COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

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

Quick Sort

Strategy

We have seen two $\Theta(n \ln(n))$ sorting algorithms:

- Heap sort which allows in-place sorting, and
- Merge sort which is faster but requires more memory

We will now look at a recursive algorithm which may be done *almost* in place but which is faster than heap sort

- 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 sub-lists and sorts them

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

Consider the following alternative:

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

Quicksort

For example, given

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

we can select the middle entry, 44, and sort 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

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

Quicksort

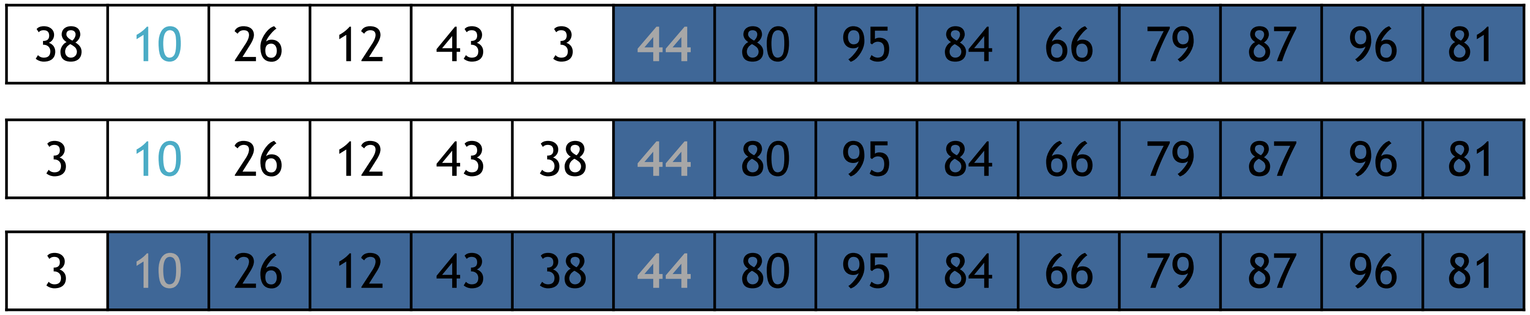
order the remaining entries to two parts: <10 and >10

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

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

Quicksort

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



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

Quicksort

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

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

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

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

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

Quicksort

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

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

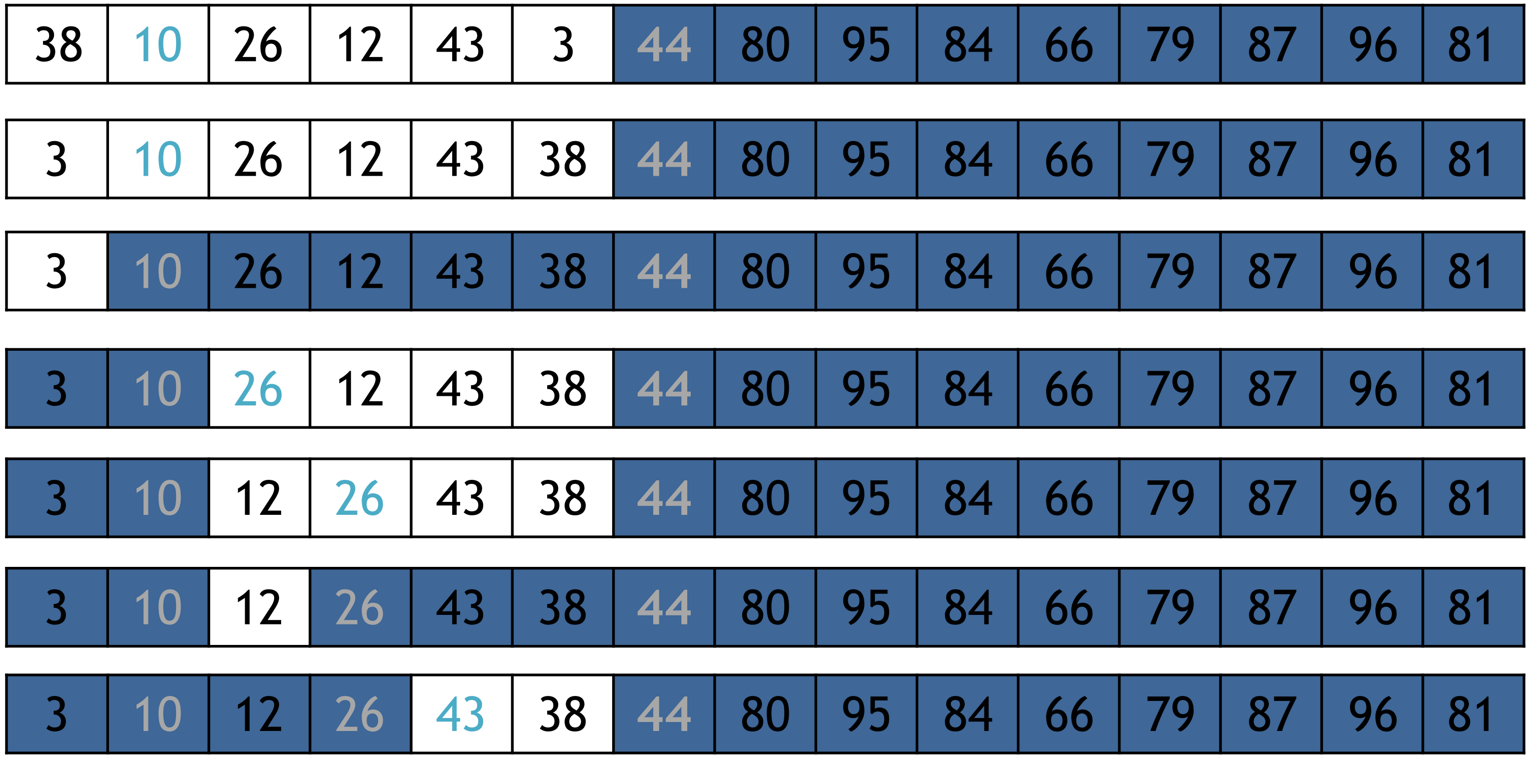
Quicksort

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

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

Quicksort

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



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

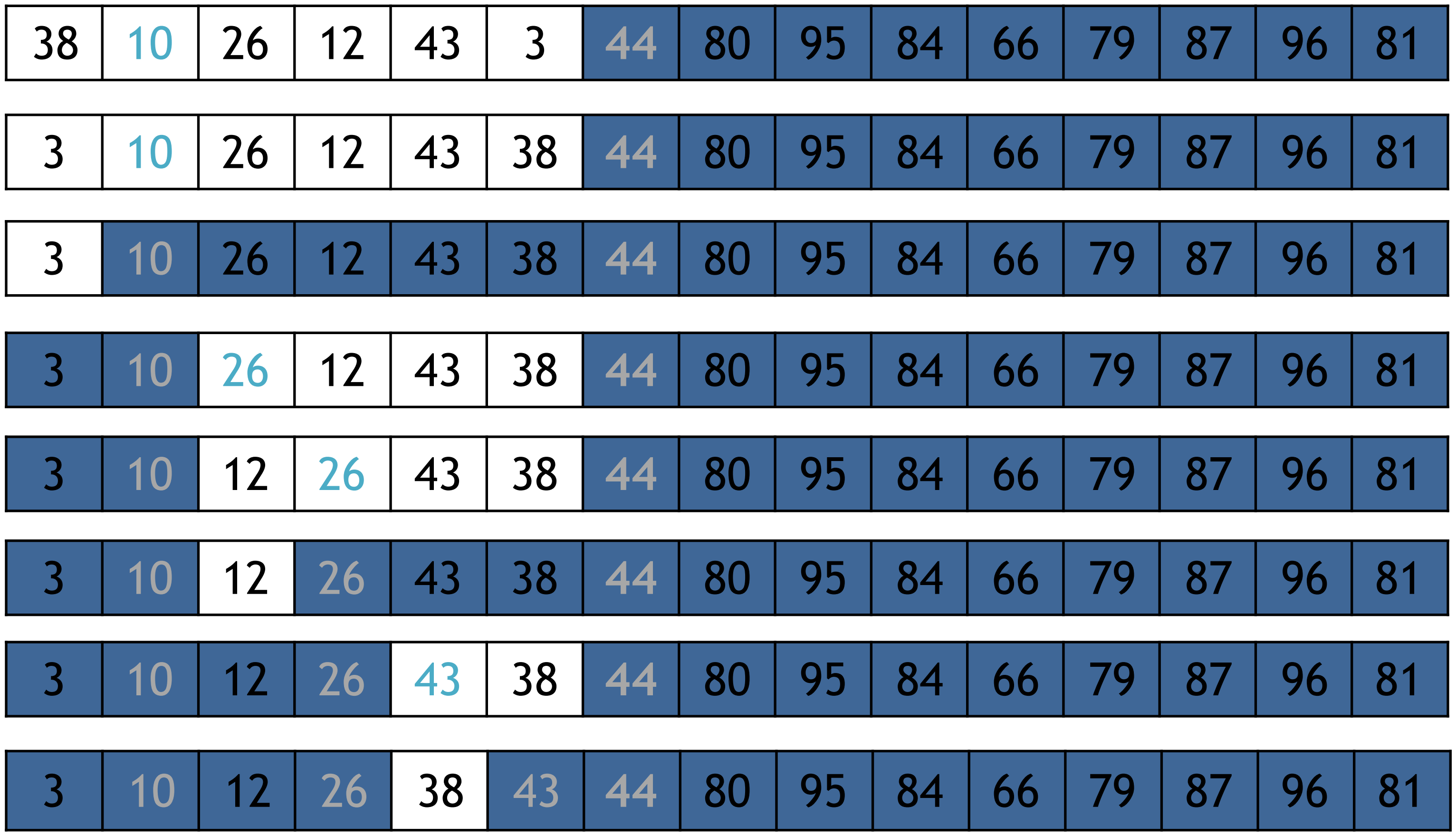
Quicksort

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

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

Quicksort

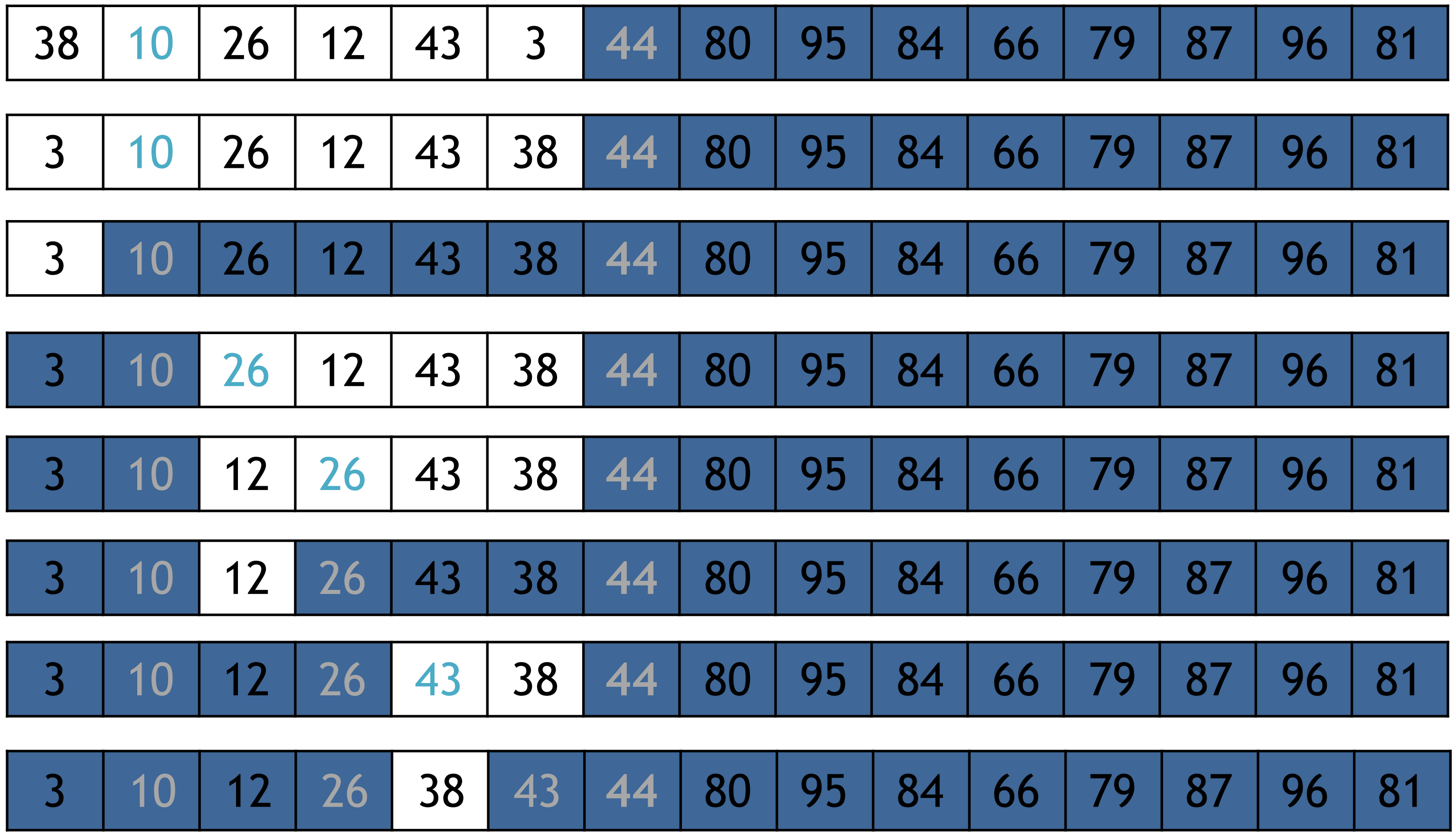
Call the quick sort on the left part, it is just one element which is sorted!



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

Quicksort

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



Quick sort

If the list is always split into two approximately equal sub-lists, the run time could be very similar to that of merge sort: $\Theta(n \ln(n))$

So why quick sort?!

Run-time analysis

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

Suppose we choose the first element as our pivot and we try ordering a sorted list:

80	38	95	84	66	10	79	**2**	26	87	96	12	43	81	3

Using 2, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3

We still have to sort a list of size $n - 1$

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

Worst-case scenario

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

we can rewrite it as:

$T(n) = T(n - 1) + c*\mathrm{n}$

we can expand it:

$T(n) = T(n - 2) + c*(\mathrm{n}-1) + c*\mathrm{n}$

keep expanding it:

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

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

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

Worst-case scenario

Our goal is to choose the median element in the list as our pivot:

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

Unfortunately, it's difficult to find

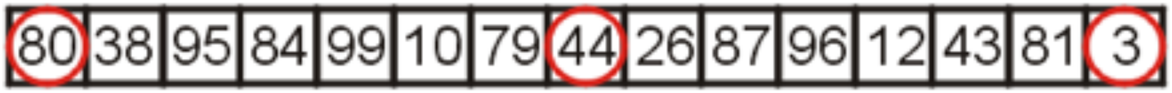
Alternate strategy: take the median of a subset of entries

- For example, take the median of the first, middle, and last entries

Median-of-three

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

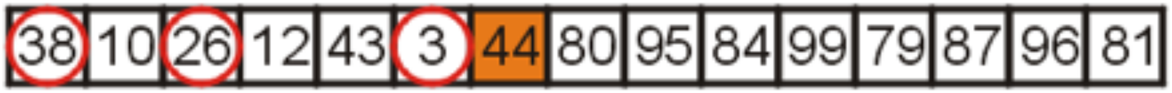
- Choose the median of the first, middle, and last entries in the list



This will usually give a better approximation of the actual median

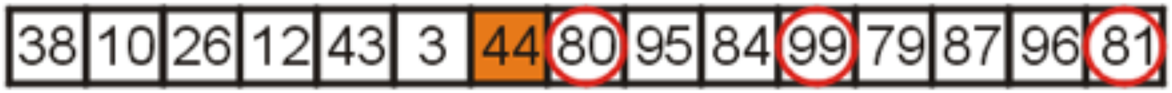
Median-of-three

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



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

Select the 26 to partition the first sub-list:



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

Select 81 to partition the second sub-list:

Implementation

```
void quickSort( Comparable [ ] array, int first, int last ){

   if( first < last ){
      int pivotIndex = partition( array, first, last);
      quickSort( array, first, pivotIndex - 1 );
      quickSort( array, pivotIndex + 1, last );
   } // first => last (base case)
}

int partition( Comparable [ ] array, int first, int last ){

   // the function should find the median and reorder the
   // array around median and return the index of median.




}
```

Implementation

How to implement `partition()`?

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

Implementation

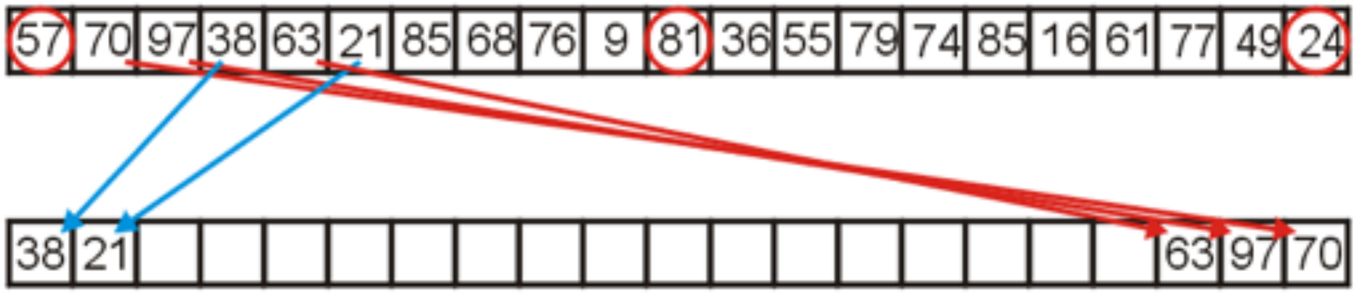
If we choose to allocate memory for an additional array, we can implement the partitioning by copying elements either to the front or the back of the additional array

Finally, we would place the pivot into the resulting hole

Implementation

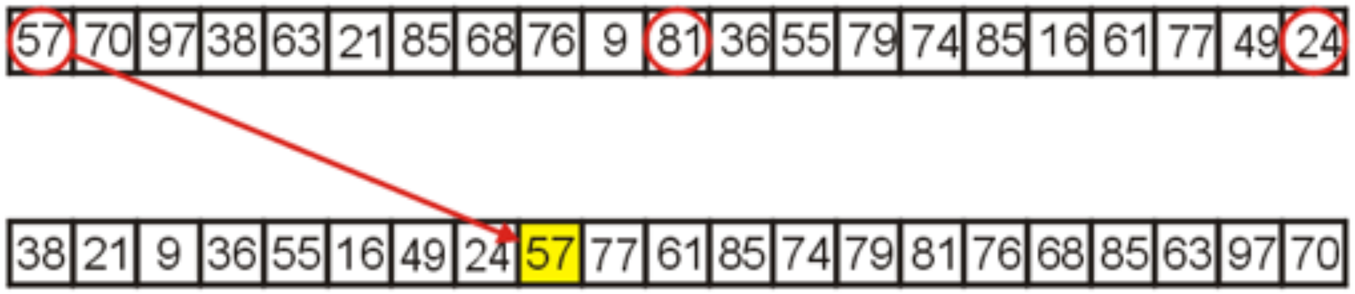
For example, consider the following:

- 57 is the median-of-three
- we go through the remaining elements, assigning them either to the front or the back of the second array



Implementation

Once we are finished, we copy the median-of-three, 57, into the resulting hole



Implementation

Not a good idea!

We can do a better job with merge sort, it always divides the numbers being sorted into two equal or near-equal arrays

Can we implement quicksort in place?

Implementation

First, we have already examined the first, middle, and last entries and chosen the median of these to be the pivot

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

In addition, we can:

– move the median entry to the last index

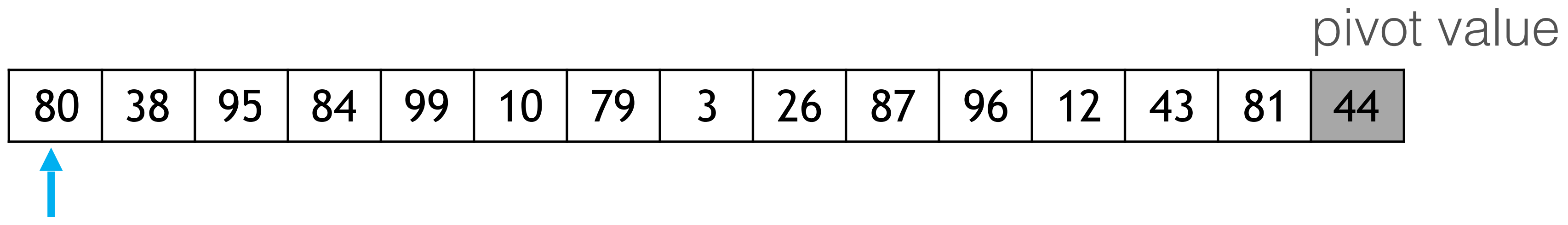
80	38	95	84	99	10	79	3	26	87	96	12	43	81	44

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

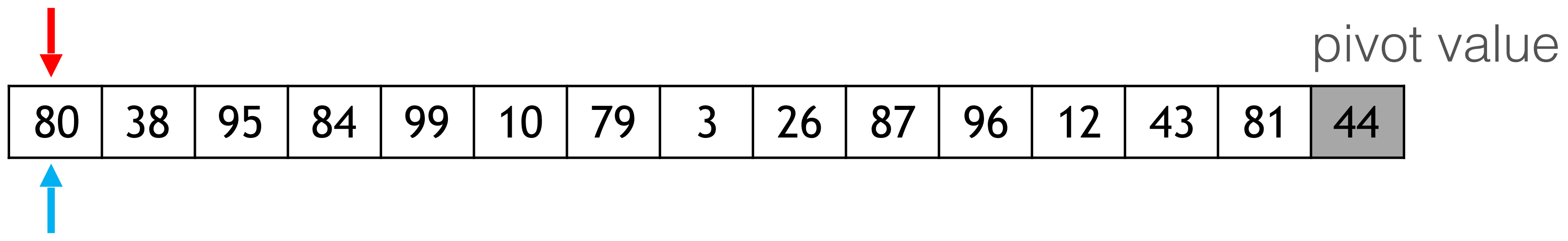
Initially, we assume that pivot-index is at the front



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

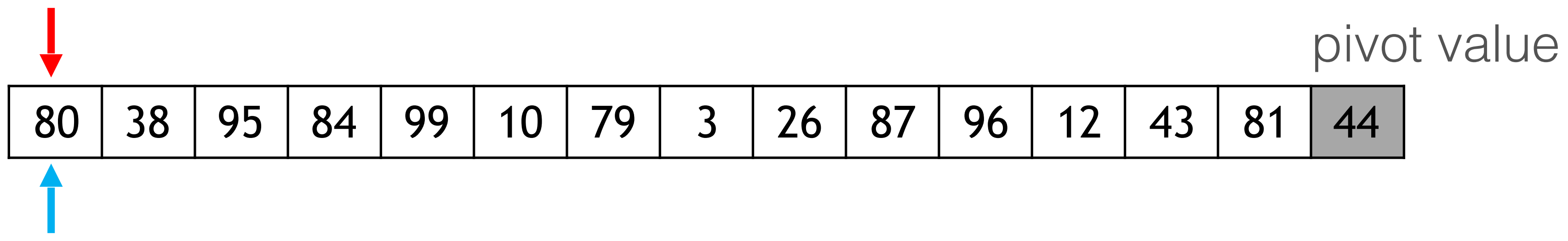


pivotIndex = 0

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



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

pivotIndex = 0

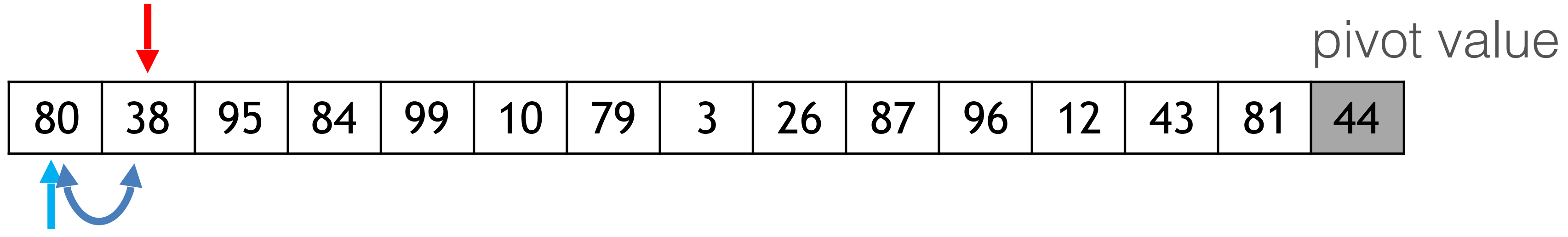
i = 0

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:

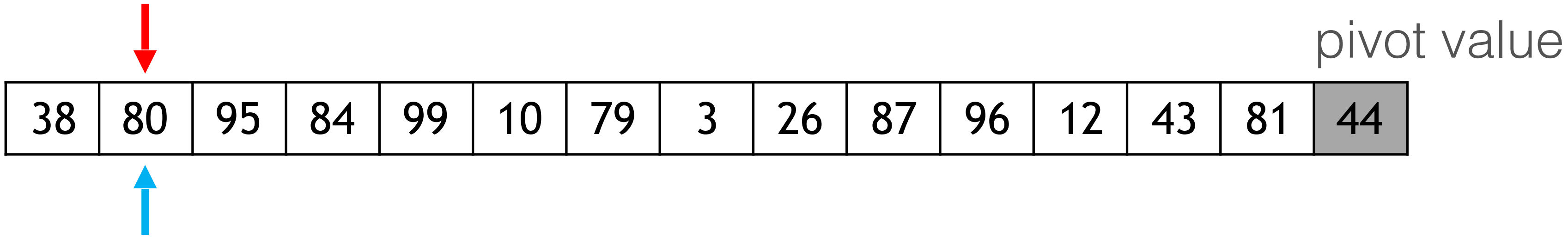
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

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



array[i] <= pivot:

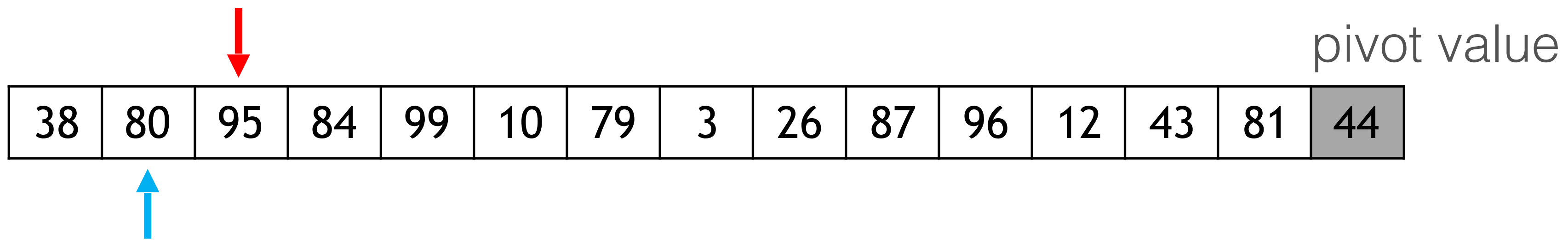
swap index `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 1

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



array[i] > pivot:

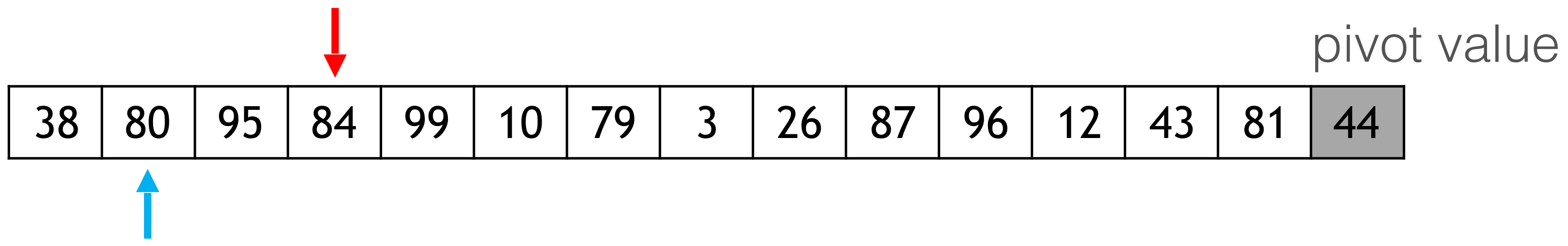
do nothing

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:

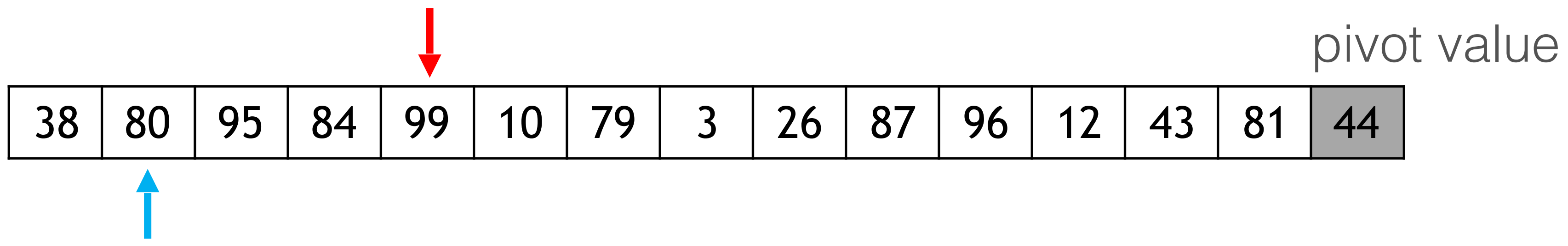
do nothing

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:

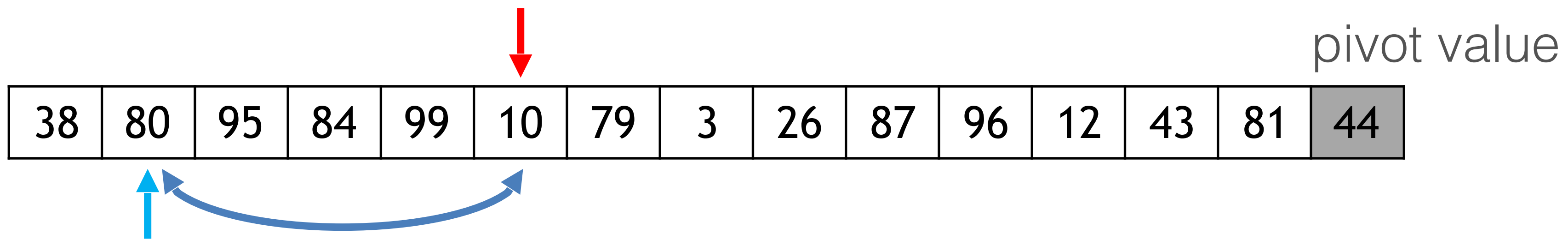
do nothing

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



array[i] <= pivot:

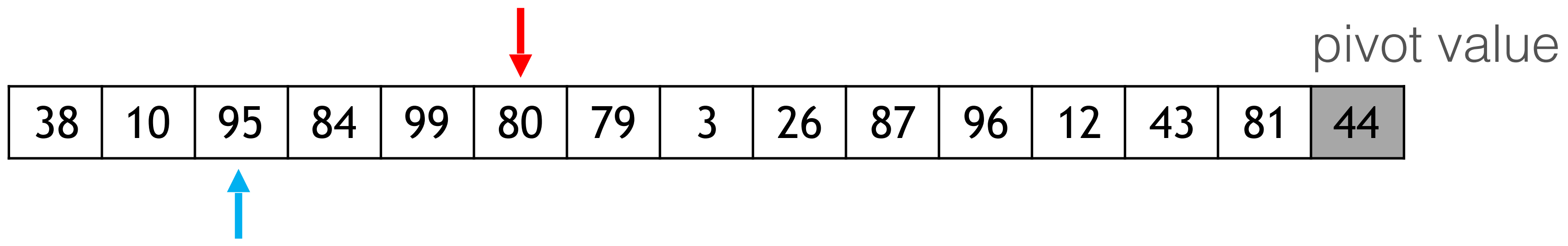
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 1

i = 5

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

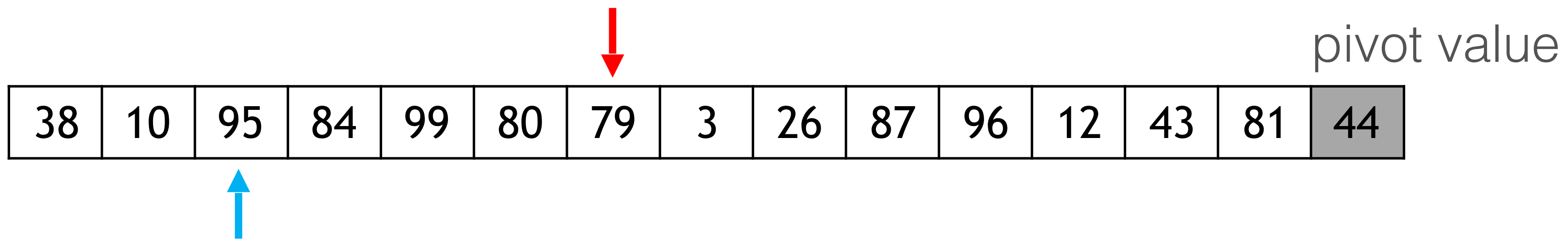
i = 5

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:

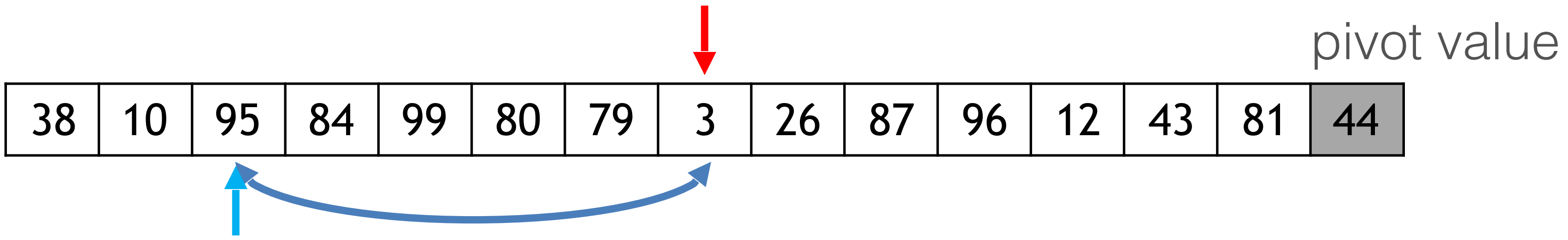
do nothing

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



array[i] <= pivot:

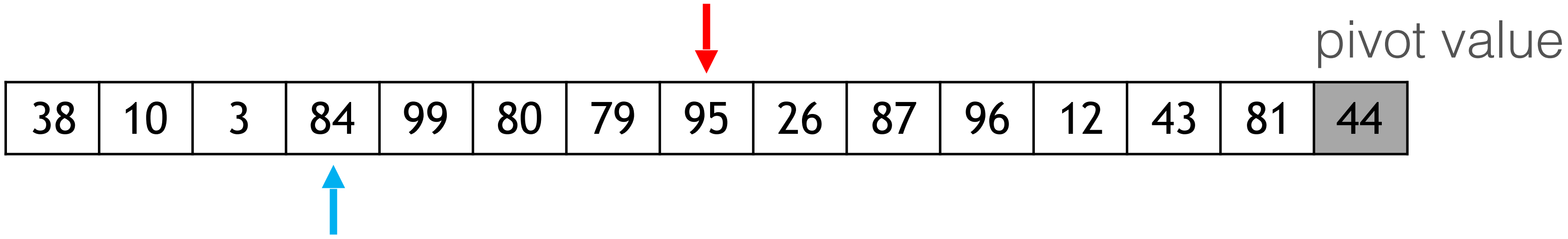
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 2

i = 7

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:

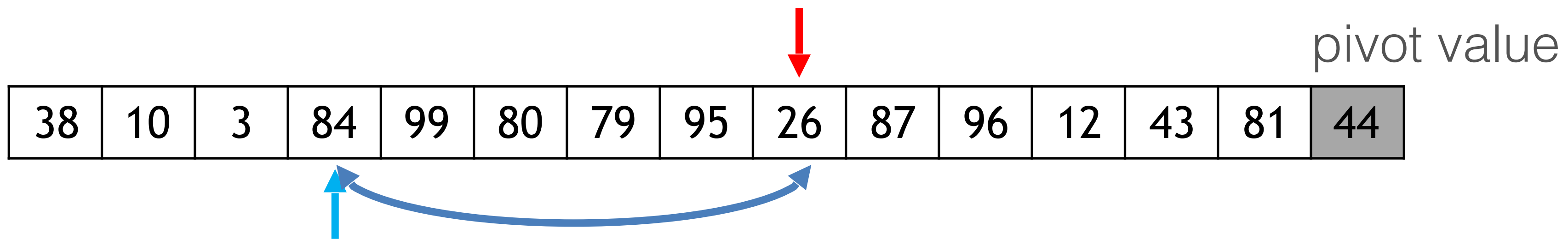
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 3

i = 7

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:

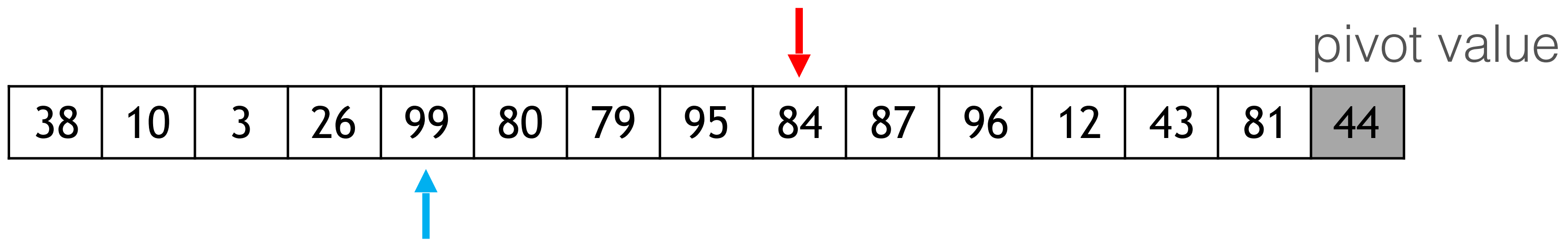
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

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



array[i] <= pivot:

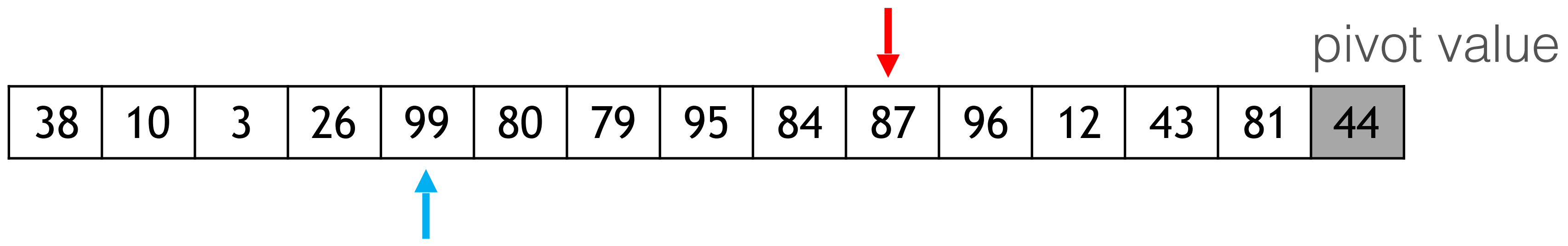
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

pivotIndex = 4

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



array[i] > pivot:

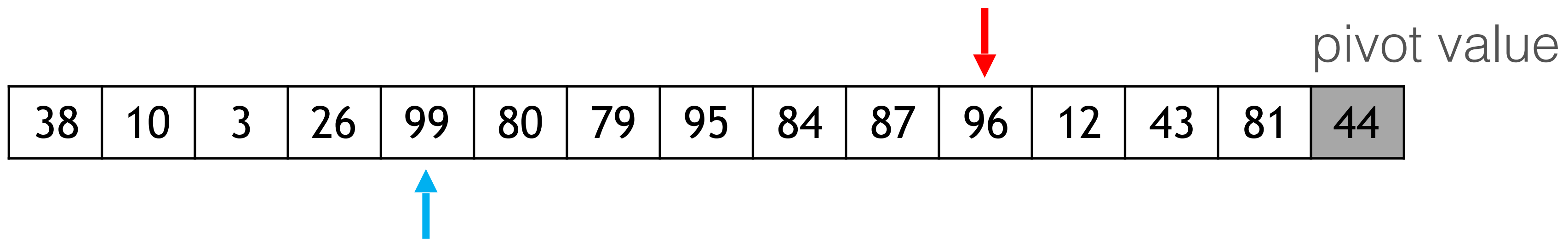
do nothing

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:

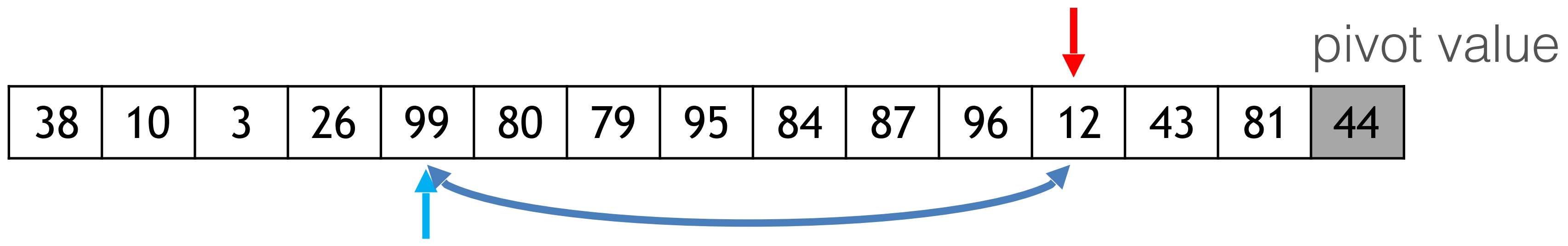
do nothing

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:

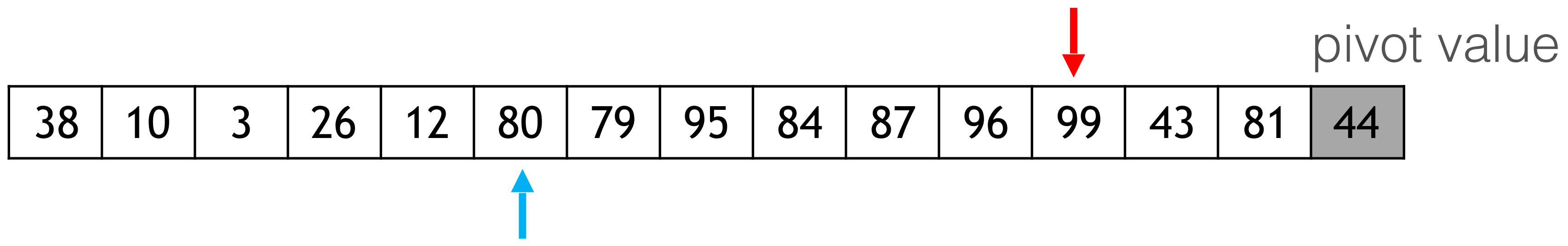
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

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:

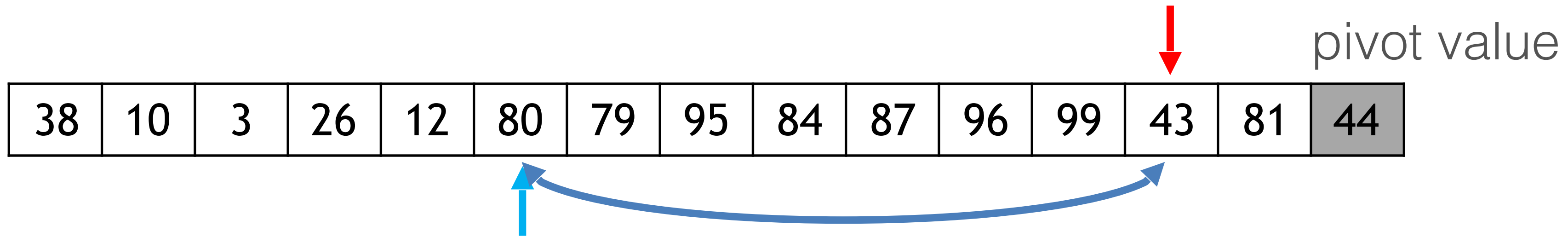
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

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:

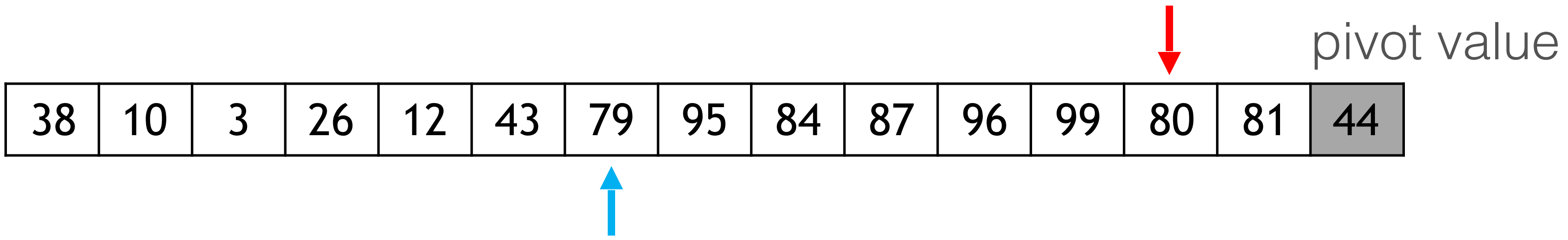
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

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:

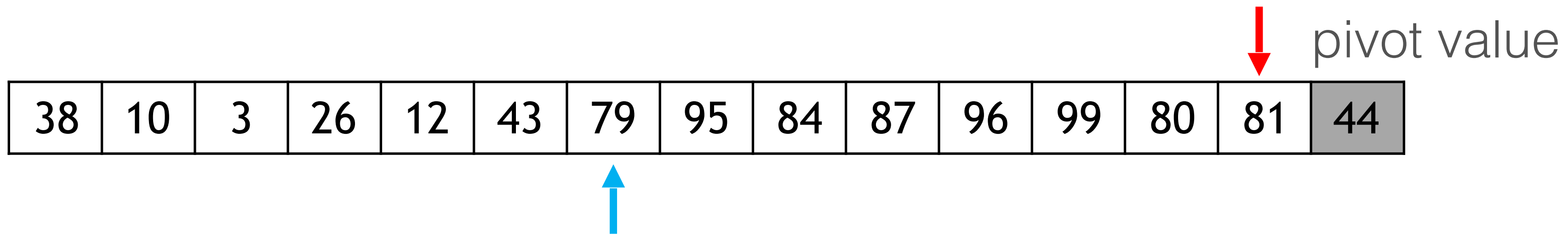
swap `array[i]` and `array[pivotIndex]` and update the `pivotIndex`

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

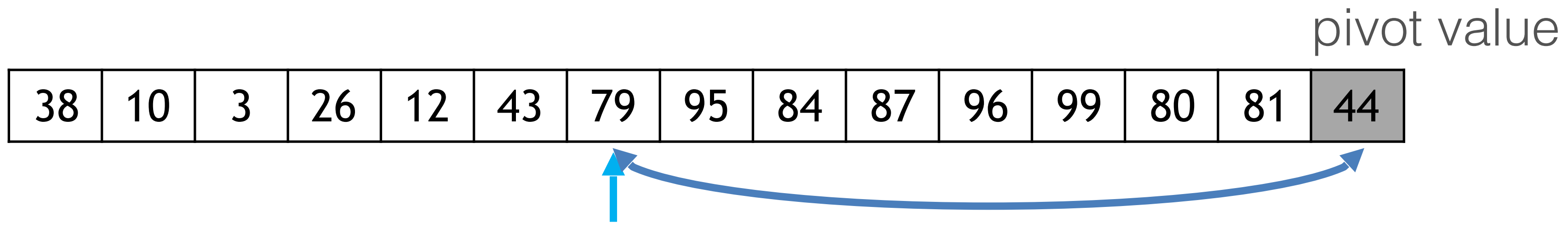
pivotIndex = 6

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



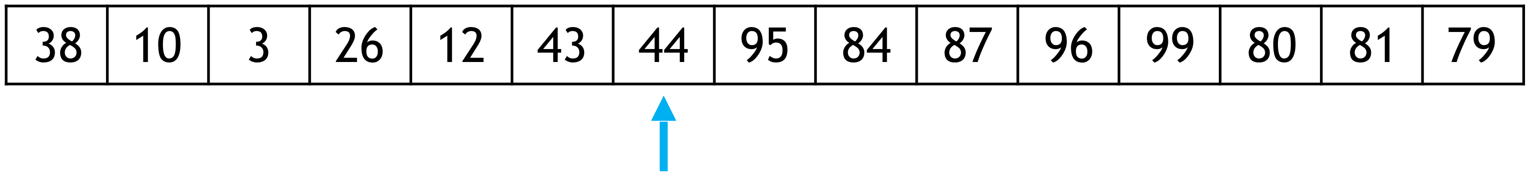
pivotIndex = 6

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`)

Not 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:

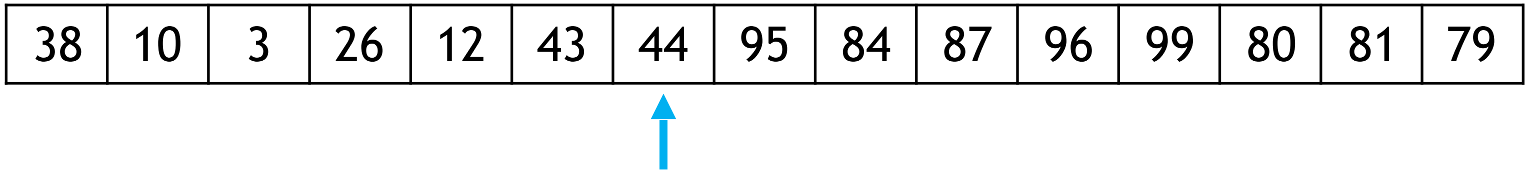
80	38	95	84	99	10	79	44	26	87	96	12	43	81	3

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)`

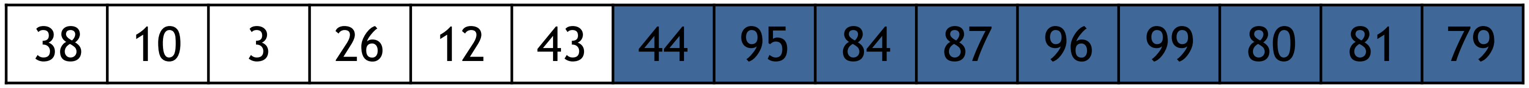


```
int pivotIndex = partition( array, 0, 14);
quickSort( array, 0, 5 );
quickSort( array, 7, 14 );
```

`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)`

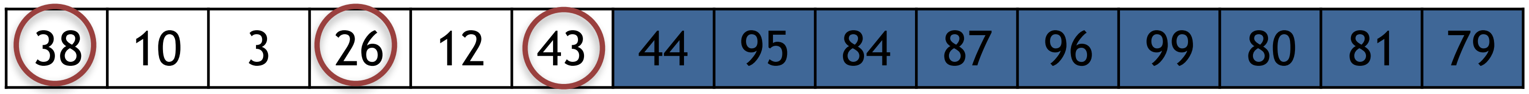
38	10	3	26	12	43	44	95	84	87	96	99	80	81	79

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

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

Quicksort example

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

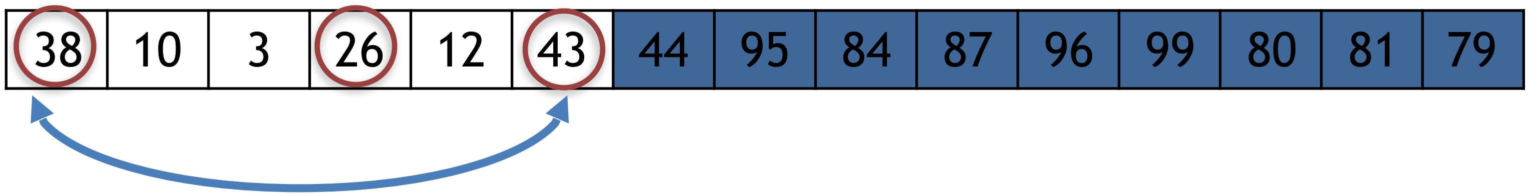


```
find median, 38
```

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

Quicksort example

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



```
find median, 38
copy median to the last index and
   continue partitioning
```

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

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 all three $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Heap Sort	$\Theta(n \ln(n))$		$\Theta(1)$	
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 heap sort or merge sort
- Uses a pivot to partition the objects
- Using the median of three pivots is a reasonably means of 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

0	1	2	3	4	5	6	7	8	9	10
34	15	65	59	68	42	40	80	50	65	23

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