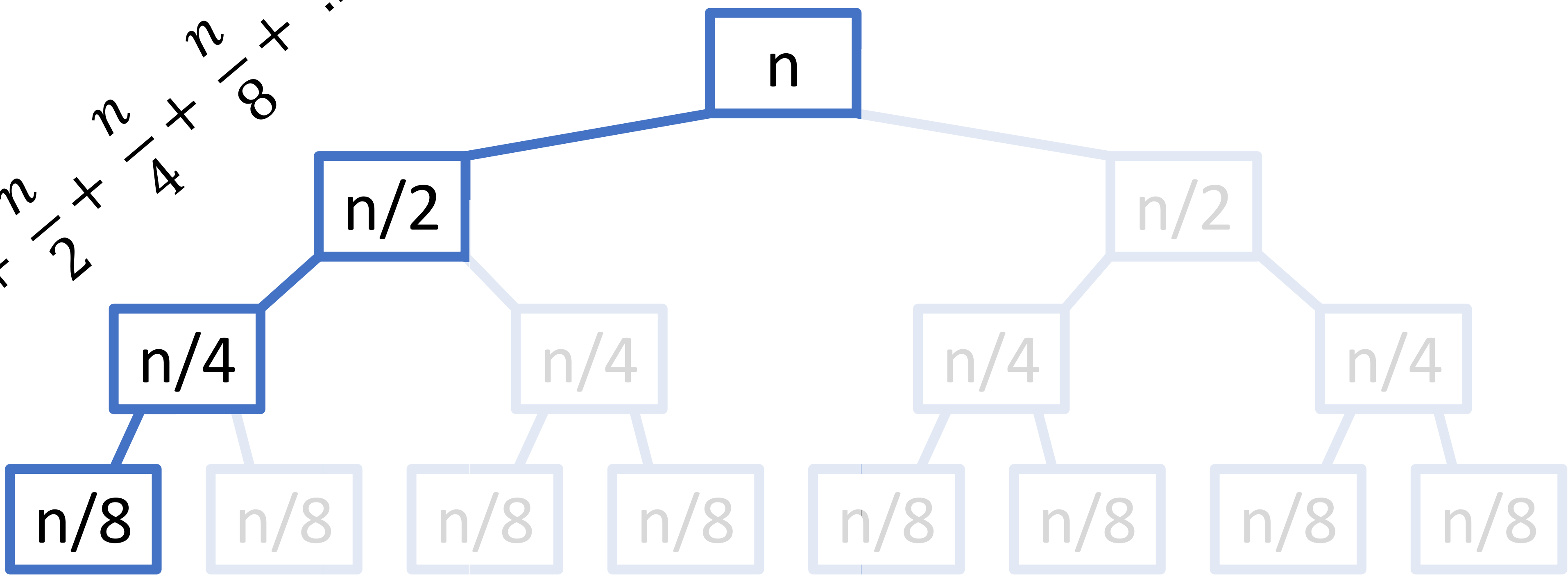
Geometric series.

Sums to $2n$

$= \ 2n \ = \ O(n)$

Like how we demonstrated Mergesort was `O(nlogn)`:

$$T = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \dots$$




At each partition, we only go down a single branch

# Stability

Quicksort is faster than Mergesort on average but is not stable.

**Primitives?** Use Quicksort:
- Faster, but unstable
- Doesn’t matter with primitives!
- A 7 is a 7 is a 7.
- No secondary characteristics

**Objects?** Use Mergesort:
- Stability more important.
- Might be multiple ways to order objects
- We may want to retain ordering based on secondary characteristics.

# **Quicksort:** In LISP?

```lisp
(defun quicksort (vec comp)
  (when (> (length vec) 1)
    (let ((ppvt 0) (pivot (aref vec (1- (length vec))))) ; last element
    (dotimes (i (1- (length vec))) ; finds position of the pivot
      (when (funcall comp (aref vec i) pivot)
        (rotatef (aref vec i) (aref vec ppvt))
        (incf ppvt)
      )
    )
    ;; swap the pivot (last element) to its proper place
    (rotatef (aref vec (1- (length vec))) (aref vec ppvt))
    (quicksort (rtl:slice vec 0 ppvt) comp)          ; sort left sublist
    (quicksort (rtl:slice vec (1+ ppvt)) comp)))     ; sort right sublist
  )
  vec
)
```

```lisp
(defun quicksort (vec comp)
    (when (> (length vec) 1)
    (let ((ppvt 0) (pivot (aref vec (1- (length vec))))) ; last element
    (dotimes (i (1- (length vec))) ; finds position of the pivot
        (when (funcall comp (aref vec i) pivot)
            (rotatef (aref vec i) (aref vec ppvt))
            (incf ppvt)
        )
    )
    ;; swap the pivot (last element) to its proper place
    (rotatef (aref vec (1- (length vec))) (aref vec ppvt))
    (quicksort (rtl:slice vec 0 ppvt) comp)          ; sort left sublist
    (quicksort (rtl:slice vec (1+ ppvt)) comp)))  ; sort right sublist
    vec
)
```

```
* (quicksort #(2 4 6 5 8 9 0 1 3 5) '<)
#(0 1 2 3 4 5 5 6 8 9)
* (quicksort #(2 4 6 5 8 9 0 1 3 5) '>)
#(9 8 6 5 5 4 3 2 1 0)
```

# In Summary

**Advanced Sorting**

- Insertion sort and its optimizations
- Sorting in O(nlogn) with Mergesort
- Quicksort and pivot choice