

M.KUMARASAMY COLLEGE OF ENGINEERING (AUTONOMOUS)

PREPERED BY

R.VENGATESH(13BCS2095)

S.SURYA(13BCS2087)

S.VIGNESH(13BCS2096)

V.VIGNESH

BALAKRISHNAN(13BCS2097)

M.VADIVEL(13BCS2089)

GUIDED BY

S.SARAVANAN
(Asst.professor)
Dept. of CSE

Outline

- Definition of Sorting
- Brute force
 - Selection sort
 - Bubble sort
 - Radix sort
- Divide and Conquer
 - Merge sort
 - Quick sort
- Conclusion

Sorting: Definition

Sorting: an operation that segregates items into groups according to specified criterion.

$A = \{ 3 \ 1 \ 6 \ 2 \ 1 \ 3 \ 4 \ 5 \ 9 \ 0 \}$

$A = \{ 0 \ 1 \ 1 \ 2 \ 3 \ 3 \ 4 \ 5 \ 6 \ 9 \}$

Selection Sort

1. We have two group of items:
 - sorted group, and
 - unsorted group
1. Initially, all items are in the unsorted group. The sorted group is empty.
 - We assume that items in the unsorted group unsorted.
 - We have to keep items in the sorted group sorted.

Selection Sort: Cont'd

1. Select the “**best**” (eg. smallest) item from the unsorted group, then put the “**best**” item at the end of the sorted group.
2. Repeat the process until the unsorted group becomes empty.

Selection Sort

5 1 3 4 6 2



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

6

2



Comparison



Data Movement



Sorted

Selection Sort

5 1 3 4 6 2



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

6

2



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

6

2



Comparison



Data Movement



Sorted

Selection Sort

5 1 3 4 6 2



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

6

2



Comparison



Data Movement



Sorted

Selection Sort

5 1 3 4 6 2

**⏏
Largest**



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

2

6



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

2

6



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort

5 **1** 3 4 2 **6**



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

2

6



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

2

6



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

2

6



Comparison



Data Movement



Sorted

Selection Sort

5

1

3

4

2

6

□

Largest



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6

▮

Largest



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison

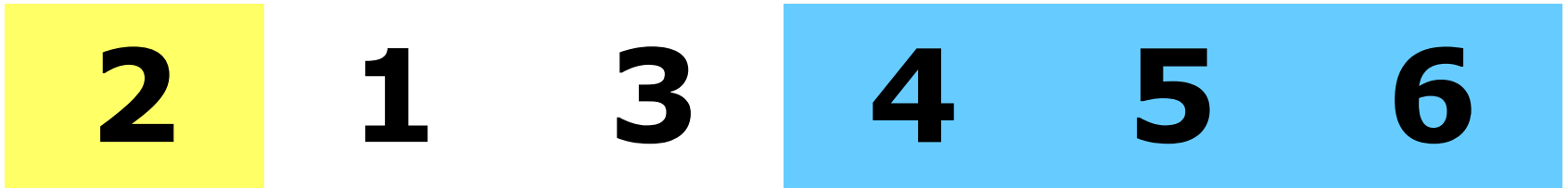


Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6

⏏
Largest



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort

2

1

3

4

5

6

□

Largest



Comparison



Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort

1 2 3 4 5 6

DONE!



Comparison



Data Movement



Sorted

Selection Sort: Analysis

- ▶ Running time:
 - Worst case: $O(N^2)$
 - Best case: $O(N^2)$

Bubble sort

- ▶ Adjacent elements are compared and exchanged.
- ▶ It is repeatedly done
- ▶ At the end the largest element is bubbled to the last position in the list

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

Bubblesort compares the numbers in pairs from left to right exchanging when necessary. Here the first number is compared to the second and as it is larger they are exchanged.

Now the next pair of numbers are compared. Again the 9 is the larger and so this pair is also exchanged.

In the third comparison, the 9 is not larger than the 12 so no exchange is made. We move on to compare the next pair without any change to the list.

The 12 is larger than the 11 so they are exchanged.

The twelve is greater than the 9 so they are exchanged

The end of the list has been reached so this is the end of the first pass. The twelve at the end of the list must be largest number in the list and so is now in the correct position. We now start a new pass from left to right.

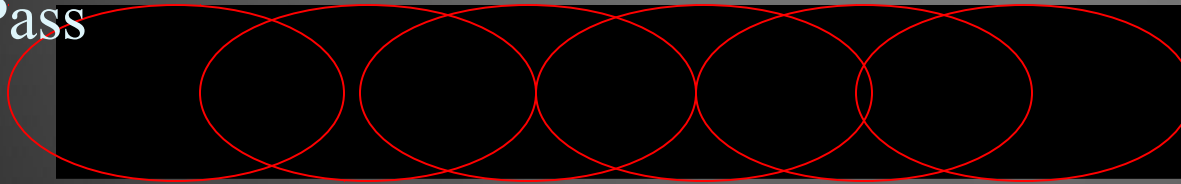
The 12 is greater than the 7 so they are exchanged.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass



Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This pass therefore only requires 6 comparisons.

Bubble Sort Example

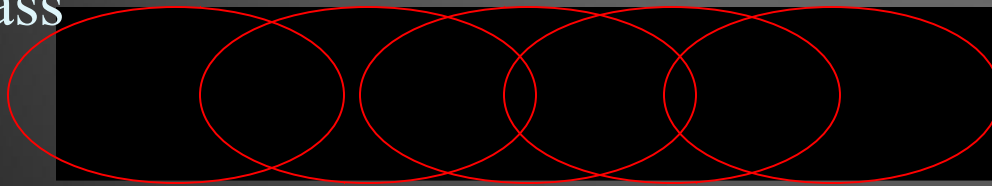
First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass



This time the 11 and 12 are in position. This pass therefore only requires 5 comparisons.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

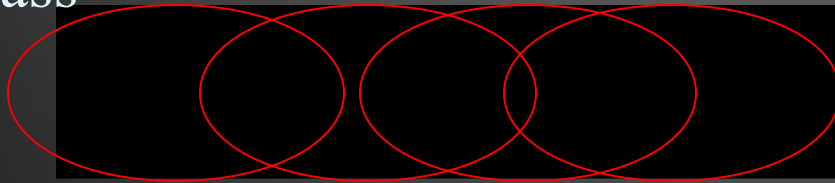
Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass

2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass



Each pass requires fewer comparisons. This time only 4 are needed.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

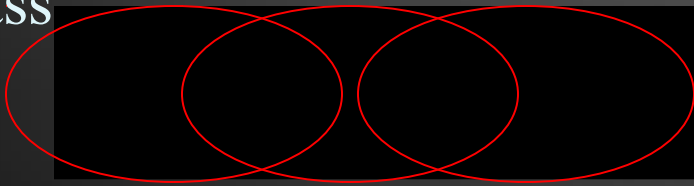
Third Pass

2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass

2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass



The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass

2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass

2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass

This pass no exchanges are made so the algorithm knows the list is sorted. It can therefore save time by not doing the final pass. With other lists this check could save much more work.

Sixth Pass

2, 3, 6, 7, 9, 9, 11, 12

Bubble Sort: Analysis

- ▶ Running time:
 - Worst case: $O(N^2)$
 - Best case: $O(N)$
- ▶ Variant:
 - bi-directional bubble sort
 - ▢ original bubble sort: only works to one direction
 - ▢ bi-directional bubble sort: works back and forth.

Radix Sort

- ▶ This sort is unusual because it does not directly compare any of the elements
- ▶ We instead create a set of buckets and repeatedly separate the elements into the buckets
- ▶ On each pass, we look at a different part of the elements

Radix Sort

- ▶ Assuming decimal elements and 10 buckets, we would put the elements into the bucket associated with its units digit
- ▶ **The buckets are actually queues** so the elements are added at the end of the bucket
- ▶ At the end of the pass, the buckets are combined in increasing order

Radix Sort

- ▶ On the **second pass**, we separate the elements **based on the “tens” digit**, and on **the third pass** we separate them **based on the “hundreds” digit**
- ▶ Each pass must make sure to process the elements in order and to put the buckets back together in the correct order

Radix Sort Analysis

- ▶ Each element is examined once for each of the digits it contains, so if the elements have **at most M digits** and there are **N elements** this algorithm has order **$O(M*N)$**
- ▶ This means that sorting is linear based on the number of elements
- ▶ Why then isn't this the only sorting algorithm used?

Radix Sort Example

Original list

310 213 023 130 013 301 222 032 201 111 323 002 330 102 231 120

Bucket Number

Contents

0

310 130 330 120

1

301 201 111 231

2

222 032 002 102

3

213 023 013 323

← The unit digit is 0

← The unit digit is 1

← The unit digit is 2

← The unit digit is 3

■ **FIGURE 4.4**

The three passes
of a radix sort

(a) Pass 1, Units Digit

Radix Sort Example (continued)

Pass 1 list

The unit digits are already in order

310 130 330 120 301 201 111 231 222 032 002 102 213 023 013 323

Bucket Number

Contents

Now start sorting the tens digit

0	<u>3</u> 01 <u>2</u> 01 <u>0</u> 02 <u>1</u> 02
1	310 111 213 013
2	120 222 023 323
3	130 330 231 032

■ FIGURE 4.4

The three passes
of a radix sort

(b) Pass 2, Tens Digit

Radix Sort Example (continued)

Pass 2 list

The unit and tens digits are already in order

301 201 002 102 310 111 213 013 120 222 023 323 130 330 231 032

Bucket Number

Contents

Now start sorting the hundreds digit

0	<u>002</u> <u>013</u> <u>023</u> <u>032</u>
1	102 111 120 130
2	201 213 222 231
3	301 310 323 330

■ FIGURE 4.4

The three passes
of a radix sort

(c) Pass 3, Hundreds Digit

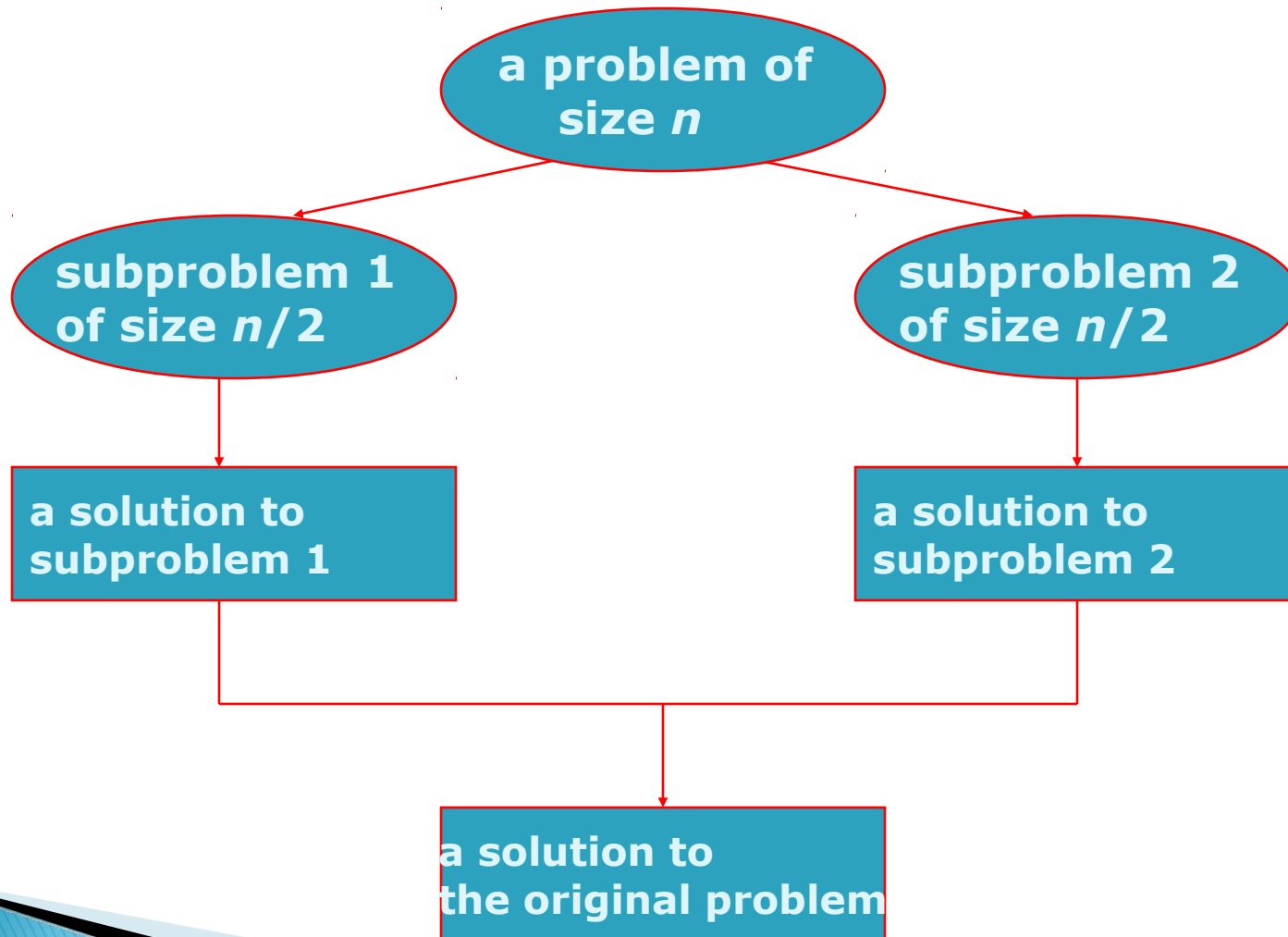
Values in the buckets are now in order

Divide and Conquer

The most well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide and conquer technique

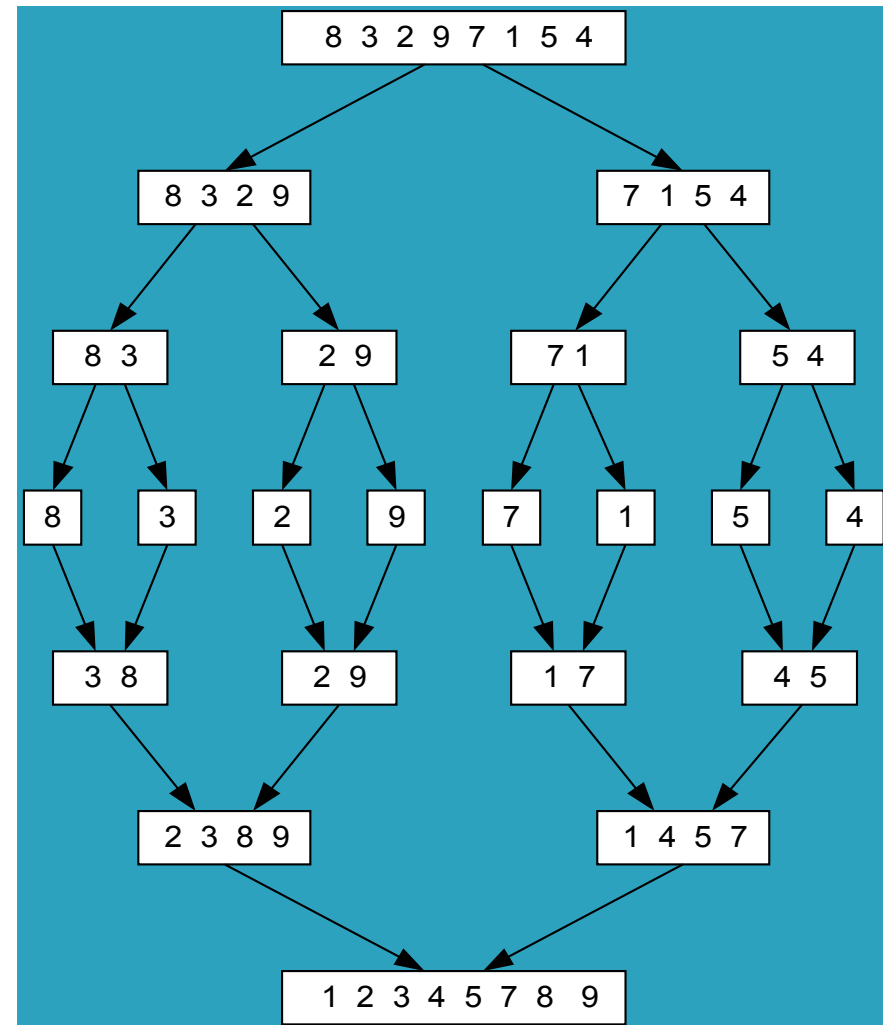


Divide and Conquer Examples

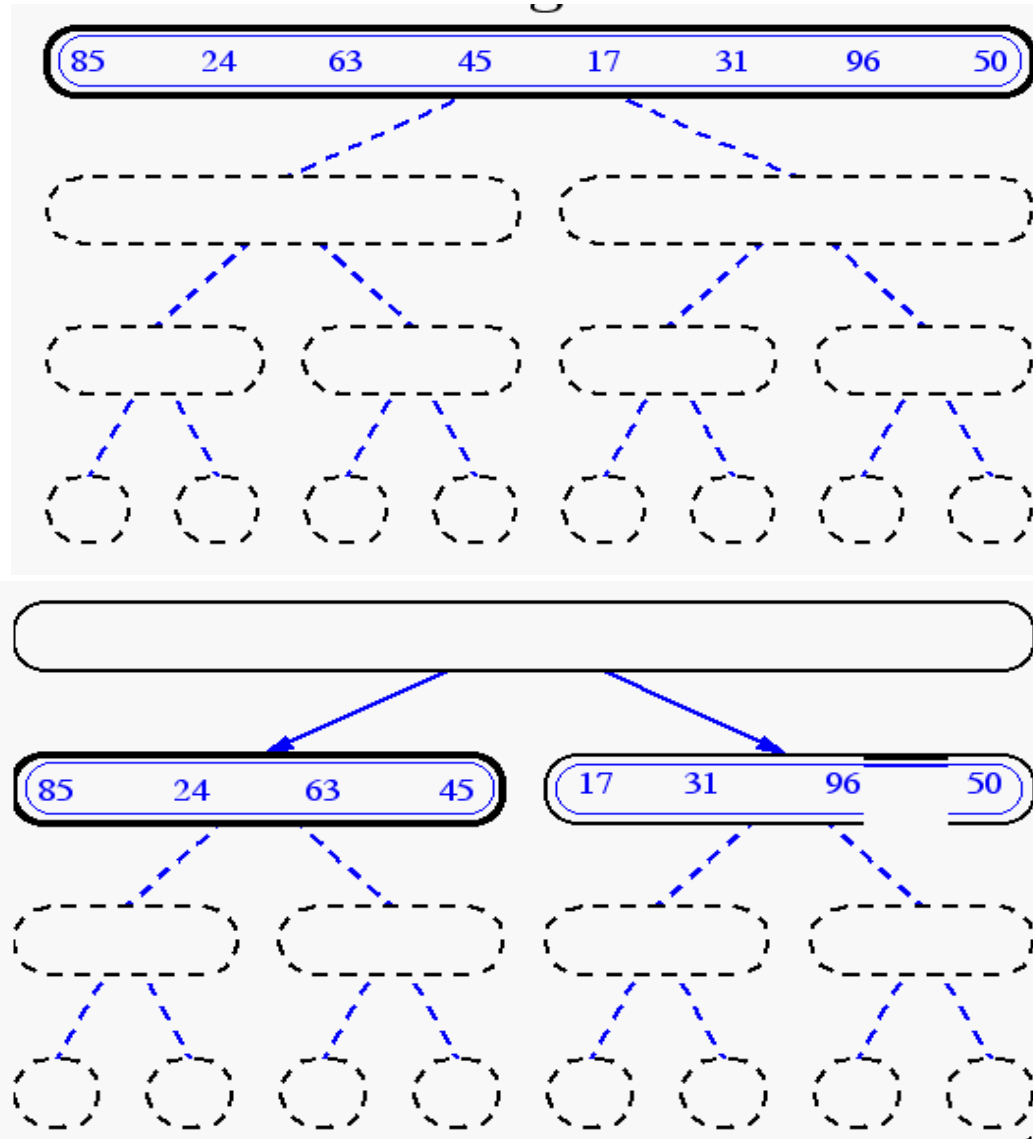
- ▶ Sorting: mergesort and quicksort
- ▶ Tree traversals
- ▶ Binary search
- ▶ Matrix multiplication-Strassen's algorithm
- ▶ Convex hull-QuickHull algorithm

Merge sort

- ❖ It stores by dividing array into two halves
- ❖ Sort each them and merge them into single sorted one



Merge-Sort Example



98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98

23

45

14

23	98
----	----

14	45
----	----

Merge

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

Merge

Merge

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45	67	98
---	----	----	----	----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----



6	14	23	33	42	45	67	98
---	----	----	----	----	----	----	----

Quicksort

- ▶ Quicksort is more widely used than any other sort.
- ▶ Quicksort is well-studied, not difficult to implement, works well on a variety of data, and consumes fewer resources than other sorts in nearly all situations.
- ▶ Quicksort is $O(n \log n)$ time, and $O(\log n)$ additional space due to recursion.

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements \geq pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements \geq pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

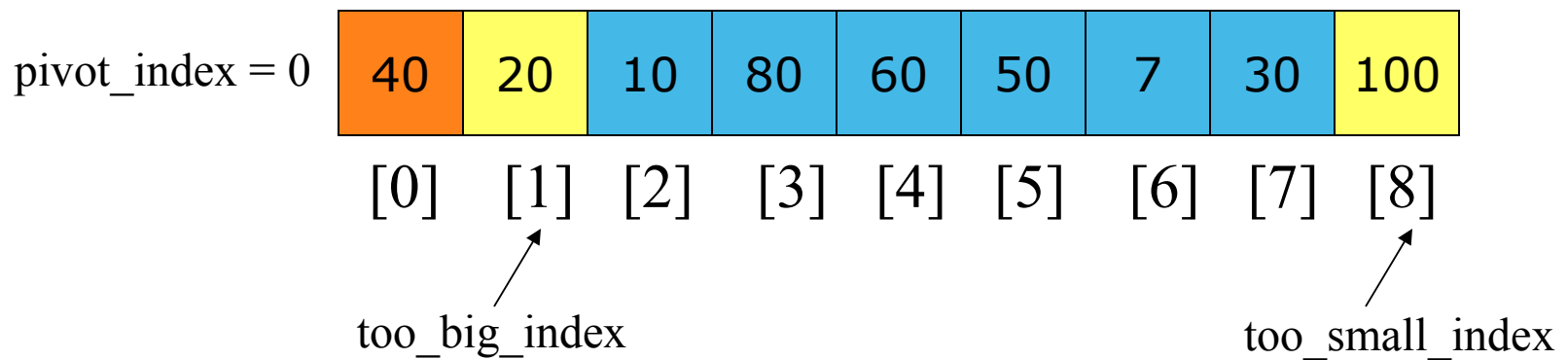
pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

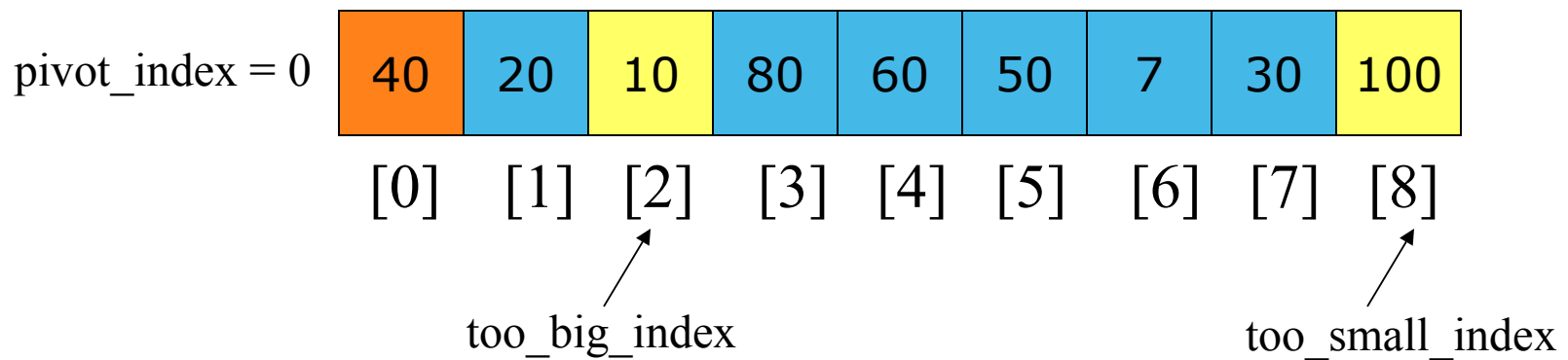
too_big_index

too_small_index

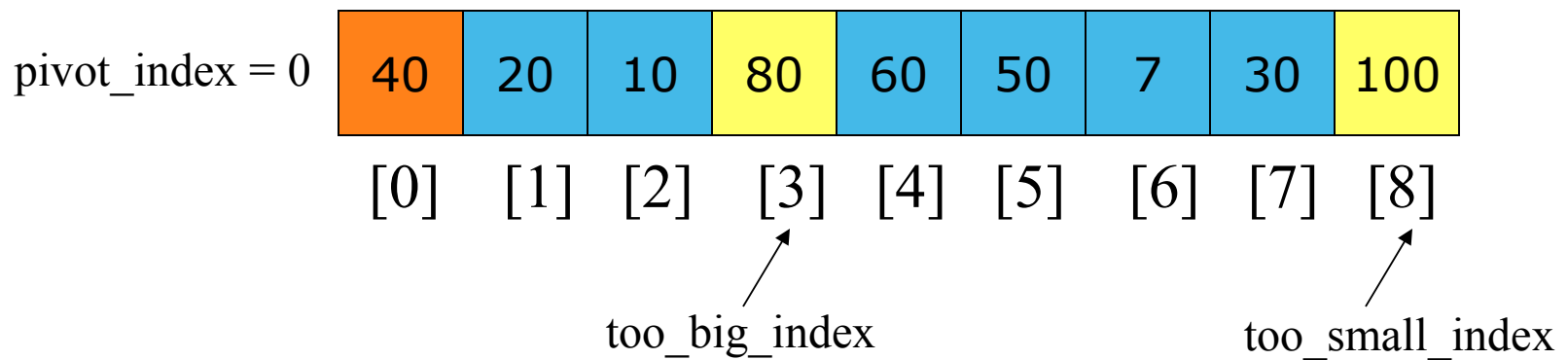
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$



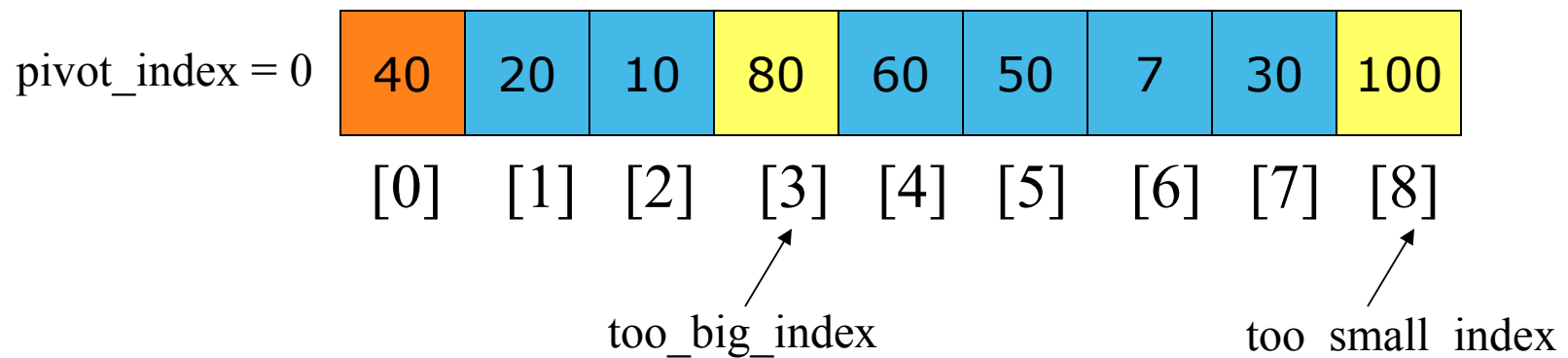
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$



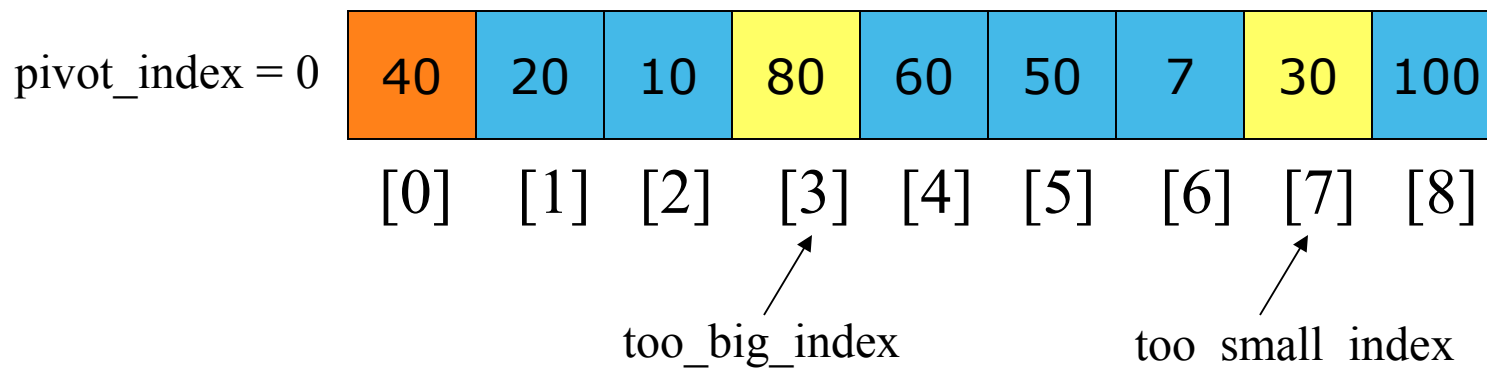
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$



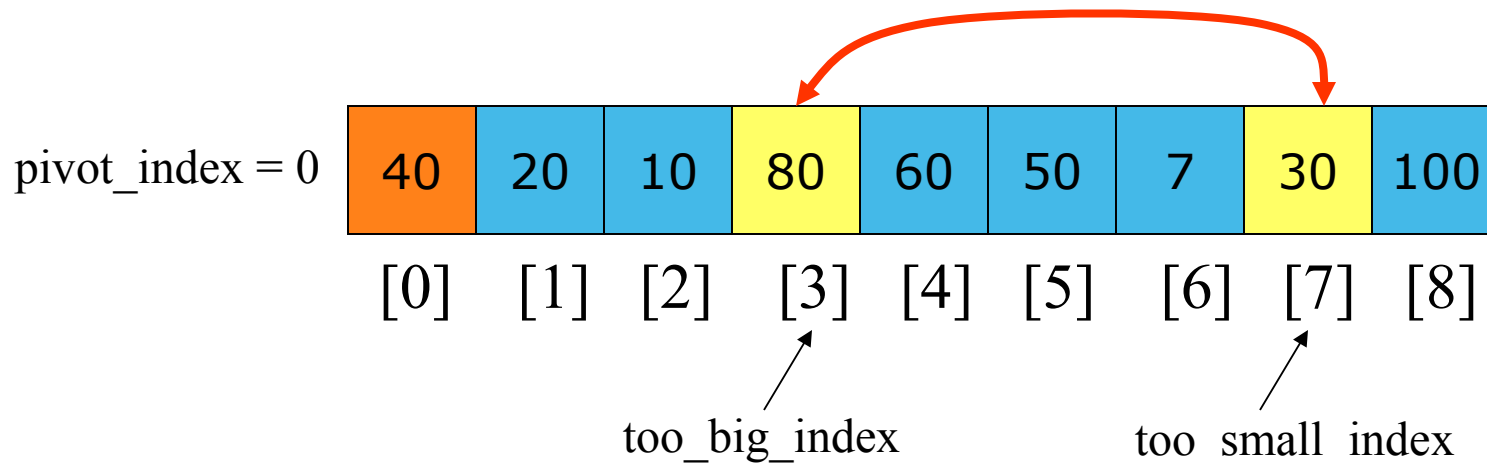
1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`



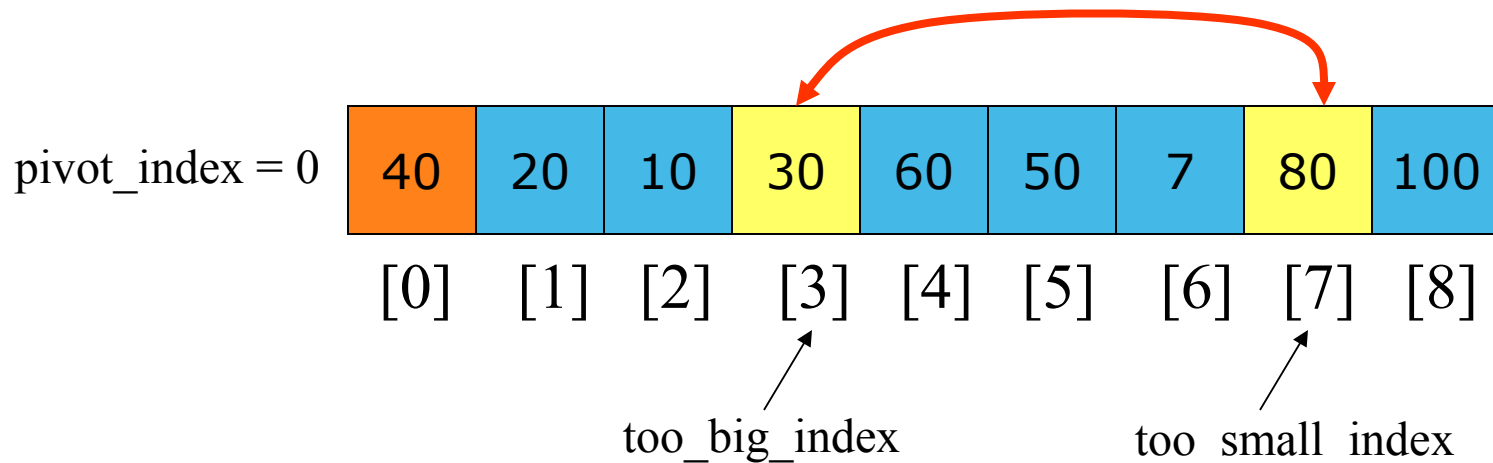
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$



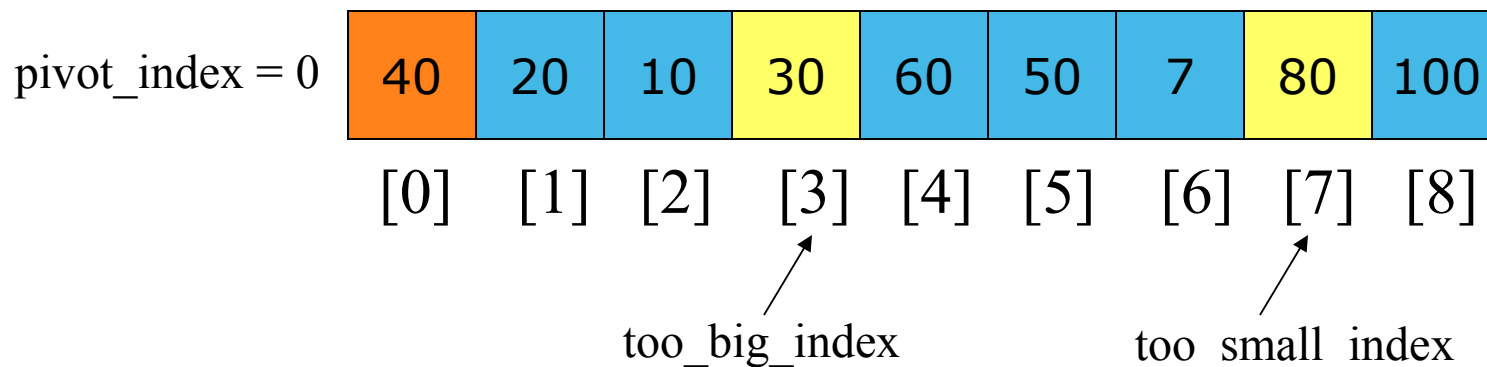
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



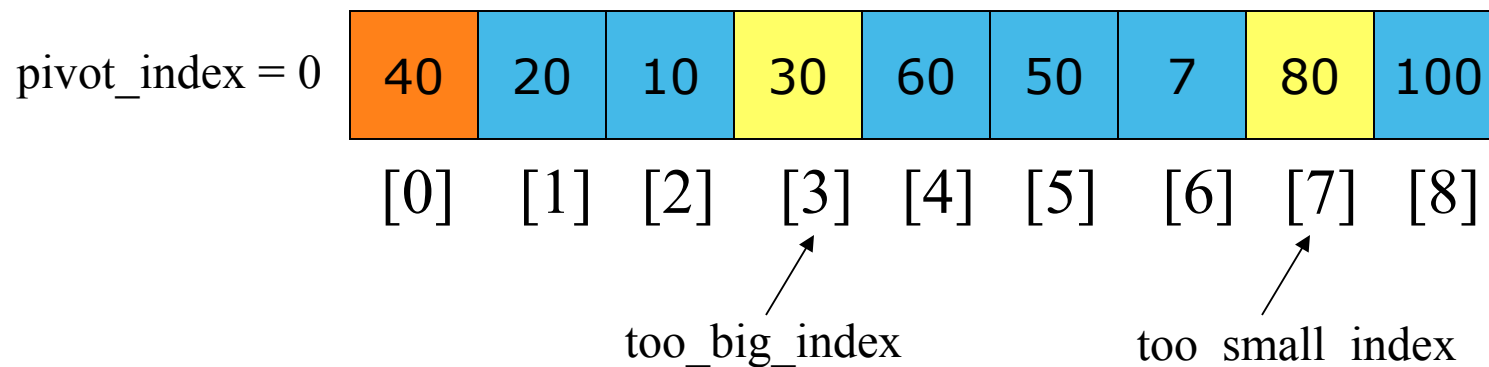
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$




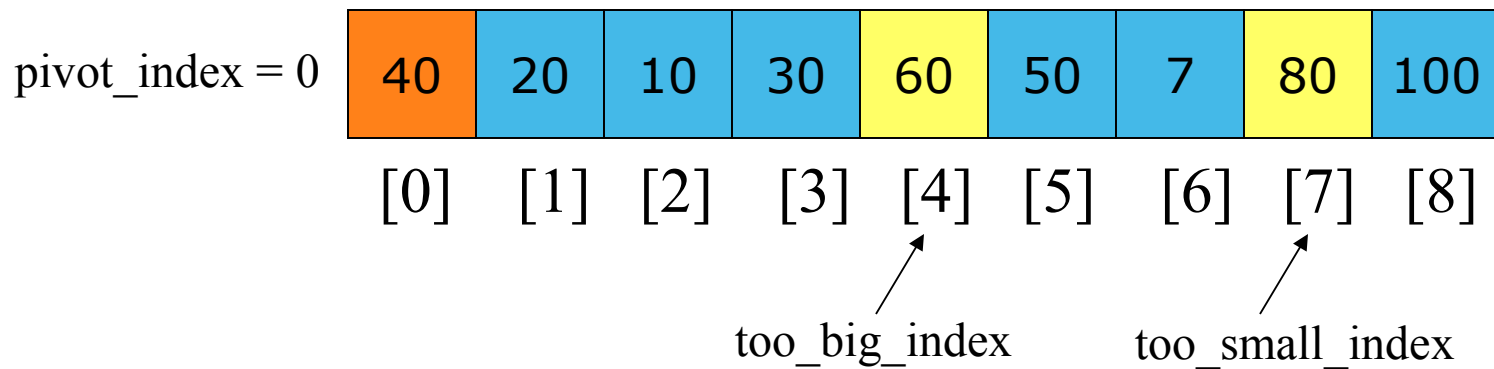
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




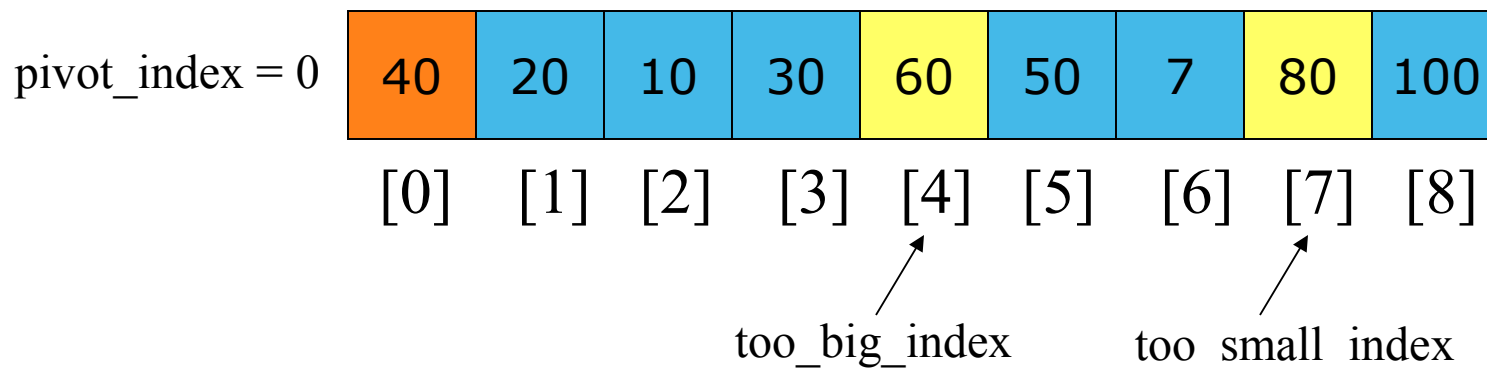
1. → While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




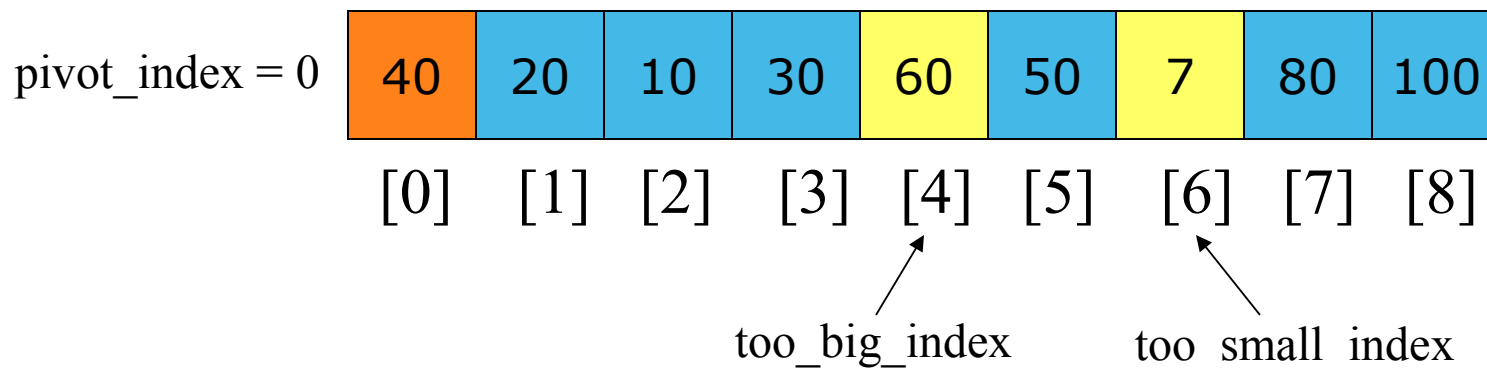
1.  While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
4. While `too_small_index > too_big_index`, go to 1.



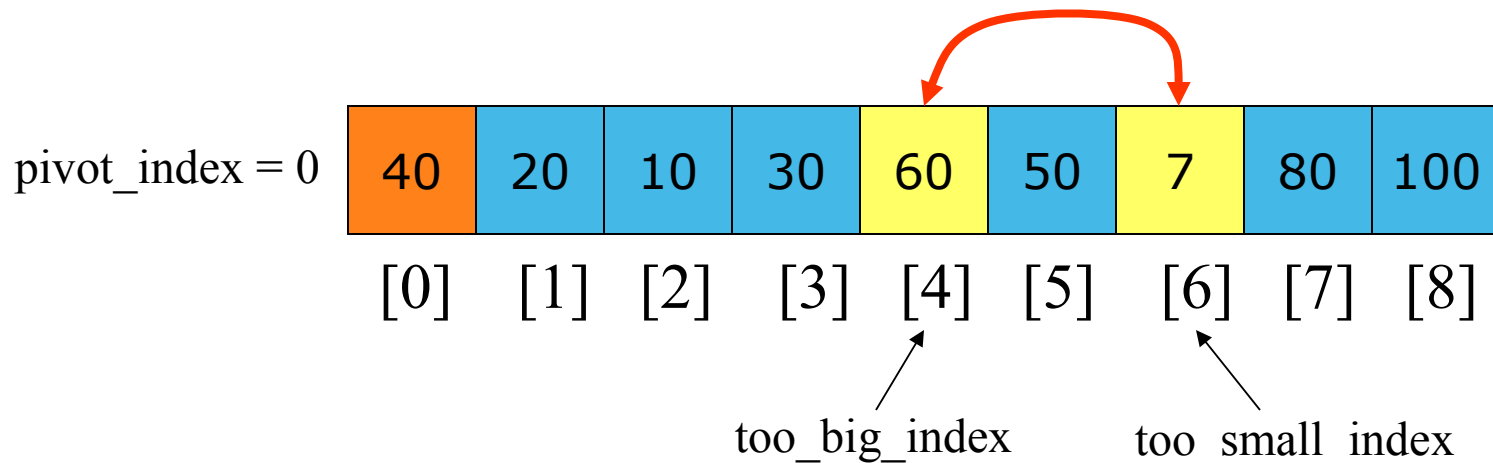
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2.  While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




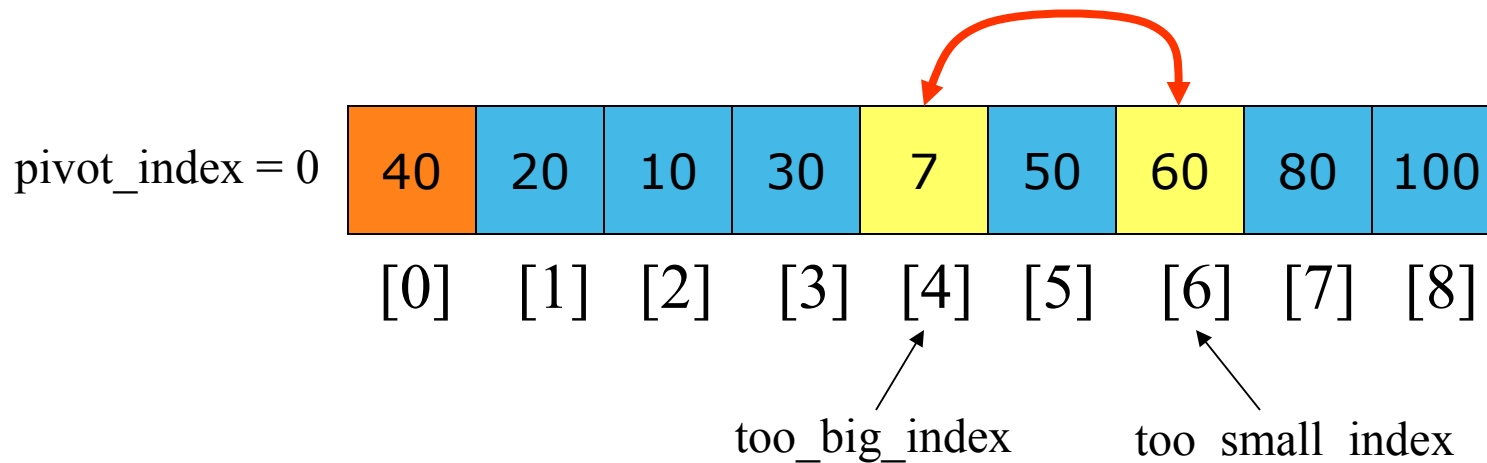
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2.  While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



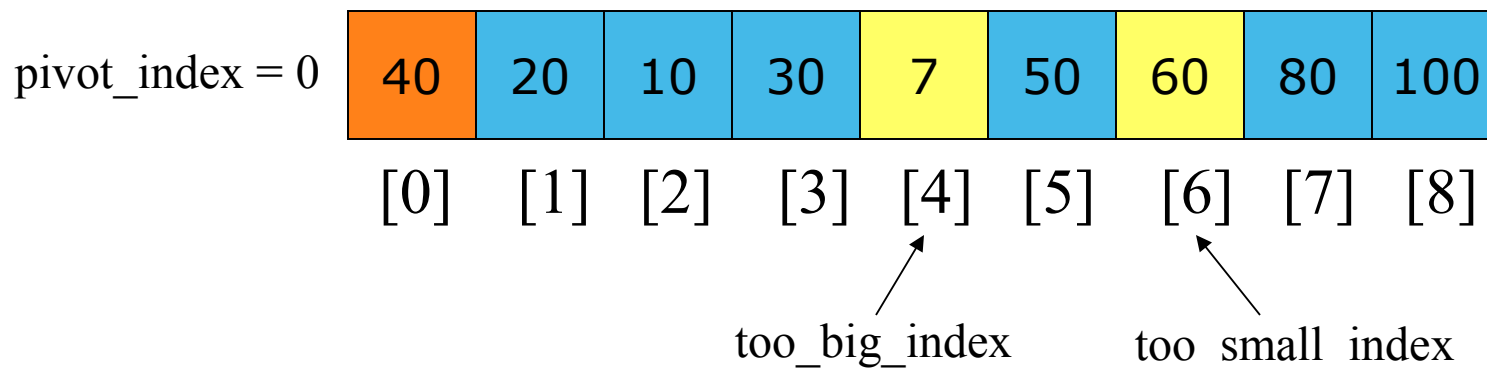
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. → If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



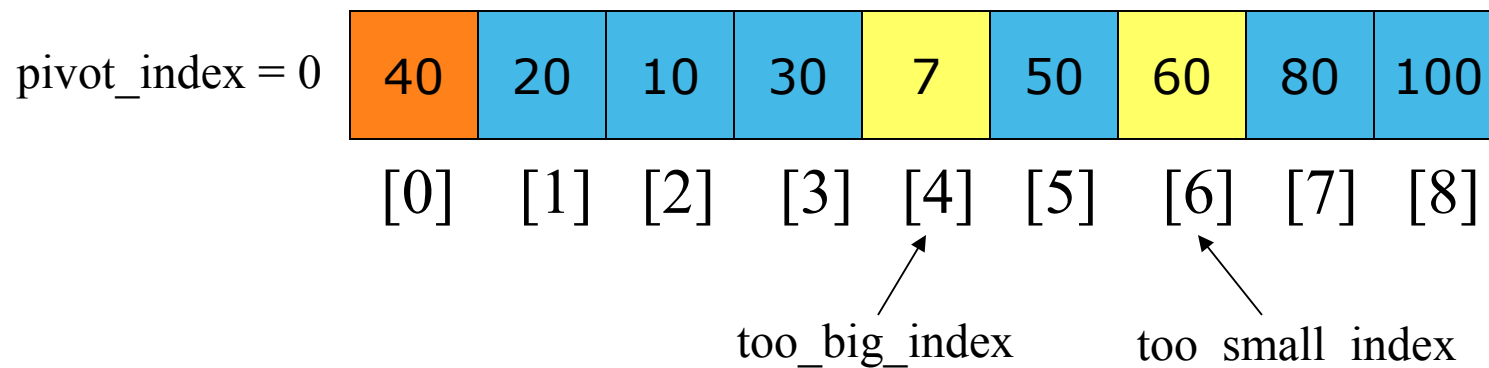
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3.  If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



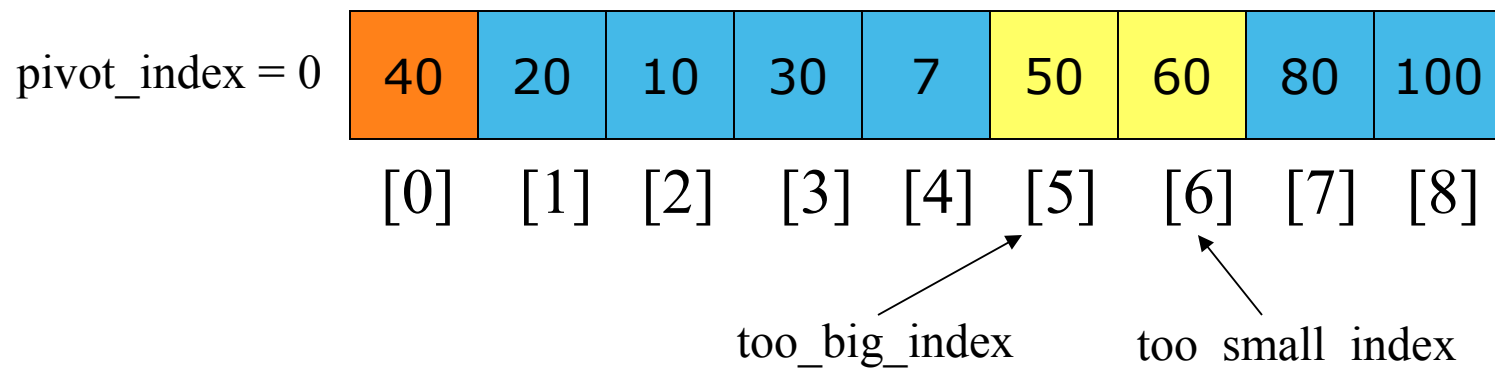
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. → While $\text{too_small_index} > \text{too_big_index}$, go to 1.




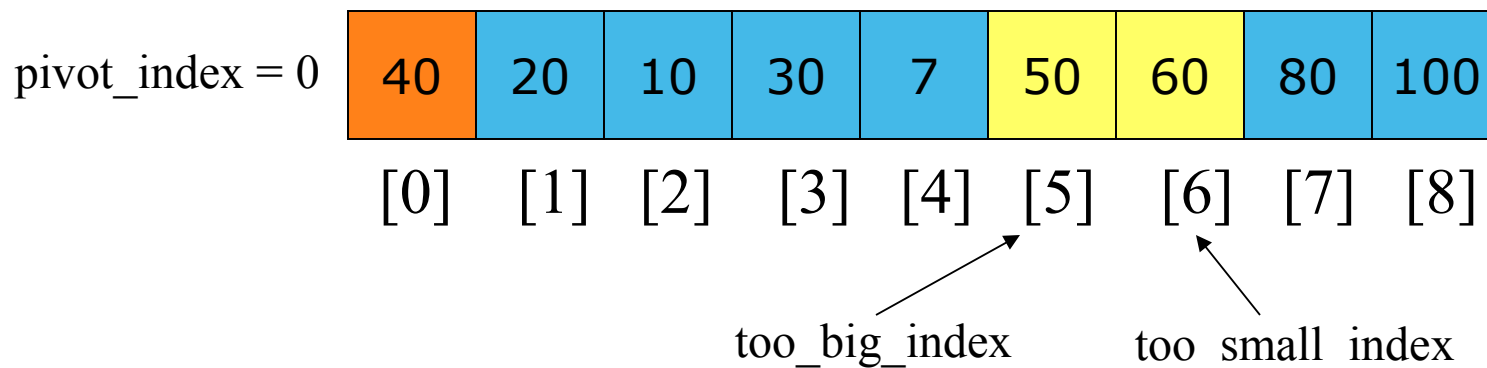
1. → While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




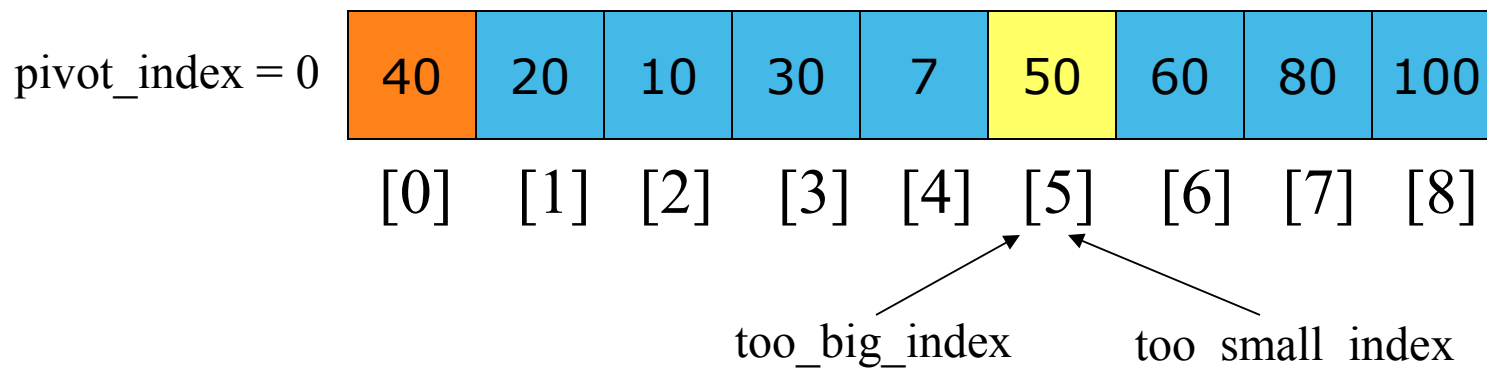
1. → While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




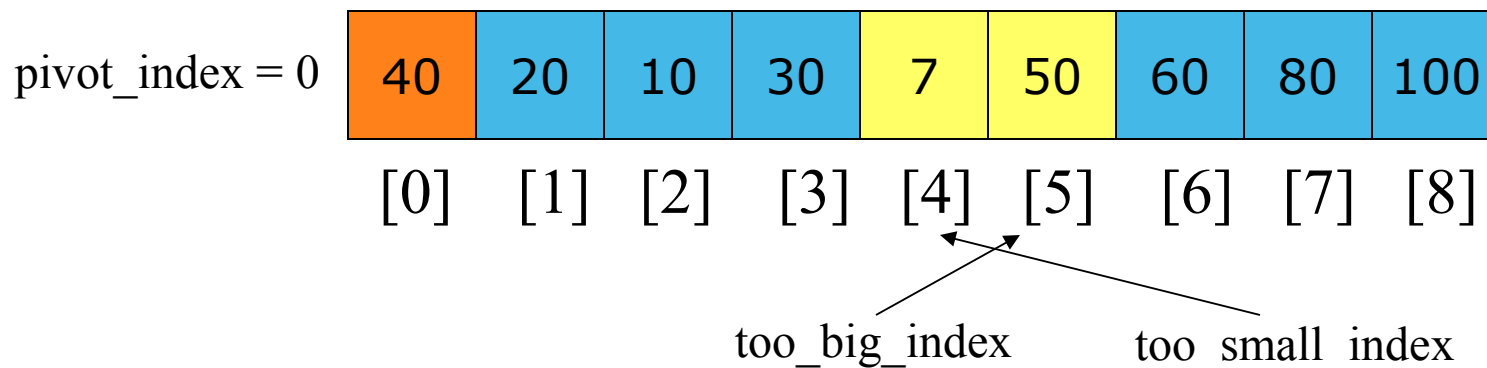
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2.  While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




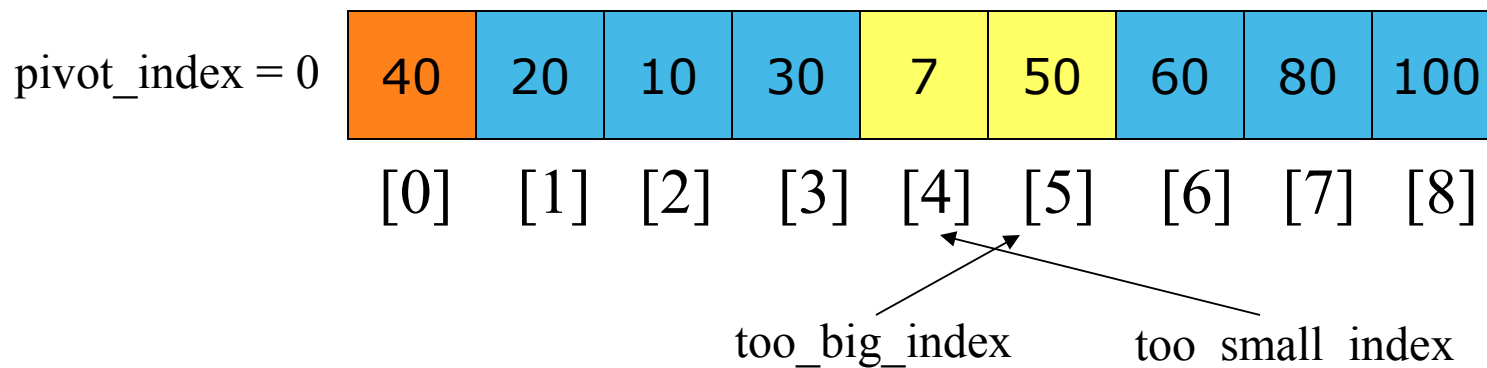
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2.  While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




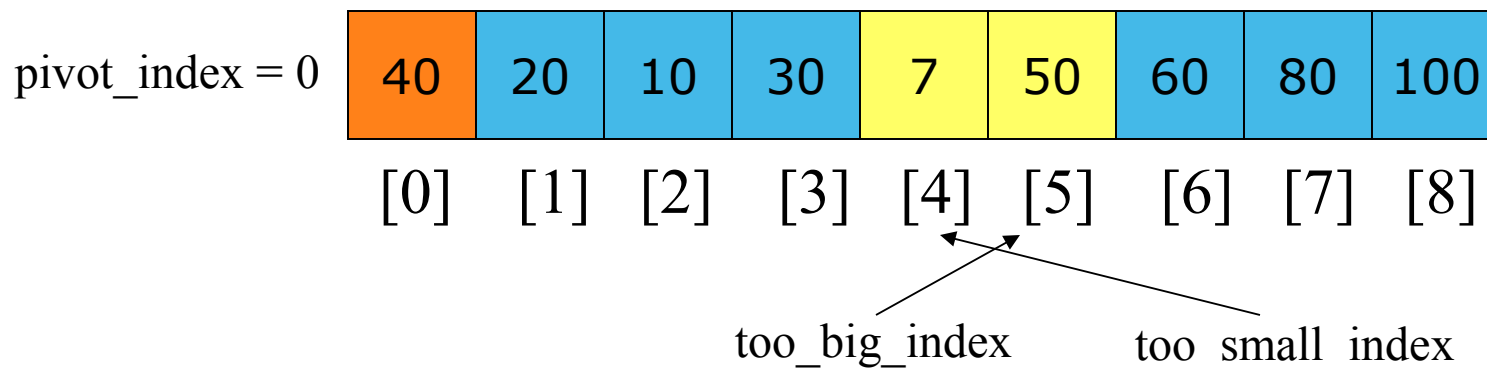
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2.  While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




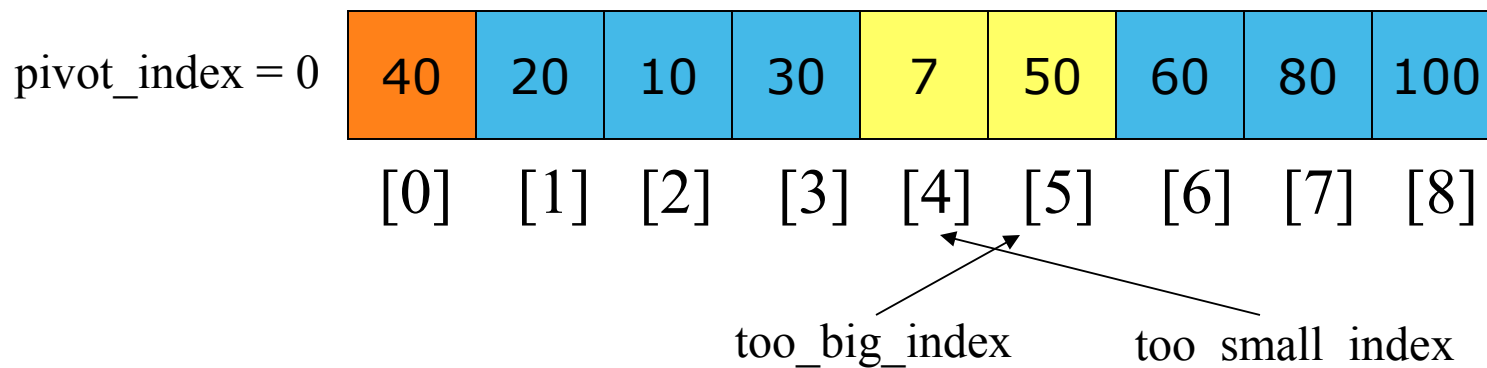
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3.  If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.




1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4.  While $\text{too_small_index} > \text{too_big_index}$, go to 1.

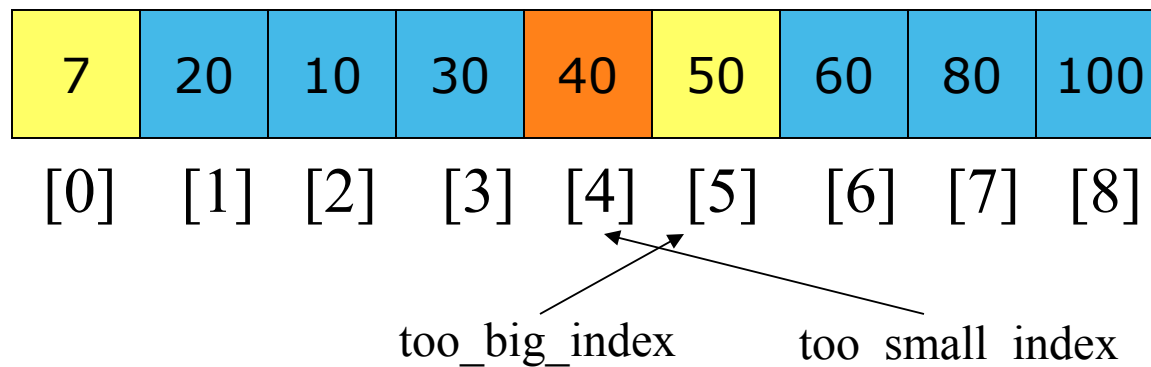


1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5.  Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5.  Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$

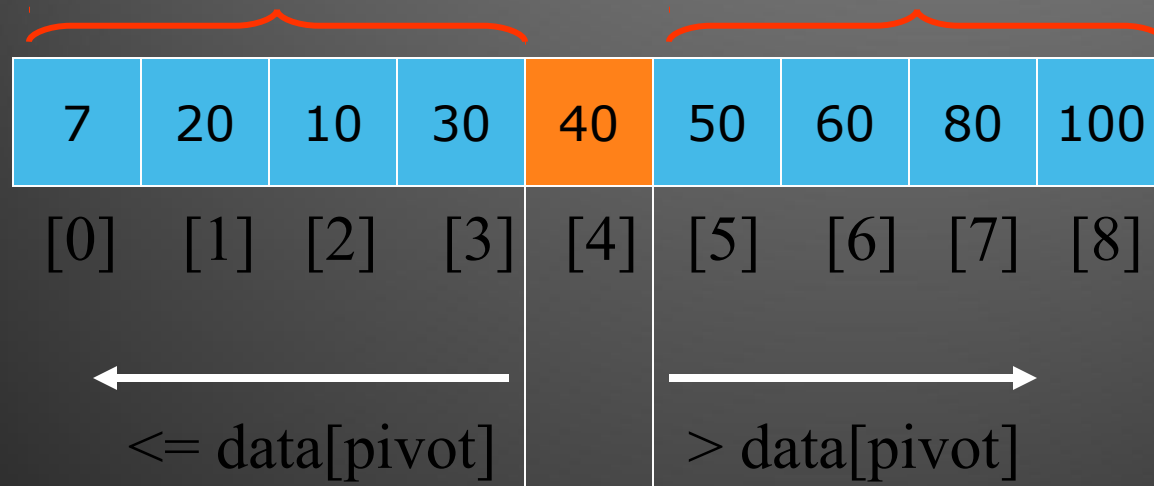
pivot_index = 4



Partition Result

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
←					→			
≤ data[pivot]					> data[pivot]			

Recursion: Quicksort Sub-arrays



Quicksort Demo

- ▶ **Quicksort** illustrates the operation of the basic algorithm. When the array is partitioned, one element is in place on the diagonal, the left subarray has its upper corner at that element, and the right subarray has its lower corner at that element. The original file is divided into two smaller parts that are sorted independently. The left subarray is always sorted first, so the sorted result emerges as a line of black dots moving right and up the diagonal.

conclusion

- ▶ In this ,we can understand that every sorting will depends upon one another.
- ▶ So, we should understand sorting is the best starting for every representation.
- ▶ From this, Quick sort will easily perform sorting.