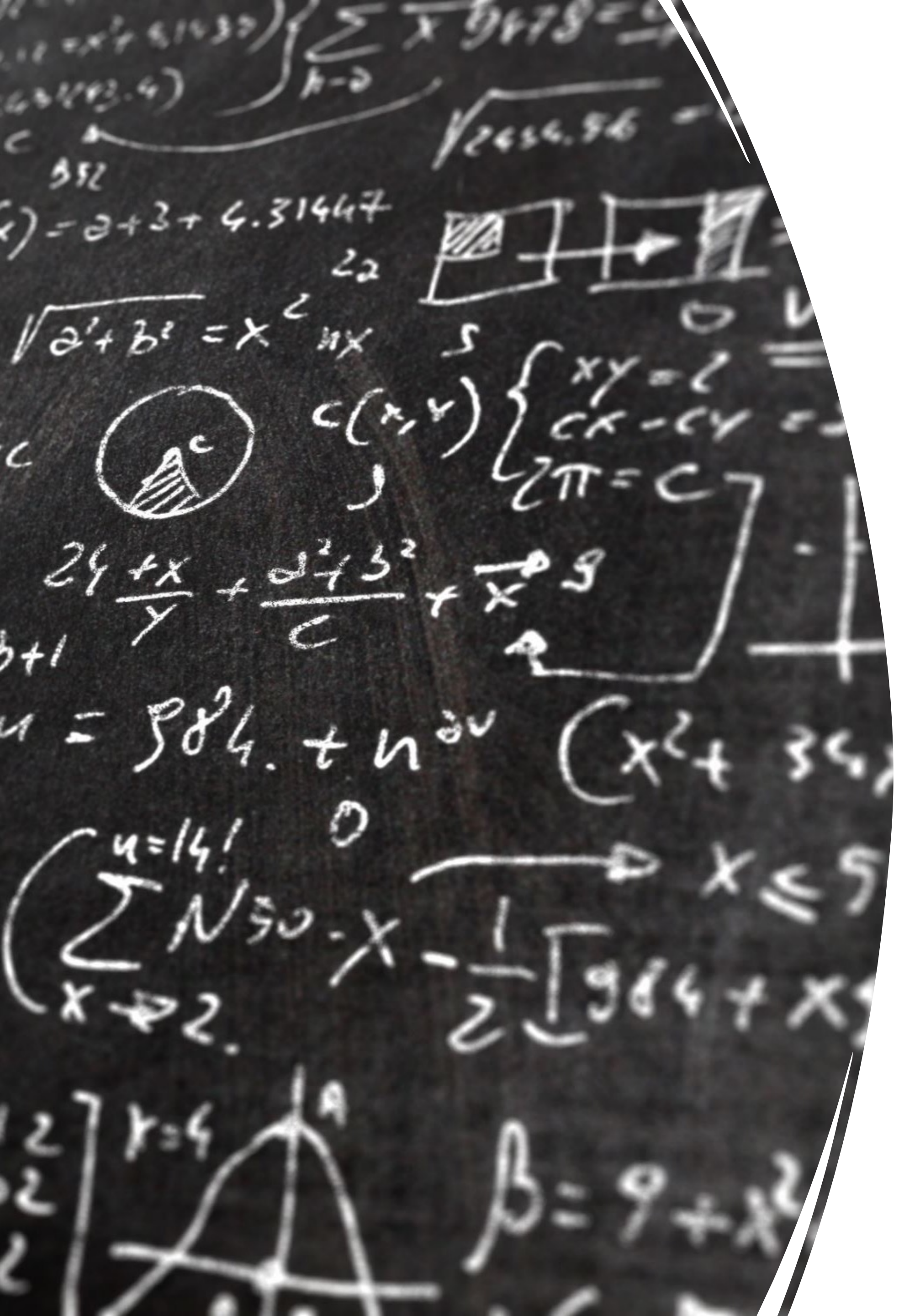


CS 113 – Computer Science I

Lecture 22 – Sorting, Midterm Review

Adam Poliak

04/11/2023

Announcements

- Midsemester feedback

- Midterm 04/13
  - In class
  - Department admin will drop off midterm

- HW08:
  - Due 04/20 - will be released by Friday
  - Inheritance and interfaces – fully autograded

- HW09:
  - Due 04/27
  - Building a fancyArray class – fully autograded

Outline

- Review
- Sorting

Midterm topics

Frame diagrams

Conditionals

Recursion/Loops

Object Oriented Programming

Searching

Object Oriented Programming

Classes vs objects

Designing classes

Mutable vs immutable objects

Instance vs static vs abstract methods

Relationship between classes

- Inheritance
- Interfaces

Classes vs Objects

Class:

- custom data types that contains
    - the data (**instance variables**)
    - the operations (**instance methods**)

Object:

- an instance of the class

Example:

- String vs “hello world”

Designing classes

All classes should have:

- Constructor:
  - Difference between value and empty constructor
- Getters/accessors
- Comparators (equal() or compareTo()) – zoom poll
- toString()
- Setters
  - We’ll see an example later where we wont want to have setters

Access modifiers

Instance variables and methods can be

`private`

- Can't be accessed directly by anyone else

`protected`

- Only subclasses can access these

`public`

- Anyone that has access to the object can access these

Mutable vs immutable objects

Whether data stored inside an object can change **(mutable**) or cannot change (**immutable**) once the object is created

Example: Zoom poll

Strings are **immutable**

Arrays are **mutable**

How would we design an **immutable** object

make instance variables `private`

do not include any setters

Static vs instance methods

Static

- Do not require an object
- no access to `this` keyword
- Examples:
  - Integer.parseInt(“99”);
  - Math.random();

Instance

- Acts on an object -> requires an objects
- has access to `this` keyword
- Examples:
  - “hello,world”.split(“,”)

Abstract methods

Contains method signatures: name, arguments, and return type

Does not include an implementation

Specify what a method does, not how it does it

Often used in interfaces

Each subclass that implements the interface can choose how to implement the method

Class relationships - inheritance

A subclass is a class that `extends` an existing class; that is, it has the attributes and methods of the existing class, plus more.

- Refer to the existing class as a *parent* or *superclass*
- When a class `extends` another class, it *inherits* the attributes and methods from the parent class

All classes by default extend *java.lang.Object.*

- Consequence: Compiler knows to call *"toString()"*

Inheritance example

```java
public class Animal {

  protected String name;
  protected double weight;

  public Animal(String name,
                double weight) {

    this.name = name;
    this.weight = weight;
  }

  public void eat(Animal prey) {
    weight += prey.getWeight();
  }

  public double getWeight() {
    return weight;
  }
}
```

```java
public class Fish extends Animal {

  private String ocean;

  public Fish(String name,
              double weight,
              String ocean){
    super(name, weight);
    this.ocean = ocean;
  }

  public void move(String newOcean) {
    ocean = newOcean;
  }

  public String getOcean() {
    return ocean;
  }
}
```

Inheritance example

```
public class Animal {

  protected String name;
  protected double weight;

  public Animal(String name,
                double weight) {

    this.name = name;
    this.weight = weight;
  }

  public void eat(Animal prey) {
    weight += prey.getWeight();
  }

  public double getWeight() {
    return weight;
  }
}
```

```
public class Fish extends Animal {

  private String ocean;

  public Fish(String name,
              double weight,
              String ocean){
    super(name, weight);
    this.ocean = ocean;
  }
```

```
public static void main(String[] args) {
  Animal worm = new Animal("worm", 0.5);
  Fish nemo = new Fish("nemo", 2.0, "pacific");
  nemo.eat(worm);
  System.out.println(nemo.getWeight());
}
```

Inheritance example

```java
public class Animal {

  protected String name;
  protected double weight;

  public Animal(String name,
                double weight) {

    this.name = name;
    this.weight = weight;
  }

  public void eat(Animal prey) {
    weight += prey.getWeight();
  }

  private double getWeight() {
    return weight;
  }
}
```

```java
public class Fish extends Animal {

  private String ocean;

  public Fish(String name,
              double weight,
              String ocean){
    super(name, weight);
    this.ocean = ocean;
  }
```

```java
ublic static void main(String[] args) {
  Animal worm = new Animal("worm", 0.5);
  Fish nemo = new Fish("nemo", 2.0, "pacific");
  nemo.eat(worm);
  System.out.println(nemo.getWeight());
```

Designing classes

Time class:

- Hour, minute, second

Date class:

- Day, month, and year
- Contains everything in Time as well

Whats the superclass and whats the subclass?
How could we make these ***immutable***?
How could we define the distance between two Time or two Date objects?

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

If we have 100 items in a list, how many items do we have to check in the worstcase scenario?

All 100

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 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 isnt what we are looking for, what should we do?

Chop the search space in half (this is why its 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 $\log n$ $(\text{we use base } 2)$. 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:

$$\log_2 512 = 9$$

Review

What else?

Outline

- Review
- **Sorting**

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

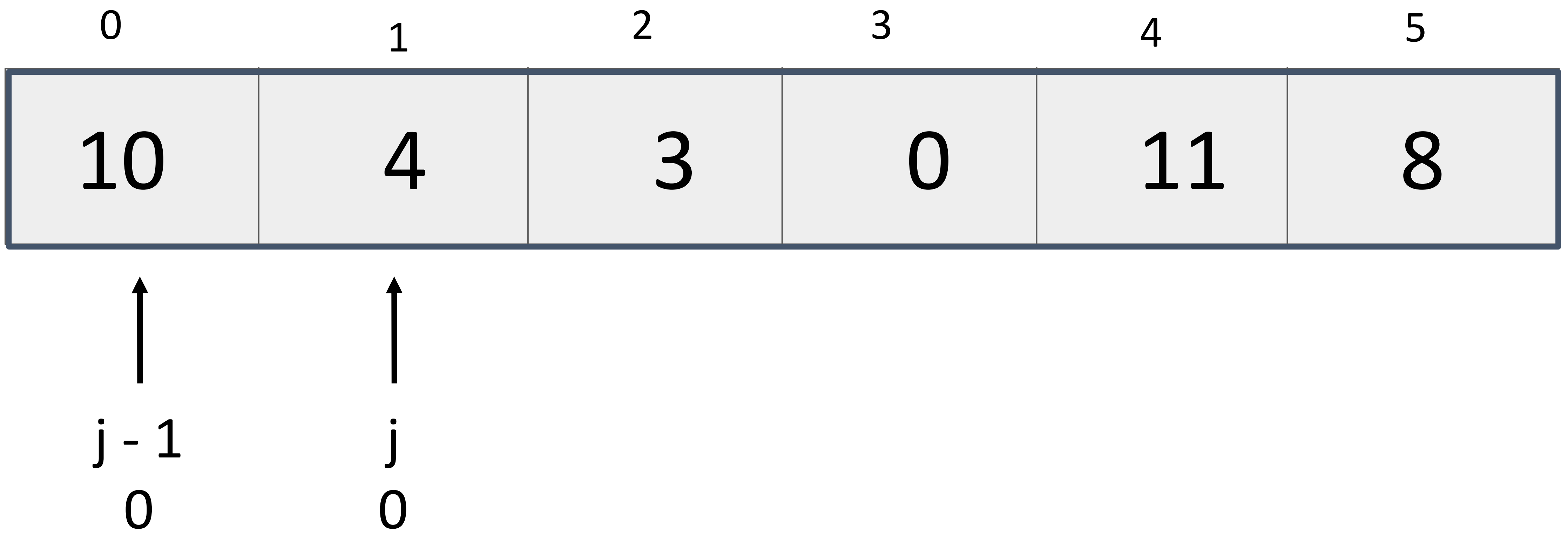
Bubble Sort

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

What do we do first?

Bubble Sort

len = 6

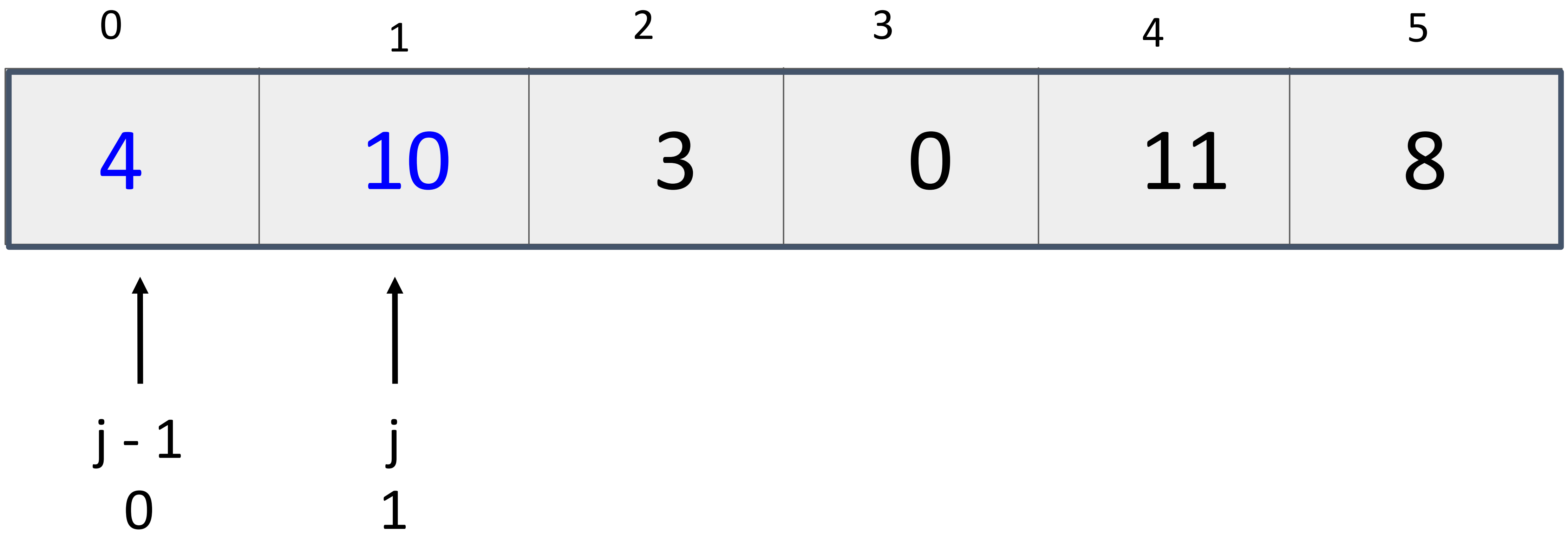


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

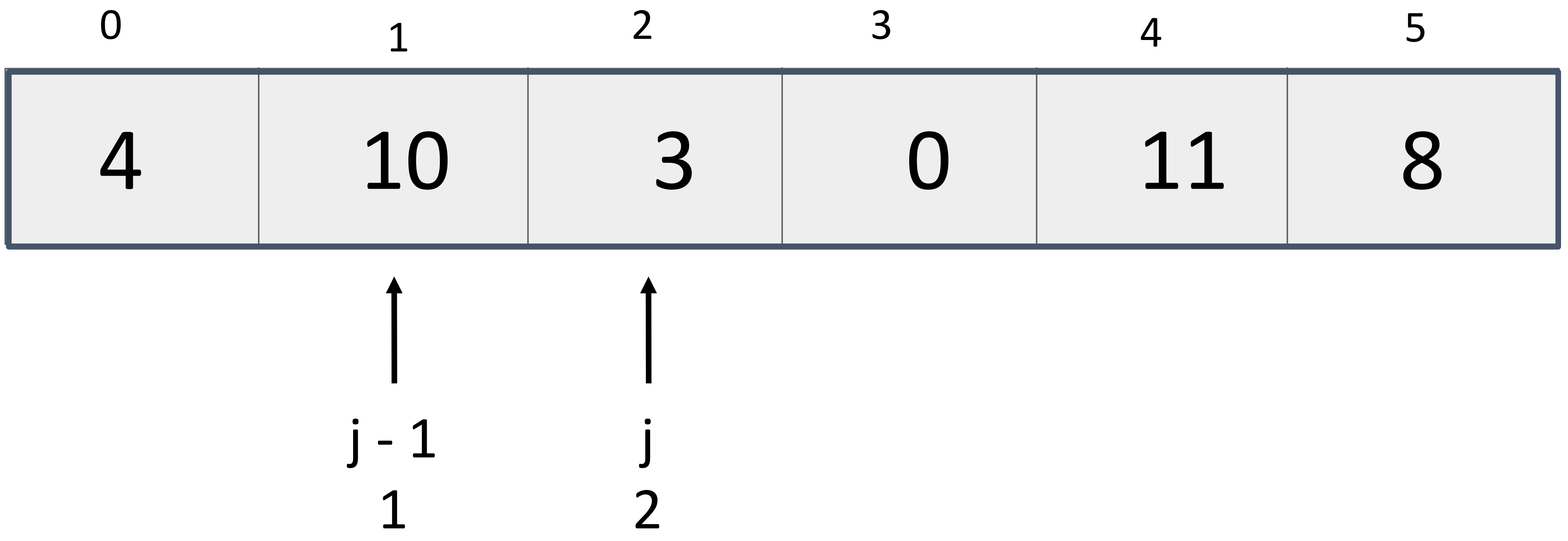


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

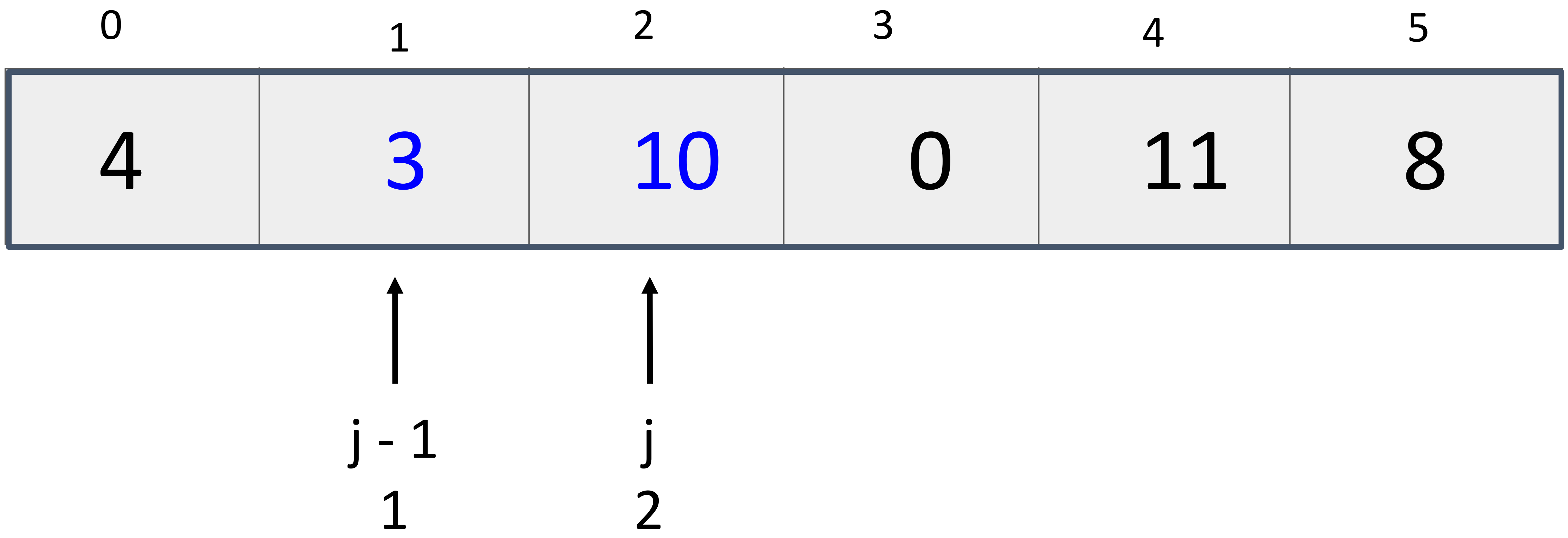


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

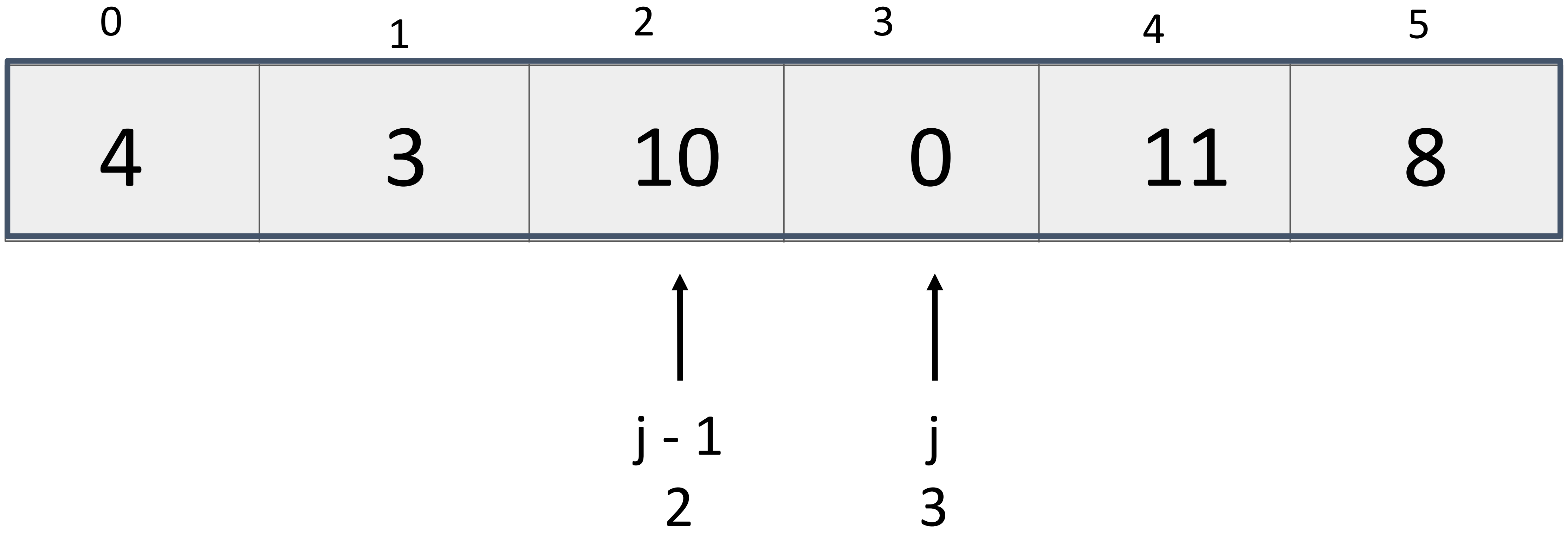


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

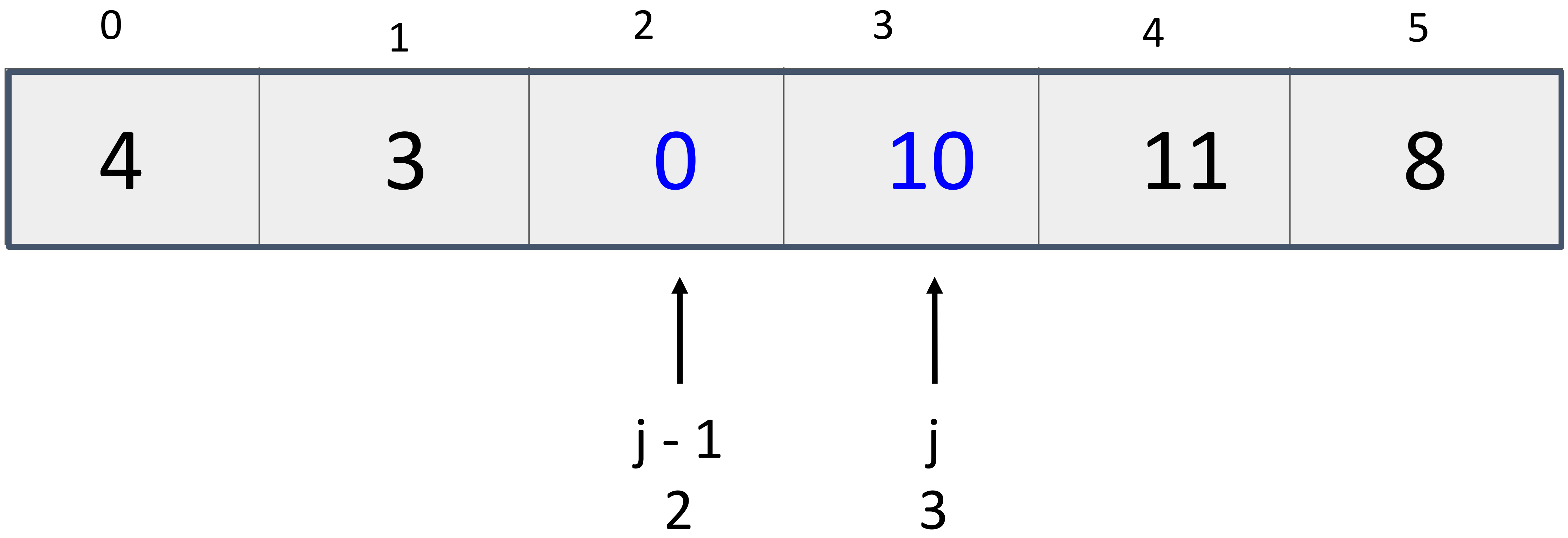


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

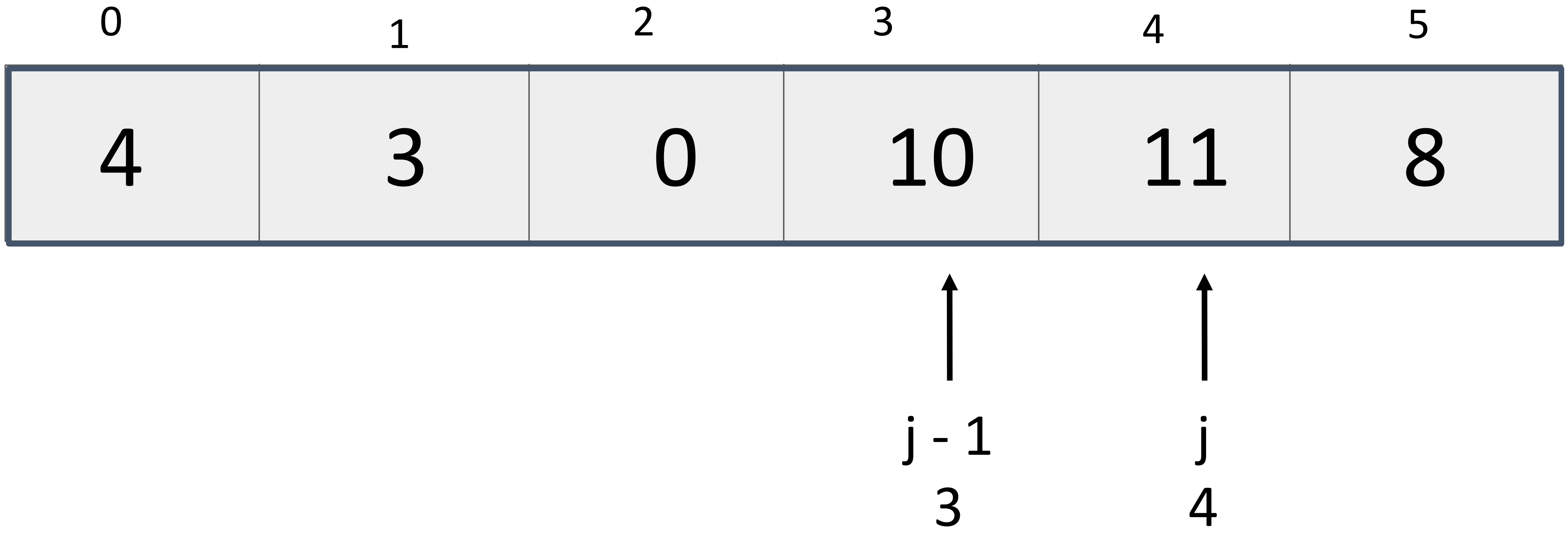


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

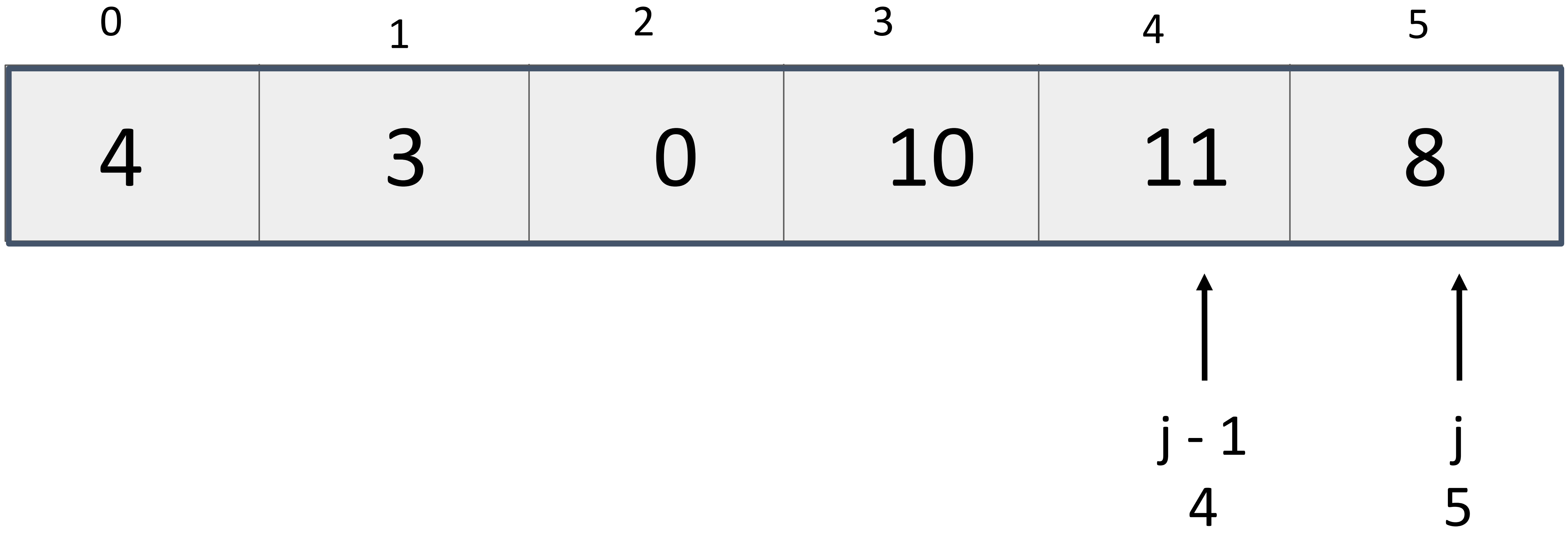


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

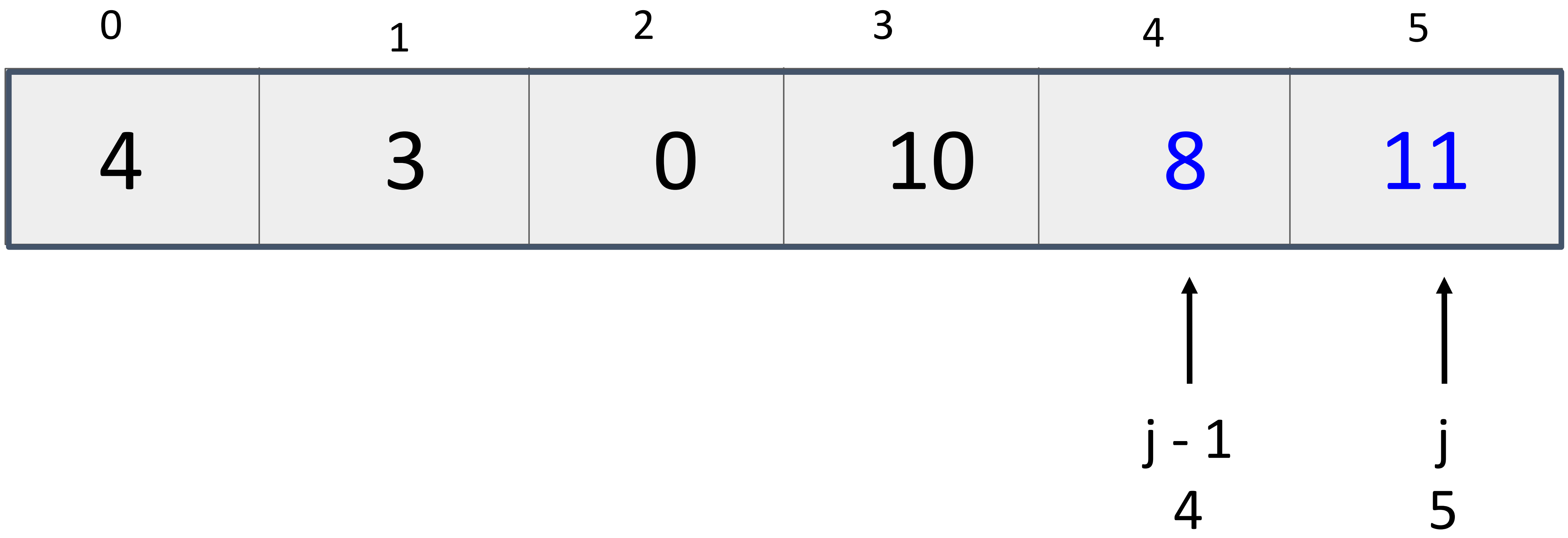


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 6

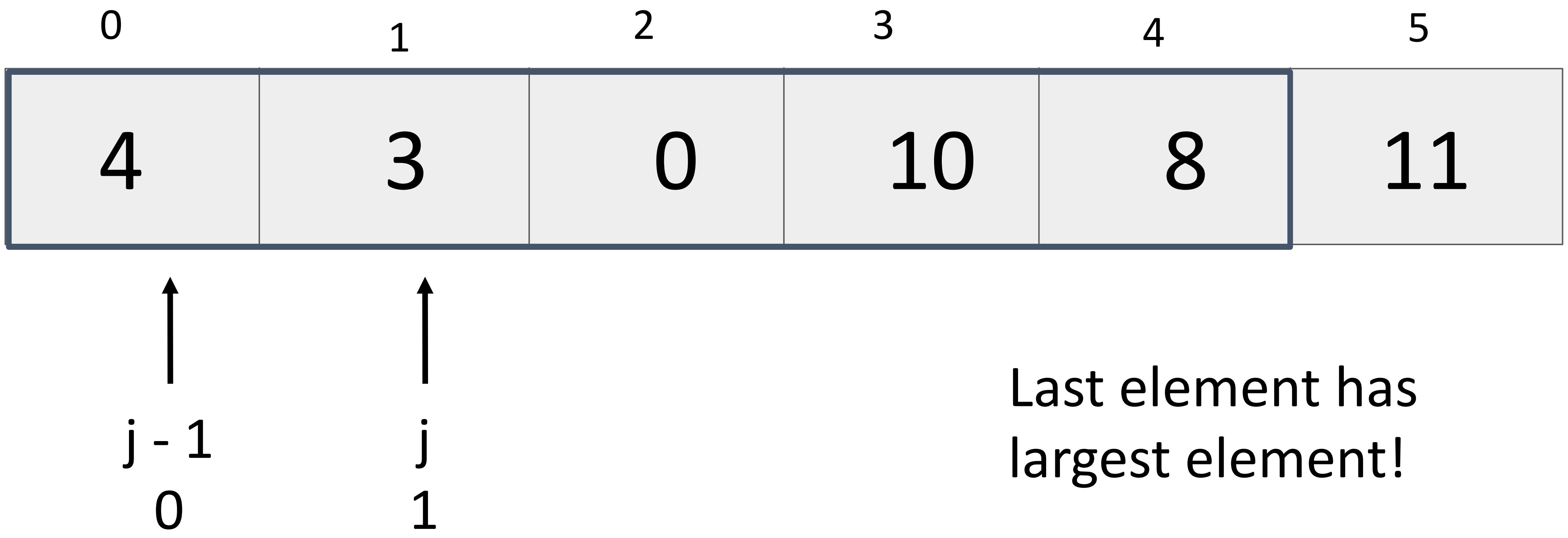


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5



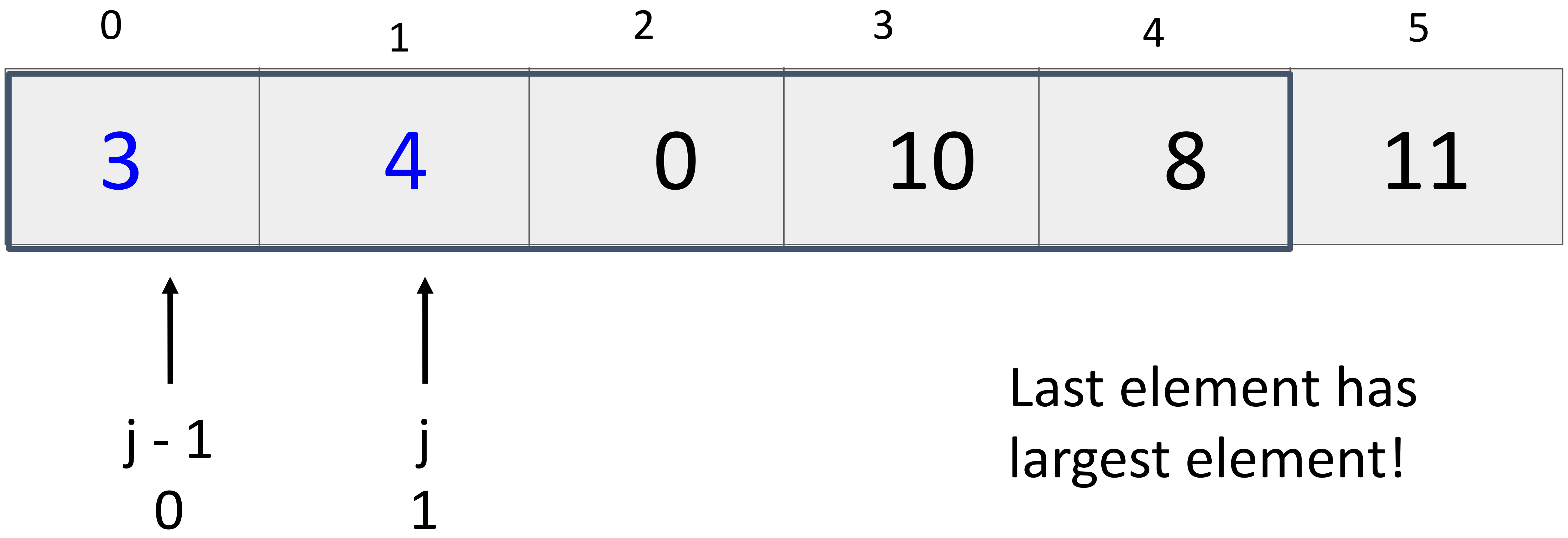
Last element has largest element!

Reset and compare pairs with shorter list!

What next?

Bubble Sort

len = 5



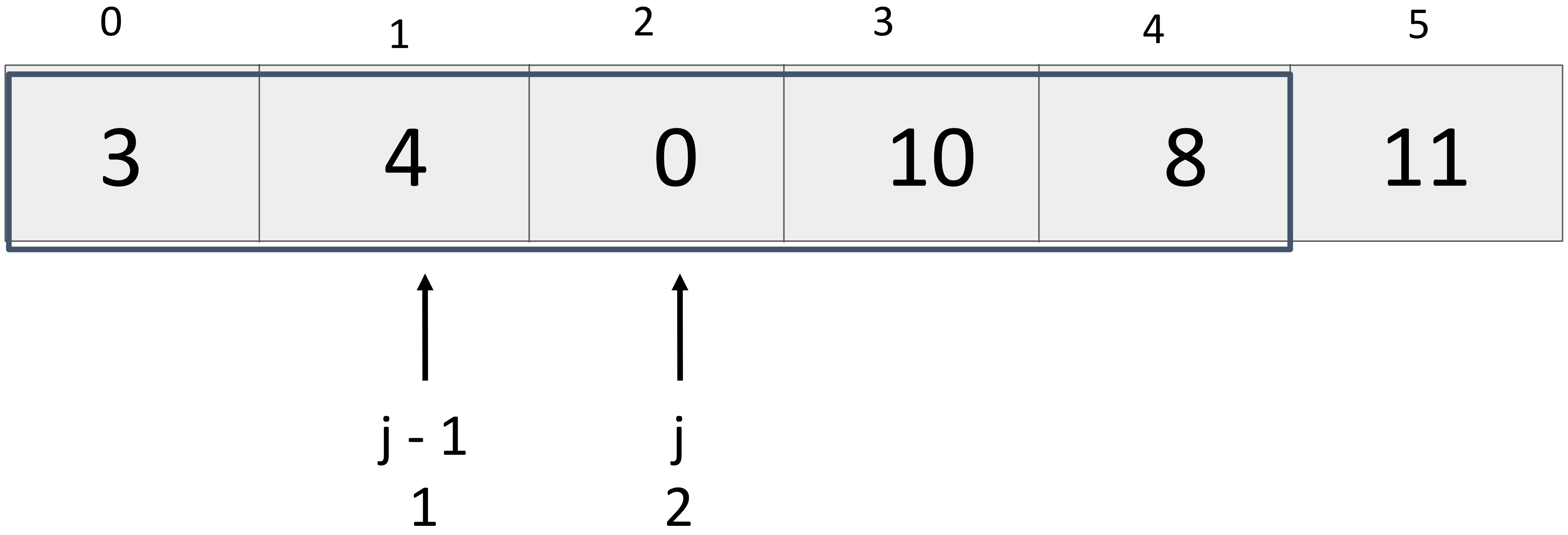
Last element has largest element!

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

What next?

Bubble Sort

len = 5

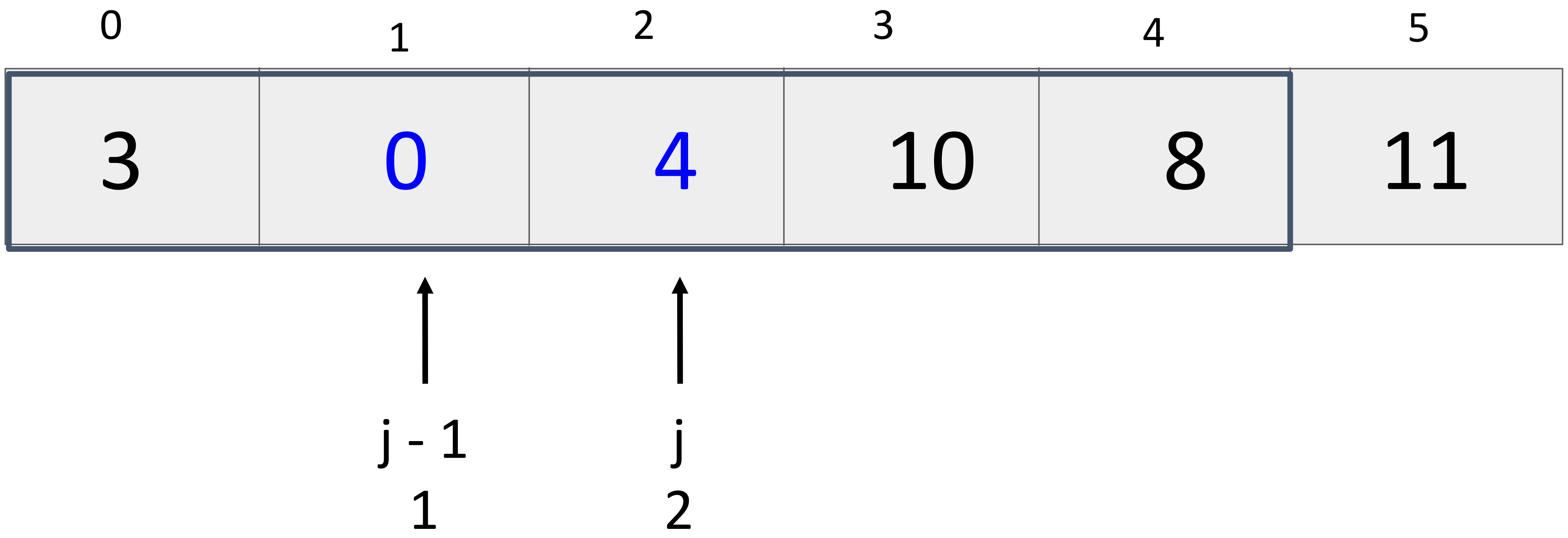


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

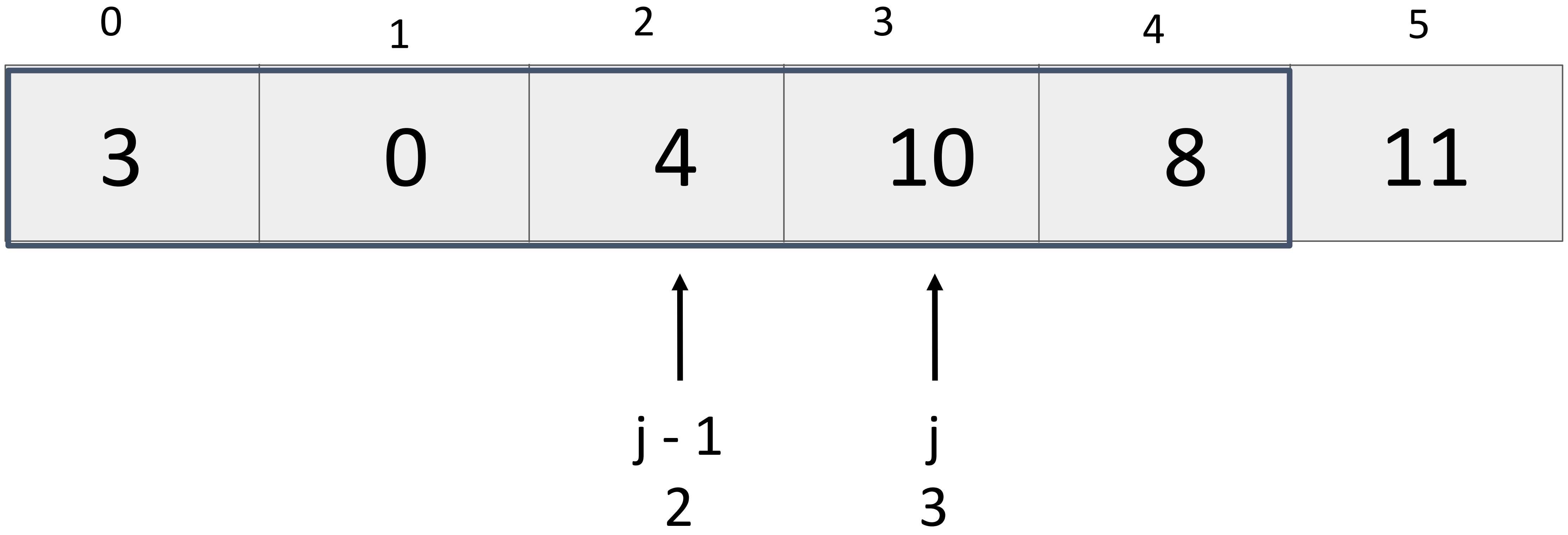


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

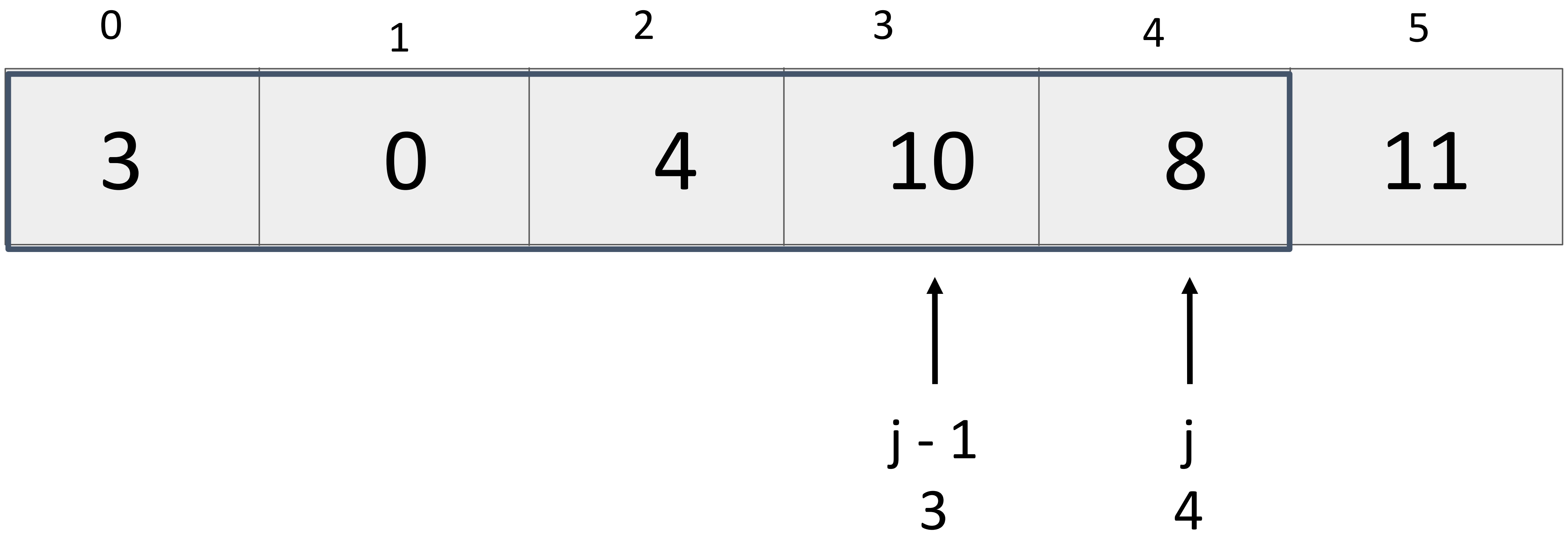


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

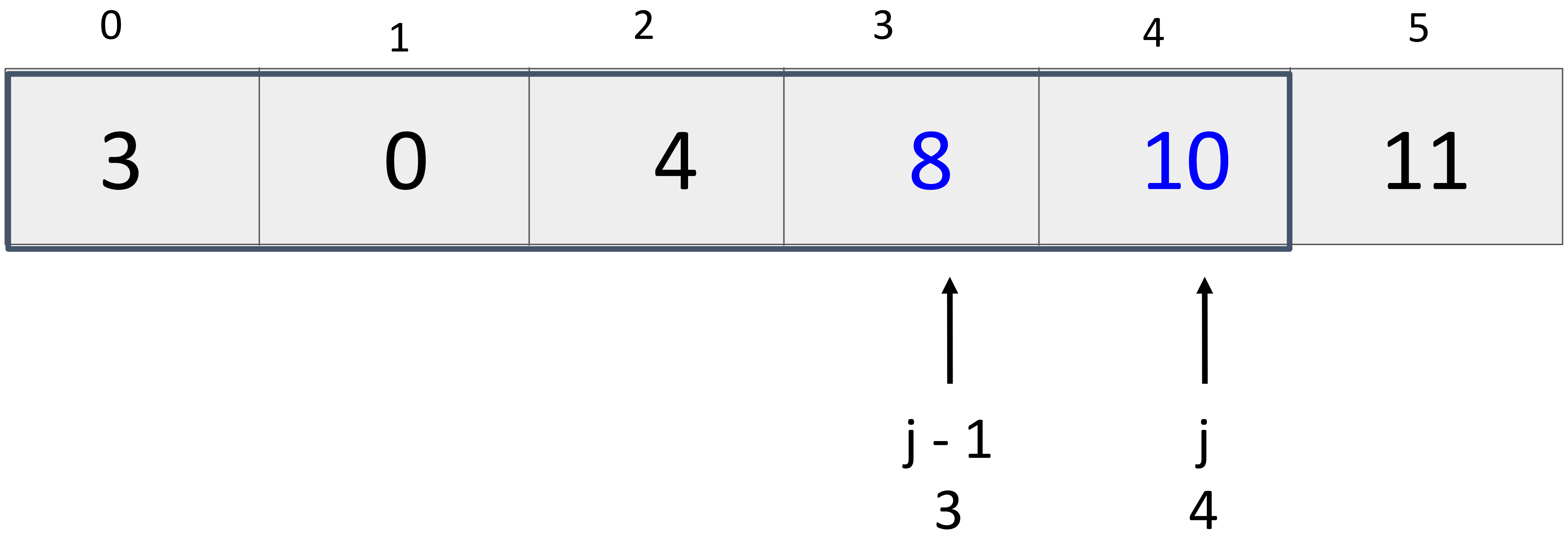


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 5

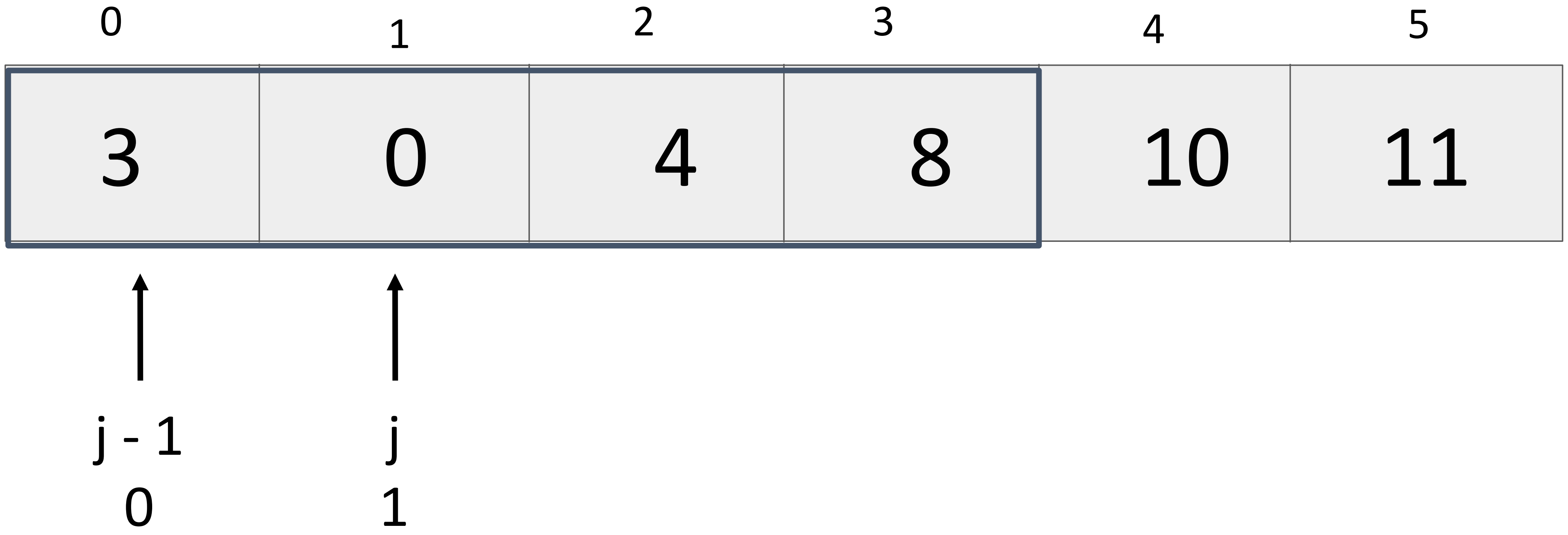


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 4

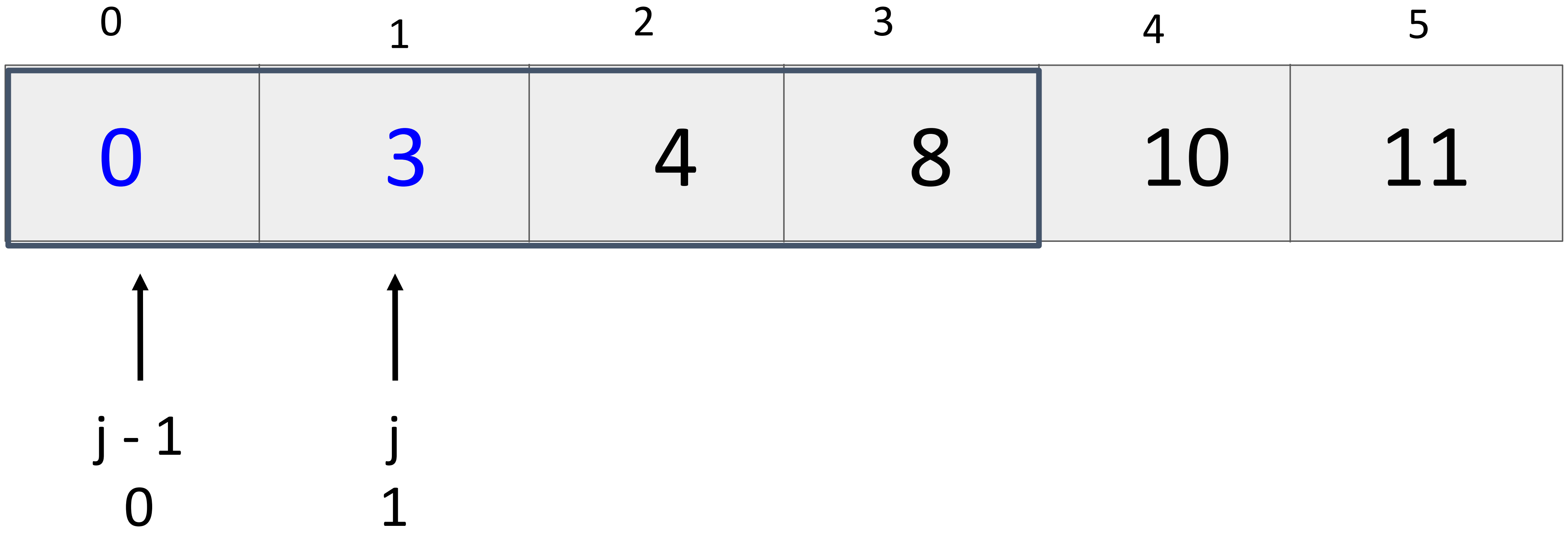


Reset and check pairs with shorter list

What next?

Bubble Sort

len = 4

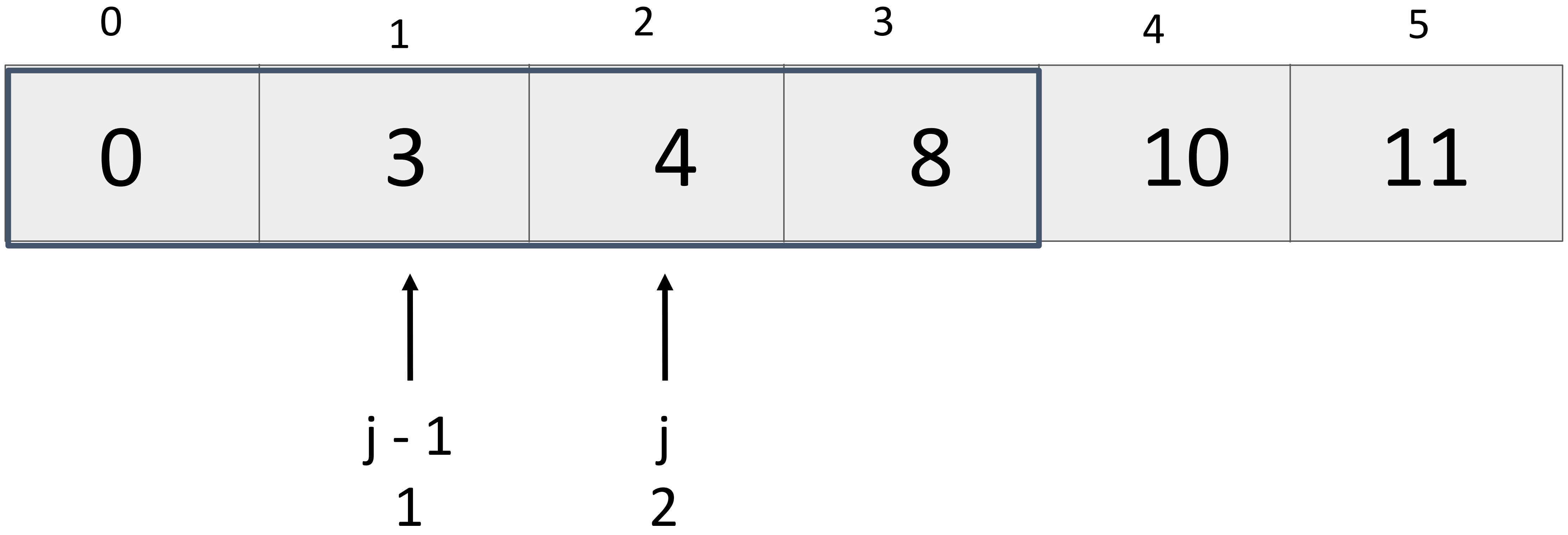


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 4

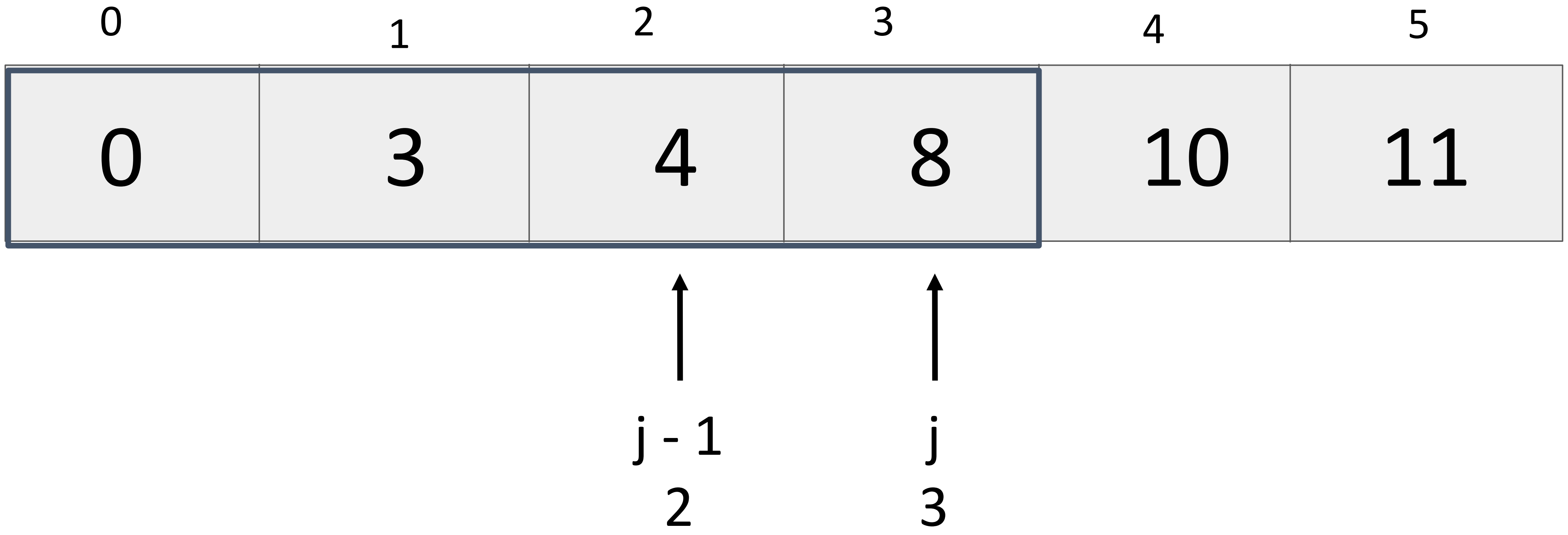


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 4

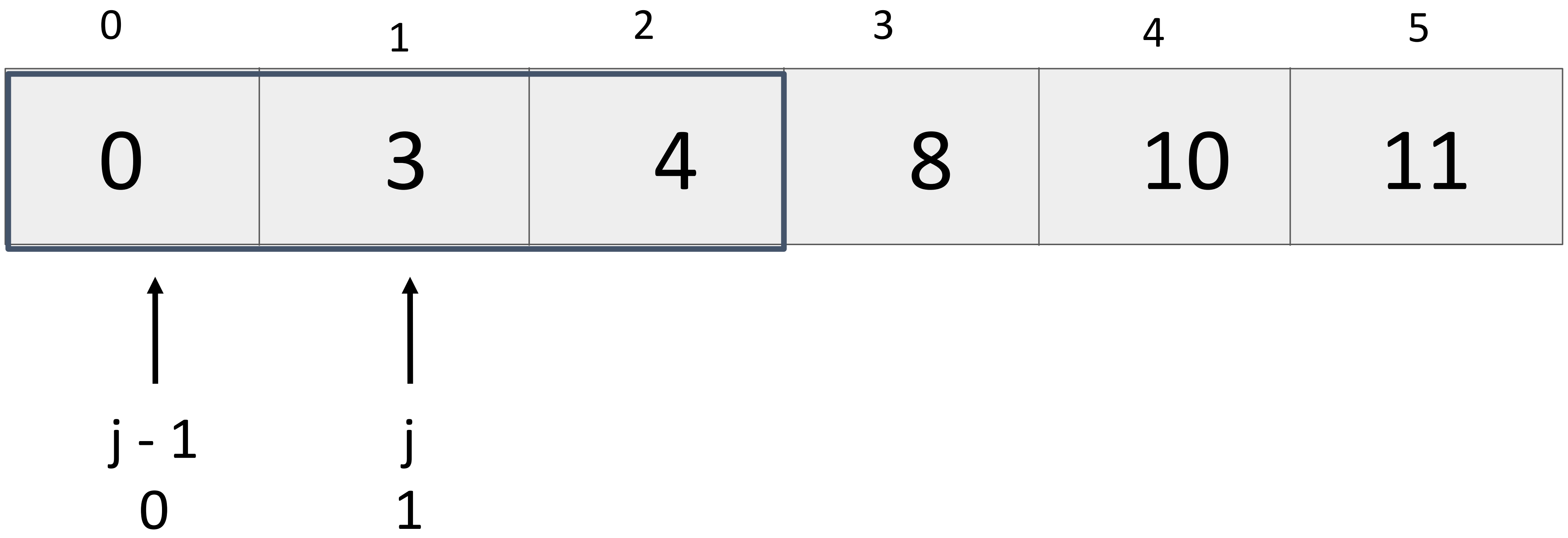


Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 3

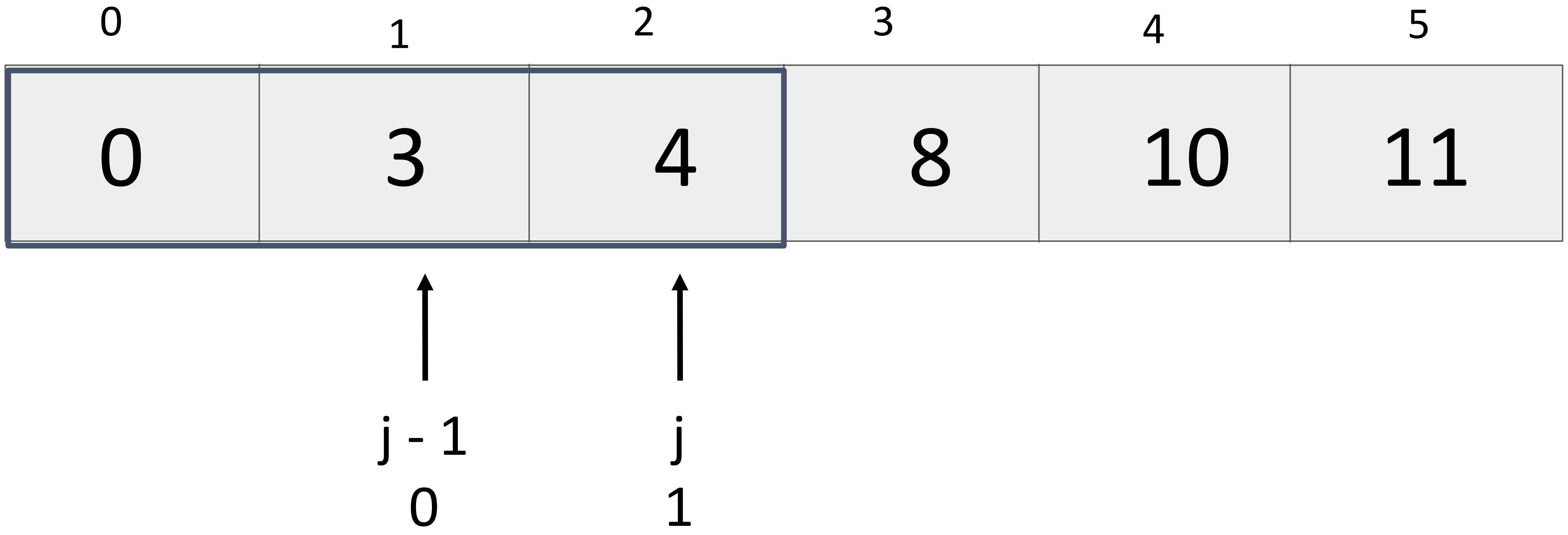


Reset; Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 3

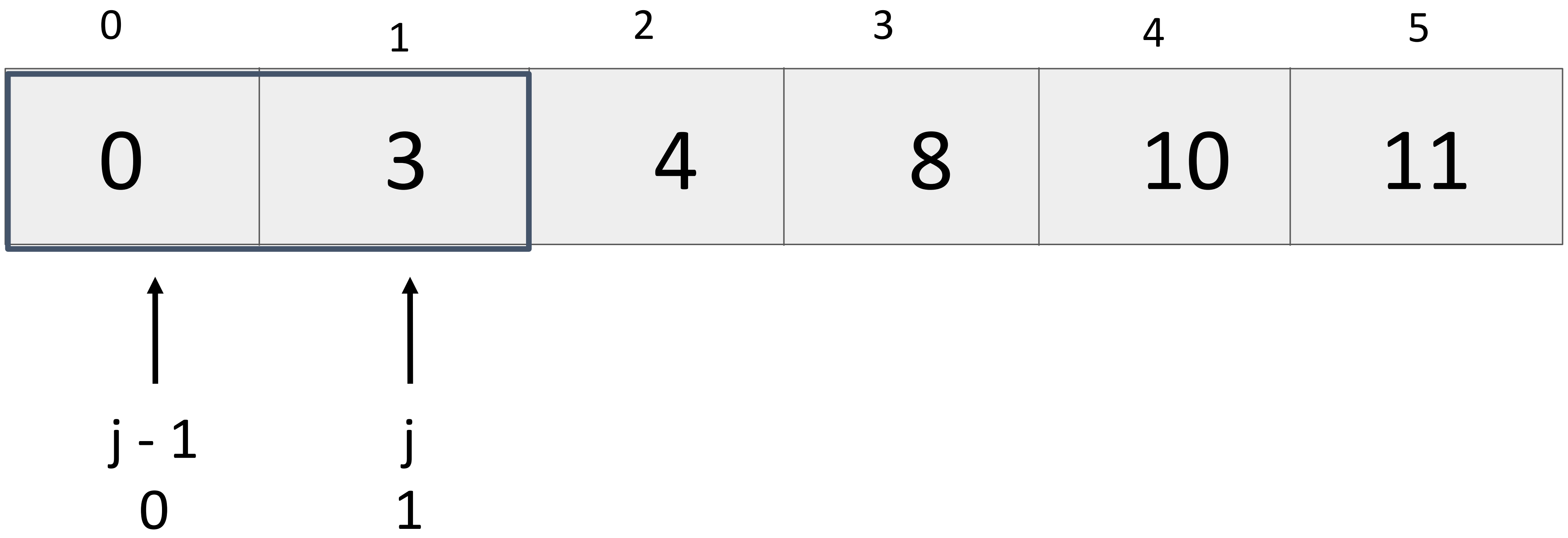


Reset; Compare j-1 and j; Swap if L[j-1] > L[j]

What next?

Bubble Sort

len = 2



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

What next?

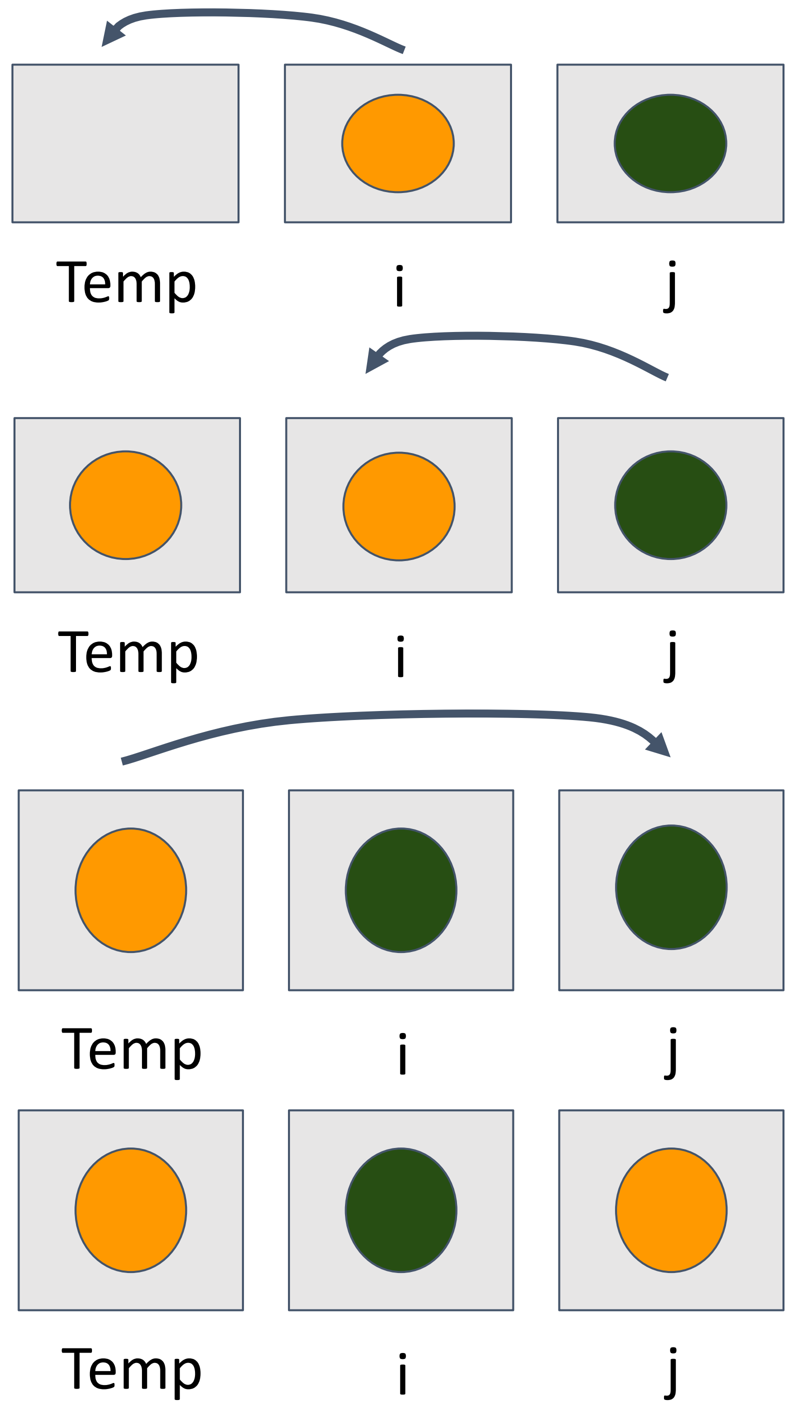
Bubble Sort

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

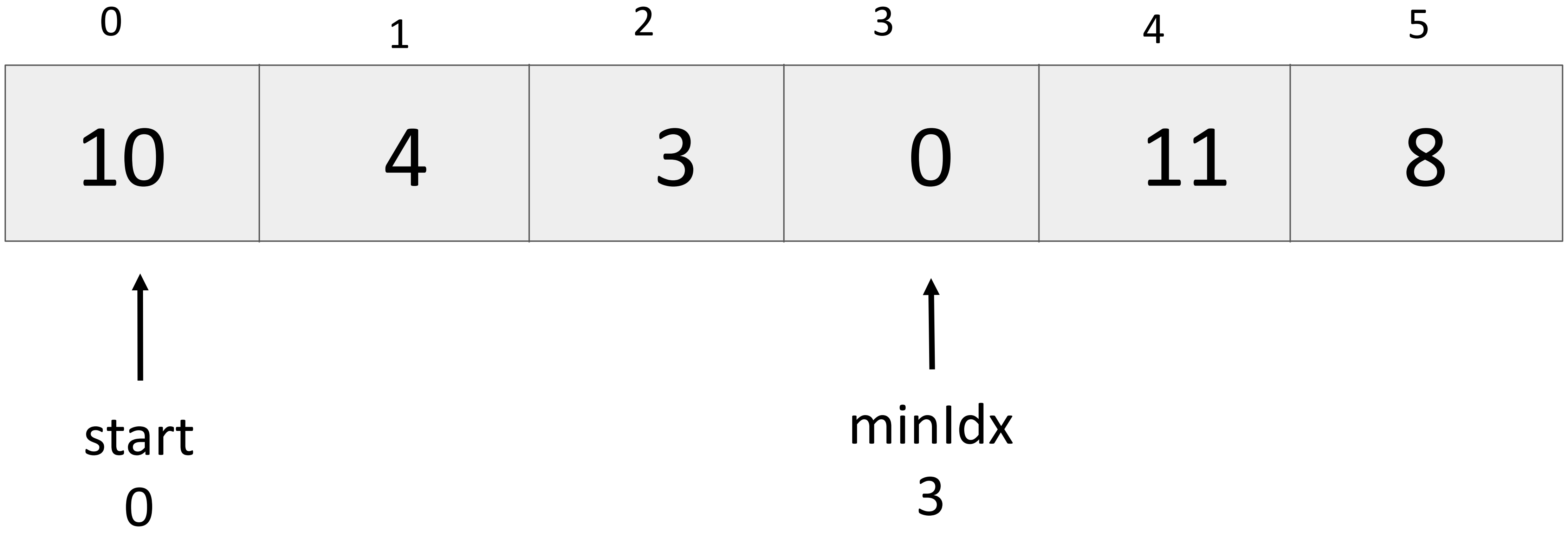
```
selectionSort(L):
    for startIdx in range(len(L)):
        minIdx = findMinimum(startIdx, L)
        swap(startIdx, minIdx, L)
```

Selection Sort

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

What do we do first?

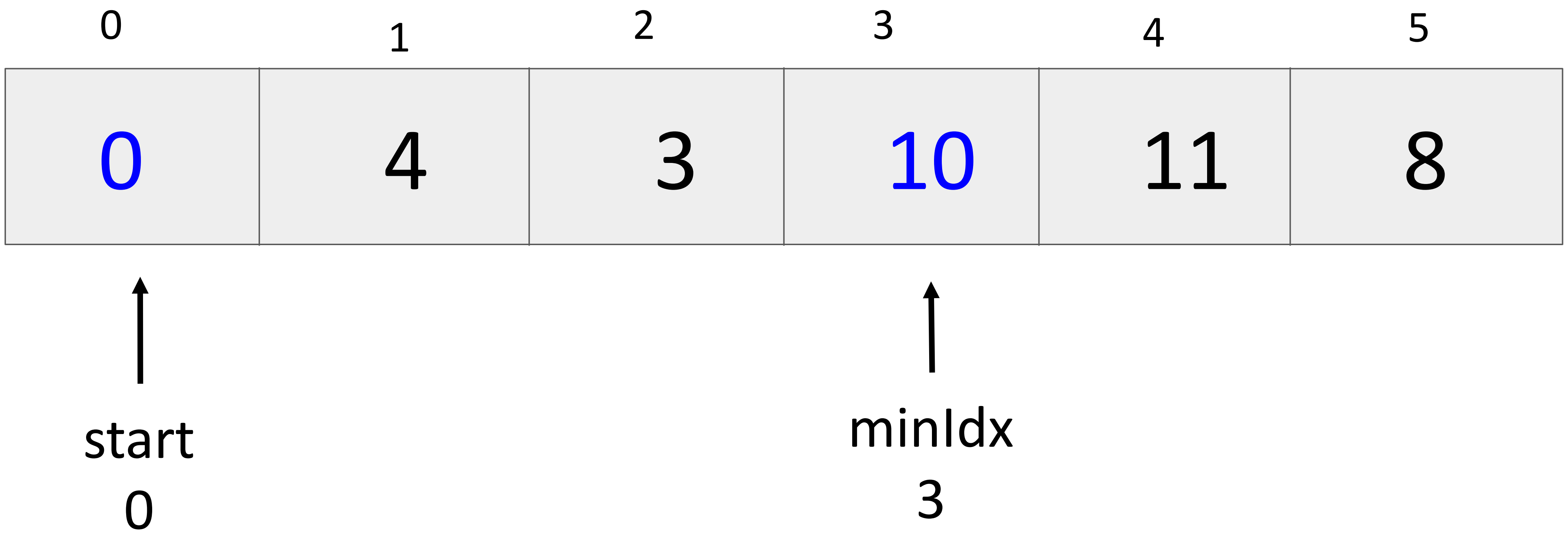
Selection Sort



Find minimum element idx between start to end

What next?

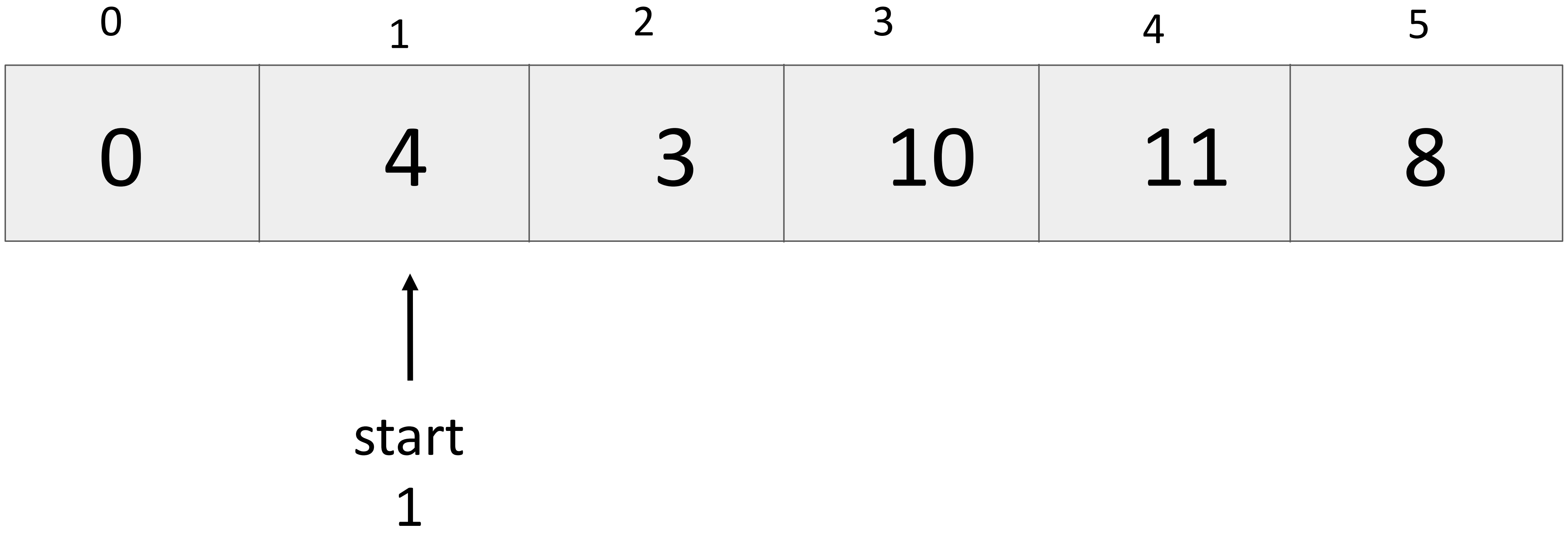
Selection Sort



Swap the elements at start and minIdx

What next?

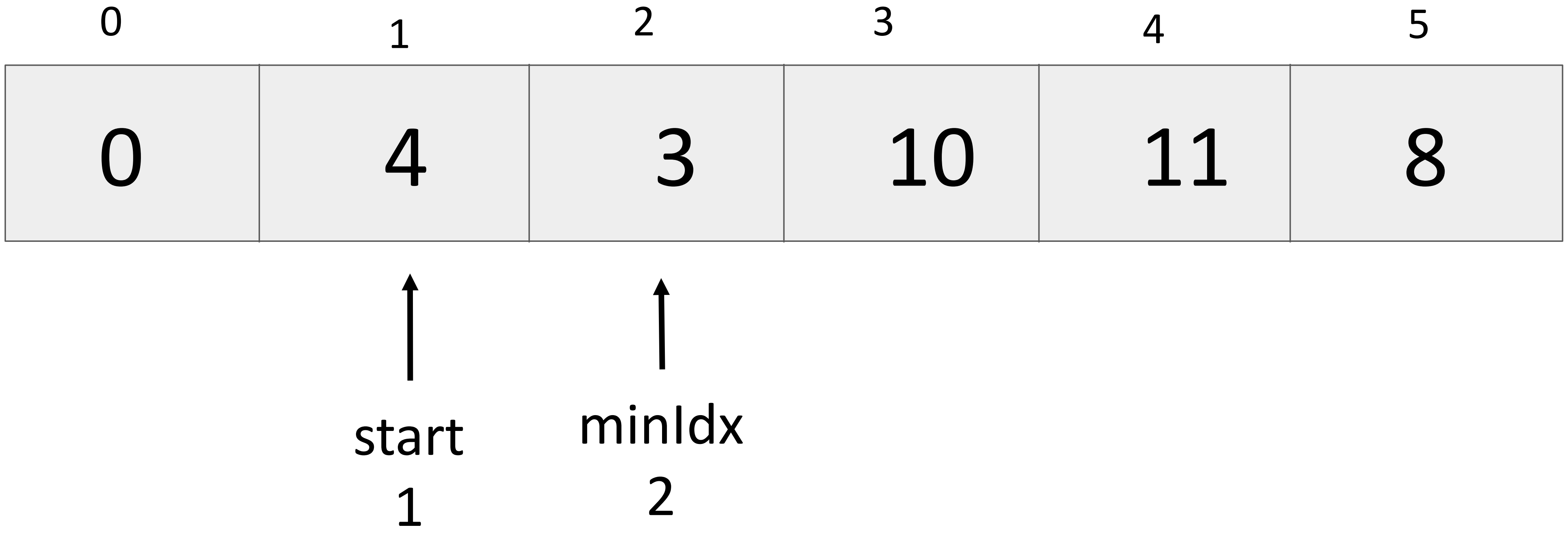
Selection Sort



Decrease the interval.

What next?

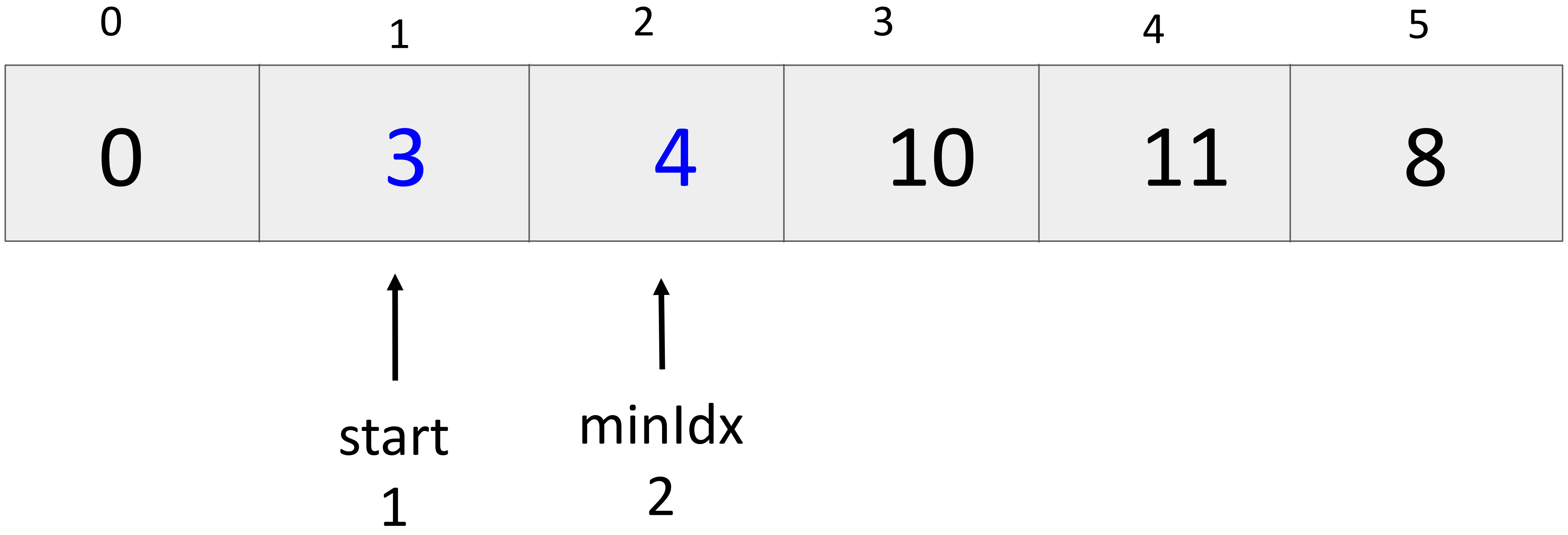
Selection Sort



Find minimum element between start to end

What next?

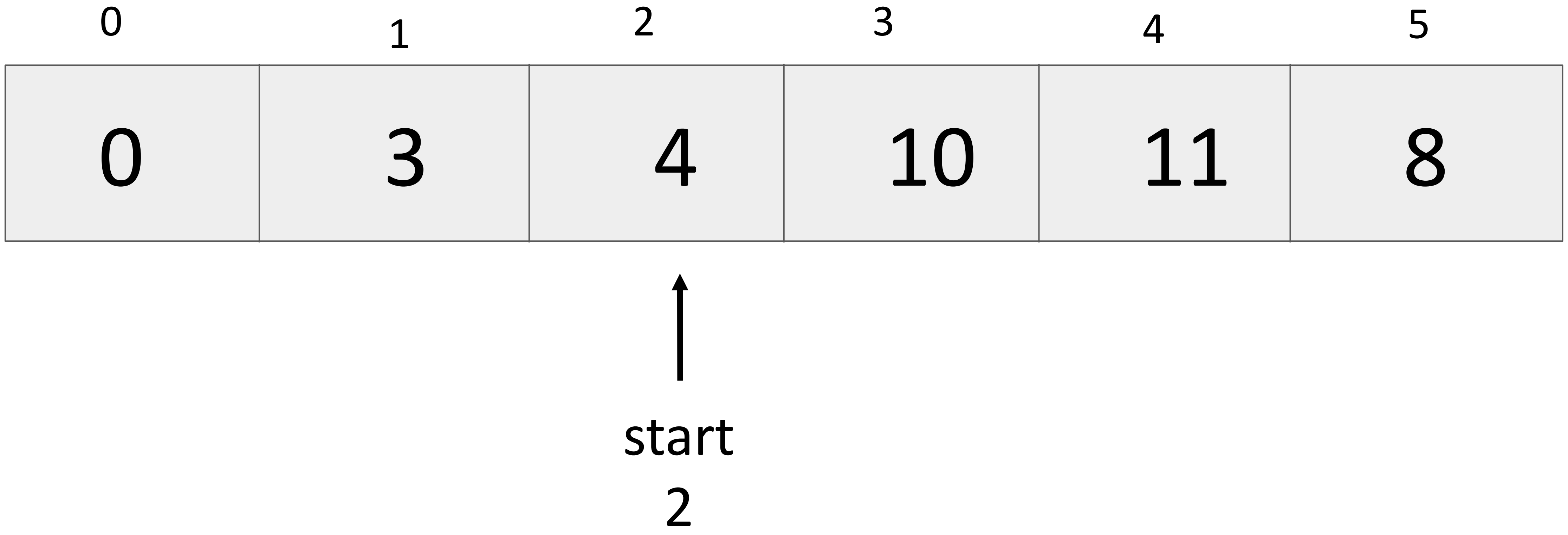
Selection Sort



Swap the elements at start and minIdx

What next?

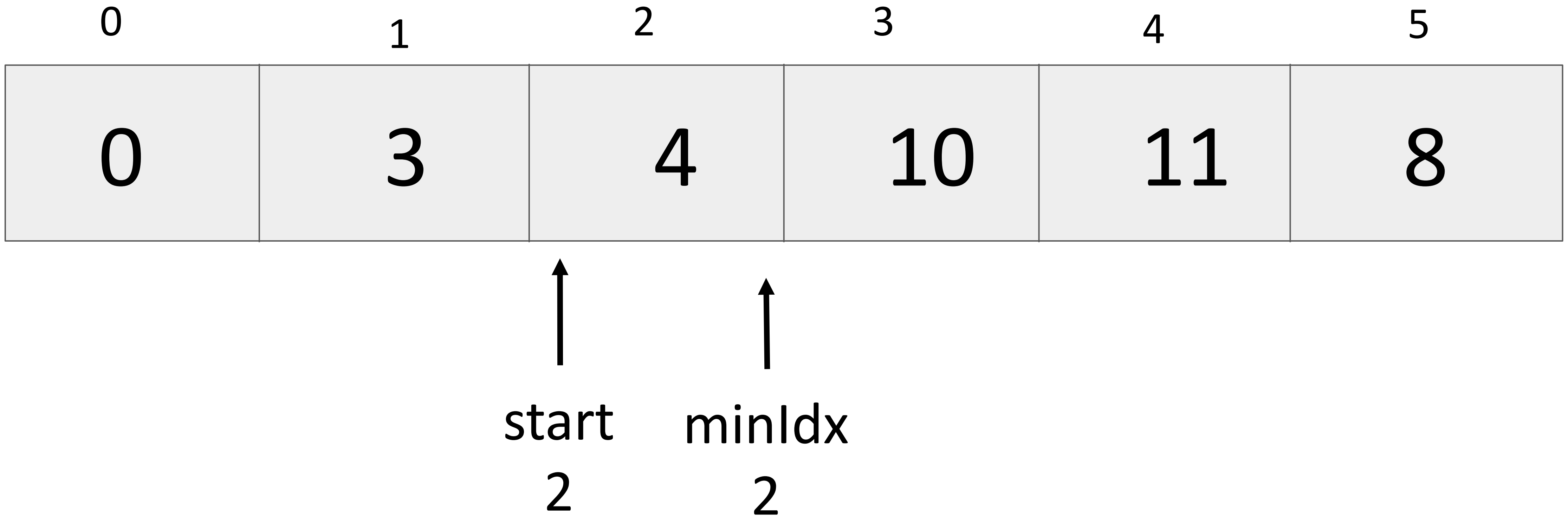
Selection Sort



Decrease the interval.

What next?

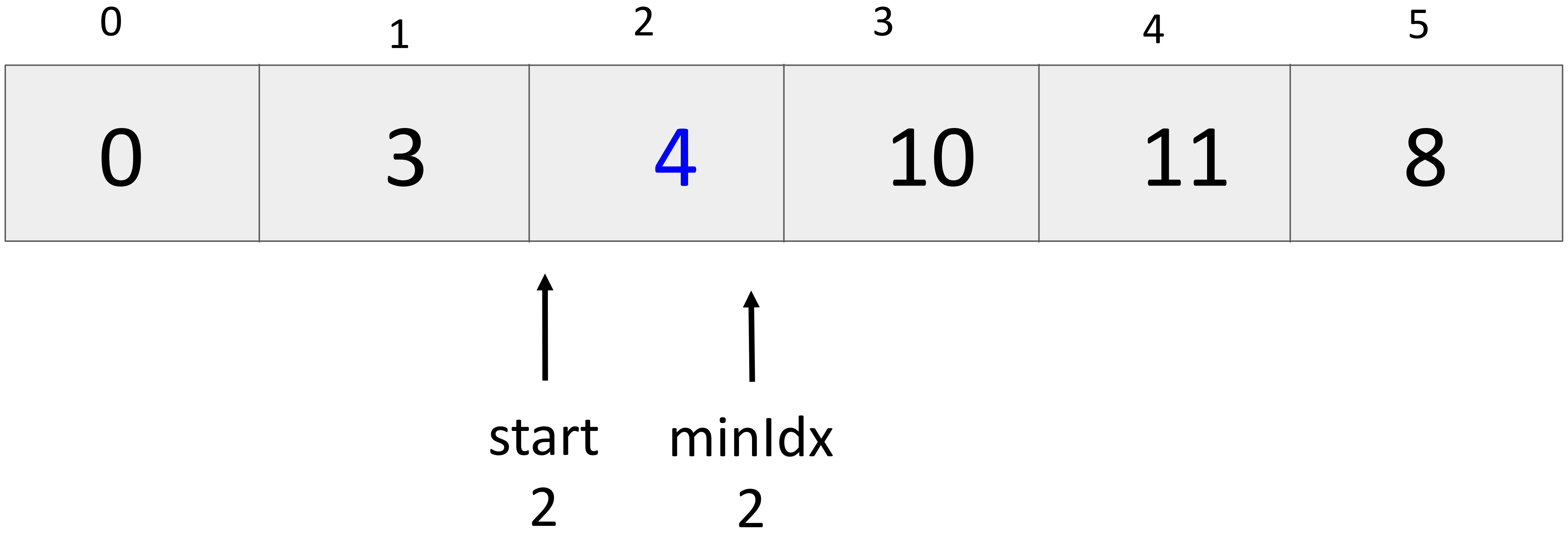
Selection Sort



Find minimum element idx between start to end

What next?

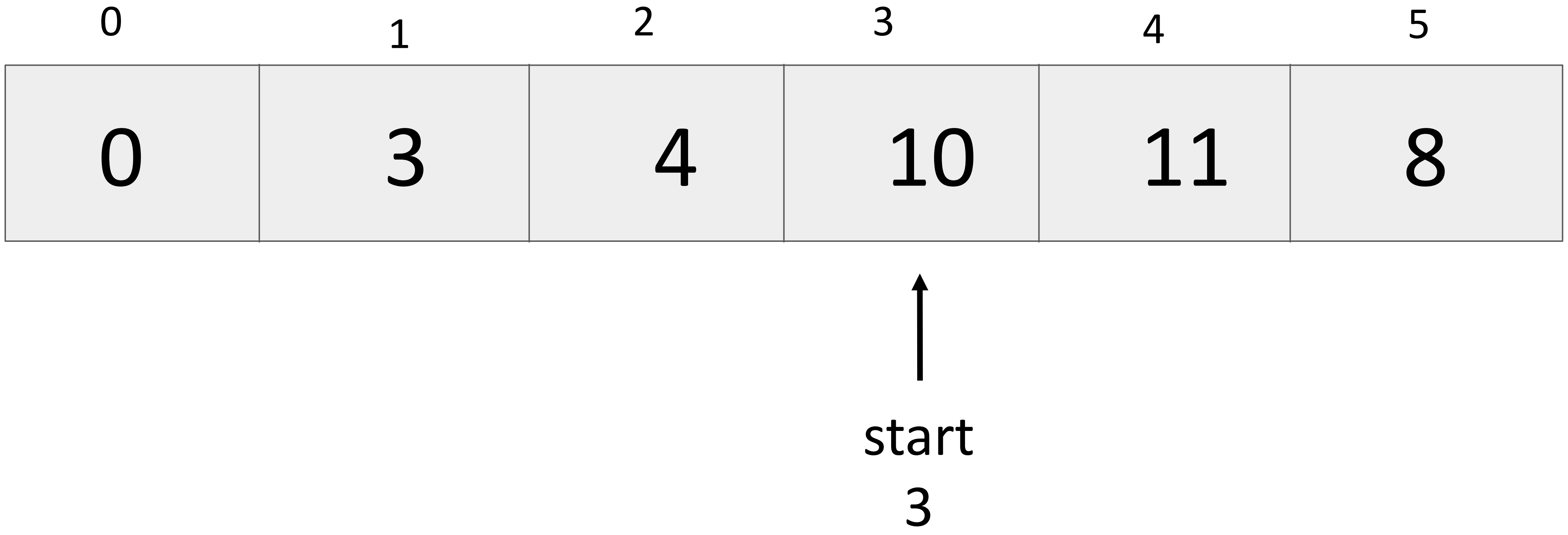
Selection Sort



Swap the elements at start and minIdx

What next?

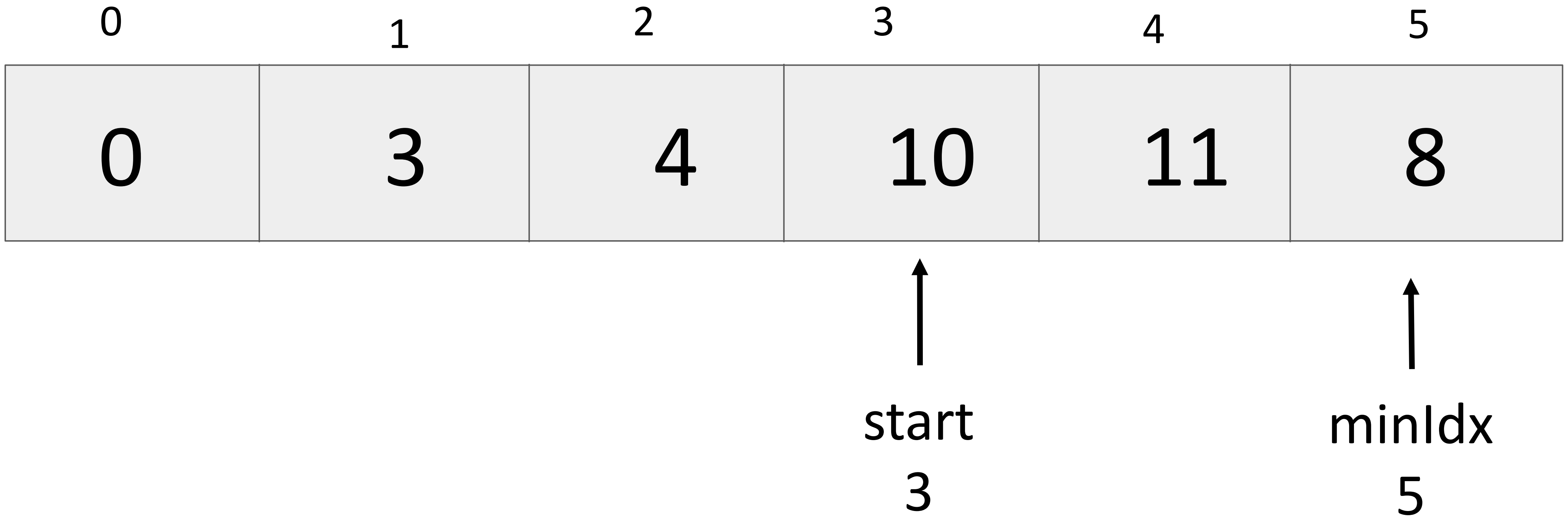
Selection Sort



Decrease the interval.

What next?

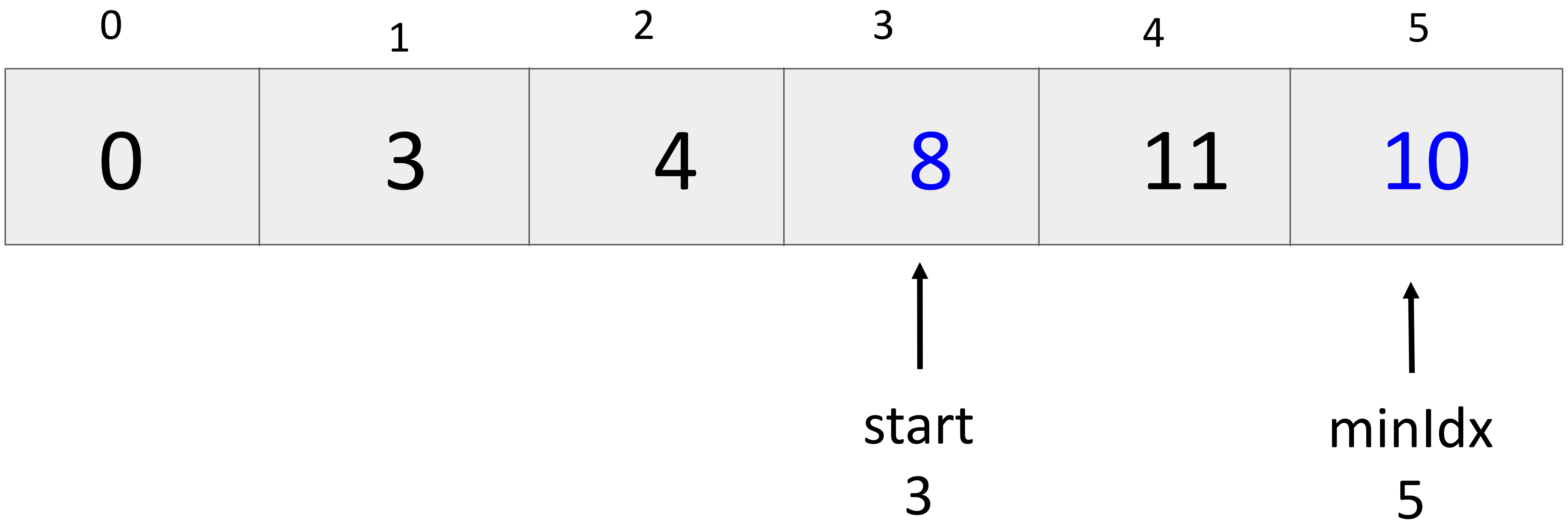
Selection Sort



Find minimum element idx between start to end

What next?

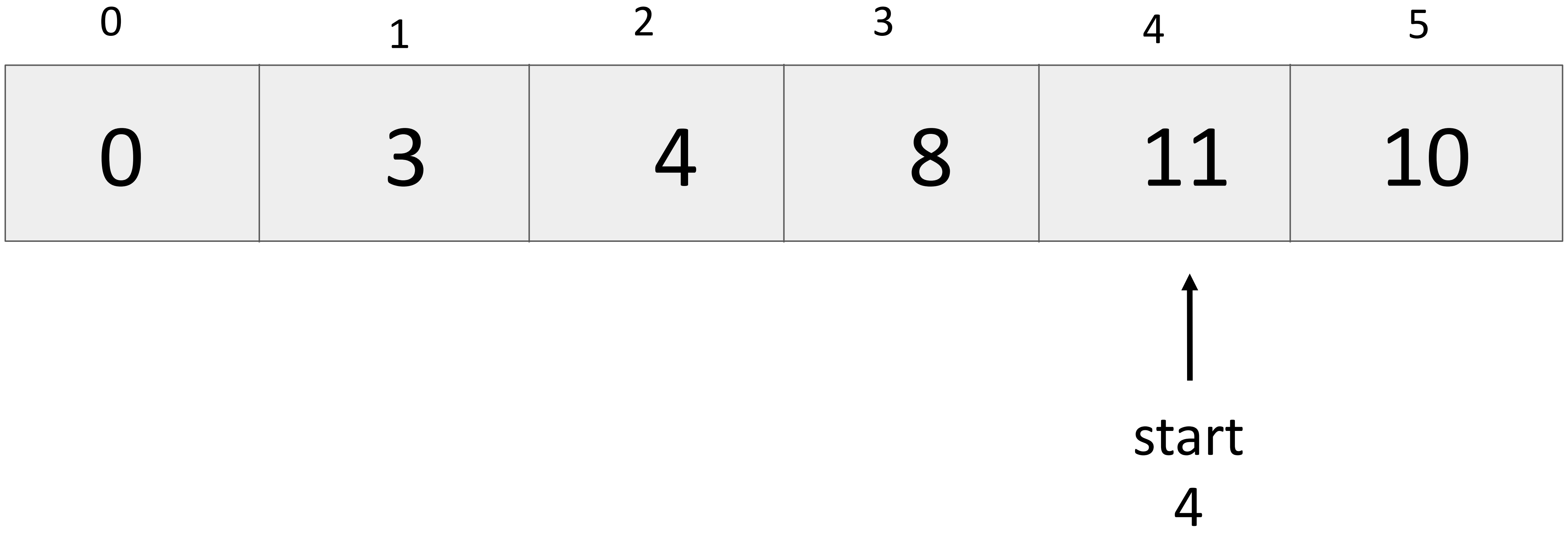
Selection Sort



Swap the elements at start and minIdx

What next?

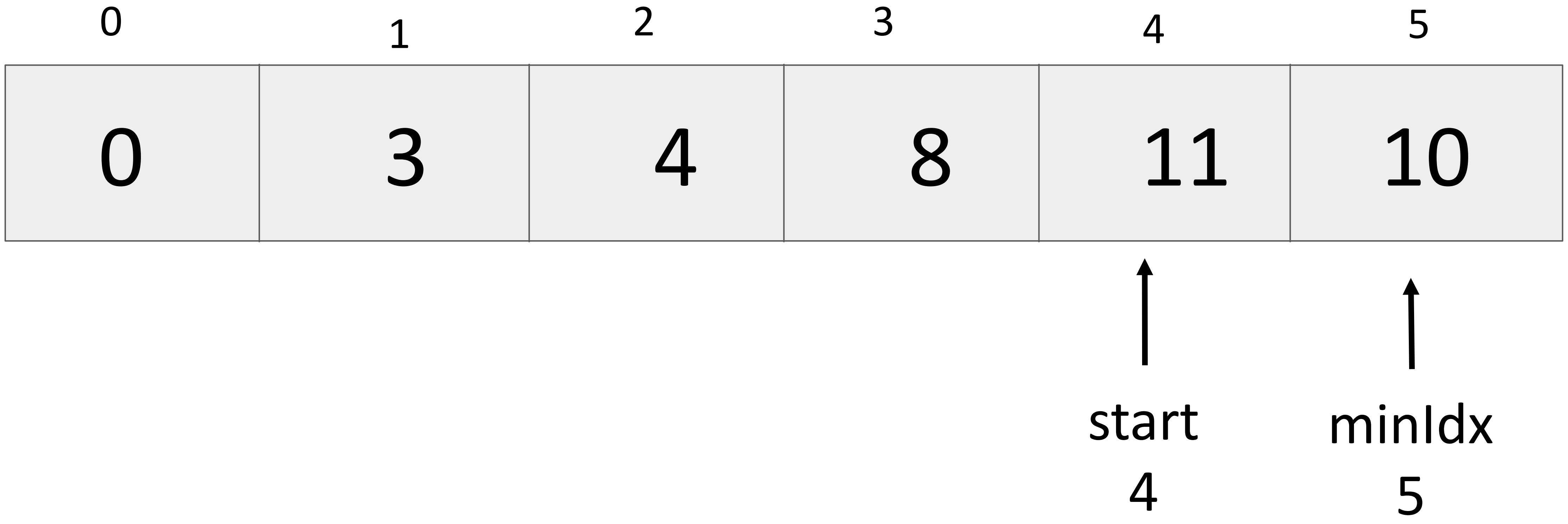
Selection Sort



Decrease the interval.

What next?

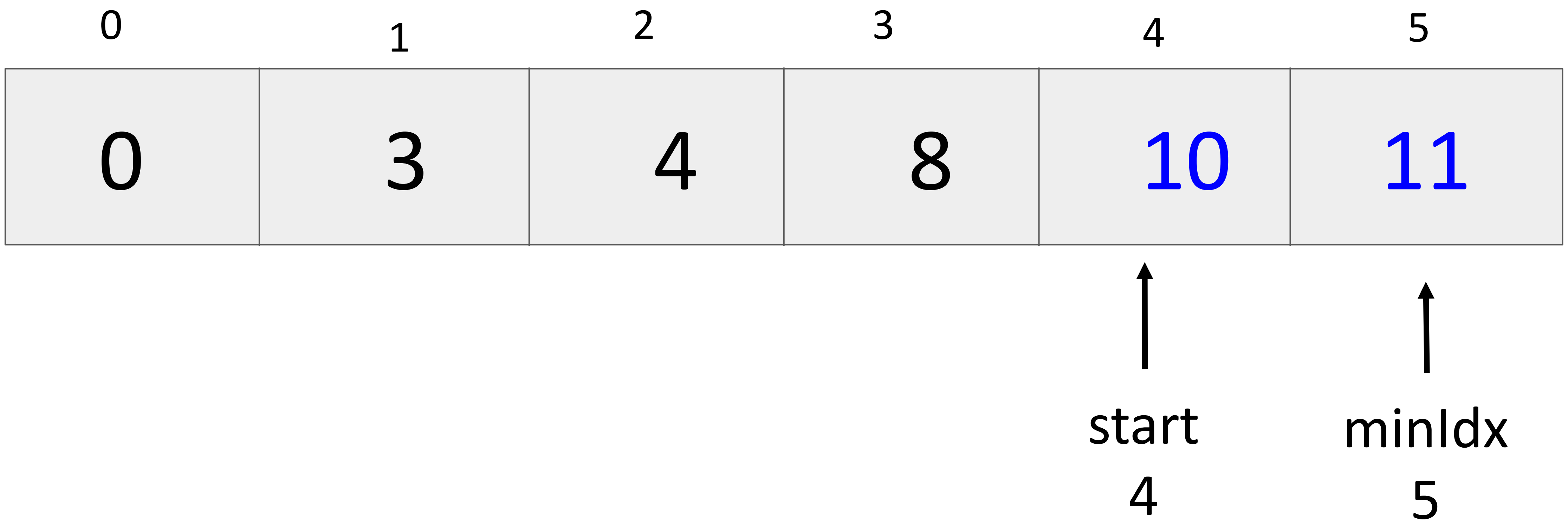
Selection Sort



Find minimum element idx between start to end

What next?

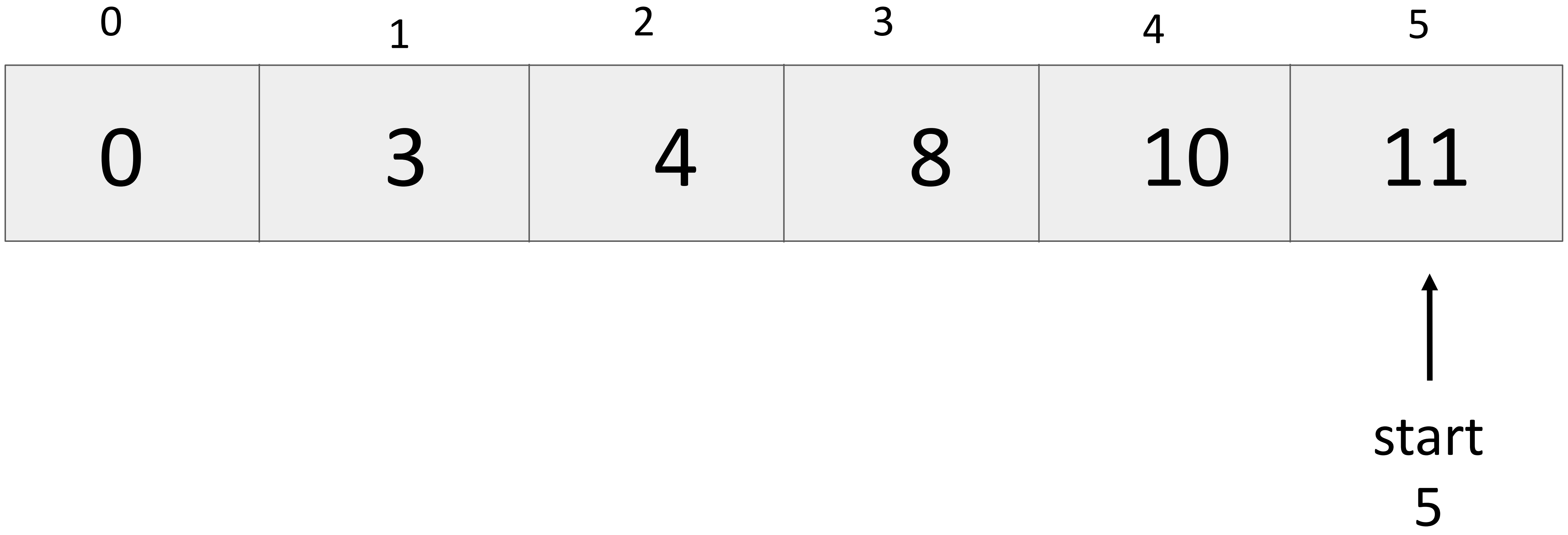
Selection Sort



Swap the elements at start and minIdx

What next?

Selection Sort



Decrease the interval.

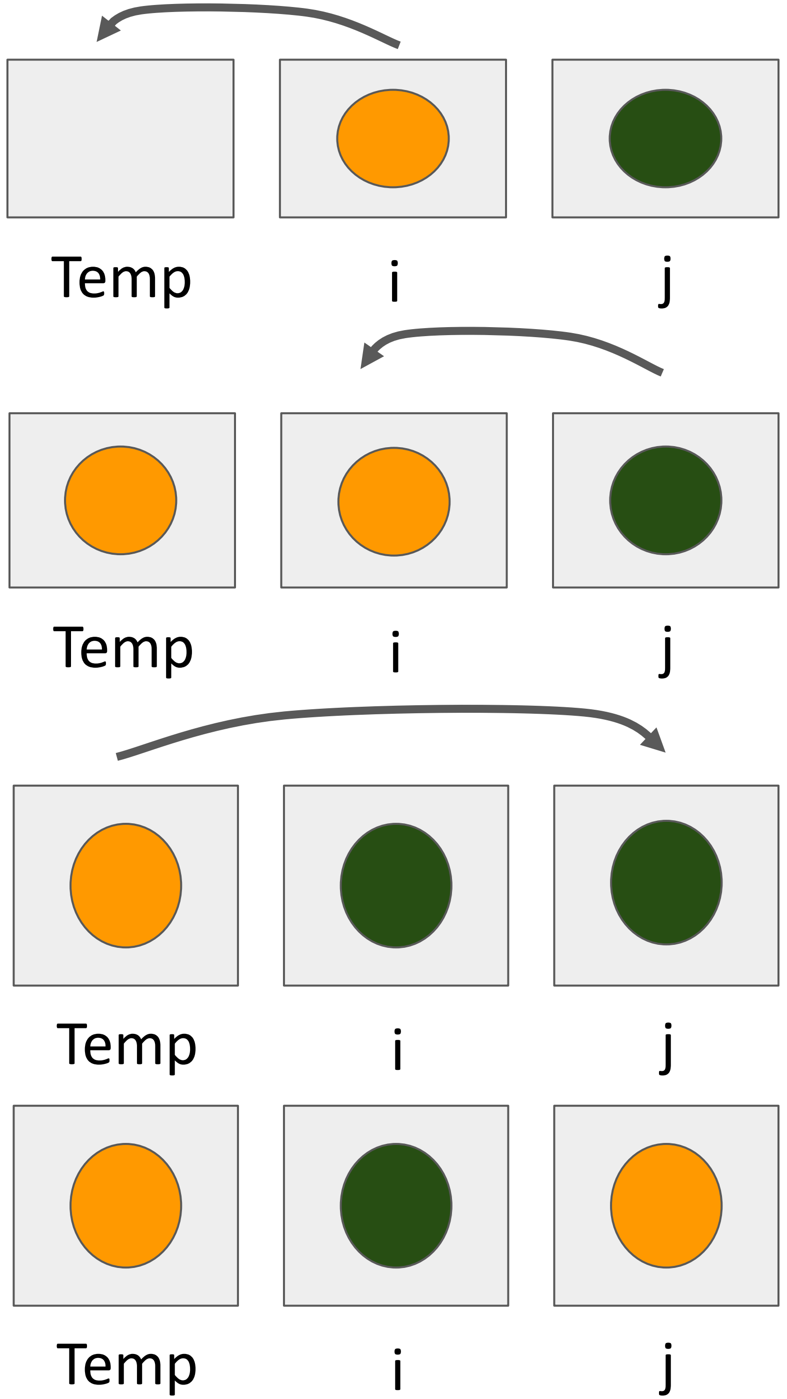
We're done!

Selection sort

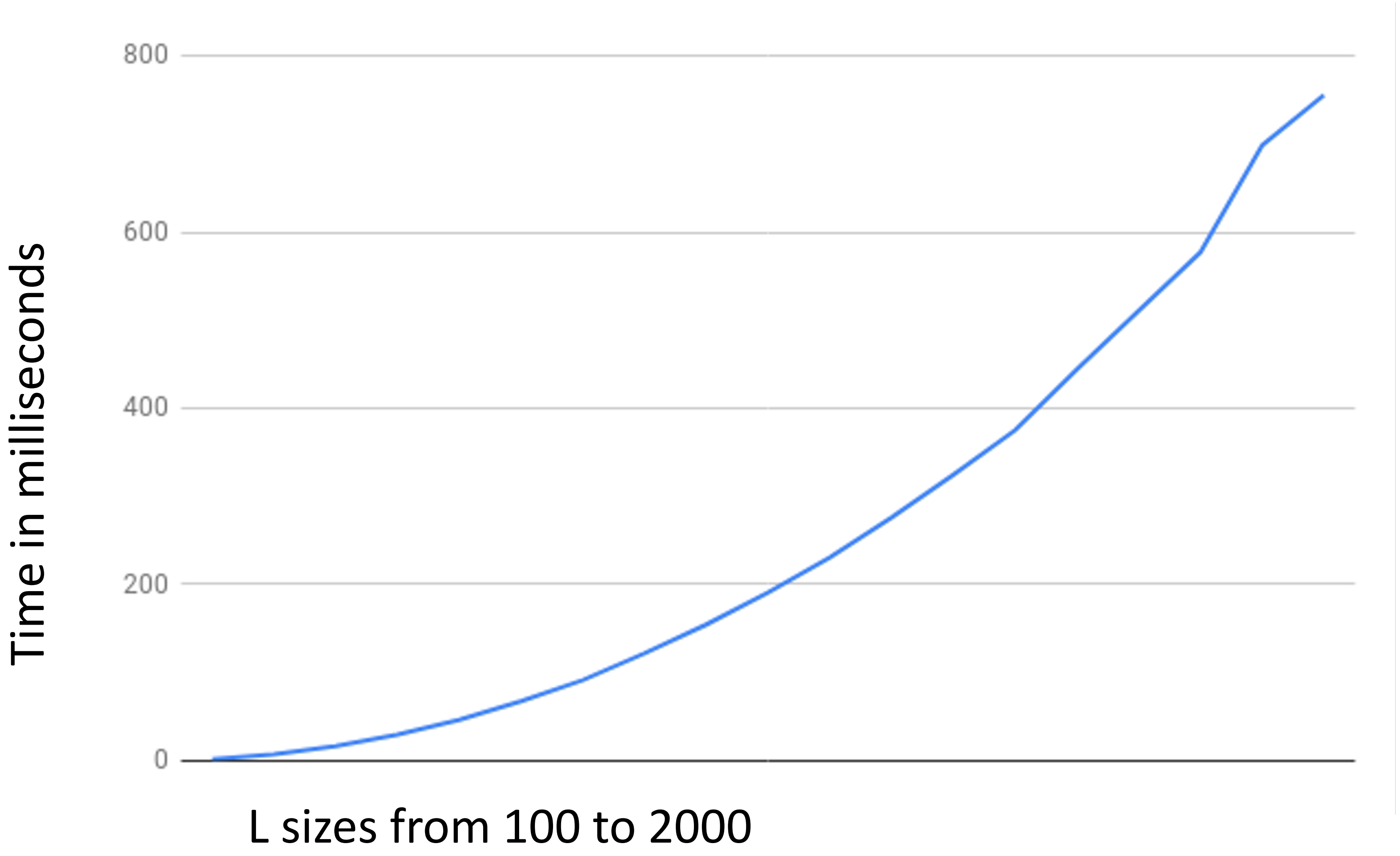
```
findMinimum(startIdx, L):
        minIdx = startIdx
    for i in range(startIdx, len(L)):
        if L[i] < L[minIdx]:
                minIdx = i
    return minIdx
```

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 $O(N^2)$


Time in milliseconds
L sizes from 100 to 2000