

# CS 113 – Computer Science I

## Lecture 20 – Binary Search & Sorting

Thursday 11/21/2024

# Announcements

HW 10 – Due Monday 11/25

Object Oriented Programming:  
Inheritance & Interface

Mid-semester feedback form: <https://forms.gle/Ed7G9oe74QQBT5sy5>

Midterm 2: Thursday December 5<sup>th</sup>

Final: Wednesday 12/18 9:30am-12:30pm Park 238

# 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

# 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

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

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

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

↑  
j - 1  
0

↑  
j  
0

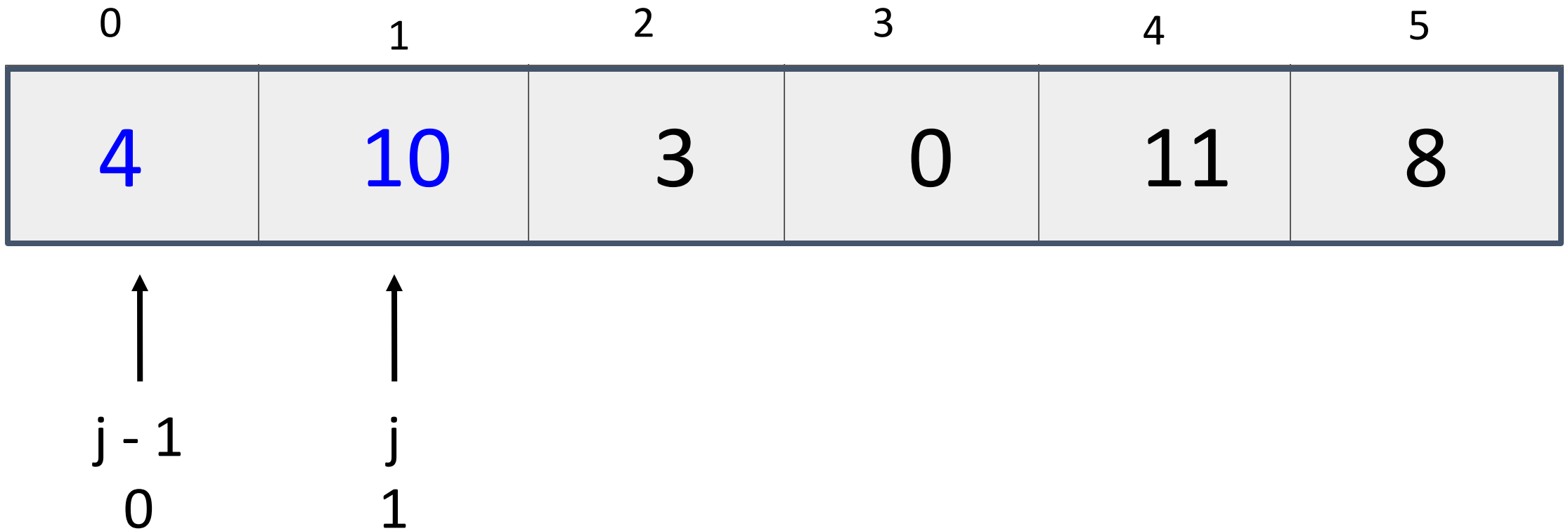
Compare j-1 and j; Swap if  $L[j-1] > L[j]$

What next? <sup>24</sup>



# Bubble Sort

len = 6



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

What next? <sup>25</sup>

# Bubble Sort

len = 6

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



$j - 1$   
1



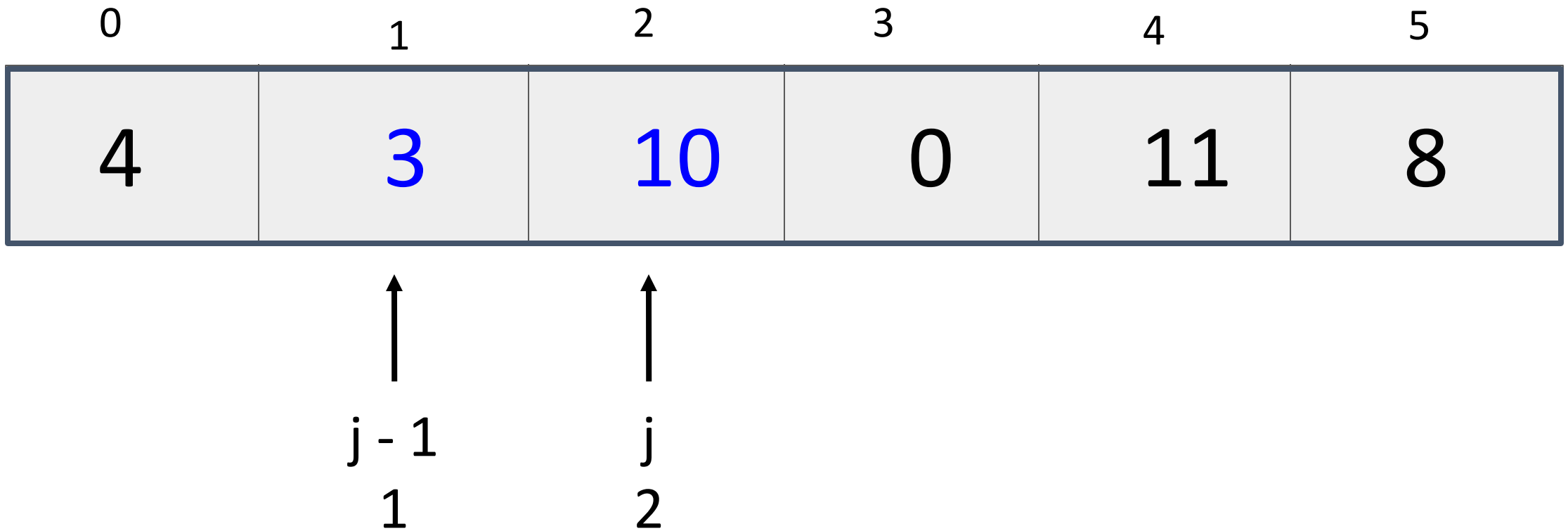
$j$   
2

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

What next? <sup>26</sup>

# Bubble Sort

len = 6



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

What next? <sup>27</sup>

# Bubble Sort

len = 6

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

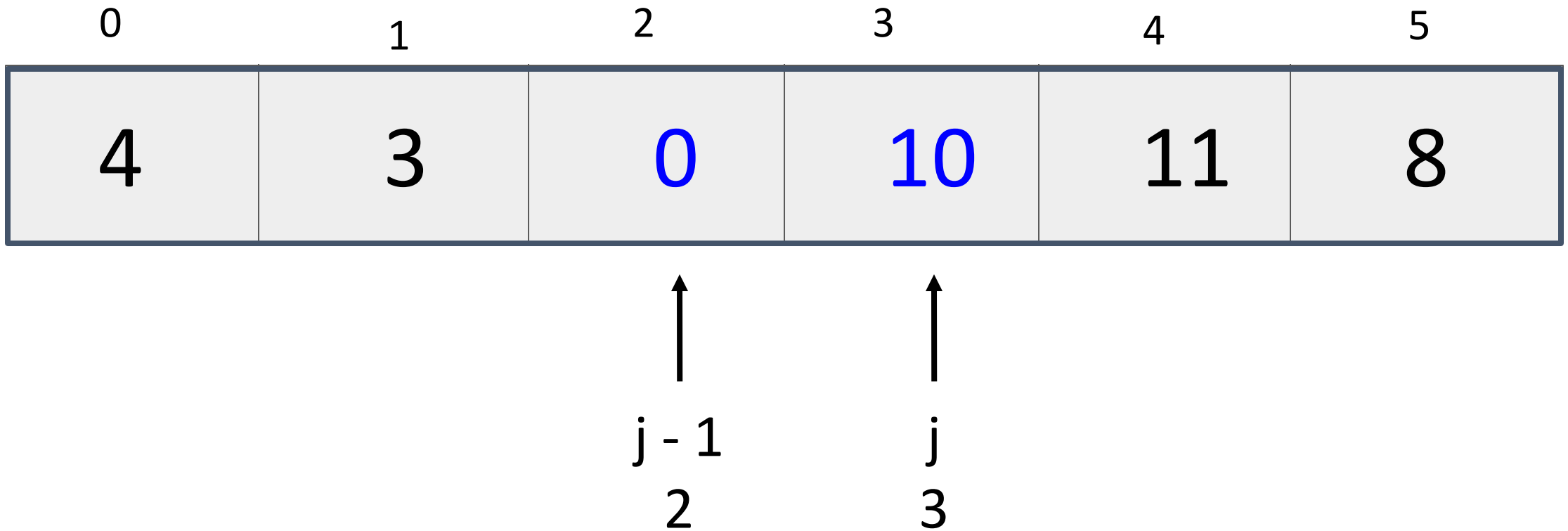
↑      ↑  
j - 1    j  
2        3

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

What next? 28

# Bubble Sort

len = 6

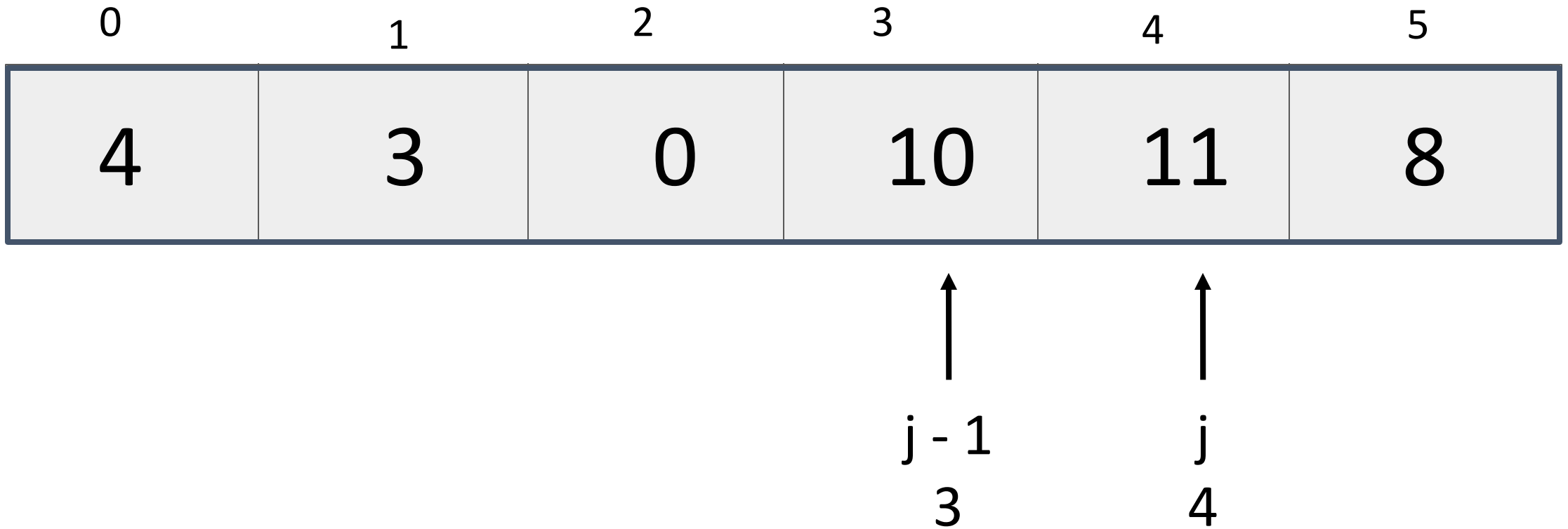


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>29</sup>

# Bubble Sort

len = 6

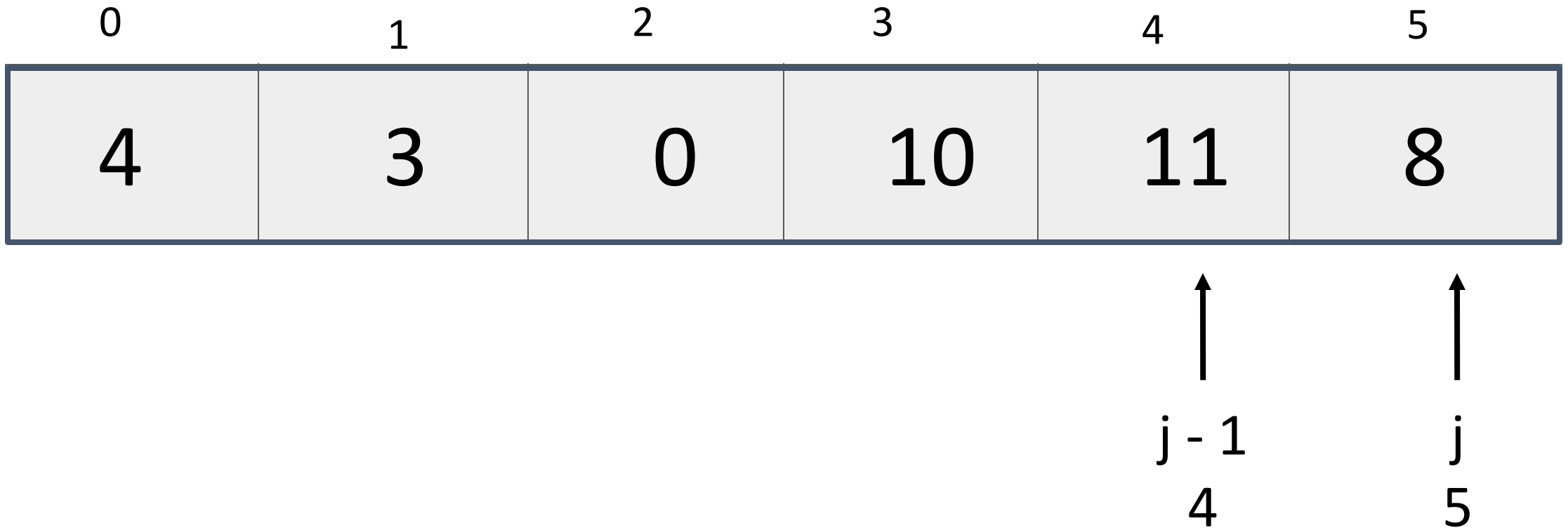


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>30</sup>

# Bubble Sort

len = 6

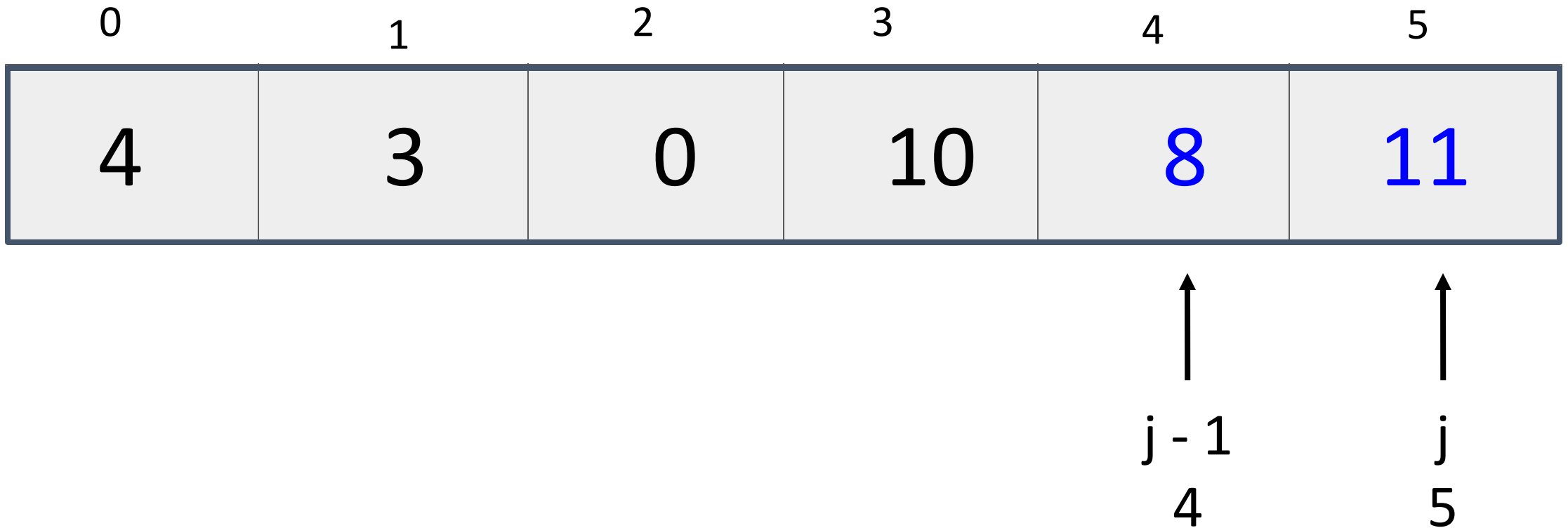


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>31</sup>

# Bubble Sort

len = 6



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

What next? <sup>32</sup>



# 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? <sup>33</sup>

# Bubble Sort

len = 5

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

↑  
j - 1  
0

↑  
j  
1

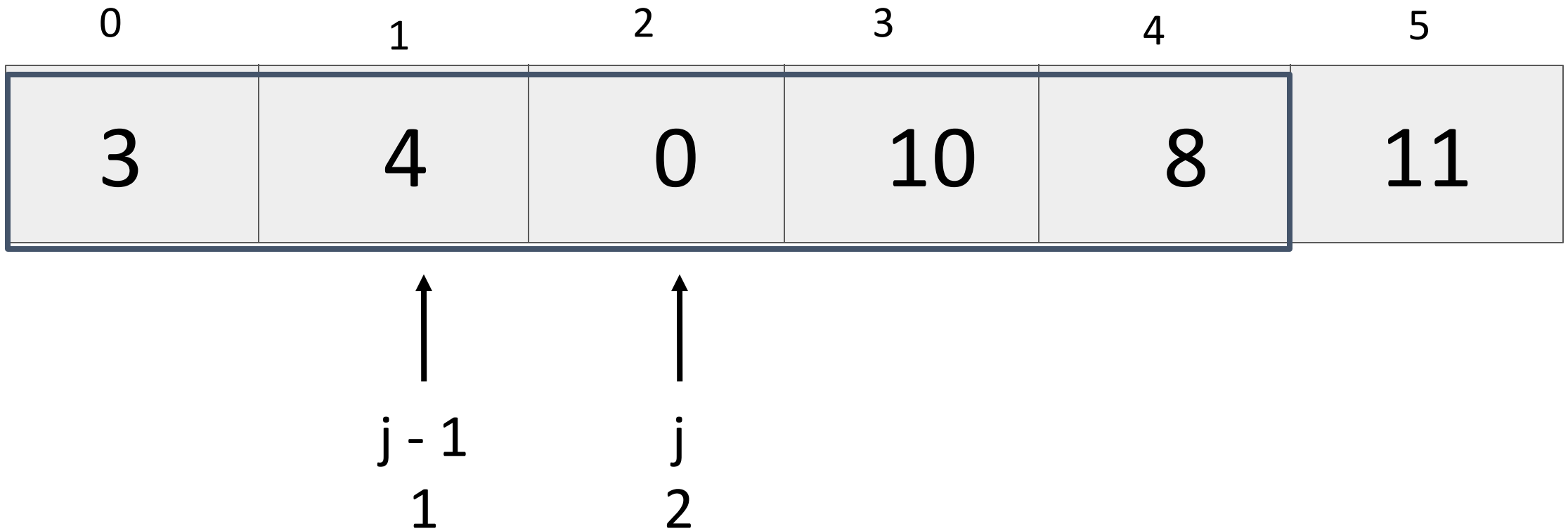
Last element has  
largest element!

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

What next? <sup>34</sup>

# Bubble Sort

len = 5

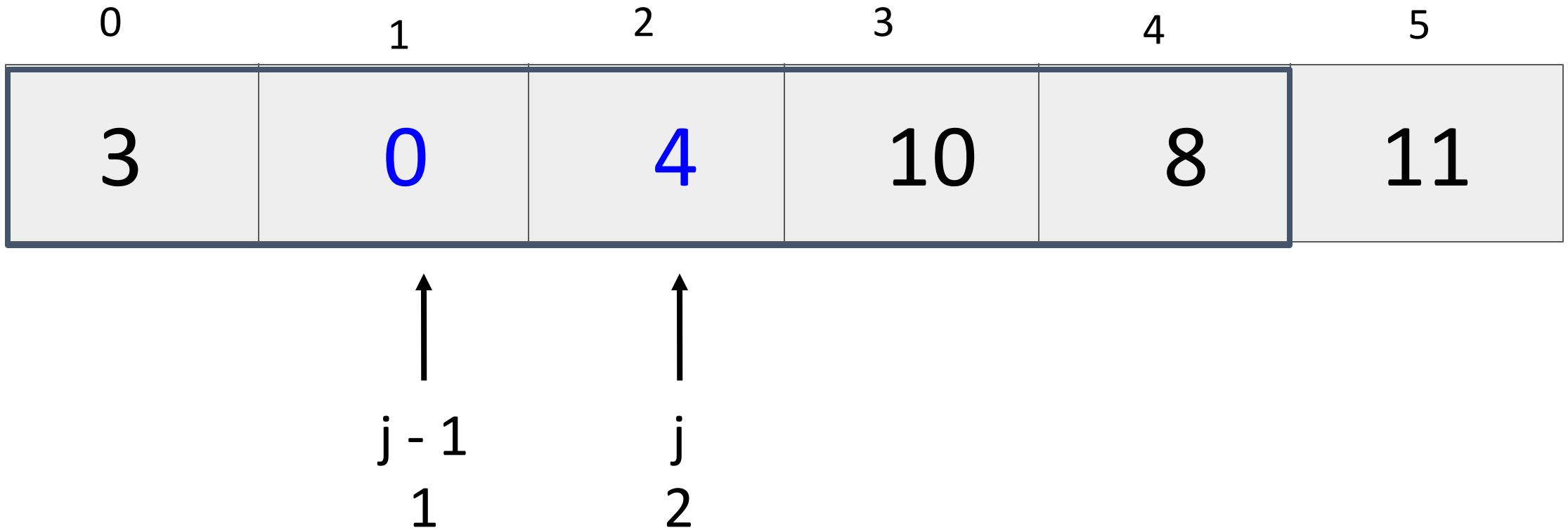


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>35</sup>

# Bubble Sort

len = 5

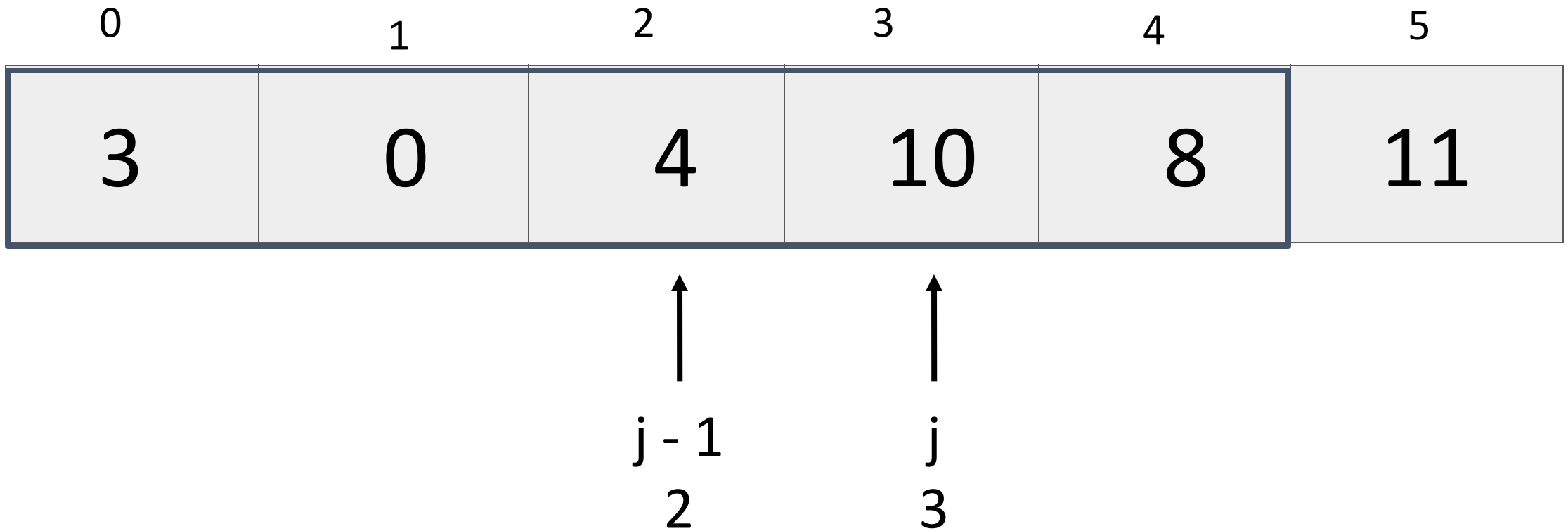


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>36</sup>

# Bubble Sort

len = 5

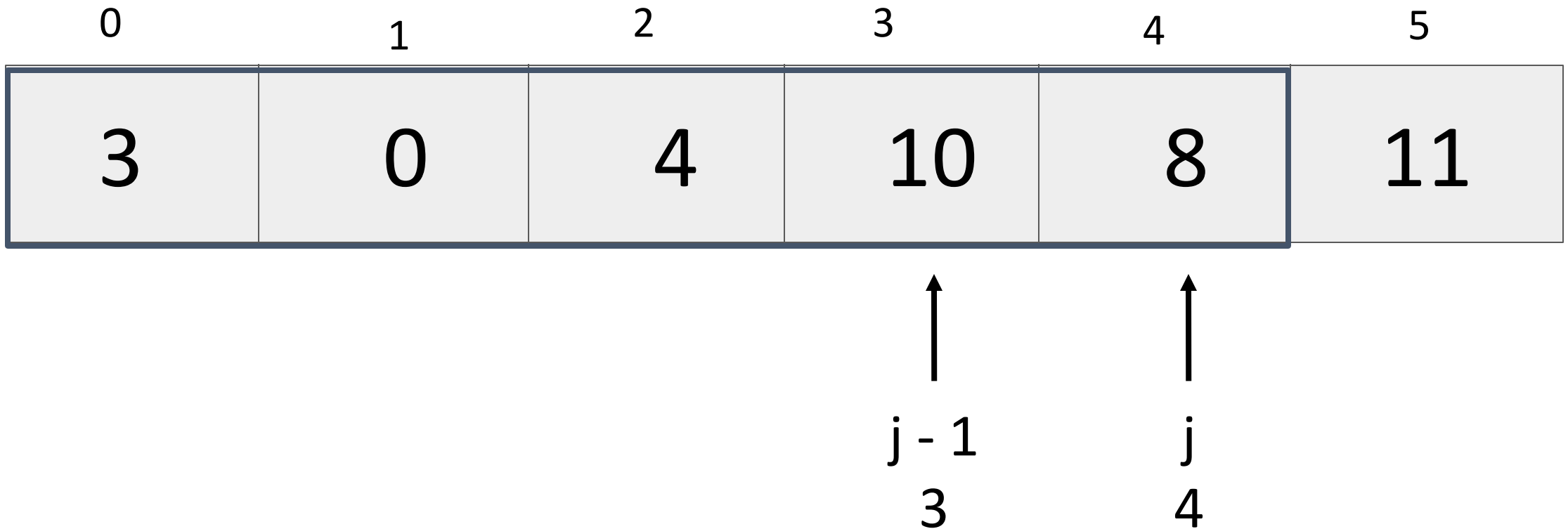


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>37</sup>

# Bubble Sort

len = 5

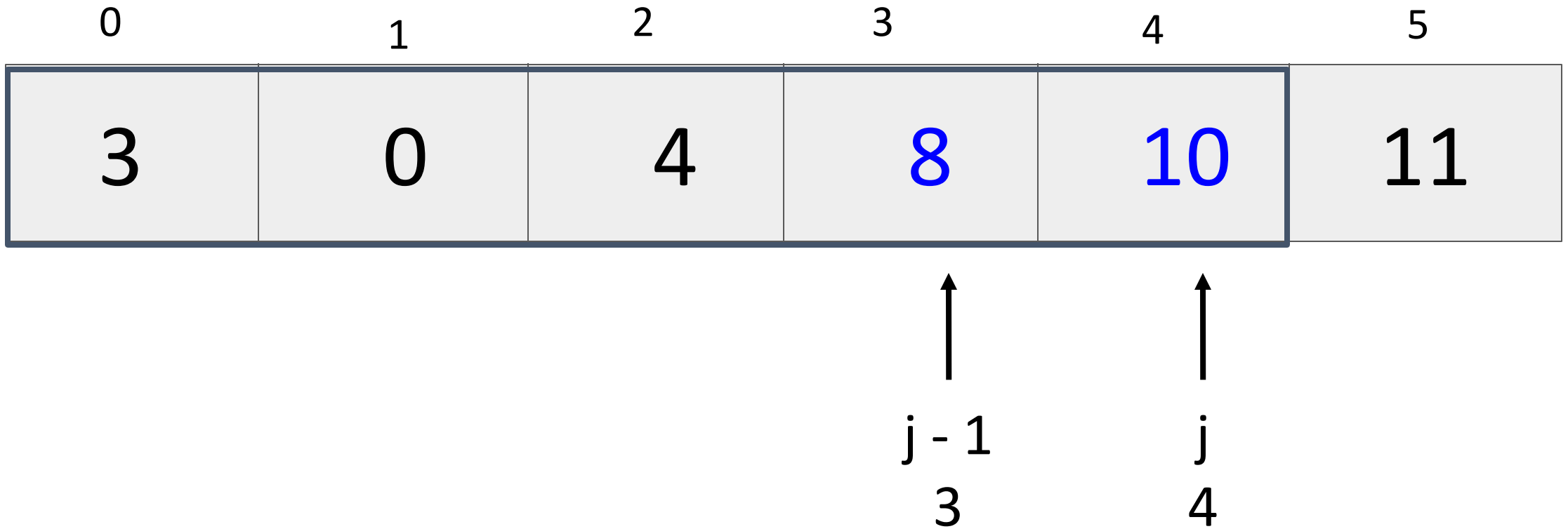


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>38</sup>

# Bubble Sort

len = 5

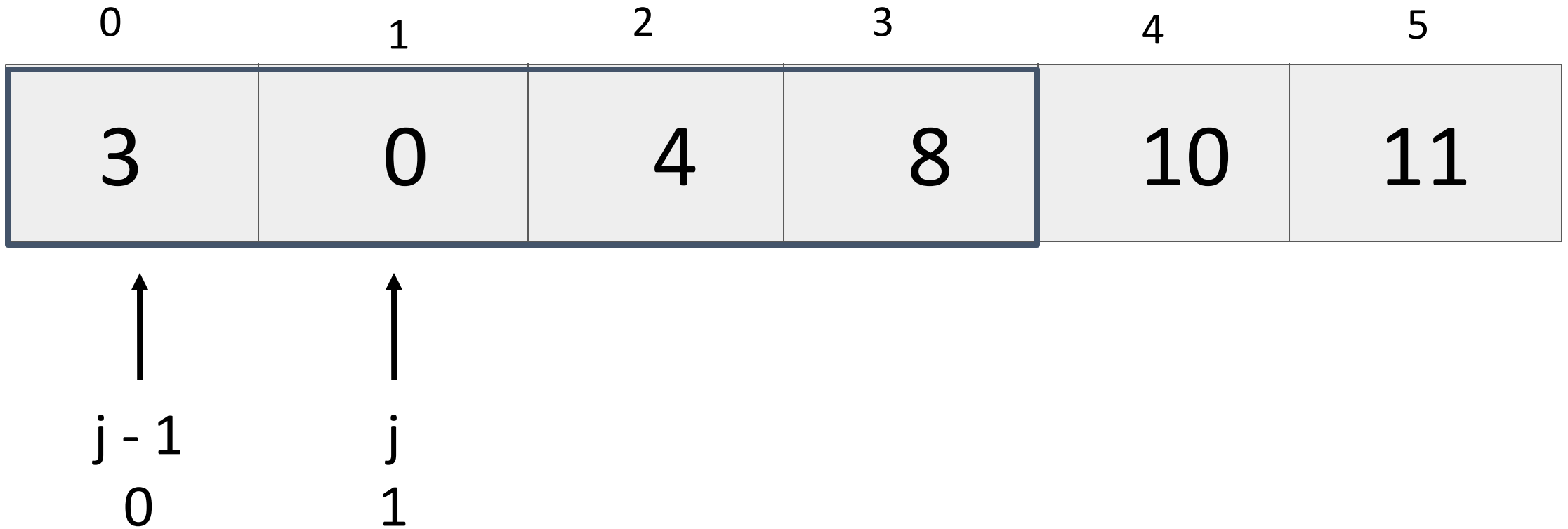


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>39</sup>

# Bubble Sort

len = 4



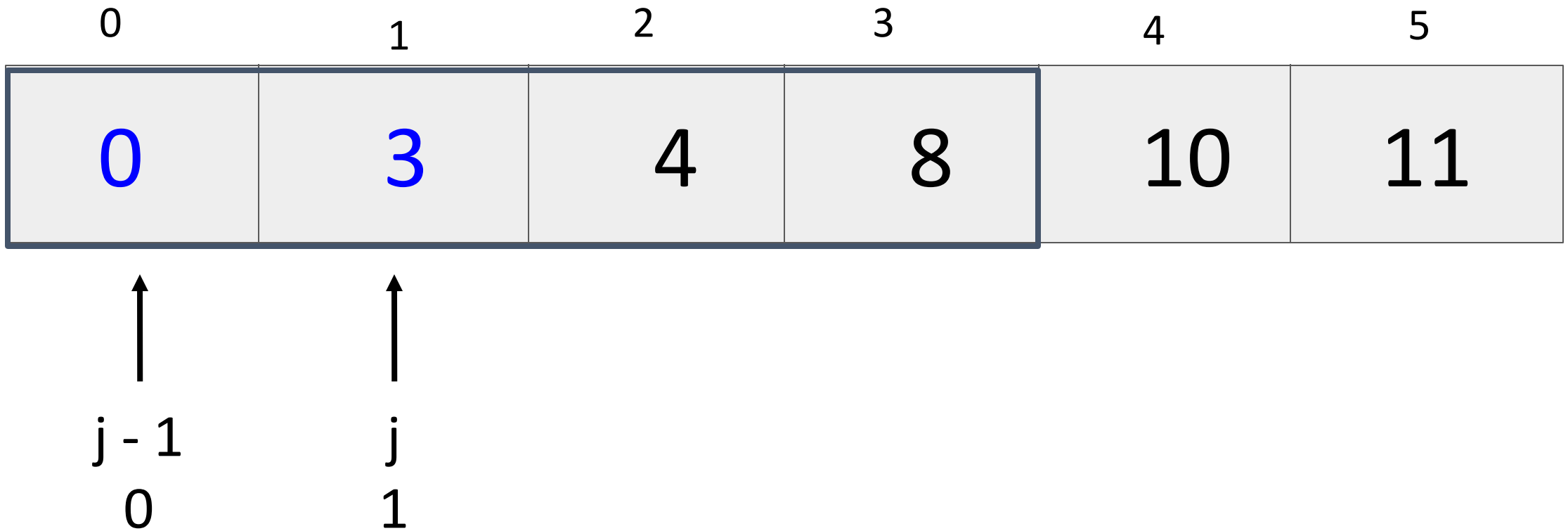
Reset and check pairs with shorter list

What next? 40



# Bubble Sort

len = 4

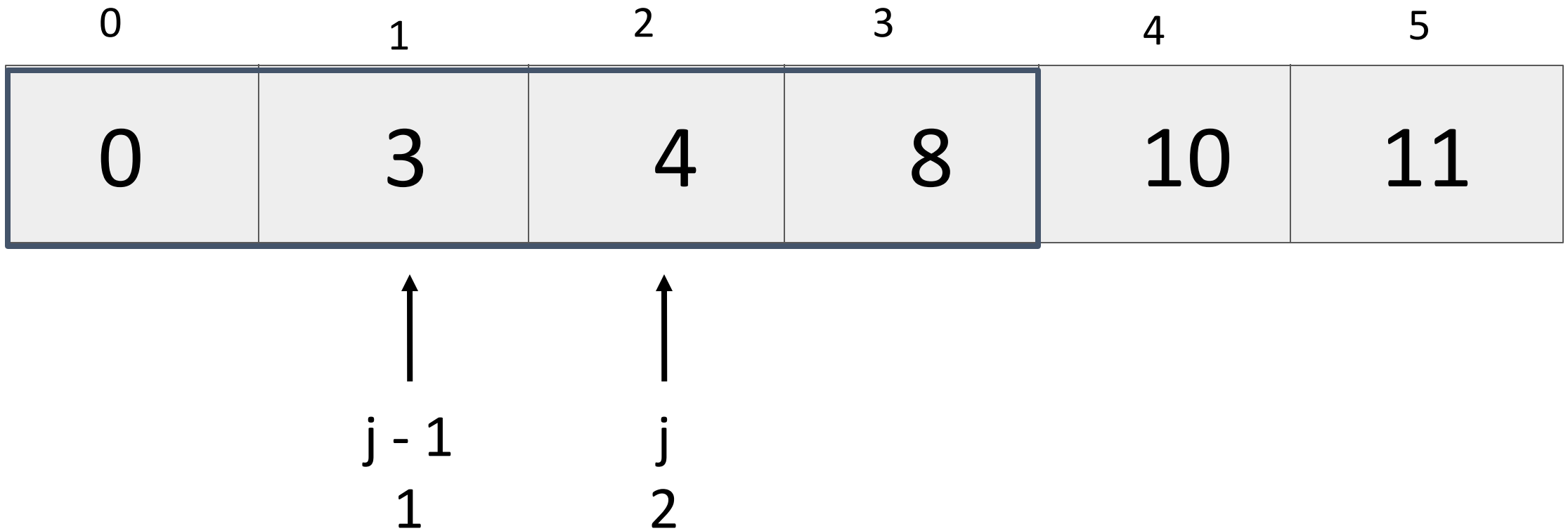


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>41</sup>

# Bubble Sort

len = 4

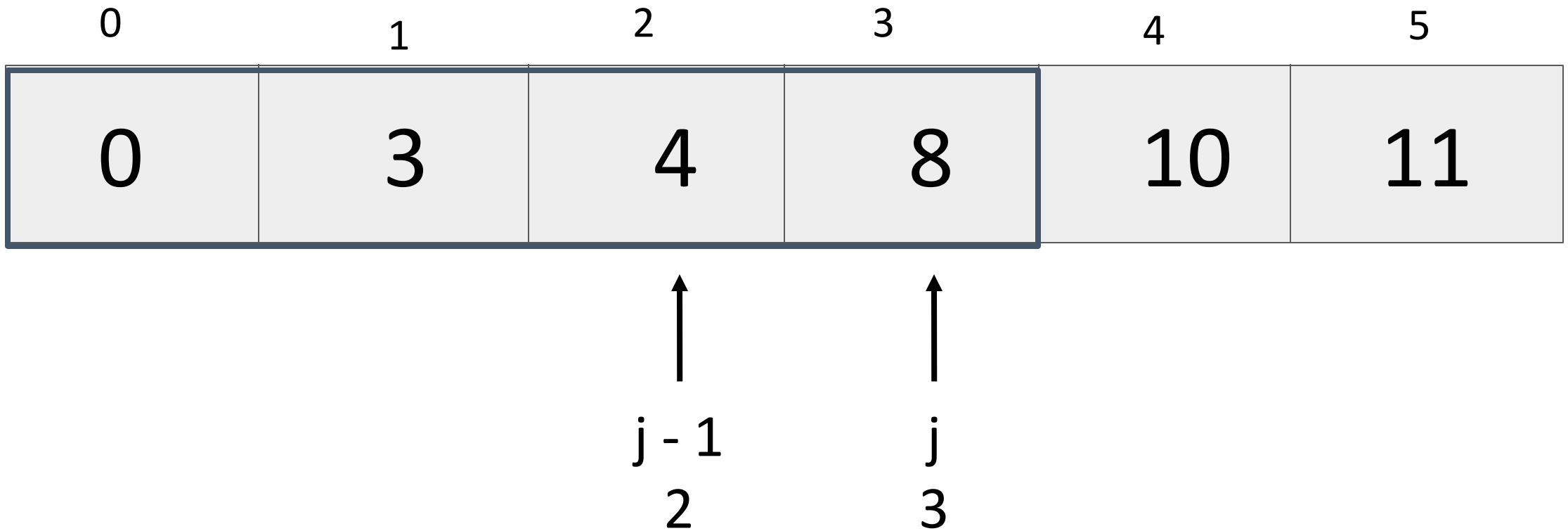


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>42</sup>

# Bubble Sort

len = 4

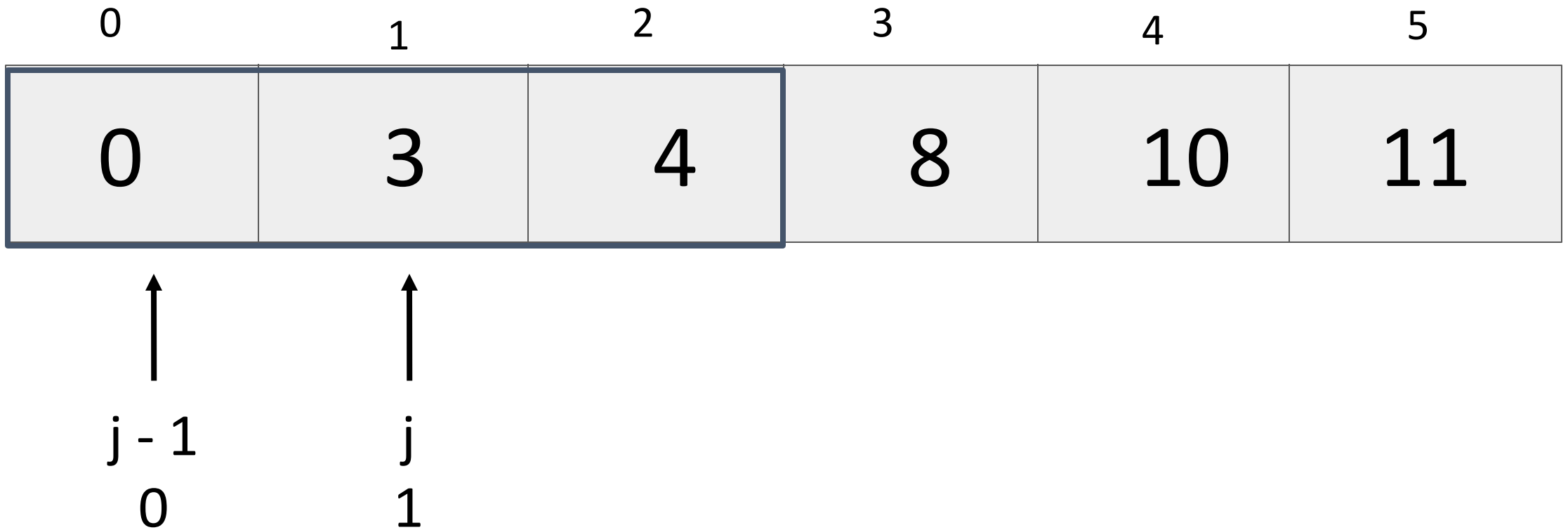


Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>43</sup>

# Bubble Sort

len = 3

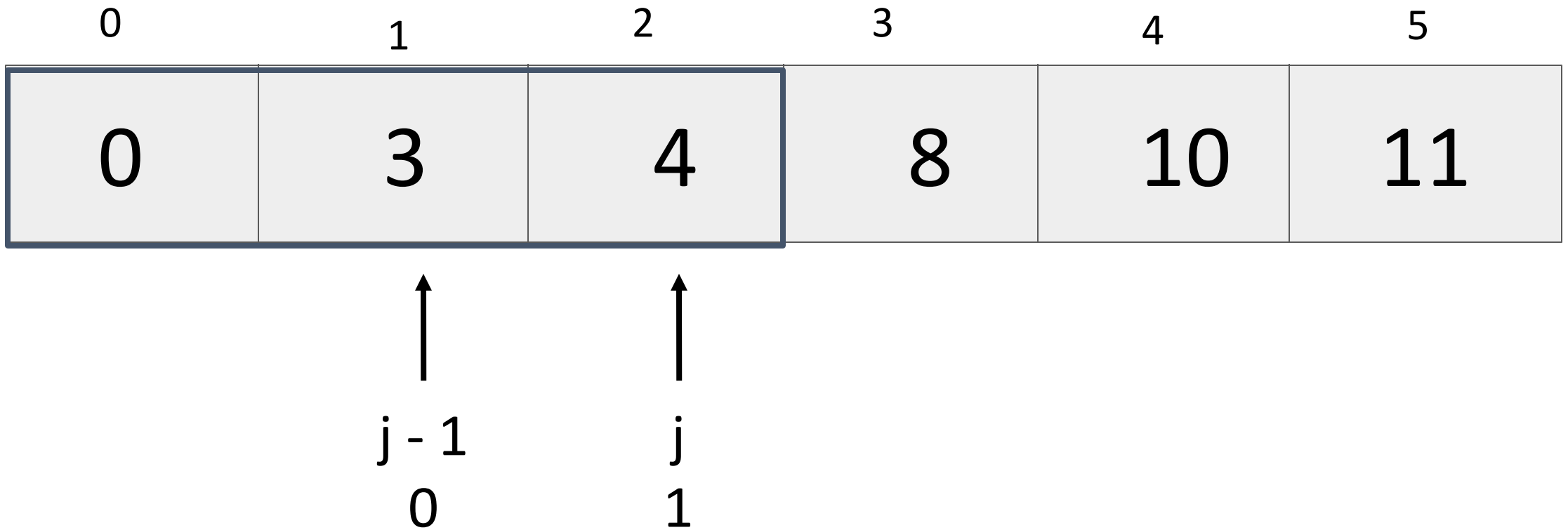


Reset; Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? 44

# Bubble Sort

len = 3

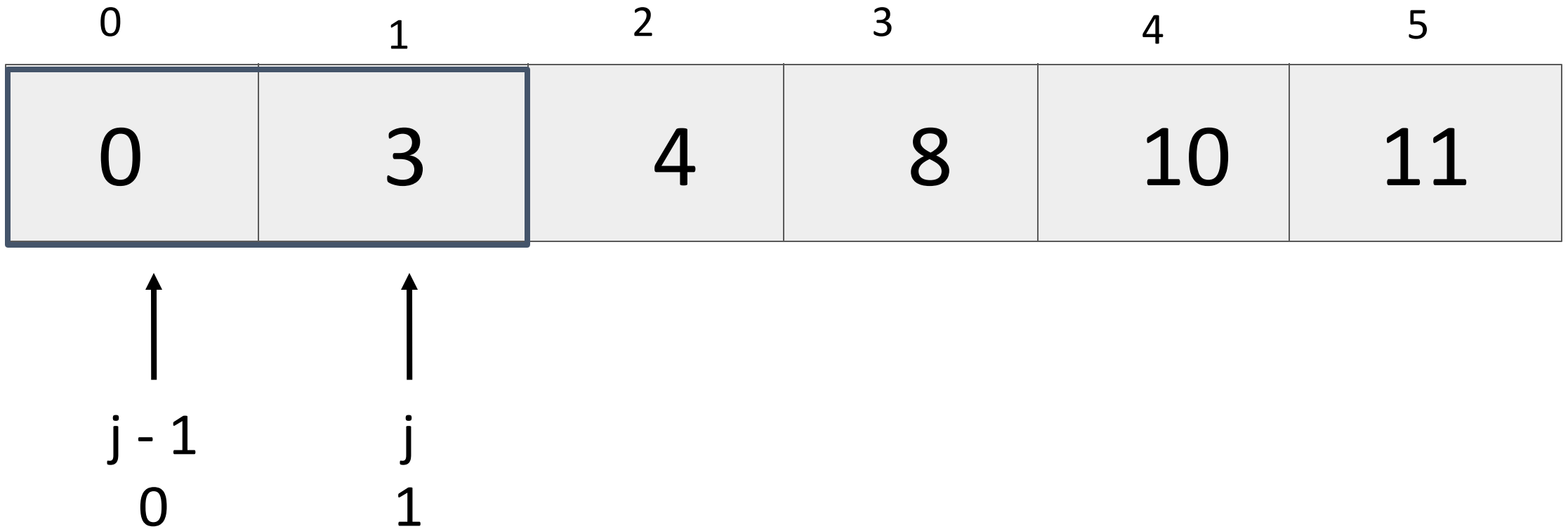


Reset; Compare  $j-1$  and  $j$ ; Swap if  $L[j-1] > L[j]$

What next? <sup>45</sup>

# Bubble Sort

len = 2



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

What next? 46

# 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)
```

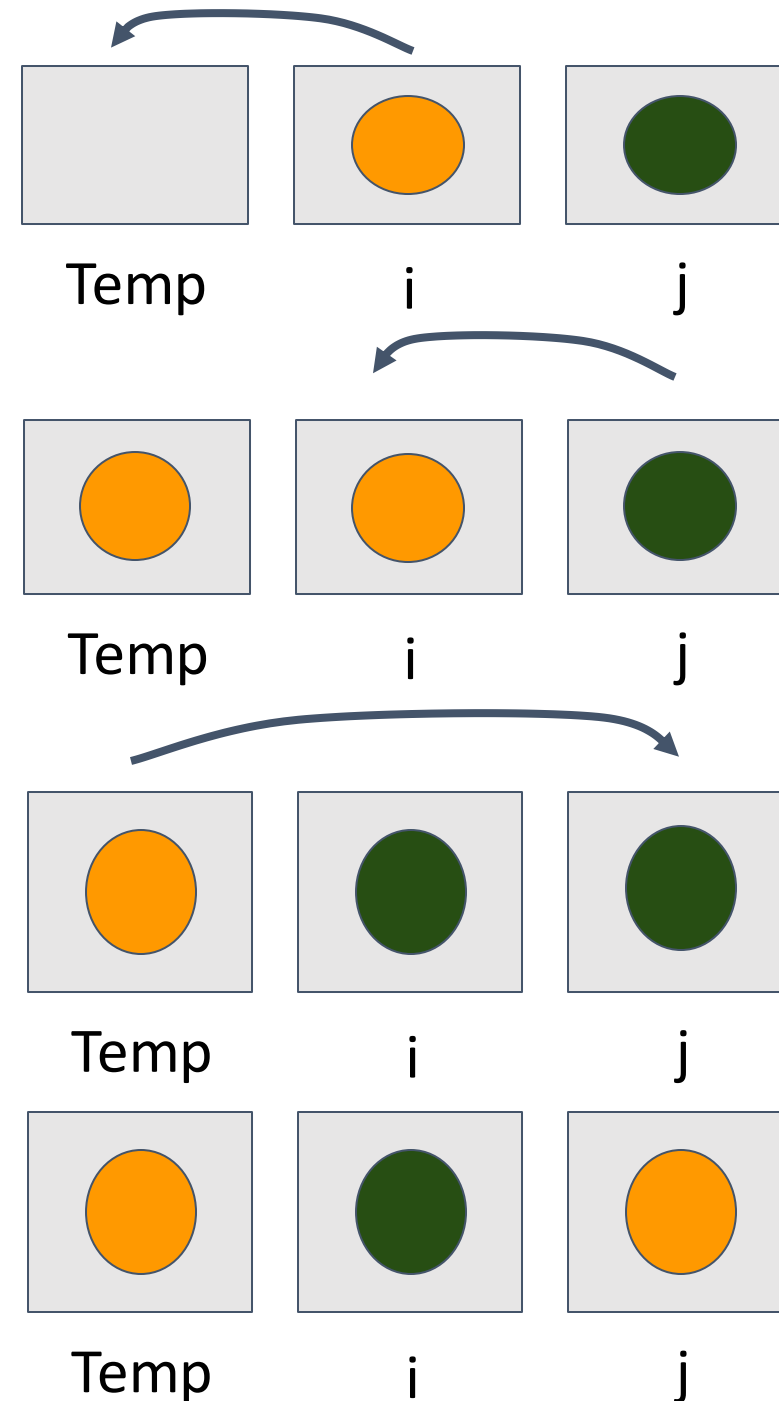
# Bubble sort

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)`

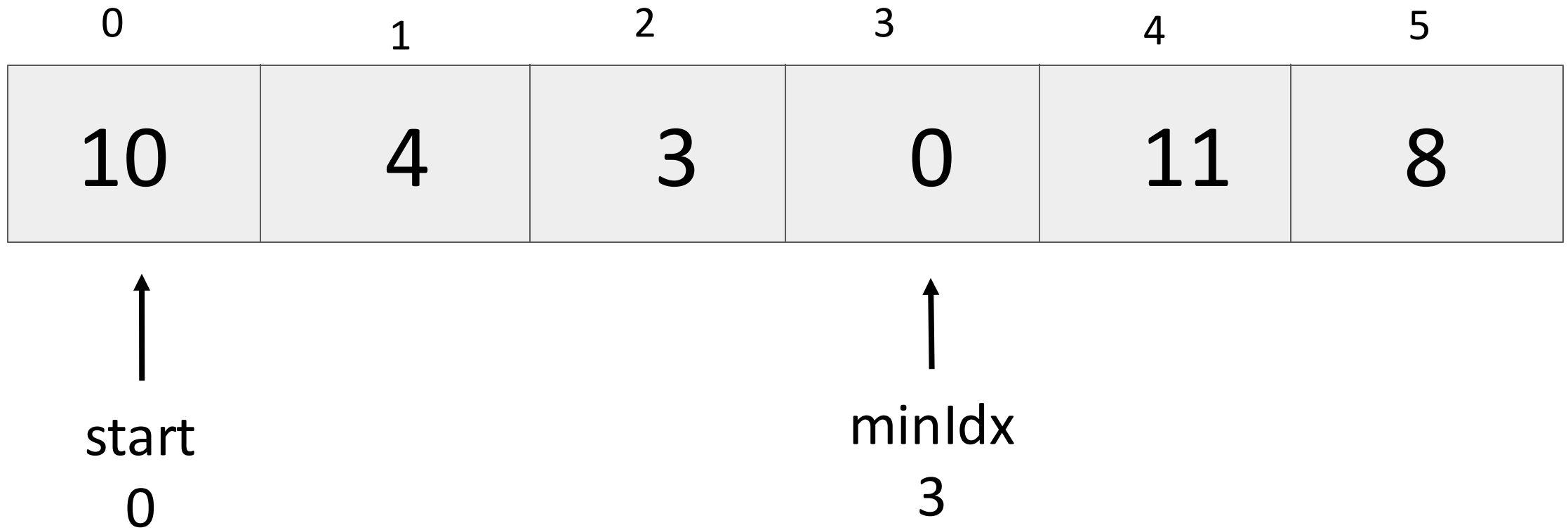
`swap(startIdx, minIdx, L)`

# Selection Sort

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

What do we do first?

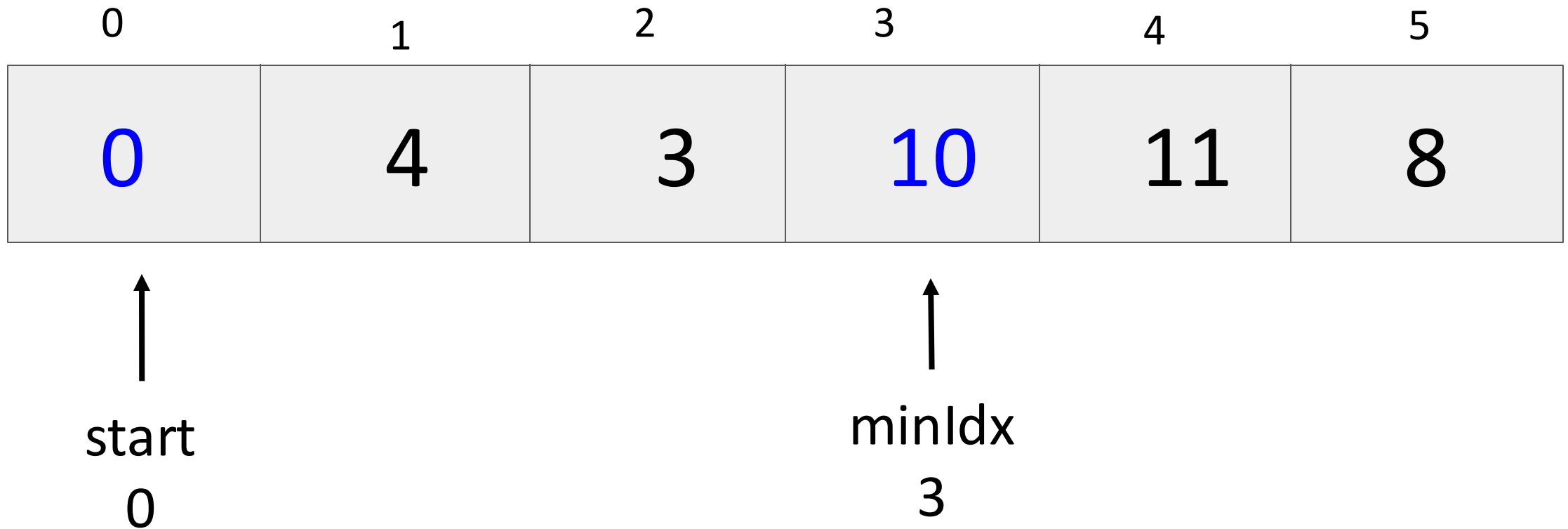
# Selection Sort



Find minimum element idx between start to end

What next? <sup>52</sup>

# Selection Sort



Swap the elements at start and minIdx

What next? <sup>53</sup>

# Selection Sort

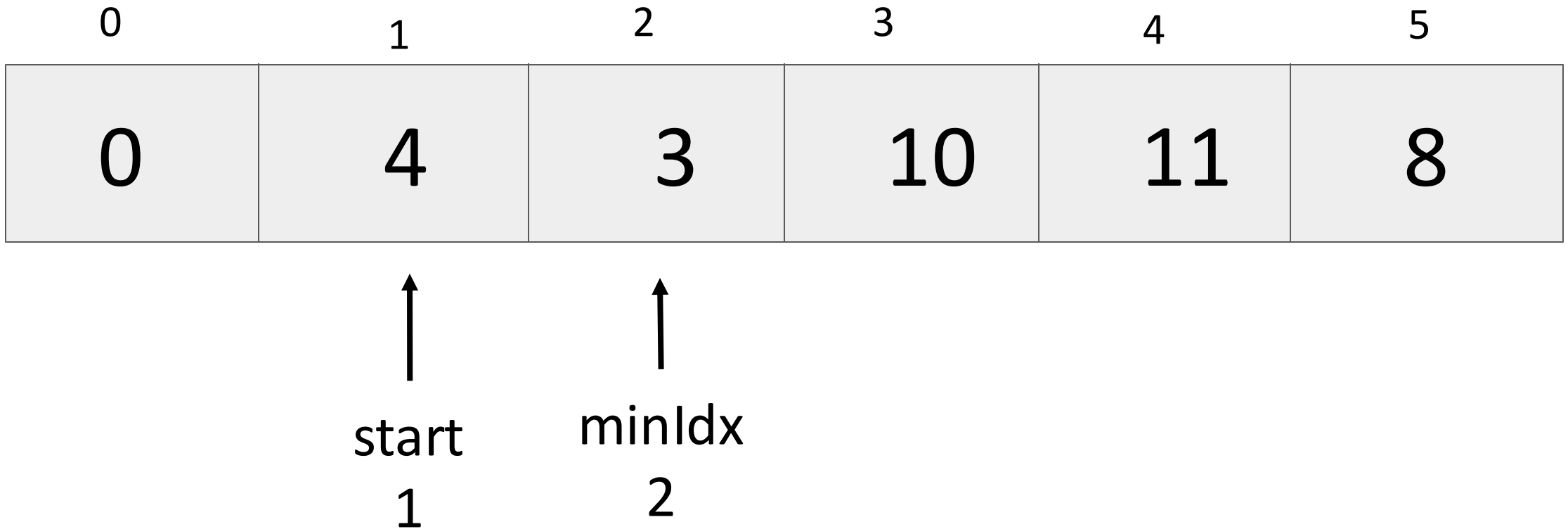
0	1	2	3	4	5
0	4	3	10	11	8

↑  
start  
1

Decrease the interval.

What next? <sup>54</sup>

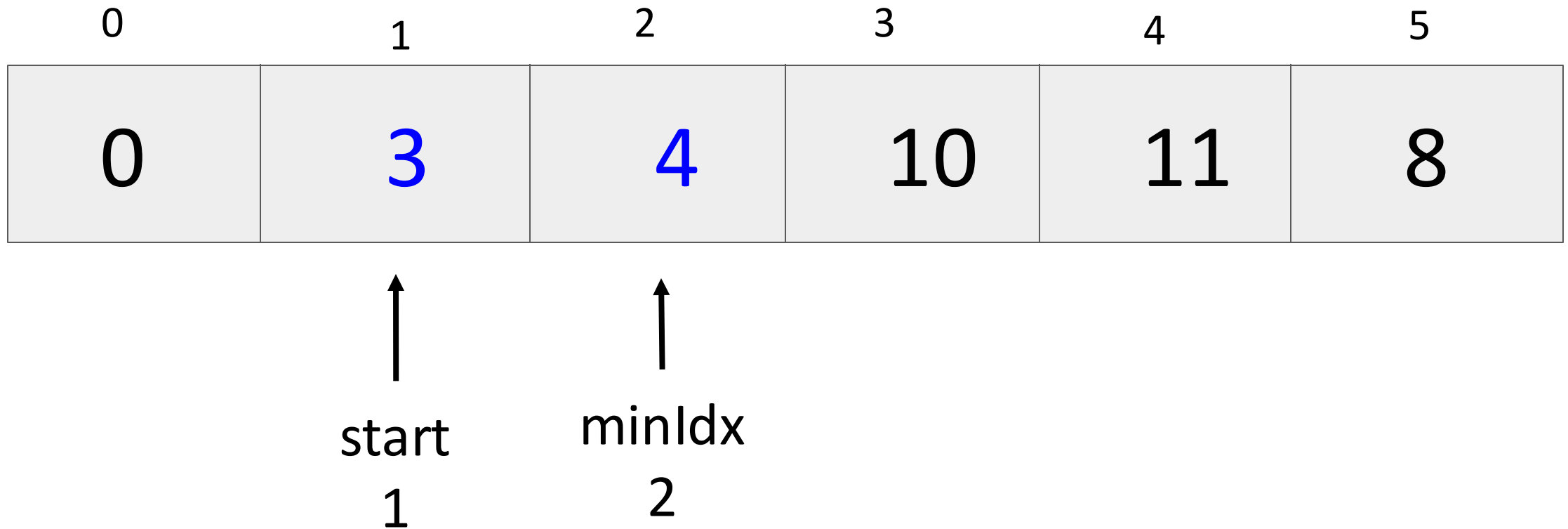
# Selection Sort



Find minimum element between start to end

What next? <sup>55</sup>

# Selection Sort



Swap the elements at start and minIdx

What next? <sup>56</sup>



# Selection Sort

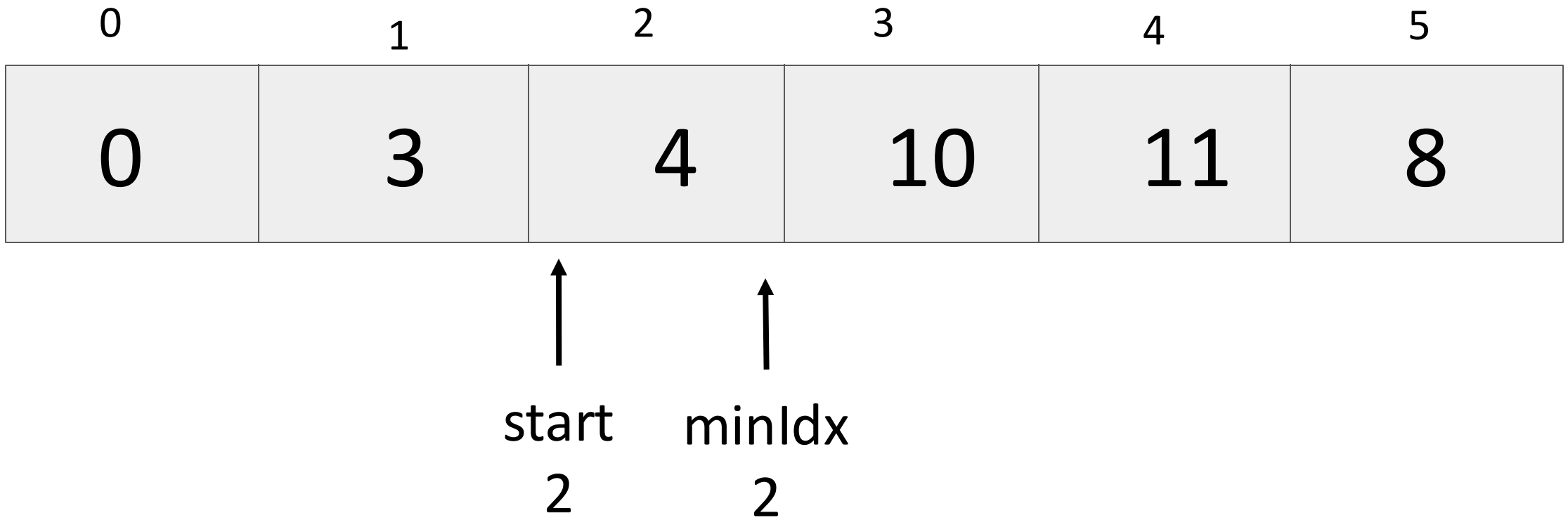
0	1	2	3	4	5
0	3	4	10	11	8

↑  
start  
2

Decrease the interval.

What next? <sup>57</sup>

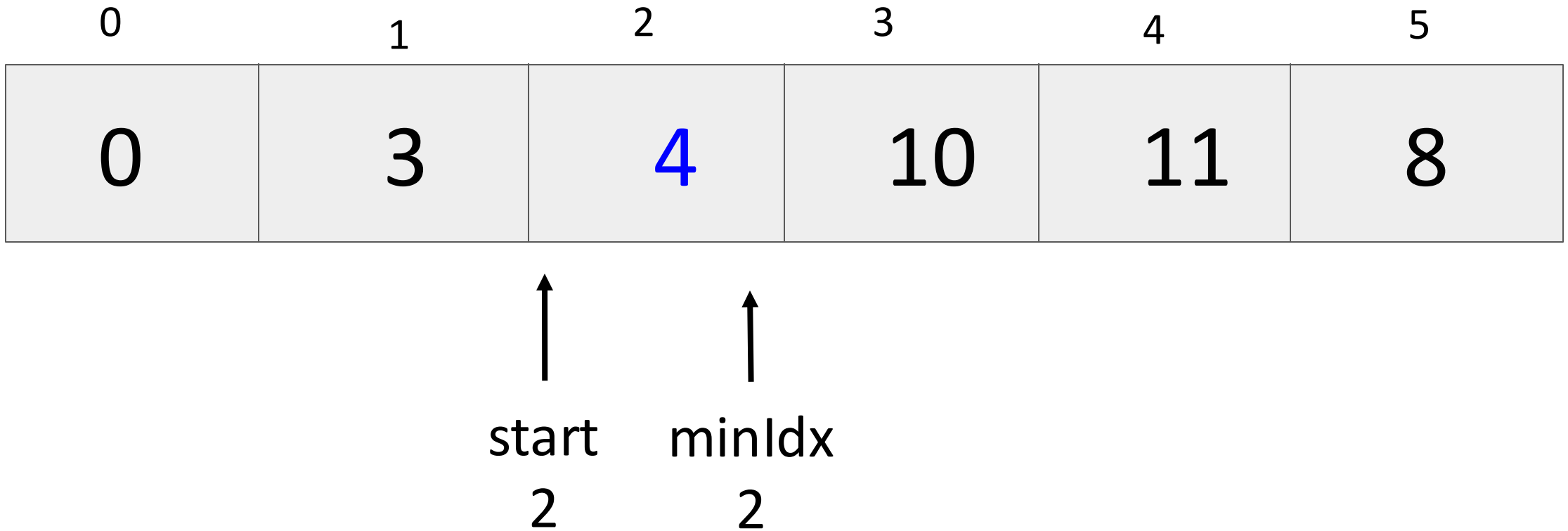
# Selection Sort



Find minimum element idx between start to end

What next? <sup>58</sup>

# Selection Sort



Swap the elements at start and minIdx

What next? <sup>59</sup>

# Selection Sort

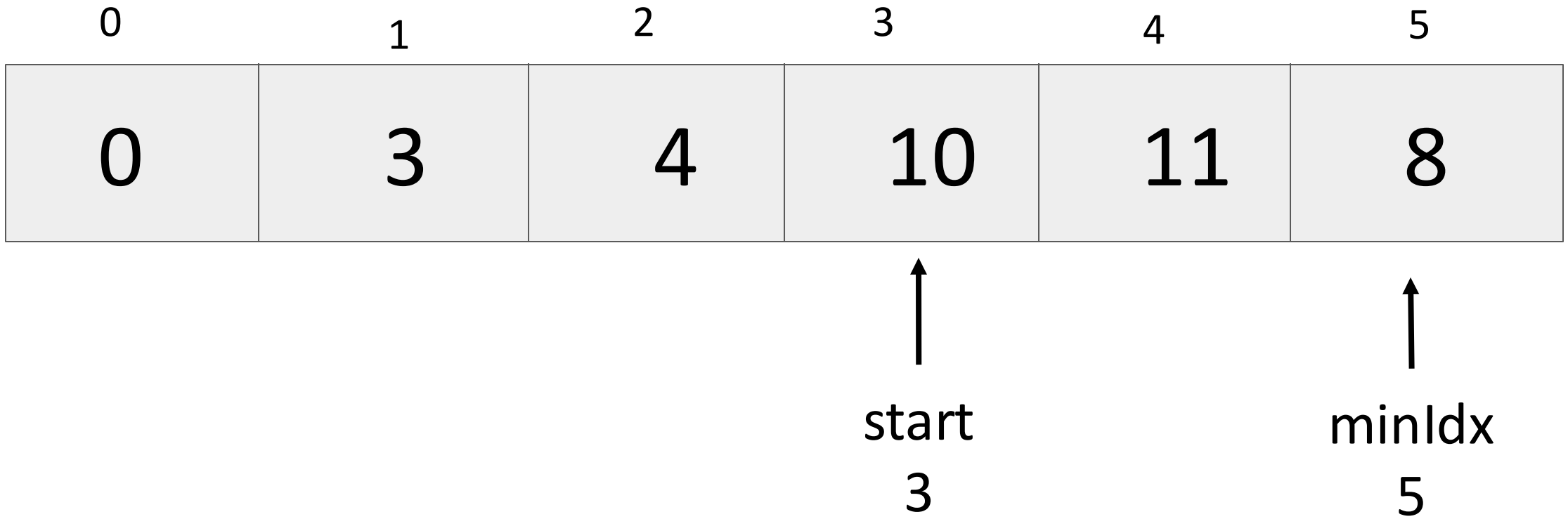
0	1	2	3	4	5
0	3	4	10	11	8

↑  
start  
3

Decrease the interval.

What next? 60

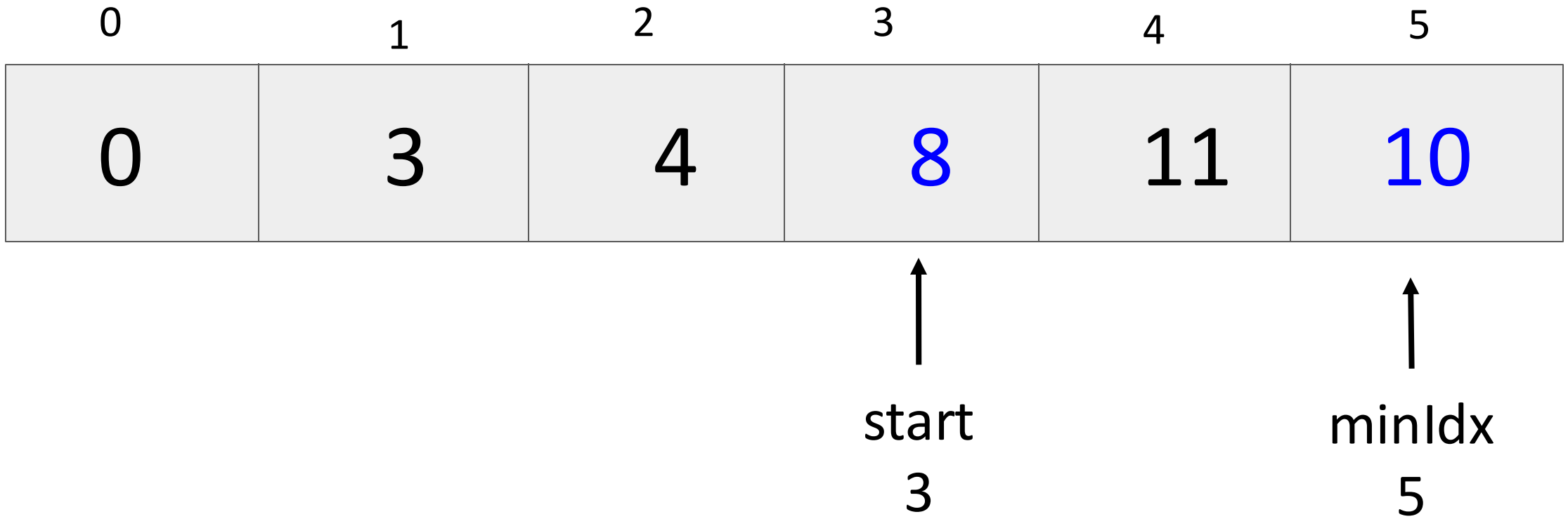
# Selection Sort



Find minimum element idx between start to end

What next? <sup>61</sup>

# Selection Sort



Swap the elements at start and minIdx

What next? <sup>62</sup>

# Selection Sort

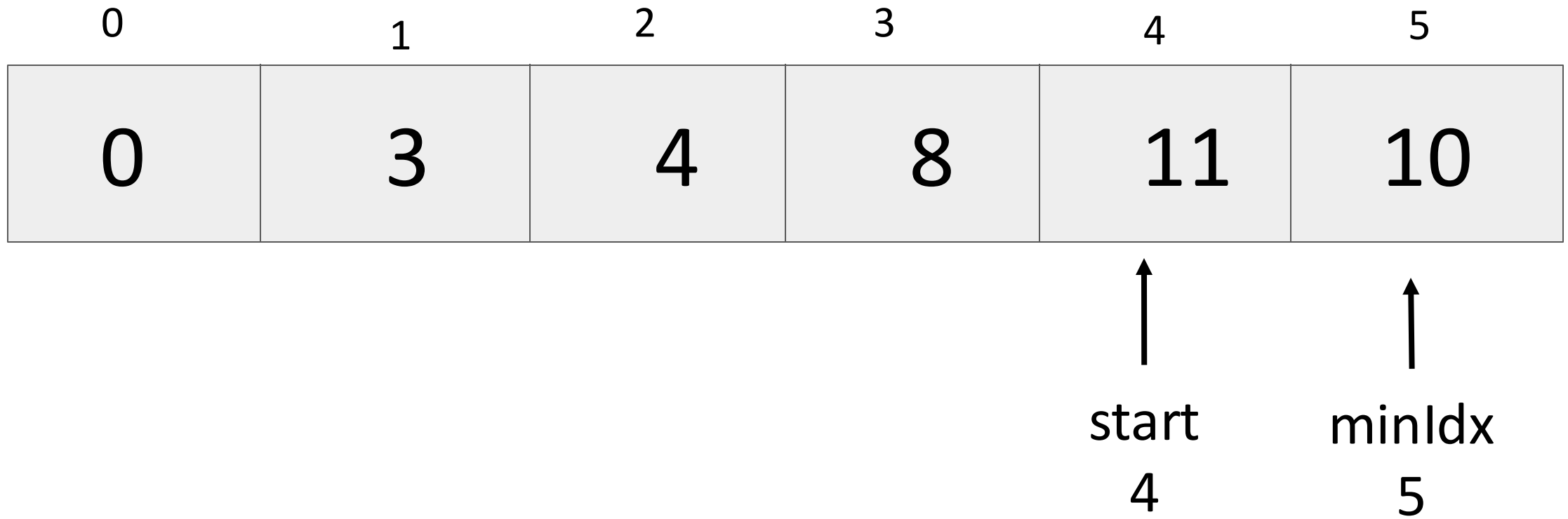
0	1	2	3	4	5
0	3	4	8	11	10

↑  
start  
4

Decrease the interval.

What next? <sup>63</sup>

# Selection Sort

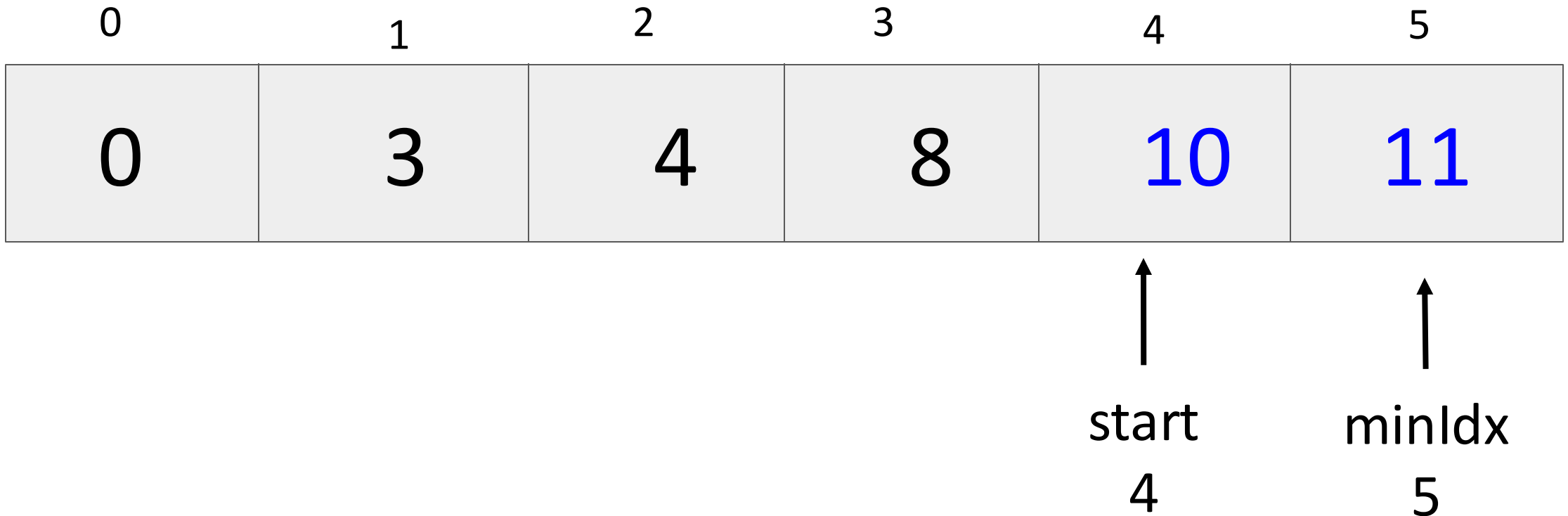


Find minimum element idx between start to end

What next? <sup>64</sup>



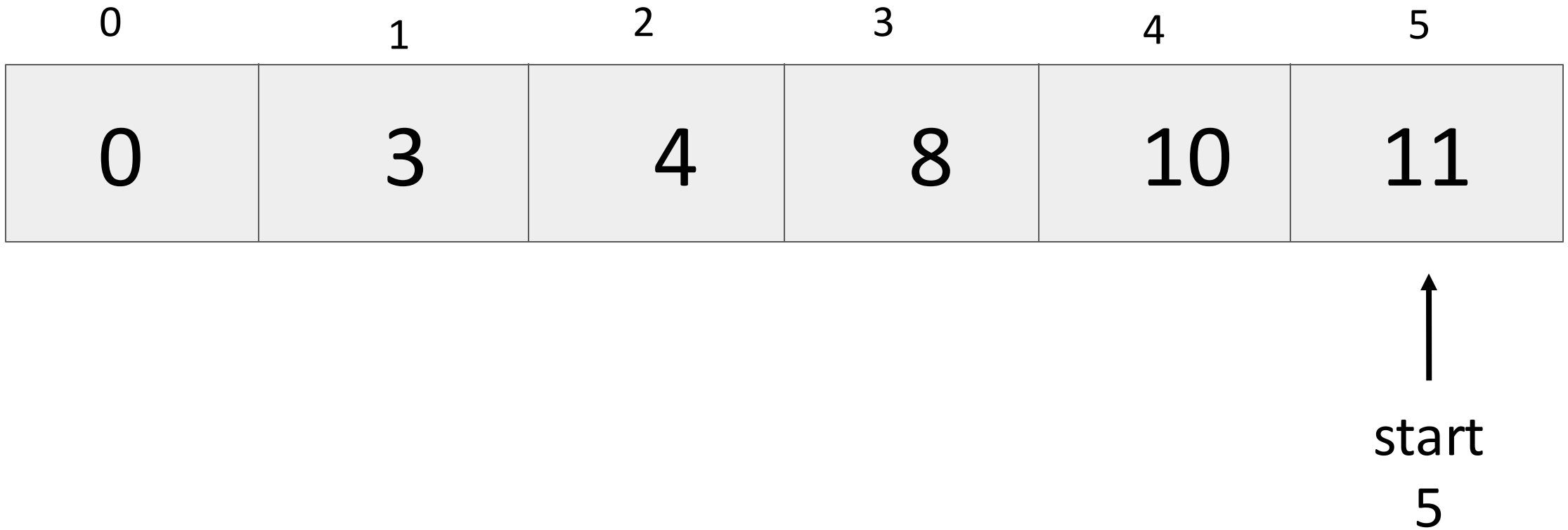
# Selection Sort



Swap the elements at start and minIdx

What next? <sup>65</sup>

# 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
```

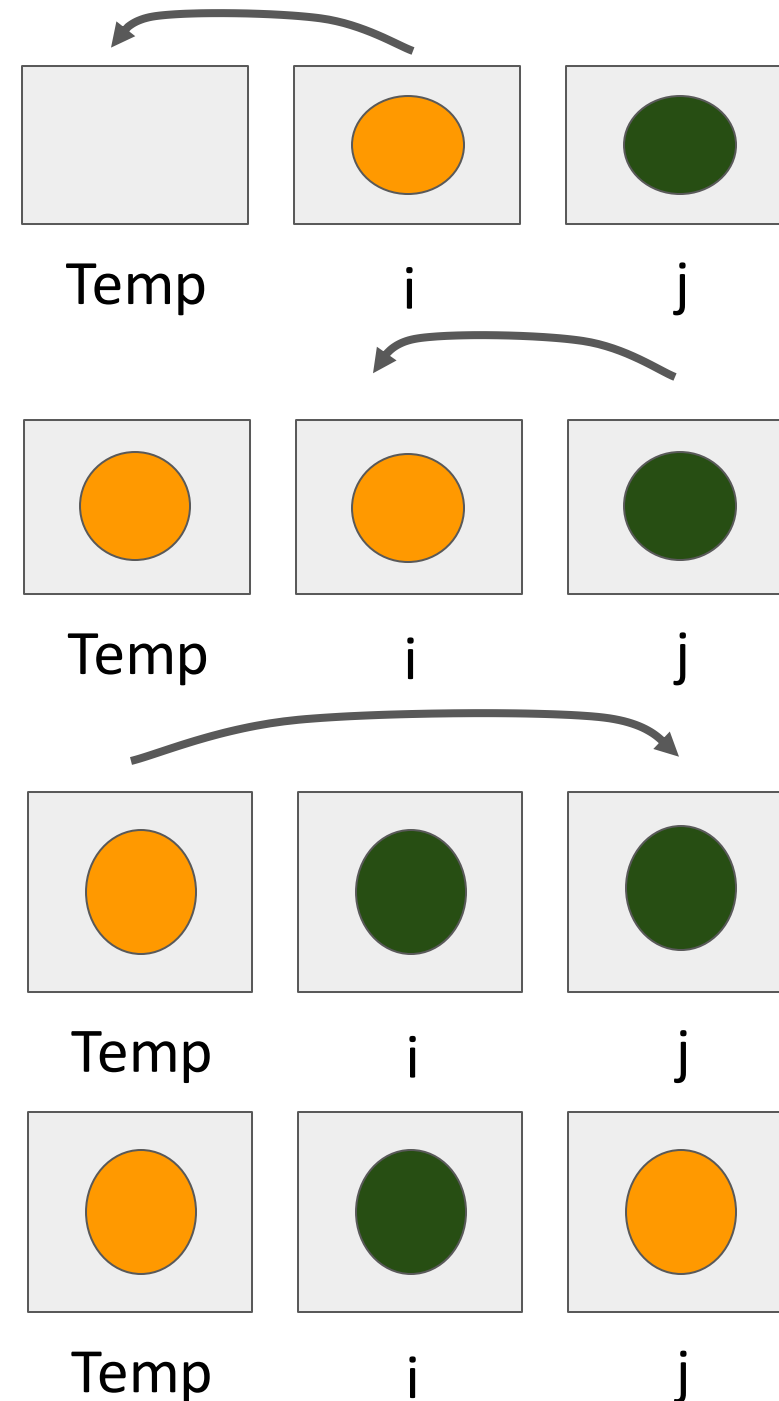
# Swap

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 $O(N^2)$

