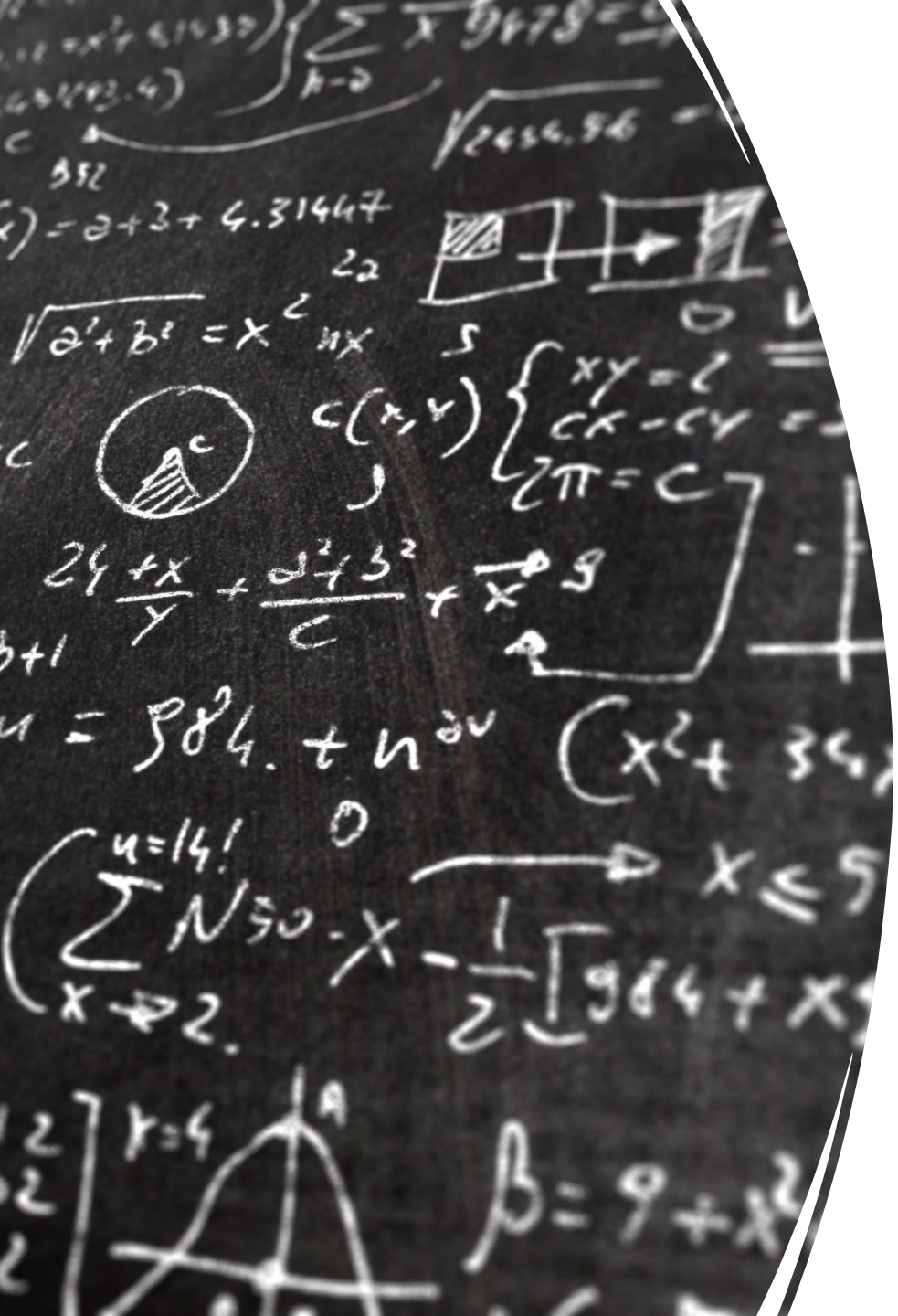


CS 113 – Computer Science I

Lecture 21 – Binary Search, Sorting

Adam Poliak

04/04/2023

Announcements

- Midsemester feedback
- Office hours Thursday
  - Cancelled
- Schedule:
  - Thursday 04/06 – NO CLASS
  - Tuesday 04/11 – remote class
  - Thursday 04/13 – Midterm

Outline

- Review – Linear Search
- Binary Search
- Sorting

Linear Search

These previous approaches are examples of 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

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

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

Yes, if the list is sorted

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

0 1 2 3 4 5

String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 99

low	mid	high	ls[mid]

Binary search

0 1 2 3 4 5

String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 99

low	mid	high	ls[mid]
0	2	5	44

Binary search

0 1 2 3 4 5

String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 99

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

Binary search

0 1 2 3 4 5
String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 30

low	mid	high	ls[mid]

Binary search

0 1 2 3 4 5
String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 30

low	mid	high	ls[mid]
0	2	5	44

Binary search

0 1 2 3 4 5
String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 30

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

Binary search

0 1 2 3 4 5
String[] ls = {-20, -4, 44, 58, 99, 145}

Search for 30

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

Binary search

0 1 2 3 4 5
String[] ls = {-20, -4, 44, 58, 99, 145}

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 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	

Guessing game – in class exercise

Pair up:

- Person A chooses a number between 1 and 64
- 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

After 2 rounds each, choose a number between 1 and 512

Binary Search run time

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

The time increases by $\log n$ $(\text{we use base } 2)$

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:

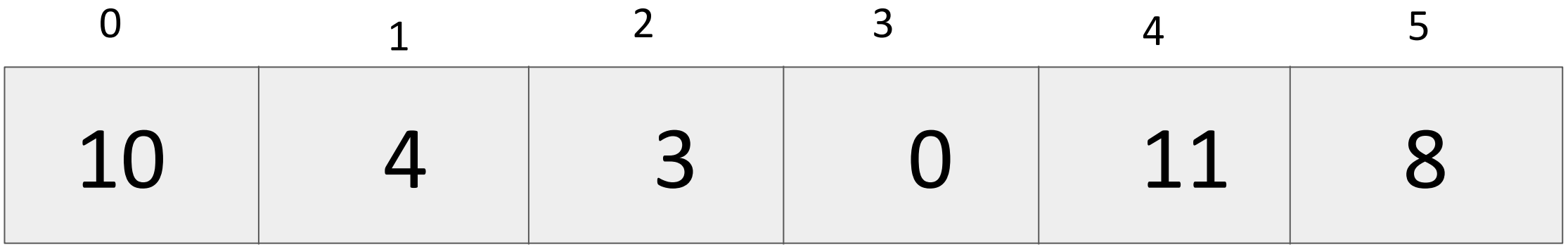
$$\log_2 512 = 9$$

Outline

- Review – Linear Search
- Binary Search
- **Sorting**

Sorting

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



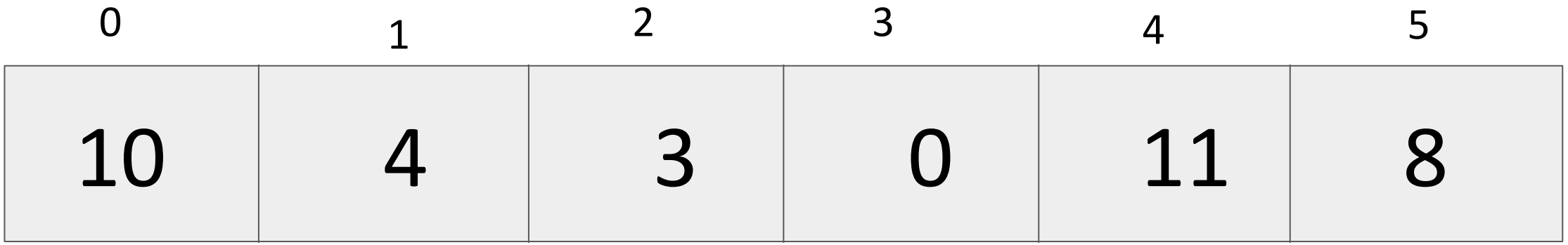
Bubble Sort

Compare two adjacent items, and swap if needed

Repeat until largest item is at the back

Repeat process until done

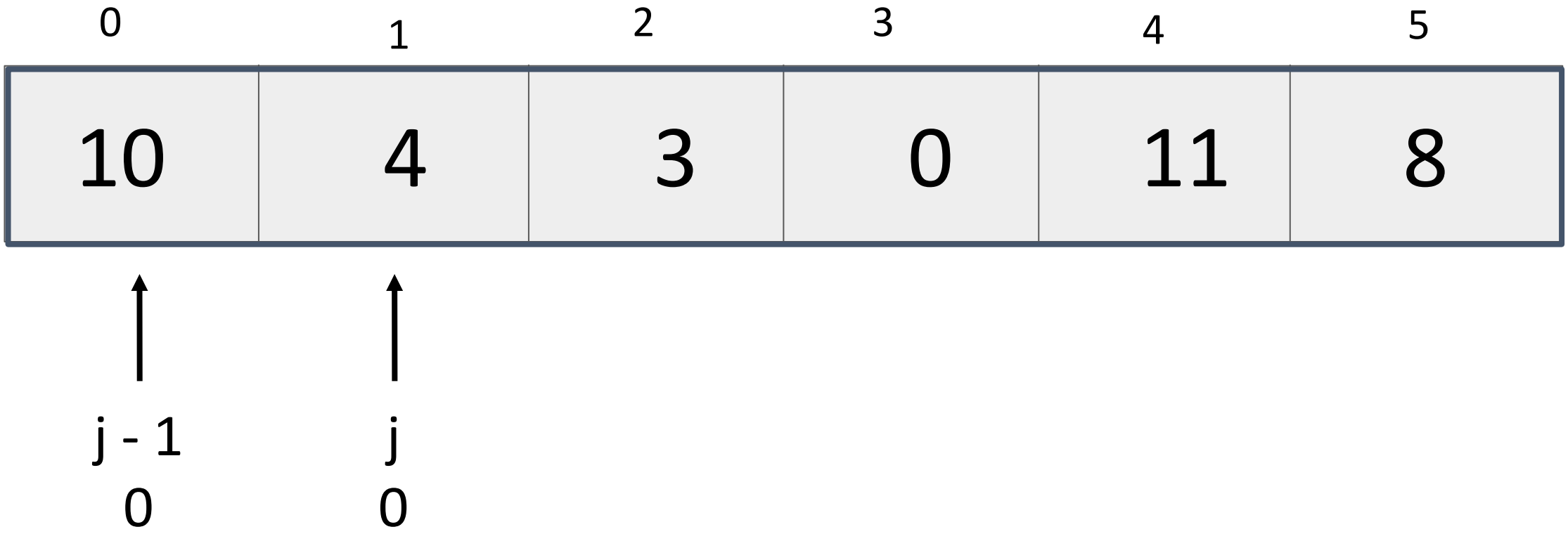
Bubble Sort



What do we do first?

Bubble Sort

len = 6

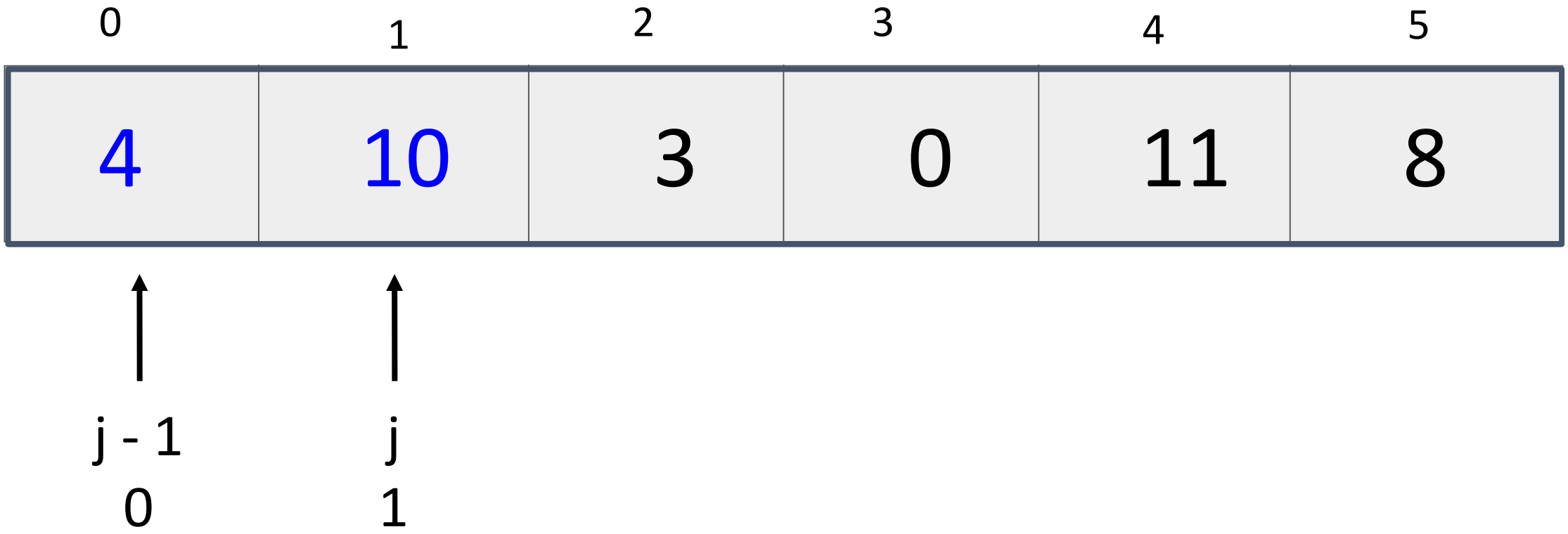


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

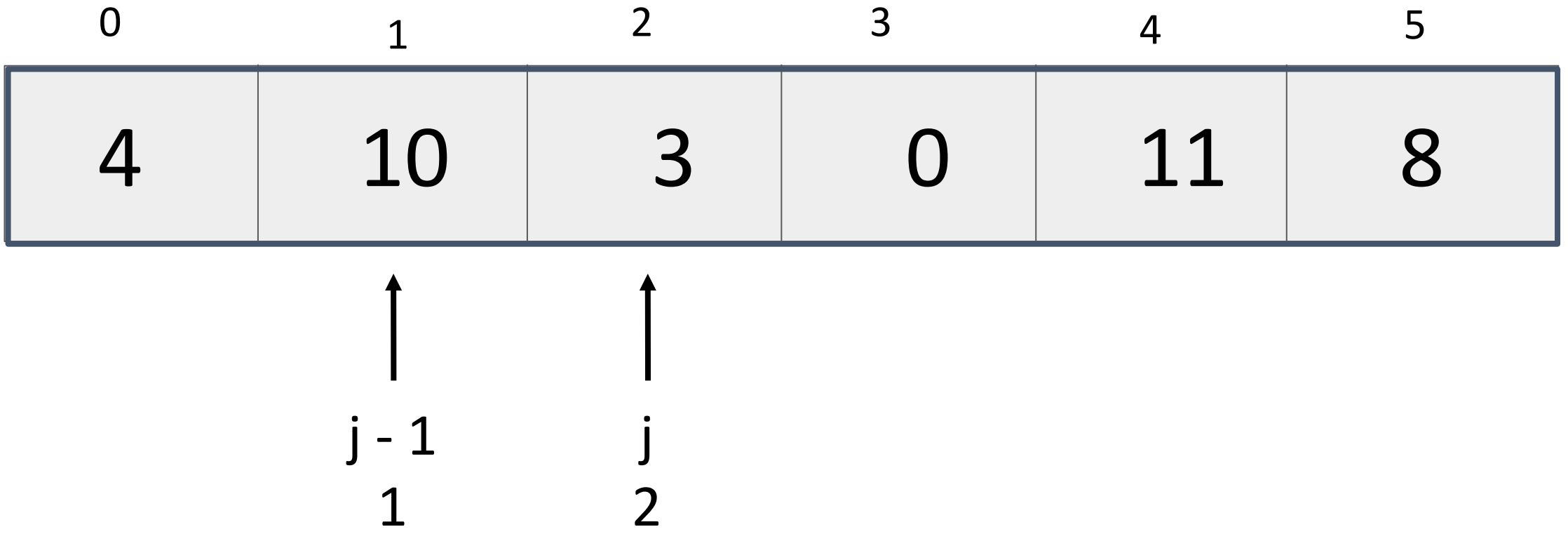


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

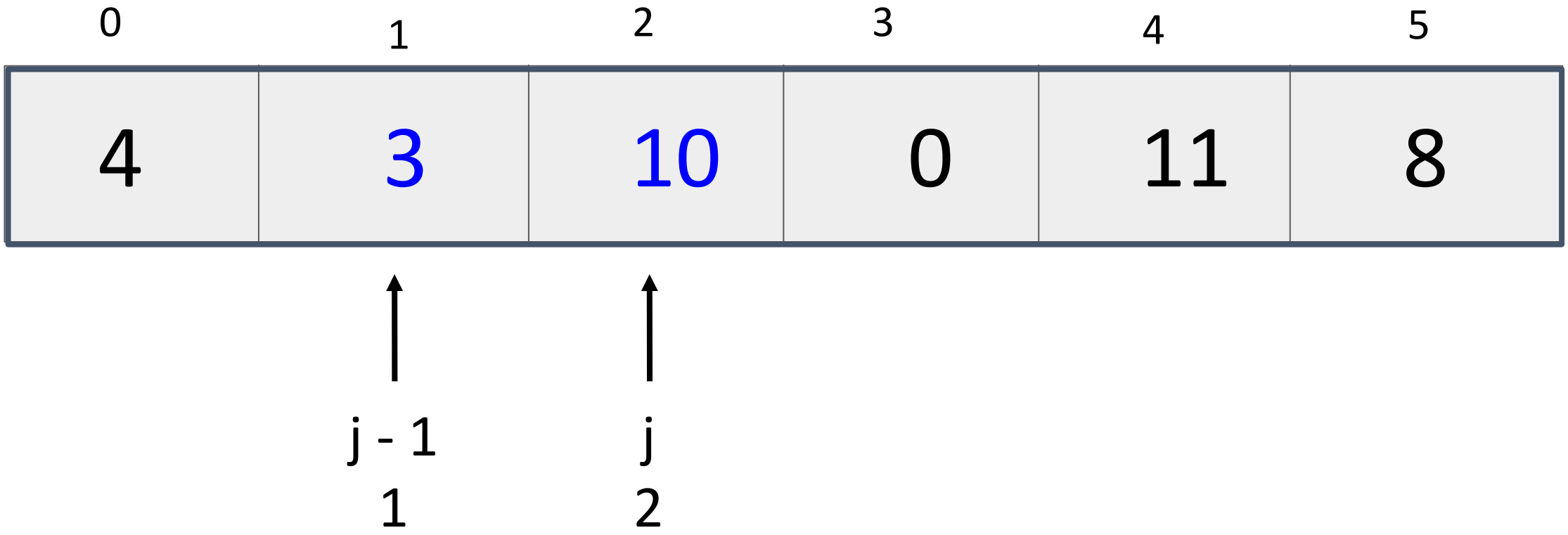


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

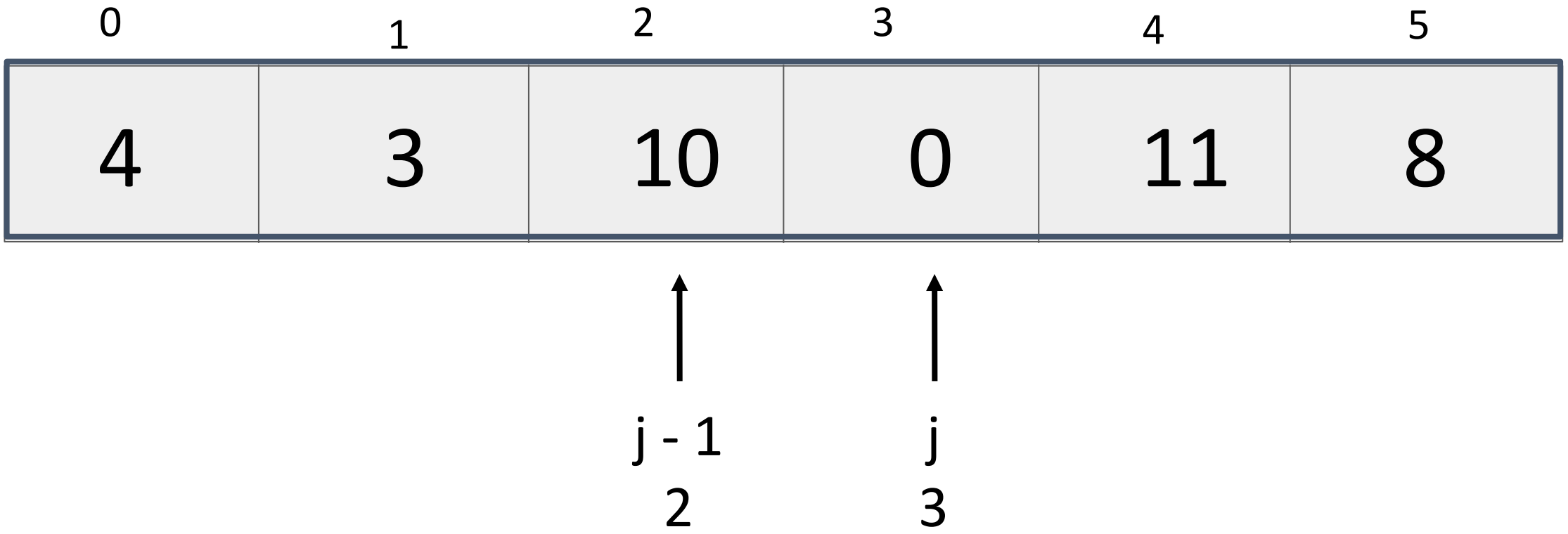


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

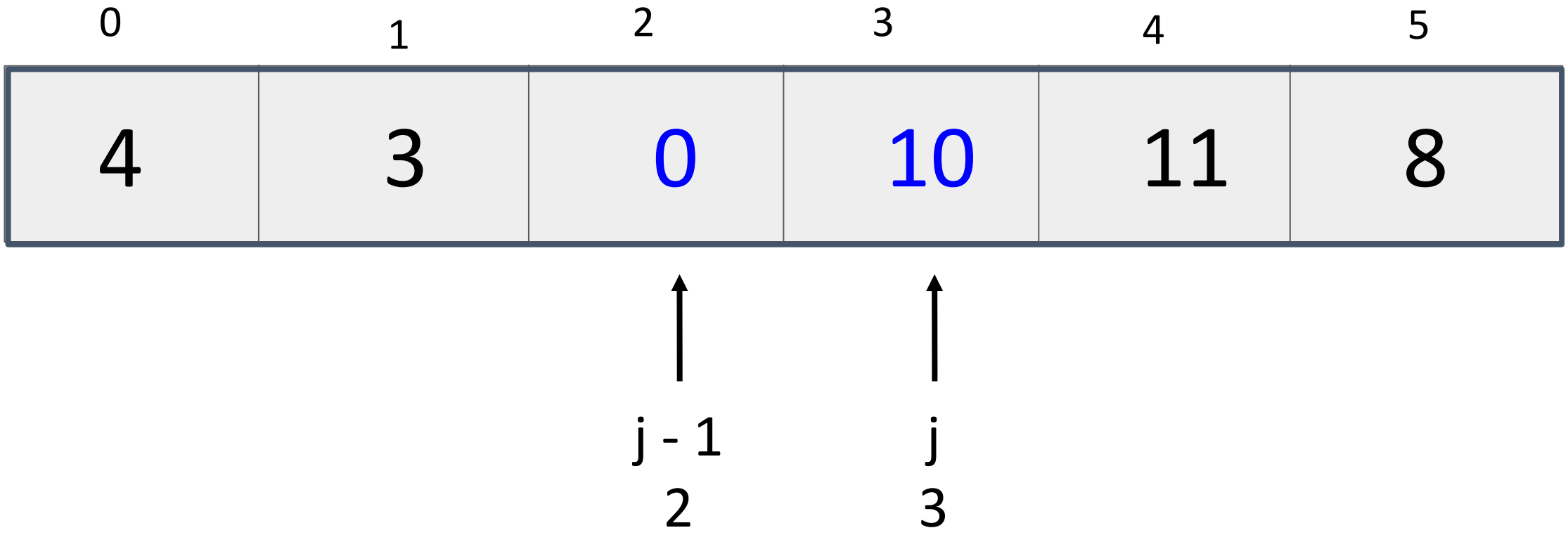


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

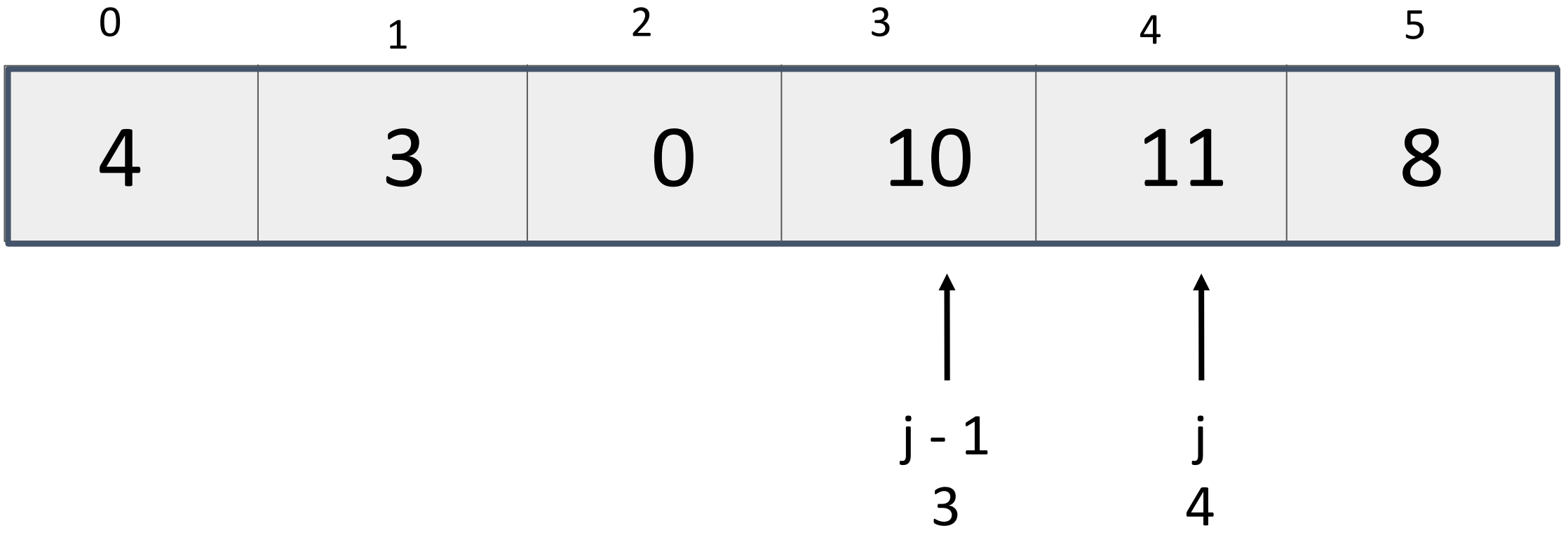


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

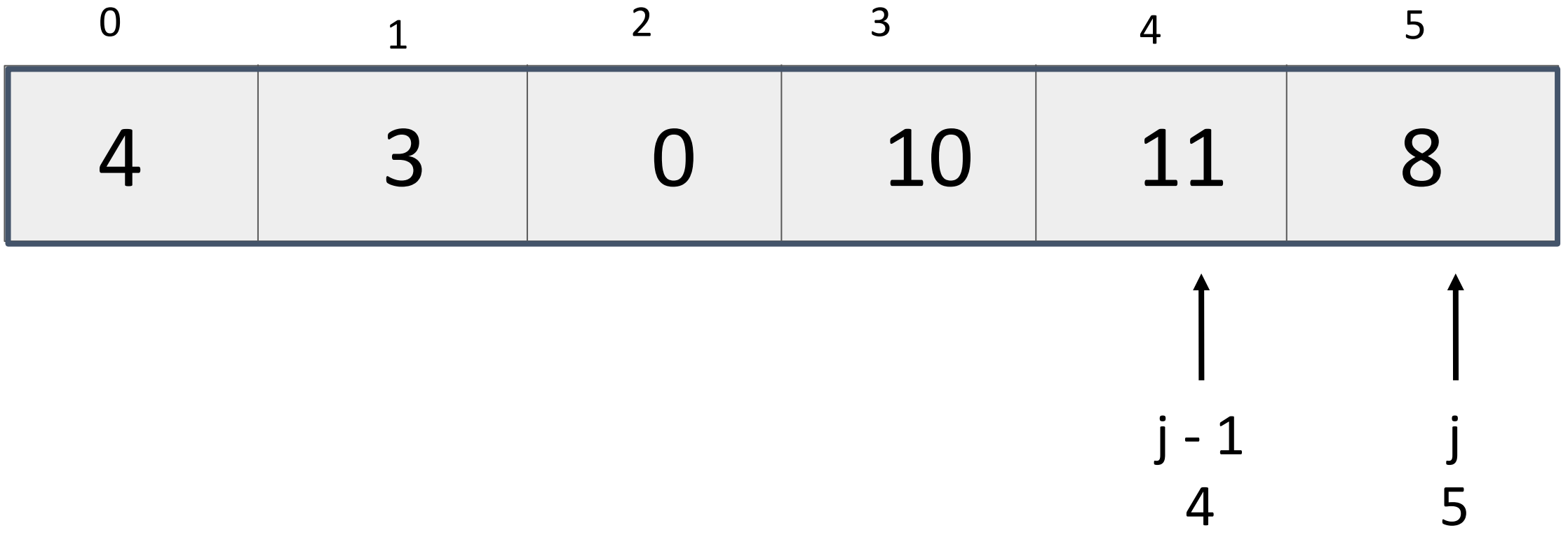


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

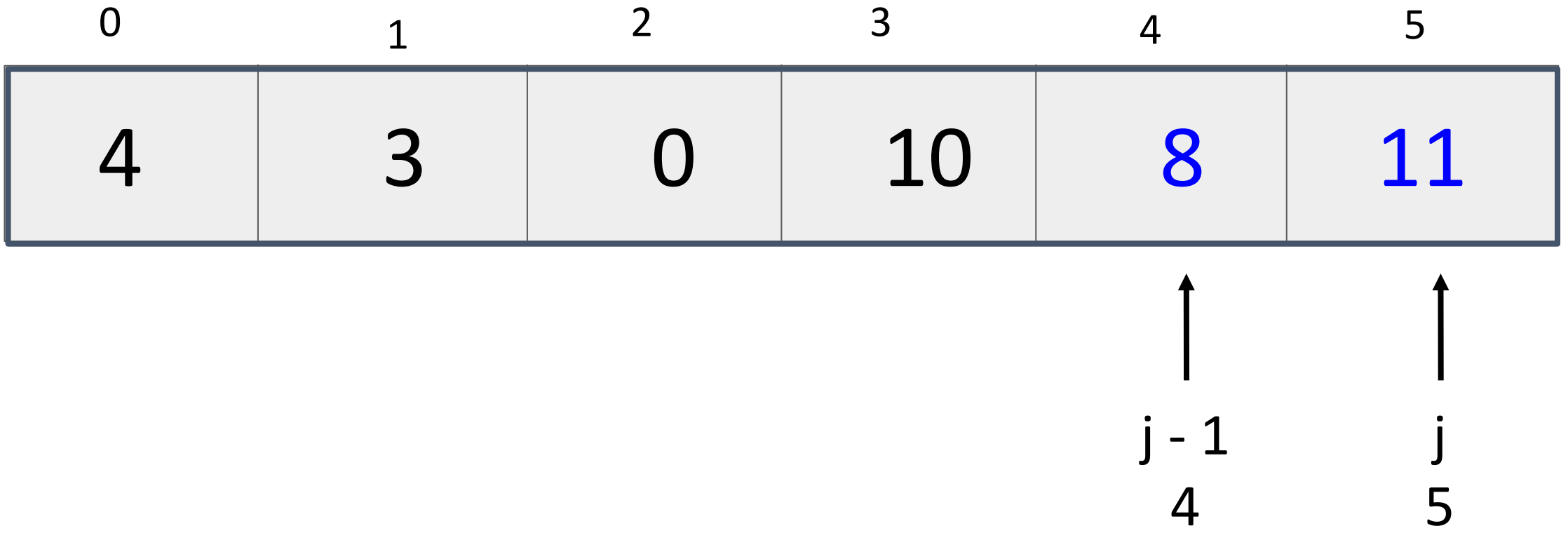


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

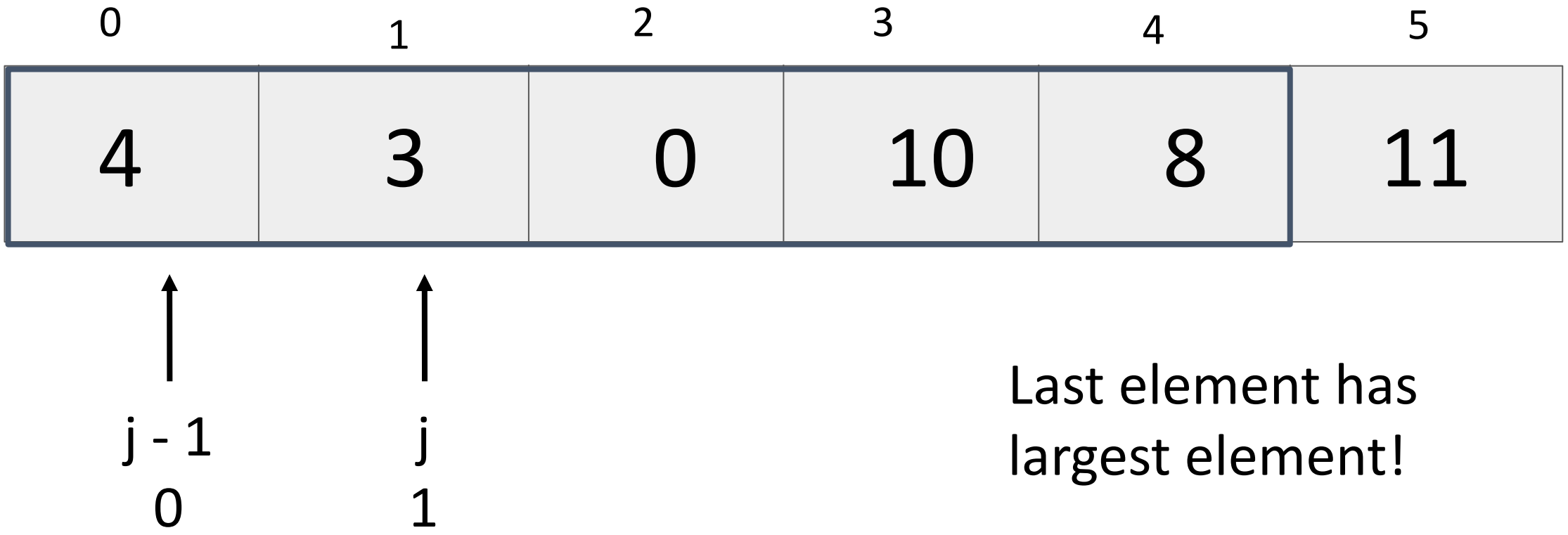


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5



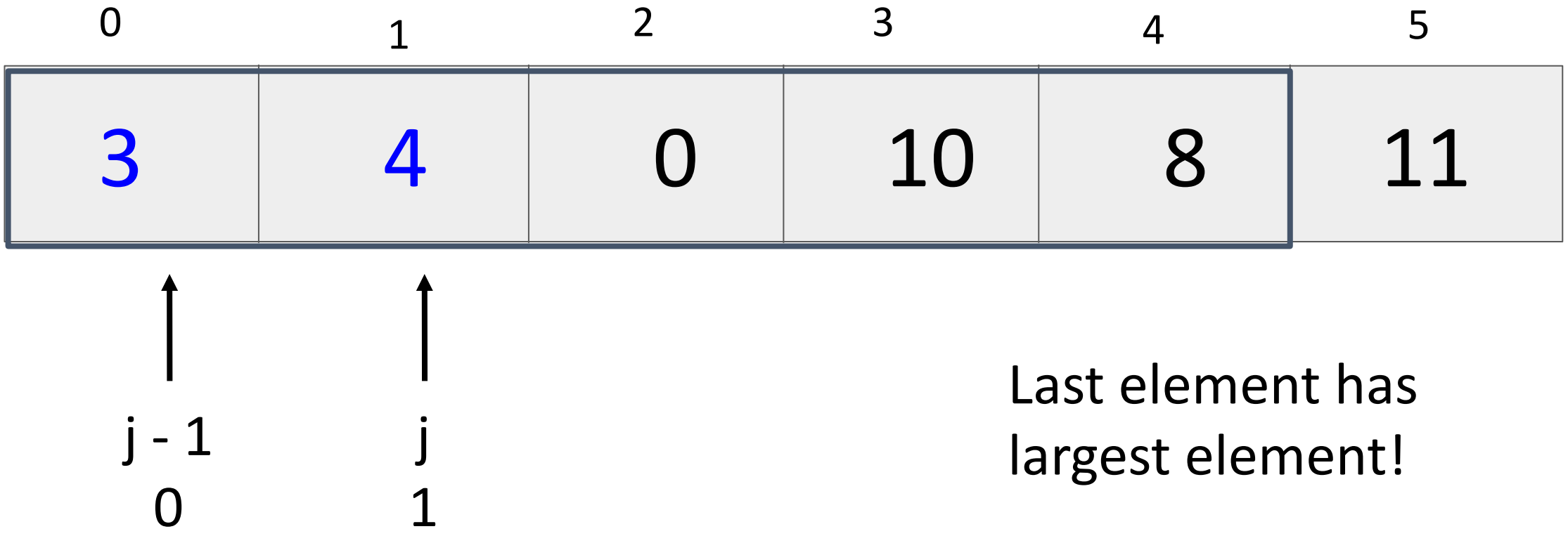
Last element has largest element!

Reset and compare pairs with shorter list!

What next?

Bubble Sort

len = 5



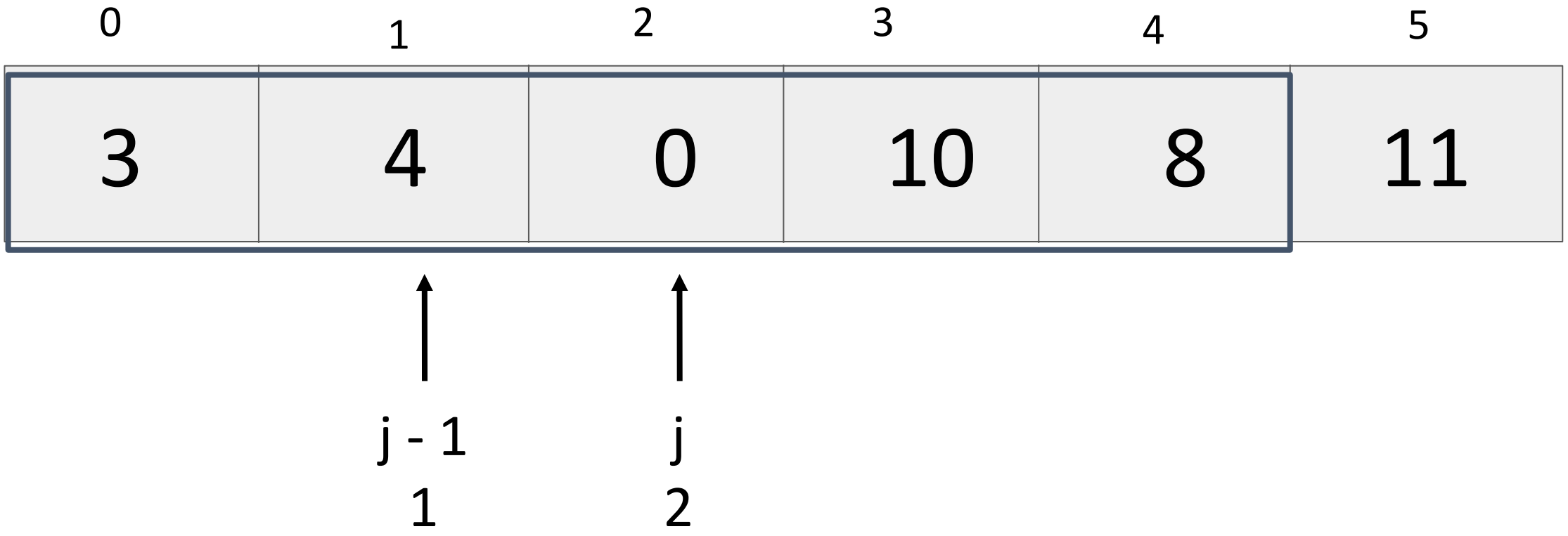
Last element has largest element!

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

What next?

Bubble Sort

len = 5

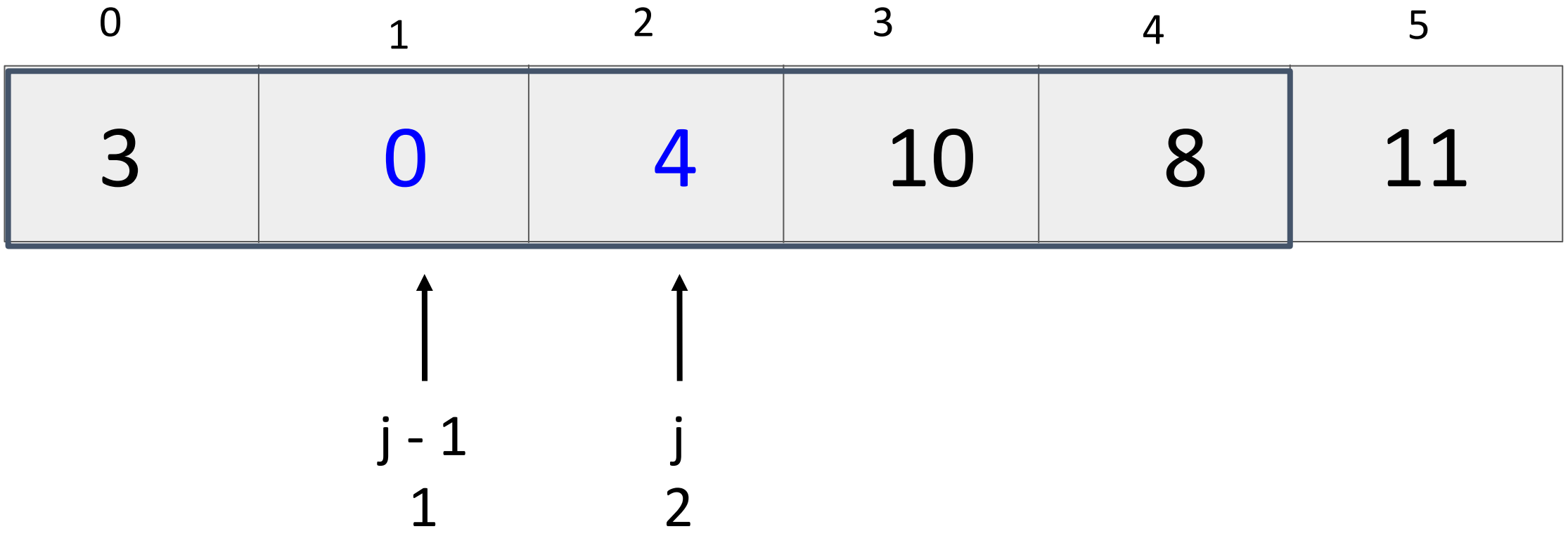


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

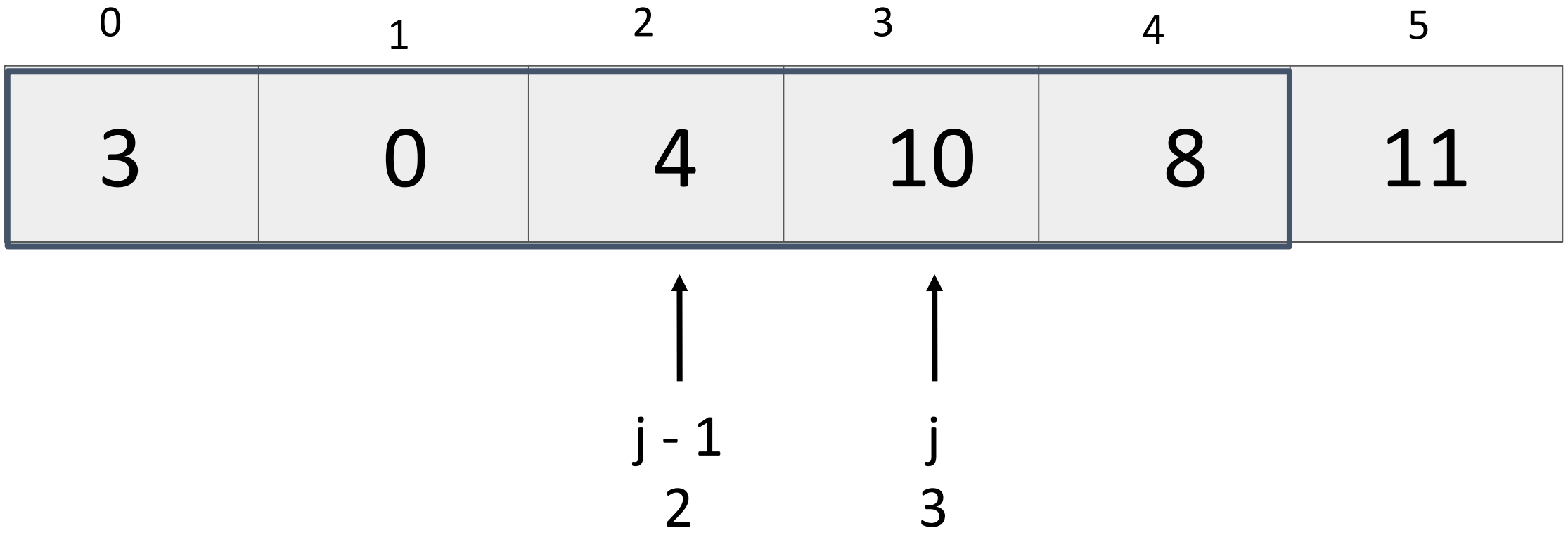


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

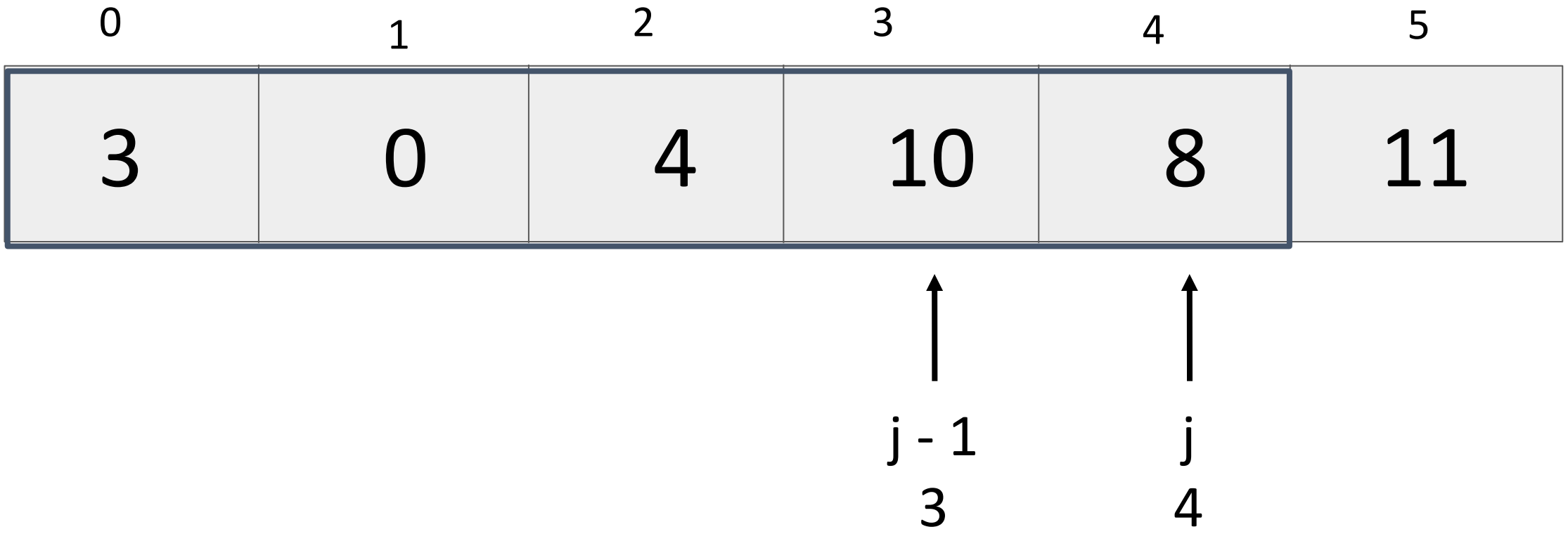


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

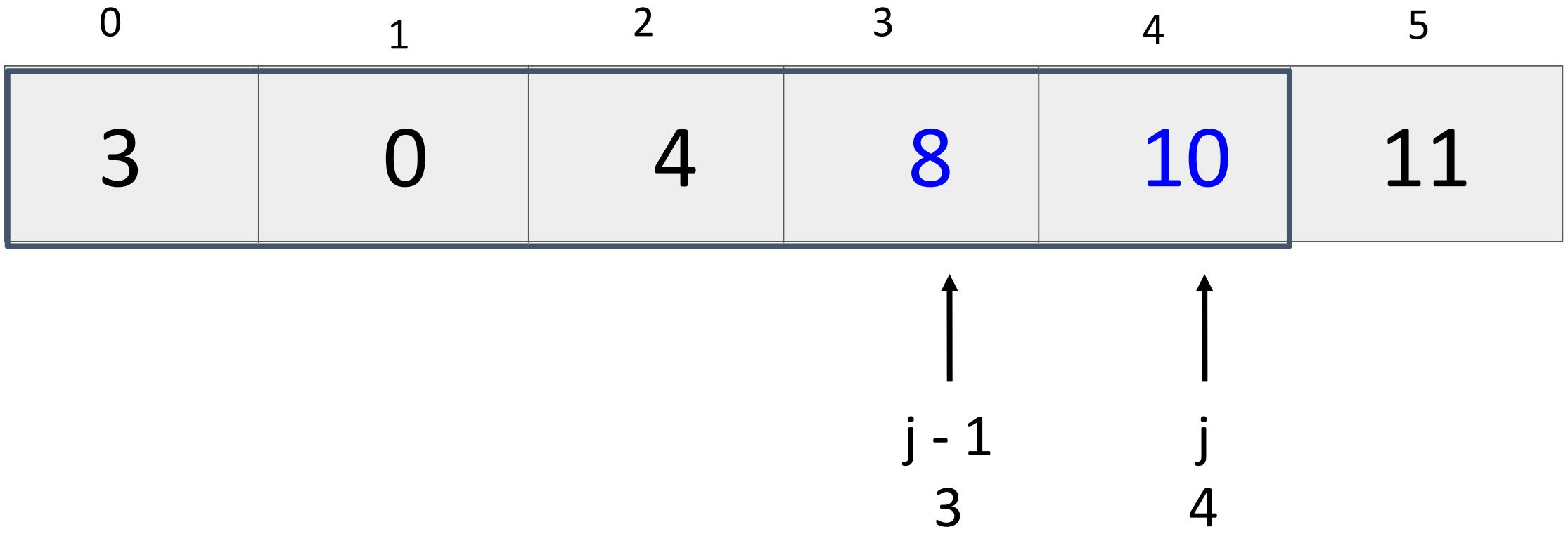


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

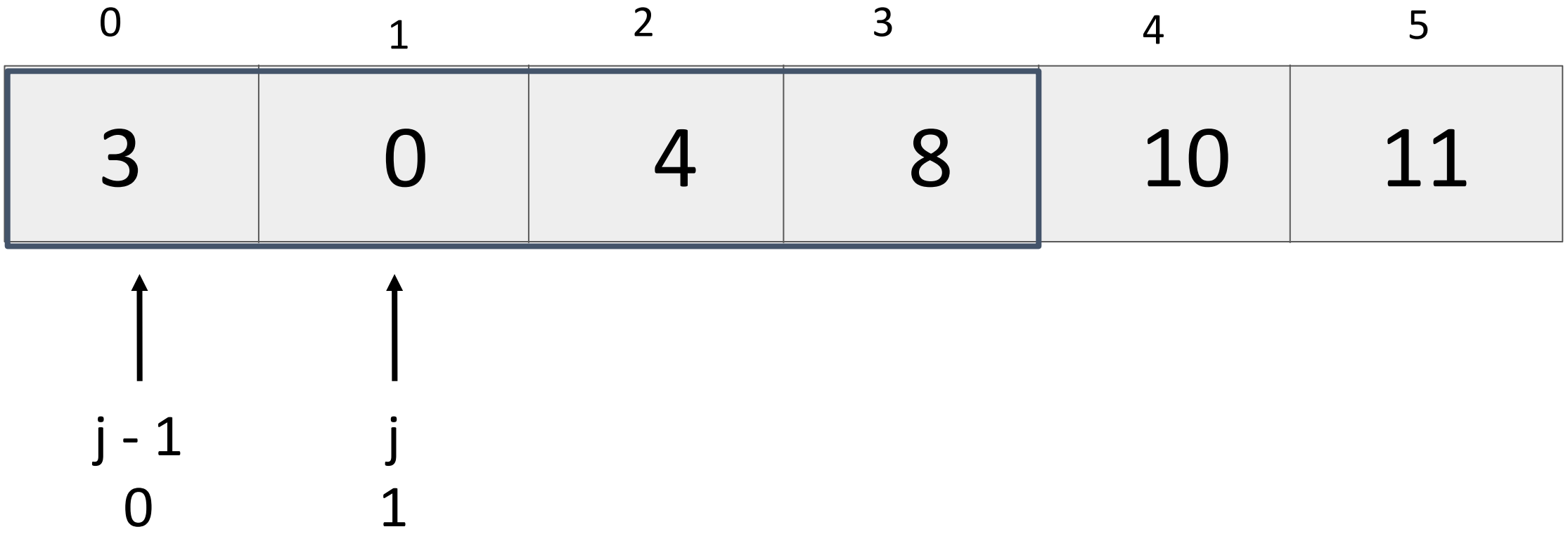


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 4

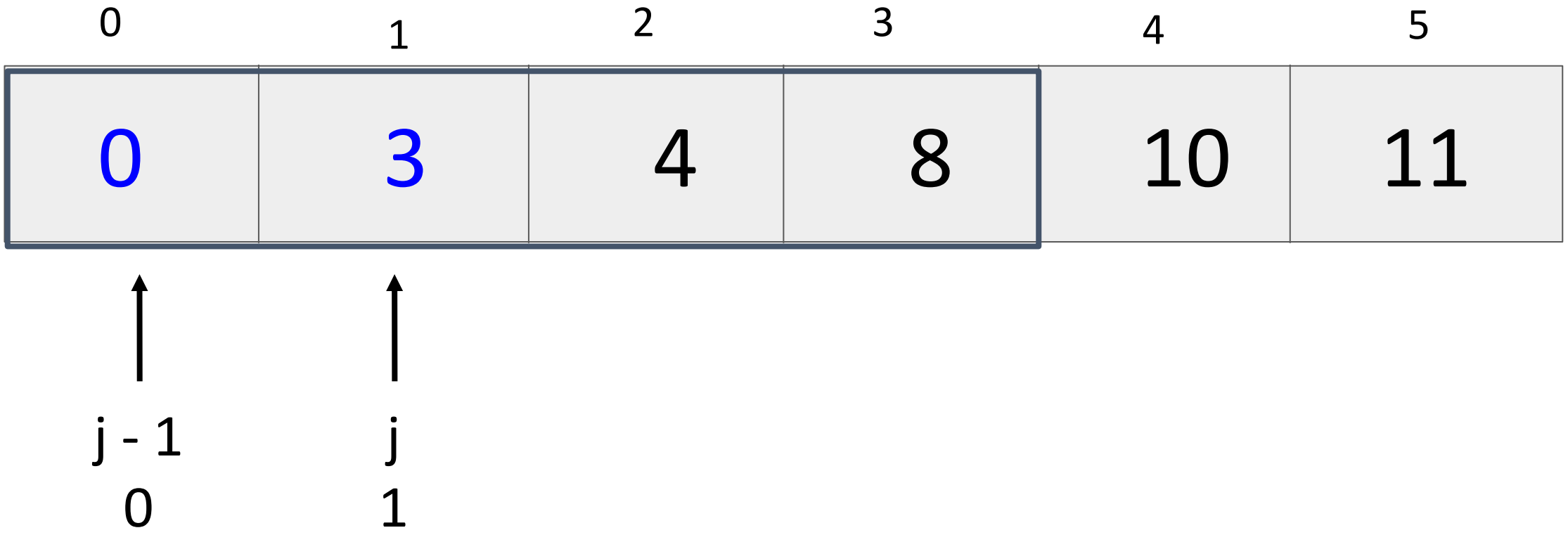


Reset and check pairs with shorter list

What next?

Bubble Sort

len = 4

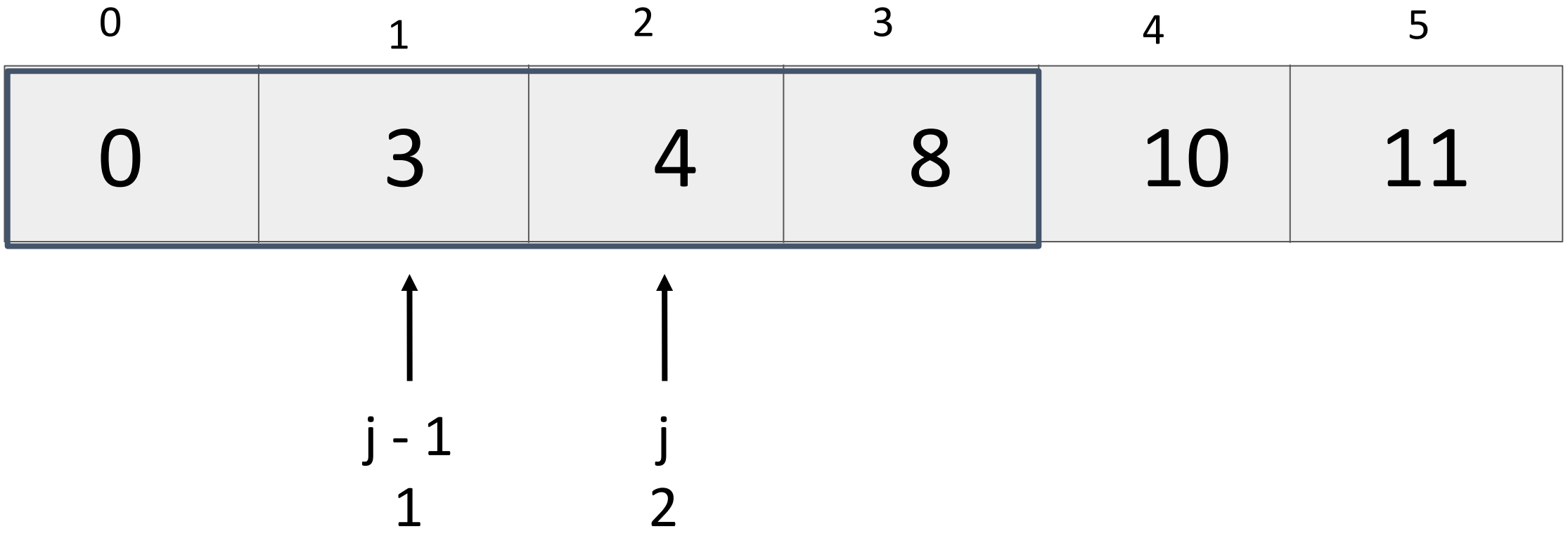


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 4

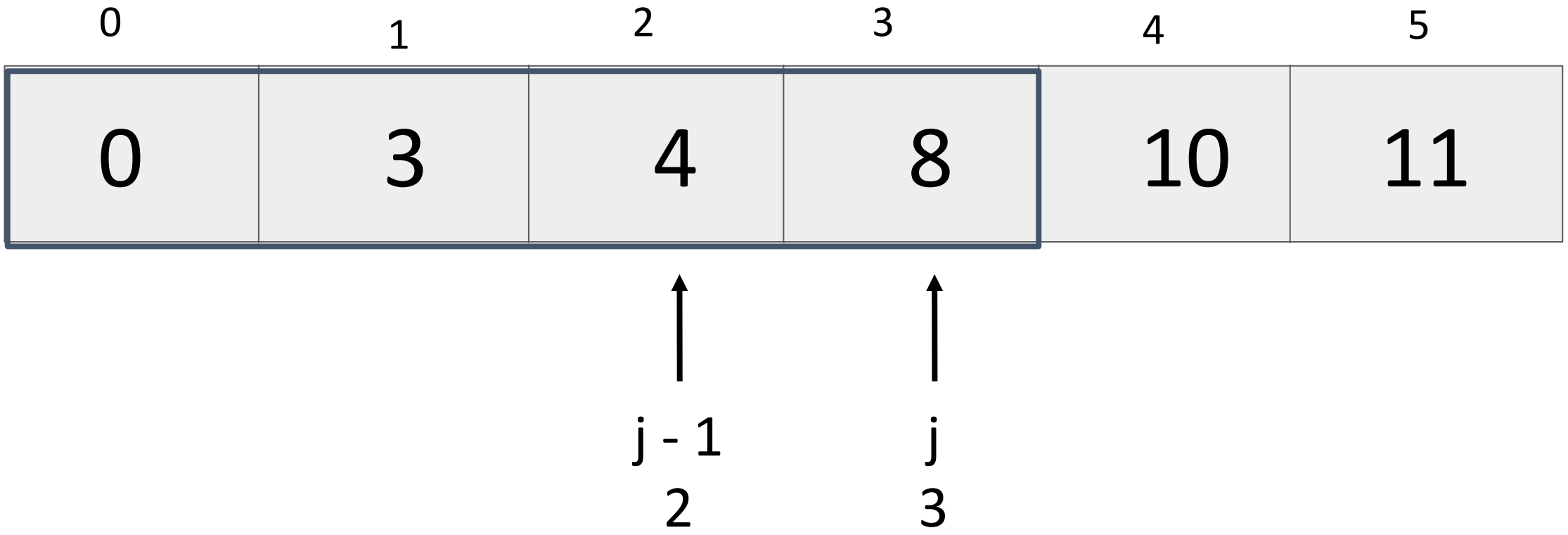


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 4

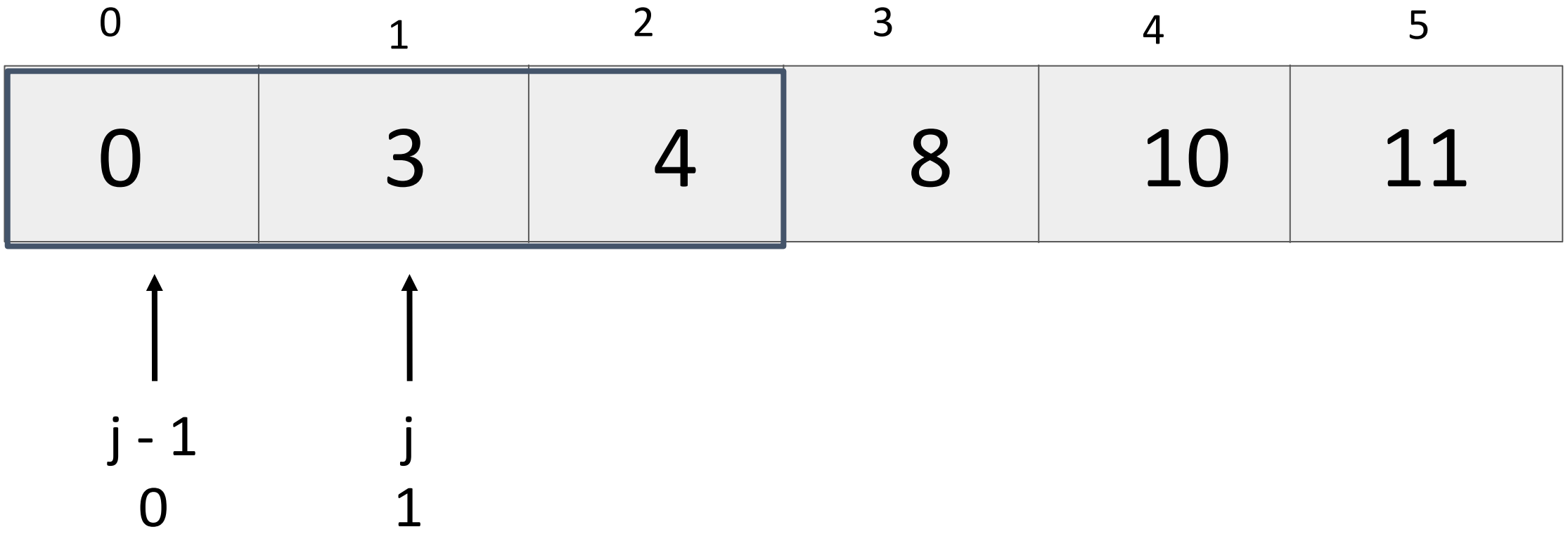


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 3

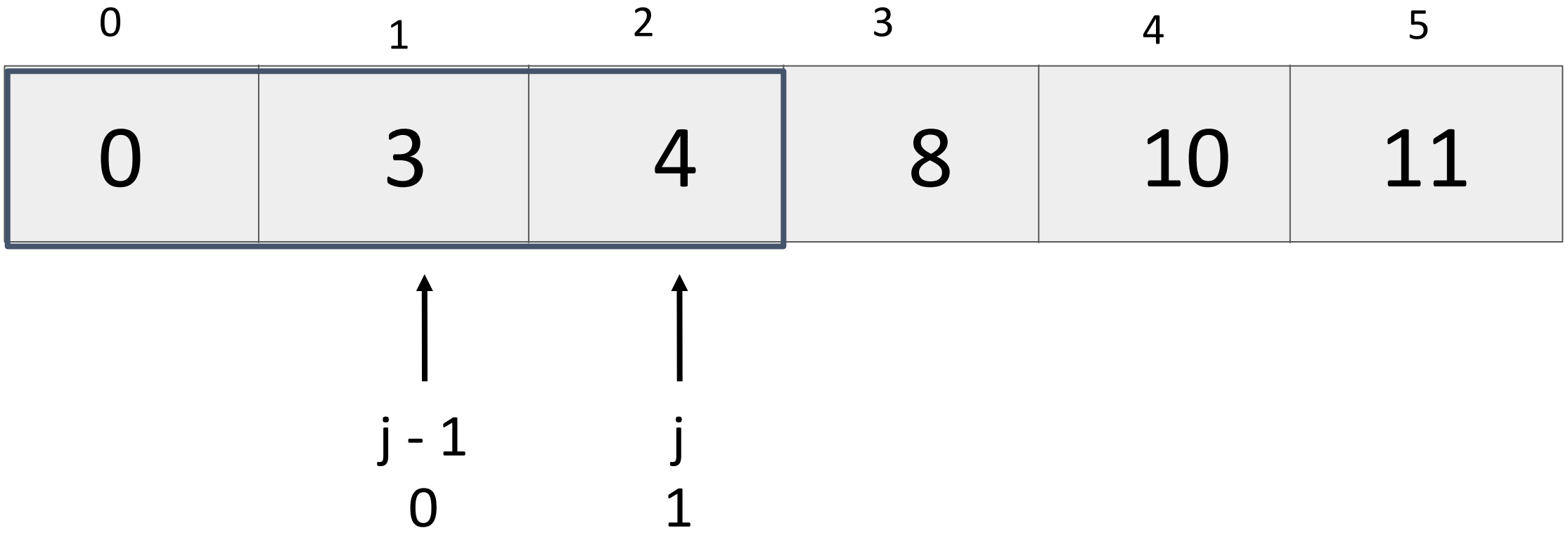


Reset; Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 3

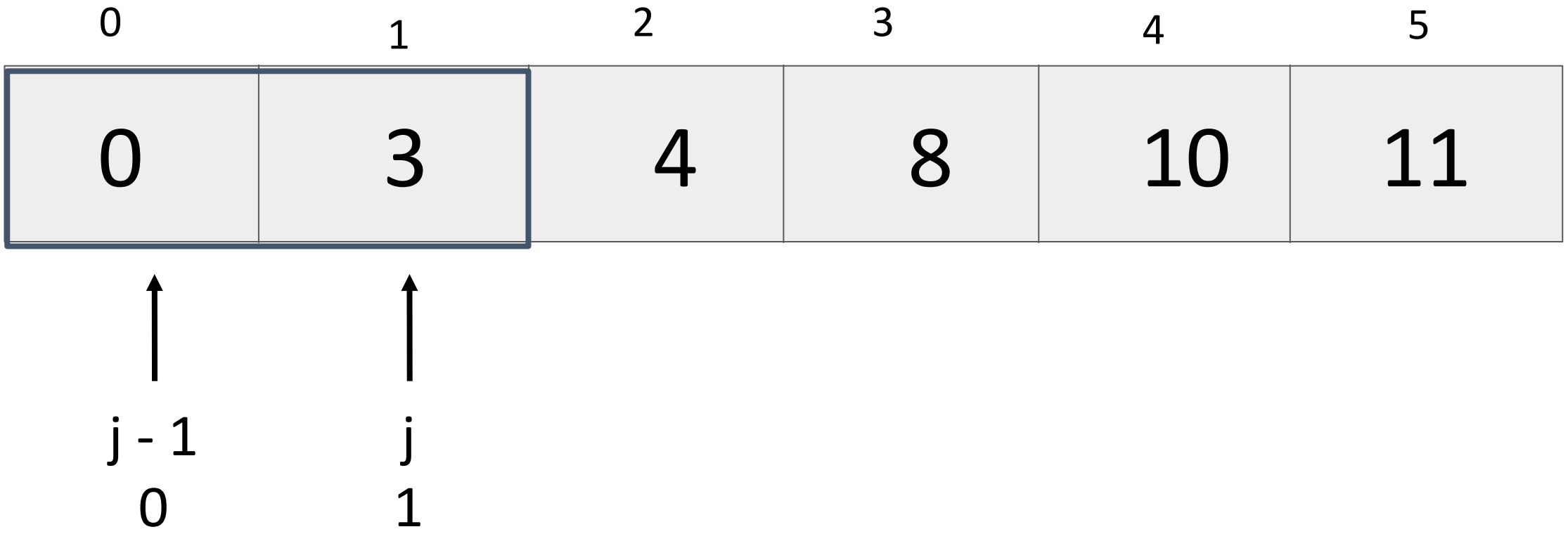


Reset; Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 2



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

What next?

Bubble Sort

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

```
bubbleSort(L):
        for len in range(len(L), 1, -1):
                for j in range(1, len): # bubble
                        if L[j-1] > L[j]:
                                swap(j-1, j, L)
```

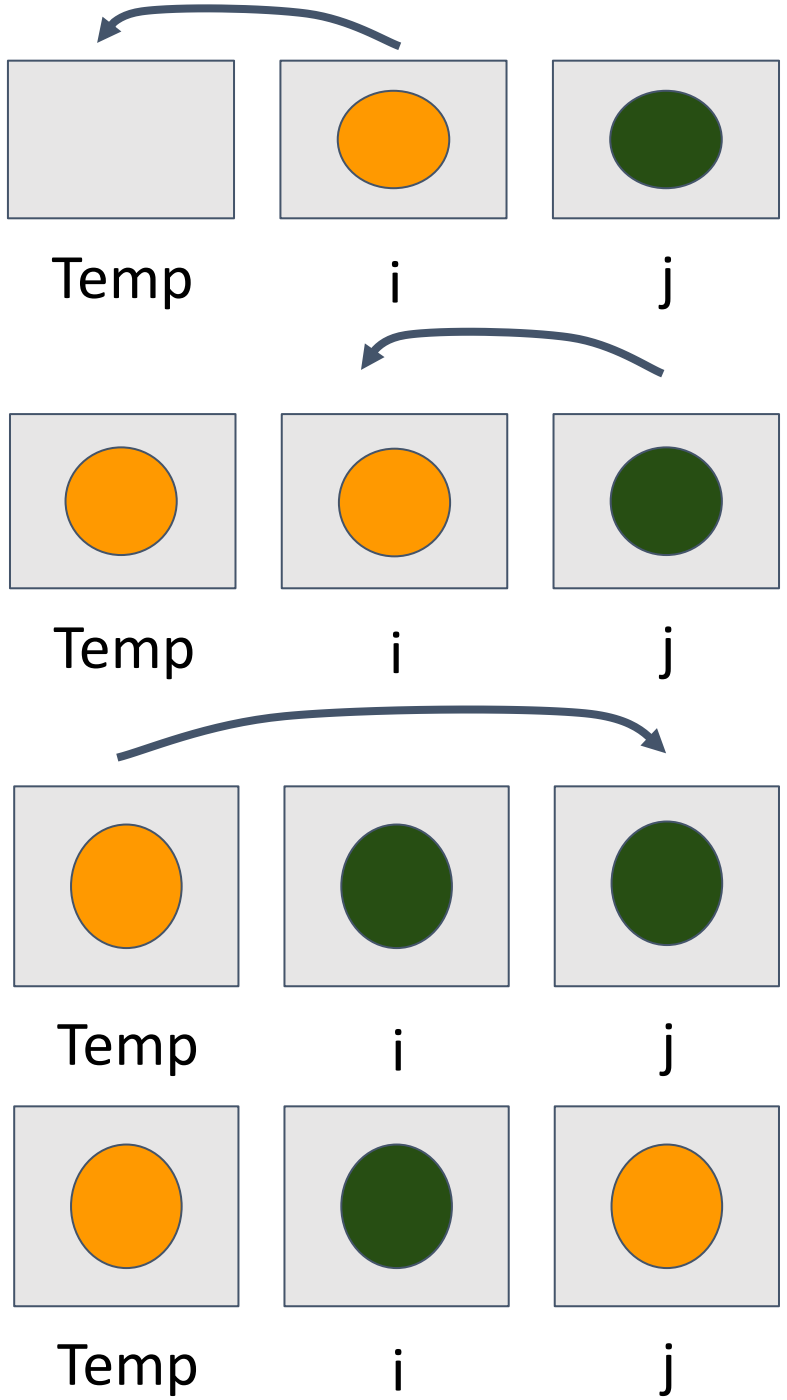
Bubble sort

swap(i, j, L):

```
    temp = L[i] # step 1

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

    L[j] = temp # step 3
```



Selection Sort

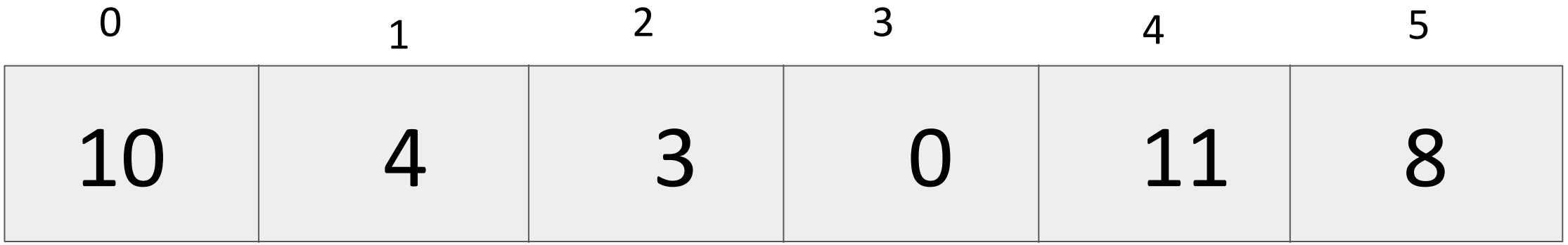
Not covering on Tuesday 04/04

Selection sort

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

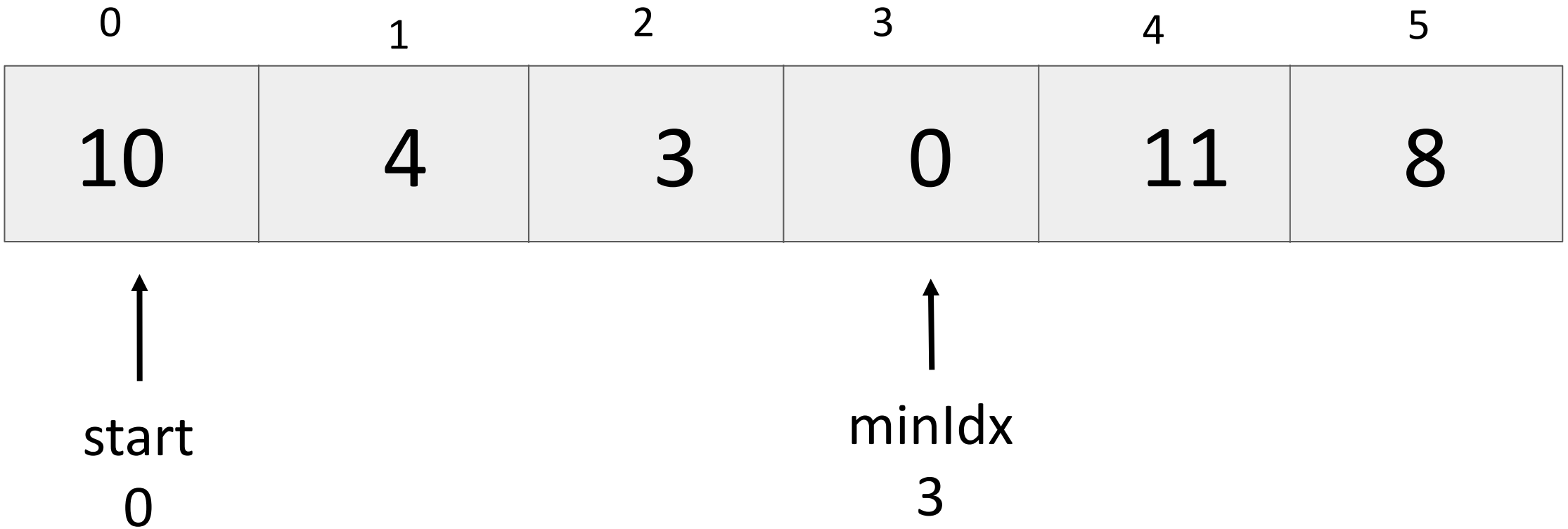
```
selectionSort(L):
    for startIdx in range(len(L)):
        minIdx = findMinimum(startIdx, L)
        swap(startIdx, minIdx, L)
```

Selection Sort



What do we do first?

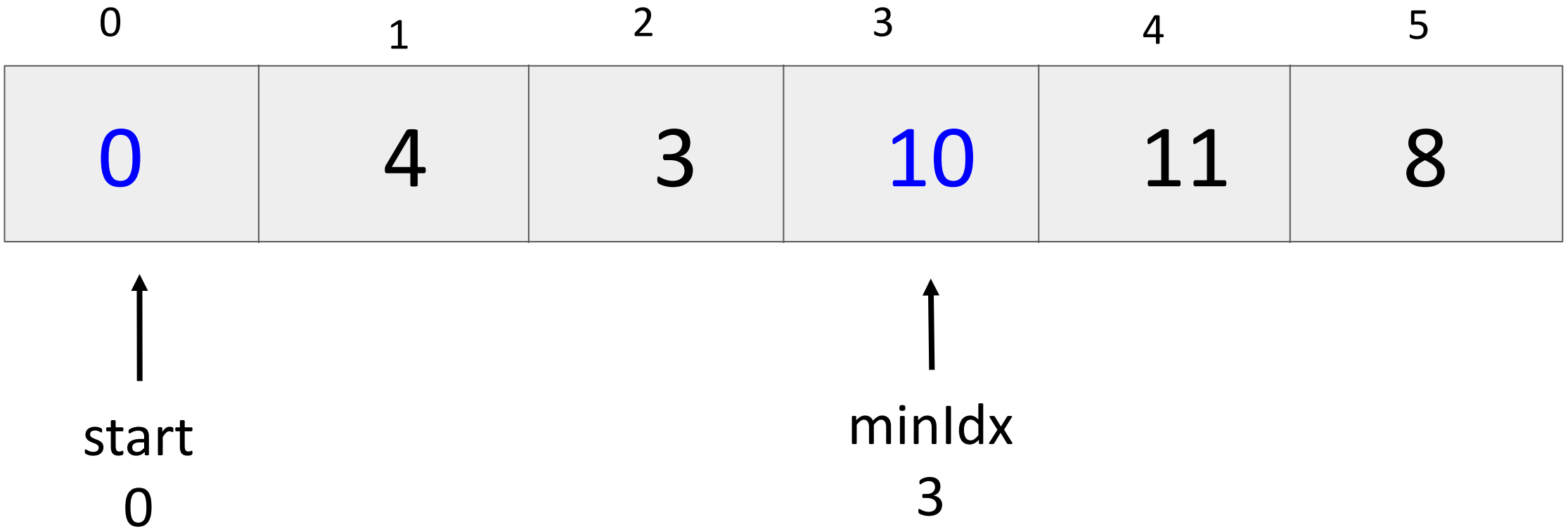
Selection Sort



Find minimum element idx between start to end

What next?

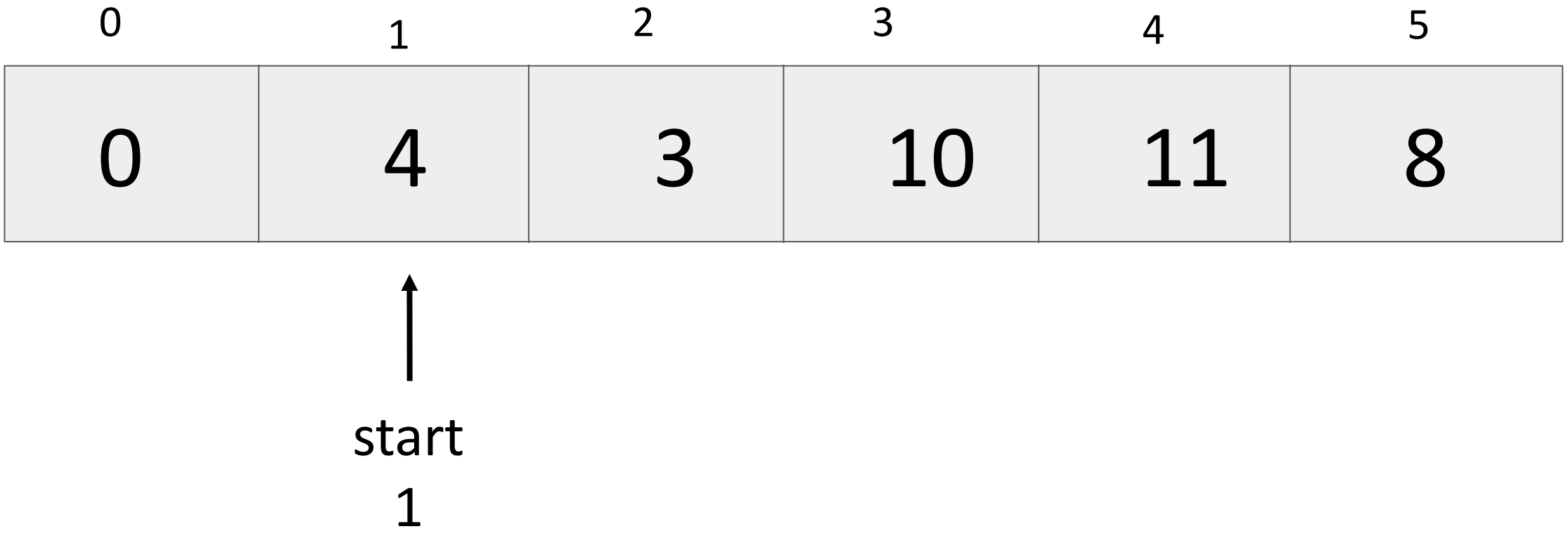
Selection Sort



Swap the elements at start and minIdx

What next?

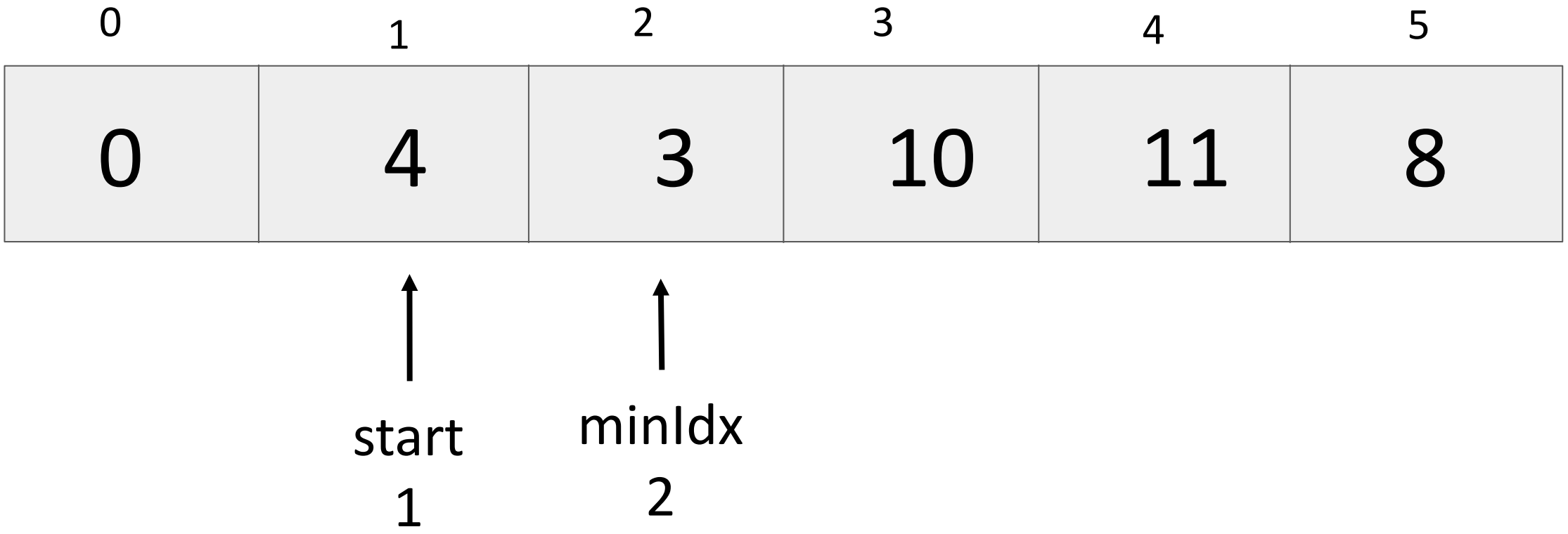
Selection Sort



Decrease the interval.

What next?

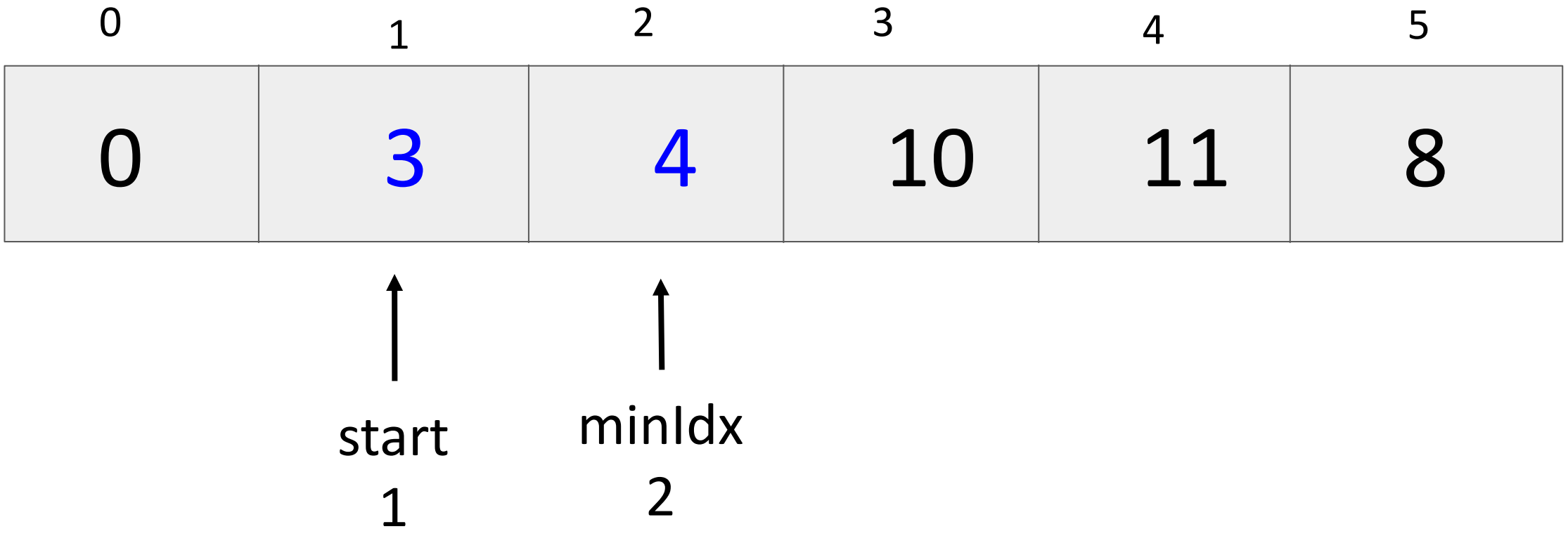
Selection Sort



Find minimum element between start to end

What next?

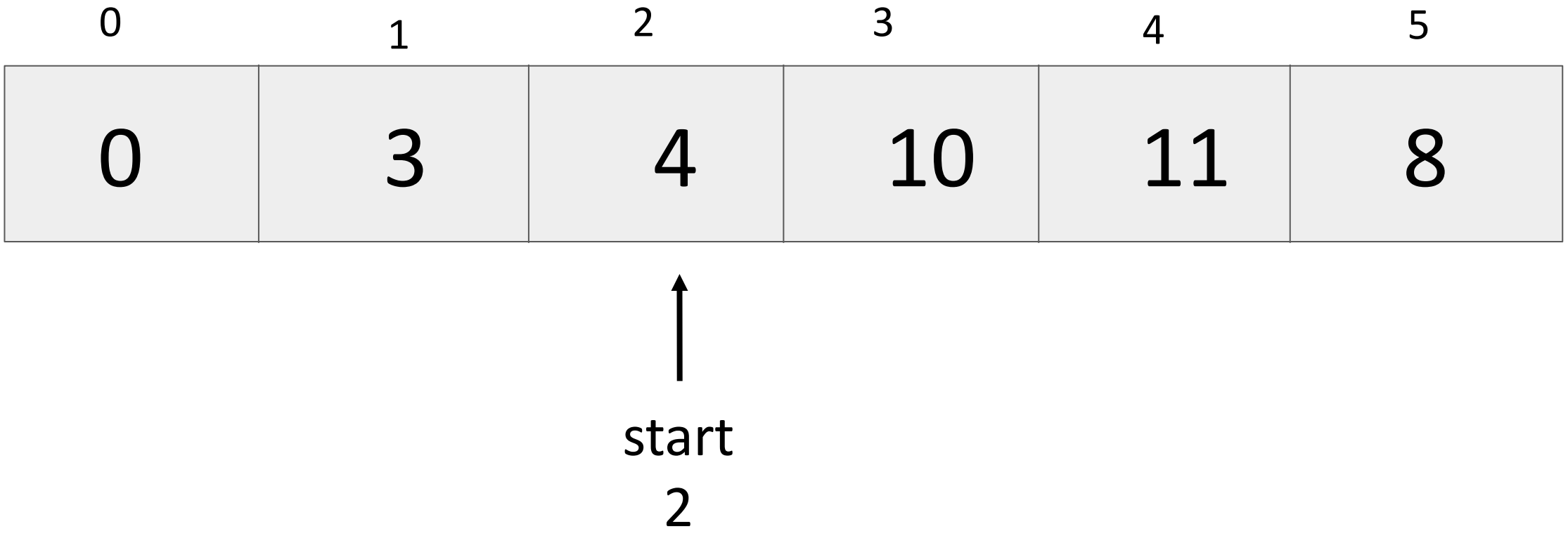
Selection Sort



Swap the elements at start and minIdx

What next?

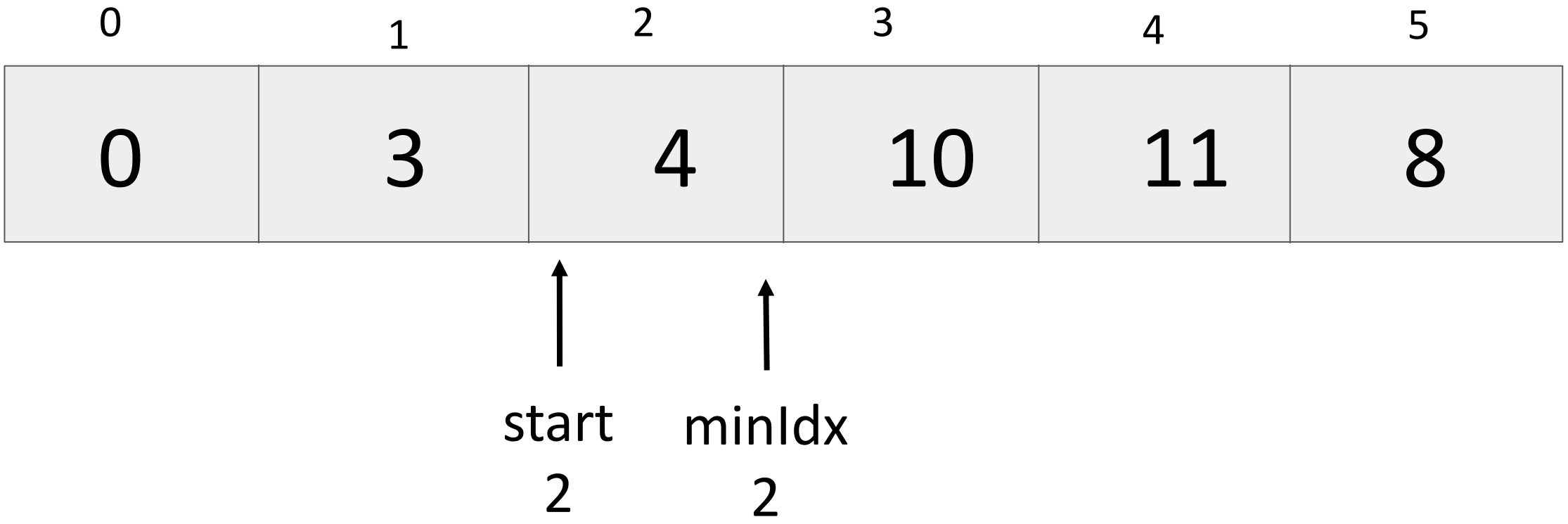
Selection Sort



Decrease the interval.

What next?

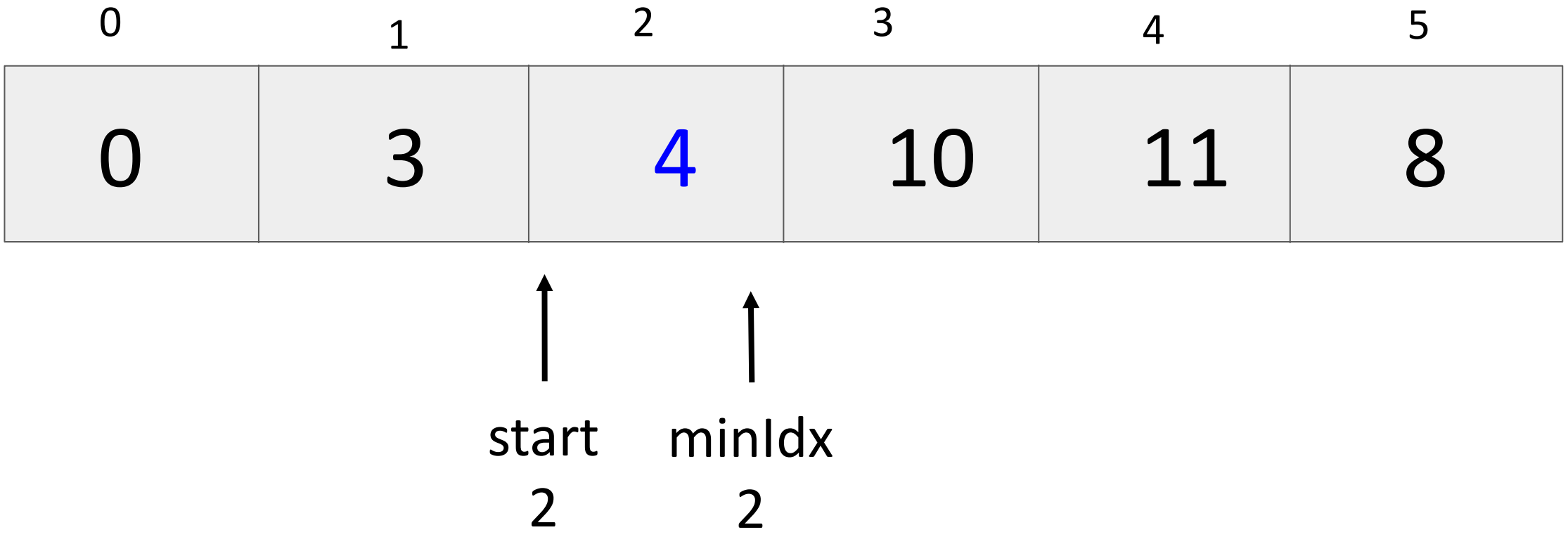
Selection Sort



Find minimum element idx between start to end

What next?

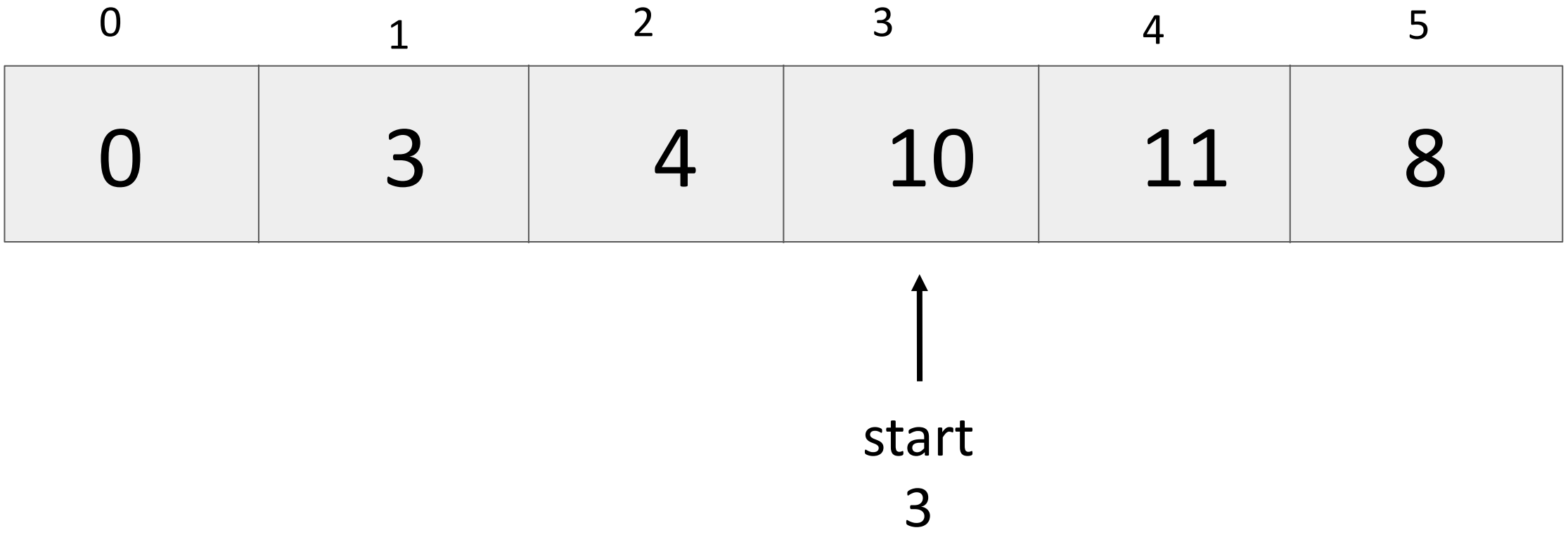
Selection Sort



Swap the elements at start and minIdx

What next?

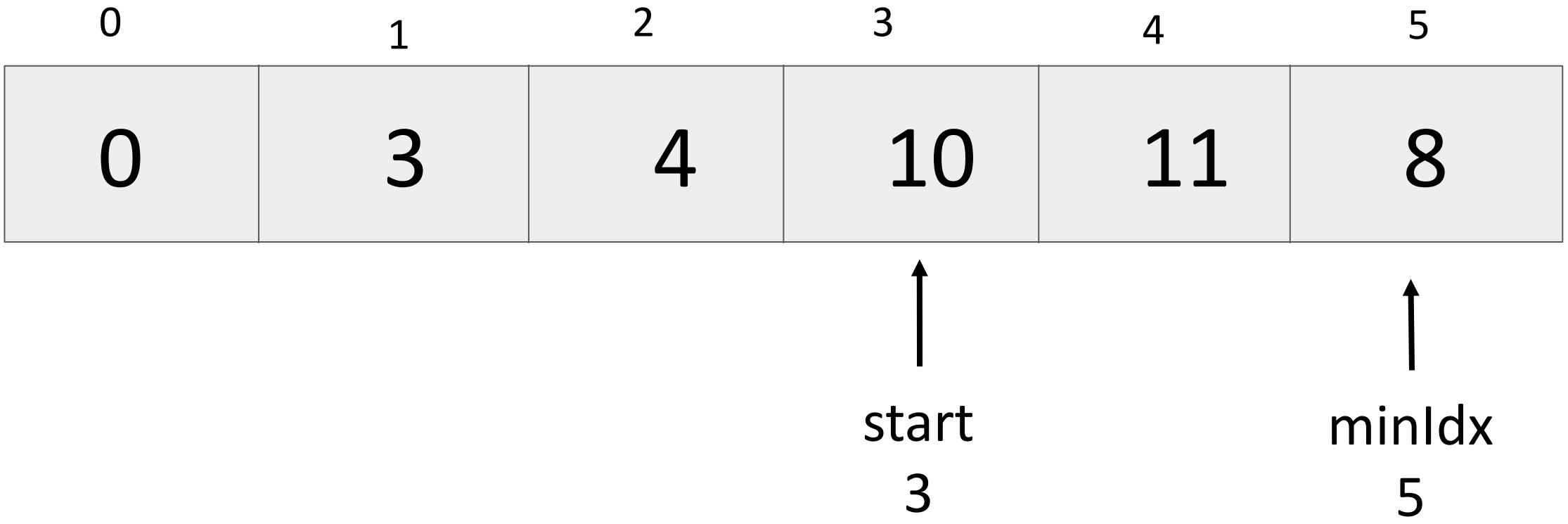
Selection Sort



Decrease the interval.

What next?

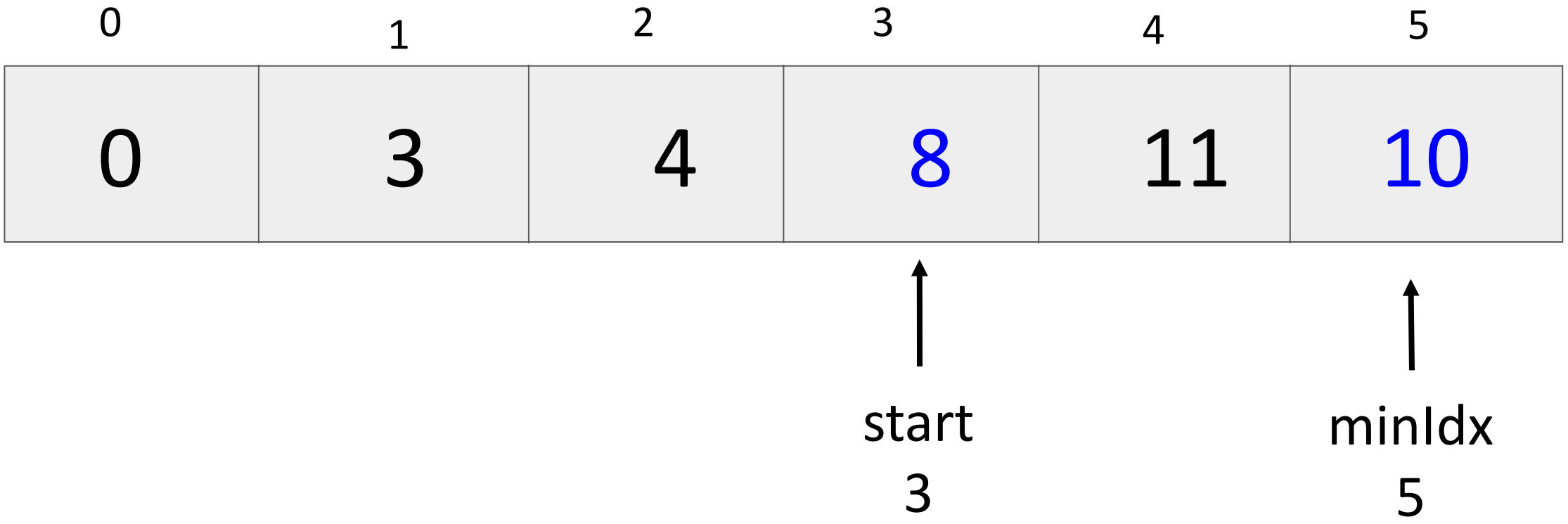
Selection Sort



Find minimum element idx between start to end

What next?

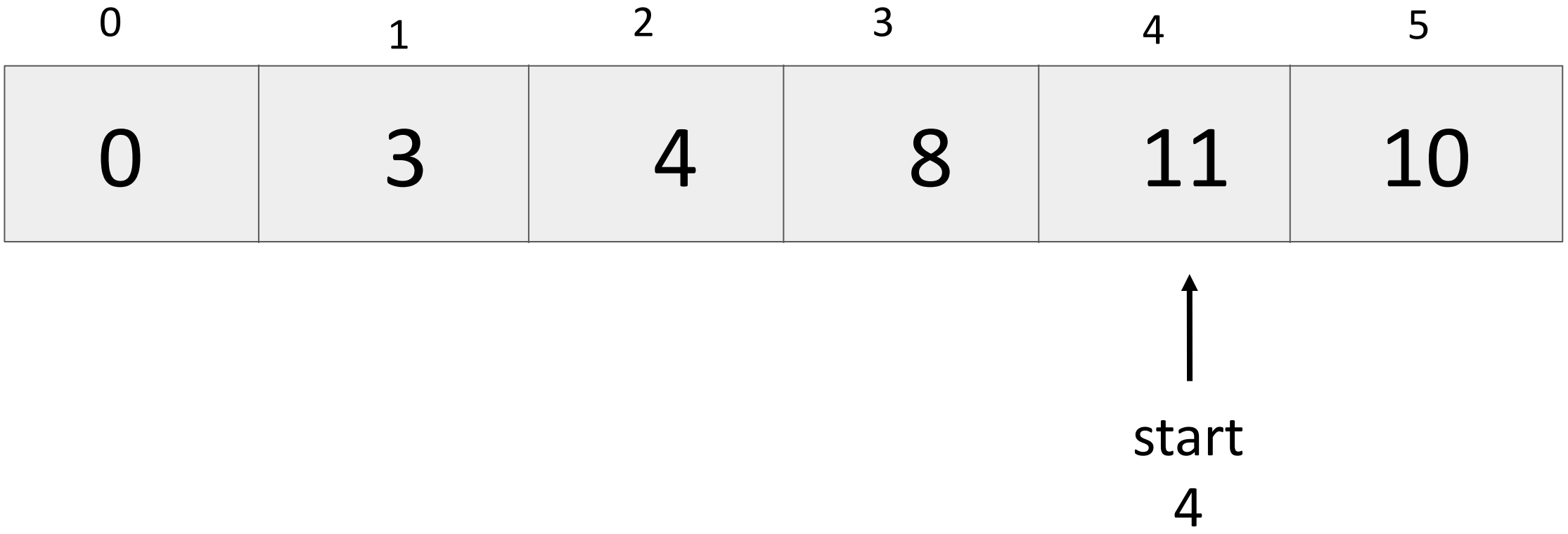
Selection Sort



Swap the elements at start and minIdx

What next?

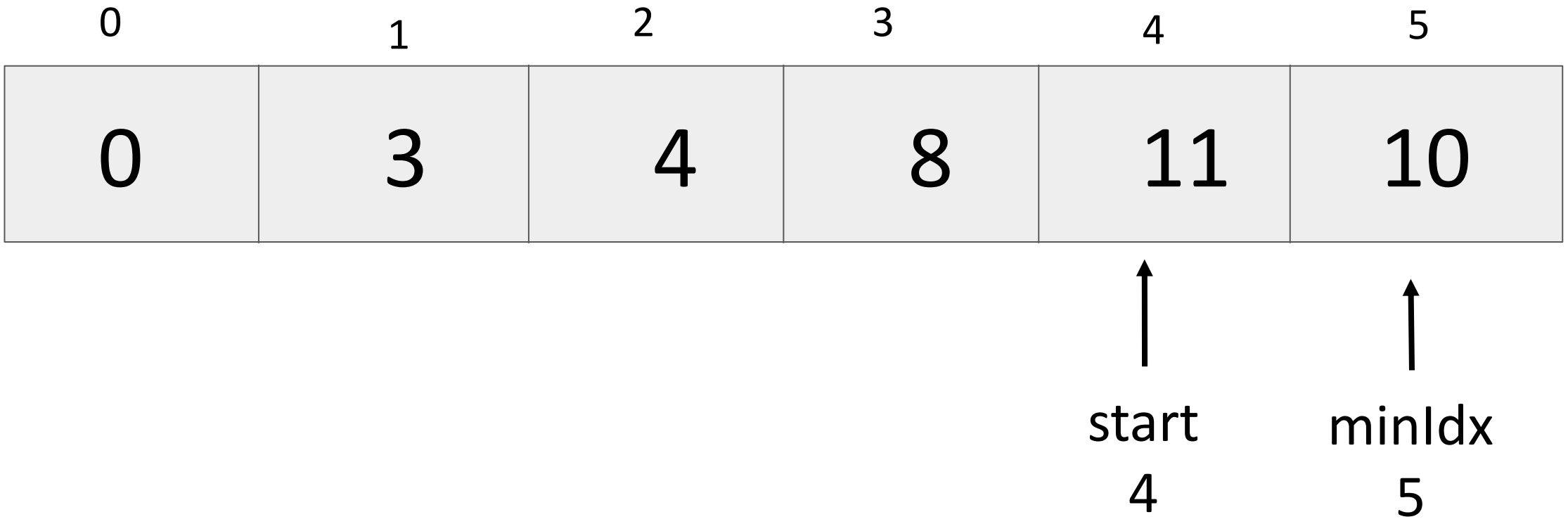
Selection Sort



Decrease the interval.

What next?

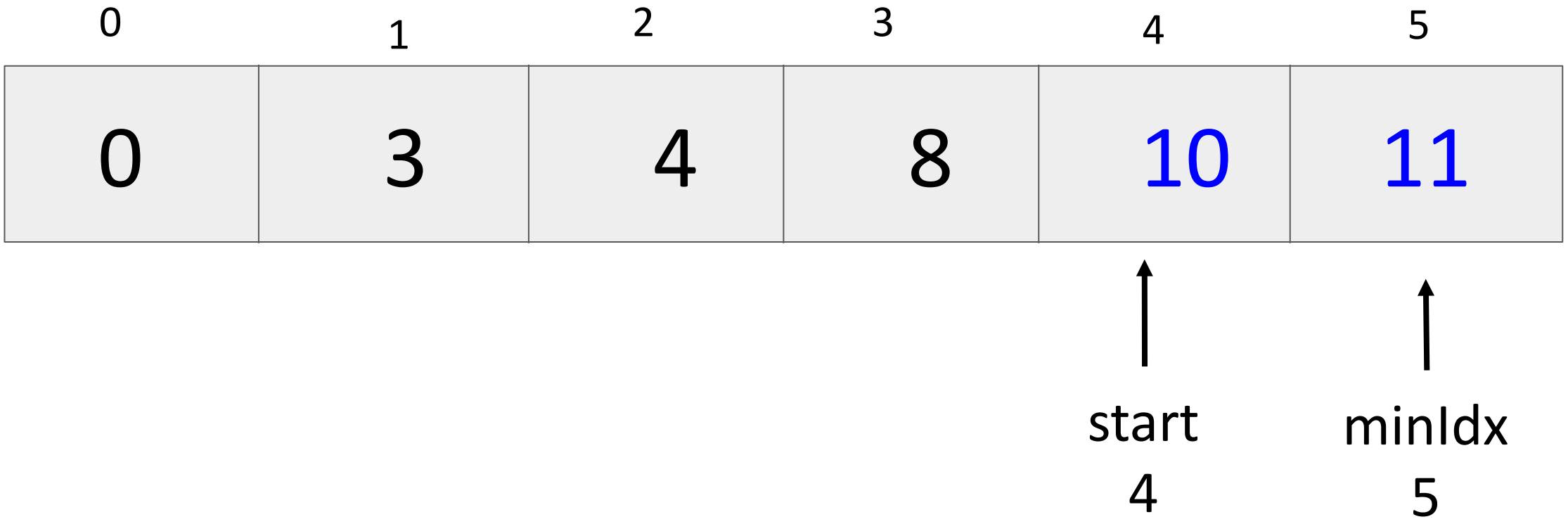
Selection Sort



Find minimum element idx between start to end

What next?

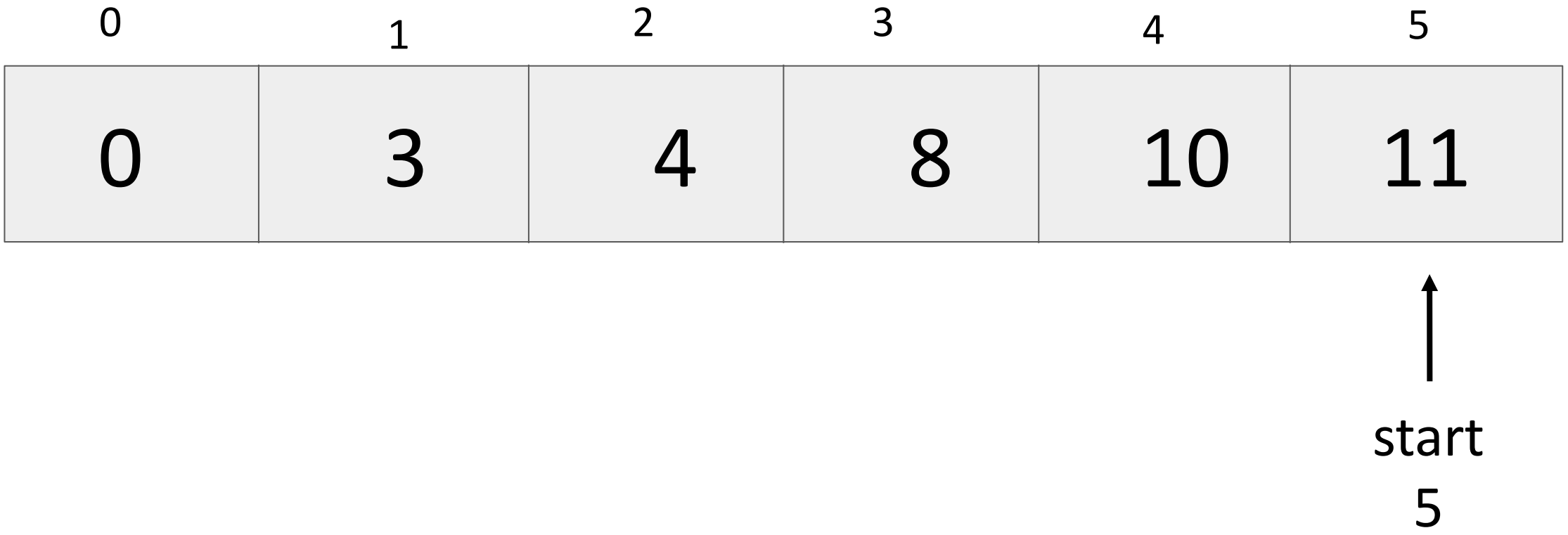
Selection Sort



Swap the elements at start and minIdx

What next?

Selection Sort



Decrease the interval.

We’re done!

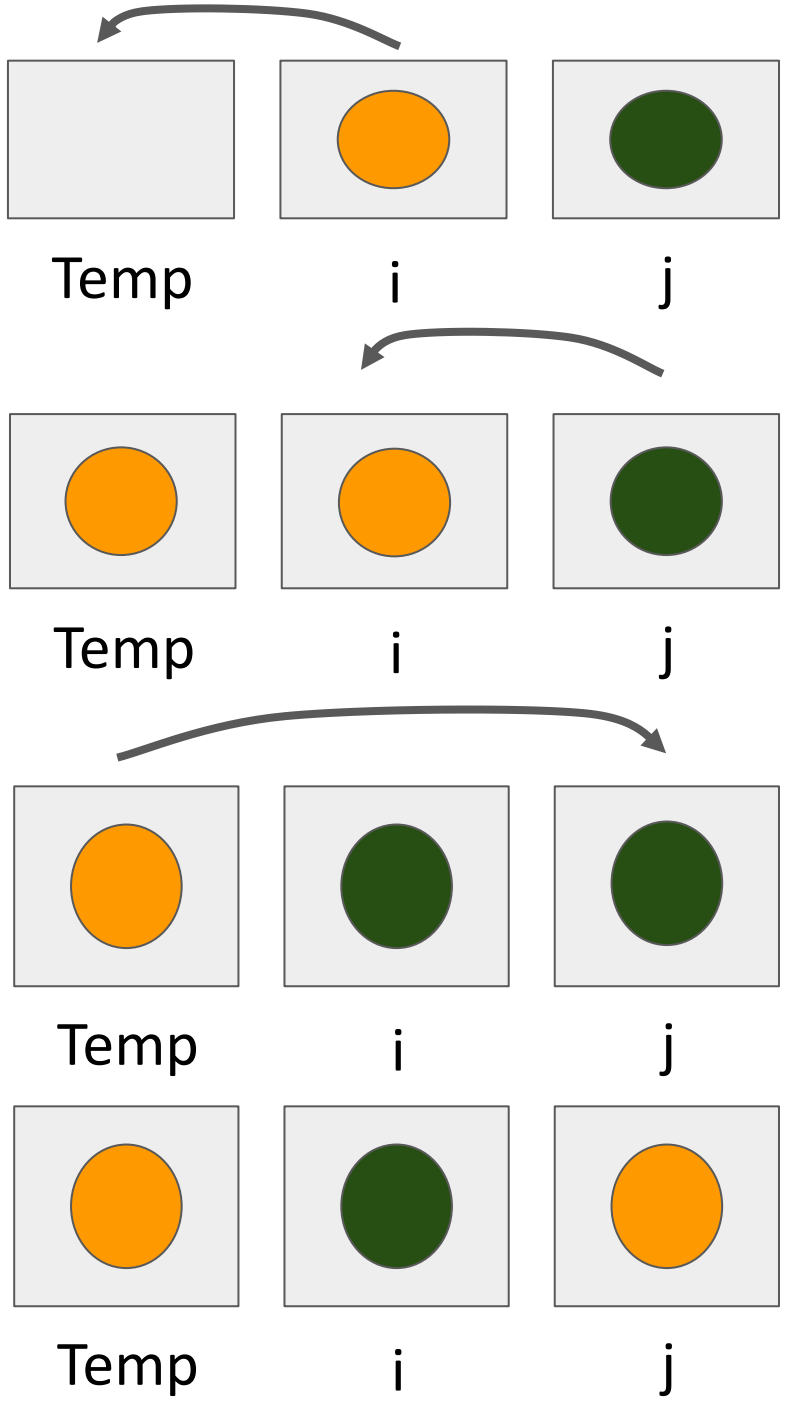
Selection sort

```
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 $O(N^2)$