COMP251: DATA STRUCTURES & ALGORITHMS

* Some slides from “Algorithms and Data Structures”
by Douglas Wilhelm Harder

Quick Sort

Strategy

We have seen Merge sort which is $\Theta(n \ln(n))$ but requires more memory (it is not in-place)

We will now look at another recursive algorithm which may be done ***almost*** in place while it is also ***usually*** faster:

- Average case: $\Theta(n \ln(n))$ time and $\Theta(\ln(n))$ memory
- Worst case: $\Theta(n^2)$ time and $\Theta(n)$ memory

We will look at strategies for avoiding the worst case

Quicksort

Merge sort splits the array into two sub-lists (at the middle) and sorts them

The larger problem is split into two sub-problems based on *location* in the array

Consider the following alternative:

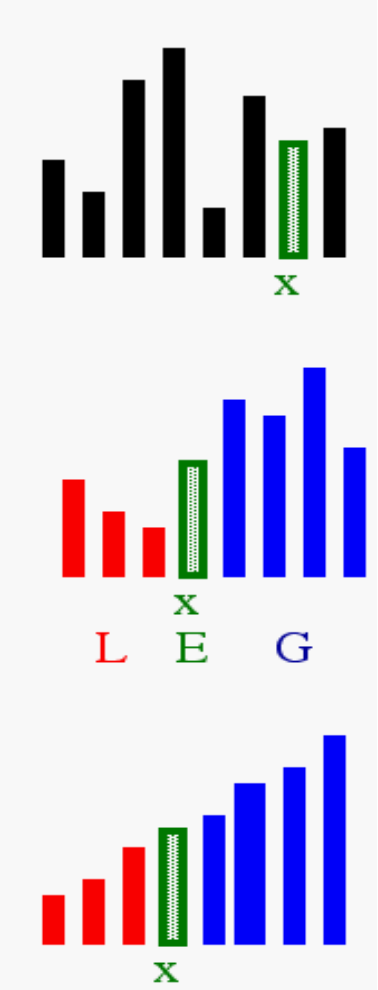
- Choose an object in the array and partition the remaining objects into two groups relative to the chosen entry

Quicksort

1) **Select**: pick an element (called pivot)

2) **Divide**: partition elements, everything smaller than x to the left of x (so x goes to its final position E)

3) **Conquer**: recursively sort left and right partitions



Quicksort

```
public void quickSort(Comparable[] arr, int low, int high)
{
    if (low <= high) // if size <= 1 already sorted
        return;
    else // size is 2 or larger
    {
        // partition range
        int pivotIndex = partition(arr, low, high);
        // sort  left  subarray
        QuickSort(arr, low, pivotIndex - 1);
        // sort  right  subarray
        QuickSort(arr, pivotIndex + 1, high);
    }
}
```

Quicksort

For example, given

80	38	95	84	66	10	79	44	26	87	96	12	43	81	3

we can select 44, and partition the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	44	80	95	84	66	79	87	96	81

Notice that 44 is now in the correct location

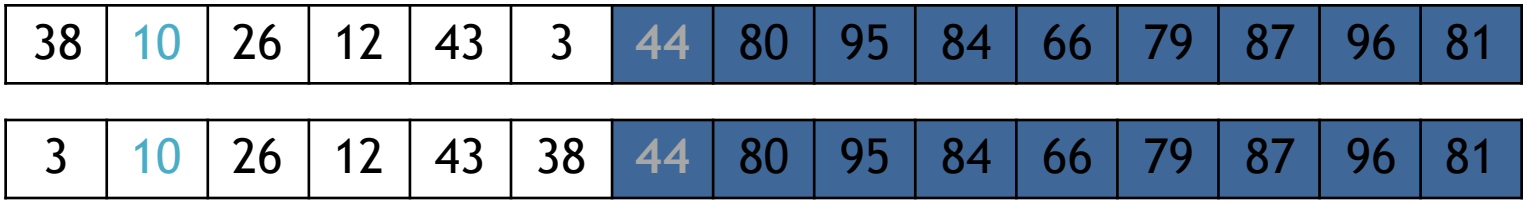
Then recursively apply the quicksort algorithm to the first six and last eight entries

Quicksort

Call the quick sort algorithm recursively: choose 10 as pivot

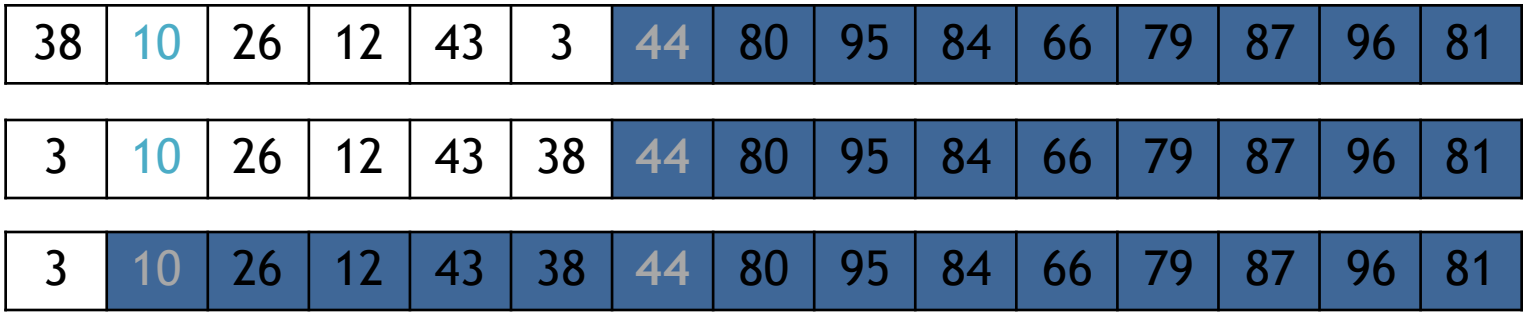
Quicksort

order the remaining entries to two parts: <10 and >10



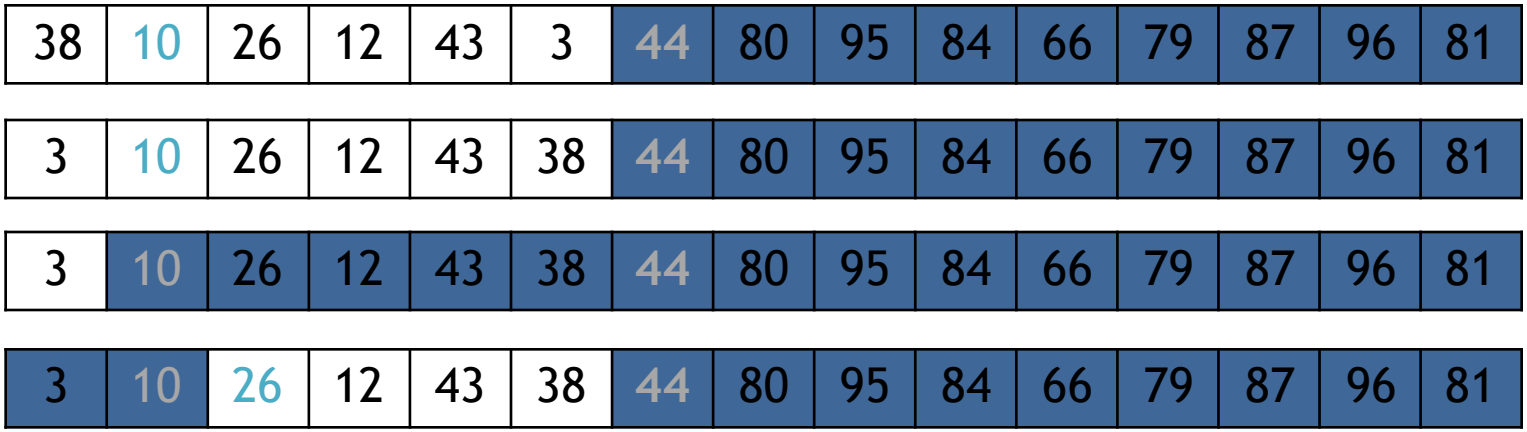
Quicksort

Call the quick sort algorithm recursively: 3 is just one element, it is sorted!



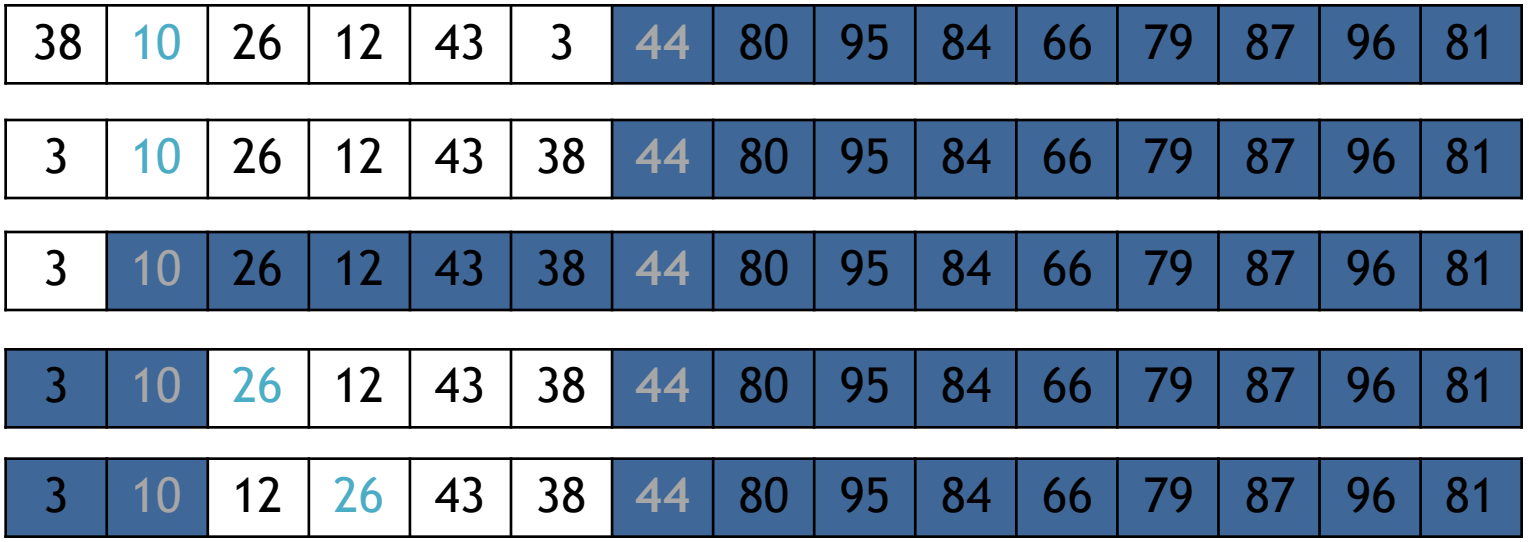
Quicksort

back in previous function call and call quick sort on the right side, pivot = 26



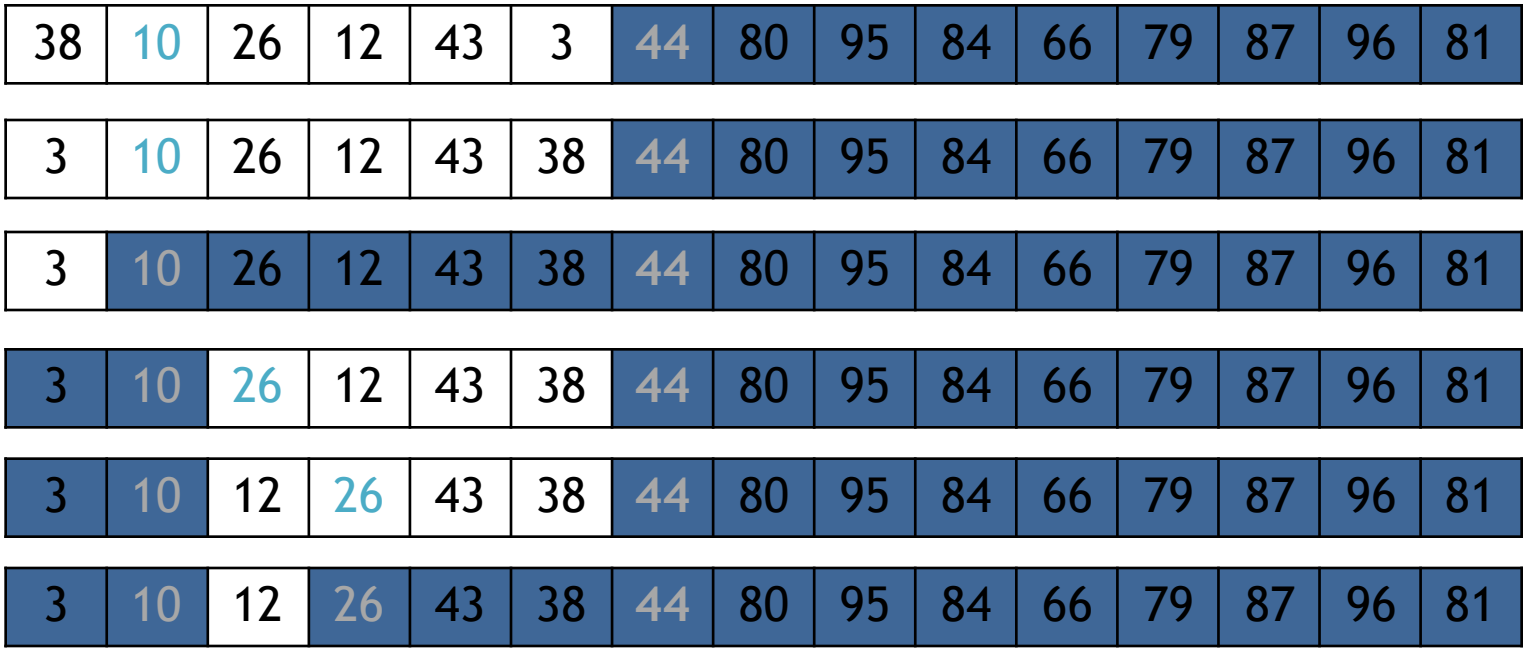
Quicksort

order the remaining entries to two parts: <26 and >26



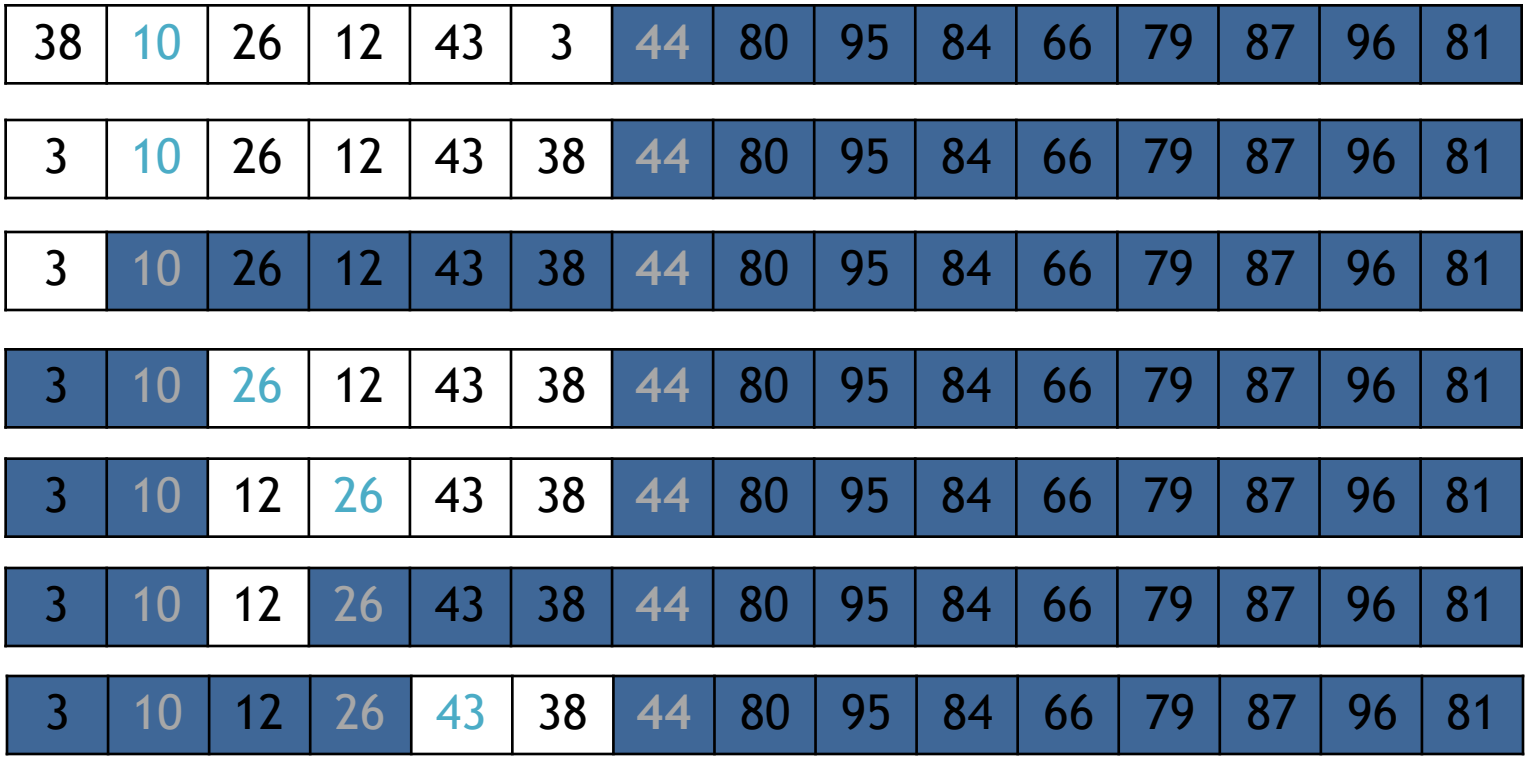
Quicksort

Call the quick sort on the left part, it is just one element which is sorted!



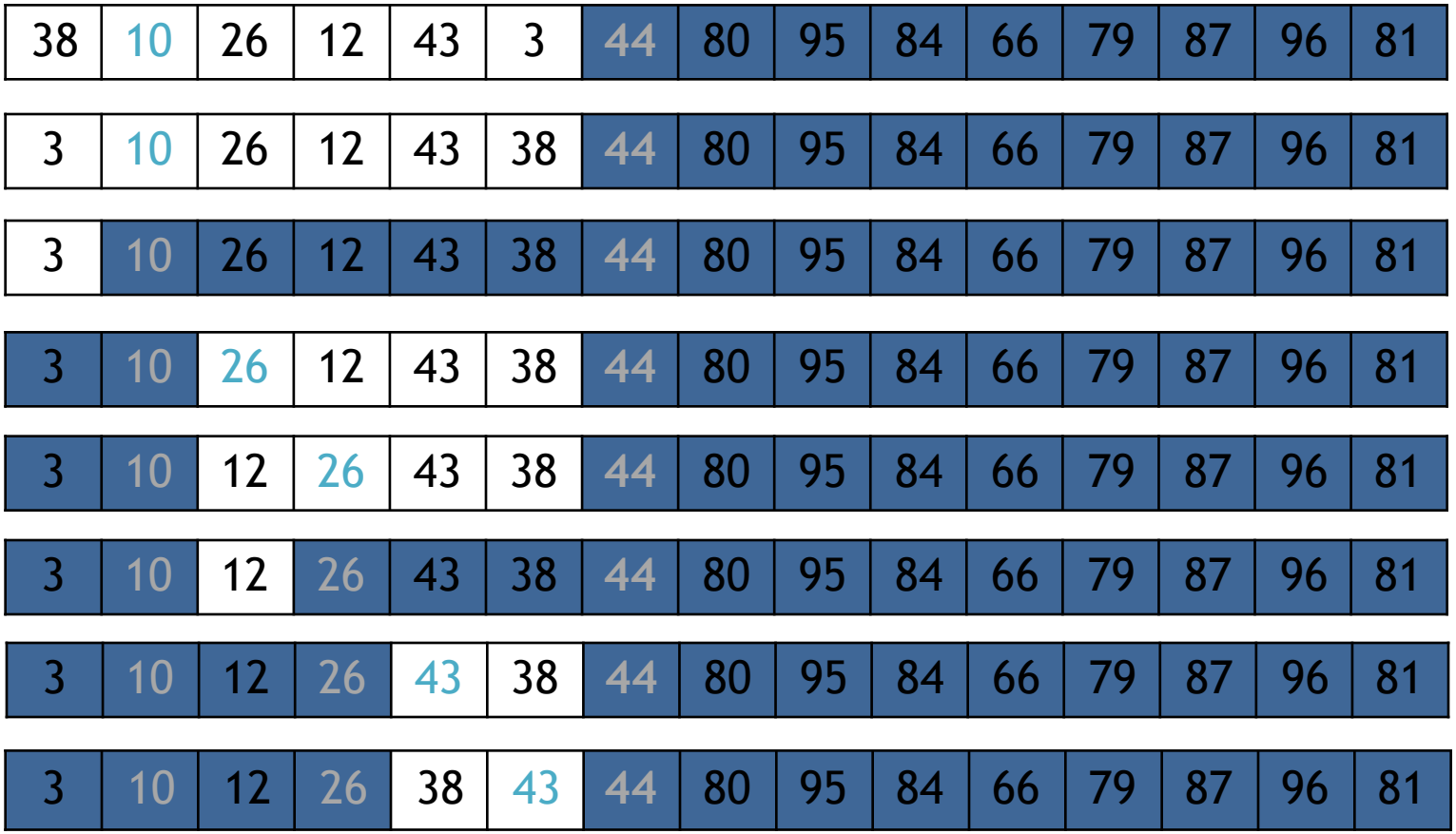
Quicksort

back in previous function call and call quick sort on the right side, pivot = 43



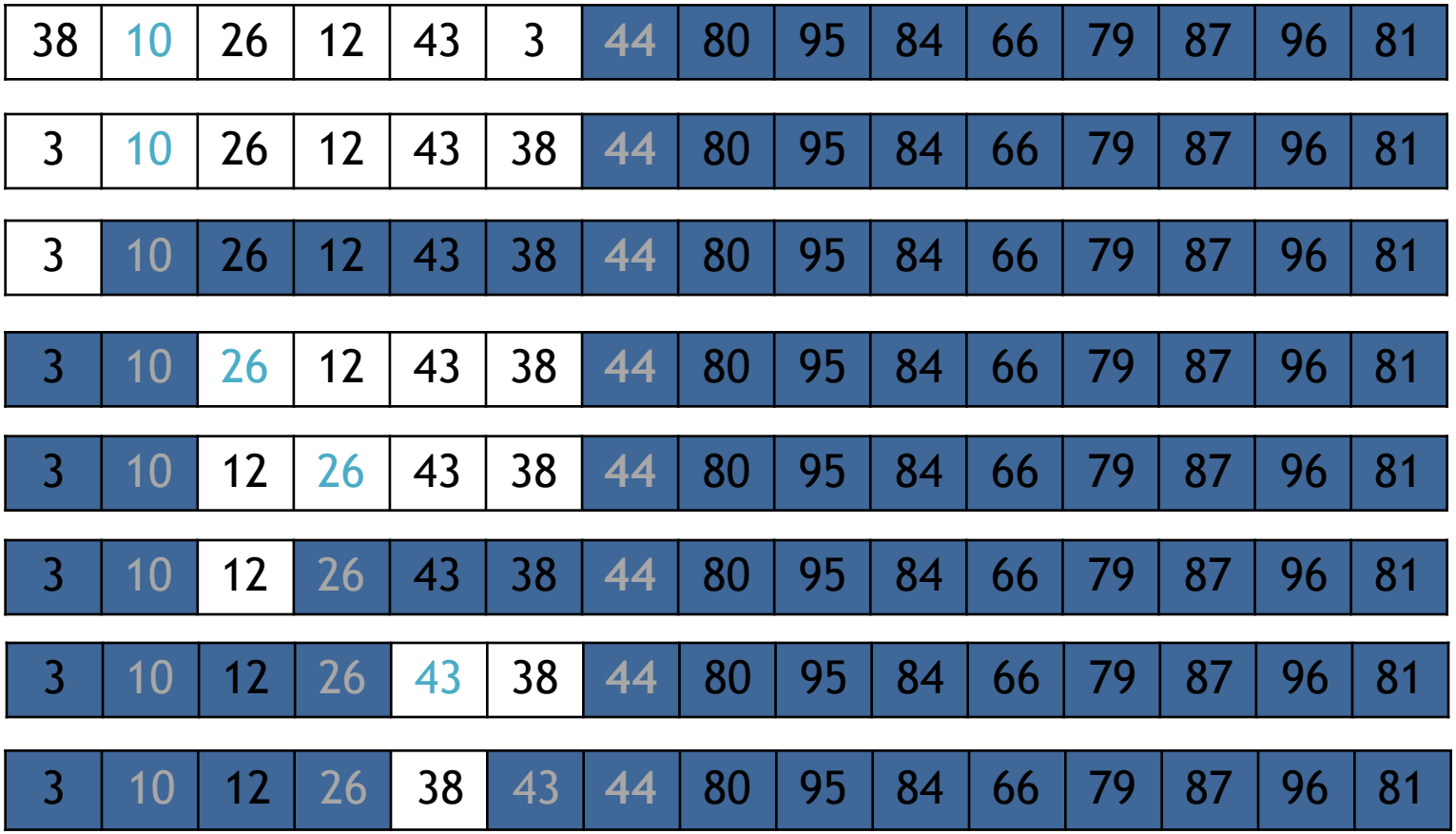
Quicksort

order the remaining entries to two parts: <43 and >43



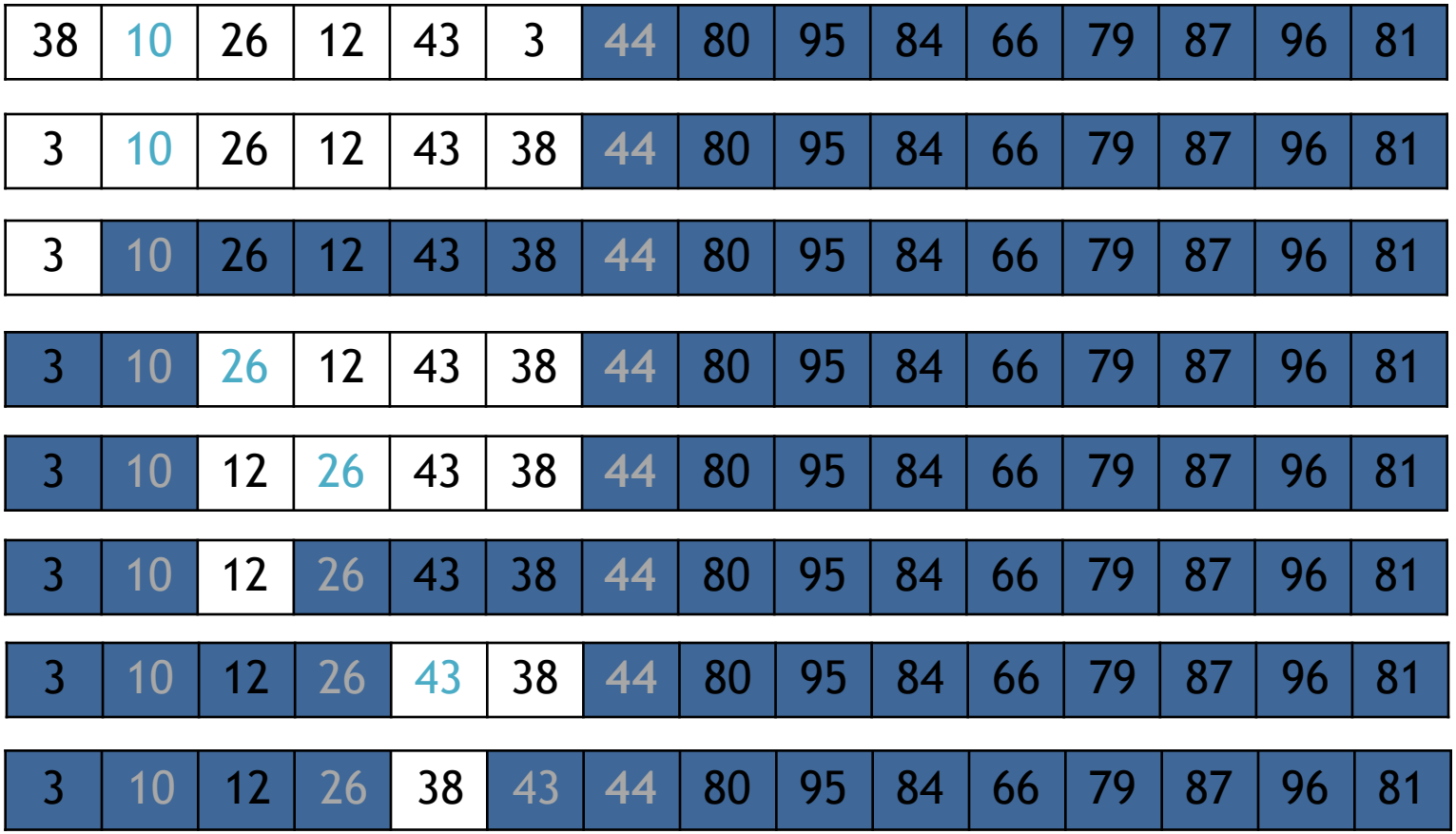
Quicksort

Call the quick sort on the left part, it is just one element which is sorted!



Quicksort

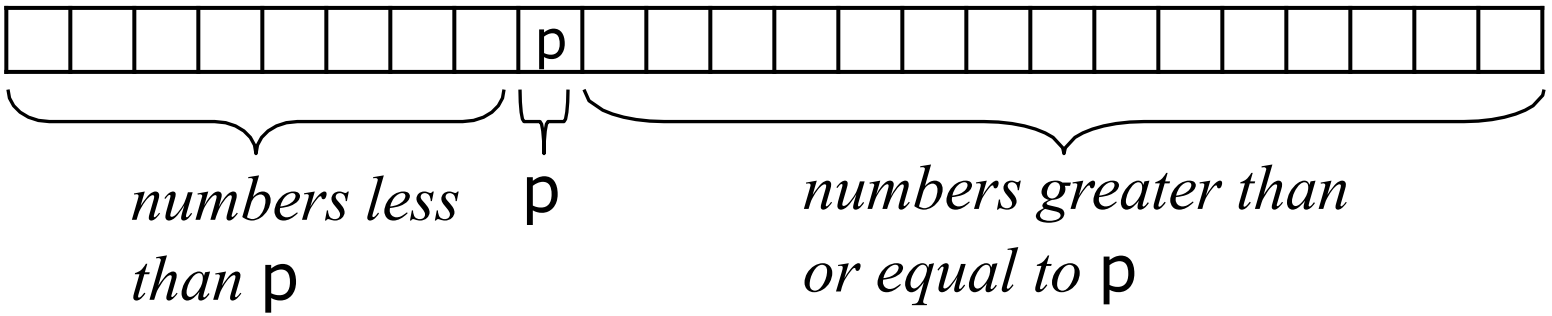
We can back track to the first function call and everything in the left side are sorted!



Partitioning

A key step in the Quick sort algorithm is partitioning the array

- We choose some (any) number **p** in the array to use as a pivot
  - We partition the array into three parts:



Implementation

```
void quickSort( Comparable [ ] arr, int low, int high ){

   if( low < high ){
      int pivotIndex = partition( arr, low, high);
      quickSort( arr, low, pivotIndex - 1 );
      quickSort( arr, pivotIndex + 1, high );
   } //low => high (arr has 1 or 0 elements, already sorted)
}

int partition( Comparable [ ] array, int low, int high ){

   // the function should choose a pivot then reorder the
   // array around pivot and return the index of pivot.




}
```

Implementation

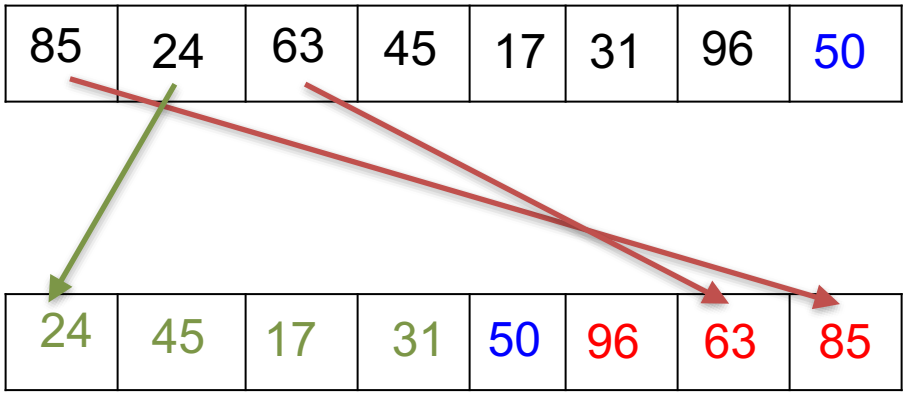
How to implement `partition()`?

(everything less than pivot to place to the left of pivot and everything greater than pivot to place to the right of pivot)

Implementation

Portioning using extra array

- –Assume 50 is the pivot
- –Iterate through the elements, Put the smaller elements in front of the temporary array and the larger element in the back
- –Once we are done, we copy the pivot, 50, into the resulting hole



Implementation

It is not in-place

Can we implement quicksort in place?

Implementation

First, we choose the pivot

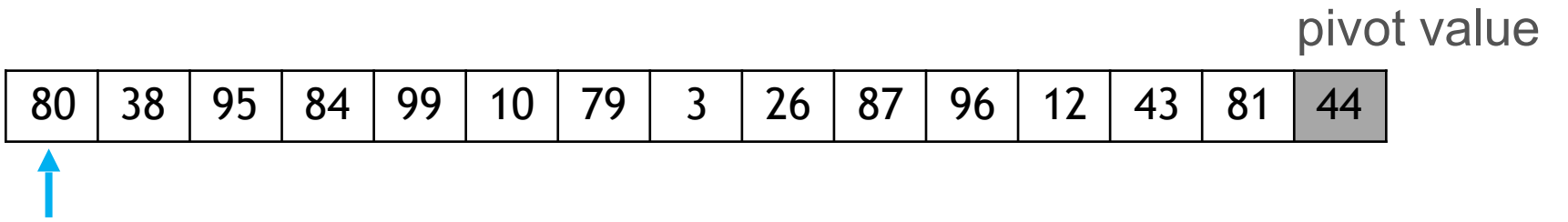
- –whatever you choose to be the pivot make sure to move it to the last index
- –For example I randomly choose index 7, I move it to the end (swapping them)

Implementation

Next, recall that our goal is to partition all remaining elements based on whether they are smaller than or greater than the pivot (and find the pivot index)

Implementation

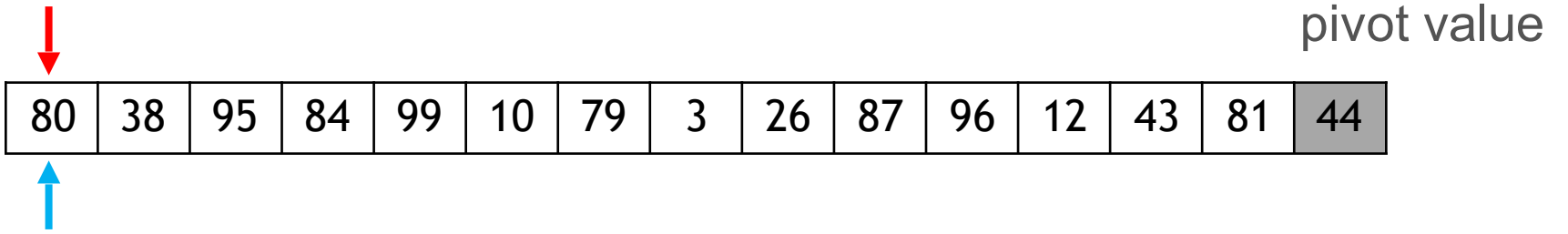
Initially, we assume that pivot-index is at the front



pivotIndex = 0

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex

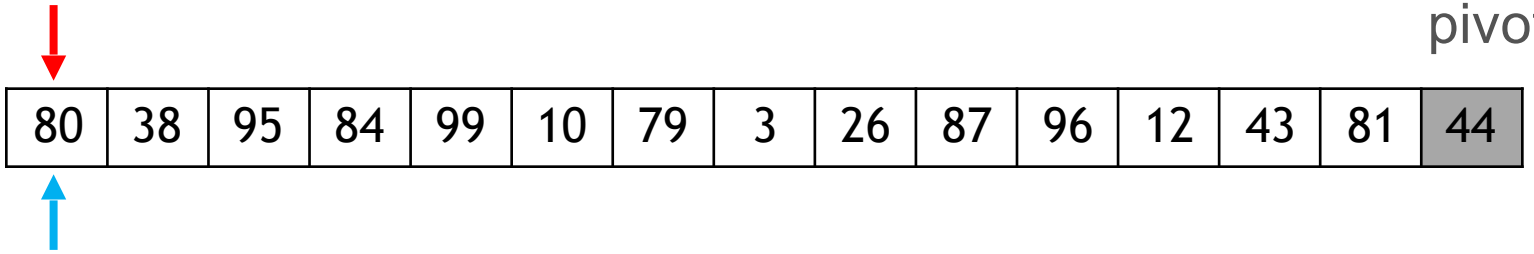


pivotIndex = 0

i = 0

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 0

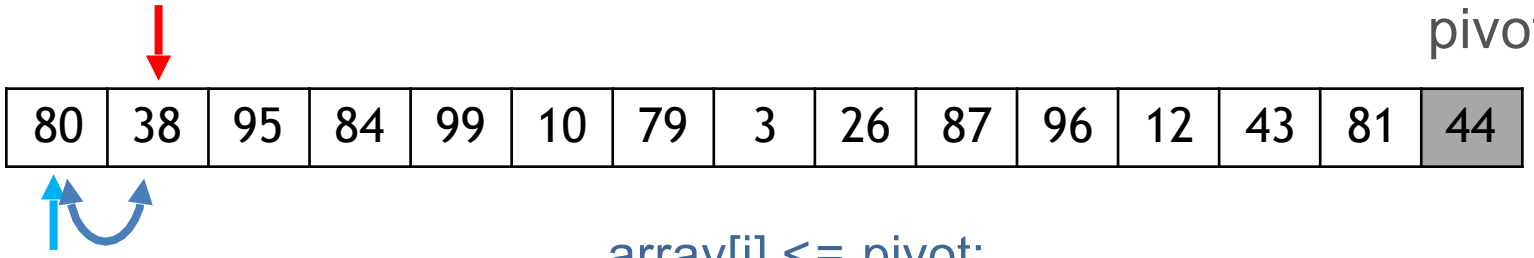
i = 0

array[i] > pivot:

do nothing (just go to the next index)

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] <= pivot:

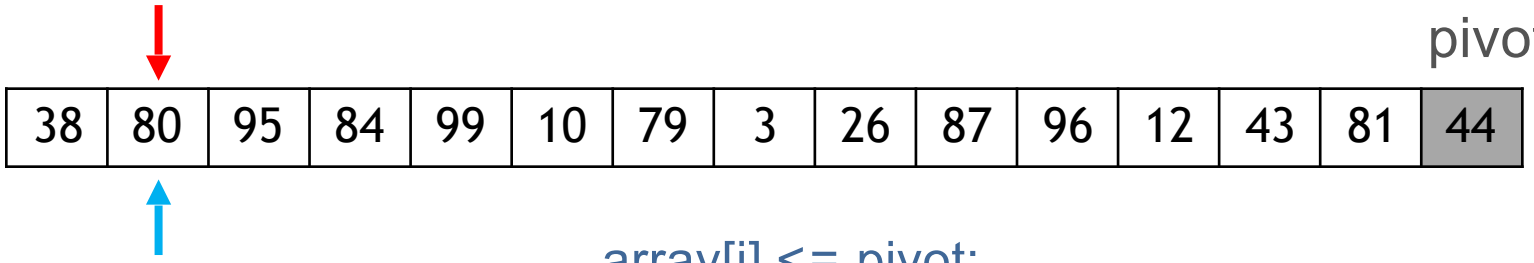
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 0

i = 1

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 1

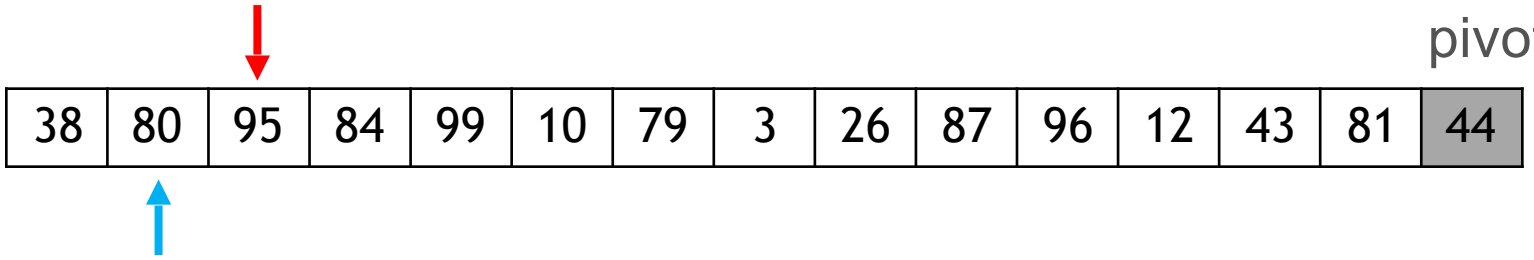
i = 1

array[i] <= pivot:

swap index `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

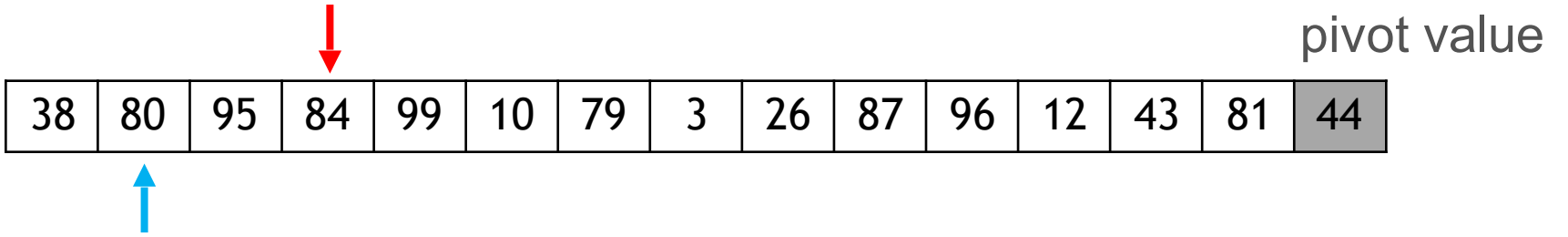
do nothing

pivotIndex = 1

i = 2

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

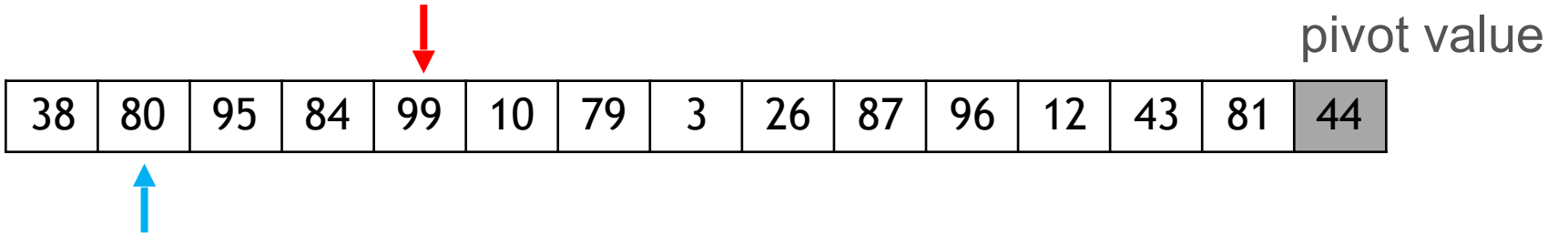
do nothing

pivotIndex = 1

i = 3

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

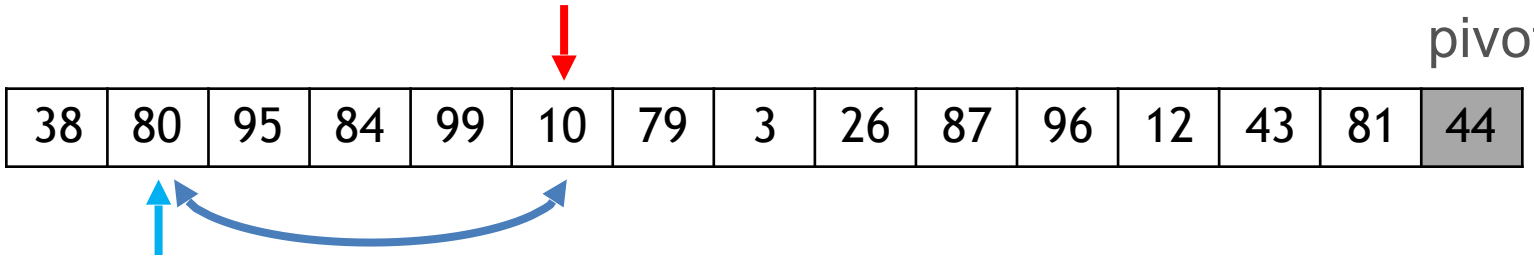
do nothing

pivotIndex = 1

i = 4

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 1

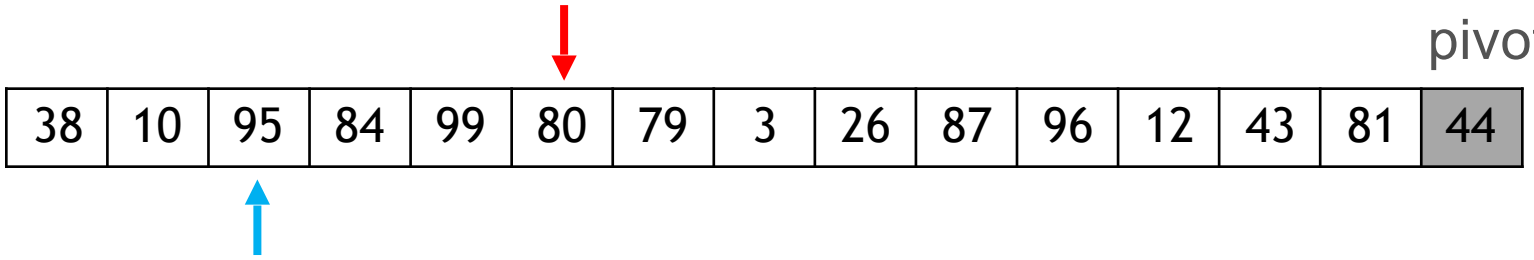
i = 5

array[i] <= pivot:

swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 2

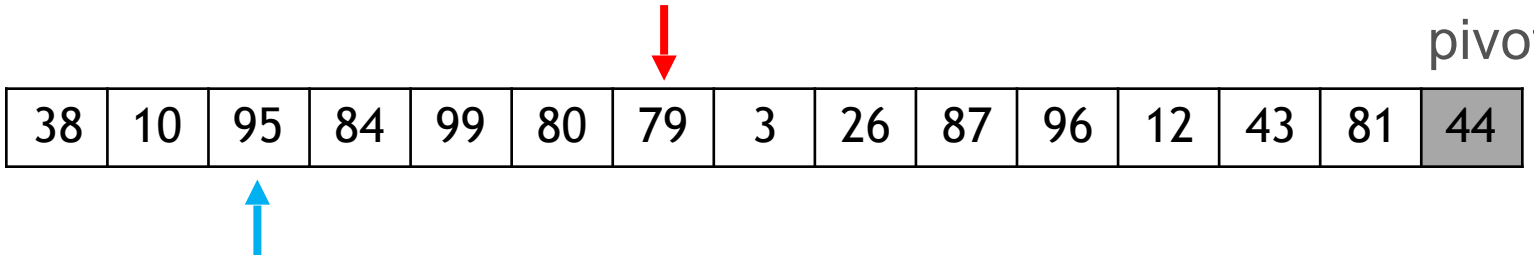
i = 5

array[i] <= pivot:

swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

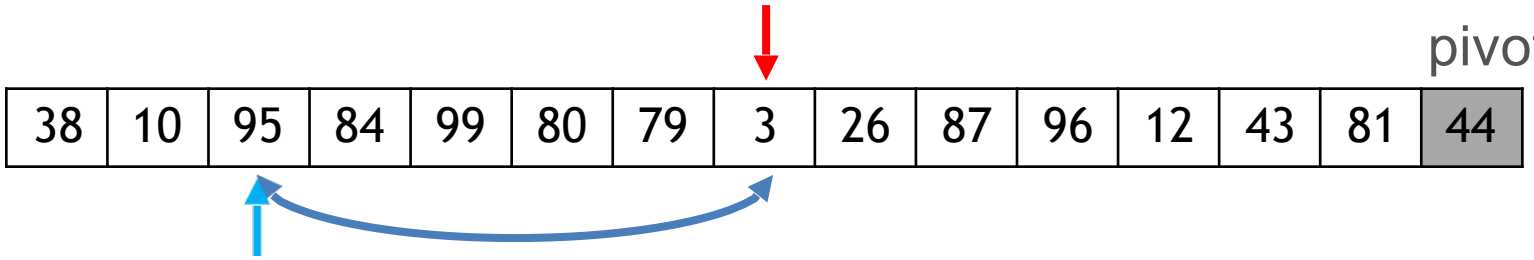
do nothing

pivotIndex = 2

i = 6

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 2

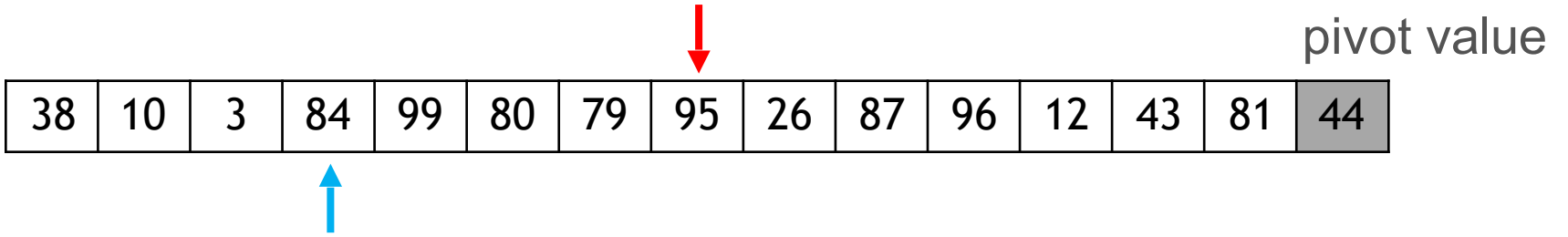
i = 7

array[i] <= pivot:

swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 3

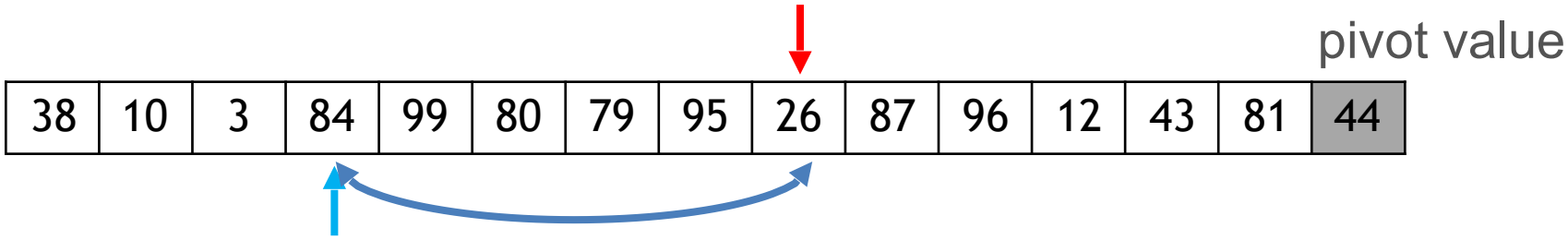
i = 7

array[i] <= pivot:

swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] <= pivot:

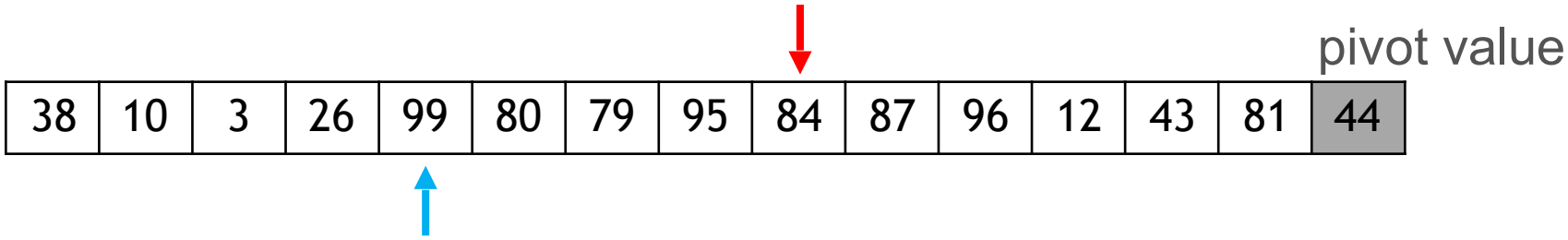
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 3

i = 8

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 4

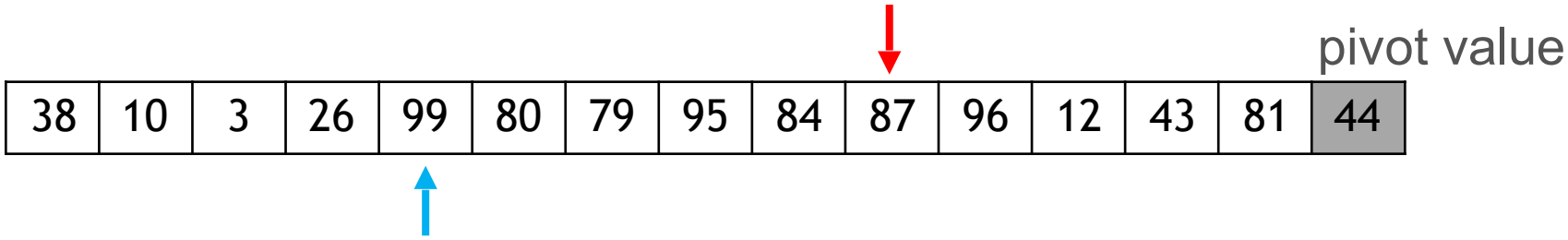
i = 8

array[i] <= pivot:

swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

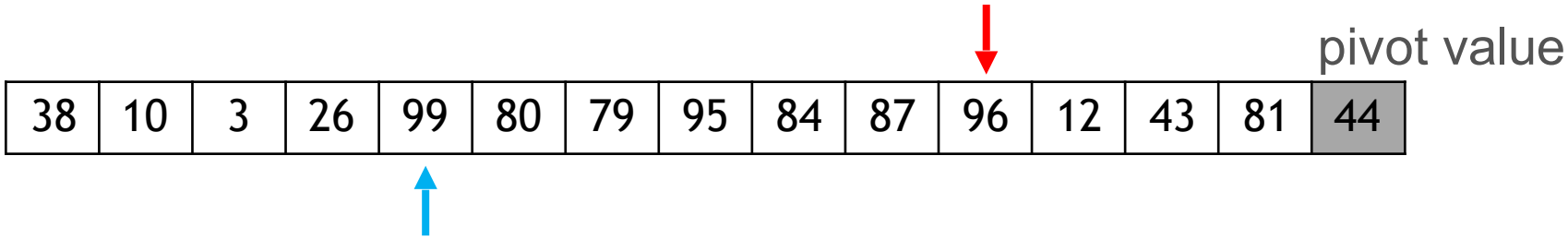
do nothing

pivotIndex = 4

i = 9

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

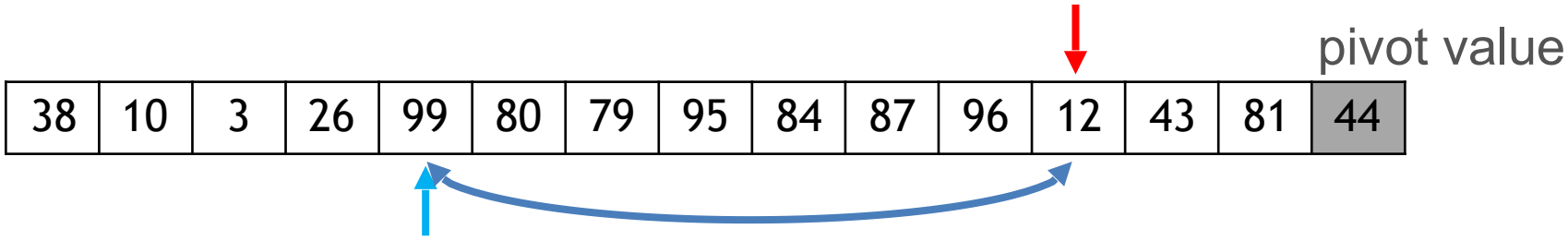
do nothing

pivotIndex = 4

i = 10

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] <= pivot:

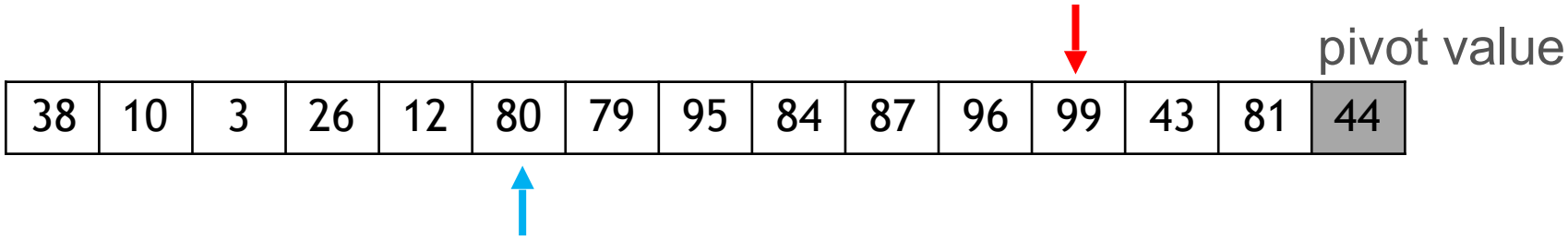
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 4

i = 11

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] <= pivot:

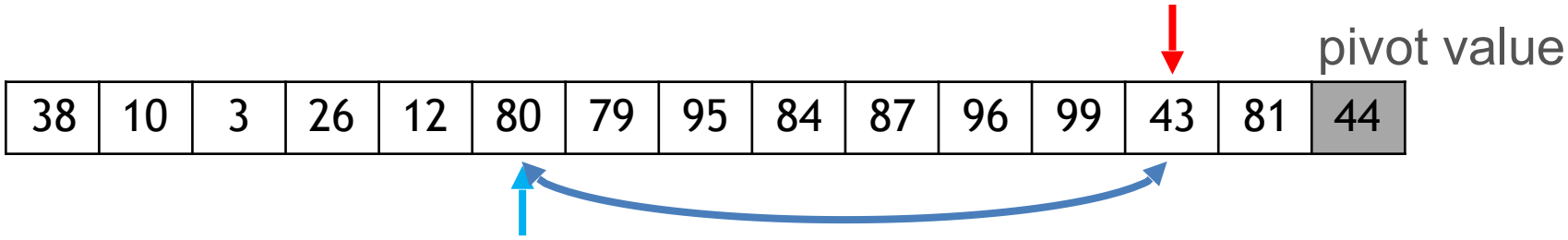
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 5

i = 11

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] <= pivot:

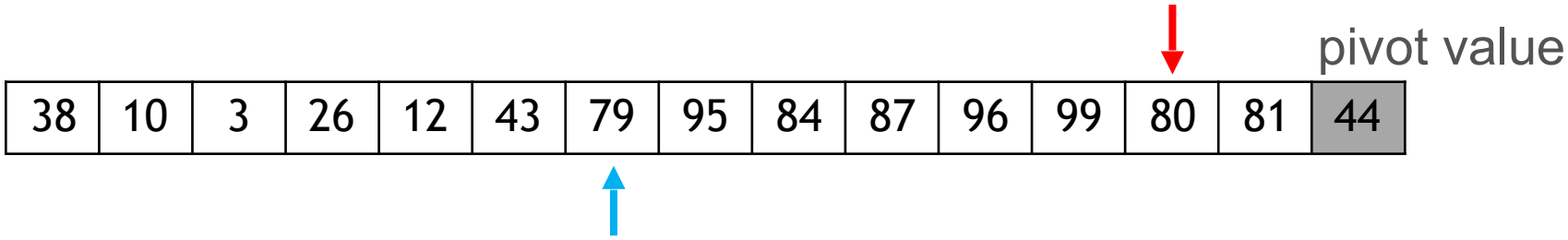
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 5

i = 12

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first-index to last-index-1
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] <= pivot:

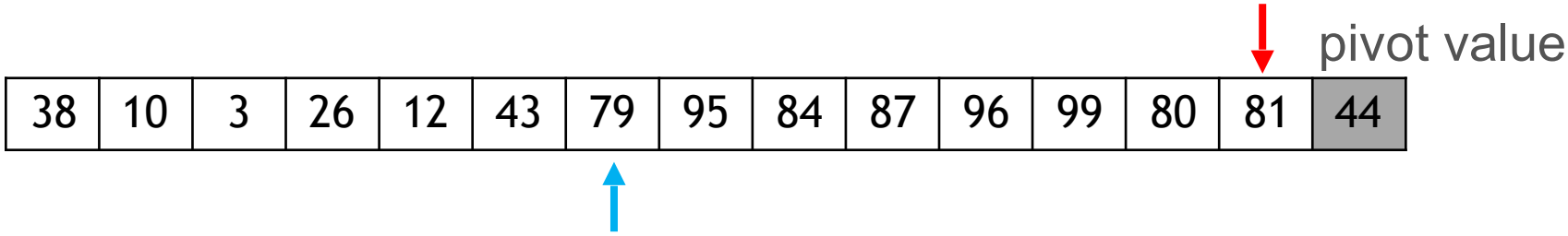
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 6

i = 12

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first index to last index-1
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



array[i] > pivot:

do nothing

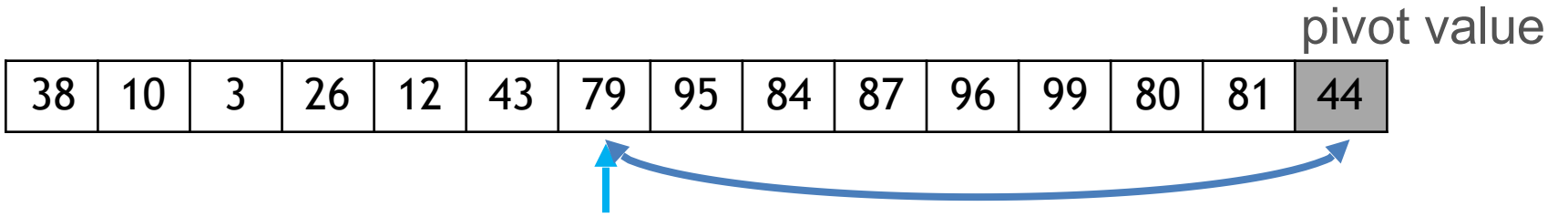
pivotIndex = 6

i = 13

we reached to the last index - 1 so we should stop

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first index to last index-1
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



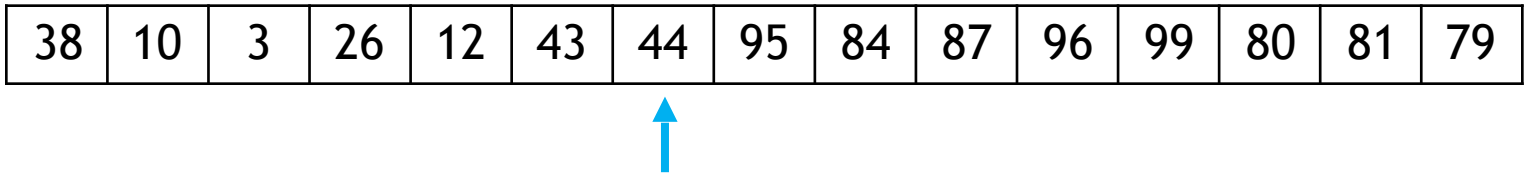
pivotIndex = 6

everything before pivotIndex are smaller than pivot, everything after `pivotIndex` are greater than pivot

Just need to put the pivot to correct index (make sure you do not loose the value at the current `pivotIndex`)

Implementation

- Initially, we assume that pivot-index is at the front
- Scan the whole list, from first index to last index-1)
- make sure that all the elements lesser than pivot are in the left of pivotIndex and all the elements greater than pivot are to the right of pivotIndex



pivotIndex = 6

Just need to put the pivot to correct index (make sure you do not loose the value at the current `pivotIndex`)

Note that after the last step, 79 is in correct position according to pivot (44)!

Quicksort example

Let's continue our previous example

The original array:

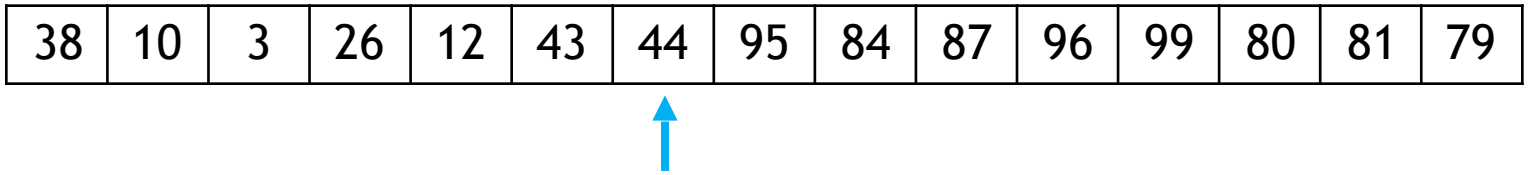
80	38	95	84	99	10	79	44	26	87	96	12	43	81	3

We called partition(array, 0, 14), which portioned array and returned 6 as pivot index

38	10	3	26	12	43	44	95	84	87	96	99	80	81	79

Quicksort example

We called `quicksort(array, 0, 14)`




```
int pivotIndex = partition( array, 0, 14);
quickSort( array, 0, 5 );
quickSort( array, 7, 14 );
```

`quicksort( array,  0, 14 )`

Quicksort example

We are calling `quicksort(array, 0, 5)`


```
quicksort( array,   0, 5 )
quicksort( array,   0, 14 )
```

Quicksort example

We are calling `quicksort(array, 0, 5)`


```
call partition( array, 0, 5);
```

```
quicksort( array,  0, 5 )
quicksort( array,  0, 14 )
```

Quicksort example

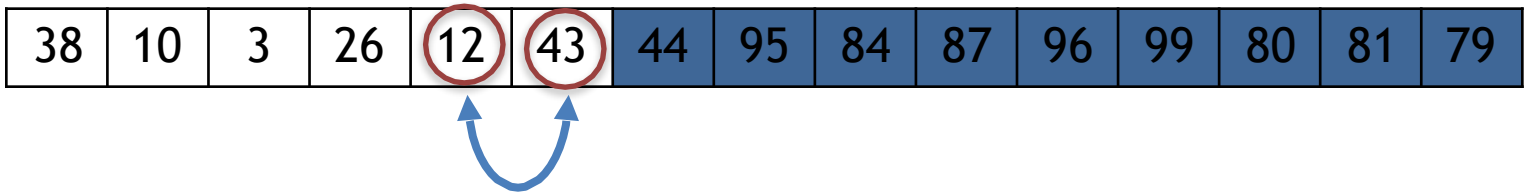
We are calling `partition(array, 0, 5)`

`choose a pivot, 12`

```
partition( array,   0, 5 )
quicksort( array,   0, 5 )
quicksort( array,   0, 14 )
```

Quicksort example

We are calling `partition(array, 0, 5)`



choose a pivot, 12

move pivot to the last index and continue partitioning

```
partition( array,  0, 5 )
quicksort( array,  0, 5 )
quicksort( array,  0, 14 )
```

Choosing Pivot

It turns out that the selection of pivot is crucial for performance of Quick Sort

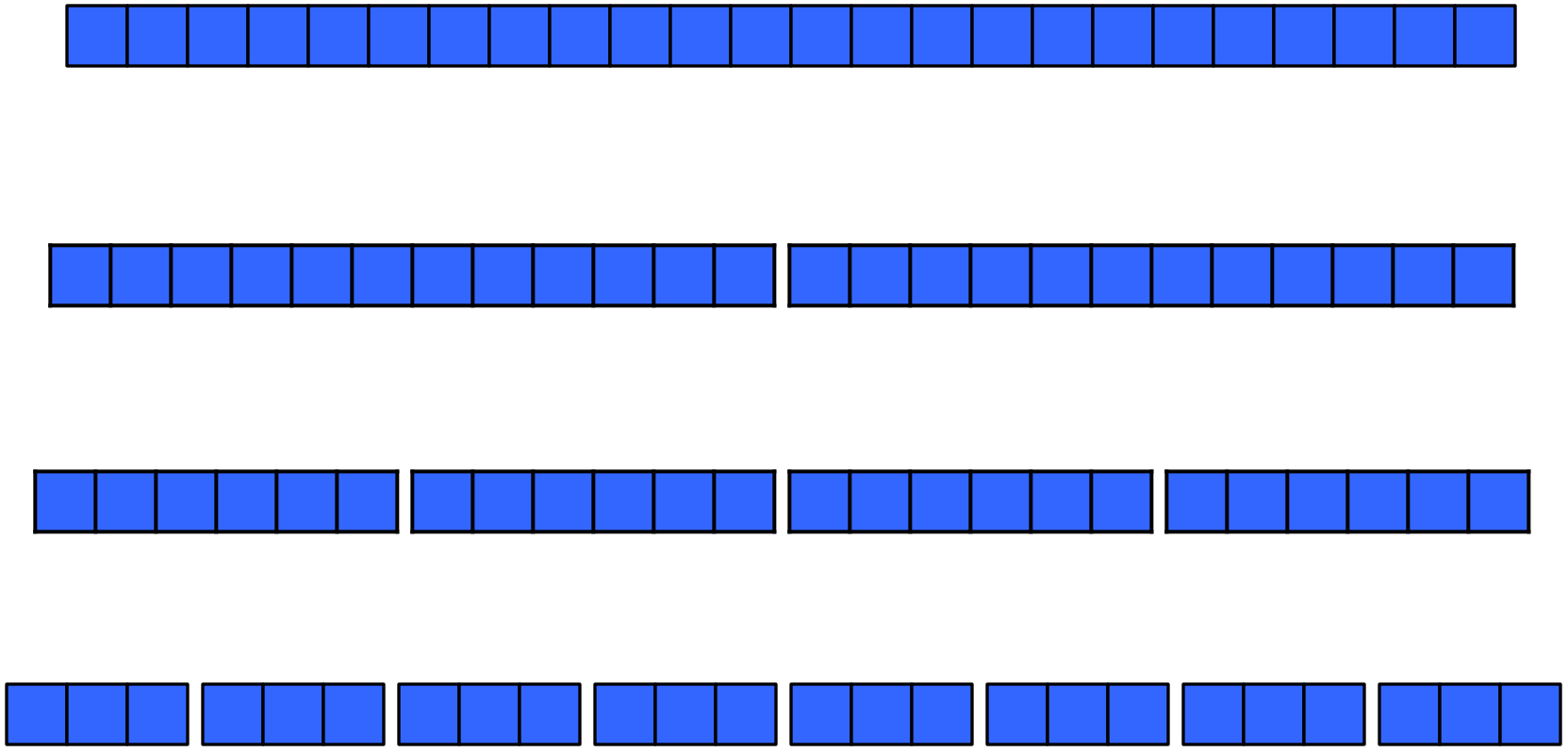
Let’s see the analysis of running time first

Run-time analysis

How much time do we need to partition an array of size *n?* Θ(n)

Run-time analysis

Best case



Run-time analysis

When could the best case happen? when the pivot is the median

The running time (time cost) can be expressed with the following recurrence:

$$T(n) = 2.T(n/2) + T(\text{partitioning array of size } n)$$
$$= 2.T(n/2) + \Theta(n)$$

The same recurrence as for merge sort, i.e., T(n) is of order $\Theta(n.\log n)$.

Run-time analysis

How much time do we need to partition an array of size *n?* Θ(n)

In the best case, the list will be split into two approximately equal sub-lists, and thus, the run time could be very similar to that of merge sort: $\Theta(n \ln(n))$

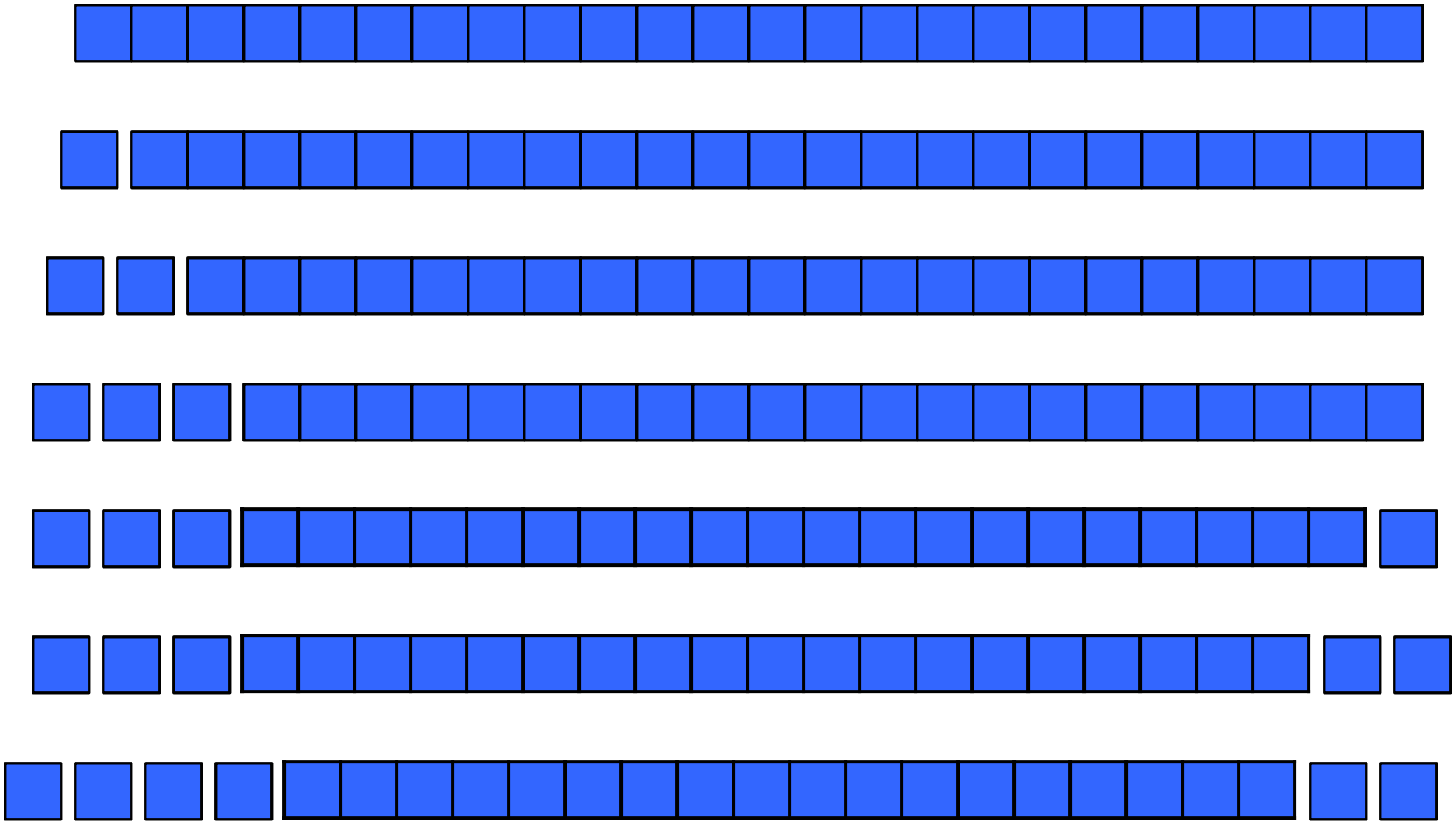
What happens if we don't get that lucky?

Worst-case scenario

In the worst case, partitioning always divides the size n array into these three parts:

- A length one part, containing the pivot itself
- A length zero part, and
- A length n-1 part, containing everything else

Worst-case scenario



Worst-case scenario

In the worst case, partitioning always divides the size n array into these three parts:

- A length one part, containing the pivot itself
- A length zero part, and
- A length n-1 part, containing everything else

When could this happen? Example: the array is sorted and the pivot is selected to be the first or the last element.

The run time is $T(n) = T(n-1) + \Theta(n)$

Worst-case scenario

$T(n) = T(n-1) + \Theta(n)$

we can rewrite it as:

$T(n) = T(n-1) + c*n$

we can expand it:

$T(n) = T(n-2) + c*(n-1) + c*n$

keep expanding it:

$T(n) = T(1) + c*(2) \ldots + c*(n-2) + c*(n-1) + c*n$

$T(n) = c + c*(2) \ldots + c*(n-2) + c*(n-1) + c*n$

$T(n) = \Theta(n^2)$ (why?)

Median-of-three

It is difficult to find the median so consider another strategy:

–Choose the median of the first, middle, and last entries in the list

This will usually give a better approximation of the actual median

Median-of-three

Partitioning the elements based on 44 results in two sub-lists, each of which must be sorted (again, using quicksort)

Select the 26 to partition the first sub-list:

Select 81 to partition the second sub-list:

Implementation of `partition()`

We should update the implementation of partitioning so that we first examine the first, middle, and last entries and chosen the median of these to be the pivot

And then:

–move the median entry to the last index

Memory Requirements

The additional memory required is $\Theta(\ln(n))$

- Memory need for recursive calls (stack frames)
- Each recursive function call places its local variables, parameters, *etc*., on a stack
  - The depth of the recursion tree is $\Theta(\ln(n))$
- What if the worst case scenario happens?

Memory Requirements

The additional memory required is $\Theta(\ln(n))$

- Memory need for recursive calls (stack frames)
- Each recursive function call places its local variables, parameters, *etc*., on a stack
  - The depth of the recursion tree is $\Theta(\ln(n))$
- What if the worst case scenario happens?
- Unfortunately, if the run time is $\Theta(n^2)$, the memory use is $\Theta(n)$

Run-time Summary

To summarize the two $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Merge Sort	$\Theta(n \ln(n))$		$\Theta(n)$	
Quicksort	$\Theta(n \ln(n))$	$\Theta(n^2)$	$\Theta(\ln(n))$	$\Theta(n)$

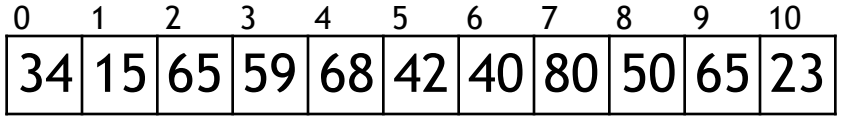
Summary

This topic covered quicksort

- On average faster than merge sort (and other $\Theta(n \ln(n))$ sorting algorithms)
- Uses a pivot to partition the objects
- Using the median of three entries is a reasonable mean for finding the pivot
- Average run time of $\Theta(n \ln(n))$ and $\Theta(\ln(n))$ memory
- Worst case run time of $\Theta(n^2)$ and $\Theta(n)$ memory

Example

Sort the following list using quicksort



Further modifications

Our implementation is by no means optimal:

An excellent paper on quicksort was written by Jon L. Bentley and M. Douglas McIlroy:
Engineering a Sort Function

found in Software—Practice and Experience, Vol. 23(11), Nov 1993