Bubble Sort

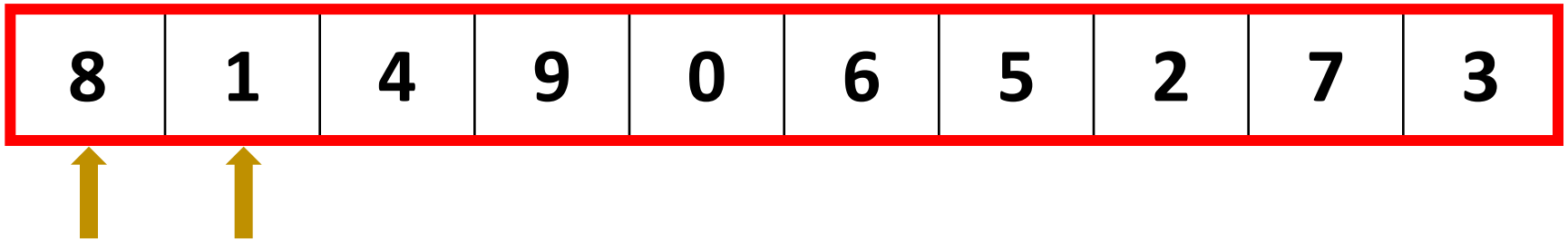
Shusen Wang

Sorting

8	1	4	9	0	6	5	2	7	3

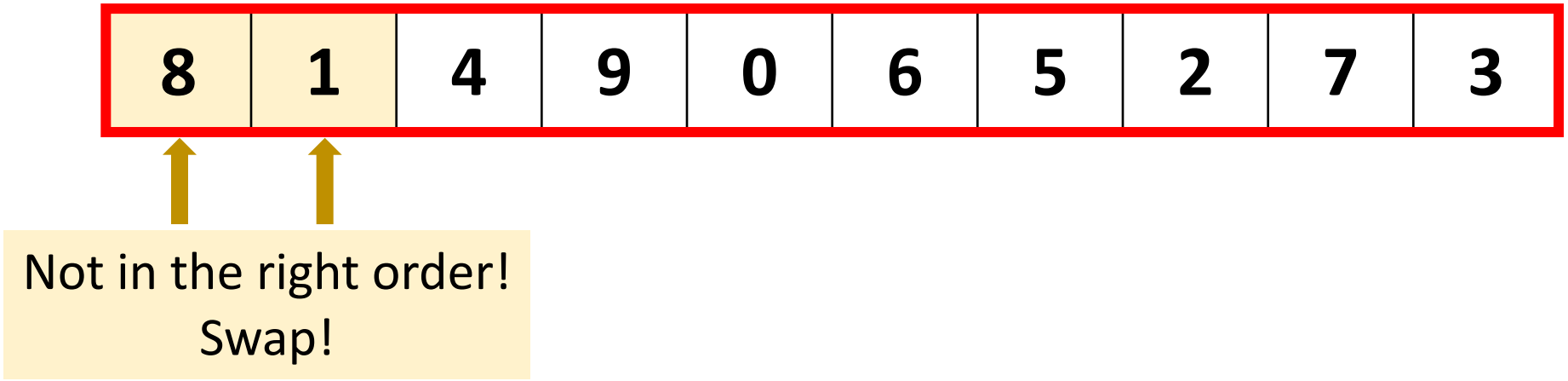
- **Input:** An array with n elements.
- **Goal:** Sort the array so that the elements are in the ascending (or descending) order.

Iteration 1



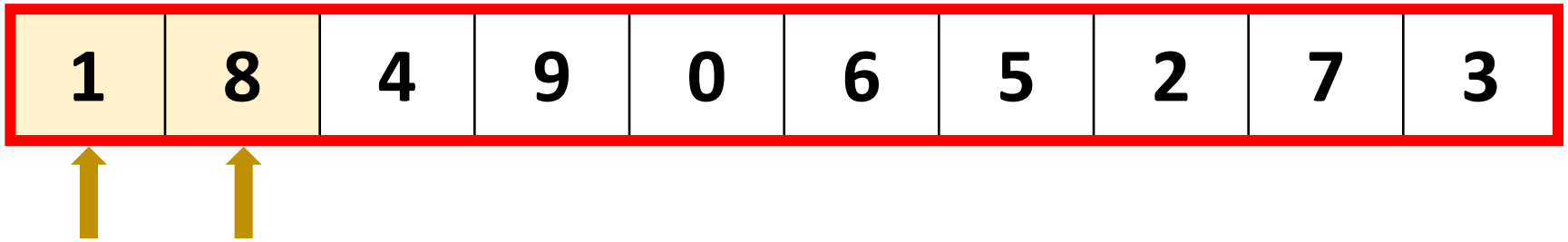
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



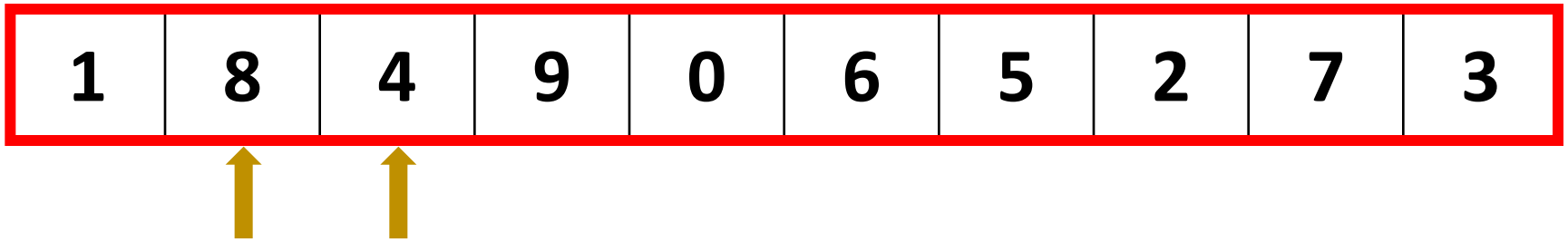
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



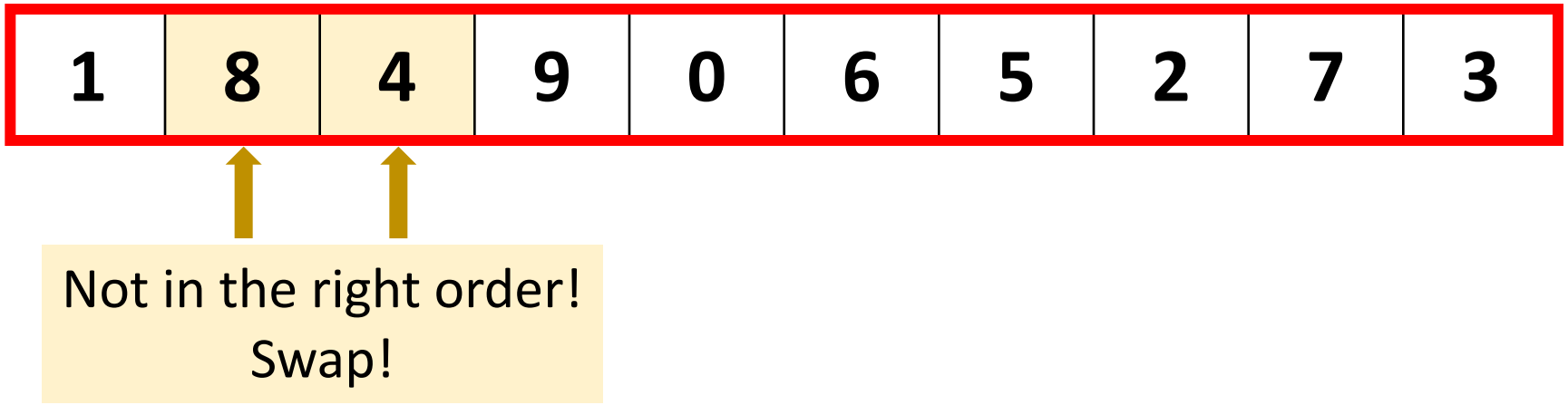
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



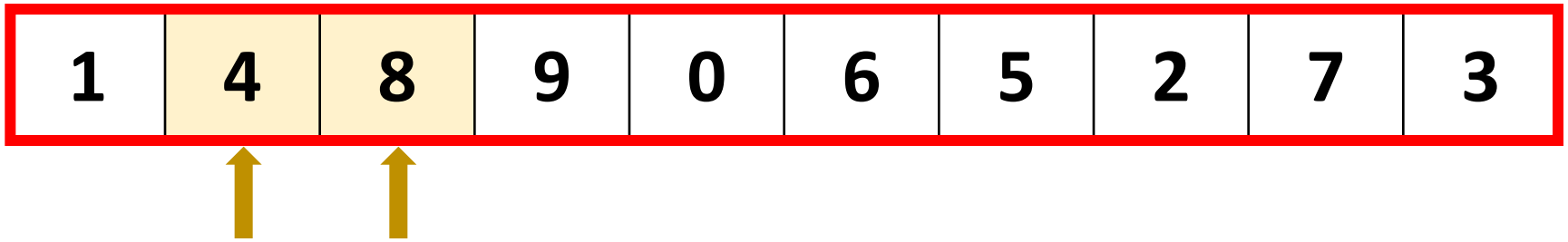
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



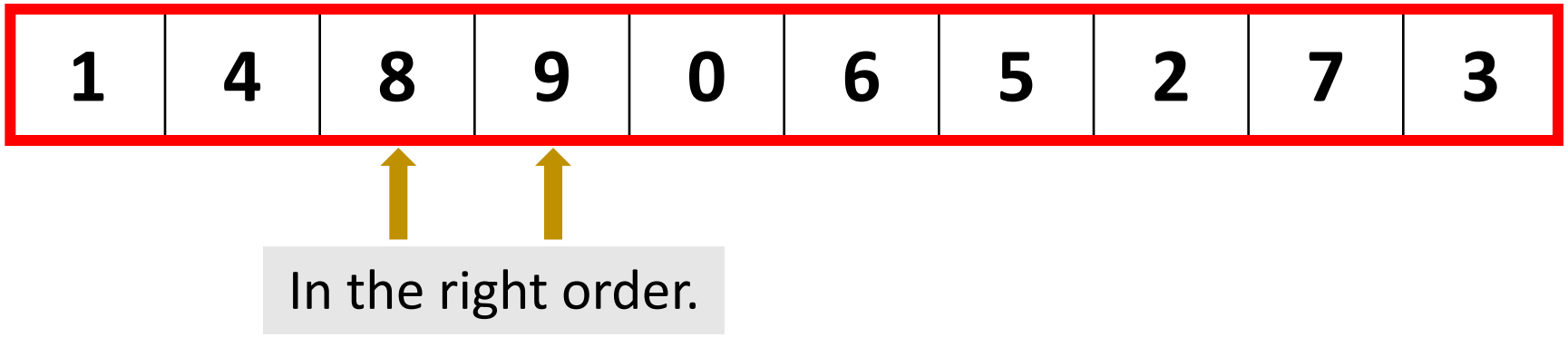
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



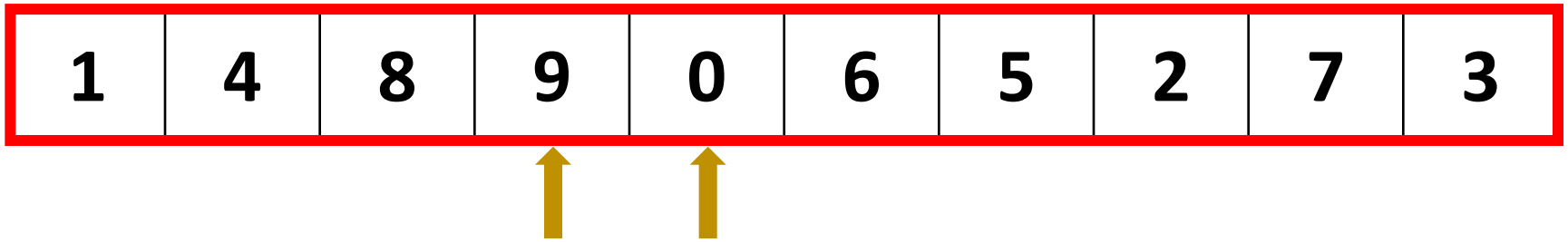
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



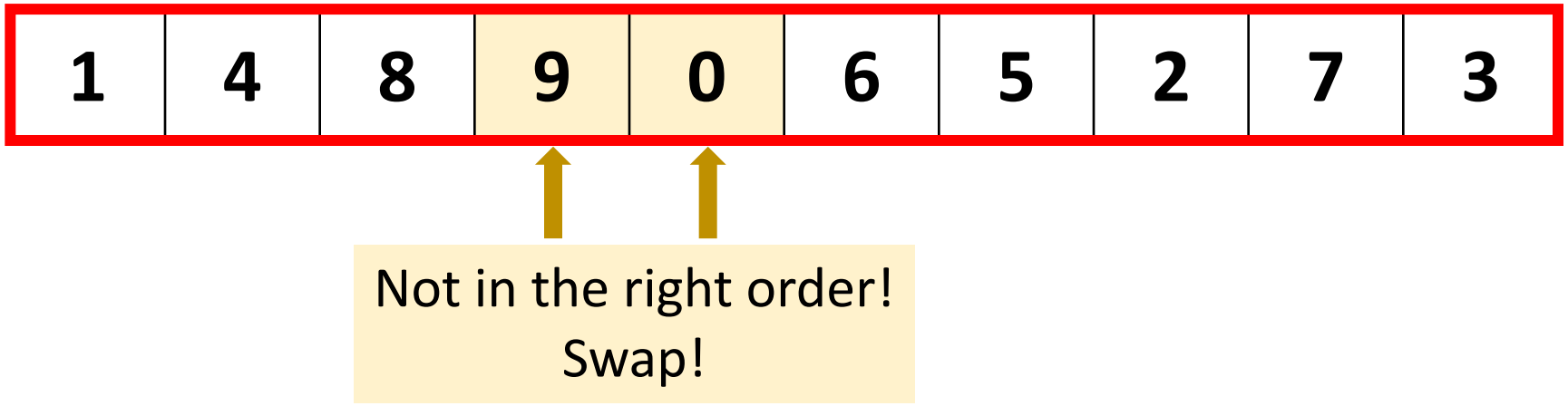
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



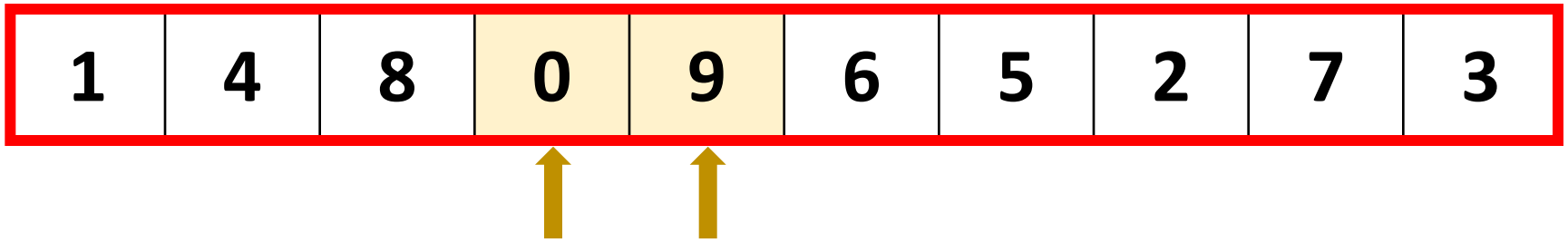
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



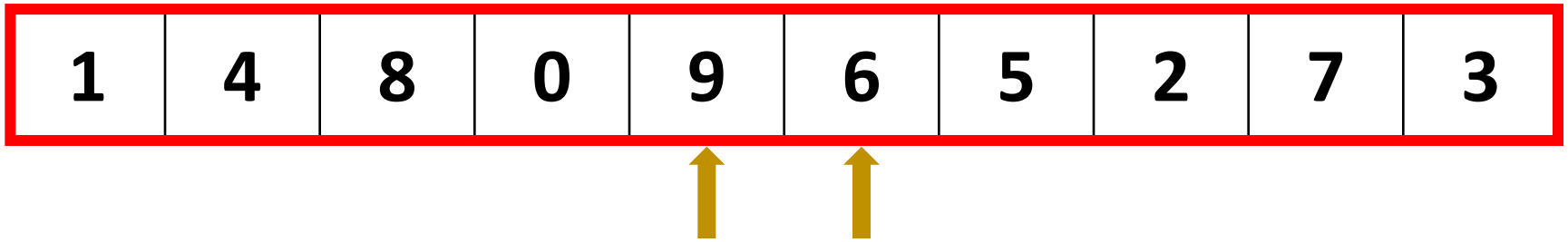
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



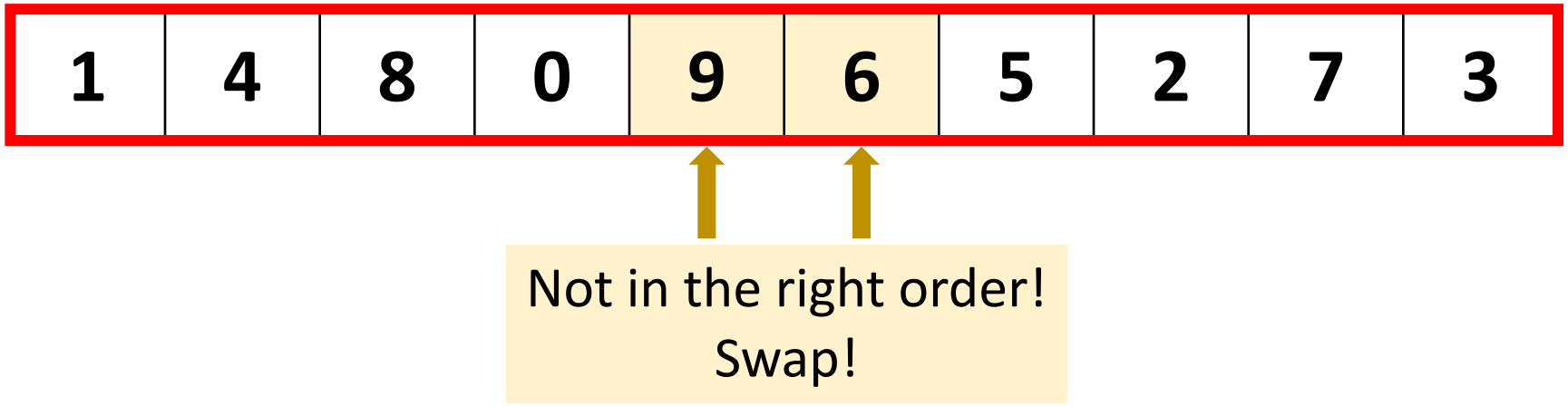
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



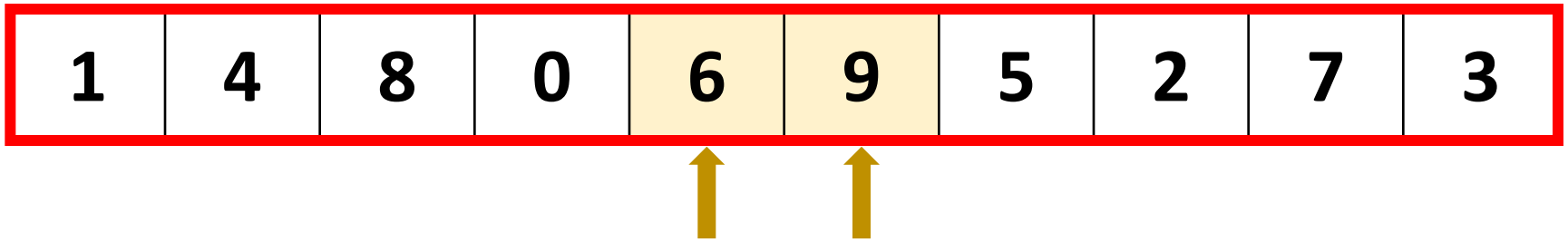
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



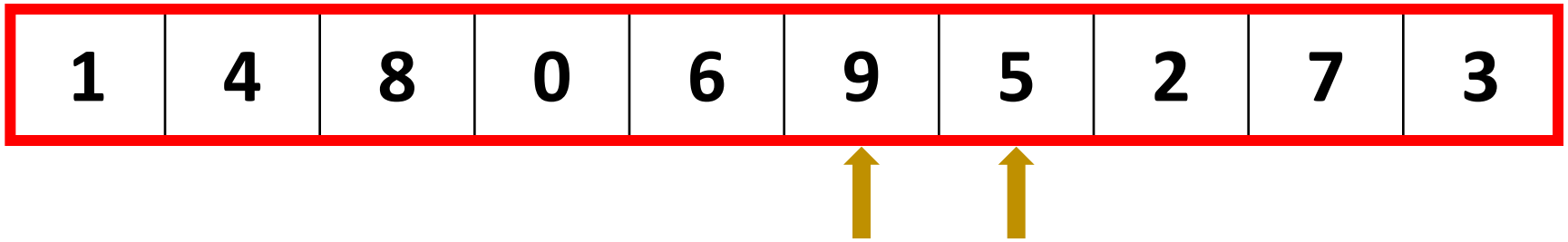
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



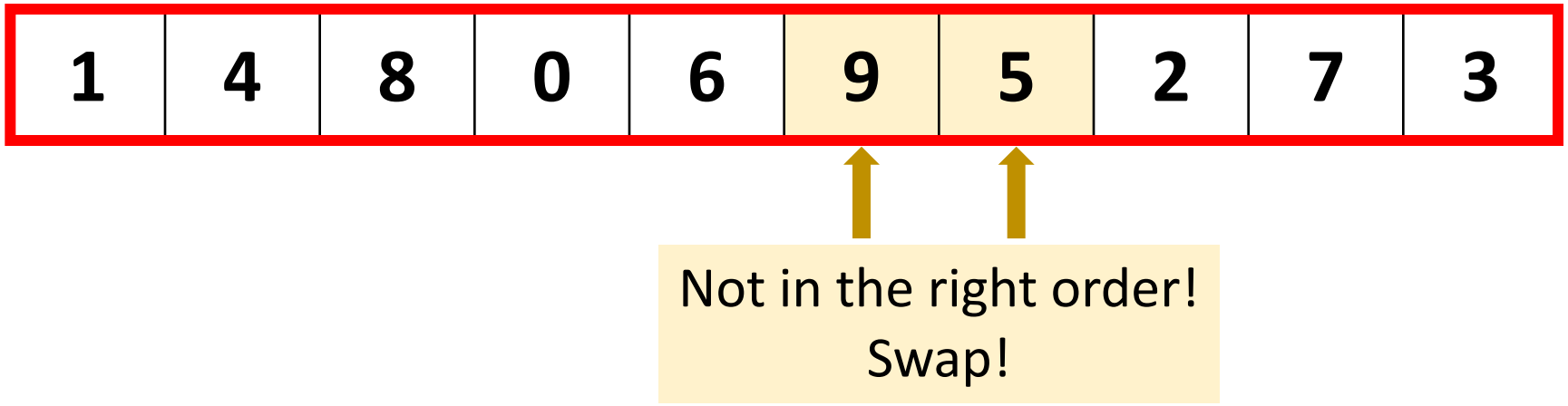
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



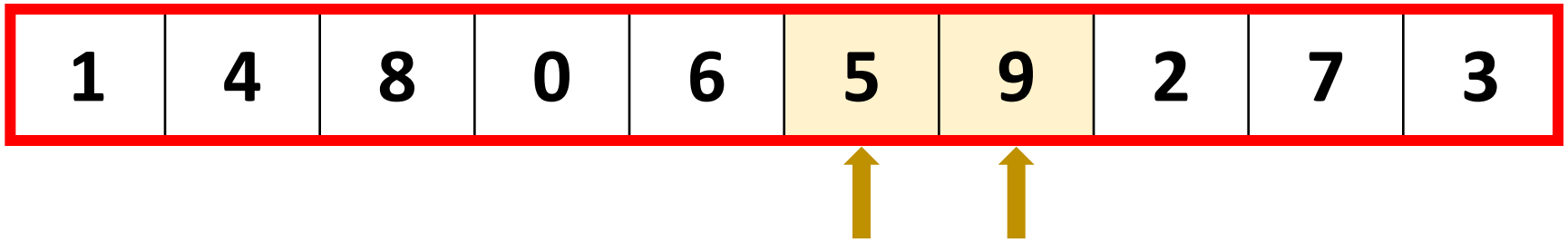
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



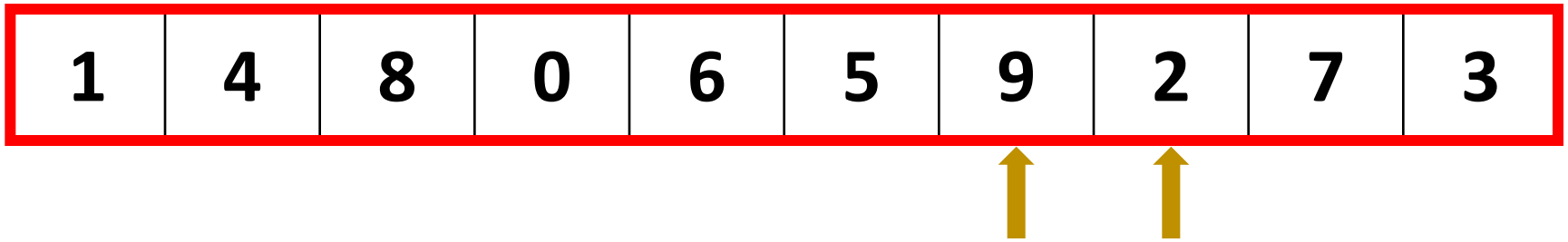
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



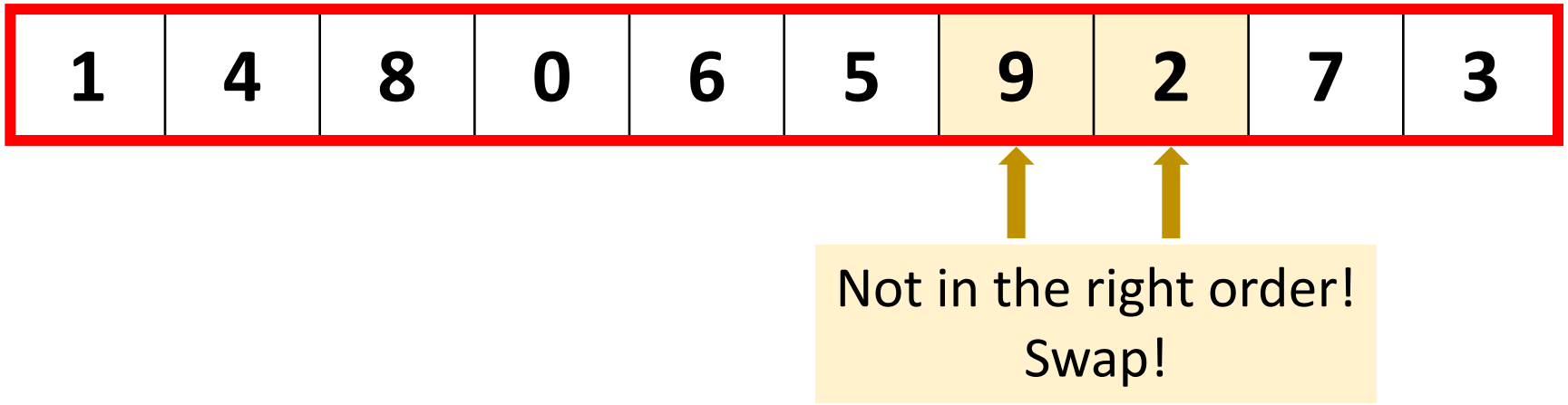
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



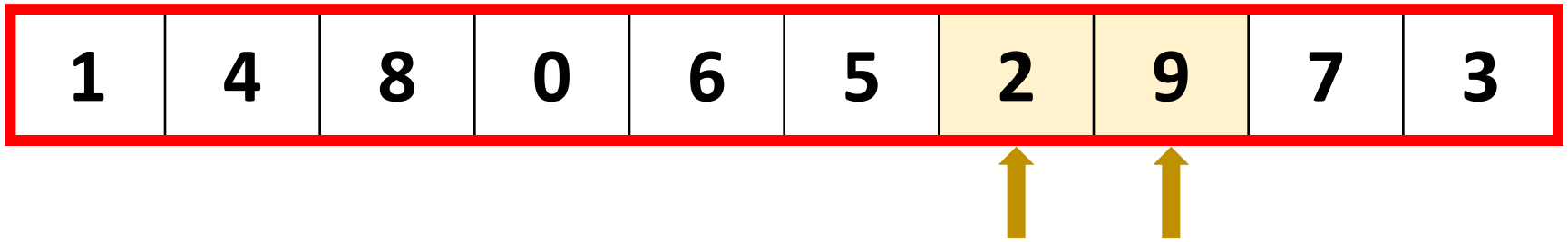
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



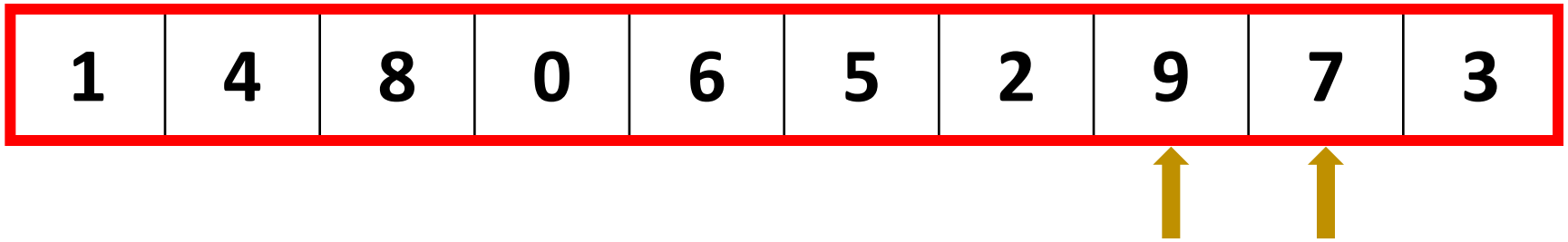
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



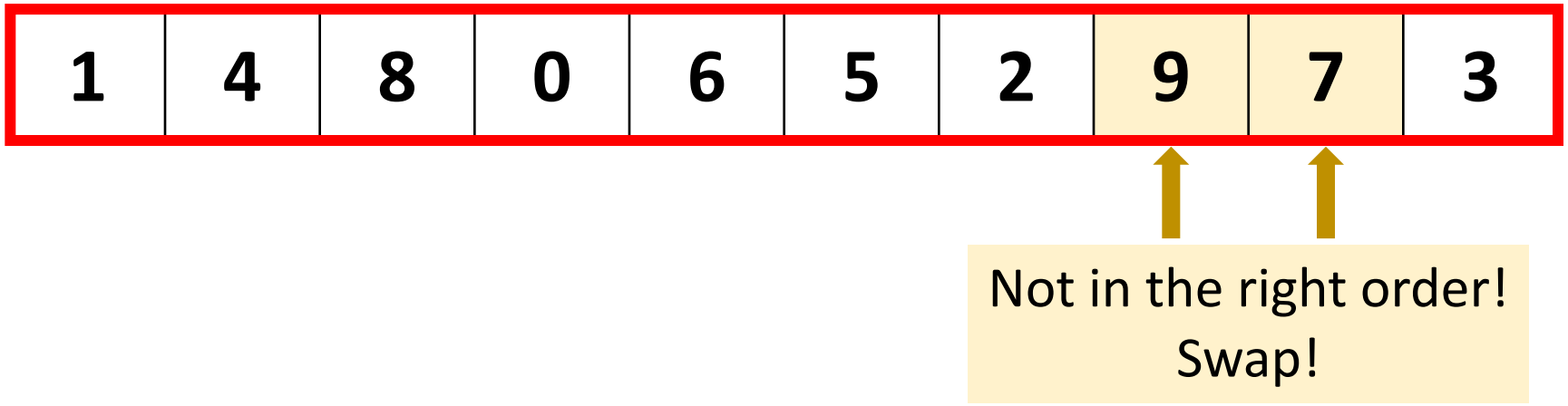
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



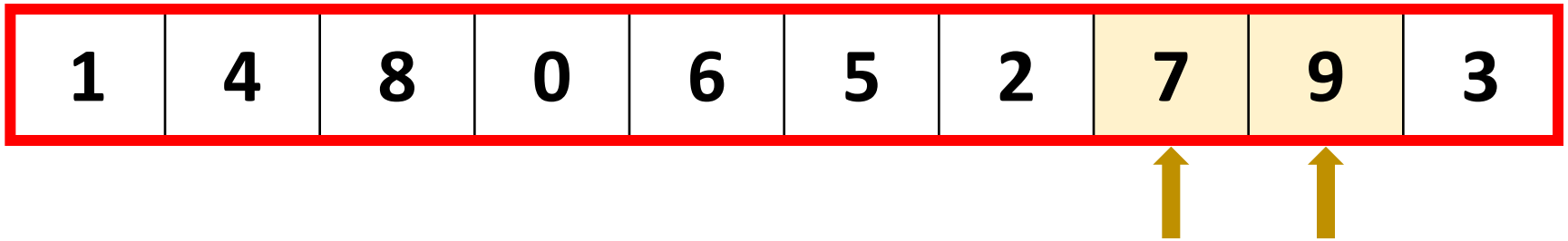
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



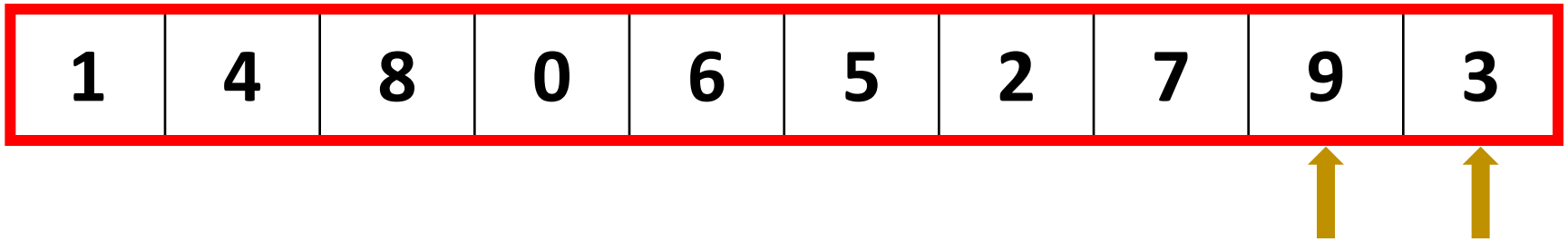
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



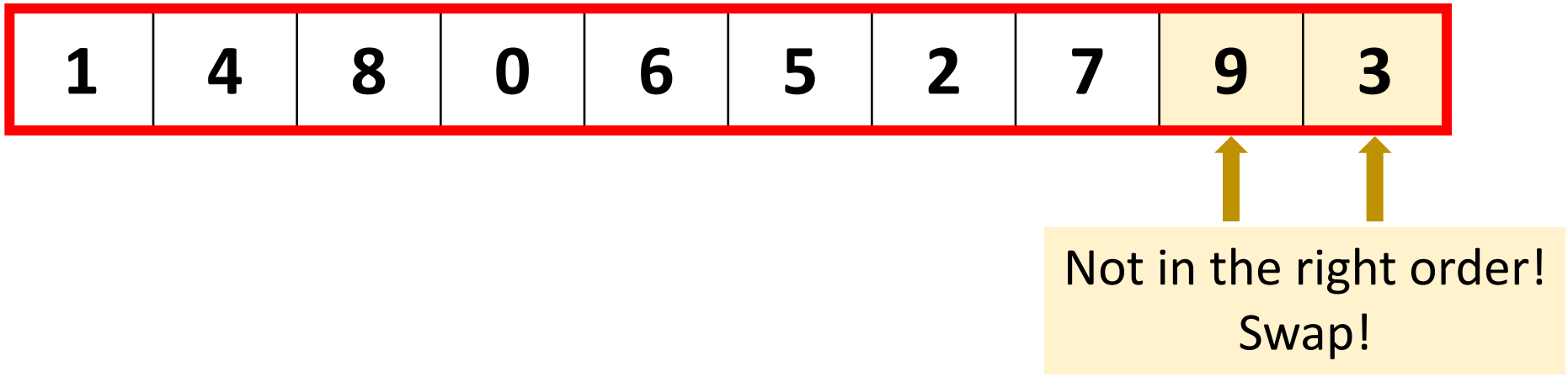
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



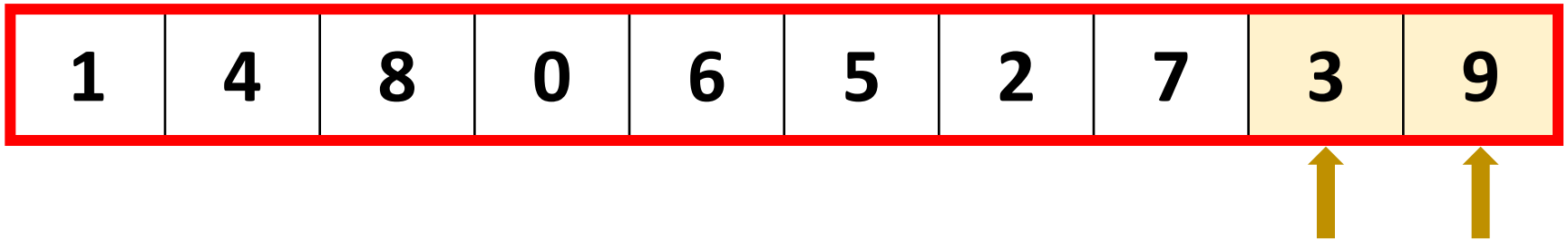
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



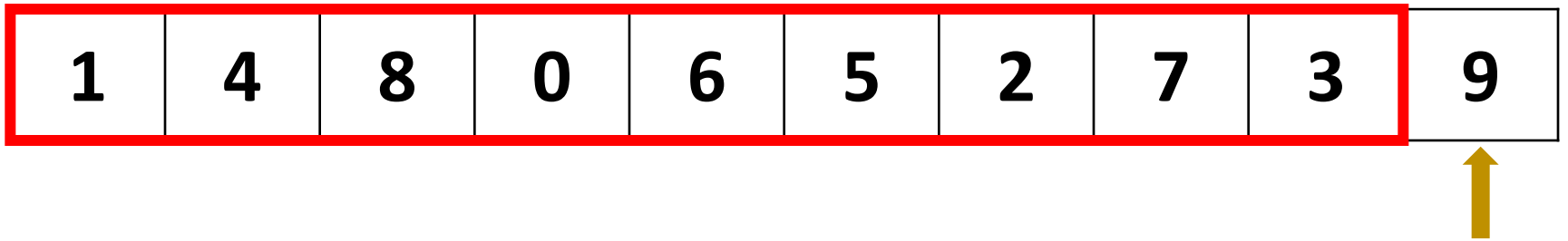
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 1



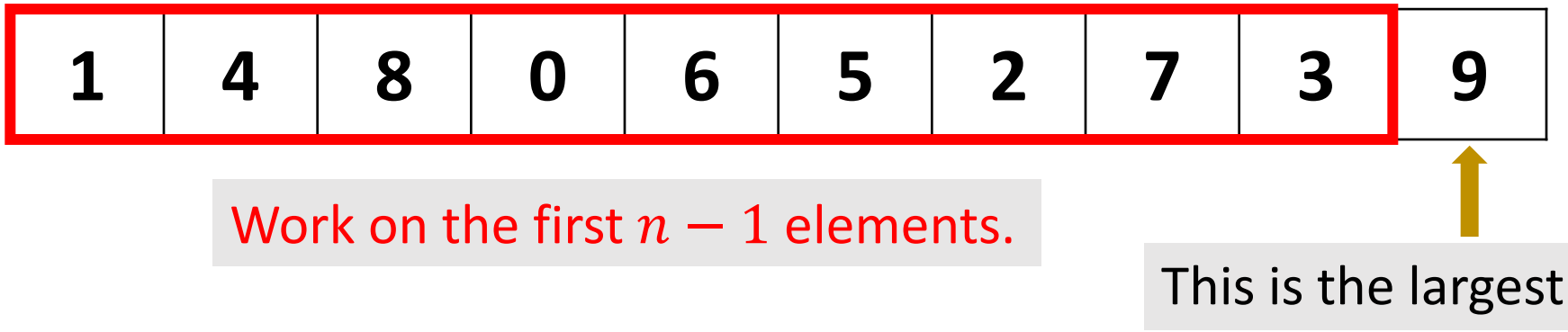
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

After Iteration 1



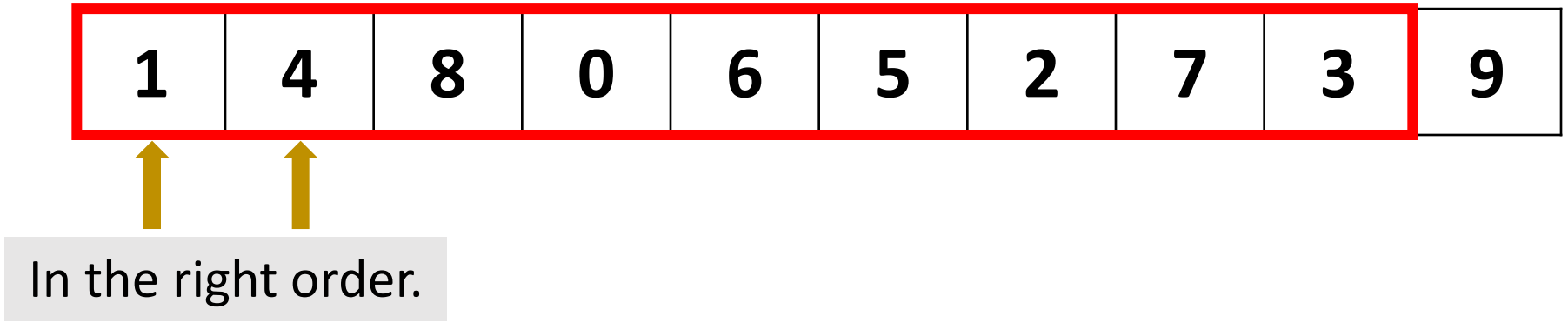
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



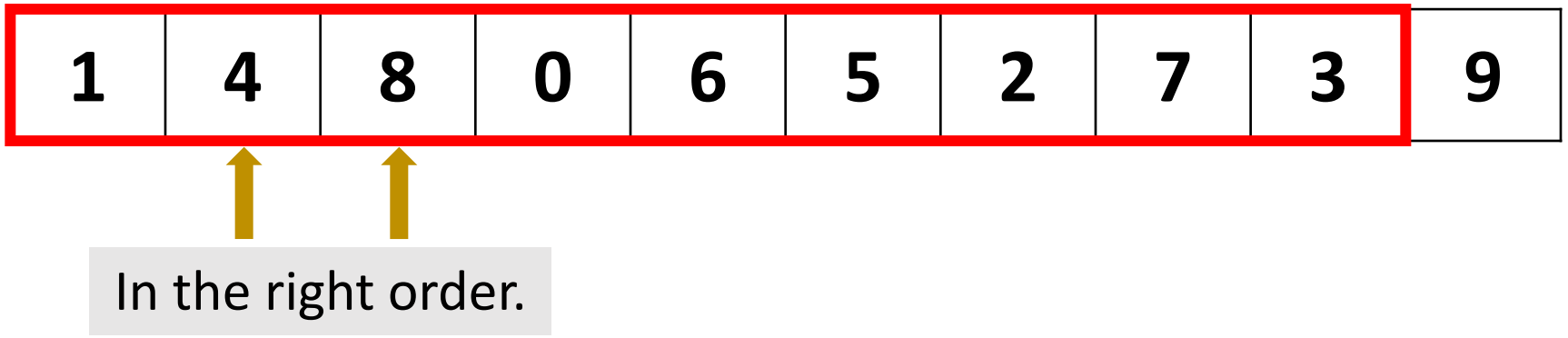
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



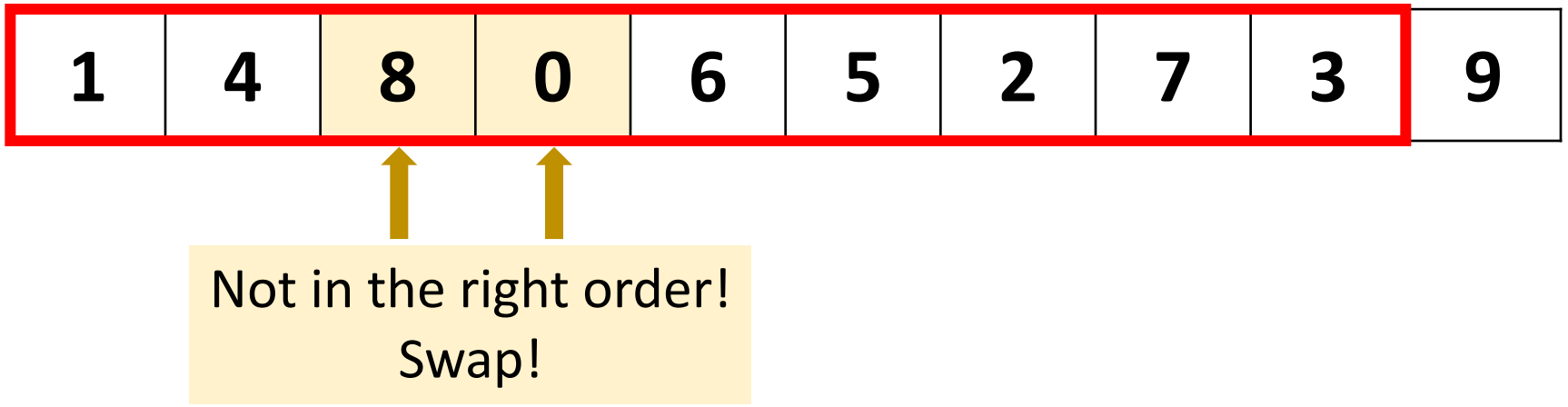
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



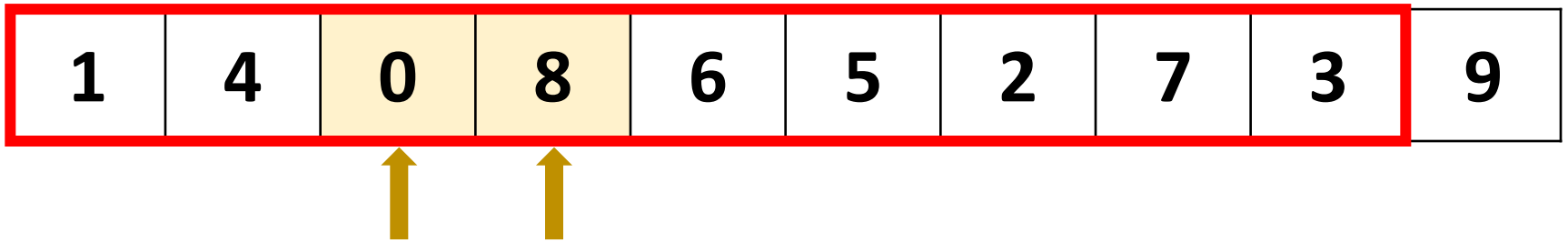
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



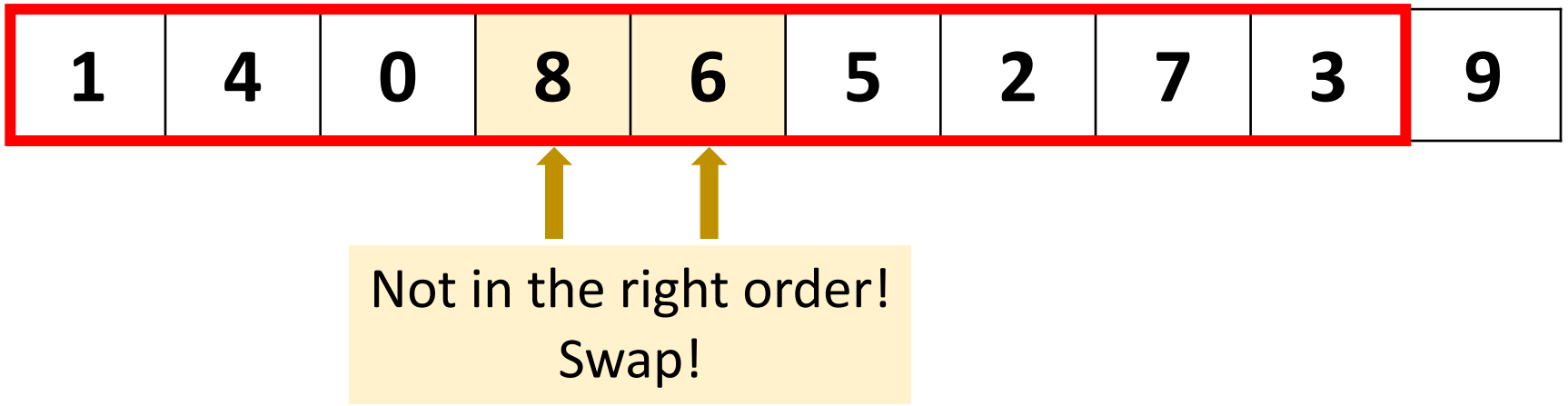
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



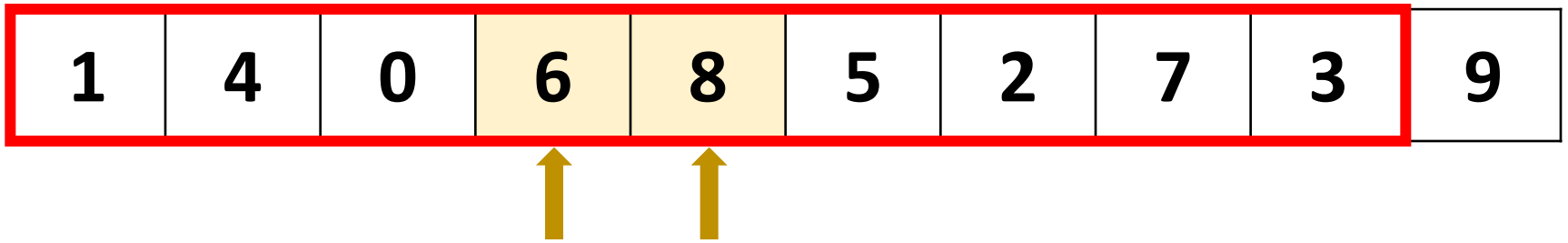
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



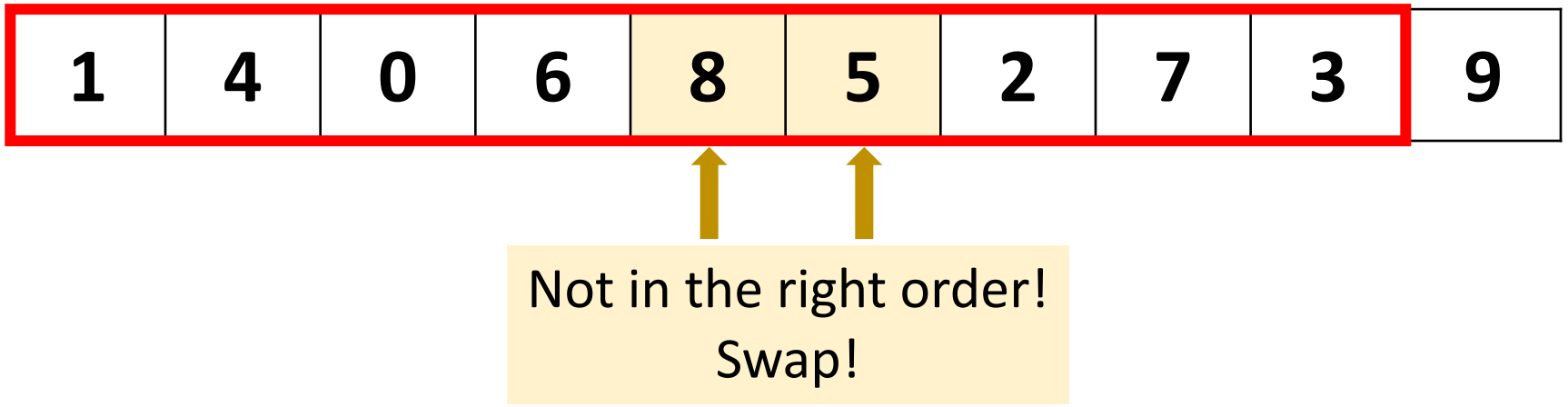
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



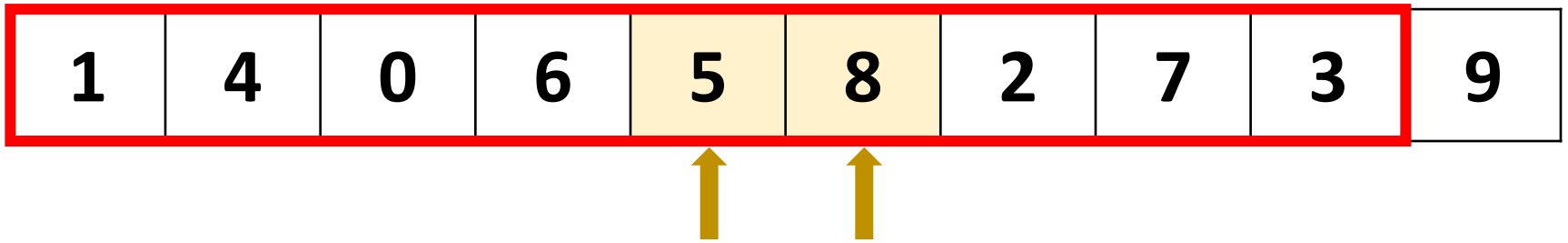
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



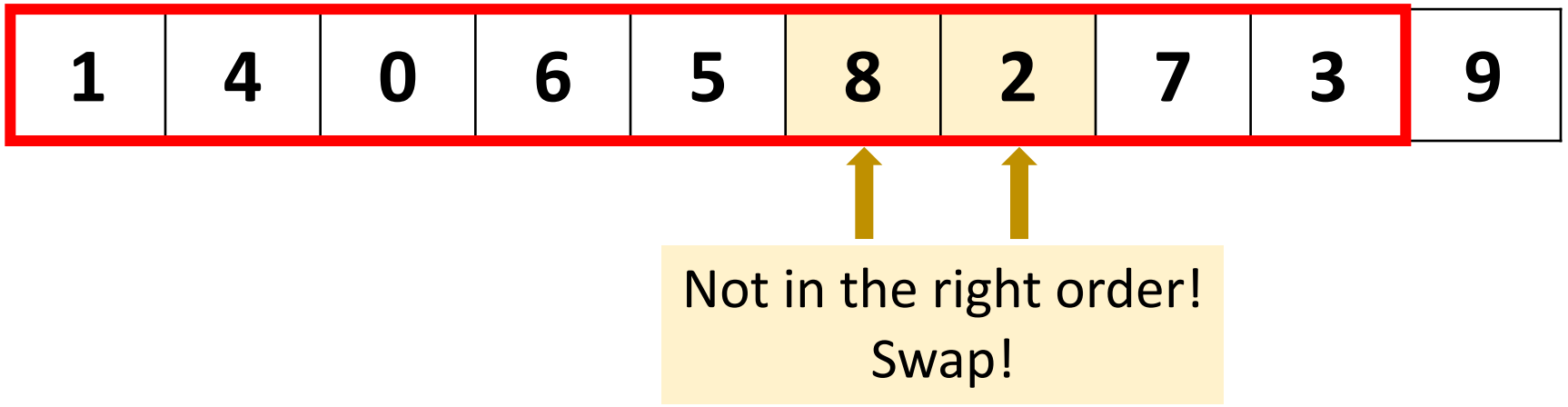
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



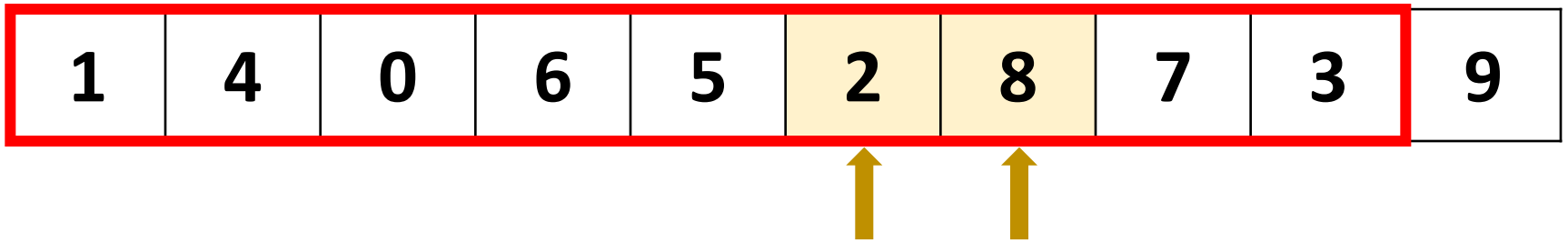
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



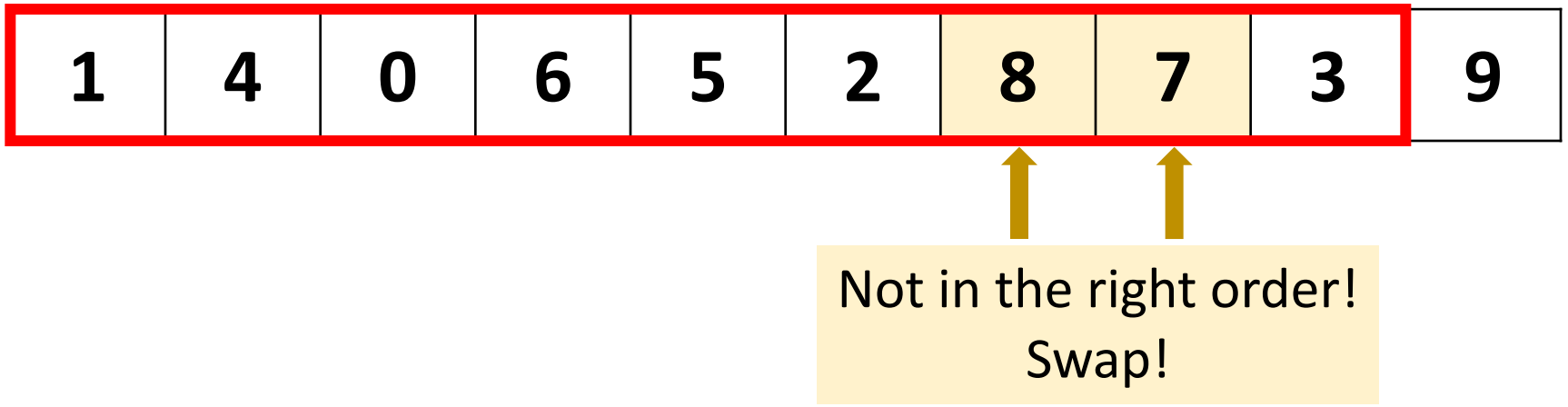
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



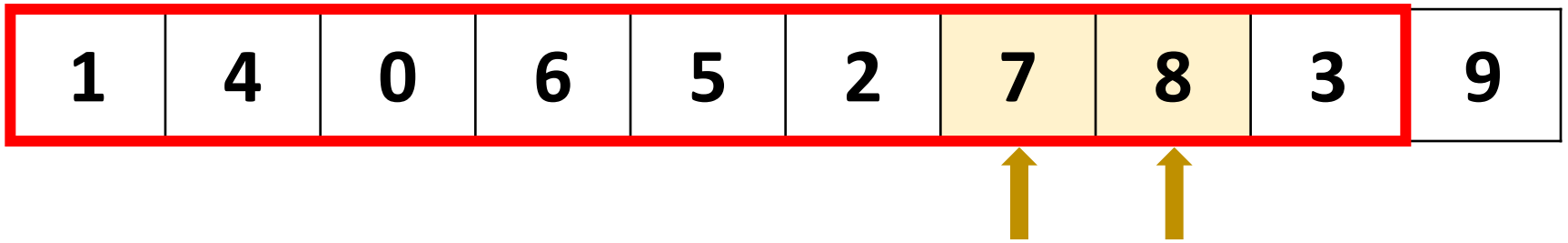
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



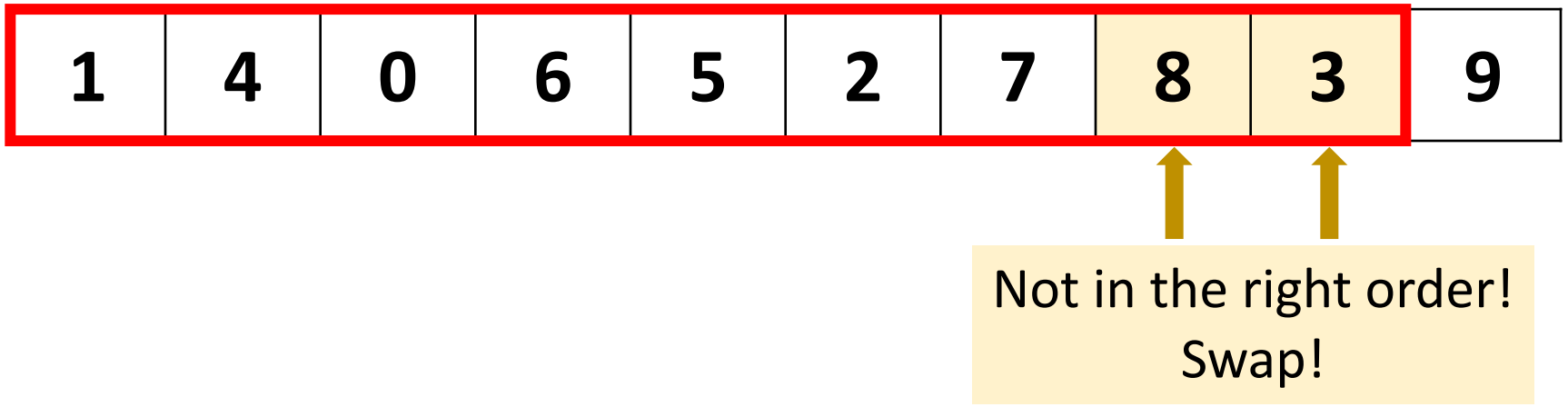
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



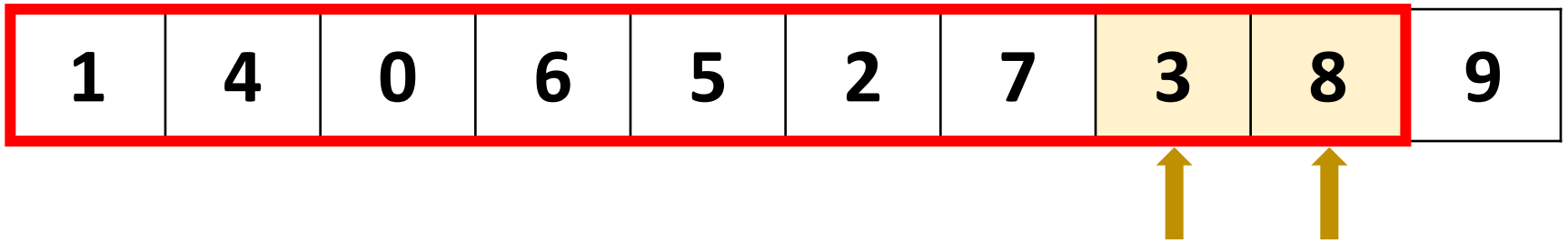
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



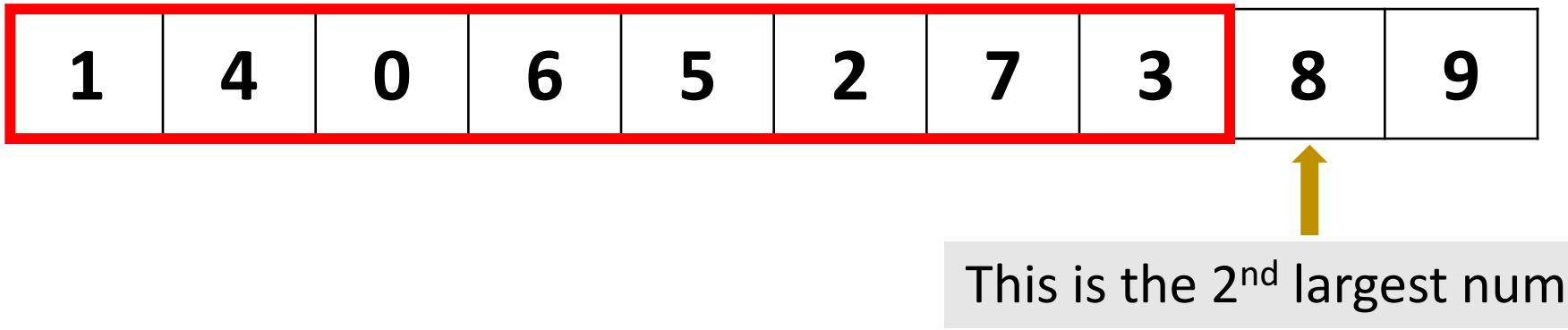
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 2



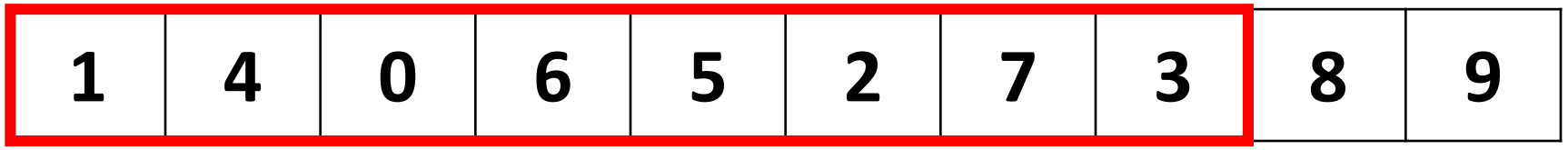
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

After Iteration 2



- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

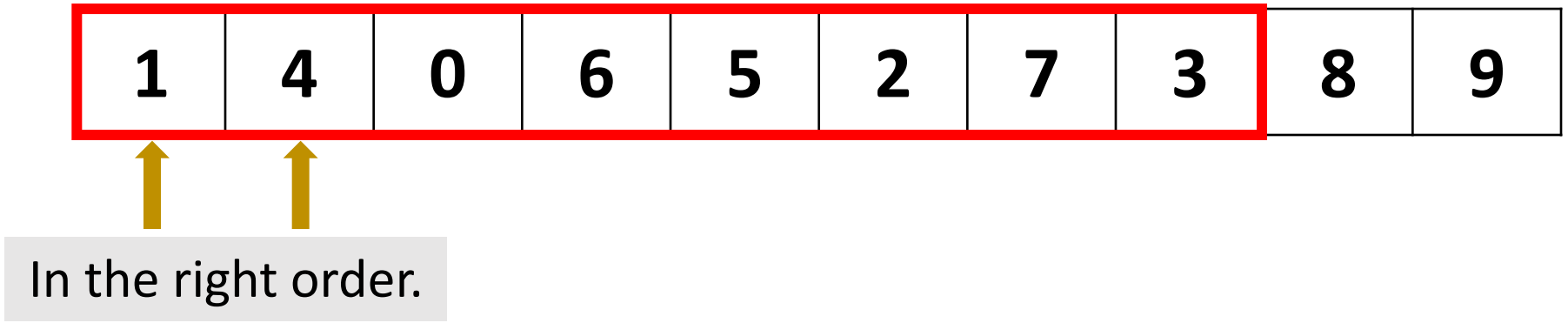
Iteration 3



Work on the first $n - 2$ elements.

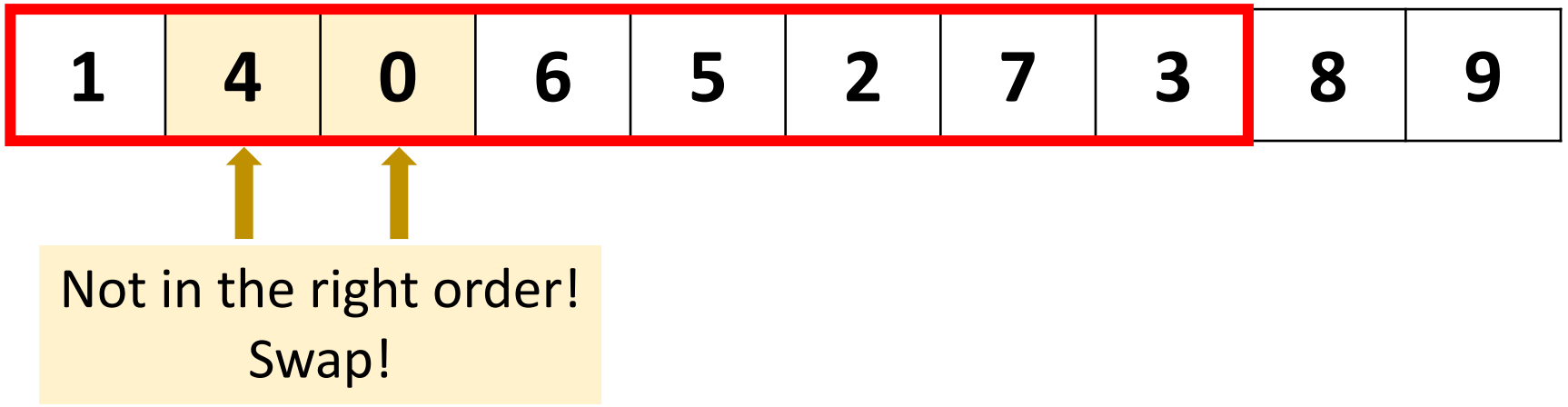
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



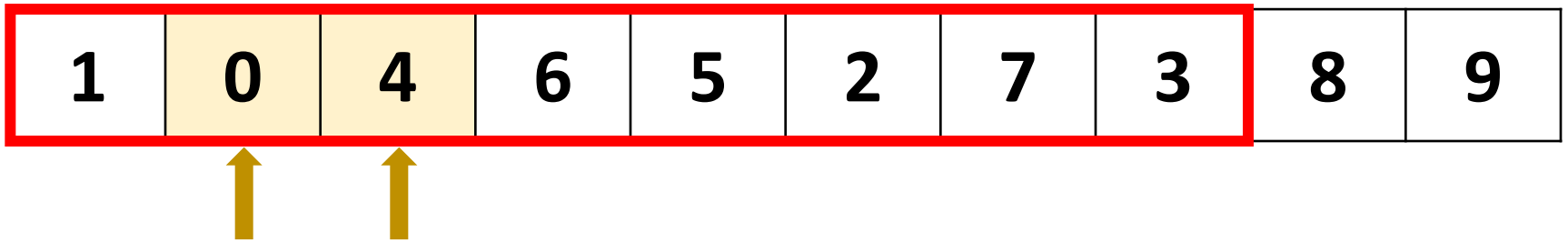
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



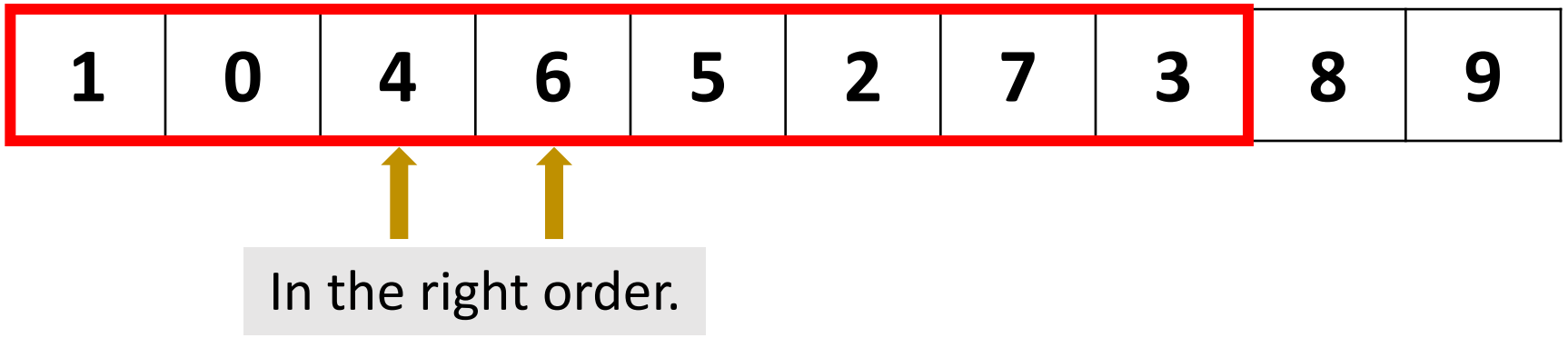
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



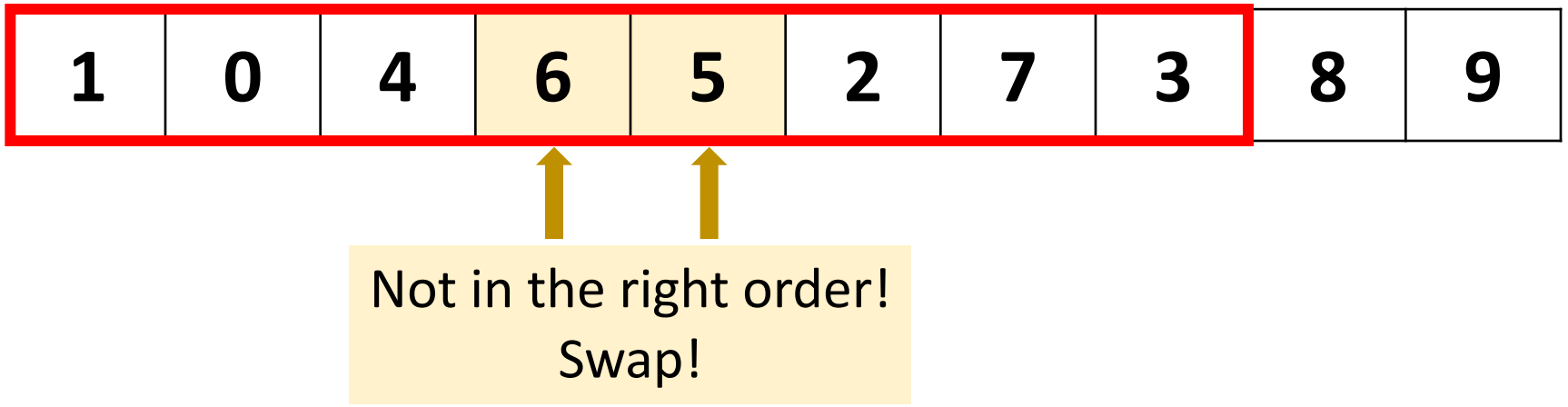
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



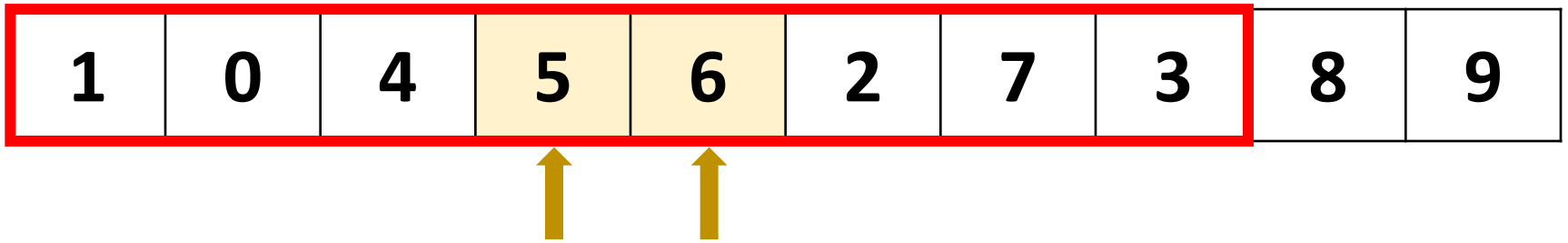
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



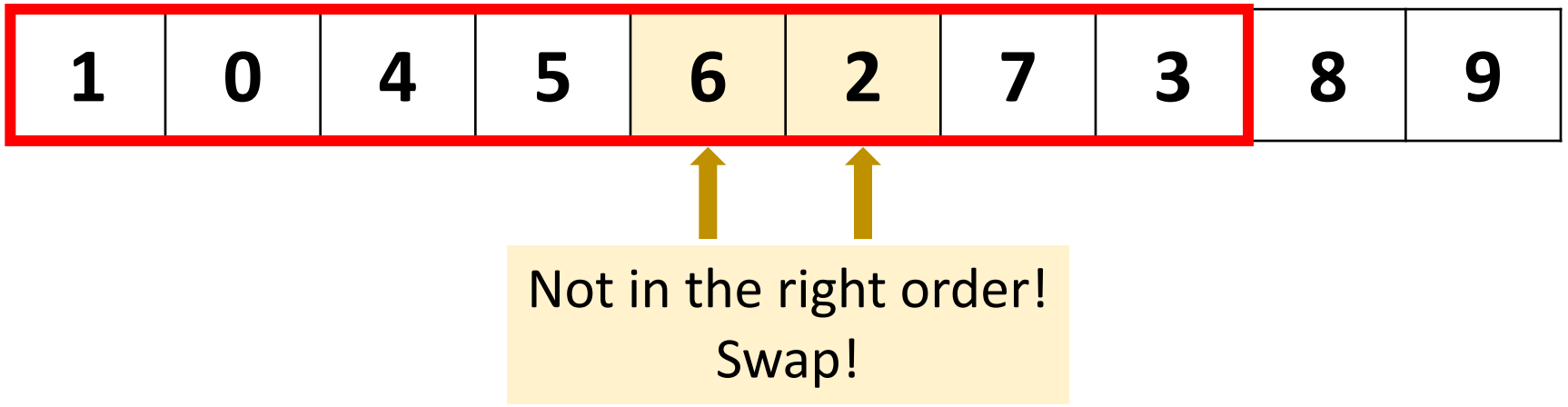
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



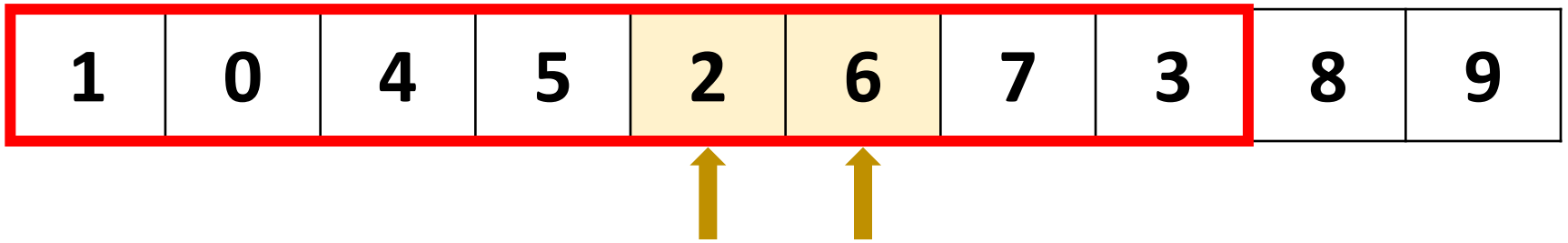
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



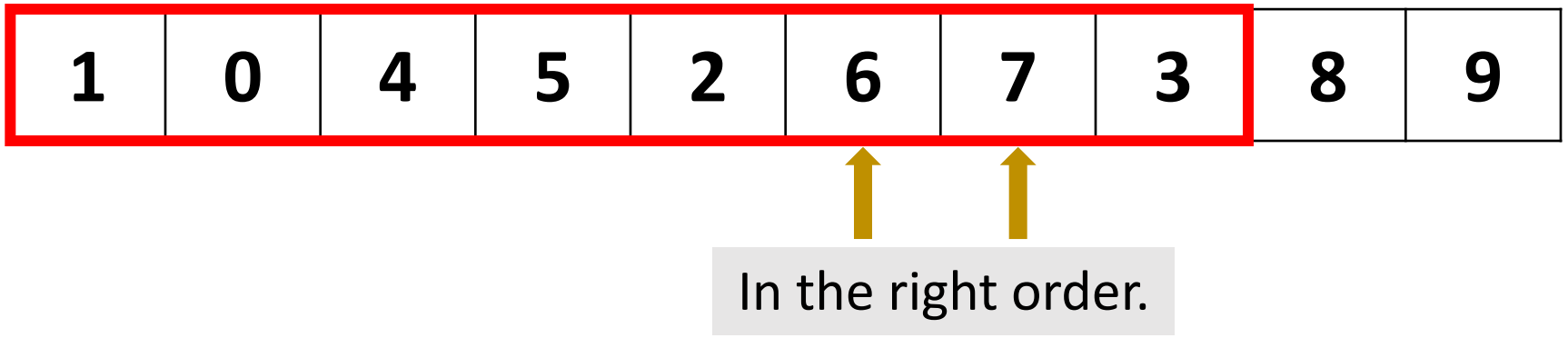
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



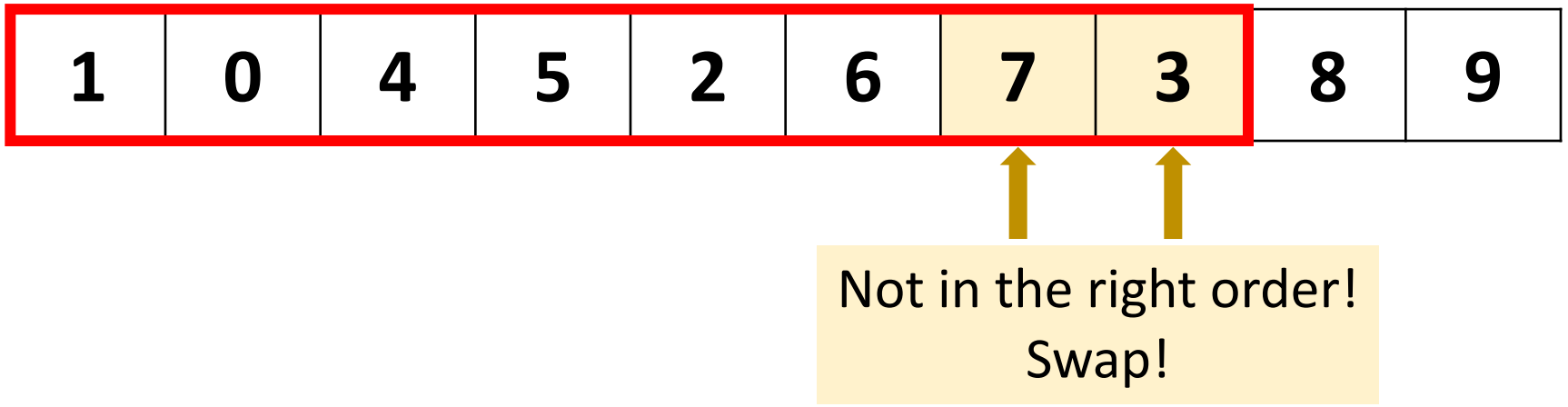
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



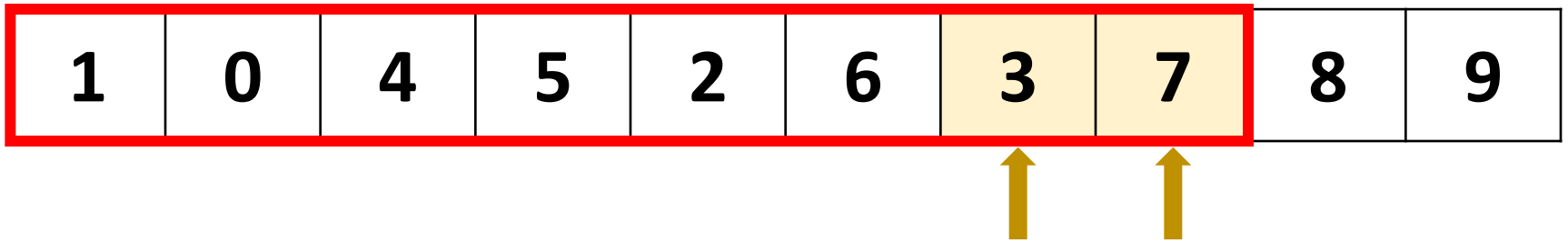
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



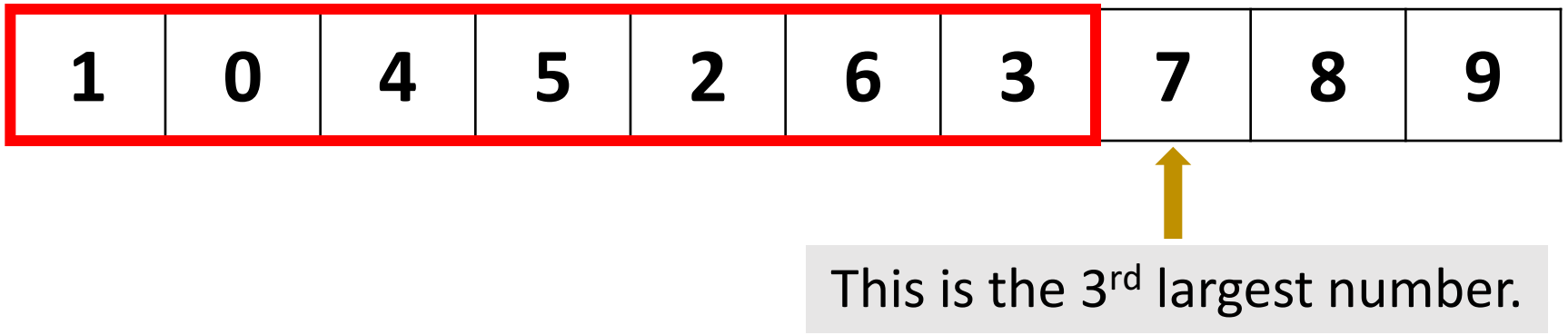
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 3



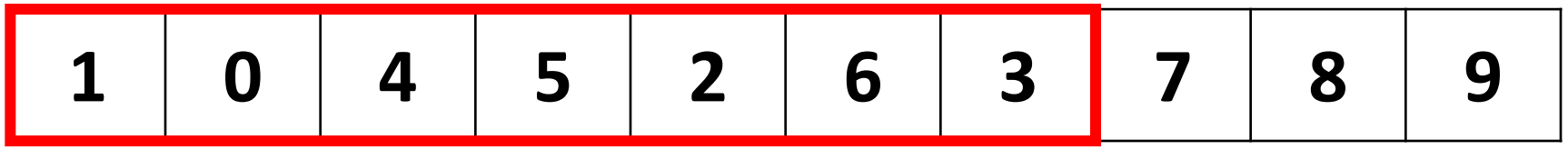
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

After Iteration 3



- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

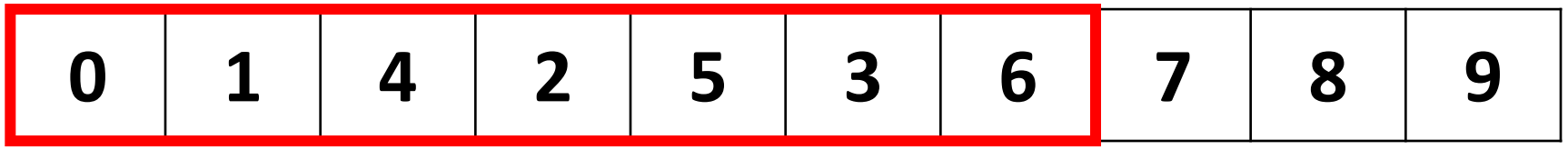
Iteration 4



Work on the first $n - 3$ elements.

- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

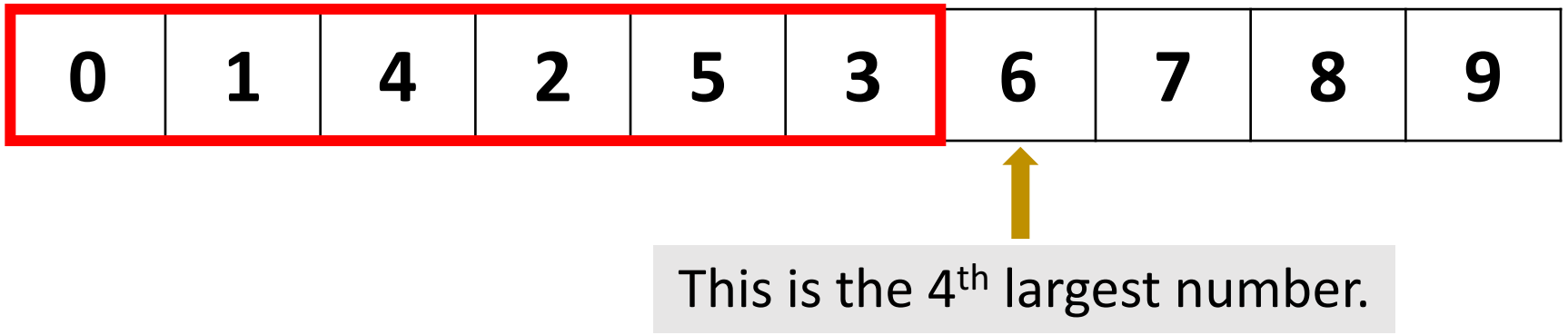
After Iteration 4



Work on the first $n - 3$ elements.

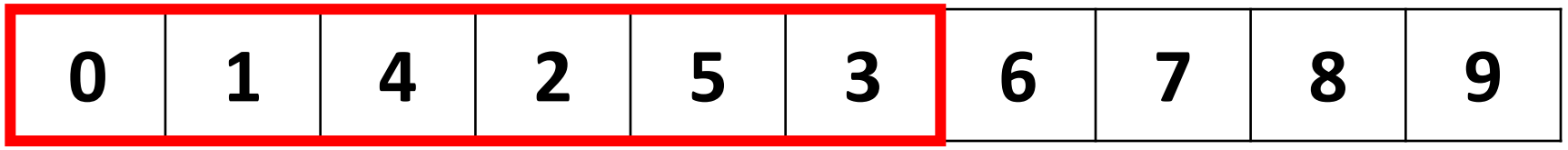
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

After Iteration 4



- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

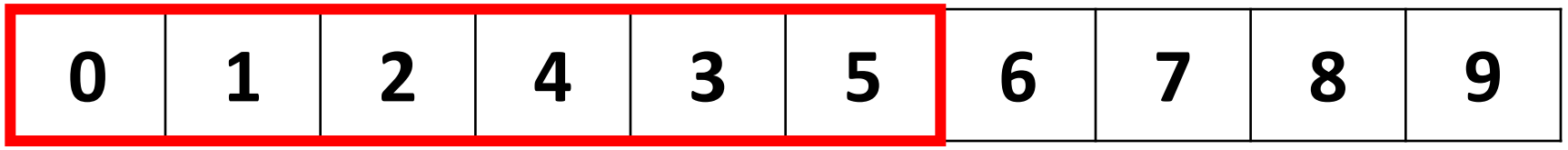
Iteration 5



Work on the first $n - 4$ elements.

- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

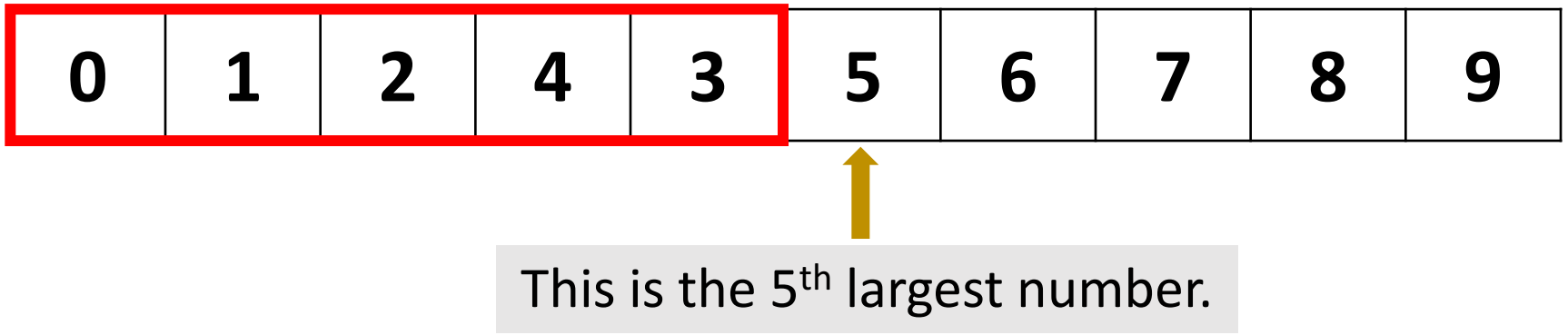
After Iteration 5



Work on the first $n - 4$ elements.

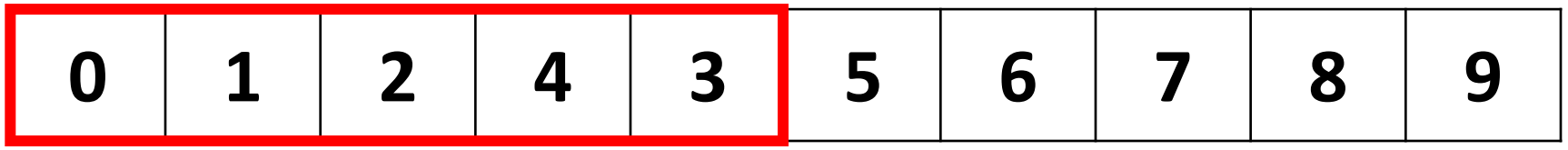
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

After Iteration 5



- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

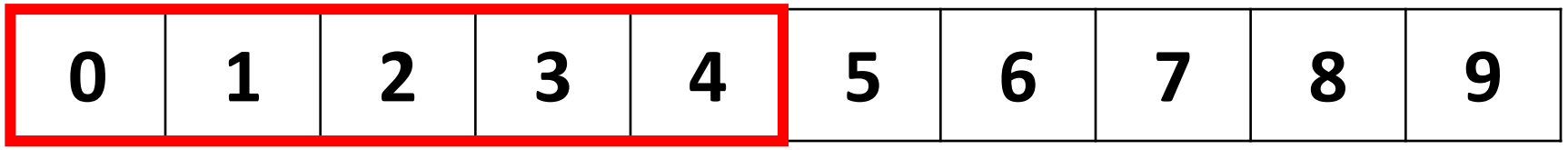
Iteration 6



Work on the first $n - 5$ elements.

- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

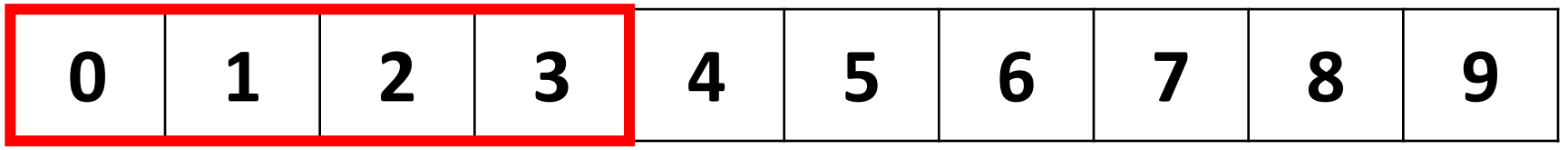
After Iteration 6



Work on the first $n - 5$ elements.

- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

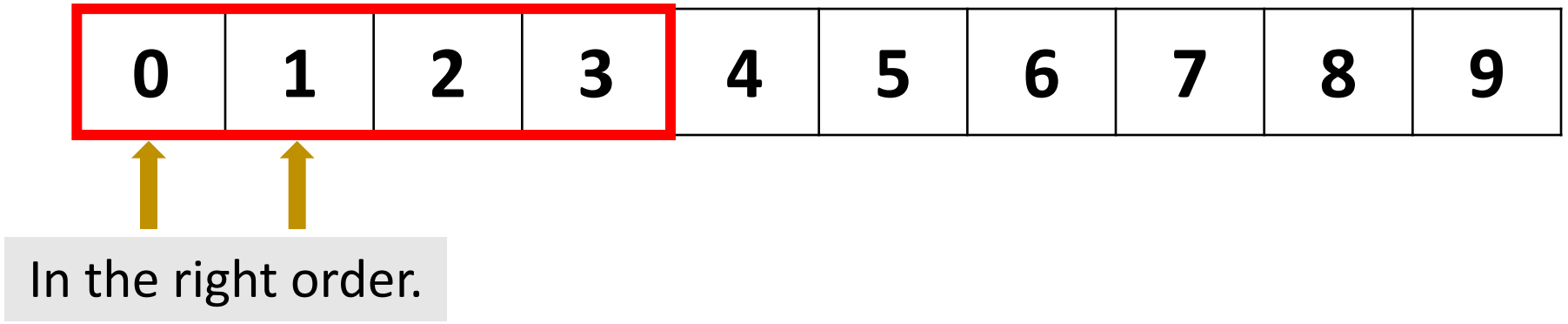
Iteration 7



Work on the first $n - 6$ elements.

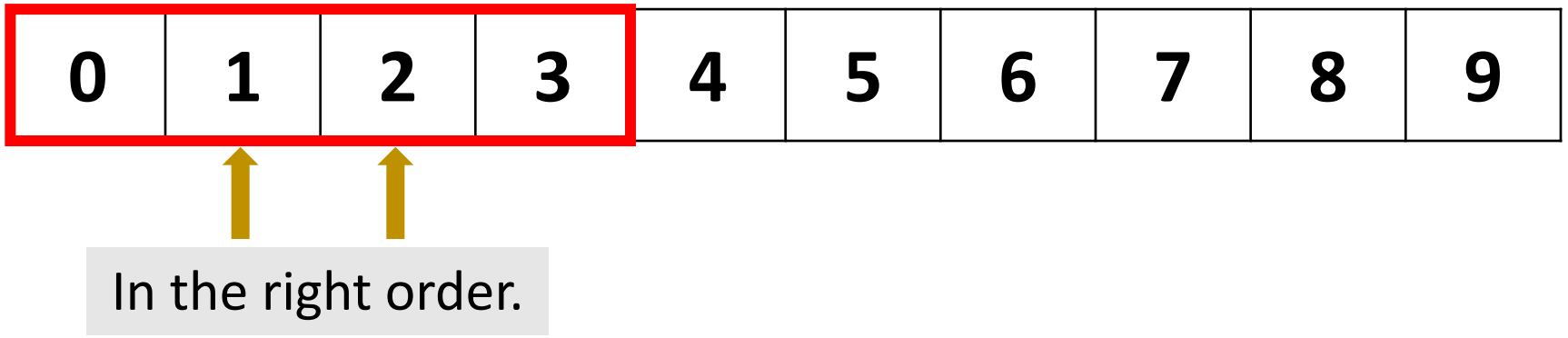
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 7



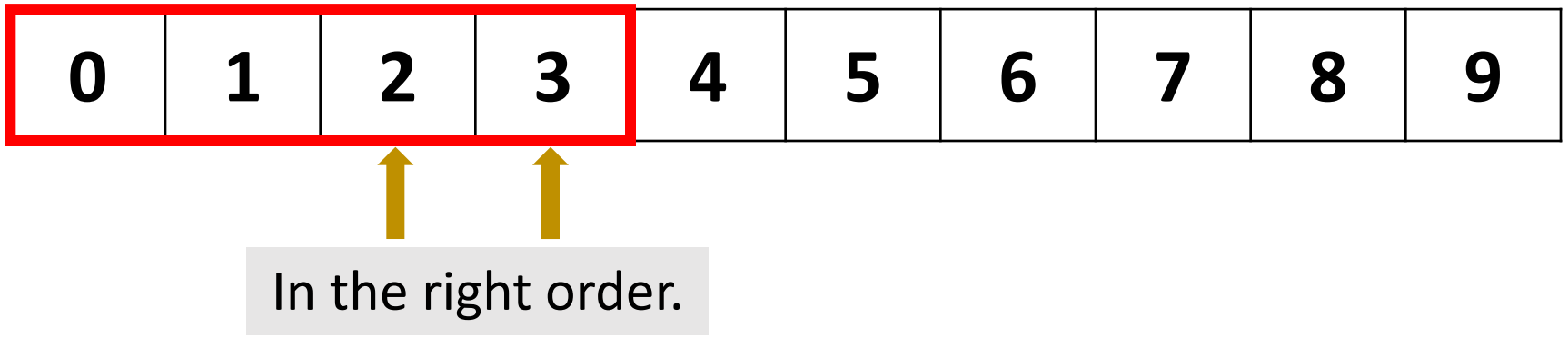
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 7



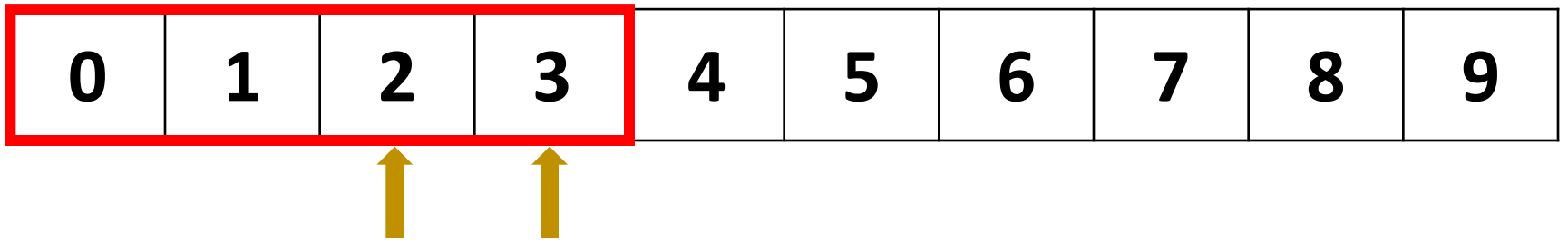
- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 7



- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Iteration 7



No swap in the 7th iteration ➔ These elements are in the ascending order.

Stop!

- Start from the beginning of the array.
- Compare every adjacent pair.
- Swap them if they are not in the right order.
- After k iterations, the last k elements are the biggest.

Naïve Implementation

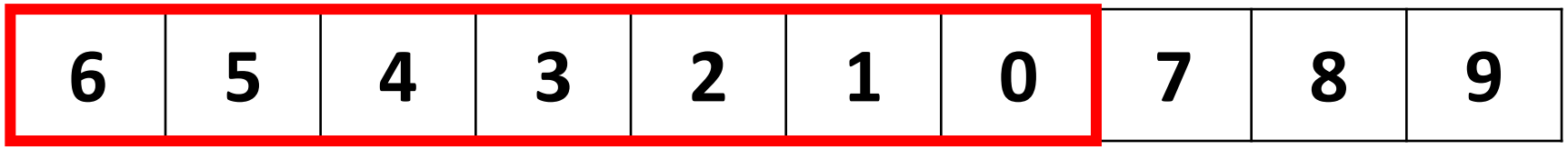
```
void bubblesort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
```

Improved Implementation

```
void bubblesort(int arr[], int n) {
    int i, j;
    bool swapped;
    for (i = 0; i < n-1; i++) {
        swapped = false;
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                swapped = true;
            }
        }
        if (swapped == false) break;
    }
}
```

Time Complexity

Worst-Case Time Complexity

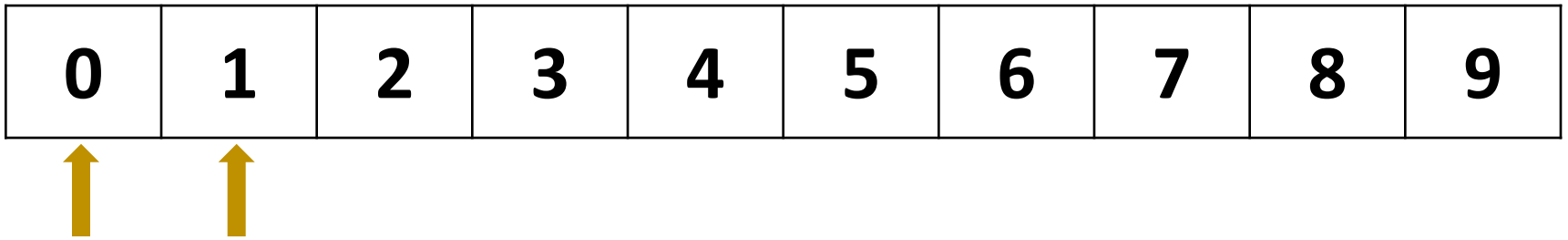


Iteration 4: Work on the first $n-3$ elements.

- The k-th iteration performs $n-k$ operations.
- In the worst case, $n-1$ iterations are needed.
- Time complexity:

$$T(n) = \sum_{k=1}^{n-1}(n-k) = O(n^2).$$

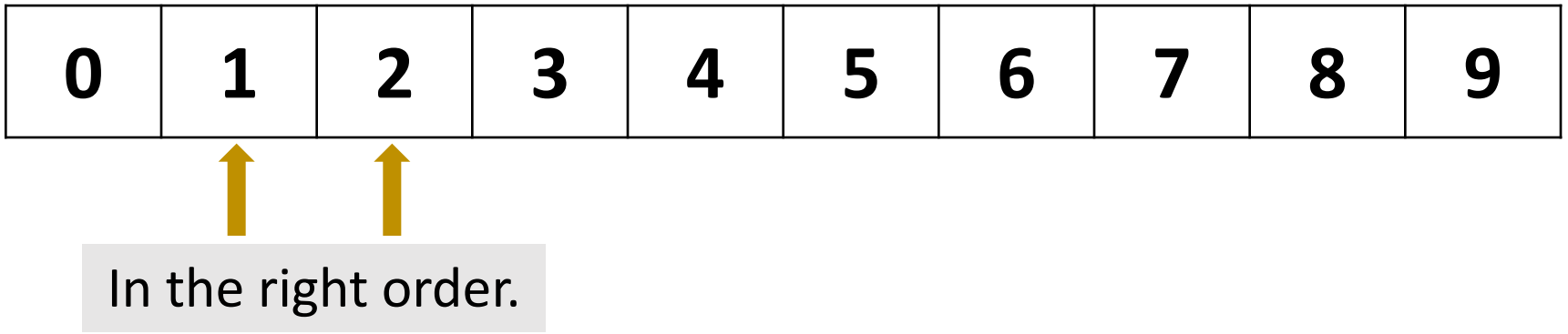
Best-Case Time Complexity



In the right order.

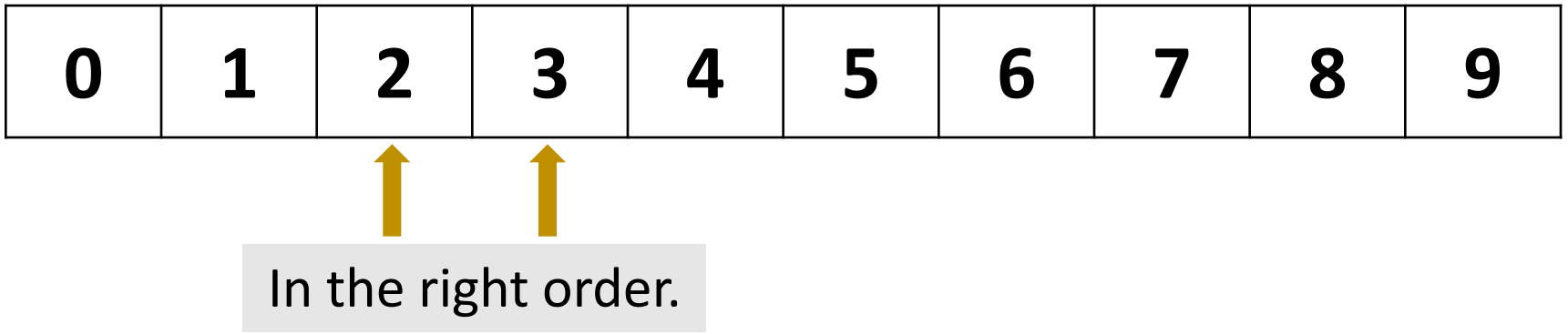
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



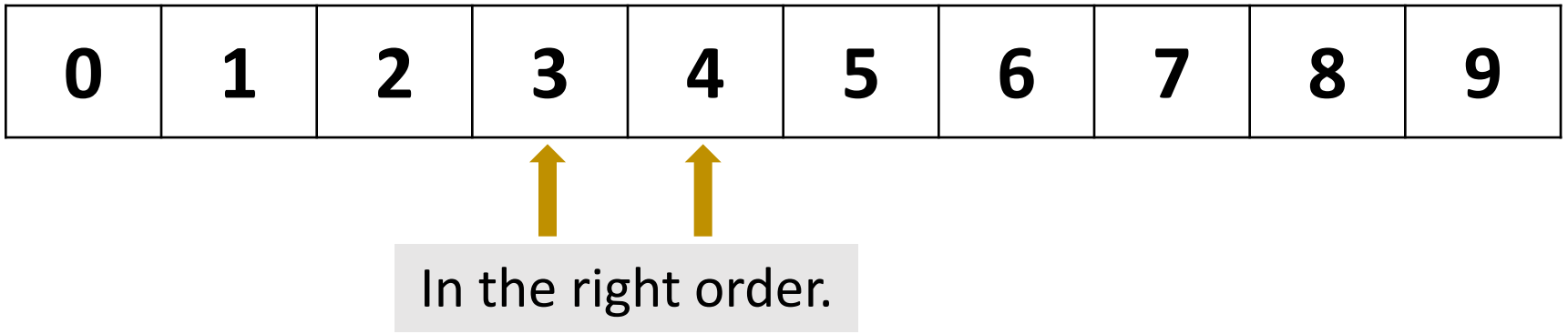
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



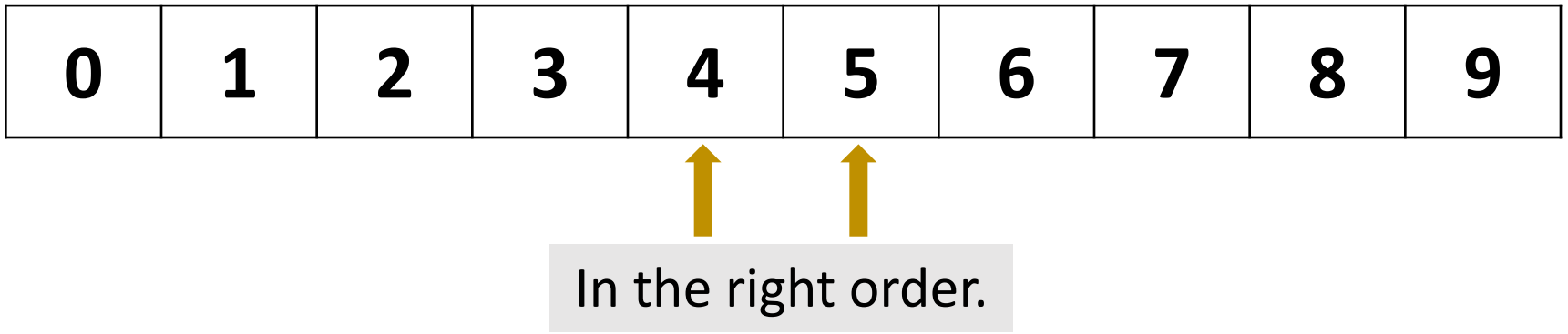
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



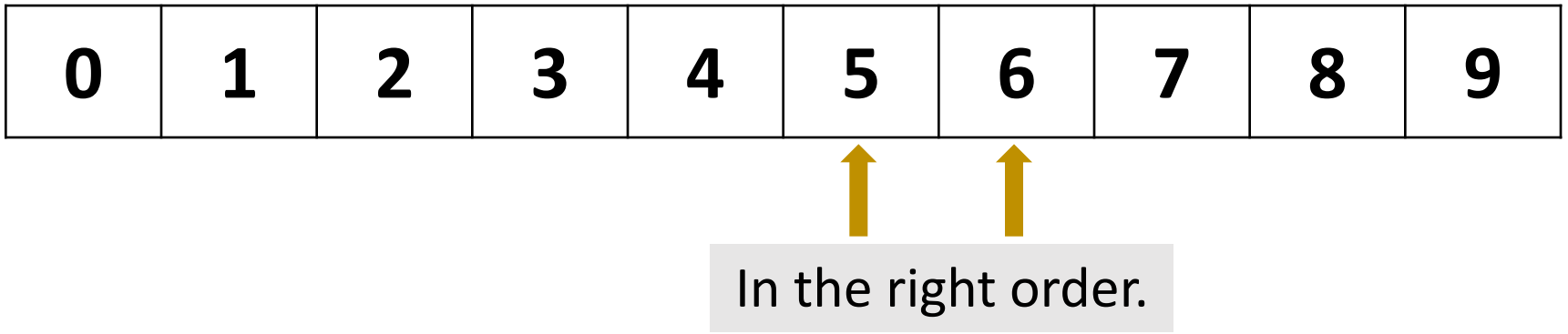
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



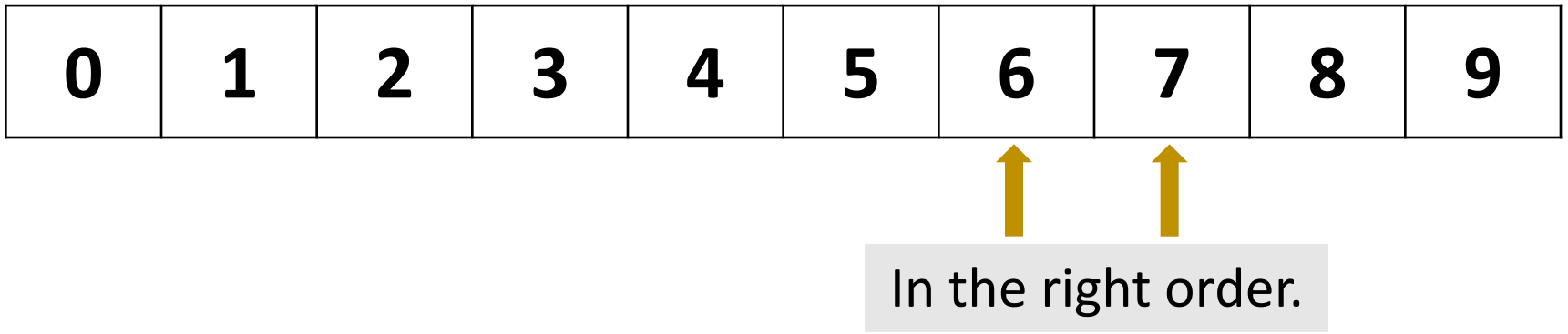
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



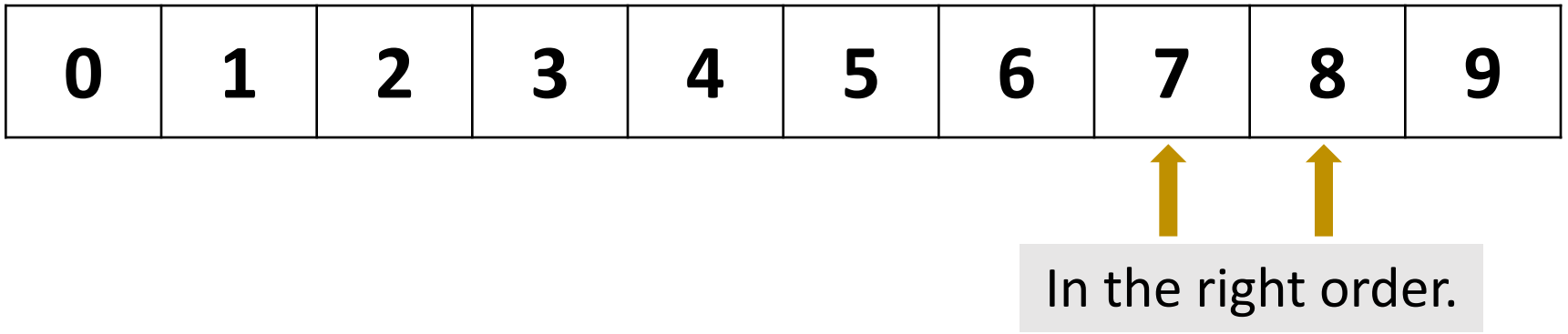
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



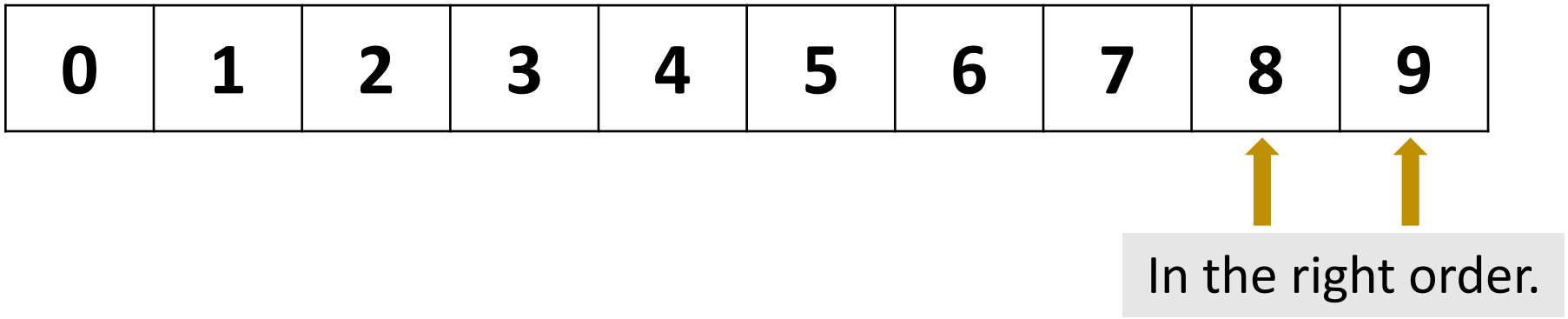
- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



- In the best case, the input array is in the ascending order.

Best-Case Time Complexity



- In the best case, the input array is in the ascending order.

Best-Case Time Complexity

0	1	2	3	4	5	6	7	8	9

- In the best case, the input array is in the ascending order.
- No swap during the 1st iteration.
- Thus we know the array is already sorted.

Best-Case Time Complexity

0	1	2	3	4	5	6	7	8	9

- In the best case, the input array is in the ascending order.
- No swap during the 1st iteration.
- Thus we know the array is already sorted.
- Return after the 1st iteration.
- Best-case time complexity: $O(n)$.

Average-Case Time Complexity

- Best case: the initial input is in the right order.
- Worst case: the initial input is in the reverse order.
- Average case: the input array is like random shuffled.
- Time complexity: $O(n^2)$.

Thank you!