

CS 113 – Computer Science I

Lecture 24 – Binary Search

Tuesday 12/05/2023

Announcements

HW09 – Big OOP – due Wednesday 12/06

HW10 – due Thursday 12/14

Faculty Interview/Mock Lecture

- Friday 12/08 – 11-11:30am
- Binary Search Tree
- Location: TBD
- Tea & Snacks

Outline

- Binary Search
- Sorting
 - Bubble Sort
 - Selection Sort

Searching

Finding whether an item is in a collection

Applications:

- Specific email in an inbox
- Word in a document
- Course in a list of course offerings
- Professor is on RateMyProfessor
- ...

Common search problems

Is an item in an array?

- Returns: True or False

Where in an array is the item?

- Returns: the index (an integer)
 - Standard: -1 if the item is not found

How many times does the item appear in an array?

- Returns: a count (an integer)

What is the min, max, or average value in an array?

- Returns: the value (double)

Linear Search

Check each item in a collection one by one

Why is this call linear search?

Time it takes to search increases *linearly* with the size of the list

Linear Search

What happens (in terms of speed) when the list is very large?

The search becomes slower

In what cases do we do the most work (i.e. perform the most comparisons)?

When the item is not in the list

In what cases do we do the least amount of work?

When the item is the first element in the list

Binary Search

If we could change the list, is there a way to search more efficiently?

Yes, if the list is sorted

Guessing game – in class exercise

Pair up:

- Person A chooses a number between 1 and 100
- Person B guesses the number
- Until the guess is correct:
 - Person A tells whether the guess is too high or too low
 - Person B guesses again

Binary Search

Assuming list is sorted in ascending order

High-level Algorithm:

- Step 1: Find the midpoint of the list:
 - if the search value is at the midpoint – we are done!
 - if the value we are searching for is above the midpoint,
 - Search right: cut our list in half and repeat step 1 with the right half of the list
 - If the value we are searching for is below the midpoint
 - Search left: cut out list in half and repeat step 1 with the left half of the list

Binary Search – Initial Values

lowIndex, highIndex, midIndex

lowIndex = 0

highIndex = length of the array – 1

midIndex = $\frac{lowIndex + highIndex}{2}$

Binary Search – Initial Values

`lowIndex, highIndex, midIndex`

If value at `midIndex` == `searchValue`:

Success!

If value at `midIndex` < `searchValue`:

`lowIndex = midIndex + 1`

update `midIndex`

If value at `midIndex` > `searchValue`:

`highIndex = midIndex - 1`

update `midIndex`

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 99

low	mid	high	ls[mid]

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 99

low	mid	high	ls[mid]
0	2	5	44

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 99

low	mid	high	ls[mid]
0	2	5	44
3	4	5	99 (found!)

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 30

low	mid	high	ls[mid]

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 30

low	mid	high	ls[mid]
0	2	5	44

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 30

low	mid	high	ls[mid]
0	2	5	44
0	0	1	-20

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 30

low	mid	high	ls[mid]
0	2	5	44
0	0	1	-20
1	1	1	-4

Binary search

String[] ls = {⁰-20, ¹-4, ²44, ³58, ⁴99, ⁵145}

Search for 30

low	mid	high	ls[mid]
0	2	5	44
0	0	1	-20
1	1	1	-4
2		1	Not found!

Binary Search – Initial Values

`lowIndex, highIndex, midIndex`

If value at `midIndex` == `searchValue`:

Success!

If value at `midIndex` < `searchValue`:

`lowIndex = midIndex + 1`

update `midIndex`

If value at `midIndex` > `searchValue`:

`highIndex = midIndex - 1`

update `midIndex`

Binary search w/ Strings

0 1 2 3 4 5 6 7

```
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};
```

Search for "cow"

low	mid	high	ls[mid]

Binary search w/ Strings

0 1 2 3 4 5 6 7

```
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};
```

Search for "cow"

low	mid	high	ls[mid]
0	3	7	"cat"

Binary search w/ Strings

0 1 2 3 4 5 6 7

```
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};
```

Search for "cow"

low	mid	high	ls[mid]
0	3	7	"cat"
4	5	7	"dog"

Binary search w/ Strings

0 1 2 3 4 5 6 7

```
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};
```

Search for "cow"

low	mid	high	ls[mid]
0	3	7	"cat"
4	5	7	"dog"
4	4	4	"cow"!

Binary search

0 1 2 3 4 5 6 7
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};

Search for "elephant"

low	mid	high	ls[mid]

Binary search

0 1 2 3 4 5 6 7
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};

Search for "elephant"

low	mid	high	ls[mid]
0	3	7	"cat"

Binary search

0 1 2 3 4 5 6 7
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};

Search for "elephant"

low	mid	high	ls[mid]
0	3	7	"cat"
4	5	7	"dog"

Binary search

0 1 2 3 4 5 6 7
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};

Search for “elephant”

low	mid	high	ls[mid]
0	3	7	“cat”
4	5	7	“dog”
6	6	7	“fish”

Binary search

0 1 2 3 4 5 6 7
String[] ls = {"bear", "bird", "bug", "cat", "cow", "dog", "fish", "lion"};

Search for "elephant"

low	mid	high	ls[mid]
0	3	7	"cat"
4	5	7	"dog"
6	6	7	"fish"
6		6	

Binary Search – Initial Values

`lowIndex, highIndex, midIndex`

If value at `midIndex` == `searchValue`:

Success!

If value at `midIndex` < `searchValue`:

`lowIndex = midIndex + 1`

update `midIndex`

If value at `midIndex` > `searchValue`:

`highIndex = midIndex - 1`

update `midIndex`

Binary Search

If the list is sorted in ascending order, we don't need to consider every element.

Which element should we check?

The middle

If the middle element isn't what we are looking for, what should we do?

Chop the search space in half (this is why it's called **binary** search)

Binary Search run time

As the size of our collection increases, the number of guesses/comparisons increases, but not *linearly*

The time increases by $\log n$ (we use base 2). Why?

Because we cut our search space in half each time

If our collection contains 8 data points, how many comparisons in worst case do we make:

$$\log_2 8 = 3$$

If our collection contains 512 data points, how many comparisons in worst case do we make:

$$\log_2 512 = 9$$

Outline

- Binary Search
- Sorting
 - Bubble Sort
 - Selection Sort

Who do we care about sorting?

Makes searching faster!

Efficient [sorting](#) is important for optimizing the [efficiency](#) of other algorithms (such as [search](#) and [merge](#) algorithms) that require input data to be in sorted lists. Sorting is also often useful for [canonicalizing](#) data and for producing human-readable output.



ChatGPT's response:

Covered in data structures, next semester!



Sorting is a fundamental operation in computer science because it is a common and essential task in many applications. Sorting refers to arranging a collection of elements in a particular order, such as numerical, alphabetical, chronological, or any other custom order. Here are a few reasons why we care about sorting in computer science:



1. Searching: Sorted data is easier to search. Searching an unsorted list requires examining every element in the list, which can be slow and inefficient for large data sets. In contrast, searching a sorted list can be done using efficient algorithms like binary search that require examining only a logarithmic number of elements.
2. Data analysis: Sorting can help in data analysis. Sorting data can help identify patterns and relationships that may not be immediately apparent in an unsorted data set. For example, sorting a list of sales data by date can reveal trends and seasonality.
3. Optimization: Sorting is often a pre-requisite for other algorithms, such as finding the median or calculating the mode of a data set. Sorting can also be used as a building block for other algorithms, such as merge sort or quick sort, which are used in many computational applications.
4. Database management: Sorting is important in managing databases. Databases often contain large amounts of data that need to be sorted in order to perform operations such as searching, filtering, and data analysis.

In summary, sorting is an essential operation in computer science that helps in searching, data analysis, optimization, and database management. It is a basic building block for many algorithms and is crucial for efficient and effective computation.

Sorting

How might we sort the list of numbers below.
Can we come up with an algorithm?

0	1	2	3	4	5
10	4	3	0	11	8

Bubble Sort

Compare two adjacent items, and swap if needed

Repeat until largest item is at the back

Repeat process until done

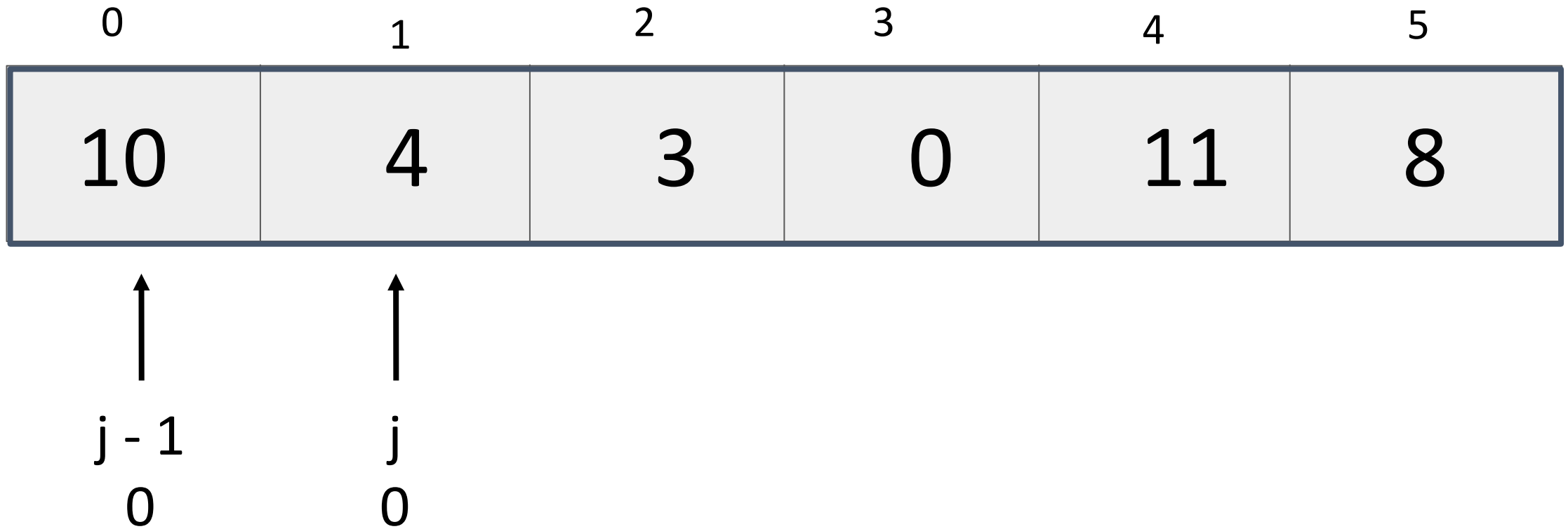
Bubble Sort

0	1	2	3	4	5
10	4	3	0	11	8

What do we do first?

Bubble Sort

len = 6

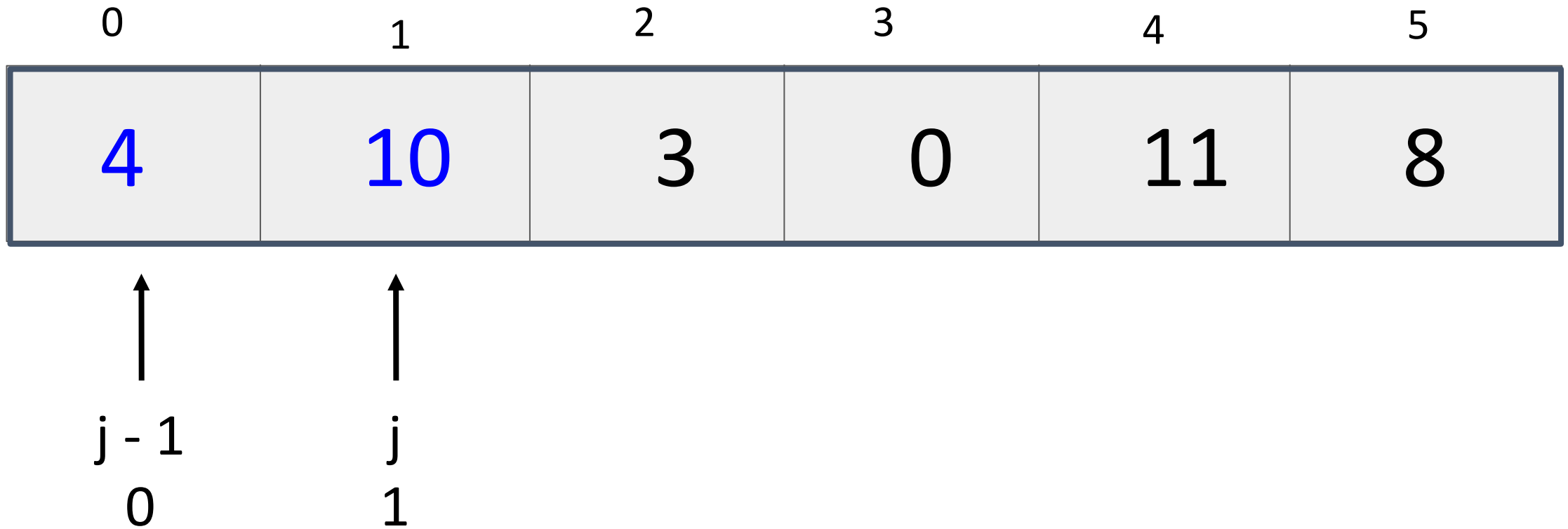


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? ⁴¹

Bubble Sort

len = 6

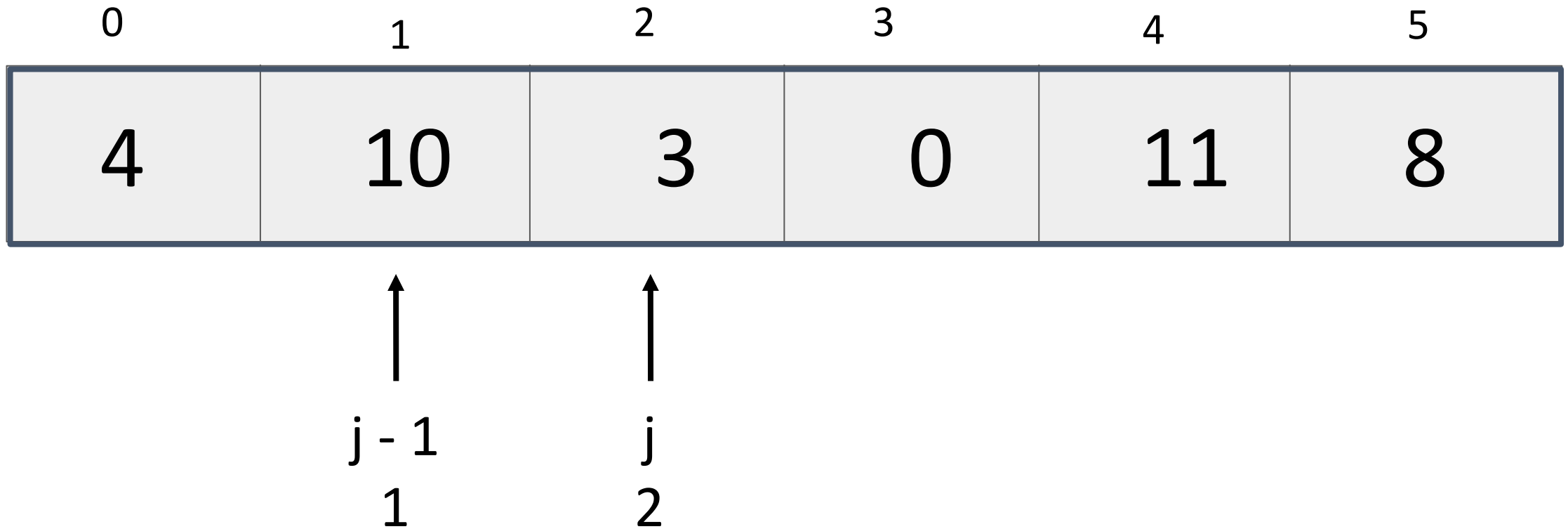


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? ⁴²

Bubble Sort

len = 6

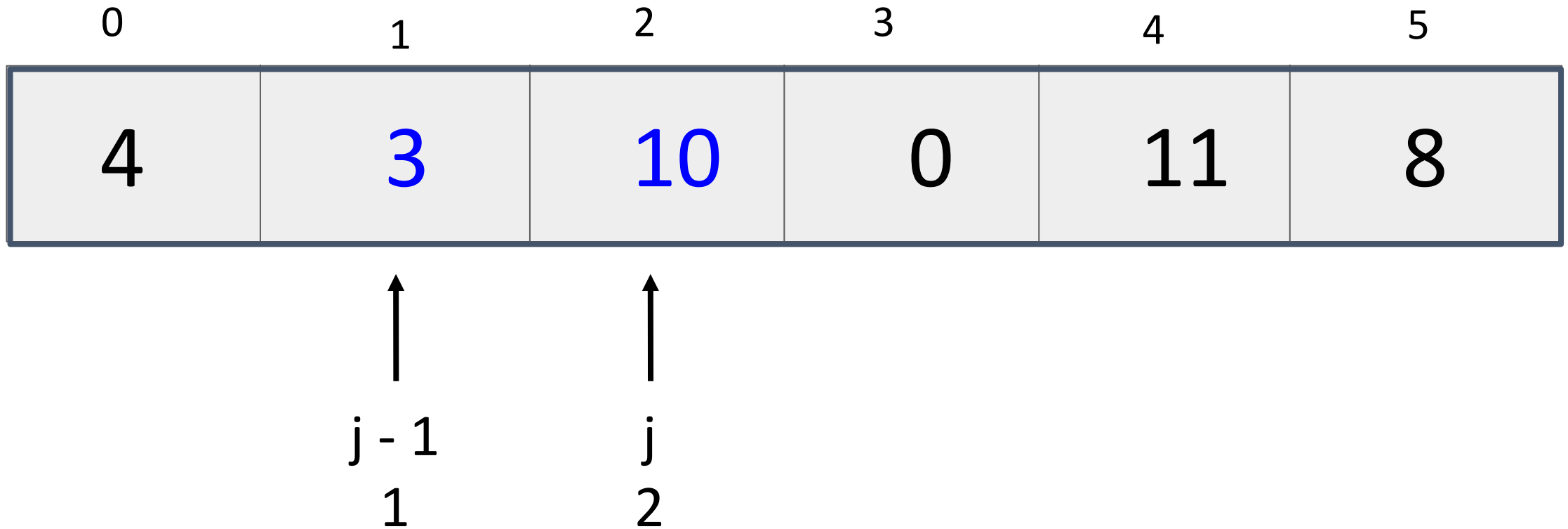


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 43

Bubble Sort

len = 6



Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 44

Bubble Sort

len = 6

0	1	2	3	4	5
4	3	10	0	11	8

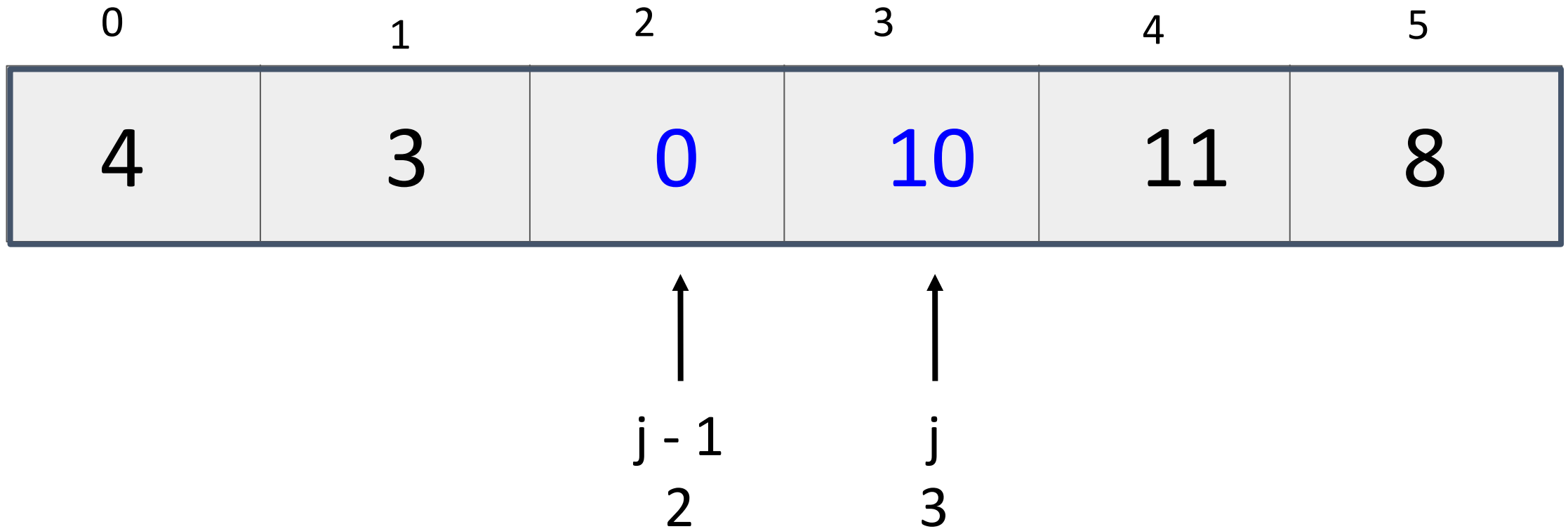
↑ ↑
j - 1 j
2 3

Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 45

Bubble Sort

len = 6

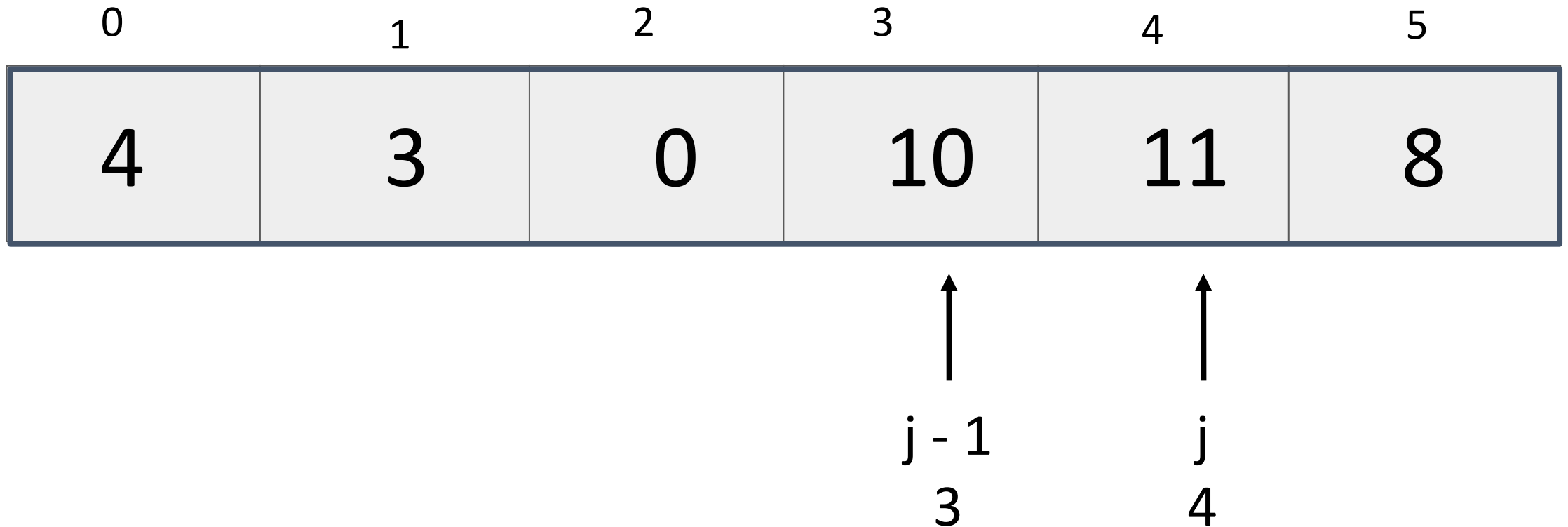


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 46

Bubble Sort

len = 6

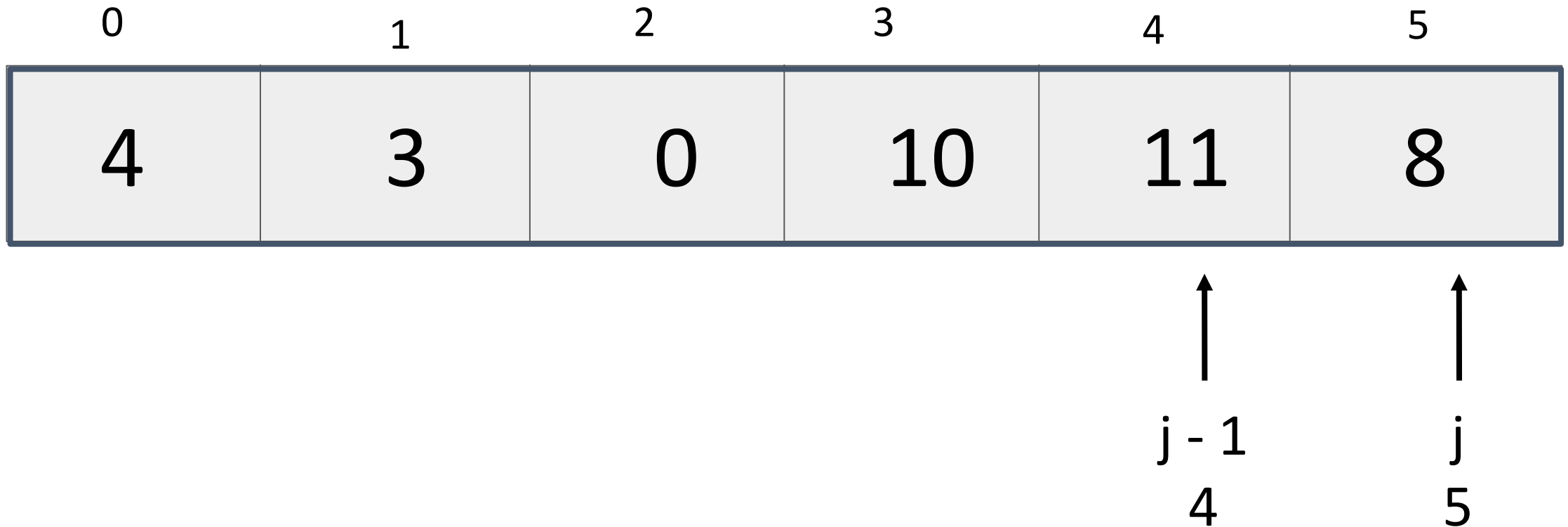


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 47

Bubble Sort

len = 6

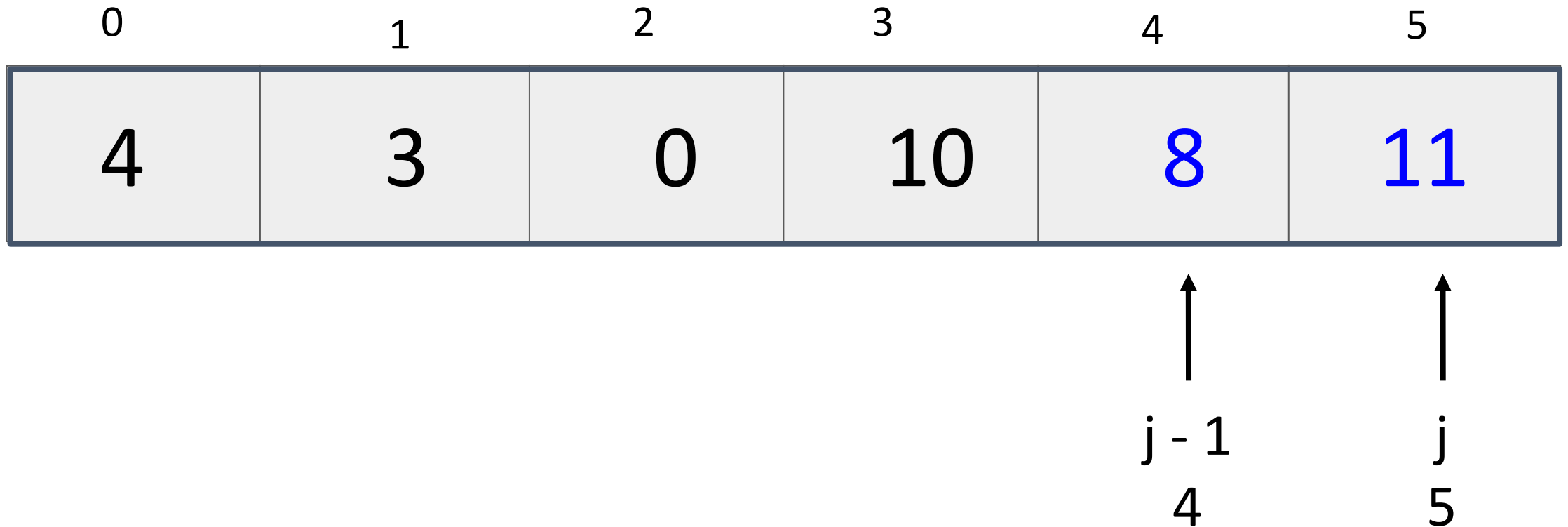


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 48

Bubble Sort

len = 6



Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 49

Bubble Sort

len = 5

0	1	2	3	4	5
4	3	0	10	8	11

↑
 $j - 1$
0

↑
 j
1

Last element has
largest element!

Reset and compare pairs with shorter list!

What next? 50

Bubble Sort

len = 5

0	1	2	3	4	5
3	4	0	10	8	11

↑
j - 1
0

↑
j
1

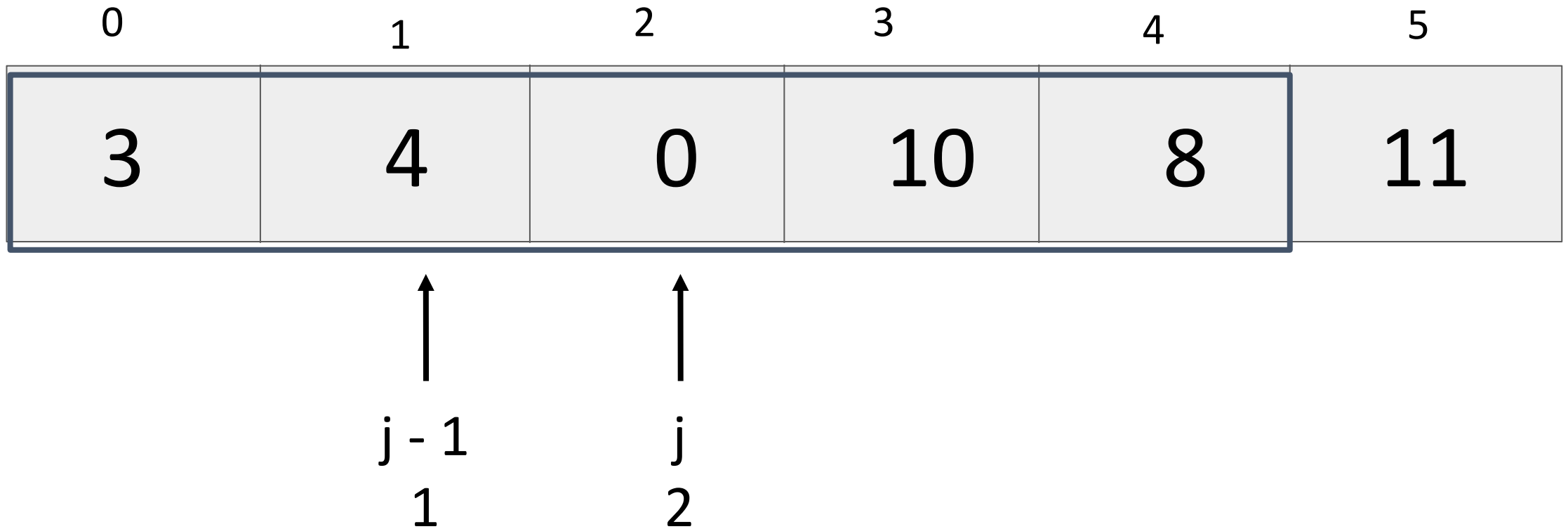
Last element has
largest element!

Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? ⁵¹

Bubble Sort

len = 5

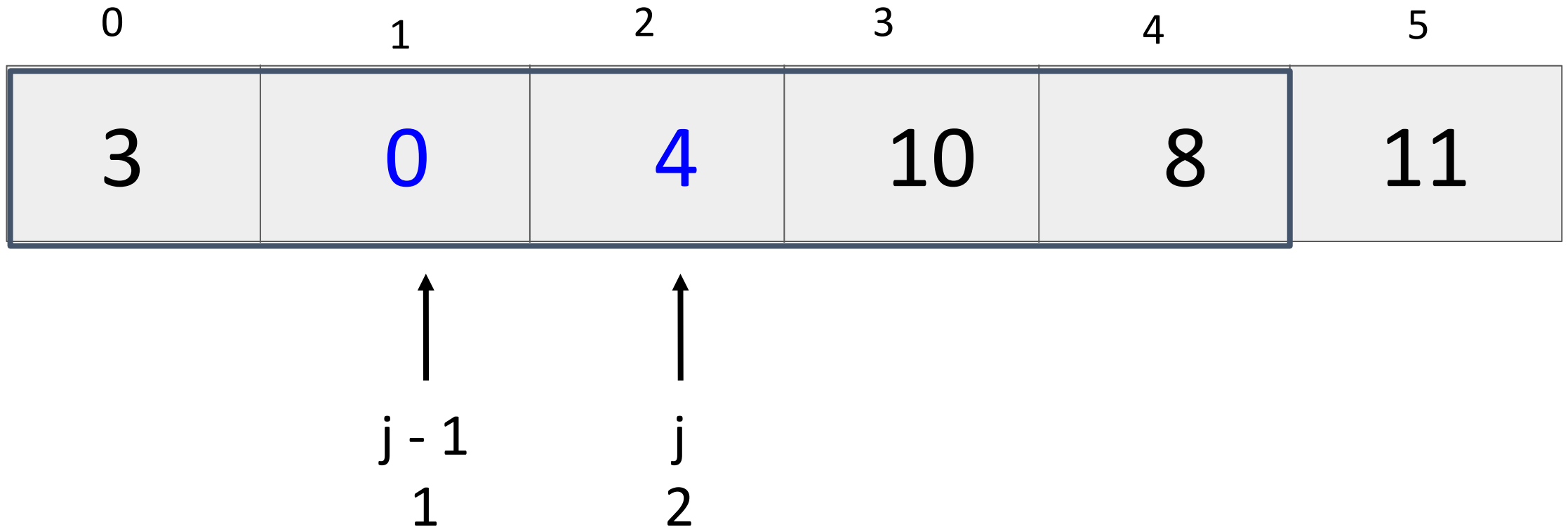


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 52

Bubble Sort

len = 5

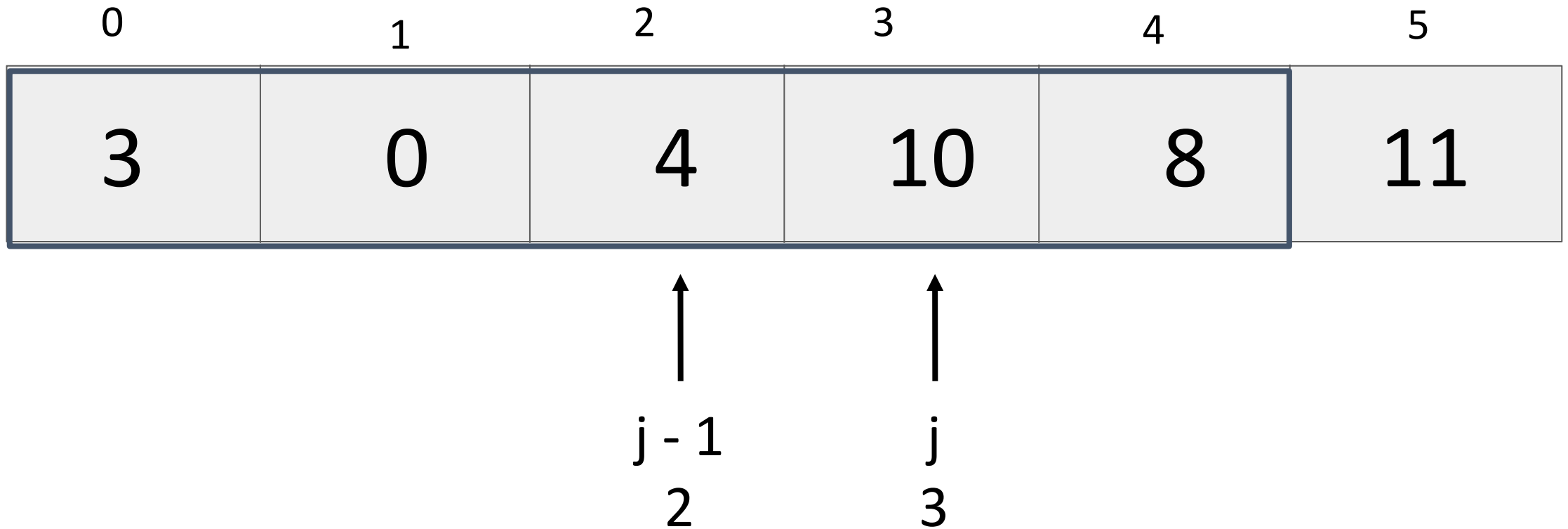


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 53

Bubble Sort

len = 5

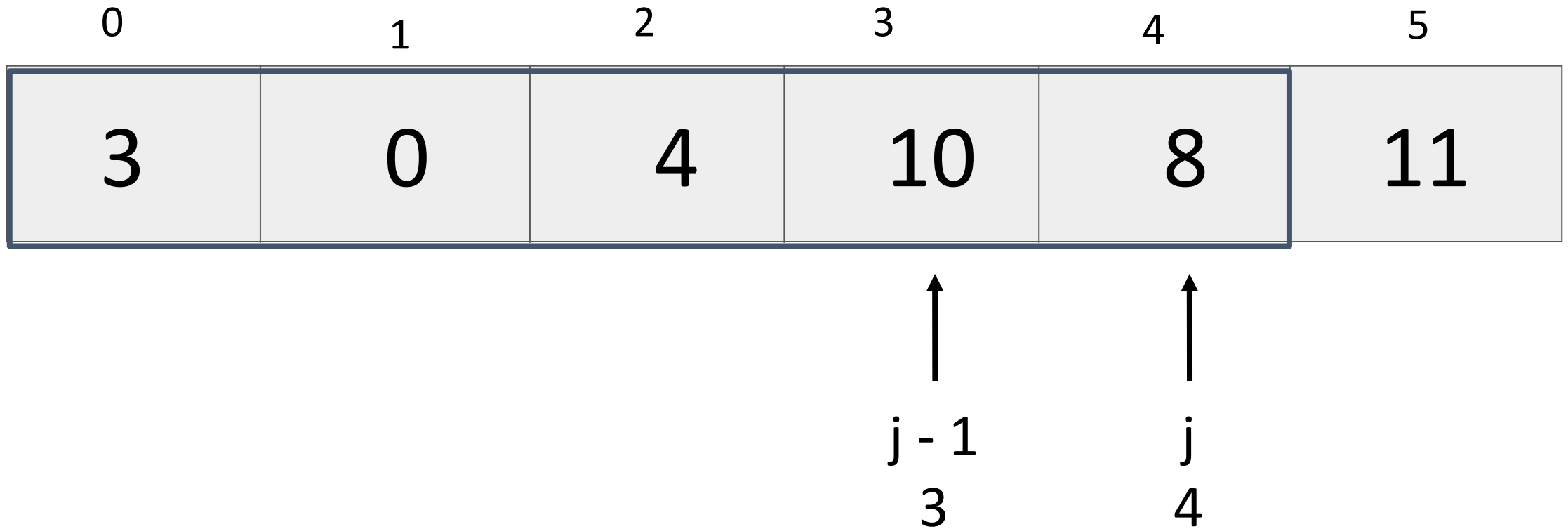


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 54

Bubble Sort

len = 5

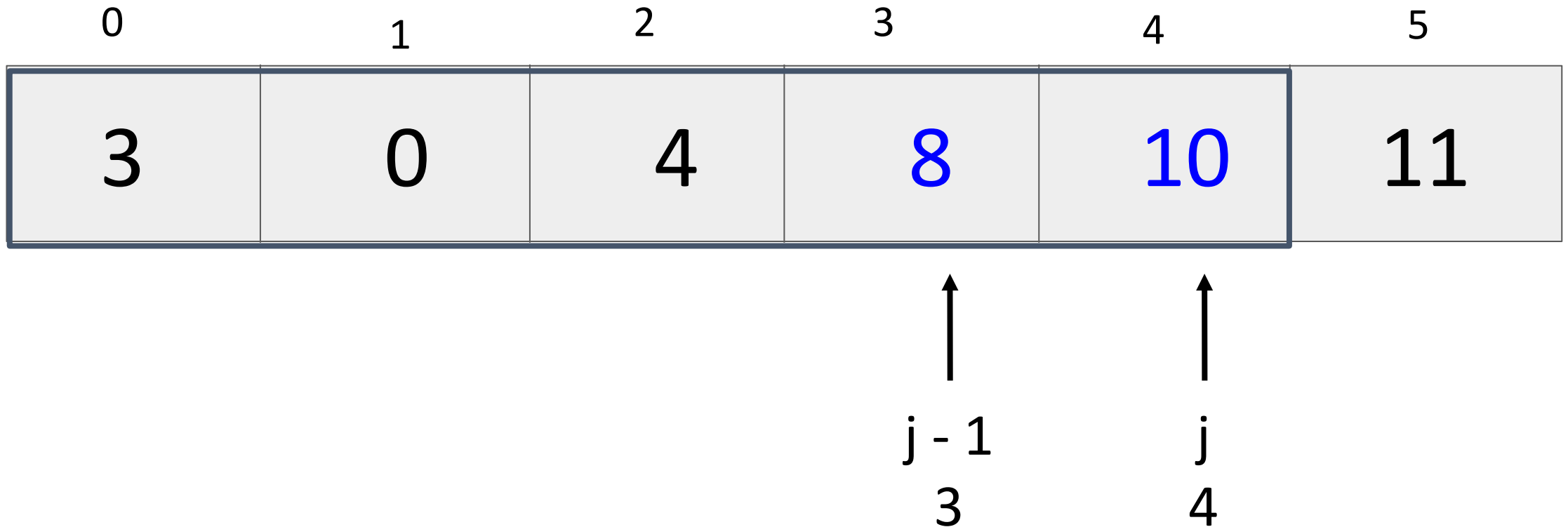


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 55

Bubble Sort

len = 5

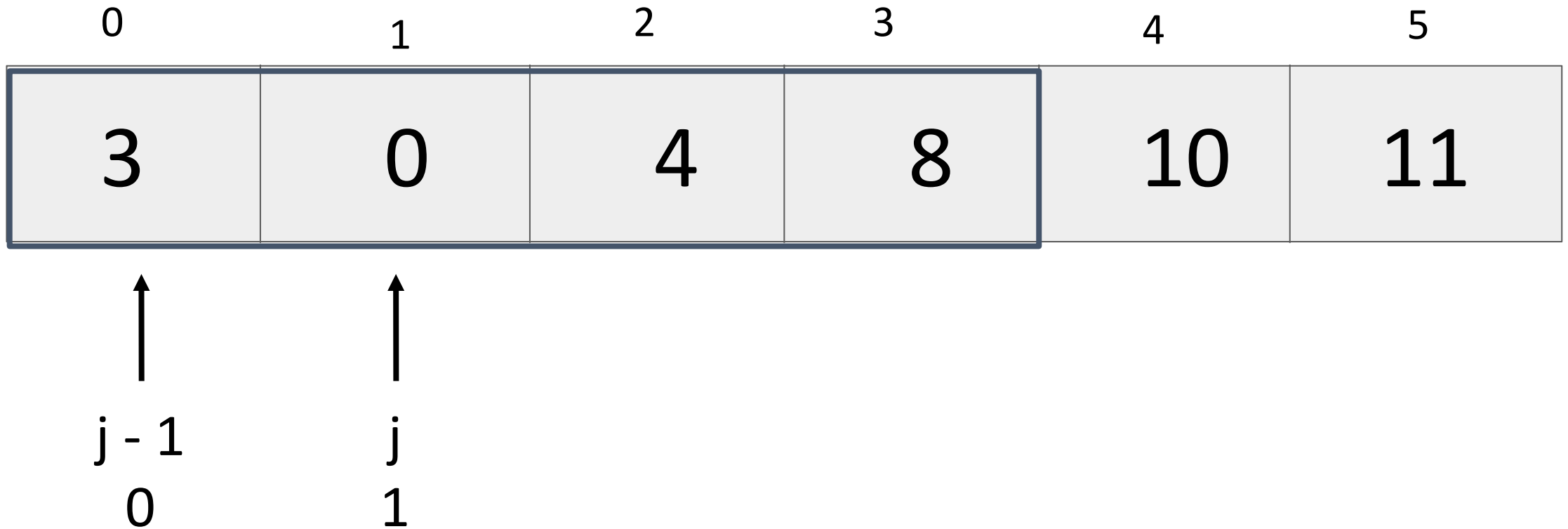


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 56

Bubble Sort

len = 4

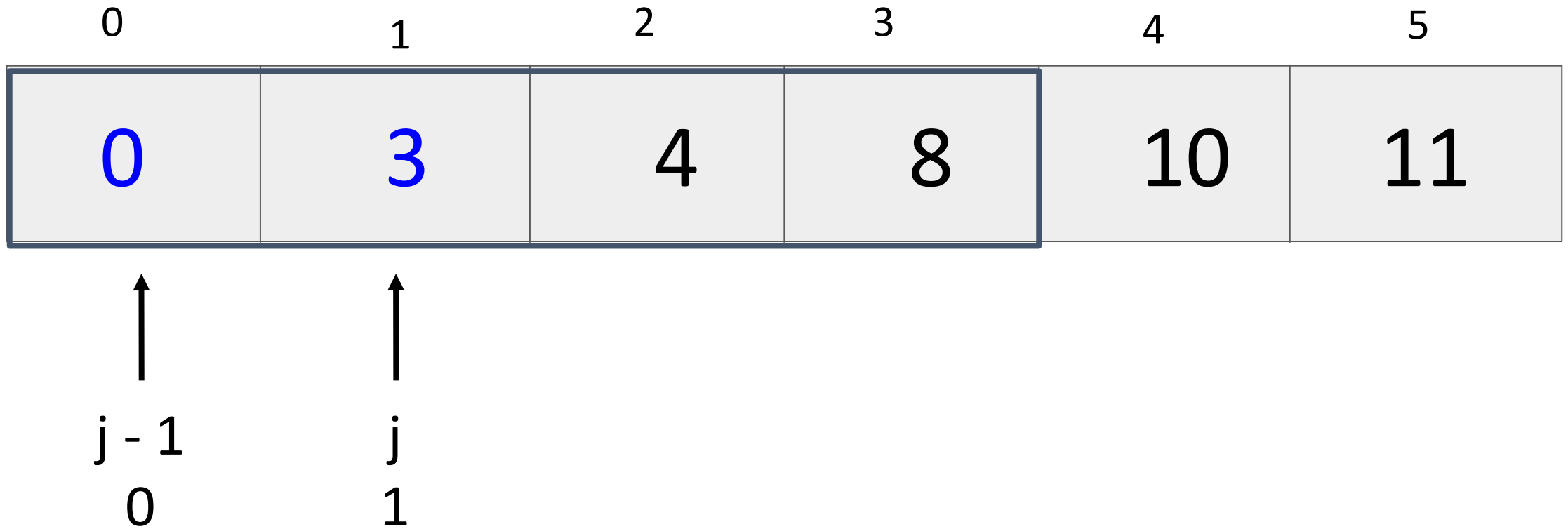


Reset and check pairs with shorter list

What next? ⁵⁷

Bubble Sort

len = 4

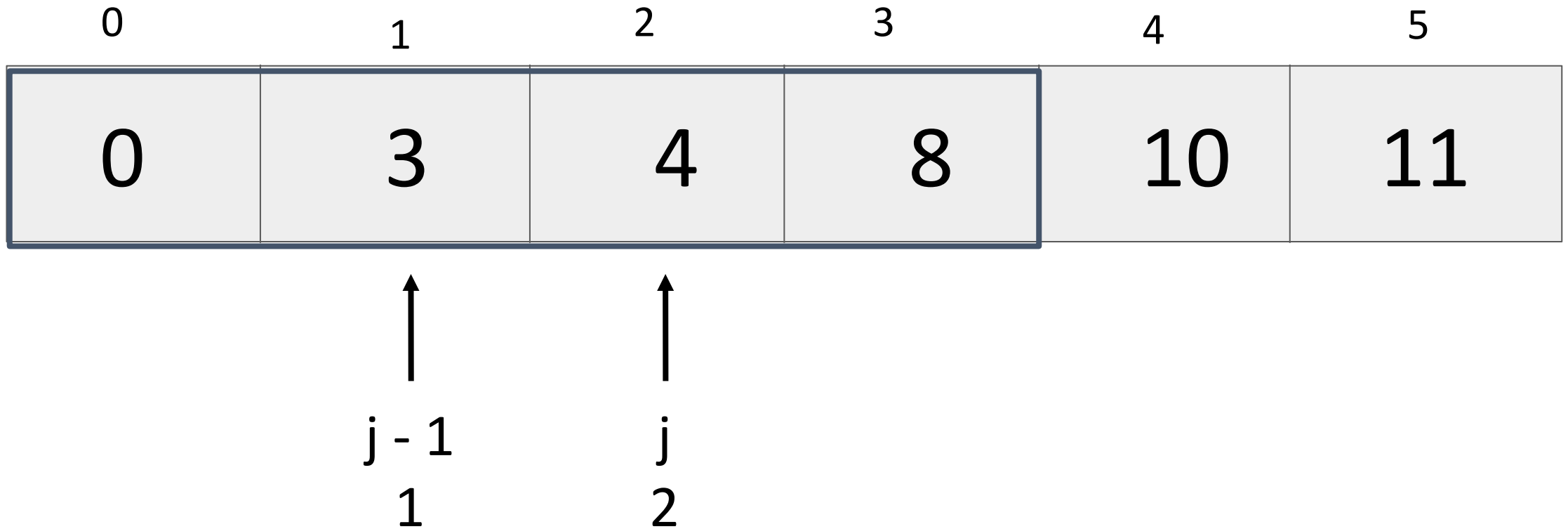


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 58

Bubble Sort

len = 4

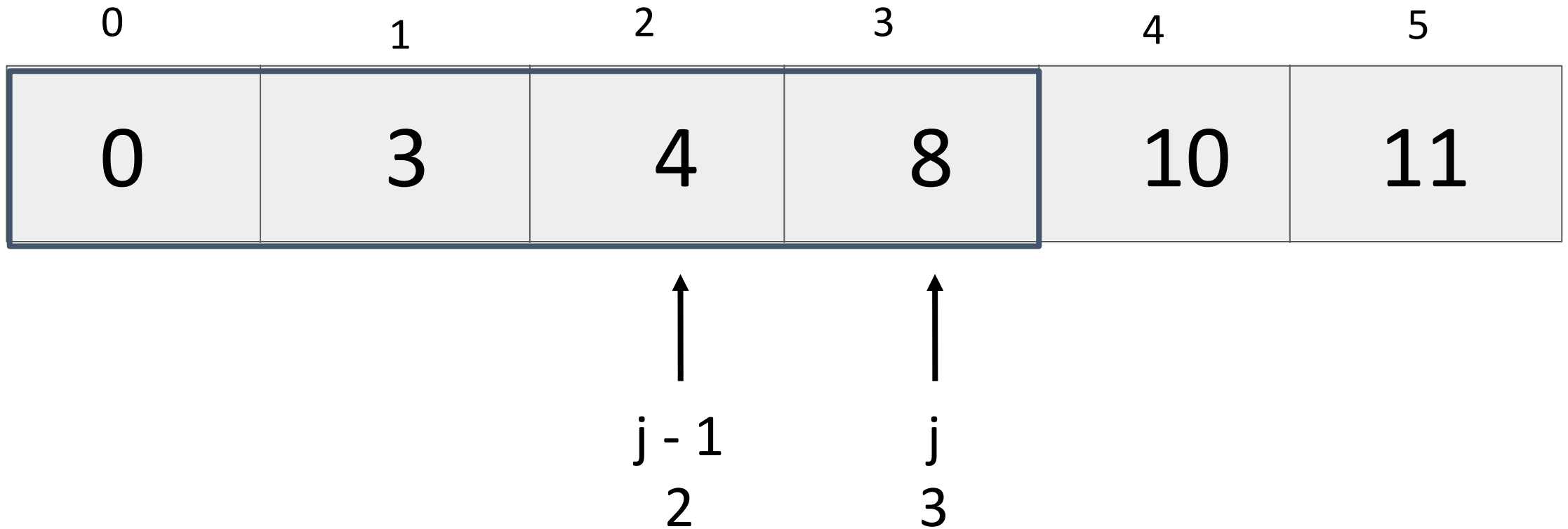


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 59

Bubble Sort

len = 4

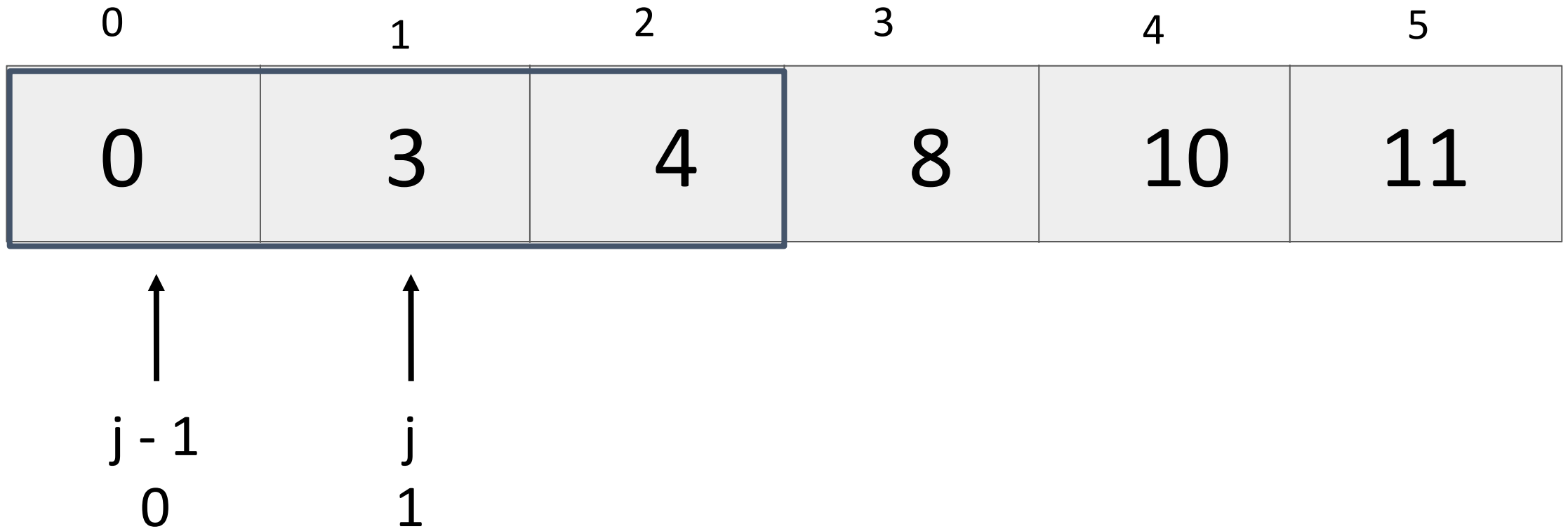


Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 60

Bubble Sort

len = 3

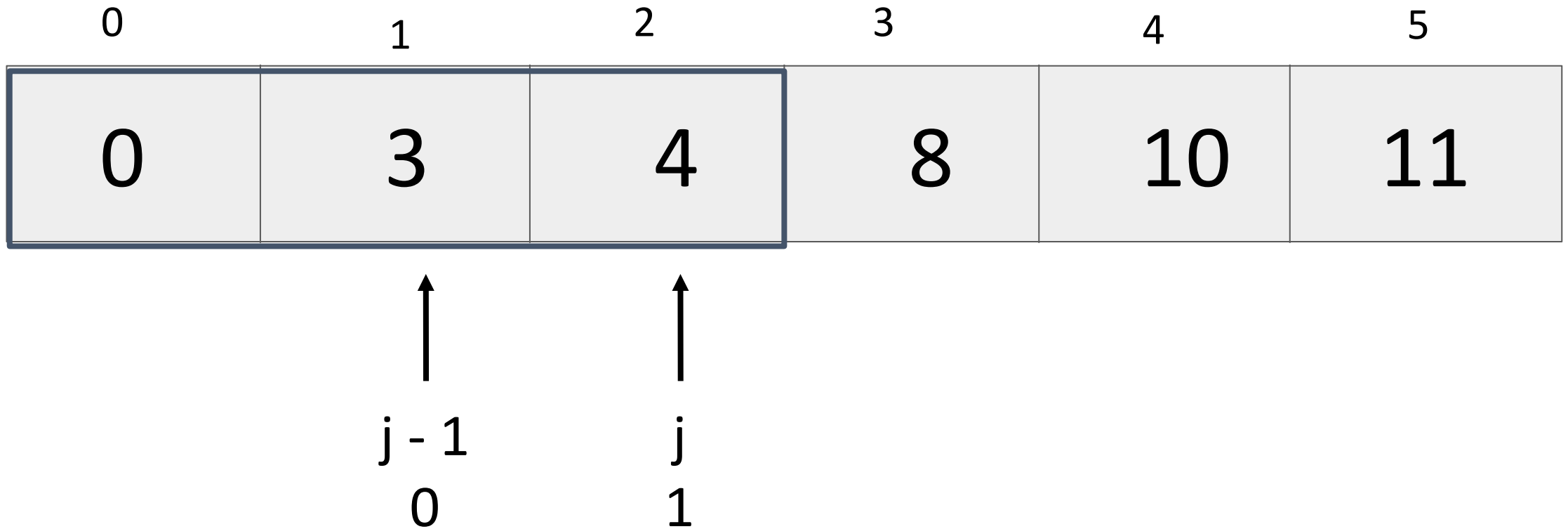


Reset; Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 61

Bubble Sort

len = 3

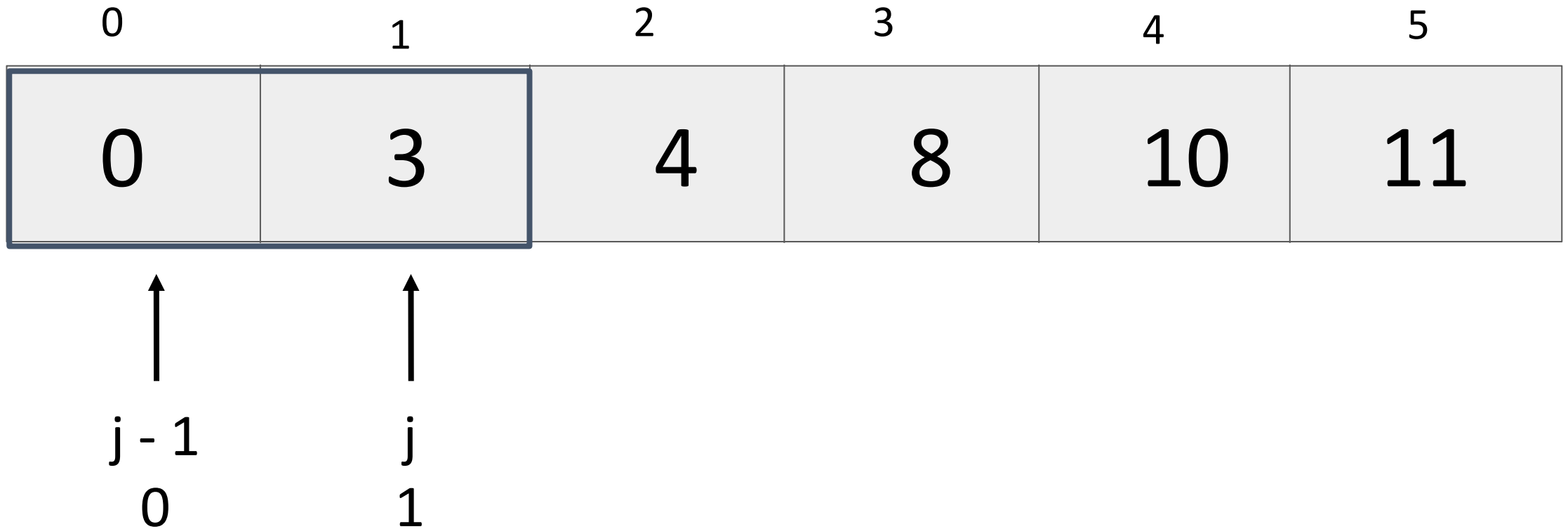


Reset; Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 62

Bubble Sort

len = 2



Reset; Compare $j-1$ and j ; Swap if $L[j-1] > L[j]$

What next? 63

Bubble Sort

Idea: bubble highest values to the end of the list; Check a shorter sublist each time

`bubbleSort(L):`

 for len in [L.length, 1):

 for j in [1, len): # bubble

 if $L[j-1] > L[j]$:

 swap(j-1, j, L)

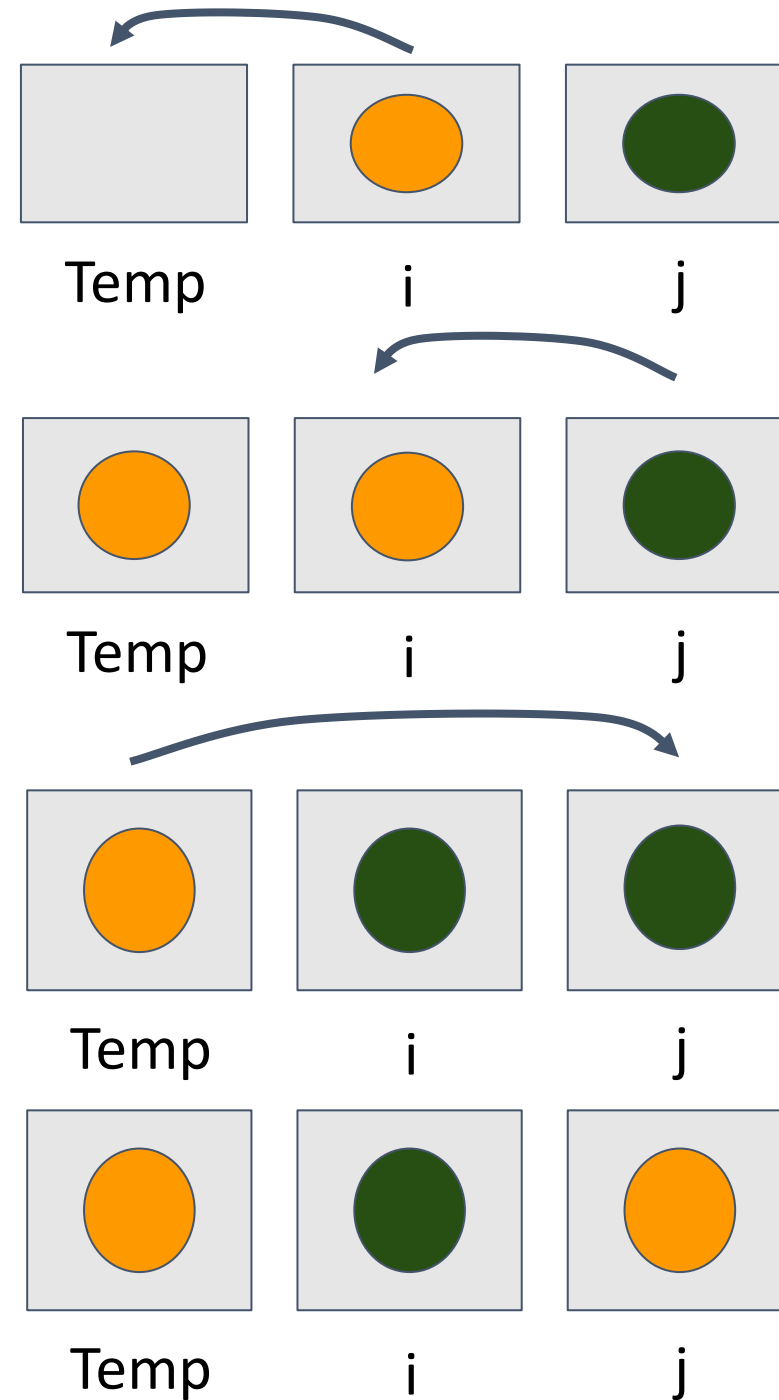
Bubble sort

swap(i, j, L):

temp = L[i] # step 1

L[i] = L[j] # step 2

L[j] = temp # step 3



Selection Sort

Selection sort

Repeatedly find the smallest item and put it at front of list

selectionSort(L):

 for startIdx in range(len(L)):

 minIdx = findMinimum(startIdx, L)

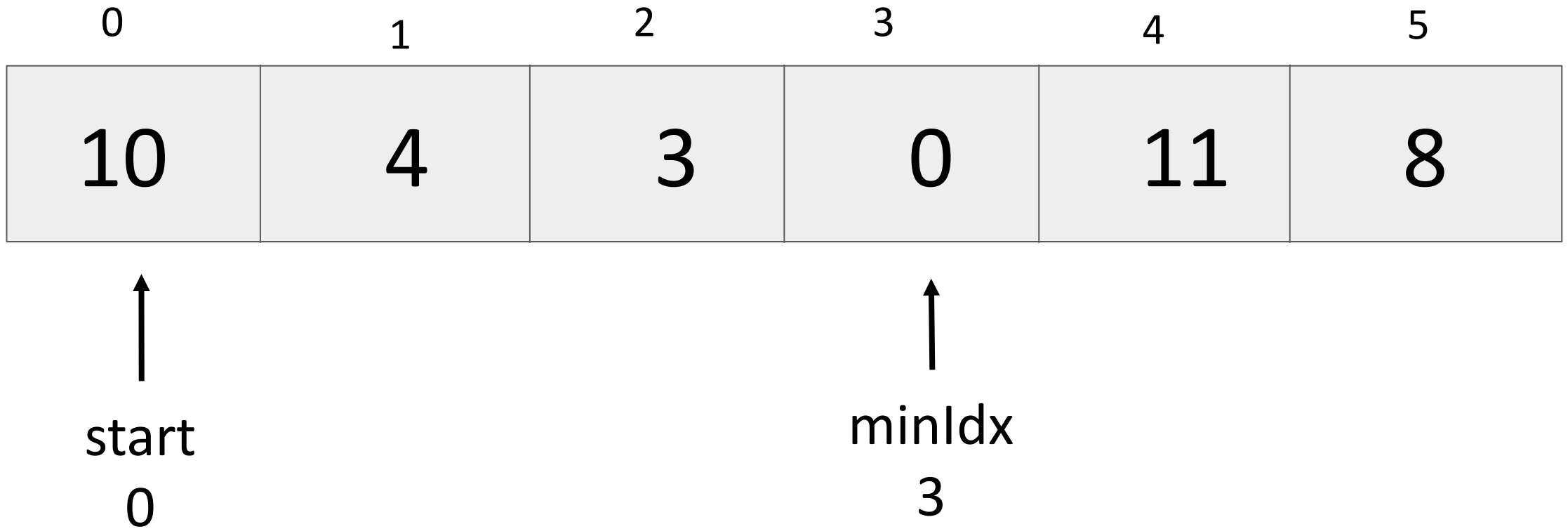
 swap(startIdx, minIdx, L)

Selection Sort

0	1	2	3	4	5
10	4	3	0	11	8

What do we do first?

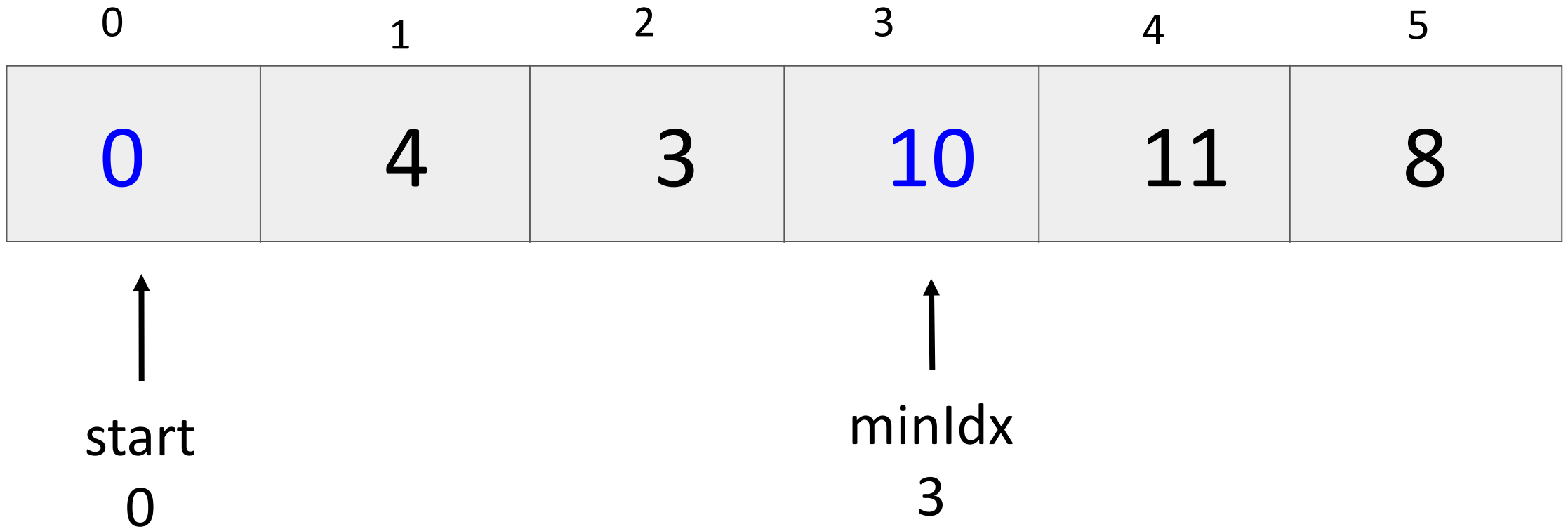
Selection Sort



Find minimum element idx between start to end

What next? 69

Selection Sort



Swap the elements at start and minIdx

What next? 70

Selection Sort

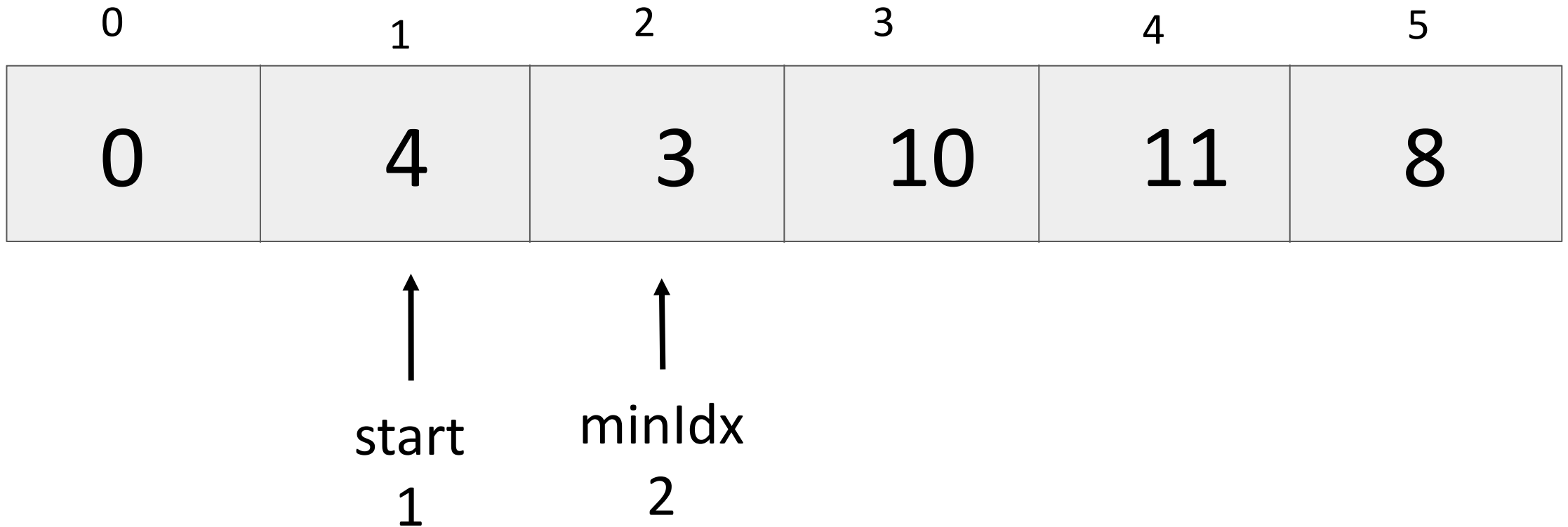
0	1	2	3	4	5
0	4	3	10	11	8

↑
start
1

Decrease the interval.

What next? ⁷¹

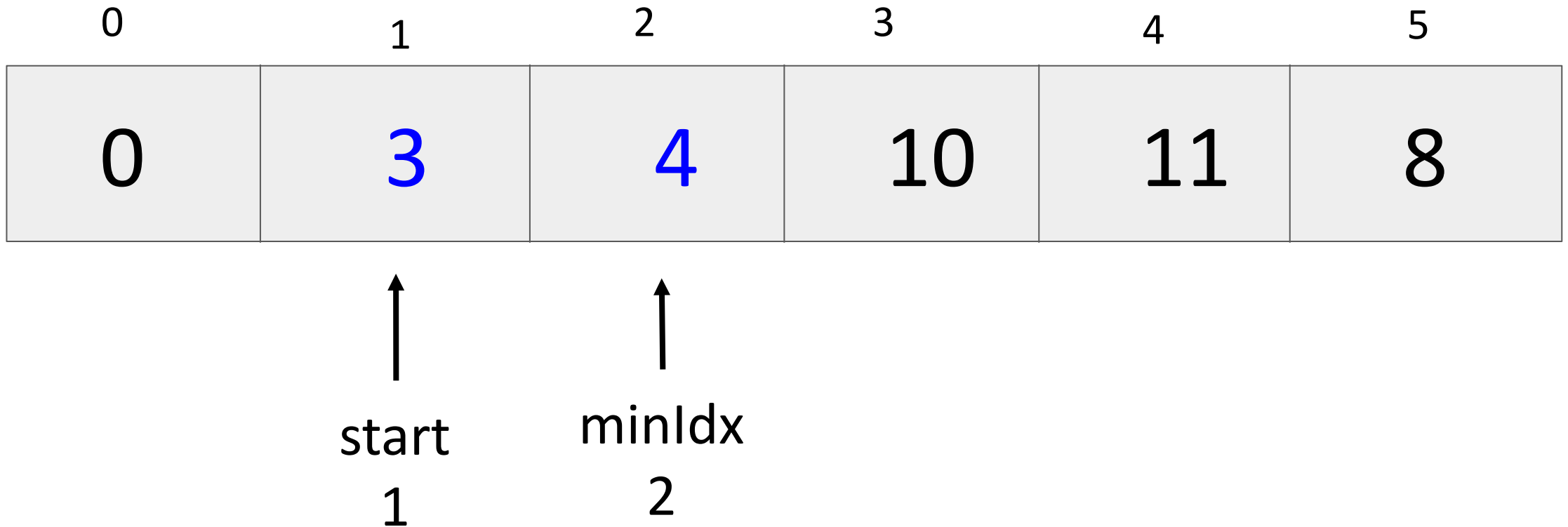
Selection Sort



Find minimum element between start to end

What next? ⁷²

Selection Sort



Swap the elements at start and minIdx

What next? ⁷³

Selection Sort

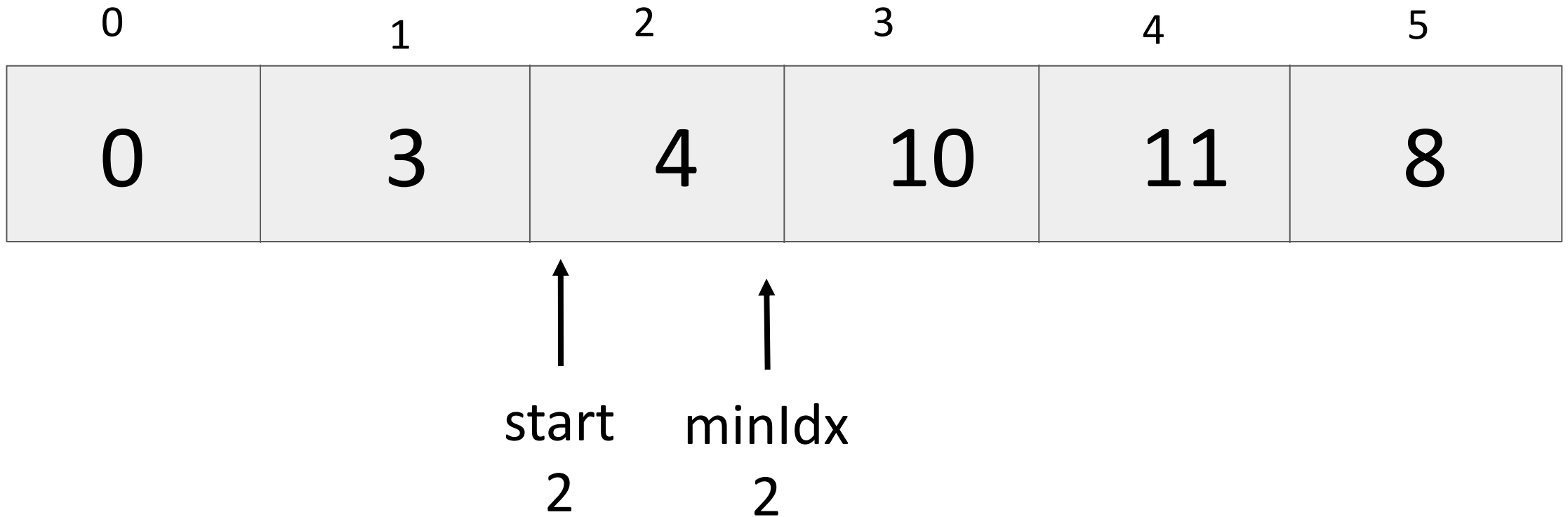
0	1	2	3	4	5
0	3	4	10	11	8

↑
start
2

Decrease the interval.

What next? ⁷⁴

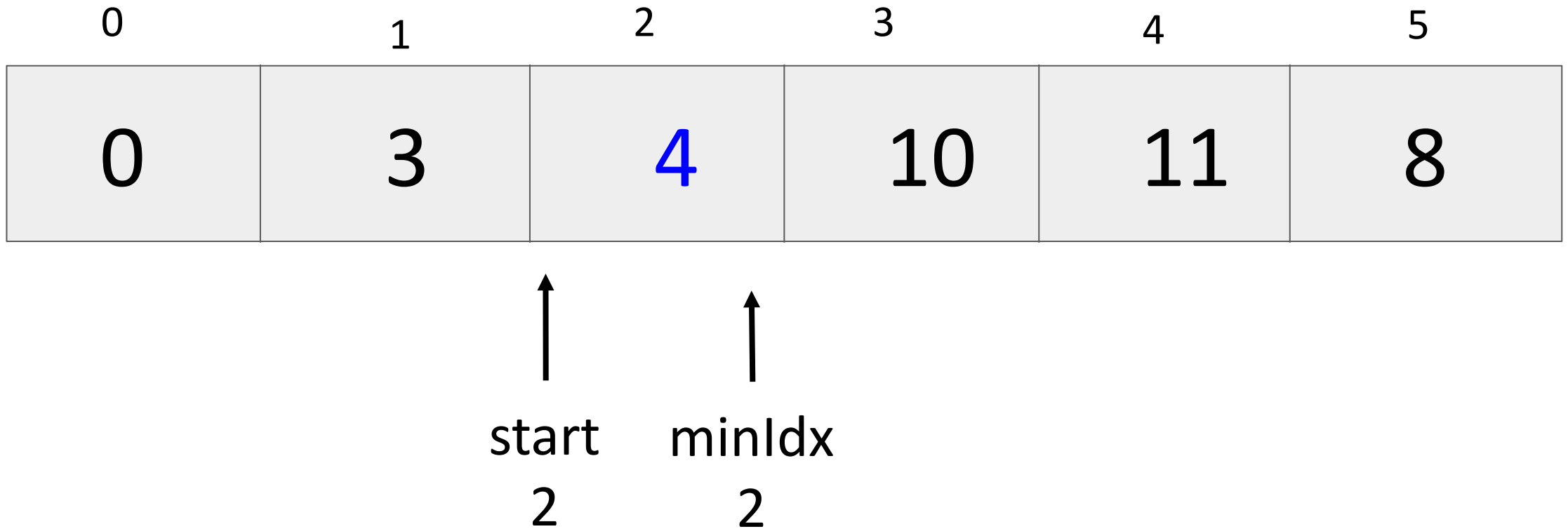
Selection Sort



Find minimum element idx between start to end

What next? ⁷⁵

Selection Sort



Swap the elements at start and minIdx

What next? ⁷⁶

Selection Sort

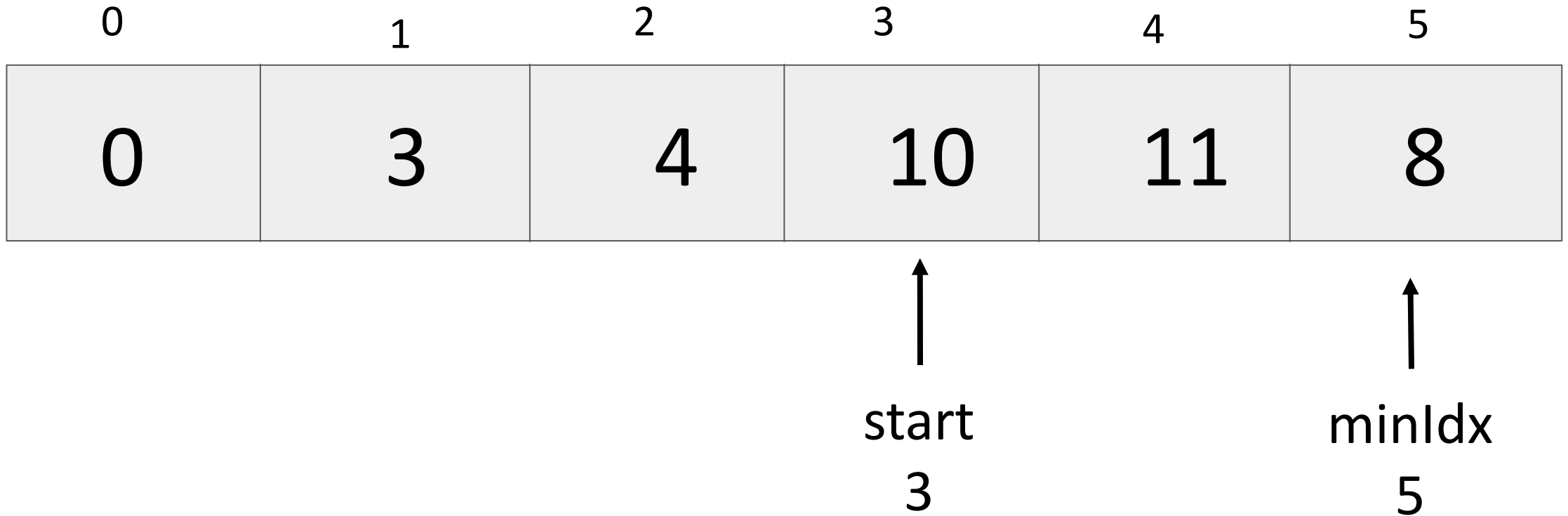
0	1	2	3	4	5
0	3	4	10	11	8

↑
start
3

Decrease the interval.

What next? ⁷⁷

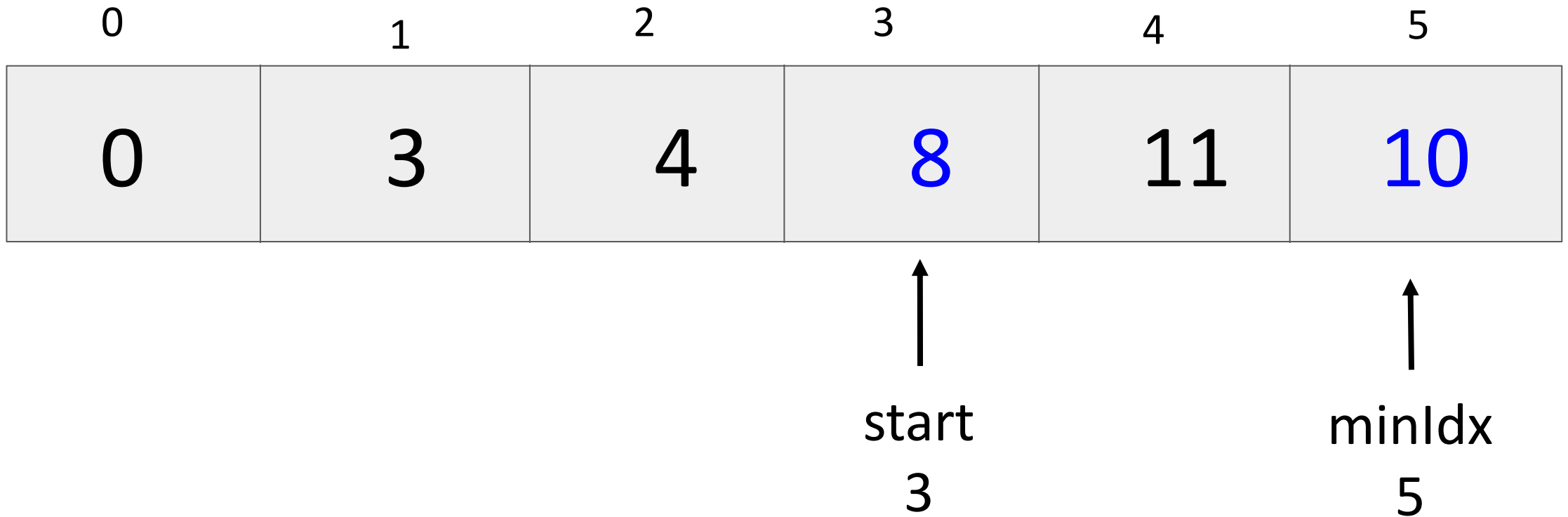
Selection Sort



Find minimum element idx between start to end

What next? ⁷⁸

Selection Sort



Swap the elements at start and minIdx

What next? ⁷⁹

Selection Sort

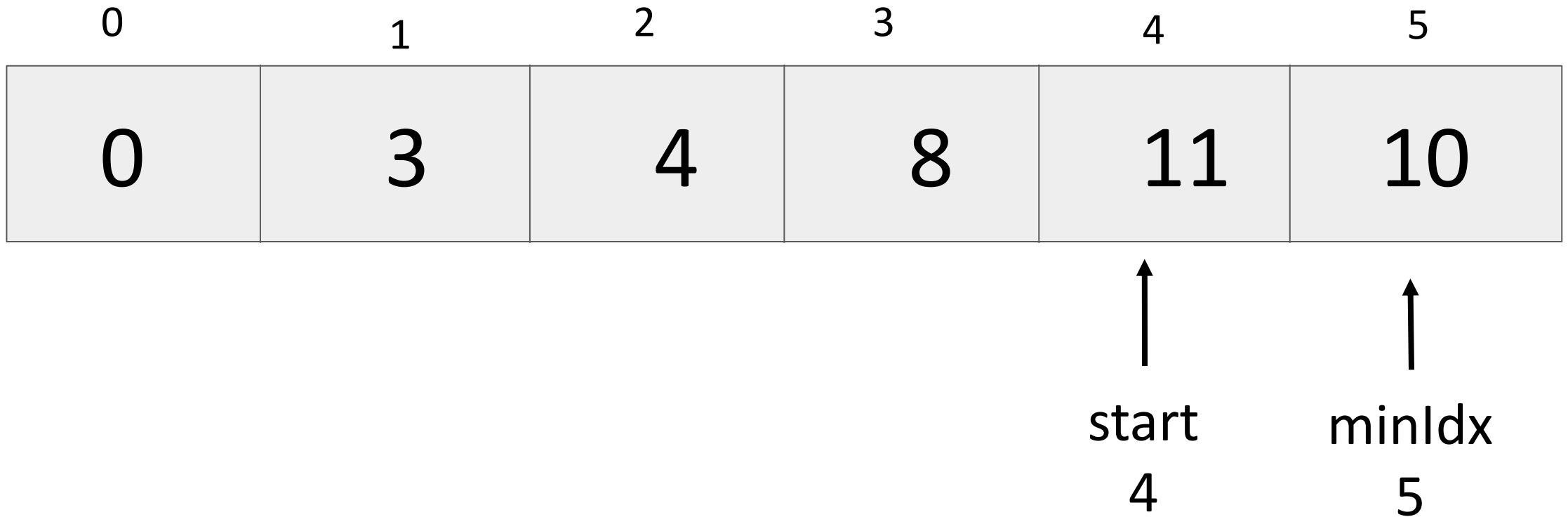
0	1	2	3	4	5
0	3	4	8	11	10

↑
start
4

Decrease the interval.

What next? 80

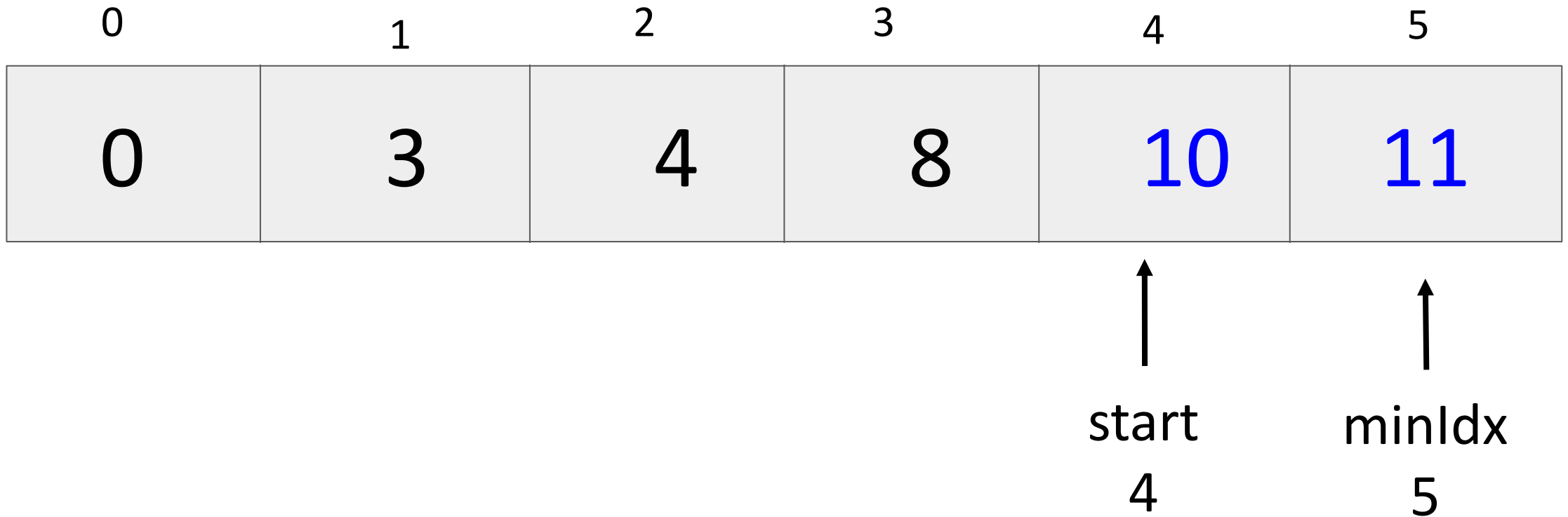
Selection Sort



Find minimum element idx between start to end

What next? 81

Selection Sort



Swap the elements at start and minIdx

What next? 82

Selection Sort

0	1	2	3	4	5
0	3	4	8	10	11

↑
start
5

Decrease the interval.

We're done!

Selection sort

```
findMinimum(startIdx, L):
```

```
    minIdx = startIdx
```

```
    for i in range(startIdx, len(L)):
```

```
        if L[i] < L[minIdx]:
```

```
            minIdx = i
```

```
    return minIdx
```

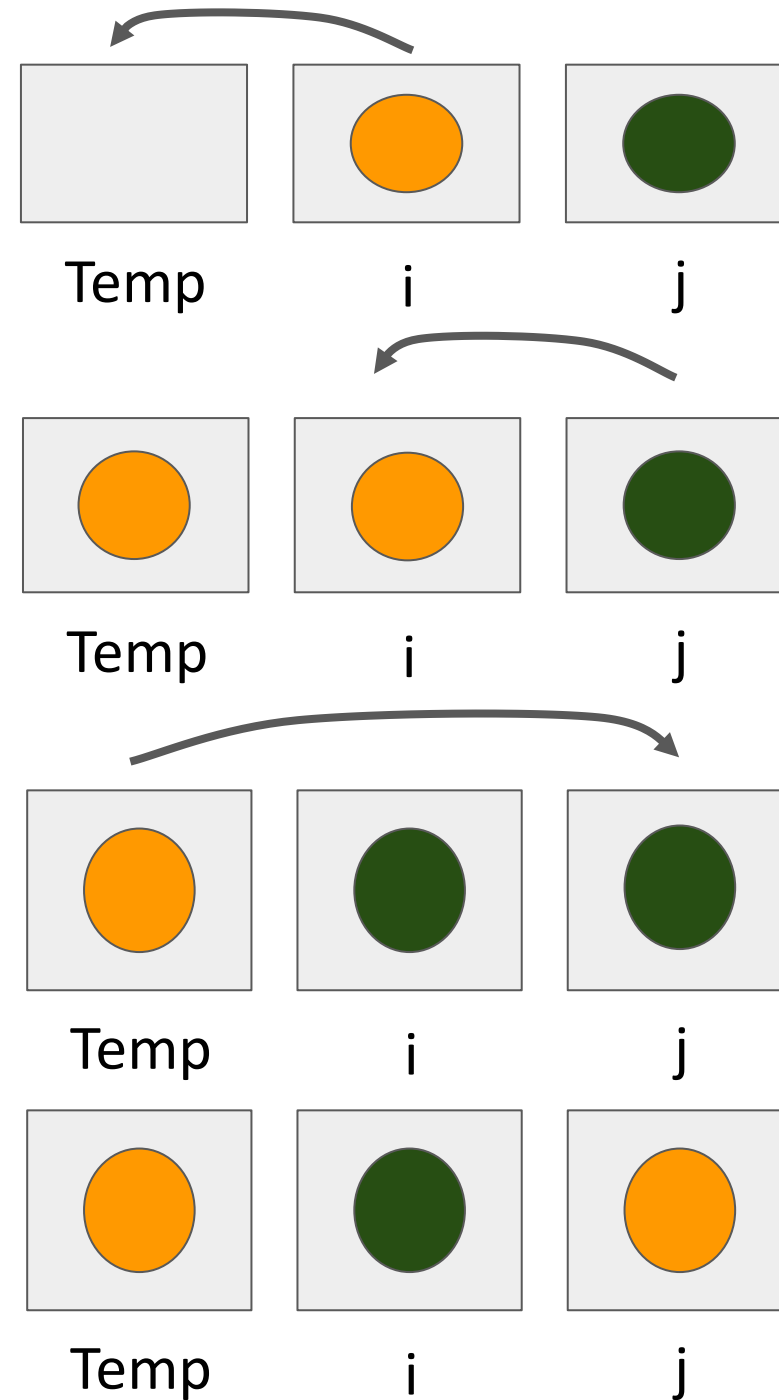
Swap

swap(i, j, L):

temp = L[i] # step 1

L[i] = L[j] # step 2

L[j] = temp # step 3



Selection sort and Bubble sort are $O(N^2)$

